

RVA22 Profile Release

Table of Contents

Copyright and license information.....1

Acknowledgements.....1

1. RISC-V Profiles.....1

2. RVA Profile Class.....3

3. RVA22 Profile Release.....4

Appendix A: Profile Comparisons.....22

Appendix B: Extension Details.....27

Appendix C: Instruction Details.....86

Appendix D: CSR Details.....1447

Appendix E: IDL Function Details.....1829

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 Class

The RVA profile class 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 class 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 class:

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 54 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 24 mandatory extensions.

- **A** Atomic instructions

Version = 2.1

- **C** Compressed instructions

Version = 2.0

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

Version = 2.2



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

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

Version = 2.2

- **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 instructions

Version = 2.0

- **U** User-mode privilege level

Version ~> 2.0

- **Za128rs** Reservation set requirement for RVA profiles

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.

- **Zba** Address generation instructions

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 as 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

- **Ziccamo** Main memory atomicity requirement for RVA profiles

Version = 1.0



Ziccamo is a profile-defined extension introduced with RVA20.

- **Ziccif** Main memory fetch requirement for RVA profiles

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 misaligned requirement for RVA profiles

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 reservability requirement for RVA profiles

Version = 1.0



Ziccrse is a profile-defined extension introduced with RVA20.

- **Zicntr** Architectural performance counters

Version = 2.0

- **Zihintpause** PAUSE instruction

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** Programmable 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 4 optional extensions.

Localized Options

- **V** Variable-length vector

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.

- **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.

Development Options

- **V** Variable-length vector

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.

- **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.

Expansion Options

- **V** Variable-length vector

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.

- **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.

Transitory Options

- **V** Variable-length vector

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.

- **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.



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 24 associated implementation-defined parameters.

LRSC_FAIL_ON_NON_EXACT_LRSC

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

LRSC_FAIL_ON_VA_SYNONYM

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

LRSC_MISALIGNED_BEHAVIOR

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

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

LRSC_RESERVATION_STRATEGY

Strategy used to handle reservation sets.

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

MISALIGNED_AMO

whether or not the implementation supports misaligned atomics in main memory

MUTABLE_MISA_A

When the [A](#) extensions is supported, indicates whether or not the extension can be disabled in the [misa.A](#) bit.

MUTABLE_MISA_C

Indicates whether or not the [C](#) extension can be disabled with the [misa.C](#) bit.

MUTABLE_MISA_D

Indicates whether or not the [D](#) extension can be disabled with the [misa.D](#) bit.

HW_MSTATUS_FS_DIRTY_UPDATE

Indicates whether or not hardware will write to [mstatus.FS](#)

Values are:

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

MSTATUS_FS_LEGAL_VALUES

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

MUTABLE_MISA_F

Indicates whether or not the F extension can be disabled with the [misa.F](#) bit.

MUTABLE_MISA_M

Indicates whether or not the M extension can be disabled with the [misa.M](#) bit.

MUTABLE_MISA_U

Indicates whether or not the U extension can be disabled with the [misa.U](#) bit.

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. Can be one of:

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

U_MODE_ENDIANNES

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](#)

HW_MSTATUS_VS_DIRTY_UPDATE

Indicates whether or not hardware will write to [mstatus.VS](#)

Values are:

never	Hardware never writes mstatus.VS
precise	Hardware writes mstatus.VS to the Dirty (3) state precisely when V registers are modified
imprecise	Hardware writes mstatus.VS imprecisely. This will result in a call to unpredictable() on any attempt to read mstatus or write vector state.

MSTATUS_VS_LEGAL_VALUES

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

MUTABLE_MISA_V

Indicates whether or not the V extension can be disabled with the [misa.V](#) bit.

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/timeeh](#) exists, and accessing it will not cause an IllegalInstruction trap

false
[time/timeeh](#) 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 35 mandatory extensions.

- **A** Atomic instructions

Version = 2.1

- **C** Compressed instructions

Version = 2.0

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

Version = 2.2



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

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

Version = 2.2

- **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 instructions

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** Supervisor counter enable

Version = 1.0



Sstvala is a new extension name introduced with RVA22.

- **Sstvala** [stval](#) requirements for RVA profiles

Version = 1.0



Sstvala is a new extension name introduced with RVA20.

- **Sstvecd** Direct exception vectoring

Version = 1.0

	<i>Sstvecd is a new extension name introduced with RVA20.</i>
• Sv39 39-bit virtual address translation (3 level)	
Version = 1.11	
• Svade Exception on PTE A/D Bits	
Version ~> 1.0	
	<i>Svbare is a new extension name introduced with RVA20.</i>
<i>It is subsequently defined in more detail with the ratification of Svadu.</i>	
• Svbare Bare virtual addressing	
Version = 1.0	
	<i>Svbare is a new extension name introduced with RVA20.</i>
• 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 requirement for RVA profiles	
Version = 1.0	
	<i>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.</i>
• Zba Address generation instructions	
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	
	<i>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.</i>
• Zic64b 64-byte cache blocks	
Version = 1.0	
	<i>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.</i>
• Zicbom Cache block management instructions	
Version ~> 1.0	
• Zicbop Cache block prefetch	
Version ~> 1.0	
	<i>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.</i>

- **Zicboz** Cache block zero instruction

Version ~> 1.0

- **Ziccamao** Main memory atomicity requirement for RVA profiles

Version = 1.0



Ziccamao is a profile-defined extension introduced with RVA20.

- **Ziccif** Main memory fetch requirement for RVA profiles

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 misaligned requirement for RVA profiles

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 reservability requirement for RVA profiles

Version = 1.0



Ziccrse is a profile-defined extension introduced with RVA20.

- **Zicntr** Architectural performance counters

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 instruction

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** Programmable 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 12 optional extensions.

Localized Options

- **Sha** The augmented hypervisor extension

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** Variable-length vector

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.

• **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

Development Options

• **Sha** The augmented hypervisor extension

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** Variable-length vector

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.

- **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

Expansion Options

- **Sha** The augmented hypervisor extension

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

	<i>Ssu64xl is a new extension name introduced with RVA20.</i>
• 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	
	<i>It is expected that Svnapot will be mandatory in the next profile release.</i>
• V Variable-length vector	
Version ~> 1.0	
	<i>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.</i>
<i>A future profile might mandate V.</i>	
• Zfh Half-precision floating point	
Version ~> 1.0	
	<i>A future profile might mandate V.</i>
• Zkn NIST Algorithm Suite	
Version ~> 1.0	
• Zkr Entropy Source	
Version ~> 1.0	
	<i>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.</i>
• Zks ShangMi Algorithm Suite	
Version ~> 1.0	
Transitory Options	
• Sha The augmented hypervisor extension	
Version ~> 1.0	
	<i>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).</i>
• Sscofpmf Counter Overflow and Privilege Mode Filtering	
Version ~> 1.0	
	<i>Platforms may choose to mandate the presence of Sscofpmf.</i>
• Sstc Supervisor-mode timer interrupts	
Version ~> 1.0	
	<i>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.</i>
• Ssu64xl 64-bit UXLEN	
Version = 1.0	
	<i>Ssu64xl is a new extension name introduced with RVA20.</i>
• 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** Variable-length vector

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.

- **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 54 associated implementation-defined parameters.

LRSC_FAIL_ON_NON_EXACT_LRSC

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

LRSC_FAIL_ON_VA_SYNONYM

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

LRSC_MISALIGNED_BEHAVIOR

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

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

LRSC_RESERVATION_STRATEGY

Strategy used to handle reservation sets.

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

MISALIGNED_AMO

whether or not the implementation supports misaligned atomics in main memory

MUTABLE_MISA_A

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

MUTABLE_MISA_C

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

MUTABLE_MISA_D

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

HW_MSTATUS_FS_DIRTY_UPDATE

Indicates whether or not hardware will write to `mstatus.FS`

Values are:

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

MSTATUS_FS_LEGAL_VALUES

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

MUTABLE_MISA_F

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

MUTABLE_MISA_M

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

ASID_WIDTH

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

MSTATUS_FS_LEGAL_VALUES

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

MSTATUS_FS_WRITABLE

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

MSTATUS_TVM_IMPLEMENTED

Whether or not `mstatus.TVM` is implemented.

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

MSTATUS_VS_LEGAL_VALUES

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

MSTATUS_VS_WRITABLE

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

MUTABLE_MISA_S

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

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.

SCOUNTENABLE_EN

Indicates which counters can delegated via [scounteren](#)

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

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

STVAL_WIDTH

The number of implemented bits in [stval](#).

Must be greater than or equal to $\max(\text{PHYS\_ADDR\_WIDTH}, \text{VA\_SIZE})$

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

S_MODE_ENDIANNES

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.

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY

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

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY indicates whether or not that exception occurs.

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

MUTABLE_MISA_U

Indicates whether or not the [U](#) extension can be disabled with the [misa.U](#) bit.

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. Can be one of:

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

U_MODE_ENDIANNES

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](#)

HW_MSTATUS_VS_DIRTY_UPDATE

Indicates whether or not hardware will write to [mstatus.VS](#)

Values are:

never	Hardware never writes mstatus.VS
precise	Hardware writes mstatus.VS to the Dirty (3) state precisely when V registers are modified
imprecise	Hardware writes mstatus.VS imprecisely. This will result in a call to unpredictable() on any attempt to read mstatus or write vector state.

MSTATUS_VS_LEGAL_VALUES

The set of values that [mstatus.VS](#) will accept from a software write.

MUTABLE_MISA_V

Indicates whether or not the [V](#) extension can be disabled with the [misa.V](#) bit.

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.

DRAFT

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 84 extensions.

Table 1. Extension Presence

Extension	RVA20	RVA22	RVA23	RVB23
A	mandatory	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
Svpptc	-	-	optional	optional
U	mandatory	mandatory	mandatory	mandatory
V	-	optional	mandatory	optional
Za128rs	mandatory	mandatory	mandatory	mandatory
Za64rs	-	-	mandatory	mandatory
Zabha	-	-	optional	optional
Zacas	-	-	optional	optional
Zama16b	-	-	optional	optional
Zawrs	-	-	mandatory	mandatory
Zba	-	mandatory	mandatory	mandatory
Zbb	-	mandatory	mandatory	mandatory
Zbc	-	-	optional	optional
Zbs	-	mandatory	mandatory	mandatory
Zcb	-	-	mandatory	mandatory
Zcmop	-	-	mandatory	mandatory

Extension	RVA20	RVA22	RVA23	RVB23
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
Zifencei	mandatory	mandatory	mandatory	mandatory
Zihintntl	-	-	mandatory	mandatory
Zihintpause	-	mandatory	mandatory	mandatory
Zihpm	optional	mandatory	mandatory	mandatory
Zimop	-	-	mandatory	mandatory
Zkn	-	optional	optional	optional
Zkr	-	optional	optional	optional
Zks	-	optional	optional	optional
Zkt	-	mandatory	mandatory	mandatory
Zvbb	-	-	mandatory	optional
Zvbc	-	-	optional	optional
Zvbfmin	-	-	optional	optional
Zvbfwma	-	-	optional	optional
Zvfh	-	-	optional	optional
Zvfhmin	-	-	mandatory	optional
Zvkg	-	-	optional	optional
Zvknc	-	-	optional	optional
Zvkng	-	-	optional	optional
Zvksc	-	-	optional	optional
Zvksg	-	-	optional	optional
Zvkt	-	-	mandatory	optional

A.2. RVA Profile Releases

The RVA Profile Class has 3 releases that reference a total of 84 extensions.

Table 2. Extension Presence

Extension	RVA20	RVA22	RVA23
A	mandatory	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

Extension	RVA20	RVA22	RVA23
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
Svpptc	-	-	optional
U	mandatory	mandatory	mandatory
V	-	optional	mandatory
Za128rs	mandatory	mandatory	mandatory
Za64rs	-	-	mandatory
Zabha	-	-	optional
Zacas	-	-	optional
Zama16b	-	-	optional
Zawrs	-	-	mandatory
Zba	-	mandatory	mandatory
Zbb	-	mandatory	mandatory
Zbc	-	-	optional
Zbs	-	mandatory	mandatory
Zcb	-	-	mandatory
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

Extension	RVA20	RVA22	RVA23
Zicfiss	-	-	optional
Zicntr	mandatory	mandatory	mandatory
Zicond	-	-	mandatory
Zifencei	mandatory	mandatory	mandatory
Zihintntl	-	-	mandatory
Zihintpause	-	mandatory	mandatory
Zihpm	optional	mandatory	mandatory
Zimop	-	-	mandatory
Zkn	-	optional	optional
Zkr	-	optional	optional
Zks	-	optional	optional
Zkt	-	mandatory	mandatory
Zvbb	-	-	mandatory
Zvbc	-	-	optional
Zvbfmin	-	-	optional
Zvbfwma	-	-	optional
Zvfh	-	-	optional
Zvfhmin	-	-	mandatory
Zvkg	-	-	optional
Zvknc	-	-	optional
Zvkng	-	-	optional
Zvksc	-	-	optional
Zvksg	-	-	optional
Zvkt	-	-	mandatory

A.3. RVA22 Profiles

The RVA22 Profile Release has 2 profiles that reference a total of 47 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
C	mandatory	mandatory
D	mandatory	mandatory
F	mandatory	mandatory
I	mandatory	mandatory
M	mandatory	mandatory
S	-	mandatory
Sha	-	optional
Sscpctr	-	mandatory
Sscofpmf	-	optional
Sscounterenw	-	mandatory
Sstc	-	optional
Sstvala	-	mandatory
Sstvecd	-	mandatory
Ssu64xl	-	optional
Sv39	-	mandatory
Sv48	-	optional
Sv57	-	optional
Svade	-	mandatory
Svbare	-	mandatory
Svinval	-	mandatory

Extension	RVA22U64	RVA22S64
Svnapot	-	optional
Svpbmt	-	mandatory
U	mandatory	mandatory
V	optional	optional
Za128rs	mandatory	mandatory
Zba	mandatory	mandatory
Zbb	mandatory	mandatory
Zbs	mandatory	mandatory
Zfh	optional	optional
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
Zifencei	-	mandatory
Zihintpause	mandatory	mandatory
Zihpm	mandatory	mandatory
Zkn	optional	optional
Zkr	-	optional
Zks	optional	optional
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

2.1.0

State

ratified

Ratification date

2019-12

Implies

- [Zaamo](#) version 1.0.0
- [Zalrsc](#) version 1.0.0

B.1.1. Synopsis

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



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

The [A](#) extension comprises instructions provided by the [Zaamo](#) and [Zalrsc](#) extensions.

B.1.2. Specifying Ordering of Atomic Instructions

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

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

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

B.1.3. Instructions

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

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

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

B.1.4. Parameters

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

LRSC_FAIL_ON_NON_EXACT_LRSC

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

LRSC_FAIL_ON_VA_SYNONYM

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

LRSC_MISALIGNED_BEHAVIOR

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

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

LRSC_RESERVATION_STRATEGY

Strategy used to handle reservation sets.

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

MISALIGNED_AMO

whether or not the implementation supports misaligned atomics in main memory

MUTABLE_MISA_A

When the [A](#) extensions is supported, indicates whether or not the extension can be disabled in the [misa.A](#) bit.

B.2. Extension C

Long Name: Compressed instructions

Version Requirement: = 2.0

C Extension Presence

profile	v2.0.0
RVA22U64	mandatory
RVA22S64	mandatory

2.0.0

State

ratified

Ratification date

2019-12

Implies

- [Zca](#) version 1.0.0

- [Zcf](#) version 1.0.0
- [Zcd](#) version 1.0.0

B.2.1. Synopsis

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

B.2.2. Overview

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

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

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



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

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



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

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

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

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

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

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



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

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



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

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

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

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

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

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

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

B.2.3. Compressed Instruction Formats

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



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

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



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

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



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

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

Table 4. Compressed 16-bit RVC instruction formats

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

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

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

B.2.4. Instructions

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

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

B.2.5. Parameters

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

MUTABLE_MISA_C

Indicates whether or not the C extension can be disabled with the [misa.C](#) bit.

B.3. Extension D

Long Name: Double-precision floating-point

Version Requirement: = 2.2

D Extension Presence

profile	v2.2.0
RVA22U64	mandatory
RVA22S64	mandatory

2.2.0

State

ratified

Ratification date

2019-12

Changes

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

Implies

- F version 2.2.0

B.3.1. Synopsis

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

B.3.2. D Register State

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



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

B.3.3. NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower n -bit types, $n < \text{FLEN}$, 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, $n < m \leq \text{FLEN}$. 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-boxed value.



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, $n < \text{FLEN}$, 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, $n < \text{FLEN}$, check if the input operands are correctly NaN-boxed, i.e., all upper FLEN- n bits are 1. If so, the n least-significant bits of the input are used as the input value, otherwise the input value is treated as an n -bit canonical NaN.



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

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

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



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

B.3.4. Instructions

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

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

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

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

B.3.5. CSRs

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

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

B.3.6. Parameters

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

MUTABLE_MISA_D

Indicates whether or not the [D](#) extension can be disabled with the [misa.D](#) bit.

B.4. Extension F

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

profile	v2.2.0
RVA22U64	mandatory
RVA22S64	mandatory

2.2.0

State

ratified

Ratification date

2019-12

Changes

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

B.4.1. Synopsis

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

B.4.2. F Register State

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



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

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

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

Floating-Point Control and Status Register

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

Floating-point control and status register

Unresolved directive in 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 $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		Reserved for future use.
110		Reserved for future use.
111	DYN	In instruction's <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, reserved.

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



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

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

Table 8. Accrued exception flag encoding.

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



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

B.4.3. NaN Generation and Propagation

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



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

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



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

B.4.4. Subnormal Arithmetic

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

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



Detecting tininess after rounding results in fewer spurious underflow signals.

B.4.5. Instructions

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

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

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

B.4.6. CSRs

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

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

B.4.7. Parameters

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

HW_MSTATUS_FS_DIRTY_UPDATE

Indicates whether or not hardware will write to [mstatus.FS](#)

Values are:

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

MSTATUS_FS_LEGAL_VALUES

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

MUTABLE_MISA_F

Indicates whether or not the F extension can be disabled with the [misa.F](#) bit.

B.5. Extension I

Long Name: Base integer ISA (RV32I or RV64I)

Version Requirement: ~> 2.1

I Extension Presence

profile	v2.1.0
RVA22U64	mandatory
RVA22S64	mandatory

2.1.0

State

ratified

Ratification date

2019-06

Changes

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

B.5.1. Synopsis

Base integer instructions — TODO

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

As per the unprivileged architecture specification, the [ecall](#) instruction causes a requested trap to the execution environment.

Misaligned loads and stores might not be supported.

The [fence.tso](#) instruction is mandatory.

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

B.5.2. Instructions

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

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

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

B.6. Extension M

Long Name: Integer multiply and divide instructions

Version Requirement: = 2.0

M Extension Presence

profile	v2.0.0
RVA22U64	mandatory
RVA22S64	mandatory

2.0.0

State

ratified

Ratification date

2019-12

B.6.1. Synopsis

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



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

B.6.2. Instructions

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

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

B.6.3. Parameters

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

MUTABLE_MISA_M

Indicates whether or not the [M](#) extension can be disabled with the [misa.M](#) bit.

B.7. Extension 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	-

1.11.0

State

ratified

Ratification date

2019-06

1.12.0

State

ratified

Ratification date

2021-12

1.13.0

State

ratified

B.7.1. Synopsis

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



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

B.7.2. Instructions

The following 2 instructions are added by extension version 1.12.0 (the minimum version of this extension that satisfies the extension requirement).

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

B.7.3. CSRs

The following 12 CSRs are added by extension version 1.12.0 (the minimum version of this extension that satisfies the extension requirement).

Name	Long Name	Address	Mode
medeleg	Machine Exception Delegation	0x302	M
mideleg	Machine Interrupt Delegation	0x303	M
satp	Supervisor Address Translation and Protection	0x180	S
scause	Supervisor Cause	0x142	S
scounteren	Supervisor Counter Enable	0x106	S
senvcfg	Supervisor Environment Configuration	0x10a	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.7.4. Parameters

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

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

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

MSTATUS_FS_WRITABLE
When [S](#) is enabled but [F](#) is not, mstatus.FS is optionally writable.

MSTATUS_TVM_IMPLEMENTED
Whether or not mstatus.TVM is implemented.

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

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

MSTATUS_VS_WRITABLE
When [S](#) is enabled but [V](#) is not, mstatus.VS is optionally writable.

MUTABLE_MISA_S
Indicates whether or not the [S](#) extension can be disabled with the [misa.S](#) bit.

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.

SCOUNTENABLE_EN

Indicates which counters can delegated via [scounteren](#)

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

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

STVAL_WIDTH

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

S_MODE_ENDIANNES

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_EC

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

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

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY

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

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY indicates whether or not that exception occurs.

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

B.8. Extension Sha

Long Name: The augmented hypervisor extension

Version Requirement: ~> 1.0

Sha Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

1.0.0

State

ratified

Implies

- H version 1.0.0
- Sstateen version 1.0.0
- Shcounterenw version 1.0.0
- Shvstvala version 1.0.0
- Shtvala version 1.0.0
- Shvstvecd version 1.0.0
- Shvsatpa version 1.0.0
- Shgatpa version 1.0.0

B.8.1. Synopsis

Sha comprises the following extensions:

- H The hypervisor extension.
- Sstateen 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.

	<i>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.</i>
	<i>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).</i>

B.8.2. Instructions

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

hfence.gvma	No synopsis available
hfence.vvma	No synopsis available
hlv.b	No synopsis available
hlv.bu	No synopsis available
hlv.d	No synopsis available
hlv.h	No synopsis available
hlv.hu	No synopsis available
hlv.w	No synopsis available
hlv.wu	No synopsis available
hlvx.hu	No synopsis available
hlvx.wu	No synopsis available
hsv.b	No synopsis available
hsv.d	No synopsis available
hsv.h	No synopsis available
hsv.w	No synopsis available
hinval.gvma	Invalidate cached address translations
hinval.vvma	Invalidate cached address translations

B.8.3. CSRs

The following 19 CSRs are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

Name	Long Name	Address	Mode
hcounteren	Hypervisor Counter Enable	0x606	S
hedeleg	Hypervisor Exception Delegation	0x602	S
hedelegh	Hypervisor Exception Delegation High	0x612	S
henvcfg	Hypervisor Environment Configuration	0x60a	S
henvcfggh	most-significant 32 bits of Hypervisor Environment Configuration	0x61a	S
hgap	Hypervisor guest address translation and protection	0x680	S
hstatus	Hypervisor Status	0x600	S
htimedelta	Hypervisor time delta	0x605	S
htimedeltah	Hypervisor time delta, upper half	0x615	S
htinst	Hypervisor Trap Instruction Register	0x64a	S
htval	Hypervisor Trap Value Register	0x643	M
mtinst	Machine Trap Instruction Register	0x34a	M
mtval2	Machine Second Trap Value Register	0x34b	M
vsatp	Virtual Supervisor Address Translation and Protection	0x280	VS
vscause	Virtual Supervisor Cause	0x242	VS
vsepc	Virtual Supervisor Exception Program Counter	0x241	VS
vsstatus	Virtual Supervisor Status	0x200	VS
vstval	Virtual supervisor Trap Value	0x243	S
vstvec	Supervisor Trap Vector	0x205	S

B.9. 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

1.0.0

State
ratified

Ratification document
github.com/riscv/riscv-profiles/releases/tag/v1.0

B.9.1. 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.10. Extension Sscofpmf

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

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

1.0.0

State
ratified

Ratification date
2023-08

Ratification document
drive.google.com/file/d/1KcjgbLM5L1ZKY8934aJl8aQwGIMz6Cbo/view?usp=drive_link

B.10.1. Synopsis

Counter Overflow and Privilege Mode Filtering

-  Platforms may choose to mandate the presence of Sscofpmf.

B.10.2. CSRs

The following 58 CSRs are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

Name	Long Name	Address	Mode
hpmcounter10h	User-mode Hardware Performance Counter 7, high half	0xc8a	U
hpmcounter11h	User-mode Hardware Performance Counter 8, high half	0xc8b	U
hpmcounter12h	User-mode Hardware Performance Counter 9, high half	0xc8c	U
hpmcounter13h	User-mode Hardware Performance Counter 10, high half	0xc8d	U
hpmcounter14h	User-mode Hardware Performance Counter 11, high half	0xc8e	U
hpmcounter15h	User-mode Hardware Performance Counter 12, high half	0xc8f	U
hpmcounter16h	User-mode Hardware Performance Counter 13, high half	0xc90	U
hpmcounter17h	User-mode Hardware Performance Counter 14, high half	0xc91	U
hpmcounter18h	User-mode Hardware Performance Counter 15, high half	0xc92	U
hpmcounter19h	User-mode Hardware Performance Counter 16, high half	0xc93	U
hpmcounter20h	User-mode Hardware Performance Counter 17, high half	0xc94	U
hpmcounter21h	User-mode Hardware Performance Counter 18, high half	0xc95	U
hpmcounter22h	User-mode Hardware Performance Counter 19, high half	0xc96	U

Name	Long Name	Address	Mode
hpmcounter23h	User-mode Hardware Performance Counter 20, high half	0xc97	U
hpmcounter24h	User-mode Hardware Performance Counter 21, high half	0xc98	U
hpmcounter25h	User-mode Hardware Performance Counter 22, high half	0xc99	U
hpmcounter26h	User-mode Hardware Performance Counter 23, high half	0xc9a	U
hpmcounter27h	User-mode Hardware Performance Counter 24, high half	0xc9b	U
hpmcounter28h	User-mode Hardware Performance Counter 25, high half	0xc9c	U
hpmcounter29h	User-mode Hardware Performance Counter 26, high half	0xc9d	U
hpmcounter30h	User-mode Hardware Performance Counter 27, high half	0xc9e	U
hpmcounter31h	User-mode Hardware Performance Counter 28, high half	0xc9f	U
hpmcounter3h	User-mode Hardware Performance Counter 0, high half	0xc83	U
hpmcounter4h	User-mode Hardware Performance Counter 1, high half	0xc84	U
hpmcounter5h	User-mode Hardware Performance Counter 2, high half	0xc85	U
hpmcounter6h	User-mode Hardware Performance Counter 3, high half	0xc86	U
hpmcounter7h	User-mode Hardware Performance Counter 4, high half	0xc87	U
hpmcounter8h	User-mode Hardware Performance Counter 5, high half	0xc88	U
hpmcounter9h	User-mode Hardware Performance Counter 6, high half	0xc89	U
mhpmevent10h	Machine Hardware Performance Counter 10 Control, High half	0x72a	M
mhpmevent11h	Machine Hardware Performance Counter 11 Control, High half	0x72b	M
mhpmevent12h	Machine Hardware Performance Counter 12 Control, High half	0x72c	M
mhpmevent13h	Machine Hardware Performance Counter 13 Control, High half	0x72d	M
mhpmevent14h	Machine Hardware Performance Counter 14 Control, High half	0x72e	M
mhpmevent15h	Machine Hardware Performance Counter 15 Control, High half	0x72f	M
mhpmevent16h	Machine Hardware Performance Counter 16 Control, High half	0x730	M
mhpmevent17h	Machine Hardware Performance Counter 17 Control, High half	0x731	M
mhpmevent18h	Machine Hardware Performance Counter 18 Control, High half	0x732	M
mhpmevent19h	Machine Hardware Performance Counter 19 Control, High half	0x733	M
mhpmevent20h	Machine Hardware Performance Counter 20 Control, High half	0x734	M
mhpmevent21h	Machine Hardware Performance Counter 21 Control, High half	0x735	M
mhpmevent22h	Machine Hardware Performance Counter 22 Control, High half	0x736	M
mhpmevent23h	Machine Hardware Performance Counter 23 Control, High half	0x737	M
mhpmevent24h	Machine Hardware Performance Counter 24 Control, High half	0x738	M
mhpmevent25h	Machine Hardware Performance Counter 25 Control, High half	0x739	M
mhpmevent26h	Machine Hardware Performance Counter 26 Control, High half	0x73a	M
mhpmevent27h	Machine Hardware Performance Counter 27 Control, High half	0x73b	M
mhpmevent28h	Machine Hardware Performance Counter 28 Control, High half	0x73c	M
mhpmevent29h	Machine Hardware Performance Counter 29 Control, High half	0x73d	M
mhpmevent30h	Machine Hardware Performance Counter 30 Control, High half	0x73e	M
mhpmevent31h	Machine Hardware Performance Counter 31 Control, High half	0x73f	M
mhpmevent3h	Machine Hardware Performance Counter 3 Control, High half	0x723	M
mhpmevent4h	Machine Hardware Performance Counter 4 Control, High half	0x724	M
mhpmevent5h	Machine Hardware Performance Counter 5 Control, High half	0x725	M
mhpmevent6h	Machine Hardware Performance Counter 6 Control, High half	0x726	M
mhpmevent7h	Machine Hardware Performance Counter 7 Control, High half	0x727	M
mhpmevent8h	Machine Hardware Performance Counter 8 Control, High half	0x728	M
mhpmevent9h	Machine Hardware Performance Counter 9 Control, High half	0x729	M

B.11. Extension Sscouterenw

Long Name: Supervisor counter enable

Version Requirement: = 1.0

Sscouterenw Extension Presence

profile	v1.0.0
RVA22U64	-

profile	v1.0.0
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2023-08

Ratification document

drive.google.com/file/d/1KcjgbLM5L1ZKY8934aJl8aQwGIMz6Cbo/view?usp=drive_link

B.11.1. 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.12. Extension Sstc

Long Name: Supervisor-mode timer interrupts

Version Requirement: ~> 1.0

Sstc Extension Presence

profile	v0.9.0
RVA22U64	-
RVA22S64	optional

0.9.0

State

ratified

Ratification document

drive.google.com/file/d/1m84Re2yK8m_vbW7TspvevCDR82MOBaSX/view?usp=drive_link

B.12.1. 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.13. Extension Sstvala

Long Name: [stval](#) requirements for RVA profiles

Version Requirement: = 1.0

Sstvala Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

1.0.0

State

ratified

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

B.13.1. 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.14. Extension Sstvecd

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

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

1.0.0

State

ratified

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

B.14.1. 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.15. Extension Ssu64xl

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

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

1.0.0

State

ratified

B.15.1. 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.16. Extension Sv39

Long Name: 39-bit virtual address translation (3 level)
Version Requirement: = 1.11
Sv39 Extension Presence

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

1.11.0

State

ratified

1.12.0

State

ratified

Ratification document

github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

1.13.0

State

ratified

B.16.1. Synopsis

39-bit virtual address translation (3 level)

B.17. 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

1.11.0

State

ratified

1.12.0

State

ratified

Ratification document

github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

1.13.0

State

ratified

B.17.1. Synopsis

48-bit virtual address translation (4 level)

B.18. 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	-	-	-
RVA22S64	optional	optional	optional

1.11.0

State

ratified

1.12.0

State

ratified

Ratification document

github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

1.13.0

State

ratified

B.18.1. Synopsis

57-bit virtual address translation (5 level)

B.19. Extension Svade

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

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2023-11

Ratification document

github.com/riscvarchive/riscv-svadu/releases/download/v1.0/riscv-svadu.pdf

B.19.1. 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.20. Extension Svbare

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

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

1.0.0

State

ratified

B.20.1. 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.21. Extension Svinval

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

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2021-11

B.21.1. 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.vma](#) 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.vma](#) and [hfence.gvma](#), respectively, instead of [sfence.vma](#). In addition, [hinval.gvma](#) uses VMIDs instead of ASIDs.

[sinval.vma](#), [hinval.vma](#), and [hinval.gvma](#) require the same permissions and raise the same exceptions as [sfence.vma](#), [hfence.vma](#), 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.vma](#) 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.vma](#), 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.21.2. Instructions

The following 5 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

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

B.22. 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

1.0.0
State
ratified

Ratification date

2021-11

B.22.1. 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.23. Extension Svpbmt

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

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

1.0.0
State
ratified

B.23.1. Synopsis

This extension mandates that the [satp](#) mode Bare must be supported.

 This extension was ratified as part of the RVA22 profile.

B.24. Extension U

Long Name: User-mode privilege level
Version Requirement: ~> 2.0
U Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0
State
ratified

Ratification date
2019-12

B.24.1. Synopsis

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

B.24.2. CSRs

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

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

B.24.3. Parameters

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

MUTABLE_MISA_U
Indicates whether or not the [U](#) extension can be disabled with the [misa.U](#) bit.

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. Can be one of:

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

U_MODE_ENDIANNES

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.25. Extension V

Long Name: Variable-length vector

Version Requirement: ~> 1.0

V Extension Presence

profile	v1.0.0
RVA22U64	optional
RVA22S64	optional

1.0.0

State

ratified

B.25.1. Synopsis

TODO



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.25.2. Instructions

The following 627 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

vaadd.vv	No synopsis available
vaadd.vx	No synopsis available
vaaddu.vv	No synopsis available
vaaddu.vx	No synopsis available
vadc.vim	No synopsis available
vadc.vvm	No synopsis available
vadc.vxm	No synopsis available
vadd.vi	No synopsis available
vadd.vv	No synopsis available
vadd.vx	No synopsis available
vand.vi	No synopsis available
vand.vv	No synopsis available
vand.vx	No synopsis available
vasub.vv	No synopsis available
vasub.vx	No synopsis available
vasubu.vv	No synopsis available
vasubu.vx	No synopsis available
vcompress.vm	No synopsis available

vcpop.m	No synopsis available
vdiv.vv	No synopsis available
vdiv.vx	No synopsis available
vdivu.vv	No synopsis available
vdivu.vx	No synopsis available
vfadd.vf	No synopsis available
vfadd.vv	No synopsis available
vfclass.v	No synopsis available
vfcvt.f.x.v	No synopsis available
vfcvt.f.xu.v	No synopsis available
vfcvt.rtz.x.f.v	No synopsis available
vfcvt.rtz.xu.f.v	No synopsis available
vfcvt.x.f.v	No synopsis available
vfcvt.xu.f.v	No synopsis available
vfdiv.vf	No synopsis available
vfdiv.vv	No synopsis available
vfirst.m	No synopsis available
vfmac.vf	No synopsis available
vfmac.vv	No synopsis available
vfmad.vf	No synopsis available
vfmad.vv	No synopsis available
vfmax.vf	No synopsis available
vfmax.vv	No synopsis available
vfmerge.vfm	No synopsis available
vfmin.vf	No synopsis available
vfmin.vv	No synopsis available
vfmsac.vf	No synopsis available
vfmsac.vv	No synopsis available
vfmsub.vf	No synopsis available
vfmsub.vv	No synopsis available
vfmul.vf	No synopsis available
vfmul.vv	No synopsis available
vfmv.f.s	No synopsis available
vfmv.s.f	No synopsis available
vfmv.v.f	No synopsis available
vfncvt.f.f.w	No synopsis available
vfncvt.f.x.w	No synopsis available
vfncvt.f.xu.w	No synopsis available
vfncvt.rod.f.f.w	No synopsis available
vfncvt.rtz.x.f.w	No synopsis available
vfncvt.rtz.xu.f.w	No synopsis available
vfncvt.x.f.w	No synopsis available
vfncvt.xu.f.w	No synopsis available
vfnmac.vf	No synopsis available
vfnmac.vv	No synopsis available
vfnmad.vf	No synopsis available
vfnmad.vv	No synopsis available
vfnmsac.vf	No synopsis available
vfnmsac.vv	No synopsis available
vfnmsub.vf	No synopsis available
vfnmsub.vv	No synopsis available
vfrdiv.vf	No synopsis available
vfrec7.v	No synopsis available

vfredmax.vs	No synopsis available
vfredmin.vs	No synopsis available
vfredosum.vs	No synopsis available
vfredusum.vs	No synopsis available
vfrsqr7.v	No synopsis available
vfrsub.vf	No synopsis available
vfsgnj.vf	No synopsis available
vfsgnj.vv	No synopsis available
vfsgnjn.vf	No synopsis available
vfsgnjn.vv	No synopsis available
vfsgnjx.vf	No synopsis available
vfsgnjx.vv	No synopsis available
vfslide1down.vf	No synopsis available
vfslide1up.vf	No synopsis available
vfsqrt.v	No synopsis available
vfsub.vf	No synopsis available
vfsub.vv	No synopsis available
vfwadd.vf	No synopsis available
vfwadd.vv	No synopsis available
vfwadd.wf	No synopsis available
vfwadd.wv	No synopsis available
vfwcvt.f.f.v	No synopsis available
vfwcvt.f.x.v	No synopsis available
vfwcvt.f.xu.v	No synopsis available
vfwcvt.rtz.x.f.v	No synopsis available
vfwcvt.rtz.xu.f.v	No synopsis available
vfwcvt.x.f.v	No synopsis available
vfwcvt.xu.f.v	No synopsis available
vfwmacc.vf	No synopsis available
vfwmacc.vv	No synopsis available
vfwmsac.vf	No synopsis available
vfwmsac.vv	No synopsis available
vfwmul.vf	No synopsis available
vfwmul.vv	No synopsis available
vfwnmacc.vf	No synopsis available
vfwnmacc.vv	No synopsis available
vfwnmsac.vf	No synopsis available
vfwnmsac.vv	No synopsis available
vfwredosum.vs	No synopsis available
vfwredusum.vs	No synopsis available
vfwsb.vf	No synopsis available
vfwsb.vv	No synopsis available
vfwsb.wf	No synopsis available
vfwsb.wv	No synopsis available
vid.v	No synopsis available
viota.m	No synopsis available
vl1re16.v	No synopsis available
vl1re32.v	No synopsis available
vl1re64.v	No synopsis available
vl1re8.v	No synopsis available
vl2re16.v	No synopsis available
vl2re32.v	No synopsis available
vl2re64.v	No synopsis available

vl2re8.v	No synopsis available
vl4re16.v	No synopsis available
vl4re32.v	No synopsis available
vl4re64.v	No synopsis available
vl4re8.v	No synopsis available
vl8re16.v	No synopsis available
vl8re32.v	No synopsis available
vl8re64.v	No synopsis available
vl8re8.v	No synopsis available
vle16.v	No synopsis available
vle16ff.v	No synopsis available
vle32.v	No synopsis available
vle32ff.v	No synopsis available
vle64.v	No synopsis available
vle64ff.v	No synopsis available
vle8.v	No synopsis available
vle8ff.v	No synopsis available
vlm.v	No synopsis available
vloxei16.v	No synopsis available
vloxei32.v	No synopsis available
vloxei64.v	No synopsis available
vloxei8.v	No synopsis available
vloxseg2ei16.v	No synopsis available
vloxseg2ei32.v	No synopsis available
vloxseg2ei64.v	No synopsis available
vloxseg2ei8.v	No synopsis available
vloxseg3ei16.v	No synopsis available
vloxseg3ei32.v	No synopsis available
vloxseg3ei64.v	No synopsis available
vloxseg3ei8.v	No synopsis available
vloxseg4ei16.v	No synopsis available
vloxseg4ei32.v	No synopsis available
vloxseg4ei64.v	No synopsis available
vloxseg4ei8.v	No synopsis available
vloxseg5ei16.v	No synopsis available
vloxseg5ei32.v	No synopsis available
vloxseg5ei64.v	No synopsis available
vloxseg5ei8.v	No synopsis available
vloxseg6ei16.v	No synopsis available
vloxseg6ei32.v	No synopsis available
vloxseg6ei64.v	No synopsis available
vloxseg6ei8.v	No synopsis available
vloxseg7ei16.v	No synopsis available
vloxseg7ei32.v	No synopsis available
vloxseg7ei64.v	No synopsis available
vloxseg7ei8.v	No synopsis available
vloxseg8ei16.v	No synopsis available
vloxseg8ei32.v	No synopsis available
vloxseg8ei64.v	No synopsis available
vloxseg8ei8.v	No synopsis available
vlse16.v	No synopsis available
vlse32.v	No synopsis available
vlse64.v	No synopsis available

vlse8.v	No synopsis available
vlseg2e16.v	No synopsis available
vlseg2e16ff.v	No synopsis available
vlseg2e32.v	No synopsis available
vlseg2e32ff.v	No synopsis available
vlseg2e64.v	No synopsis available
vlseg2e64ff.v	No synopsis available
vlseg2e8.v	No synopsis available
vlseg2e8ff.v	No synopsis available
vlseg3e16.v	No synopsis available
vlseg3e16ff.v	No synopsis available
vlseg3e32.v	No synopsis available
vlseg3e32ff.v	No synopsis available
vlseg3e64.v	No synopsis available
vlseg3e64ff.v	No synopsis available
vlseg3e8.v	No synopsis available
vlseg3e8ff.v	No synopsis available
vlseg4e16.v	No synopsis available
vlseg4e16ff.v	No synopsis available
vlseg4e32.v	No synopsis available
vlseg4e32ff.v	No synopsis available
vlseg4e64.v	No synopsis available
vlseg4e64ff.v	No synopsis available
vlseg4e8.v	No synopsis available
vlseg4e8ff.v	No synopsis available
vlseg5e16.v	No synopsis available
vlseg5e16ff.v	No synopsis available
vlseg5e32.v	No synopsis available
vlseg5e32ff.v	No synopsis available
vlseg5e64.v	No synopsis available
vlseg5e64ff.v	No synopsis available
vlseg5e8.v	No synopsis available
vlseg5e8ff.v	No synopsis available
vlseg6e16.v	No synopsis available
vlseg6e16ff.v	No synopsis available
vlseg6e32.v	No synopsis available
vlseg6e32ff.v	No synopsis available
vlseg6e64.v	No synopsis available
vlseg6e64ff.v	No synopsis available
vlseg6e8.v	No synopsis available
vlseg6e8ff.v	No synopsis available
vlseg7e16.v	No synopsis available
vlseg7e16ff.v	No synopsis available
vlseg7e32.v	No synopsis available
vlseg7e32ff.v	No synopsis available
vlseg7e64.v	No synopsis available
vlseg7e64ff.v	No synopsis available
vlseg7e8.v	No synopsis available
vlseg7e8ff.v	No synopsis available
vlseg8e16.v	No synopsis available
vlseg8e16ff.v	No synopsis available
vlseg8e32.v	No synopsis available
vlseg8e32ff.v	No synopsis available

vlseg8e64.v	No synopsis available
vlseg8e64ff.v	No synopsis available
vlseg8e8.v	No synopsis available
vlseg8e8ff.v	No synopsis available
vlsseg2e16.v	No synopsis available
vlsseg2e32.v	No synopsis available
vlsseg2e64.v	No synopsis available
vlsseg2e8.v	No synopsis available
vlsseg3e16.v	No synopsis available
vlsseg3e32.v	No synopsis available
vlsseg3e64.v	No synopsis available
vlsseg3e8.v	No synopsis available
vlsseg4e16.v	No synopsis available
vlsseg4e32.v	No synopsis available
vlsseg4e64.v	No synopsis available
vlsseg4e8.v	No synopsis available
vlsseg5e16.v	No synopsis available
vlsseg5e32.v	No synopsis available
vlsseg5e64.v	No synopsis available
vlsseg5e8.v	No synopsis available
vlsseg6e16.v	No synopsis available
vlsseg6e32.v	No synopsis available
vlsseg6e64.v	No synopsis available
vlsseg6e8.v	No synopsis available
vlsseg7e16.v	No synopsis available
vlsseg7e32.v	No synopsis available
vlsseg7e64.v	No synopsis available
vlsseg7e8.v	No synopsis available
vlsseg8e16.v	No synopsis available
vlsseg8e32.v	No synopsis available
vlsseg8e64.v	No synopsis available
vlsseg8e8.v	No synopsis available
vluxei16.v	No synopsis available
vluxei32.v	No synopsis available
vluxei64.v	No synopsis available
vluxei8.v	No synopsis available
vluxseg2ei16.v	No synopsis available
vluxseg2ei32.v	No synopsis available
vluxseg2ei64.v	No synopsis available
vluxseg2ei8.v	No synopsis available
vluxseg3ei16.v	No synopsis available
vluxseg3ei32.v	No synopsis available
vluxseg3ei64.v	No synopsis available
vluxseg3ei8.v	No synopsis available
vluxseg4ei16.v	No synopsis available
vluxseg4ei32.v	No synopsis available
vluxseg4ei64.v	No synopsis available
vluxseg4ei8.v	No synopsis available
vluxseg5ei16.v	No synopsis available
vluxseg5ei32.v	No synopsis available
vluxseg5ei64.v	No synopsis available
vluxseg5ei8.v	No synopsis available
vluxseg6ei16.v	No synopsis available

vluxseg6ei32.v	No synopsis available
vluxseg6ei64.v	No synopsis available
vluxseg6ei8.v	No synopsis available
vluxseg7ei16.v	No synopsis available
vluxseg7ei32.v	No synopsis available
vluxseg7ei64.v	No synopsis available
vluxseg7ei8.v	No synopsis available
vluxseg8ei16.v	No synopsis available
vluxseg8ei32.v	No synopsis available
vluxseg8ei64.v	No synopsis available
vluxseg8ei8.v	No synopsis available
vmacc.vv	No synopsis available
vmacc.vx	No synopsis available
vmadc.vi	No synopsis available
vmadc.vim	No synopsis available
vmadc.vv	No synopsis available
vmadc.vvm	No synopsis available
vmadc.vx	No synopsis available
vmadc.vxm	No synopsis available
vmadd.vv	No synopsis available
vmadd.vx	No synopsis available
vmand.mm	No synopsis available
vmandn.mm	No synopsis available
vmax.vv	No synopsis available
vmax.vx	No synopsis available
vmaxu.vv	No synopsis available
vmaxu.vx	No synopsis available
vmerge.vim	No synopsis available
vmerge.vvm	No synopsis available
vmerge.vxm	No synopsis available
vmfeq.vf	No synopsis available
vmfeq.vv	No synopsis available
vmfge.vf	No synopsis available
vmfgt.vf	No synopsis available
vmfle.vf	No synopsis available
vmfle.vv	No synopsis available
vmflt.vf	No synopsis available
vmflt.vv	No synopsis available
vmfne.vf	No synopsis available
vmfne.vv	No synopsis available
vmin.vv	No synopsis available
vmin.vx	No synopsis available
vminu.vv	No synopsis available
vminu.vx	No synopsis available
vmnand.mm	No synopsis available
vmnor.mm	No synopsis available
vmor.mm	No synopsis available
vmorn.mm	No synopsis available
vmsbc.vv	No synopsis available
vmsbc.vvm	No synopsis available
vmsbc.vx	No synopsis available
vmsbc.vxm	No synopsis available
vmsbf.m	No synopsis available

vmseq.vi	No synopsis available
vmseq.vv	No synopsis available
vmseq.vx	No synopsis available
vmsgt.vi	No synopsis available
vmsgt.vx	No synopsis available
vmsgtu.vi	No synopsis available
vmsgtu.vx	No synopsis available
vmsif.m	No synopsis available
vmsle.vi	No synopsis available
vmsle.vv	No synopsis available
vmsle.vx	No synopsis available
vmsleu.vi	No synopsis available
vmsleu.vv	No synopsis available
vmsleu.vx	No synopsis available
vmslt.vv	No synopsis available
vmslt.vx	No synopsis available
vmsltu.vv	No synopsis available
vmsltu.vx	No synopsis available
vmsne.vi	No synopsis available
vmsne.vv	No synopsis available
vmsne.vx	No synopsis available
vmsof.m	No synopsis available
vmul.vv	No synopsis available
vmul.vx	No synopsis available
vmulh.vv	No synopsis available
vmulh.vx	No synopsis available
vmulhsu.vv	No synopsis available
vmulhsu.vx	No synopsis available
vmulhu.vv	No synopsis available
vmulhu.vx	No synopsis available
vmv.s.x	No synopsis available
vmv.v.i	No synopsis available
vmv.v.v	No synopsis available
vmv.v.x	No synopsis available
vmv.x.s	No synopsis available
vmv1r.v	No synopsis available
vmv2r.v	No synopsis available
vmv4r.v	No synopsis available
vmv8r.v	No synopsis available
vmxnor.mm	No synopsis available
vmxor.mm	No synopsis available
vnclip.wi	No synopsis available
vnclip.wv	No synopsis available
vnclip.wx	No synopsis available
vnclipu.wi	No synopsis available
vnclipu.wv	No synopsis available
vnclipu.wx	No synopsis available
vnmsac.vv	No synopsis available
vnmsac.vx	No synopsis available
vnmsub.vv	No synopsis available
vnmsub.vx	No synopsis available
vnsra.wi	No synopsis available
vnsra.wv	No synopsis available

vnsra.wx	No synopsis available
vnsrl.wi	No synopsis available
vnsrl.wv	No synopsis available
vnsrl.wx	No synopsis available
vor.vi	No synopsis available
vor.vv	No synopsis available
vor.vx	No synopsis available
vredand.vs	No synopsis available
vredmax.vs	No synopsis available
vredmaxu.vs	No synopsis available
vredmin.vs	No synopsis available
vredminu.vs	No synopsis available
vredor.vs	No synopsis available
vredsum.vs	No synopsis available
vredxor.vs	No synopsis available
vrem.vv	No synopsis available
vrem.vx	No synopsis available
vremu.vv	No synopsis available
vremu.vx	No synopsis available
vrgather.vi	No synopsis available
vrgather.vv	No synopsis available
vrgather.vx	No synopsis available
vrgatherei16.vv	No synopsis available
vrsub.vi	No synopsis available
vrsub.vx	No synopsis available
vs1r.v	No synopsis available
vs2r.v	No synopsis available
vs4r.v	No synopsis available
vs8r.v	No synopsis available
vsadd.vi	No synopsis available
vsadd.vv	No synopsis available
vsadd.vx	No synopsis available
vsaddu.vi	No synopsis available
vsaddu.vv	No synopsis available
vsaddu.vx	No synopsis available
vsbc.vvm	No synopsis available
vsbc.vxm	No synopsis available
vse16.v	No synopsis available
vse32.v	No synopsis available
vse64.v	No synopsis available
vse8.v	No synopsis available
vsetivli	No synopsis available
vsetvl	No synopsis available
vsetvli	No synopsis available
vsext.vf2	No synopsis available
vsext.vf4	No synopsis available
vsext.vf8	No synopsis available
vslide1down.vx	No synopsis available
vslide1up.vx	No synopsis available
vslidedown.vi	No synopsis available
vslidedown.vx	No synopsis available
vslideup.vi	No synopsis available
vslideup.vx	No synopsis available

vsll.vi	No synopsis available
vsll.vv	No synopsis available
vsll.vx	No synopsis available
vsm.v	No synopsis available
vsmul.vv	No synopsis available
vsmul.vx	No synopsis available
vsoxei16.v	No synopsis available
vsoxei32.v	No synopsis available
vsoxei64.v	No synopsis available
vsoxei8.v	No synopsis available
vsoxseg2ei16.v	No synopsis available
vsoxseg2ei32.v	No synopsis available
vsoxseg2ei64.v	No synopsis available
vsoxseg2ei8.v	No synopsis available
vsoxseg3ei16.v	No synopsis available
vsoxseg3ei32.v	No synopsis available
vsoxseg3ei64.v	No synopsis available
vsoxseg3ei8.v	No synopsis available
vsoxseg4ei16.v	No synopsis available
vsoxseg4ei32.v	No synopsis available
vsoxseg4ei64.v	No synopsis available
vsoxseg4ei8.v	No synopsis available
vsoxseg5ei16.v	No synopsis available
vsoxseg5ei32.v	No synopsis available
vsoxseg5ei64.v	No synopsis available
vsoxseg5ei8.v	No synopsis available
vsoxseg6ei16.v	No synopsis available
vsoxseg6ei32.v	No synopsis available
vsoxseg6ei64.v	No synopsis available
vsoxseg6ei8.v	No synopsis available
vsoxseg7ei16.v	No synopsis available
vsoxseg7ei32.v	No synopsis available
vsoxseg7ei64.v	No synopsis available
vsoxseg7ei8.v	No synopsis available
vsoxseg8ei16.v	No synopsis available
vsoxseg8ei32.v	No synopsis available
vsoxseg8ei64.v	No synopsis available
vsoxseg8ei8.v	No synopsis available
vsra.vi	No synopsis available
vsra.vv	No synopsis available
vsra.vx	No synopsis available
vsrl.vi	No synopsis available
vsrl.vv	No synopsis available
vsrl.vx	No synopsis available
vsse16.v	No synopsis available
vsse32.v	No synopsis available
vsse64.v	No synopsis available
vsse8.v	No synopsis available
vsseg2e16.v	No synopsis available
vsseg2e32.v	No synopsis available
vsseg2e64.v	No synopsis available
vsseg2e8.v	No synopsis available
vsseg3e16.v	No synopsis available

vsseg3e32.v	No synopsis available
vsseg3e64.v	No synopsis available
vsseg3e8.v	No synopsis available
vsseg4e16.v	No synopsis available
vsseg4e32.v	No synopsis available
vsseg4e64.v	No synopsis available
vsseg4e8.v	No synopsis available
vsseg5e16.v	No synopsis available
vsseg5e32.v	No synopsis available
vsseg5e64.v	No synopsis available
vsseg5e8.v	No synopsis available
vsseg6e16.v	No synopsis available
vsseg6e32.v	No synopsis available
vsseg6e64.v	No synopsis available
vsseg6e8.v	No synopsis available
vsseg7e16.v	No synopsis available
vsseg7e32.v	No synopsis available
vsseg7e64.v	No synopsis available
vsseg7e8.v	No synopsis available
vsseg8e16.v	No synopsis available
vsseg8e32.v	No synopsis available
vsseg8e64.v	No synopsis available
vsseg8e8.v	No synopsis available
vssra.vi	No synopsis available
vssra.vv	No synopsis available
vssra.vx	No synopsis available
vssrl.vi	No synopsis available
vssrl.vv	No synopsis available
vssrl.vx	No synopsis available
vssseg2e16.v	No synopsis available
vssseg2e32.v	No synopsis available
vssseg2e64.v	No synopsis available
vssseg2e8.v	No synopsis available
vssseg3e16.v	No synopsis available
vssseg3e32.v	No synopsis available
vssseg3e64.v	No synopsis available
vssseg3e8.v	No synopsis available
vssseg4e16.v	No synopsis available
vssseg4e32.v	No synopsis available
vssseg4e64.v	No synopsis available
vssseg4e8.v	No synopsis available
vssseg5e16.v	No synopsis available
vssseg5e32.v	No synopsis available
vssseg5e64.v	No synopsis available
vssseg5e8.v	No synopsis available
vssseg6e16.v	No synopsis available
vssseg6e32.v	No synopsis available
vssseg6e64.v	No synopsis available
vssseg6e8.v	No synopsis available
vssseg7e16.v	No synopsis available
vssseg7e32.v	No synopsis available
vssseg7e64.v	No synopsis available
vssseg7e8.v	No synopsis available

vssseg8e16.v	No synopsis available
vssseg8e32.v	No synopsis available
vssseg8e64.v	No synopsis available
vssseg8e8.v	No synopsis available
vssub.vv	No synopsis available
vssub.vx	No synopsis available
vssubu.vv	No synopsis available
vssubu.vx	No synopsis available
vsub.vv	No synopsis available
vsub.vx	No synopsis available
vsuxei16.v	No synopsis available
vsuxei32.v	No synopsis available
vsuxei64.v	No synopsis available
vsuxei8.v	No synopsis available
vsuxseg2ei16.v	No synopsis available
vsuxseg2ei32.v	No synopsis available
vsuxseg2ei64.v	No synopsis available
vsuxseg2ei8.v	No synopsis available
vsuxseg3ei16.v	No synopsis available
vsuxseg3ei32.v	No synopsis available
vsuxseg3ei64.v	No synopsis available
vsuxseg3ei8.v	No synopsis available
vsuxseg4ei16.v	No synopsis available
vsuxseg4ei32.v	No synopsis available
vsuxseg4ei64.v	No synopsis available
vsuxseg4ei8.v	No synopsis available
vsuxseg5ei16.v	No synopsis available
vsuxseg5ei32.v	No synopsis available
vsuxseg5ei64.v	No synopsis available
vsuxseg5ei8.v	No synopsis available
vsuxseg6ei16.v	No synopsis available
vsuxseg6ei32.v	No synopsis available
vsuxseg6ei64.v	No synopsis available
vsuxseg6ei8.v	No synopsis available
vsuxseg7ei16.v	No synopsis available
vsuxseg7ei32.v	No synopsis available
vsuxseg7ei64.v	No synopsis available
vsuxseg7ei8.v	No synopsis available
vsuxseg8ei16.v	No synopsis available
vsuxseg8ei32.v	No synopsis available
vsuxseg8ei64.v	No synopsis available
vsuxseg8ei8.v	No synopsis available
vwadd.vv	No synopsis available
vwadd.vx	No synopsis available
vwadd.wv	No synopsis available
vwadd.wx	No synopsis available
vwaddu.vv	No synopsis available
vwaddu.vx	No synopsis available
vwaddu.wv	No synopsis available
vwaddu.wx	No synopsis available
vwmacc.vv	No synopsis available
vwmacc.vx	No synopsis available
vwmaccsu.vv	No synopsis available

vwmaccsu.vx	No synopsis available
vwmaccu.vv	No synopsis available
vwmaccu.vx	No synopsis available
vwmaccus.vx	No synopsis available
vwmul.vv	No synopsis available
vwmul.vx	No synopsis available
vwmulsu.vv	No synopsis available
vwmulsu.vx	No synopsis available
vwmulu.vv	No synopsis available
vwmulu.vx	No synopsis available
vwredsum.vs	No synopsis available
vwredsumu.vs	No synopsis available
vwsub.vv	No synopsis available
vwsub.vx	No synopsis available
vwsub.wv	No synopsis available
vwsub.wx	No synopsis available
vwsubu.vv	No synopsis available
vwsubu.vx	No synopsis available
vwsubu.wv	No synopsis available
vwsubu.wx	No synopsis available
vxor.vi	No synopsis available
vxor.vv	No synopsis available
vxor.vx	No synopsis available
vzext.vf2	No synopsis available
vzext.vf4	No synopsis available
vzext.vf8	No synopsis available

B.25.3. Parameters

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

HW_MSTATUS_VS_DIRTY_UPDATE

Indicates whether or not hardware will write to [mstatus.VS](#)

Values are:

never	Hardware never writes mstatus.VS
precise	Hardware writes mstatus.VS to the Dirty (3) state precisely when V registers are modified
imprecise	Hardware writes mstatus.VS imprecisely. This will result in a call to unpredictable() on any attempt to read mstatus or write vector state.

MSTATUS_VS_LEGAL_VALUES

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

MUTABLE_MISA_V

Indicates whether or not the [V](#) extension can be disabled with the [misa.V](#) bit.

B.26. Extension Za128rs

Long Name: Reservation set requirement for RVA profiles

Version Requirement: = 1.0

Za128rs Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

B.26.1. 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.27. Extension Zba

Long Name: Address generation instructions

Version Requirement: ~> 1.0

Zba Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2021-06

B.27.1. 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.27.2. Instructions

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

add.uw	Add unsigned word
sh1add.uw	Shift unsigned word left by 1 and add
sh1add	Shift left by 1 and add
sh2add.uw	Shift unsigned word left by 2 and add
sh2add	Shift left by 2 and add
sh3add.uw	Shift unsigned word left by 3 and add
sh3add	Shift left by 3 and add
slli.uw	Shift left unsigned word (Immediate)

B.28. Extension Zbb

Long Name: Basic bit manipulation

Version Requirement: ~> 1.0

Zbb Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2021-06

B.28.1. Synopsis

Basic bit manipulation

B.28.2. Instructions

The following 23 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

andn	AND with inverted operand
orn	OR with inverted operand
rev8	Byte-reverse register (RV64 encoding)
rol	Rotate left (Register)
rolw	Rotate left word (Register)
ror	Rotate right (Register)
rori	Rotate right (Immediate)
roriw	Rotate right word (Immediate)
rorw	Rotate right word (Register)
xnor	Exclusive NOR
clz	Count leading zero bits
clzw	Count leading zero bits in word
cpop	Count set bits
cpopw	Count set bits in word
ctz	Count trailing zero bits
ctzw	Count trailing zero bits in word
max	Maximum
maxu	Unsigned maximum
min	Minimum
minu	Unsigned minimum
orc.b	Bitware OR-combine, byte granule
sext.b	Sign-extend byte
sext.h	Sign-extend halfword

B.29. Extension Zbs

Long Name: Single-bit instructions

Version Requirement: ~> 1.0

Zbs Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2021-06

Ratification document

drive.google.com/drive/u/0/folders/1_wqb-rXOVkGa6rqmugN3kwCftWDf1daU

B.29.1. 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.29.2. Instructions

The following 8 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies 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.30. Extension Zfh

Long Name: Half-precision floating point

Version Requirement: ~> 1.0

Zfh Extension Presence

profile	v1.0.0
RVA22U64	optional
RVA22S64	optional

1.0.0

State

ratified

B.30.1. 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.30.2. Instructions

The following 42 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

fcvt.h.q	No synopsis available
fcvt.q.h	No synopsis available
fadd.h	No synopsis available
fclass.h	No synopsis available
fcvt.d.h	No synopsis available
fcvt.h.d	No synopsis available
fcvt.h.l	No synopsis available
fcvt.h.lu	No synopsis available
fcvt.h.s	Convert half-precision float to a single-precision float
fcvt.h.w	No synopsis available
fcvt.h.wu	No synopsis available
fcvt.l.h	No synopsis available
fcvt.lu.h	No synopsis available
fcvt.s.h	Convert single-precision float to a half-precision float
fcvt.w.h	No synopsis available
fcvt.wu.h	No synopsis available
fdiv.h	No synopsis available
feq.h	No synopsis available
fle.h	No synopsis available
fleq.h	No synopsis available
flh	Half-precision floating-point load
fli.h	No synopsis available
flt.h	No synopsis available
fltq.h	No synopsis available
fmadd.h	No synopsis available
fmax.h	No synopsis available

fmaxm.h	No synopsis available
fmin.h	No synopsis available
fminm.h	No synopsis available
fmsub.h	No synopsis available
fmul.h	No synopsis available
fmv.x.h	Move half-precision value from floating-point to integer register
fnmadd.h	No synopsis available
fnmsub.h	No synopsis available
fround.h	No synopsis available
froundnx.h	No synopsis available
fsgnj.h	No synopsis available
fsgnjn.h	No synopsis available
fsgnjx.h	No synopsis available
fsh	Half-precision floating-point store
fsqrt.h	No synopsis available
fsub.h	No synopsis available

B.31. Extension Zfhmin

Long Name: Minimal half-precision Floating-point

Version Requirement: ~> 1.0

Zfhmin Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2021-11

B.31.1. 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.

i

[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.

i

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.31.2. Instructions

The following 5 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies 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.x.h	Move half-precision value from floating-point to integer register
fsh	Half-precision floating-point store

B.32. Extension Zic64b

Long Name: 64-byte cache blocks

Version Requirement: = 1.0

Zic64b Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification document

github.com/riscv/riscv-profiles/releases/tag/v1.0

B.32.1. 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.33. Extension Zicbom

Long Name: Cache block management instructions

Version Requirement: ~> 1.0

Zicbom Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2022-05

B.33.1. Synopsis

Cache block management instructions

B.33.2. Instructions

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

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

B.33.3. Parameters

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

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.34. Extension Zicbop

Long Name: Cache block prefetch

Version Requirement: ~> 1.0

Zicbop Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2022-05

B.34.1. Synopsis

Cache block prefetch instruction



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.34.2. Parameters

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

B.35. Extension Zicboz

Long Name: Cache block zero instruction

Version Requirement: ~> 1.0

Zicboz Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

Ratification date

2022-05

B.35.1. Synopsis

Cache block zero instruction

B.35.2. Instructions

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

<code>cbo.zero</code>	Cache Block Zero

B.35.3. Parameters

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

B.36. Extension Ziccamoa

Long Name: Main memory atomicity requirement for RVA profiles

Version Requirement: = 1.0

Ziccamoa Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

B.36.1. 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.37. Extension Ziccif

Long Name: Main memory fetch requirement for RVA profiles

Version Requirement: = 1.0

Ziccif Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

B.37.1. 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.38. Extension Zicclsm

Long Name: Main memory misaligned requirement for RVA profiles

Version Requirement: = 1.0

Zicclsm Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State

ratified

B.38.1. Synopsis

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

- i

This extension was ratified as part of the RVA20 profile.
- i

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.
- i

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.39. Extension Ziccrse

Long Name: Main memory reservability requirement for RVA profiles

Version Requirement: = 1.0

Ziccrse Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

1.0.0

State
ratified

B.39.1. Synopsis

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

- i

This extension was ratified as part of the RVA20 profile.
- i

Ziccrse is a profile-defined extension introduced with RVA20.

B.40. Extension Zicntr

Long Name: Architectural performance counters

Version Requirement: = 2.0

Zicntr Extension Presence

profile	v2.0.0
RVA22U64	mandatory
RVA22S64	mandatory

2.0.0

State
ratified

Ratification date
2019-12

B.40.1. Synopsis

Architectural performance counters

B.40.2. CSRs

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

Name	Long Name	Address	Mode
cycle	Cycle counter for RDCYCLE Instruction	0xc00	U
cycleh	High-half cycle counter for RDCYCLE Instruction	0xc80	U
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U
instreth	Instructions retired counter, high bits	0xc82	U
mcycle	Machine Cycle Counter	0xb00	M
mcycleh	High-half machine Cycle Counter	0xb80	M

Name	Long Name	Address	Mode
minstret	Machine Instructions Retired Counter	0xb02	M
minstreth	Machine Instructions Retired Counter	0xb82	M
time	Timer for RDTIME Instruction	0xc01	U
timeh	High-half timer for RDTIME Instruction	0xc81	U

B.40.3. Parameters

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

TIME_CSR_IMPLEMENTED

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

Possible values:

true

[time/timeh](#) exists, and accessing it will not cause an IllegalInstruction trap

false

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

B.41. Extension Zifencei

Long Name: Instruction fence

Version Requirement: = 2.0

Zifencei Extension Presence

profile	v2.0.0
RVA22U64	-
RVA22S64	mandatory

2.0.0

State

ratified

B.41.1. Synopsis

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



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



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

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

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

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

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

operations.

i

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.41.2. Instructions

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

fence.i	Instruction fence

B.42. Extension Zihintpause

Long Name: PAUSE instruction
Version Requirement: = 2.0
Zihintpause Extension Presence

profile	v2.0.0
RVA22U64	mandatory
RVA22S64	mandatory

2.0.0
State
ratified

B.42.1. 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.

i

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.

i

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.

i

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.

i

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.43. Extension Zihpmp

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

profile	v2.0.0
RVA22U64	mandatory
RVA22S64	mandatory

2.0.0

State

ratified

B.43.1. Synopsis

Programmable hardware performance counters



The number of counters is platform-specific.

B.43.2. CSRs

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

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

B.44. Extension Zkn

Long Name: NIST Algorithm Suite

Version Requirement: ~> 1.0

Zkn Extension Presence

profile	v1.0.0
RVA22U64	optional
RVA22S64	optional

1.0.0

State

ratified

Implies
• Zbkb version 1.0.0 • Zbkc version 1.0.0 • Zbkx version 1.0.0 • Zkne version 1.0.0 • Zknd version 1.0.0 • Zknh version 1.0.0

B.44.1. Synopsis

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

- [Zbkb](#)
- [Zbkc](#)
- [Zbkx](#)
- [Zkne](#)
- [Zknd](#)
- [Zknh](#)

B.44.2. Instructions

The following 45 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

andn	AND with inverted operand
clmul	Carry-less multiply (low-part)
clmulh	Carry-less multiply (high-part)
orn	OR with inverted operand
rev8	Byte-reverse register (RV64 encoding)
rol	Rotate left (Register)
rolw	Rotate left word (Register)
ror	Rotate right (Register)
rori	Rotate right (Immediate)
roriw	Rotate right word (Immediate)
row	Rotate right word (Register)
xnor	Exclusive NOR
brev8	Reverse bits in bytes
pack	No synopsis available
packh	No synopsis available
packw	No synopsis available
unzip	Bit deinterleave
zip	Bit interleave
xperm4	Crossbar permutation (nibbles)
xperm8	Crossbar permutation (bytes)
aes64ks1i	AES Key Schedule Instruction 1
aes64ks2	AES Key Schedule Instruction 2
aes32dsi	AES final round decryption instruction for RV32
aes32dsmi	AES middle round decryption instruction for RV32
aes64ds	AES decrypt final round
aes64dsm	AES decrypt middle round
aes64im	AES Decrypt KeySchedule MixColumns
aes32esi	No synopsis available
aes32esmi	No synopsis available
aes64es	AES encrypt final round
aes64esm	AES encrypt middle round
sha256sig0	No synopsis available
sha256sig1	No synopsis available

sha256sum0	No synopsis available
sha256sum1	No synopsis available
sha512sig0	No synopsis available
sha512sig0h	No synopsis available
sha512sig0l	No synopsis available
sha512sig1	No synopsis available
sha512sig1h	No synopsis available
sha512sig1l	No synopsis available
sha512sum0	No synopsis available
sha512sum0r	No synopsis available
sha512sum1	No synopsis available
sha512sum1r	No synopsis available

B.45. Extension Zkr

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

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

1.0.0

State

ratified

B.45.1. 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.46. Extension Zks

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

profile	v1.0.0
RVA22U64	optional
RVA22S64	optional

1.0.0

State

ratified

Implies

- [Zbkb](#) version 1.0.0
- [Zbkc](#) version 1.0.0
- [Zbkx](#) version 1.0.0
- [Zkne](#) version 1.0.0
- [Zknd](#) version 1.0.0
- [Zknh](#) version 1.0.0

B.46.1. Synopsis

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

- [Zbkb](#)
- [Zbkc](#)
- [Zbkx](#)

- [Zksed](#)
- [Zksh](#)

B.46.2. Instructions

The following 49 instructions are added by extension version 1.0.0 (the minimum version of this extension that satisfies the extension requirement).

andn	AND with inverted operand
clmul	Carry-less multiply (low-part)
clmulh	Carry-less multiply (high-part)
orn	OR with inverted operand
rev8	Byte-reverse register (RV64 encoding)
rol	Rotate left (Register)
rolw	Rotate left word (Register)
ror	Rotate right (Register)
rori	Rotate right (Immediate)
roriw	Rotate right word (Immediate)
row	Rotate right word (Register)
xnor	Exclusive NOR
brev8	Reverse bits in bytes
pack	No synopsis available
packh	No synopsis available
packw	No synopsis available
unzip	Bit deinterleave
zip	Bit interleave
xperm4	Crossbar permutation (nibbles)
xperm8	Crossbar permutation (bytes)
aes64ks1i	AES Key Schedule Instruction 1
aes64ks2	AES Key Schedule Instruction 2
aes32dsi	AES final round decryption instruction for RV32
aes32dsmi	AES middle round decryption instruction for RV32
aes64ds	AES decrypt final round
aes64dsm	AES decrypt middle round
aes64im	AES Decrypt KeySchedule MixColumns
aes32esi	No synopsis available
aes32esmi	No synopsis available
aes64es	AES encrypt final round
aes64esm	AES encrypt middle round
sha256sig0	No synopsis available
sha256sig1	No synopsis available
sha256sum0	No synopsis available
sha256sum1	No synopsis available
sha512sig0	No synopsis available
sha512sig0h	No synopsis available
sha512sig0l	No synopsis available
sha512sig1	No synopsis available
sha512sig1h	No synopsis available
sha512sig1l	No synopsis available
sha512sum0	No synopsis available
sha512sum0r	No synopsis available
sha512sum1	No synopsis available
sha512sum1r	No synopsis available
sm3p0	No synopsis available
sm3p1	No synopsis available

sm4ed	No synopsis available
sm4ks	No synopsis available

B.47. 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

1.0.0

State

ratified

Ratification date

2021-11

1.0.1

State

ratified

Changes

- Fix typos to show that [c.srli](#), [c.srai](#), and [c.slli](#) are Zkt instructions in RV64.

B.47.1. 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'

RV32	RV64	Mnemonic	Instruction
✓	✓	slli <i>rd, rs1, imm</i>	'slli'
✓	✓	srli <i>rd, rs1, imm</i>	'srli'
✓	✓	srai <i>rd, rs1, imm</i>	'srai'
✓	✓	add <i>rd, rs1, rs2</i>	'add'
✓	✓	sub <i>rd, rs1, rs2</i>	'sub'
✓	✓	sll <i>rd, rs1, rs2</i>	'sll'
✓	✓	slt <i>rd, rs1, rs2</i>	'slt'
✓	✓	sltu <i>rd, rs1, rs2</i>	'sltu'
✓	✓	xor <i>rd, rs1, rs2</i>	'xor'
✓	✓	srl <i>rd, rs1, rs2</i>	'srl'
✓	✓	sra <i>rd, rs1, rs2</i>	'sra'
✓	✓	or <i>rd, rs1, rs2</i>	'or'
✓	✓	and <i>rd, rs1, rs2</i>	'and'
	✓	addiw <i>rd, rs1, imm</i>	'addiw'
	✓	slliw <i>rd, rs1, imm</i>	'slliw'
	✓	srliw <i>rd, rs1, imm</i>	'srliw'
	✓	sraiw <i>rd, rs1, imm</i>	'sraiw'
	✓	addw <i>rd, rs1, rs2</i>	'addw'
	✓	subw <i>rd, rs1, rs2</i>	'subw'
	✓	sllw <i>rd, rs1, rs2</i>	'sllw'
	✓	srlw <i>rd, rs1, rs2</i>	'srlw'
	✓	sraw <i>rd, rs1, rs2</i>	'sraw'

RVM (Multiply)

Multiplication is included; division and remaindering excluded.

RV32	RV64	Mnemonic	Instruction
✓	✓	mul <i>rd, rs1, rs2</i>	'mul'
✓	✓	mulh <i>rd, rs1, rs2</i>	'mulh'
✓	✓	mulhsu <i>rd, rs1, rs2</i>	'mulhsu'
✓	✓	mulhu <i>rd, rs1, rs2</i>	'mulhu'
	✓	mulw <i>rd, rs1, rs2</i>	'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'

RV32	RV64	Mnemonic	Instruction
✓	✓	pack	'pack-sc'
✓	✓	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.

DRAFT

Appendix C: Instruction Details

DRAFT

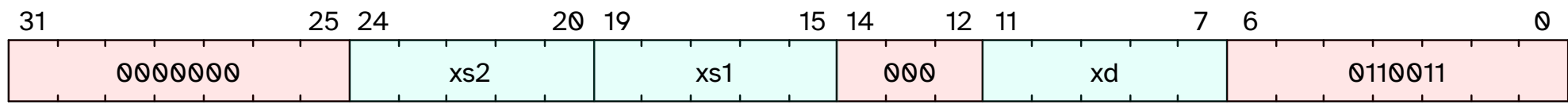
C.1. add

Integer add

This instruction is defined by:

- I, version >= I@2.1.0

C.1.1. Encoding



C.1.2. Description

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

C.1.3. Access

M	S	U
Always	Always	Always

C.1.4. Decode Variables

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

C.1.5. IDL Operation

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

C.1.6. Sail Operation

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

C.1.7. Exceptions

This instruction does not generate synchronous exceptions.

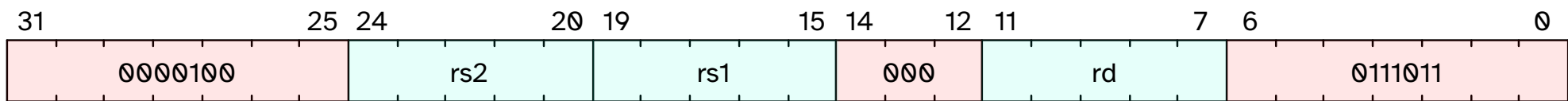
C.2. add.uw

Add unsigned word

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.2.1. Encoding



C.2.2. Description

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

C.2.3. Access

M	S	U
Always	Always	Always

C.2.4. Decode Variables

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

C.2.5. IDL Operation

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

C.2.6. Sail Operation

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

C.2.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

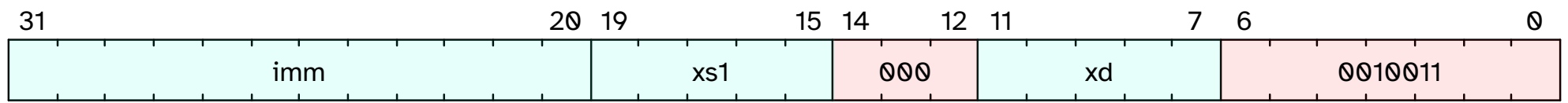
C.3. addi

Add immediate

This instruction is defined by:

- I, version >= I@2.1.0

C.3.1. Encoding



C.3.2. Description

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

C.3.3. Access

M	S	U
Always	Always	Always

C.3.4. Decode Variables

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

C.3.5. IDL Operation

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

C.3.6. Sail Operation

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

C.3.7. Exceptions

This instruction does not generate synchronous exceptions.

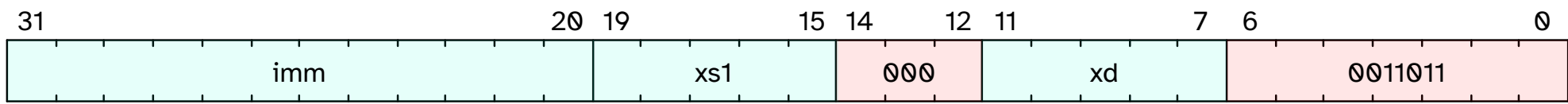
C.4. addiw

Add immediate word

This instruction is defined by:

- I, version >= I@2.1.0

C.4.1. Encoding



C.4.2. Description

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

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

C.4.6. Sail Operation

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

C.4.7. Exceptions

This instruction does not generate synchronous exceptions.

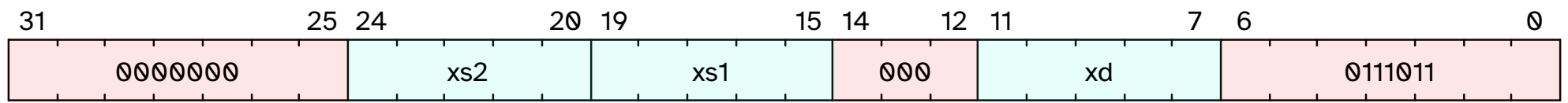
C.5. addw

Add word

This instruction is defined by:

- I, version >= I@2.1.0

C.5.1. Encoding



C.5.2. Description

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

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

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

C.5.6. Sail Operation

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

C.5.7. Exceptions

This instruction does not generate synchronous exceptions.

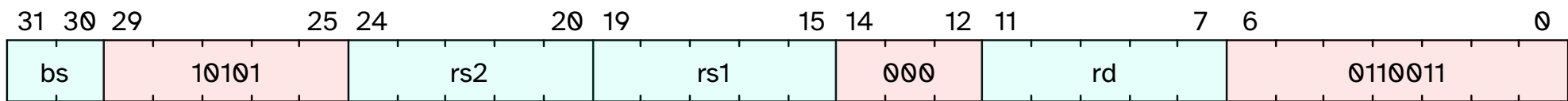
C.6. aes32dsi

AES final round decryption instruction for RV32

This instruction is defined by:

- Zknd, version >= Zknd@1.0.0

C.6.1. Encoding



C.6.2. Description

This instruction sources a single byte from rs2 according to bs. To this it applies the inverse AES SBox operation, and XOR’s the result with rs1. This instruction must always be implemented such that its execution latency does not depend on the data being operated on.

C.6.3. Access

M	S	U
Always	Always	Always

C.6.4. Decode Variables

```
Bits<2> bs = $encoding[31:30];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.6.5. IDL Operation

C.6.6. Exceptions

This instruction does not generate synchronous exceptions.

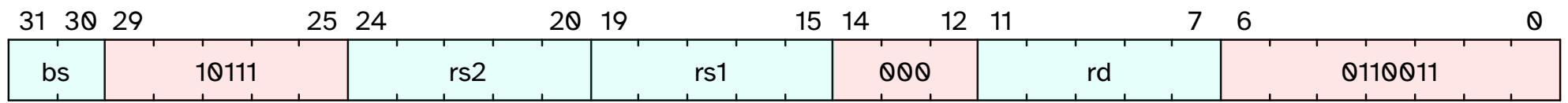
C.7. aes32dsmi

AES middle round decryption instruction for RV32

This instruction is defined by:

- Zknd, version >= Zknd@1.0.0

C.7.1. Encoding



C.7.2. Description

This instruction sources a single byte from rs2 according to bs. To this it applies the inverse AES SBox operation, and a partial inverse MixColumn, before XOR'ing the result with rs1. This instruction must always be implemented such that its execution latency does not depend on the data being operated on.

C.7.3. Access

M	S	U
Always	Always	Always

C.7.4. Decode Variables

```
Bits<2> bs = $encoding[31:30];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.7.5. IDL Operation

C.7.6. Exceptions

This instruction does not generate synchronous exceptions.

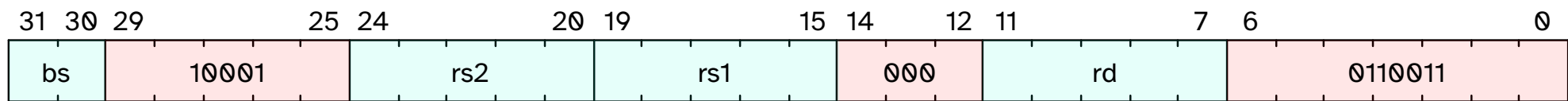
C.8. aes32esi

No synopsis available

This instruction is defined by:

- Zkne, version >= Zkne@1.0.0

C.8.1. Encoding



C.8.2. Description

No description available.

C.8.3. Access

M	S	U
Always	Always	Always

C.8.4. Decode Variables

```
Bits<2> bs = $encoding[31:30];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.8.5. IDL Operation

C.8.6. Exceptions

This instruction does not generate synchronous exceptions.

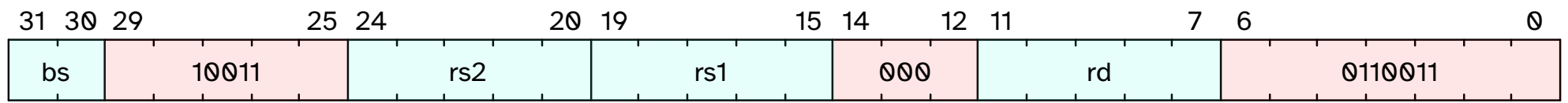
C.9. aes32esmi

No synopsis available

This instruction is defined by:

- Zkne, version >= Zkne@1.0.0

C.9.1. Encoding



C.9.2. Description

No description available.

C.9.3. Access

M	S	U
Always	Always	Always

C.9.4. Decode Variables

```
Bits<2> bs = $encoding[31:30];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.9.5. IDL Operation

C.9.6. Exceptions

This instruction does not generate synchronous exceptions.

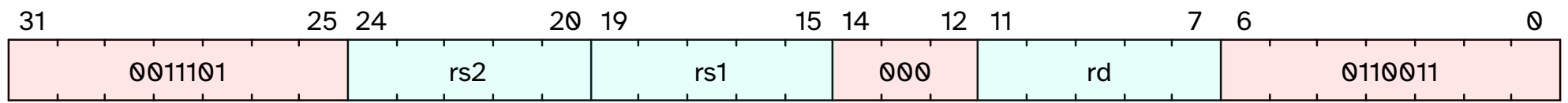
C.10. aes64ds

AES decrypt final round

This instruction is defined by:

- Zknd, version >= Zknd@1.0.0

C.10.1. Encoding



C.10.2. Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the Inverse ShiftRows and SubBytes steps.

C.10.3. Access

M	S	U
Always	Always	Always

C.10.4. Decode Variables

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

C.10.5. IDL Operation

C.10.6. Exceptions

This instruction does not generate synchronous exceptions.

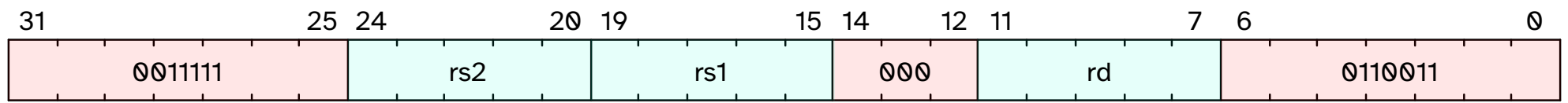
C.11. aes64dsm

AES decrypt middle round

This instruction is defined by:

- Zknd, version >= Zknd@1.0.0

C.11.1. Encoding



C.11.2. Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the Inverse ShiftRows, SubBytes and MixColumns steps.

C.11.3. Access

M	S	U
Always	Always	Always

C.11.4. Decode Variables

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

C.11.5. IDL Operation

C.11.6. Exceptions

This instruction does not generate synchronous exceptions.

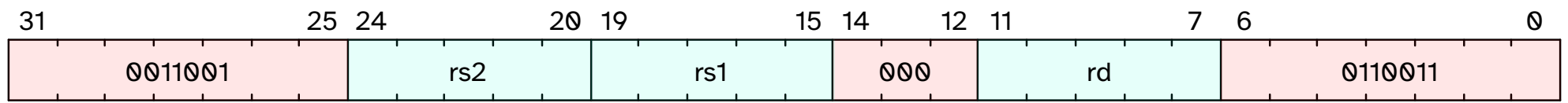
C.12. aes64es

AES encrypt final round

This instruction is defined by:

- Zkne, version >= Zkne@1.0.0

C.12.1. Encoding



C.12.2. Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the ShiftRows and SubBytes steps.

C.12.3. Access

M	S	U
Always	Always	Always

C.12.4. Decode Variables

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

C.12.5. IDL Operation

C.12.6. Exceptions

This instruction does not generate synchronous exceptions.

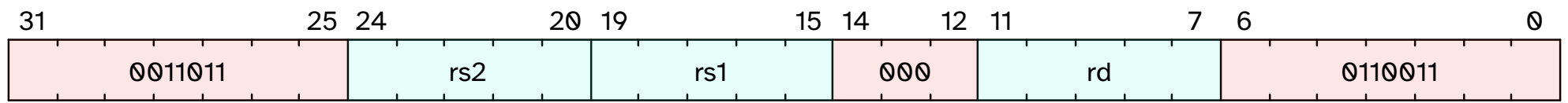
C.13. aes64esm

AES encrypt middle round

This instruction is defined by:

- Zkne, version >= Zkne@1.0.0

C.13.1. Encoding



C.13.2. Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the Inverse ShiftRows, SubBytes and MixColumns steps.

C.13.3. Access

M	S	U
Always	Always	Always

C.13.4. Decode Variables

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

C.13.5. IDL Operation

C.13.6. Exceptions

This instruction does not generate synchronous exceptions.

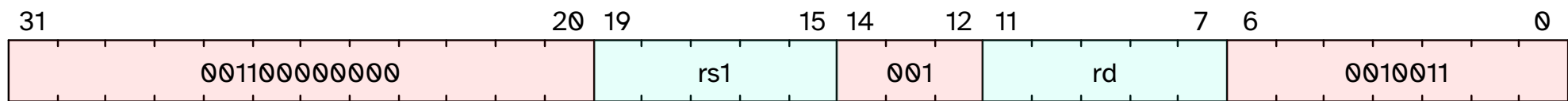
C.14. aes64im

AES Decrypt KeySchedule MixColumns

This instruction is defined by:

- Zknd, version >= Zknd@1.0.0

C.14.1. Encoding



C.14.2. Description

The instruction applies the inverse MixColumns transformation to two columns of the state array, packed into a single 64-bit register. It is used to create the inverse cipher KeySchedule, according to the equivalent inverse cipher construction in (NIST, 2001) (Page 23, Section 5.3.5).

C.14.3. Access

M	S	U
Always	Always	Always

C.14.4. Decode Variables

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

C.14.5. IDL Operation

C.14.6. Exceptions

This instruction does not generate synchronous exceptions.

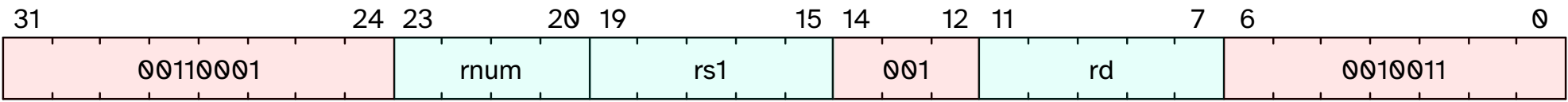
C.15. aes64ks1i

AES Key Schedule Instruction 1

This instruction is defined by:

- anyOf:
 - Zknd, version >= Zknd@1.0.0
 - Zkne, version >= Zkne@1.0.0

C.15.1. Encoding



C.15.2. Description

This instruction implements the rotation, SubBytes and Round Constant addition steps of the AES block cipher Key Schedule.

rnum must be in the range 0x0..0xA. The values 0xB..0xF are reserved.

C.15.3. Access

M	S	U
Always	Always	Always

C.15.4. Decode Variables

```
Bits<4> rnum = $encoding[23:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.15.5. IDL Operation

C.15.6. Exceptions

This instruction does not generate synchronous exceptions.

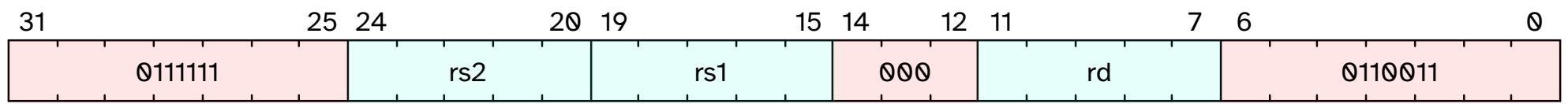
C.16. aes64ks2

AES Key Schedule Instruction 2

This instruction is defined by:

- anyOf:
 - Zknd, version >= Zknd@1.0.0
 - Zkne, version >= Zkne@1.0.0

C.16.1. Encoding



C.16.2. Description

This instruction implements the additional XOR’ing of key words as part of the AES block cipher Key Schedule.

C.16.3. Access

M	S	U
Always	Always	Always

C.16.4. Decode Variables

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

C.16.5. IDL Operation

C.16.6. Exceptions

This instruction does not generate synchronous exceptions.

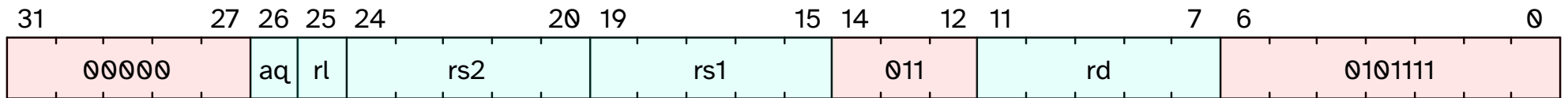
C.17. amoadd.d

Atomic fetch-and-add doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.17.1. Encoding



C.17.2. Description

Atomically:

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

C.17.3. Access

M	S	U
Always	Always	Always

C.17.4. Decode Variables

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

C.17.5. IDL Operation

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

C.17.6. Sail Operation

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



```

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

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

```

C.17.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

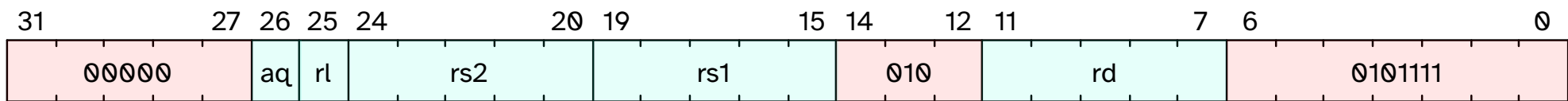
C.18. amoadd.w

Atomic fetch-and-add word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.18.1. Encoding



C.18.2. Description

Atomically:

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

C.18.3. Access

M	S	U
Always	Always	Always

C.18.4. Decode Variables

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

C.18.5. IDL Operation

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

C.18.6. Sail Operation

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

```

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

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

```

C.18.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

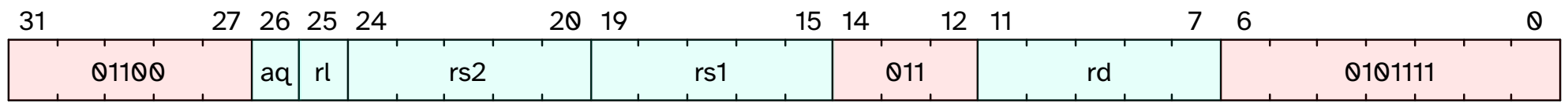
C.19. amoand.d

Atomic fetch-and-and doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.19.1. Encoding



C.19.2. Description

Atomically:

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

C.19.3. Access

M	S	U
Always	Always	Always

C.19.4. Decode Variables

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

C.19.5. IDL Operation

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

C.19.6. Sail Operation

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

```



```

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

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

```


C.19.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

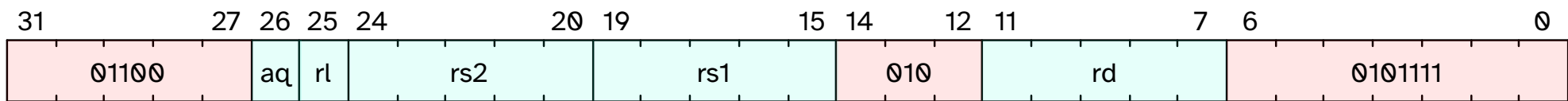
C.20. amoand.w

Atomic fetch-and-and word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.20.1. Encoding



C.20.2. Description

Atomically:

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

C.20.3. Access

M	S	U
Always	Always	Always

C.20.4. Decode Variables

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

C.20.5. IDL Operation

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

C.20.6. Sail Operation

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

```

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

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

```

C.20.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

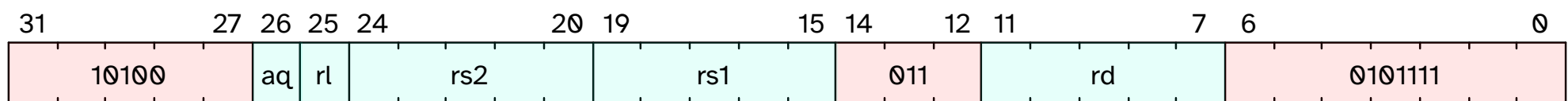
C.21. amomax.d

Atomic MAX doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.21.1. Encoding



C.21.2. Description

Atomically:

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

C.21.3. Access

M	S	U
Always	Always	Always

C.21.4. Decode Variables

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

C.21.5. IDL Operation

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

C.21.6. Sail Operation

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

```

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

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

```


C.21.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

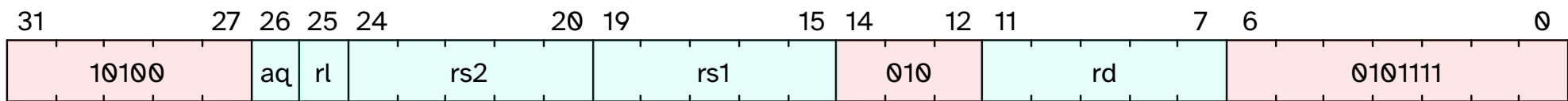
C.22. amomax.w

Atomic MAX word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.22.1. Encoding



C.22.2. Description

Atomically:

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

C.22.3. Access

M	S	U
Always	Always	Always

C.22.4. Decode Variables

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

C.22.5. IDL Operation

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

C.22.6. Sail Operation

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

```

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

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

```

C.22.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

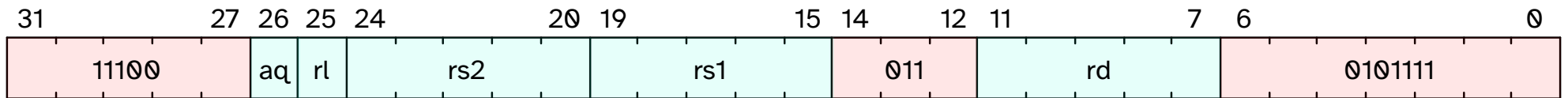
C.23. amomaxu.d

Atomic MAX unsigned doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.23.1. Encoding



C.23.2. Description

Atomically:

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

C.23.3. Access

M	S	U
Always	Always	Always

C.23.4. Decode Variables

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

C.23.5. IDL Operation

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

C.23.6. Sail Operation

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

```

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

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

```

C.23.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

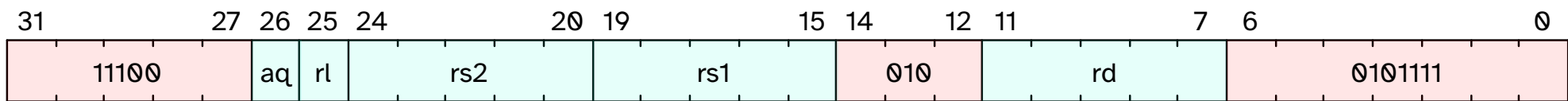
C.24. amomaxu.w

Atomic MAX unsigned word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.24.1. Encoding



C.24.2. Description

Atomically:

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

C.24.3. Access

M	S	U
Always	Always	Always

C.24.4. Decode Variables

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

C.24.5. IDL Operation

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

C.24.6. Sail Operation

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



```

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

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

```

C.24.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

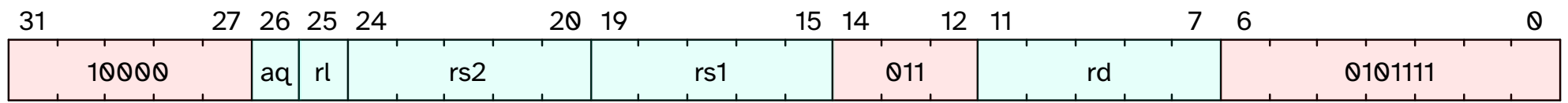
C.25. amomin.d

Atomic MIN doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.25.1. Encoding



C.25.2. Description

Atomically:

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

C.25.3. Access

M	S	U
Always	Always	Always

C.25.4. Decode Variables

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

C.25.5. IDL Operation

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

C.25.6. Sail Operation

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

```

```

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

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

```

C.25.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

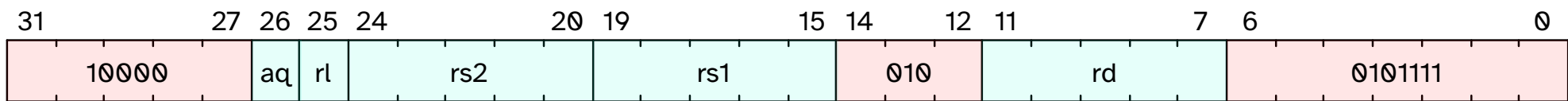
C.26. amomin.w

Atomic MIN word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.26.1. Encoding



C.26.2. Description

Atomically:

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

C.26.3. Access

M	S	U
Always	Always	Always

C.26.4. Decode Variables

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

C.26.5. IDL Operation

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

C.26.6. Sail Operation

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



```

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

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

```


C.26.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

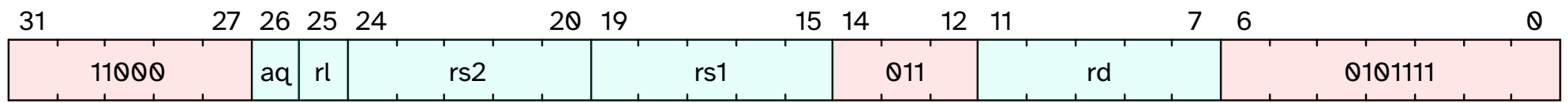
C.27. amominu.d

Atomic MIN unsigned doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.27.1. Encoding



C.27.2. Description

Atomically:

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

C.27.3. Access

M	S	U
Always	Always	Always

C.27.4. Decode Variables

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

C.27.5. IDL Operation

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

C.27.6. Sail Operation

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

```

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

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

```

C.27.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

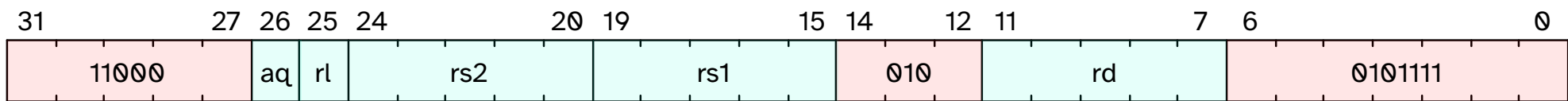
C.28. amominu.w

Atomic MIN unsigned word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.28.1. Encoding



C.28.2. Description

Atomically:

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

C.28.3. Access

M	S	U
Always	Always	Always

C.28.4. Decode Variables

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

C.28.5. IDL Operation

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

C.28.6. Sail Operation

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

```

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

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

```


C.28.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

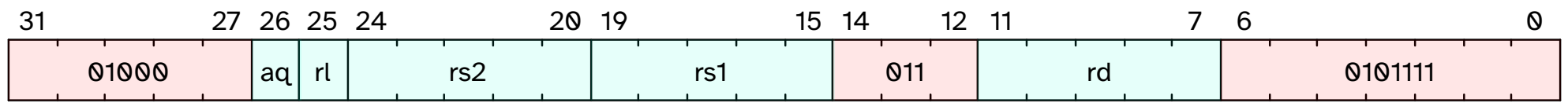
C.29. amoor.d

Atomic fetch-and-or doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.29.1. Encoding



C.29.2. Description

Atomically:

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

C.29.3. Access

M	S	U
Always	Always	Always

C.29.4. Decode Variables

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

C.29.5. IDL Operation

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

C.29.6. Sail Operation

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

```

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

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

```

C.29.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

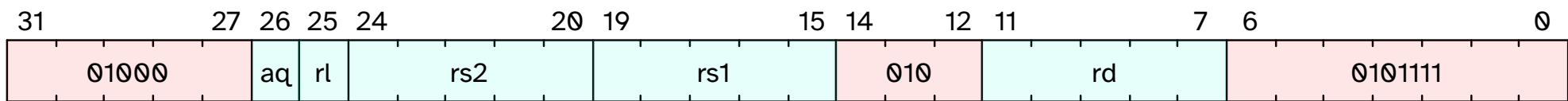
C.30. amoor.w

Atomic fetch-and-or word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.30.1. Encoding



C.30.2. Description

Atomically:

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

C.30.3. Access

M	S	U
Always	Always	Always

C.30.4. Decode Variables

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

C.30.5. IDL Operation

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

C.30.6. Sail Operation

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

```

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

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

```

C.30.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

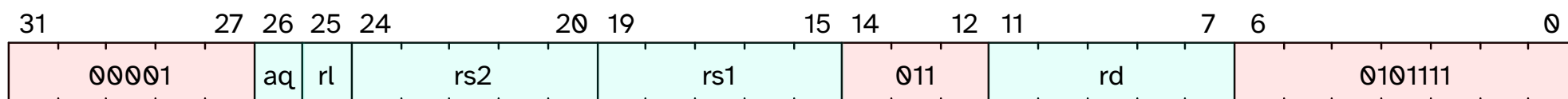
C.31. amoswap.d

Atomic SWAP doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.31.1. Encoding



C.31.2. Description

Atomically:

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

C.31.3. Access

M	S	U
Always	Always	Always

C.31.4. Decode Variables

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

C.31.5. IDL Operation

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

C.31.6. Sail Operation

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



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

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

C.31.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

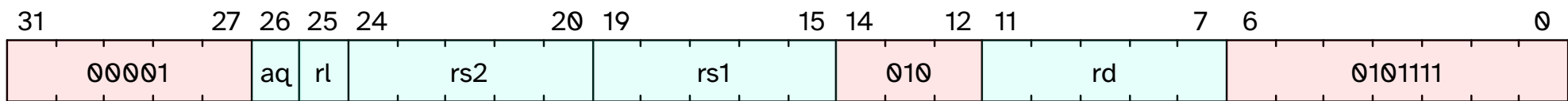
C.32. amoswap.w

Atomic SWAP word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.32.1. Encoding



C.32.2. Description

Atomically:

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

C.32.3. Access

M	S	U
Always	Always	Always

C.32.4. Decode Variables

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

C.32.5. IDL Operation

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

C.32.6. Sail Operation

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

```


This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

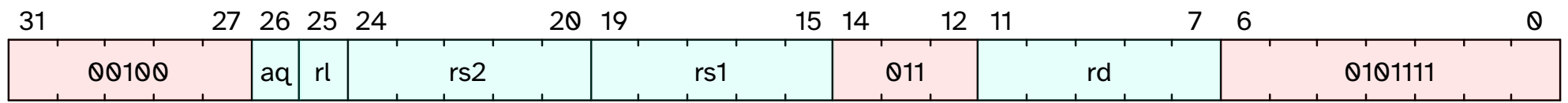
C.33. amoxor.d

Atomic fetch-and-xor doubleword

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.33.1. Encoding



C.33.2. Description

Atomically:

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

C.33.3. Access

M	S	U
Always	Always	Always

C.33.4. Decode Variables

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

C.33.5. IDL Operation

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

C.33.6. Sail Operation

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



```

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

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

```


C.33.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

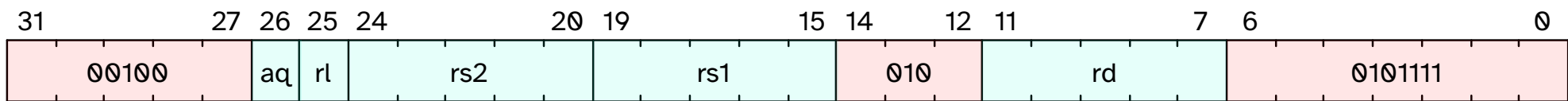
C.34. amoxor.w

Atomic fetch-and-xor word

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

C.34.1. Encoding



C.34.2. Description

Atomically:

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

C.34.3. Access

M	S	U
Always	Always	Always

C.34.4. Decode Variables

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

C.34.5. IDL Operation

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

C.34.6. Sail Operation

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

```

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

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

```

C.34.7. Exceptions

This instruction may result in the following synchronous exceptions:

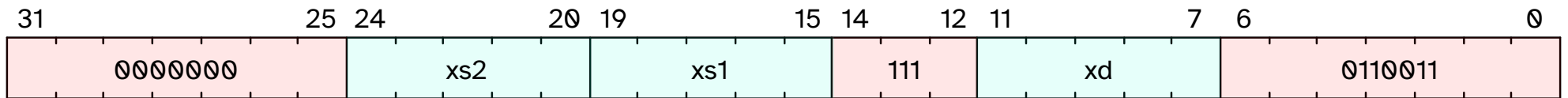
- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

C.35. and**And**

This instruction is defined by:

- I, version >= I@2.1.0

C.35.1. Encoding**C.35.2. Description**

And xs1 with xs2, and store the result in xd

C.35.3. Access

M	S	U
Always	Always	Always

C.35.4. Decode Variables

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

C.35.5. IDL Operation

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

C.35.6. 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.35.7. Exceptions

This instruction does not generate synchronous exceptions.

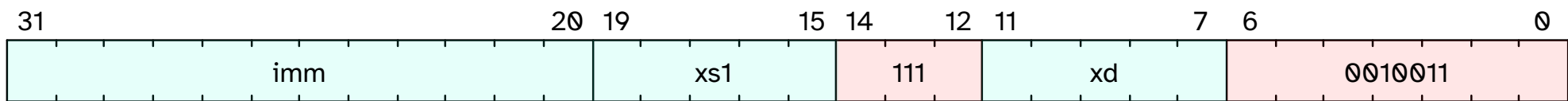
C.36. andi

And immediate

This instruction is defined by:

- I, version >= I@2.1.0

C.36.1. Encoding



C.36.2. Description

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

C.36.3. Access

M	S	U
Always	Always	Always

C.36.4. Decode Variables

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

C.36.5. IDL Operation

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

C.36.6. 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.36.7. Exceptions

This instruction does not generate synchronous exceptions.

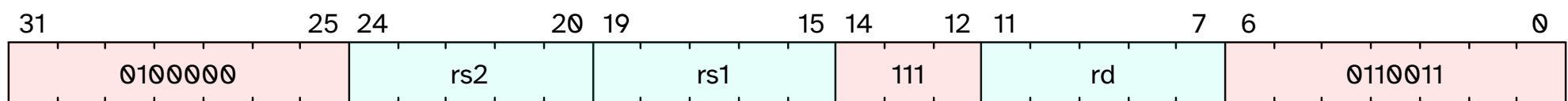
C.37. andn

AND with inverted operand

This instruction is defined by:

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

C.37.1. Encoding



C.37.2. Description

This instruction performs the bitwise logical AND operation between rs1 and the bitwise inversion of rs2.

C.37.3. Access

M	S	U
Always	Always	Always

C.37.4. Decode Variables

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

C.37.5. IDL Operation

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

C.37.6. 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_ROR  => 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.37.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

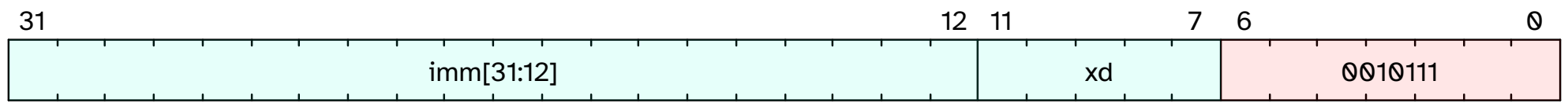
C.38. auipc

Add upper immediate to pc

This instruction is defined by:

- I, version >= I@2.1.0

C.38.1. Encoding



C.38.2. Description

Add an immediate to the current PC.

C.38.3. Access

M	S	U
Always	Always	Always

C.38.4. Decode Variables

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

C.38.5. IDL Operation

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

C.38.6. 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.38.7. Exceptions

This instruction does not generate synchronous exceptions.

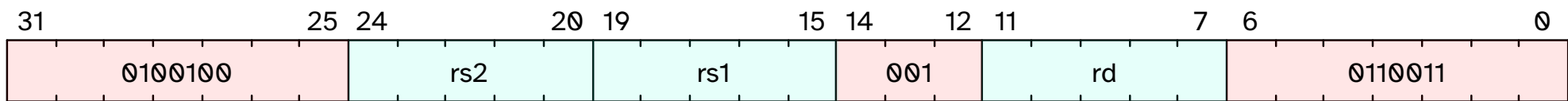
C.39. bclr

Single-Bit clear (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

C.39.1. Encoding



C.39.2. Description

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

C.39.3. Access

M	S	U
Always	Always	Always

C.39.4. Decode Variables

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

C.39.5. IDL Operation

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

C.39.6. 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.39.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.40. bclri

Single-Bit clear (Immediate)

This instruction is defined by:

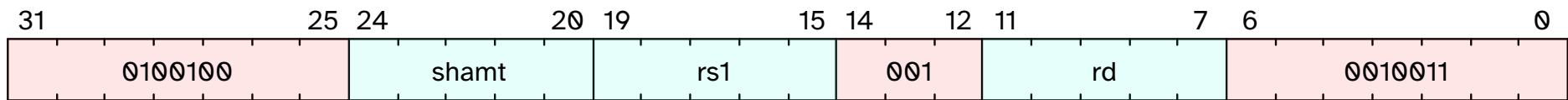
- Zbs, version $\geq$ Zbs@1.0.0

C.40.1. Encoding

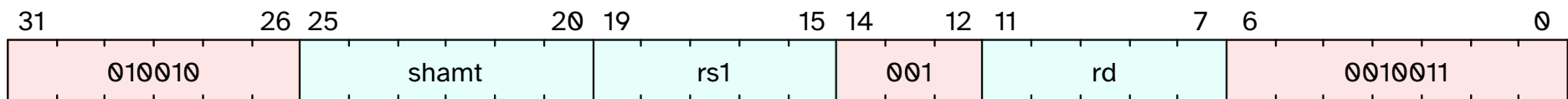


This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.40.2. Description

This instruction returns rs1 with a single bit cleared at the index specified in shamt. The index is read from the lower $\log_2(\text{XLEN})$ bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.40.3. Access

M	S	U
Always	Always	Always

C.40.4. Decode Variables

RV32

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

RV64

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

C.40.5. IDL Operation

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

C.40.6. 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 {
        RISC_V_BCLRI => rs1_val & ~(mask),
        RISC_V_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
    }
```

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

C.40.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

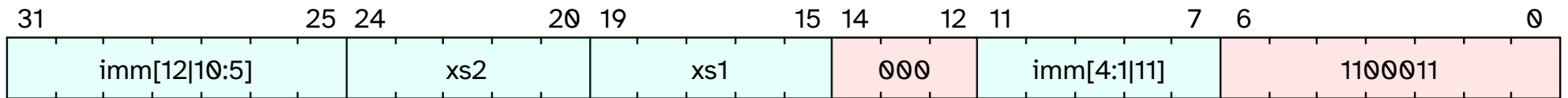
C.41. beq

Branch if equal

This instruction is defined by:

- I, version $\geq$ I@2.1.0

C.41.1. Encoding



C.41.2. 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.41.3. Access

M	S	U
Always	Always	Always

C.41.4. 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.41.5. IDL Operation

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

C.41.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISC_V_BEQ => xs1_val == xs2_val,
    RISC_V_BNE => xs1_val != xs2_val,
    RISC_V_BLT => xs1_val <_s xs2_val,
    RISC_V_BGE => xs1_val >=_s xs2_val,
    RISC_V_BLTU => xs1_val <_u xs2_val,
    RISC_V_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.41.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

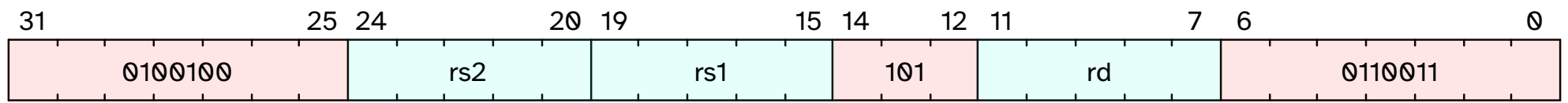
C.42. bext

Single-Bit extract (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

C.42.1. Encoding



C.42.2. Description

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

C.42.3. Access

M	S	U
Always	Always	Always

C.42.4. Decode Variables

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

C.42.5. IDL Operation

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

C.42.6. 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.42.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.43. bexti

Single-Bit extract (Immediate)

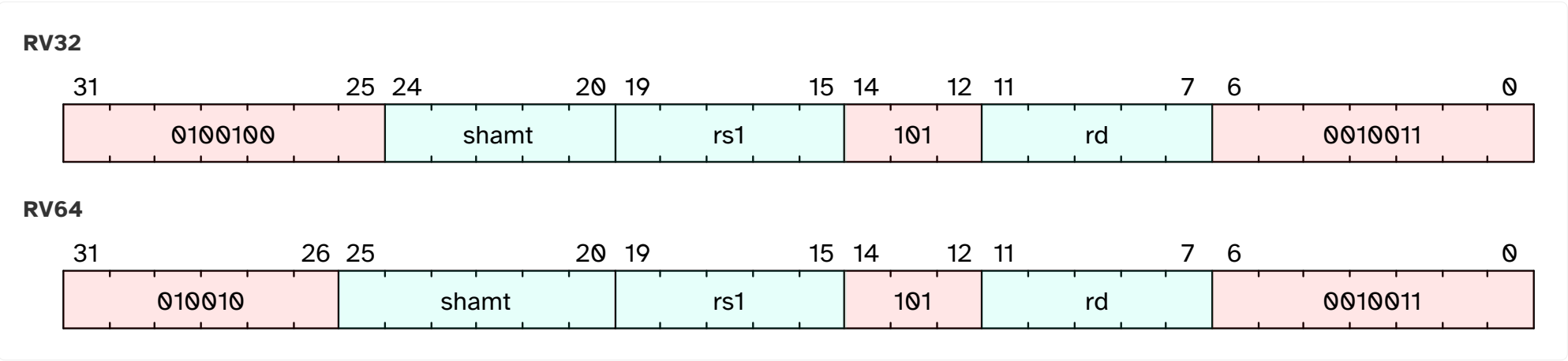
This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

C.43.1. Encoding



This instruction has different encodings in RV32 and RV64.



C.43.2. Description

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

C.43.3. Access

M	S	U
Always	Always	Always

C.43.4. Decode Variables

RV32

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

RV64

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

C.43.5. IDL Operation

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

C.43.6. 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 {
    RISC_V_BCLRI => rs1_val & ~(mask),
    RISC_V_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.43.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

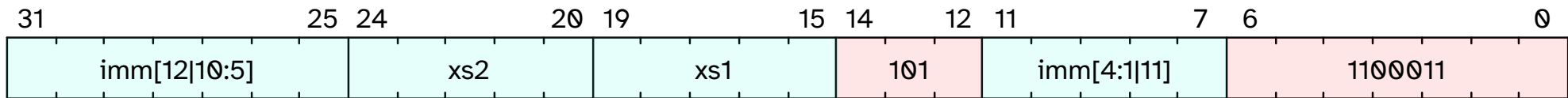
C.44. bge

Branch if greater than or equal

This instruction is defined by:

- I, version >= I@2.1.0

C.44.1. Encoding



C.44.2. 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.44.3. Access

M	S	U
Always	Always	Always

C.44.4. 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.44.5. IDL Operation

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

C.44.6. 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.44.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

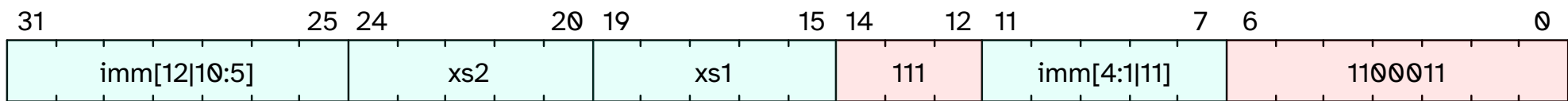
C.45. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

- I, version >= I@2.1.0

C.45.1. Encoding



C.45.2. 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 MisaignedAddress exception if PC + imm is misaligned.

C.45.3. Access

M	S	U
Always	Always	Always

C.45.4. 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.45.5. IDL Operation

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

C.45.6. 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.45.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

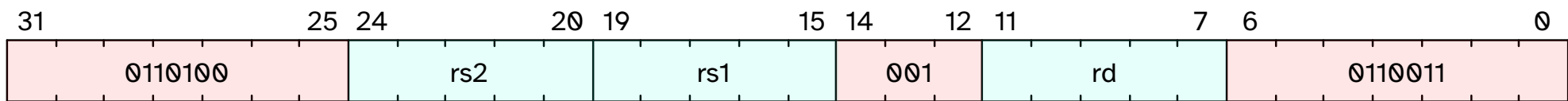
C.46. binv

Single-Bit invert (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

C.46.1. Encoding



C.46.2. Description

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

C.46.3. Access

M	S	U
Always	Always	Always

C.46.4. Decode Variables

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

C.46.5. IDL Operation

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

C.46.6. 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.46.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.47. binvi

Single-Bit invert (Immediate)

This instruction is defined by:

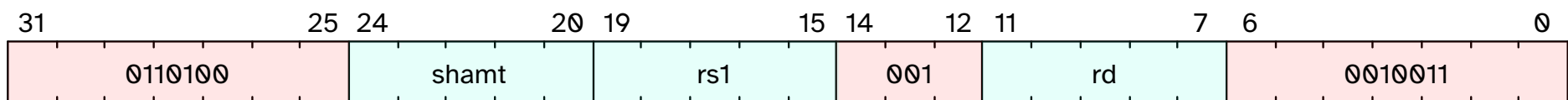
- Zbs, version $\geq$ Zbs@1.0.0

C.47.1. Encoding

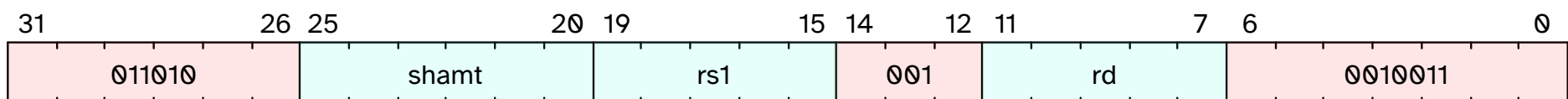


This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.47.2. Description

This instruction returns rs1 with a single bit inverted at the index specified in shamt. The index is read from the lower $\log_2(\text{XLEN})$ bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.47.3. Access

M	S	U
Always	Always	Always

C.47.4. Decode Variables

RV32

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

RV64

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

C.47.5. IDL Operation

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

C.47.6. 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 {
    RISC_V_BCLRI => rs1_val & ~(mask),
    RISC_V_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
  }
```



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

C.47.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

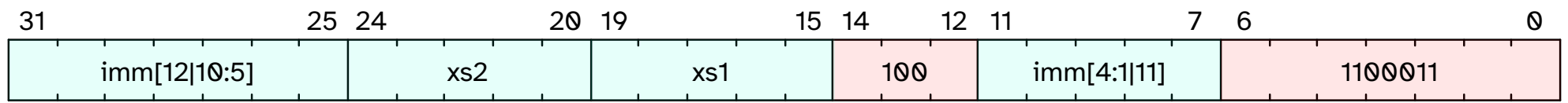
C.48. blt

Branch if less than

This instruction is defined by:

- I, version >= I@2.1.0

C.48.1. Encoding



C.48.2. 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.48.3. Access

M	S	U
Always	Always	Always

C.48.4. 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.48.5. IDL Operation

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

C.48.6. 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.48.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

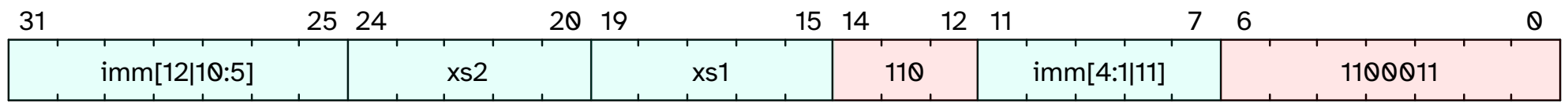
C.49. bltu

Branch if less than unsigned

This instruction is defined by:

- I, version >= I@2.1.0

C.49.1. Encoding



C.49.2. 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.49.3. Access

M	S	U
Always	Always	Always

C.49.4. 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.49.5. IDL Operation

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

C.49.6. 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.49.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

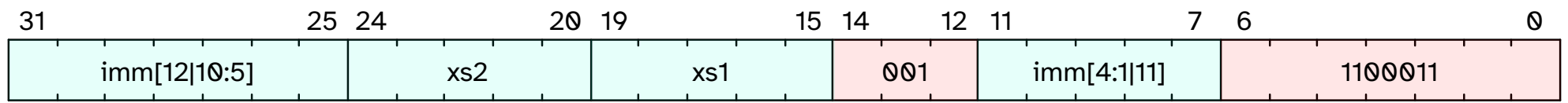
C.50. bne

Branch if not equal

This instruction is defined by:

- I, version >= I@2.1.0

C.50.1. Encoding



C.50.2. 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.50.3. Access

M	S	U
Always	Always	Always

C.50.4. 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.50.5. IDL Operation

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

C.50.6. 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.50.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

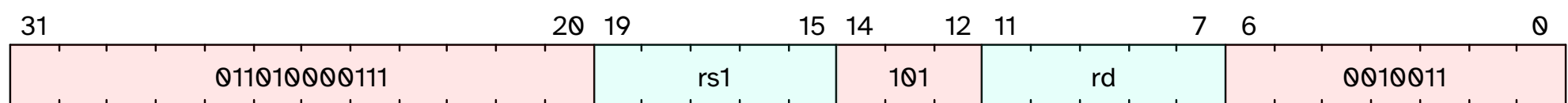
C.51. brev8

Reverse bits in bytes

This instruction is defined by:

- Zbkb, version $\geq$ Zbkb@1.0.0

C.51.1. Encoding



C.51.2. Description

This instruction reverses the order of the bits in every byte of a register.

C.51.3. Access

M	S	U
Always	Always	Always

C.51.4. Decode Variables

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

C.51.5. IDL Operation

```
XReg input = X[rs1];
XReg output = 0;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
    for (U32 j = 0; j < 8; j = j + 1) {
        output[(i * 8) + (7 - j)] = input[(i * 8) + j];
    }
}
X[rd] = output;
```

C.51.6. Sail Operation

```
result : xlenbits = EXTZ(0b0);
foreach (i from 0 to sizeof(xlen) by 8) {
    result[i+7..i] = reverse_bits_in_byte(X(rs1)[i+7..i]);
};
X(rd) = result;
```

C.51.7. Exceptions

This instruction does not generate synchronous exceptions.

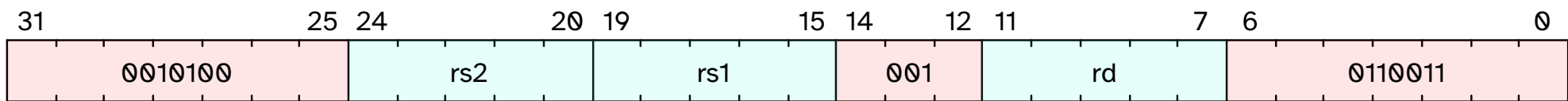
C.52. bset

Single-Bit set (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

C.52.1. Encoding



C.52.2. Description

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

C.52.3. Access

M	S	U
Always	Always	Always

C.52.4. Decode Variables

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

C.52.5. IDL Operation

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

C.52.6. 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.52.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.53. bseti

Single-Bit set (Immediate)

This instruction is defined by:

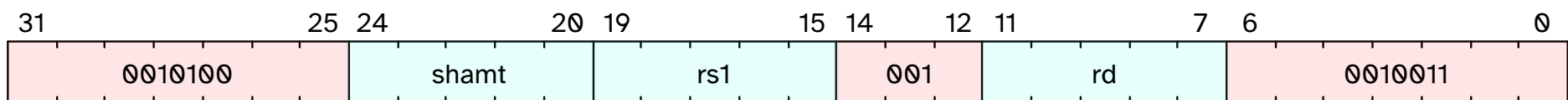
- Zbs, version $\geq$ Zbs@1.0.0

C.53.1. Encoding

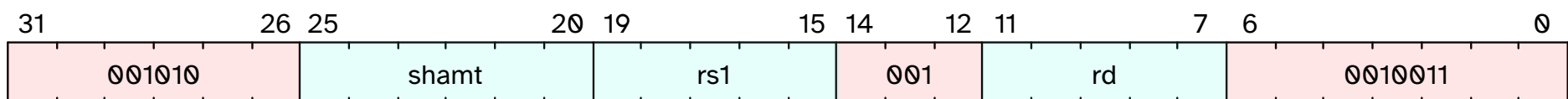


This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.53.2. Description

This instruction returns rs1 with a single bit set at the index specified in shamt. The index is read from the lower $\log_2(\text{XLEN})$ bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.53.3. Access

M	S	U
Always	Always	Always

C.53.4. Decode Variables

RV32

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

RV64

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

C.53.5. IDL Operation

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

C.53.6. 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 {
    RISC_V_BCLRI => rs1_val & ~(mask),
    RISC_V_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
  }
```

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

C.53.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

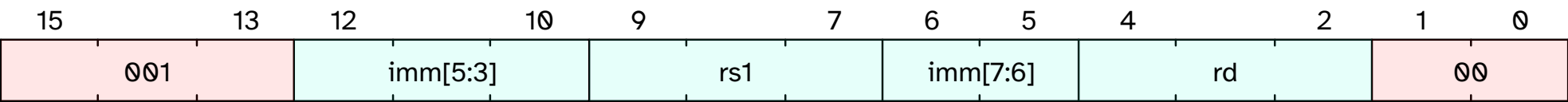
C.54. c.fld

Load double-precision

This instruction is defined by:

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

C.54.1. Encoding



C.54.2. Description

Loads a double precision floating-point value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1. It expands to fld rd, offset(rs1).

C.54.3. Access

M	S	U
Always	Always	Always

C.54.4. Decode Variables

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

C.54.5. IDL Operation

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

C.54.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

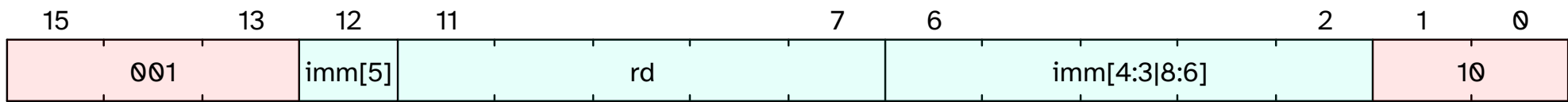
C.55. c.fldsp

Load doubleword into floating-point register from stack

This instruction is defined by:

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

C.55.1. Encoding



C.55.2. Description

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

C.55.3. Access

M	S	U
Always	Always	Always

C.55.4. Decode Variables

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

C.55.5. IDL Operation

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

C.55.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

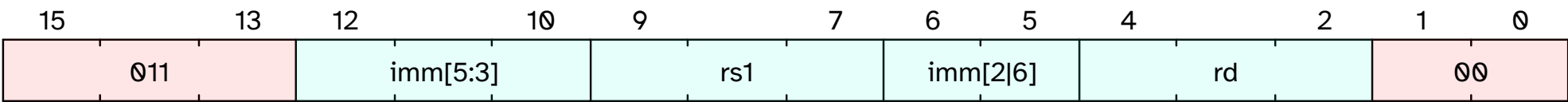
C.56. c.flw

Load single-precision

This instruction is defined by:

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

C.56.1. Encoding



C.56.2. Description

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

C.56.3. Access

M	S	U
Always	Always	Always

C.56.4. Decode Variables

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

C.56.5. IDL Operation

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

C.56.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

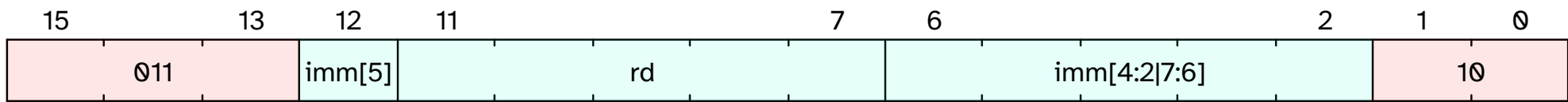
C.57. c.flwsp

Load word into floating-point register from stack

This instruction is defined by:

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

C.57.1. Encoding



C.57.2. Description

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

C.57.3. Access

M	S	U
Always	Always	Always

C.57.4. Decode Variables

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

C.57.5. IDL Operation

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

C.57.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

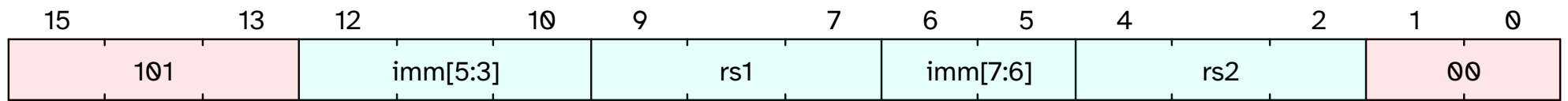
C.58. c.fsd

Store double-precision

This instruction is defined by:

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

C.58.1. Encoding



C.58.2. Description

Stores a double precision floating-point value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1. It expands to `fsd rs2, offset(rs1)`.

C.58.3. Access

M	S	U
Always	Always	Always

C.58.4. Decode Variables

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

C.58.5. IDL Operation

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

C.58.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

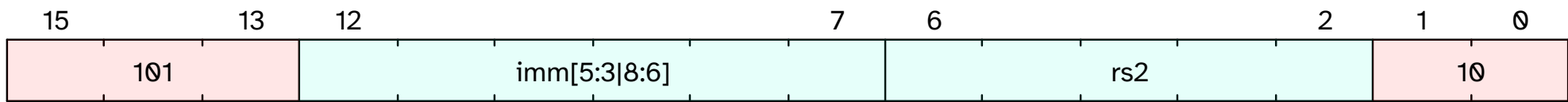
C.59. c.fsdsp

Store double-precision value to stack

This instruction is defined by:

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

C.59.1. Encoding



C.59.2. Description

Stores a double-precision floating-point value in floating-point register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to `fsd rs2, offset(x2)`.

C.59.3. Access

M	S	U
Always	Always	Always

C.59.4. Decode Variables

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

C.59.5. IDL Operation

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

C.59.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

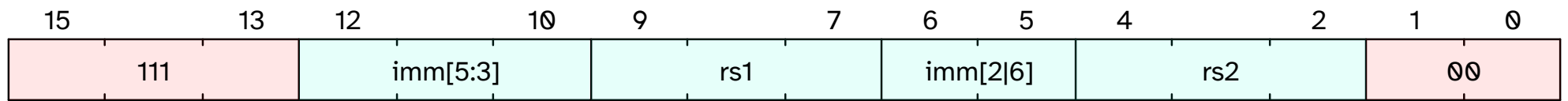
C.60. c.fsw

Store single-precision

This instruction is defined by:

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

C.60.1. Encoding



C.60.2. Description

Stores a single precision floating-point value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1. It expands to `fsw rs2, offset(rs1)`.

C.60.3. Access

M	S	U
Always	Always	Always

C.60.4. Decode Variables

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

C.60.5. IDL Operation

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

C.60.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

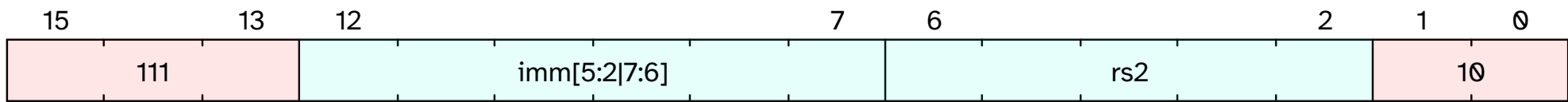
C.61. c.fswsp

Store single-precision value to stack

This instruction is defined by:

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

C.61.1. Encoding



C.61.2. Description

Stores a single-precision floating-point value in floating-point register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to `fsw rs2, offset(x2)`.

C.61.3. Access

M	S	U
Always	Always	Always

C.61.4. Decode Variables

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

C.61.5. IDL Operation

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

C.61.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

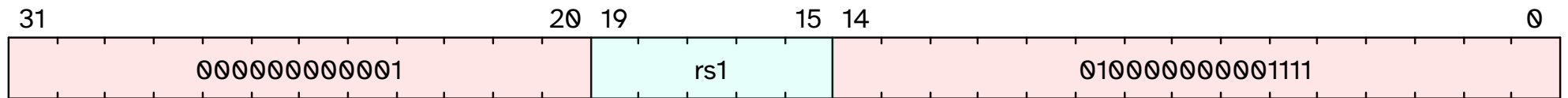
C.62. cbo.clean

Cache Block Clean

This instruction is defined by:

- Zicbom, version $\geq$ Zicbom@1.0.0

C.62.1. Encoding



C.62.2. 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 $\leq$ [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.62.3. 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.62.4. Decode Variables

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

C.62.5. Exceptions

This instruction does not generate synchronous exceptions.

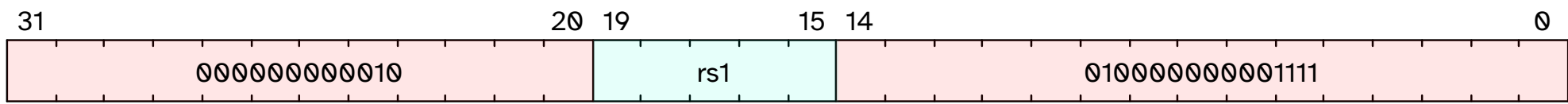
C.63. cbo.flush

Cache Block Flush

This instruction is defined by:

- Zicbom, version >= Zicbom@1.0.0

C.63.1. Encoding



C.63.2. 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.63.3. 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.63.4. Decode Variables

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

C.63.5. Exceptions

This instruction does not generate synchronous exceptions.

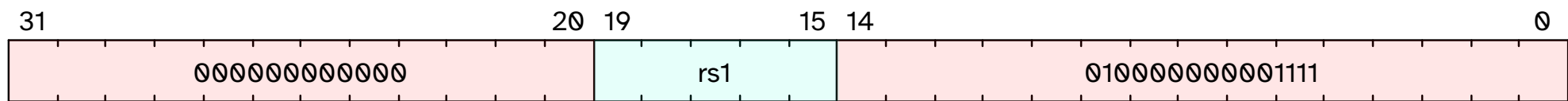
C.64. cbo.inval

Cache Block Invalidate

This instruction is defined by:

- Zicbom, version >= Zicbom@1.0.0

C.64.1. Encoding



C.64.2. 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	Virtual Instruction	Virtual 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/AM0 Guest-Page Fault` if virtual memory translation fails during G-stage translation.
<%- end -%>
* `Store/AM0 Page Fault` if virtual memory translation fails <% if ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AM0 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.64.3. 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	Virtual Instruction	Virtual 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.64.4. Decode Variables

Bits<5> rs1 = \$encoding[19:15];

C.64.5. Exceptions

This instruction does not generate synchronous exceptions.

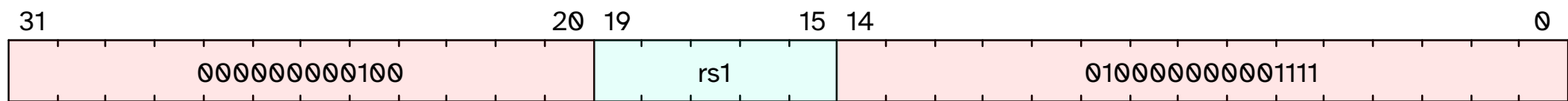
C.65. cbo.zero

Cache Block Zero

This instruction is defined by:

- Zicboz, version >= Zicboz@1.0.0

C.65.1. Encoding



C.65.2. 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/AM0 Guest-Page Fault` if virtual memory translation fails during 6-stage translation.
<%- end -%>
* `Store/AM0 Page Fault` if virtual memory translation fails <% if ext?(:H) %>when V=0 or during VS-stage translation<%
end %>
* `Store/AM0 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.65.3. 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.65.4. Decode Variables

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

C.65.5. IDL Operation

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



```

} else {
    XReg mask = CACHE_BLOCK_SIZE - 1;
    XReg cache_block_vaddr = X[rs1] & ~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.65.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoPageFault
- VirtualInstruction

DRAFT

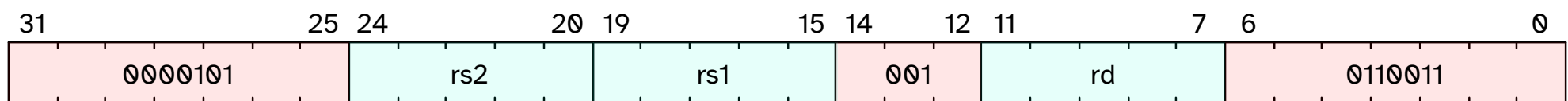
C.66. clmul

Carry-less multiply (low-part)

This instruction is defined by:

- anyOf:
 - Zbc, version >= Zbc@1.0.0
 - Zbkc, version >= Zbkc@1.0.0

C.66.1. Encoding



C.66.2. Description

`clmul` produces the lower half of the 2*XLEN carry-less product

C.66.3. Access

M	S	U
Always	Always	Always

C.66.4. Decode Variables

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

C.66.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg rs1_val = X[rs1];
XReg rs2_val = X[rs2];
XReg output = 0;
for (U32 i = 0; i < xlen(); i++) {
  output = (((rs2_val >> i) & 1) == 1) ? output ^ (rs1_val << i) : output;
}
X[rd] = output;
```

C.66.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  result : xlenbits = zeros();
  foreach (i from 0 to (xlen_val - 1))
    if rs2_val[i] == bitone then result = result ^ (rs1_val << i);
  X(rd) = result;
  RETIRE_SUCCESS
}
```

C.66.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

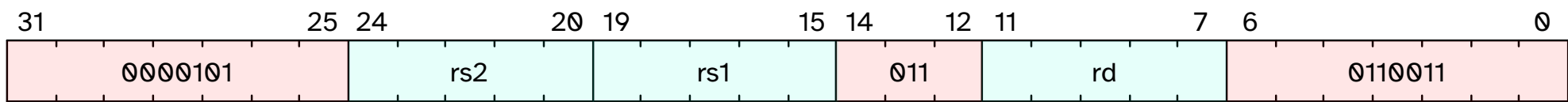
C.67. clmulh

Carry-less multiply (high-part)

This instruction is defined by:

- anyOf:
 - Zbc, version >= Zbc@1.0.0
 - Zbkc, version >= Zbkc@1.0.0

C.67.1. Encoding



C.67.2. Description

clmulh produces the upper half of the 2*XLEN carry-less product

C.67.3. Access

M	S	U
Always	Always	Always

C.67.4. Decode Variables

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

C.67.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg rs1_val = X[rs1];
XReg rs2_val = X[rs2];
XReg output = 0;
for (U32 i = 1; i <= xlen(); i++) {
  output = (rs2_val : output;
}
X[rd] = output;
```

C.67.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  result : xlenbits = zeros();
  foreach (i from 0 to (xlen_val - 1))
    if rs2_val[i] == bitone then result = result ^ (rs1_val >> (xlen_val - i));
  X(rd) = result;
  RETIRE_SUCCESS
}
```

C.67.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

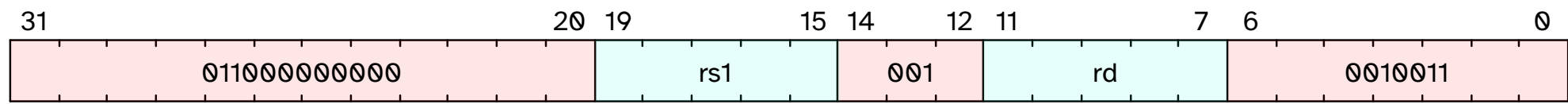
C.68. clz

Count leading zero bits

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.68.1. Encoding



C.68.2. Description

This instruction 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.68.3. Access

M	S	U
Always	Always	Always

C.68.4. Decode Variables

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

C.68.5. IDL Operation

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

C.68.6. 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.68.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

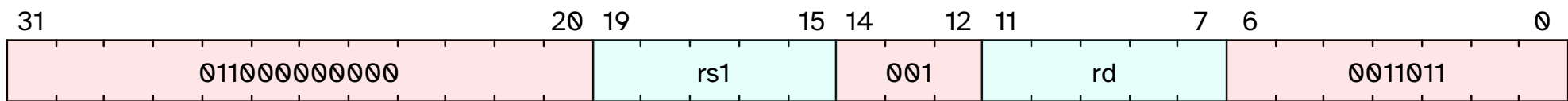
C.69. clzw

Count leading zero bits in word

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.69.1. Encoding



C.69.2. Description

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

C.69.3. Access

M	S	U
Always	Always	Always

C.69.4. Decode Variables

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

C.69.5. IDL Operation

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

C.69.6. Sail Operation

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

C.69.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

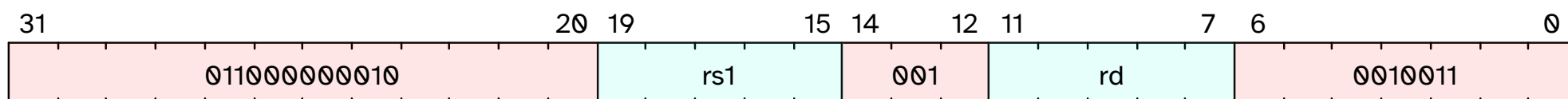
C.70. cpop

Count set bits

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.70.1. Encoding



C.70.2. Description

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

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

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

Listing 1. Software Hint

C.70.3. Access

M	S	U
Always	Always	Always

C.70.4. Decode Variables

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

C.70.5. IDL Operation

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

C.70.6. 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.70.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

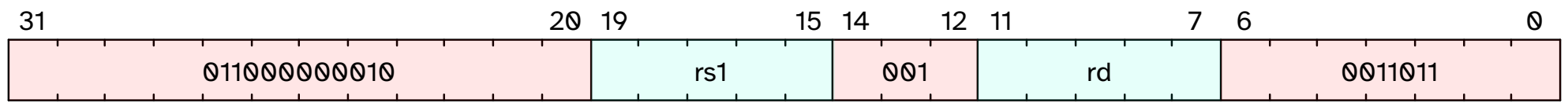
C.71. cpopw

Count set bits in word

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.71.1. Encoding



C.71.2. Description

This instructions counts the number of 1’s (i.e., set bits) in the least-significant word of the source register.

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

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

Listing 2. Software Hint

C.71.3. Access

M	S	U
Always	Always	Always

C.71.4. Decode Variables

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

C.71.5. IDL Operation

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

C.71.6. Sail Operation

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

C.71.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

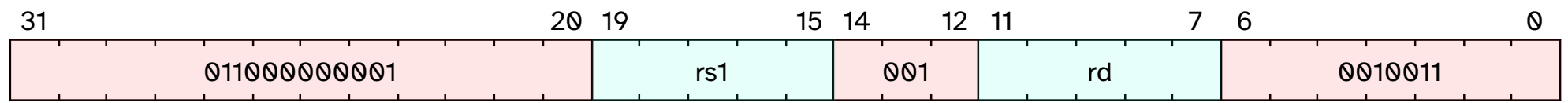
C.72. ctz

Count trailing zero bits

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.72.1. Encoding



C.72.2. Description

This instruction 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.72.3. Access

M	S	U
Always	Always	Always

C.72.4. Decode Variables

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

C.72.5. IDL Operation

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

C.72.6. 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.72.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

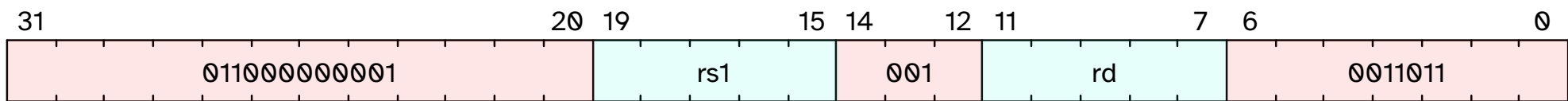
C.73. ctzw

Count trailing zero bits in word

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.73.1. Encoding



C.73.2. Description

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

C.73.3. Access

M	S	U
Always	Always	Always

C.73.4. Decode Variables

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

C.73.5. IDL Operation

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

C.73.6. Sail Operation

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

C.73.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

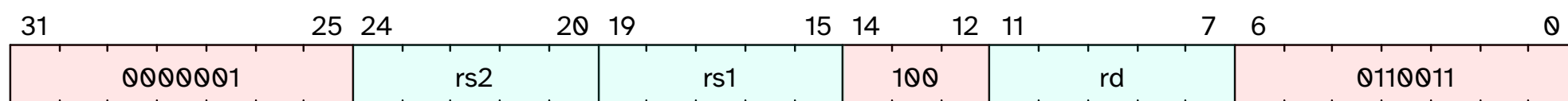
C.74. div

Signed division

This instruction is defined by:

- M, version $\geq$ M@2.0.0

C.74.1. Encoding



C.74.2. Description

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

Division by zero will put -1 into rd.

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

C.74.3. Access

M	S	U
Always	Always	Always

C.74.4. Decode Variables

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

C.74.5. IDL Operation

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

C.74.6. 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.74.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

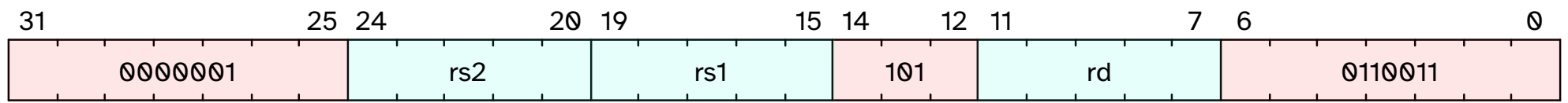
C.75. divu

Unsigned division

This instruction is defined by:

- M, version >= M@2.0.0

C.75.1. Encoding



C.75.2. Description

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

The remainder is discarded.

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

C.75.3. Access

M	S	U
Always	Always	Always

C.75.4. Decode Variables

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

C.75.5. IDL Operation

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

C.75.6. 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.75.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

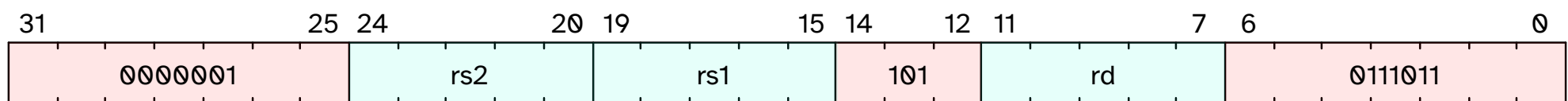
C.76. divuw

Unsigned 32-bit division

This instruction is defined by:

- M, version $\geq$ M@2.0.0

C.76.1. Encoding



C.76.2. Description

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

The remainder is discarded.

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

C.76.3. Access

M	S	U
Always	Always	Always

C.76.4. Decode Variables

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

C.76.5. IDL Operation

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

C.76.6. Sail Operation

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


C.76.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

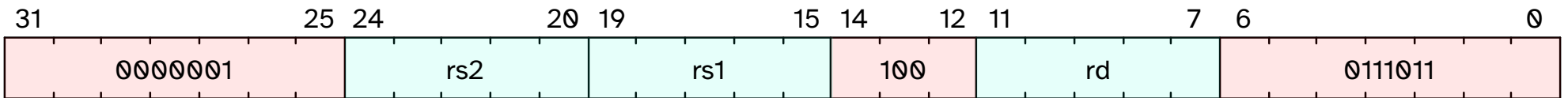
C.77. divw

Signed 32-bit division

This instruction is defined by:

- M, version $\geq$ M@2.0.0

C.77.1. Encoding



C.77.2. Description

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

The remainder is discarded.

Division by zero will put -1 into rd.

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

C.77.3. Access

M	S	U
Always	Always	Always

C.77.4. Decode Variables

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

C.77.5. IDL Operation

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

C.77.6. Sail Operation

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

```
}  
}
```

C.77.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

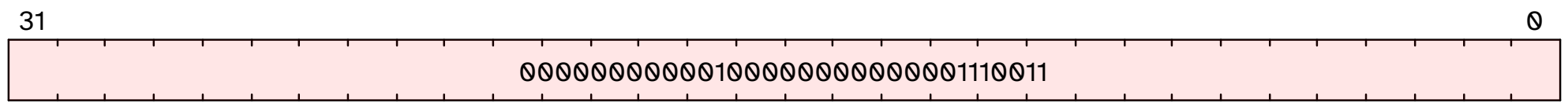
C.78. ebreak

Breakpoint exception

This instruction is defined by:

- I, version >= I@2.1.0

C.78.1. Encoding



C.78.2. 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.78.3. Access

M	S	U
Always	Always	Always

C.78.4. Decode Variables

C.78.5. IDL Operation

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

C.78.6. Sail Operation

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

C.78.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint


```

        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.79.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

DRAFT

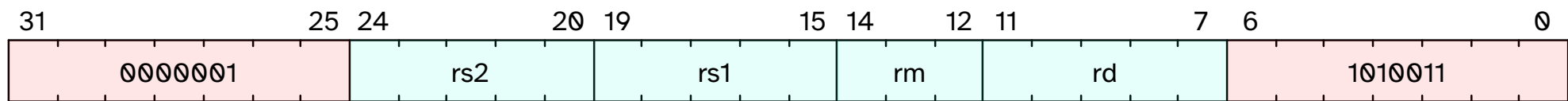
C.80. fadd.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.80.1. Encoding



C.80.2. Description

No description available.

C.80.3. Access

M	S	U
Always	Always	Always

C.80.4. Decode Variables

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

C.80.5. IDL Operation

C.80.6. Exceptions

This instruction does not generate synchronous exceptions.

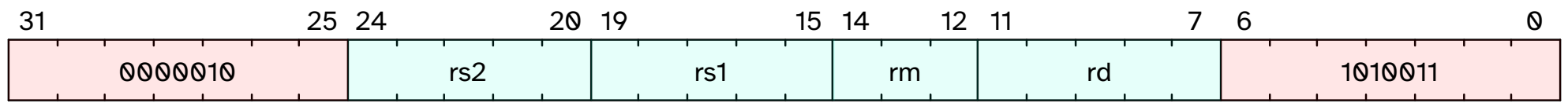
C.81. fadd.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.81.1. Encoding



C.81.2. Description

No description available.

C.81.3. Access

M	S	U
Always	Always	Always

C.81.4. Decode Variables

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

C.81.5. IDL Operation

C.81.6. Exceptions

This instruction does not generate synchronous exceptions.

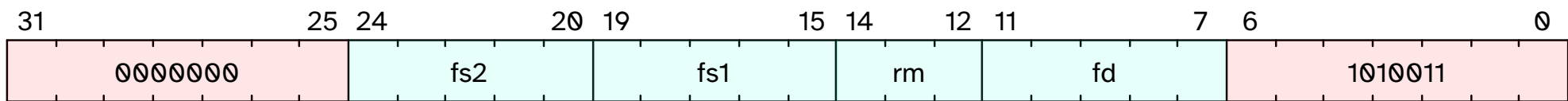
C.82. fadd.s

Single-precision floating-point addition

This instruction is defined by:

- F, version >= F@2.2.0

C.82.1. Encoding



C.82.2. Description

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

C.82.3. Access

M	S	U
Always	Always	Always

C.82.4. 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.82.5. IDL Operation

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

C.82.6. 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.82.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

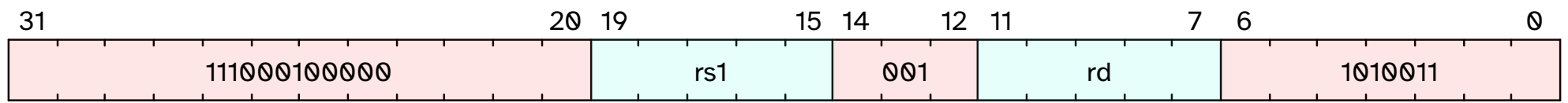
C.83. fclass.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.83.1. Encoding



C.83.2. Description

No description available.

C.83.3. Access

M	S	U
Always	Always	Always

C.83.4. Decode Variables

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

C.83.5. IDL Operation

C.83.6. Exceptions

This instruction does not generate synchronous exceptions.

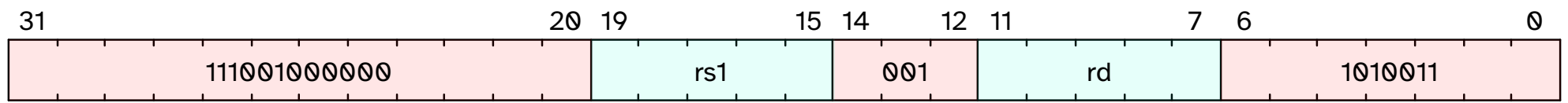
C.84. fclass.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.84.1. Encoding



C.84.2. Description

No description available.

C.84.3. Access

M	S	U
Always	Always	Always

C.84.4. Decode Variables

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

C.84.5. IDL Operation

C.84.6. Exceptions

This instruction does not generate synchronous exceptions.

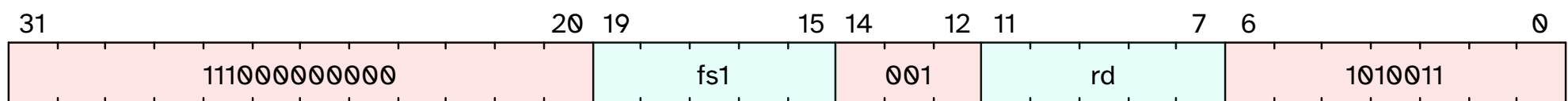
C.85. fclass.s

Single-precision floating-point classify

This instruction is defined by:

- F, version $\geq$ F@2.2.0

C.85.1. Encoding



C.85.2. Description

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

Table 10. Format of result of fclass instruction.

rd bit	Meaning
0	<i>rs1</i> is $-\infty$.
1	<i>rs1</i> is a negative normal number.
2	<i>rs1</i> is a negative subnormal number.
3	<i>rs1</i> is -0 .
4	<i>rs1</i> is $+0$.
5	<i>rs1</i> is a positive subnormal number.
6	<i>rs1</i> is a positive normal number.
7	<i>rs1</i> is $+\infty$.
8	<i>rs1</i> is a signaling NaN.
9	<i>rs1</i> is a quiet NaN.

C.85.3. Access

M	S	U
Always	Always	Always

C.85.4. Decode Variables

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

C.85.5. IDL Operation

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

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

C.85.6. 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.85.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

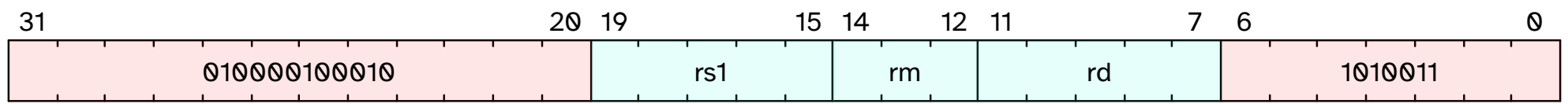
C.86. fcvt.d.h

No synopsis available

This instruction is defined by:

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

C.86.1. Encoding



C.86.2. Description

No description available.

C.86.3. Access

M	S	U
Always	Always	Always

C.86.4. Decode Variables

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

C.86.5. IDL Operation

C.86.6. Exceptions

This instruction does not generate synchronous exceptions.

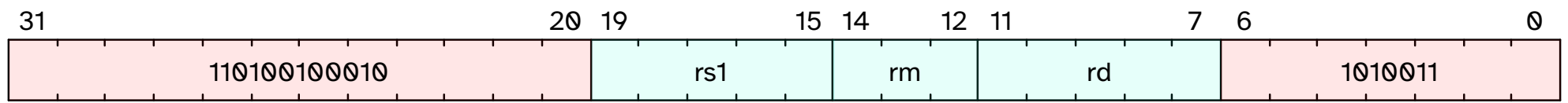
C.87. fcvt.d.l

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.87.1. Encoding



C.87.2. Description

No description available.

C.87.3. Access

M	S	U
Always	Always	Always

C.87.4. Decode Variables

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

C.87.5. IDL Operation

C.87.6. Exceptions

This instruction does not generate synchronous exceptions.

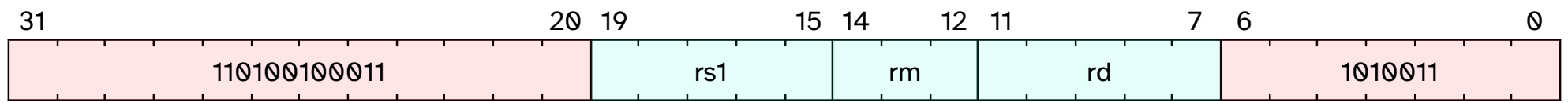
C.88. fcvt.d.lu

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.88.1. Encoding



C.88.2. Description

No description available.

C.88.3. Access

M	S	U
Always	Always	Always

C.88.4. Decode Variables

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

C.88.5. IDL Operation

C.88.6. Exceptions

This instruction does not generate synchronous exceptions.

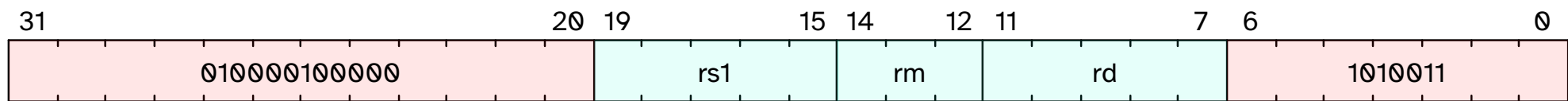
C.89. fcvt.d.s

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.89.1. Encoding



C.89.2. Description

No description available.

C.89.3. Access

M	S	U
Always	Always	Always

C.89.4. Decode Variables

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

C.89.5. IDL Operation

C.89.6. Exceptions

This instruction does not generate synchronous exceptions.

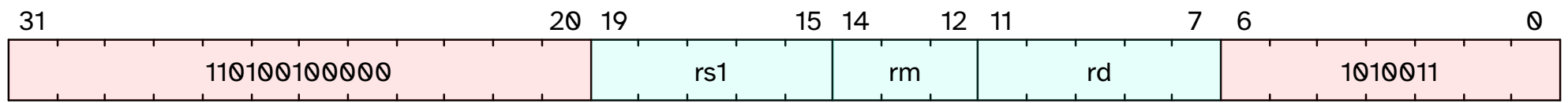
C.90. fcvt.d.w

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.90.1. Encoding



C.90.2. Description

No description available.

C.90.3. Access

M	S	U
Always	Always	Always

C.90.4. Decode Variables

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

C.90.5. IDL Operation

C.90.6. Exceptions

This instruction does not generate synchronous exceptions.

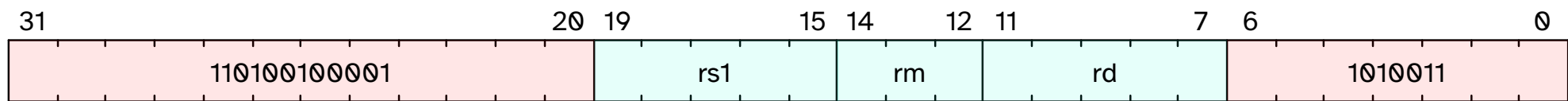
C.91. fcvt.d.wu

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.91.1. Encoding



C.91.2. Description

No description available.

C.91.3. Access

M	S	U
Always	Always	Always

C.91.4. Decode Variables

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

C.91.5. IDL Operation

C.91.6. Exceptions

This instruction does not generate synchronous exceptions.

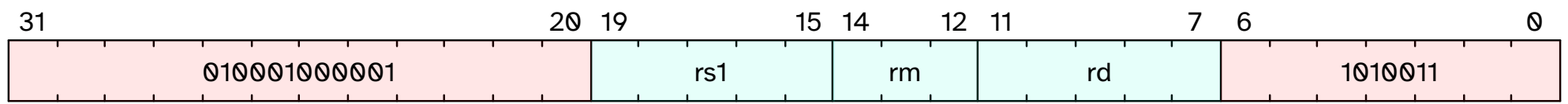
C.92. fcvt.h.d

No synopsis available

This instruction is defined by:

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

C.92.1. Encoding



C.92.2. Description

No description available.

C.92.3. Access

M	S	U
Always	Always	Always

C.92.4. Decode Variables

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

C.92.5. IDL Operation

C.92.6. Exceptions

This instruction does not generate synchronous exceptions.

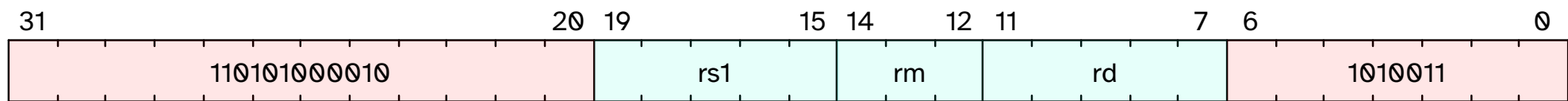
C.93. fcvt.h.l

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.93.1. Encoding



C.93.2. Description

No description available.

C.93.3. Access

M	S	U
Always	Always	Always

C.93.4. Decode Variables

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

C.93.5. IDL Operation

C.93.6. Exceptions

This instruction does not generate synchronous exceptions.

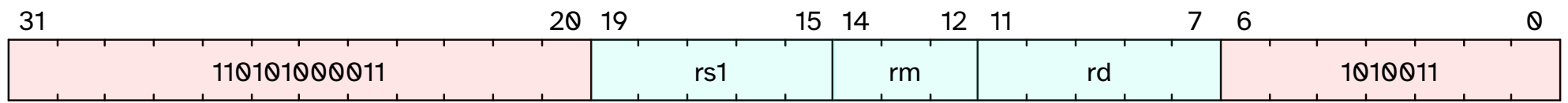
C.94. fcvt.h.lu

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.94.1. Encoding



C.94.2. Description

No description available.

C.94.3. Access

M	S	U
Always	Always	Always

C.94.4. Decode Variables

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

C.94.5. IDL Operation

C.94.6. Exceptions

This instruction does not generate synchronous exceptions.

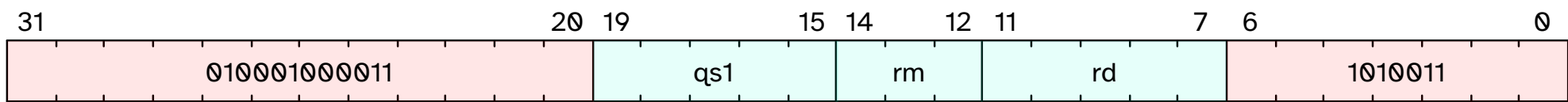
C.95. fcvt.h.q

No synopsis available

This instruction is defined by:

- allOf:
 - Q, version >= Q@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.95.1. Encoding



C.95.2. Description

No description available.

C.95.3. Access

M	S	U
Always	Always	Always

C.95.4. Decode Variables

```
Bits<5> qs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

C.95.5. IDL Operation

C.95.6. Exceptions

This instruction does not generate synchronous exceptions.

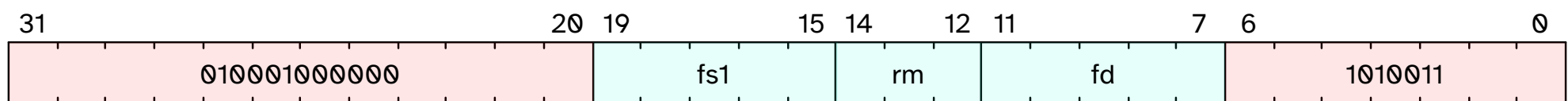
C.96. fcvt.h.s

Convert half-precision float to a single-precision float

This instruction is defined by:

- allOf:
 - Zfh, version $\geq$ Zfh@1.0.0
 - Zfhmin, version $\geq$ Zfhmin@1.0.0

C.96.1. Encoding



C.96.2. 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.96.3. Access

M	S	U
Always	Always	Always

C.96.4. Decode Variables

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

C.96.5. 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.96.6. 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.96.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

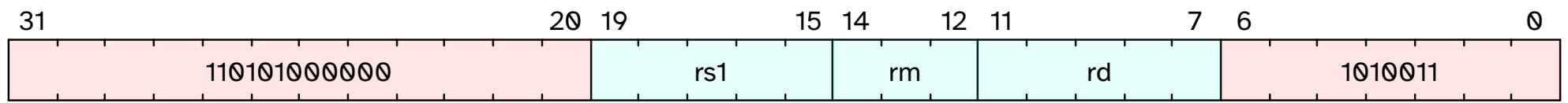
C.97. fcvt.h.w

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.97.1. Encoding



C.97.2. Description

No description available.

C.97.3. Access

M	S	U
Always	Always	Always

C.97.4. Decode Variables

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

C.97.5. IDL Operation

C.97.6. Exceptions

This instruction does not generate synchronous exceptions.

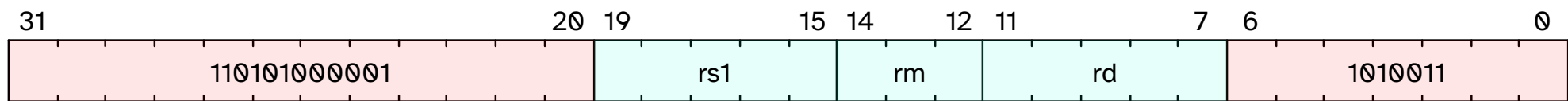
C.98. fcvt.h.wu

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.98.1. Encoding



C.98.2. Description

No description available.

C.98.3. Access

M	S	U
Always	Always	Always

C.98.4. Decode Variables

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

C.98.5. IDL Operation

C.98.6. Exceptions

This instruction does not generate synchronous exceptions.

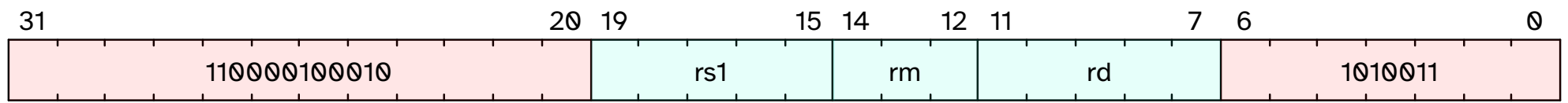
C.99. fcvt.l.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.99.1. Encoding



C.99.2. Description

No description available.

C.99.3. Access

M	S	U
Always	Always	Always

C.99.4. Decode Variables

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

C.99.5. IDL Operation

C.99.6. Exceptions

This instruction does not generate synchronous exceptions.

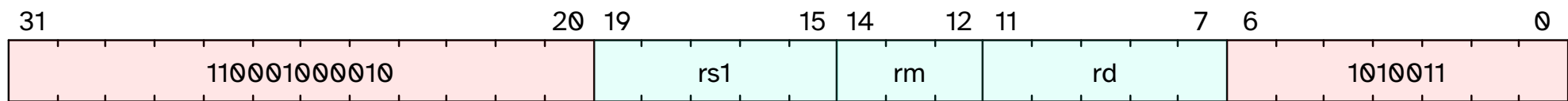
C.100. fcvt.Lh

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.100.1. Encoding



C.100.2. Description

No description available.

C.100.3. Access

M	S	U
Always	Always	Always

C.100.4. Decode Variables

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

C.100.5. IDL Operation

C.100.6. Exceptions

This instruction does not generate synchronous exceptions.

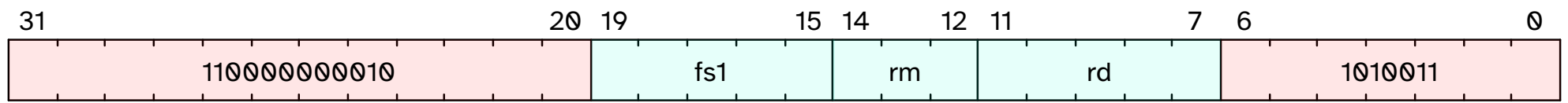
C.101. fcvt.ls

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

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

C.101.5. IDL Operation

C.101.6. 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.101.7. Exceptions

This instruction does not generate synchronous exceptions.

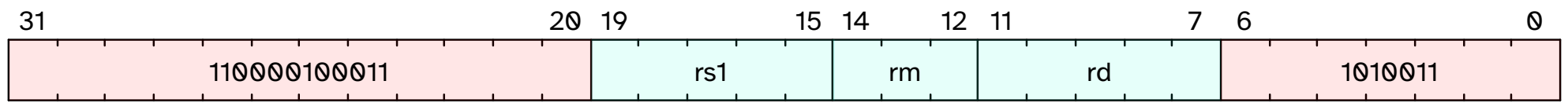
C.102. fcvt.lu.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.102.1. Encoding



C.102.2. Description

No description available.

C.102.3. Access

M	S	U
Always	Always	Always

C.102.4. Decode Variables

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

C.102.5. IDL Operation

C.102.6. Exceptions

This instruction does not generate synchronous exceptions.

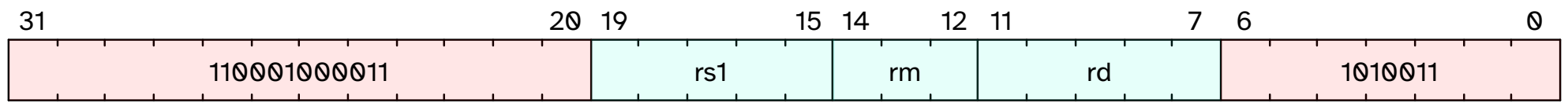
C.103. fcvt.lu.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.103.1. Encoding



C.103.2. Description

No description available.

C.103.3. Access

M	S	U
Always	Always	Always

C.103.4. Decode Variables

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

C.103.5. IDL Operation

C.103.6. Exceptions

This instruction does not generate synchronous exceptions.

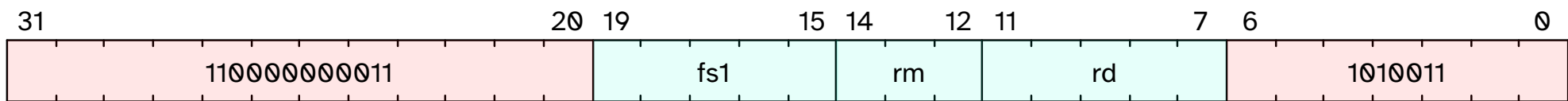
C.104. fcvt.lu.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.104.1. Encoding



C.104.2. Description

No description available.

C.104.3. Access

M	S	U
Always	Always	Always

C.104.4. Decode Variables

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

C.104.5. IDL Operation

C.104.6. 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.104.7. Exceptions

This instruction does not generate synchronous exceptions.

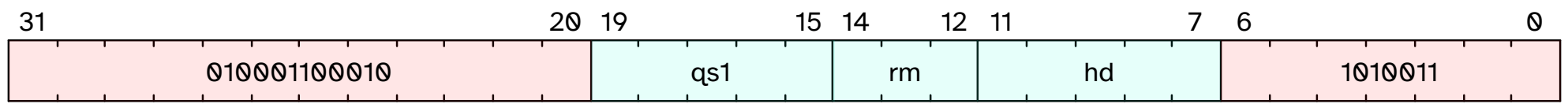
C.105. fcvt.q.h

No synopsis available

This instruction is defined by:

- allOf:
 - Q, version >= Q@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.105.1. Encoding



C.105.2. Description

No description available.

C.105.3. Access

M	S	U
Always	Always	Always

C.105.4. Decode Variables

```
Bits<5> qs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> hd = $encoding[11:7];
```

C.105.5. IDL Operation

C.105.6. Exceptions

This instruction does not generate synchronous exceptions.

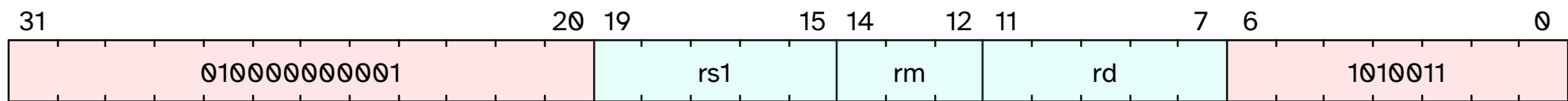
C.106. fcvt.s.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.106.1. Encoding



C.106.2. Description

No description available.

C.106.3. Access

M	S	U
Always	Always	Always

C.106.4. Decode Variables

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

C.106.5. IDL Operation

C.106.6. Exceptions

This instruction does not generate synchronous exceptions.

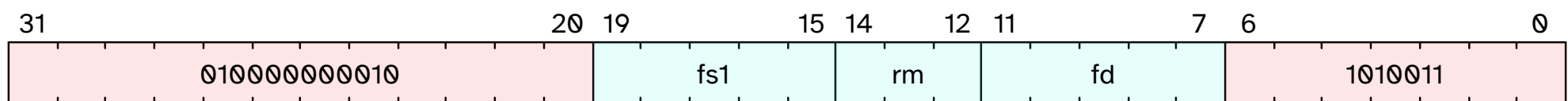
C.107. fcvt.s.h

Convert single-precision float to a half-precision float

This instruction is defined by:

- allOf:
 - Zfh, version $\geq$ Zfh@1.0.0
 - Zfhmin, version $\geq$ Zfhmin@1.0.0

C.107.1. Encoding



C.107.2. 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.107.3. Access

M	S	U
Always	Always	Always

C.107.4. Decode Variables

```

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

```

C.107.5. 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.107.6. 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.107.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

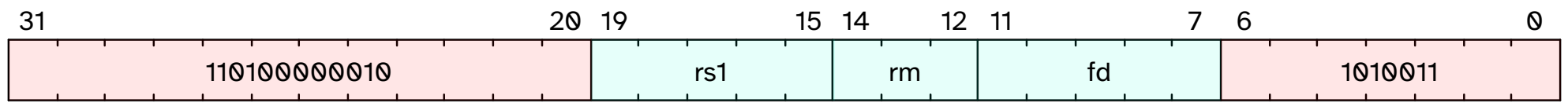
C.108. fcvt.s.l

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.108.1. Encoding



C.108.2. Description

No description available.

C.108.3. Access

M	S	U
Always	Always	Always

C.108.4. Decode Variables

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

C.108.5. IDL Operation

C.108.6. 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.108.7. Exceptions

This instruction does not generate synchronous exceptions.

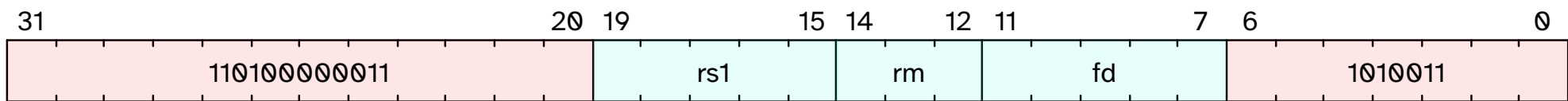
C.109. fcvt.s.lu

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.109.1. Encoding



C.109.2. Description

No description available.

C.109.3. Access

M	S	U
Always	Always	Always

C.109.4. Decode Variables

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

C.109.5. IDL Operation

C.109.6. Sail Operation

```
{
  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.109.7. Exceptions

This instruction does not generate synchronous exceptions.

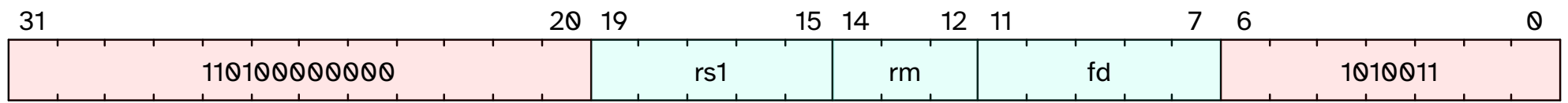
C.110. fcvt.s.w

Convert signed 32-bit integer to single-precision float

This instruction is defined by:

- F, version >= F@2.2.0

C.110.1. Encoding



C.110.2. Description

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

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

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

C.110.3. Access

M	S	U
Always	Always	Always

C.110.4. Decode Variables

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

C.110.5. IDL Operation

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

C.110.6. 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.110.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

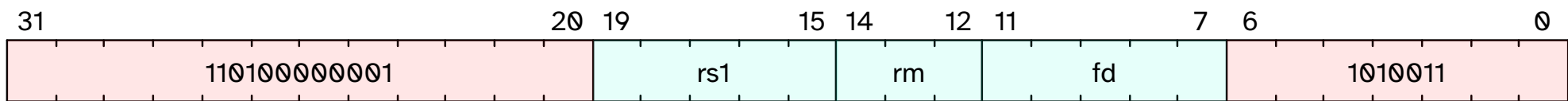
C.111. fcvt.s.wu

Convert unsigned 32-bit integer to single-precision float

This instruction is defined by:

- F, version >= F@2.2.0

C.111.1. Encoding



C.111.2. Description

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

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

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

C.111.3. Access

M	S	U
Always	Always	Always

C.111.4. Decode Variables

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

C.111.5. IDL Operation

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

C.111.6. 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.111.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

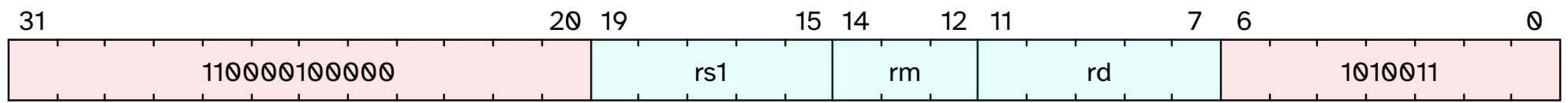
C.112. fcvt.w.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.112.1. Encoding



C.112.2. Description

No description available.

C.112.3. Access

M	S	U
Always	Always	Always

C.112.4. Decode Variables

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

C.112.5. IDL Operation

C.112.6. Exceptions

This instruction does not generate synchronous exceptions.

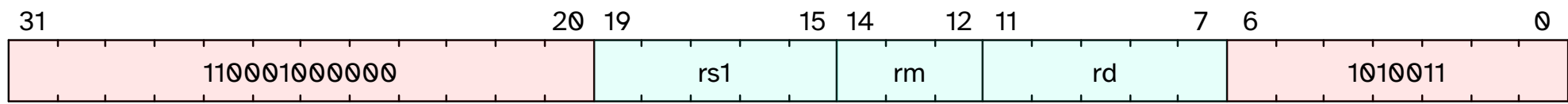
C.113. fcvt.w.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.113.1. Encoding



C.113.2. Description

No description available.

C.113.3. Access

M	S	U
Always	Always	Always

C.113.4. Decode Variables

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

C.113.5. IDL Operation

C.113.6. Exceptions

This instruction does not generate synchronous exceptions.

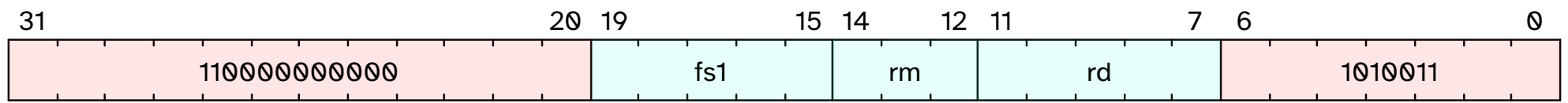
C.114. fcvt.w.s

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

This instruction is defined by:

- F, version >= F@2.2.0

C.114.1. Encoding



C.114.2. Description

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

For XLEN >32, [fcvt.w.s](#) sign-extends the 32-bit result to the destination register width.

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

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

	Value
Minimum valid input (after rounding)	-2 ³¹
Maximum valid input (after rounding)	2 ³¹ - 1
Output for out-of-range negative input	-2 ³¹
Output for <code><code>-&infin;</code></code>	-2 ³¹
Output for out-of-range positive input	2 ³¹ - 1
Output for <code><code>+&infin;</code></code> for <code><code>NaN</code></code>	2 ³¹ - 1

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

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

C.114.3. Access

M	S	U
Always	Always	Always

C.114.4. Decode Variables

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

C.114.5. IDL Operation

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

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

C.114.6. 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.114.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

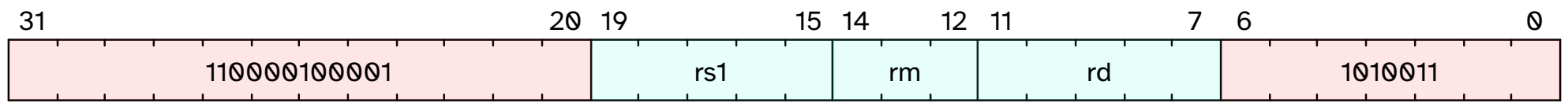
C.115. fcvt.wu.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.115.1. Encoding



C.115.2. Description

No description available.

C.115.3. Access

M	S	U
Always	Always	Always

C.115.4. Decode Variables

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

C.115.5. IDL Operation

C.115.6. Exceptions

This instruction does not generate synchronous exceptions.

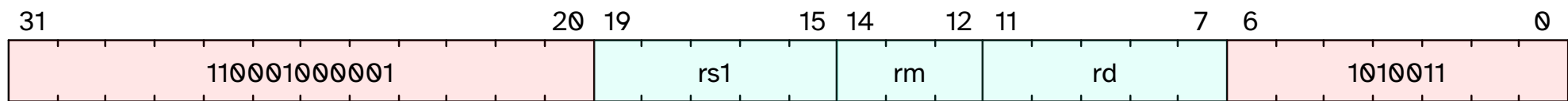
C.116. fcvt.wu.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

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

C.116.5. IDL Operation

C.116.6. Exceptions

This instruction does not generate synchronous exceptions.

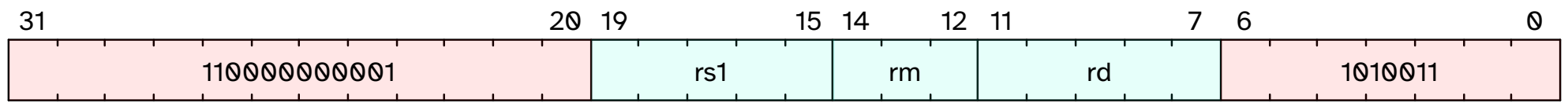
C.117. fcvt.wu.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.117.1. Encoding



C.117.2. Description

No description available.

C.117.3. Access

M	S	U
Always	Always	Always

C.117.4. Decode Variables

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

C.117.5. IDL Operation

C.117.6. 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.117.7. Exceptions

This instruction does not generate synchronous exceptions.

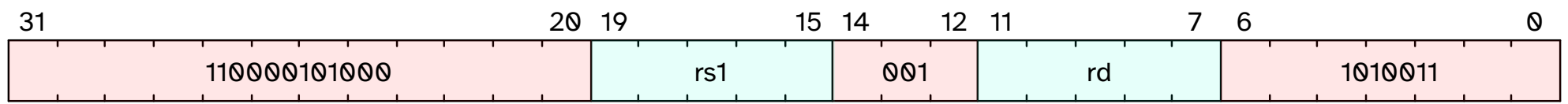
C.118. fcvtmod.w.d

No synopsis available

This instruction is defined by:

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

C.118.1. Encoding



C.118.2. Description

No description available.

C.118.3. Access

M	S	U
Always	Always	Always

C.118.4. Decode Variables

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

C.118.5. IDL Operation

C.118.6. Exceptions

This instruction does not generate synchronous exceptions.

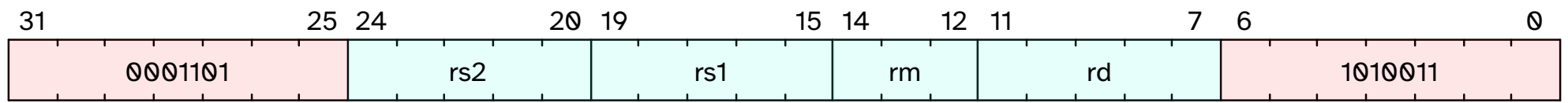
C.119. fdiv.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.119.1. Encoding



C.119.2. Description

No description available.

C.119.3. Access

M	S	U
Always	Always	Always

C.119.4. Decode Variables

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

C.119.5. IDL Operation

C.119.6. Exceptions

This instruction does not generate synchronous exceptions.

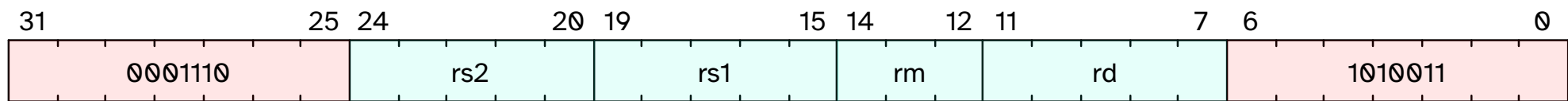
C.120. fdiv.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.120.1. Encoding



C.120.2. Description

No description available.

C.120.3. Access

M	S	U
Always	Always	Always

C.120.4. Decode Variables

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

C.120.5. IDL Operation

C.120.6. Exceptions

This instruction does not generate synchronous exceptions.

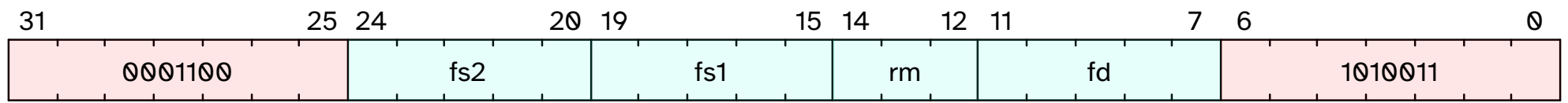
C.121. fdiv.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.121.1. Encoding



C.121.2. Description

No description available.

C.121.3. Access

M	S	U
Always	Always	Always

C.121.4. Decode Variables

```
Bits<5> 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. 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.121.7. Exceptions

This instruction does not generate synchronous exceptions.

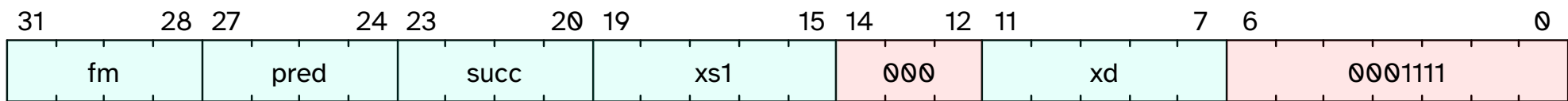
C.122. fence

Memory oxdering fence

This instruction is defined by:

- I, version >= I@2.1.0

C.122.1. Encoding



C.122.2. Description

Oxders memory operations.

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

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

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

Table 11. Fence mode encoding

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

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

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

The xs1 and xd fields are unused and ignored.

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

Table 12. Effective PR/PW/SR/SW in (H)S-mode

Table 13. Effective PR/PW/SR/SW in U-mode

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

<%- end -%>

C.122.3. Access

M	S	U
Always	Always	Always

C.122.4. 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.122.5. IDL Operation

```

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

```

C.122.6. 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.122.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

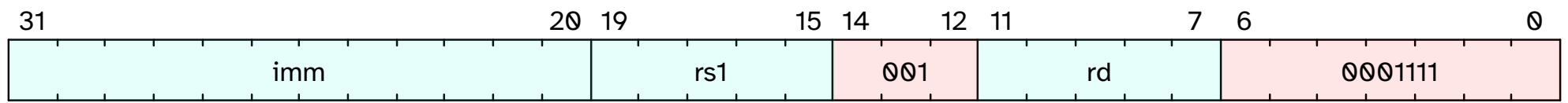
C.123. fence.i

Instruction fence

This instruction is defined by:

- Zifencei, version >= Zifencei@2.0.0

C.123.1. Encoding



C.123.2. Description

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

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



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

C.123.3. Access

M	S	U
Always	Always	Always

C.123.4. Decode Variables

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

C.123.5. IDL Operation

```
ifence();
```

C.123.6. Sail Operation

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

C.123.7. Exceptions

This instruction does not generate synchronous exceptions.

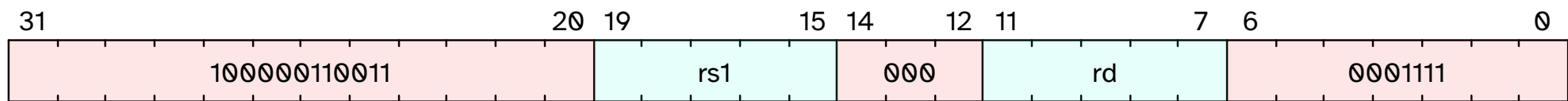
C.124. fence.tso

Memory ordering fence, total store ordering

This instruction is defined by:

- I, version >= I@2.1.0

C.124.1. Encoding



C.124.2. Description

Orders memory operations.

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

The rs1 and rd fields are unused and ignored.

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

C.124.3. Access

M	S	U
Always	Always	Always

C.124.4. Decode Variables

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

C.124.5. IDL Operation

```
fence_tso();
```

C.124.6. 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.124.7. Exceptions

This instruction does not generate synchronous exceptions.

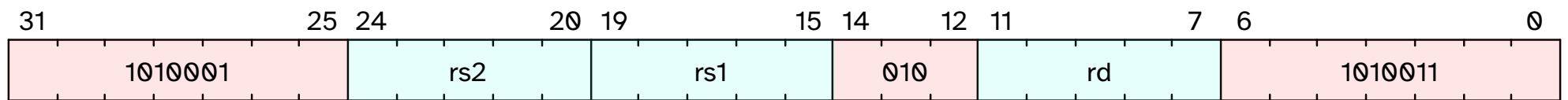
C.125. feq.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

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

C.125.5. IDL Operation

C.125.6. Exceptions

This instruction does not generate synchronous exceptions.

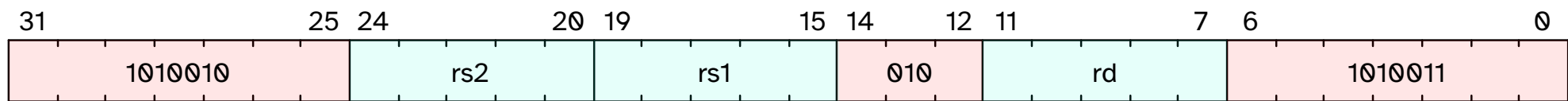
C.126. feq.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.126.1. Encoding



C.126.2. Description

No description available.

C.126.3. Access

M	S	U
Always	Always	Always

C.126.4. Decode Variables

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

C.126.5. IDL Operation

C.126.6. Exceptions

This instruction does not generate synchronous exceptions.

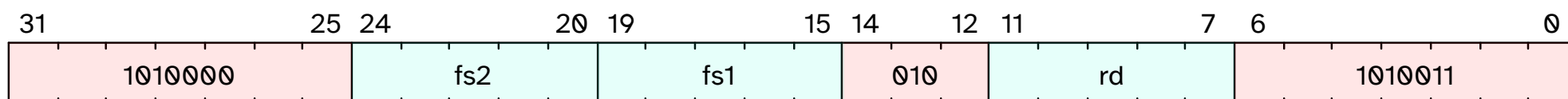
C.127. feq.s

Single-precision floating-point equal

This instruction is defined by:

- F, version $\geq$ F@2.2.0

C.127.1. Encoding



C.127.2. Description

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

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

Positive zero is considered equal to negative zero.

C.127.3. Access

M	S	U
Always	Always	Always

C.127.4. Decode Variables

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

C.127.5. IDL Operation

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

C.127.6. 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.127.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

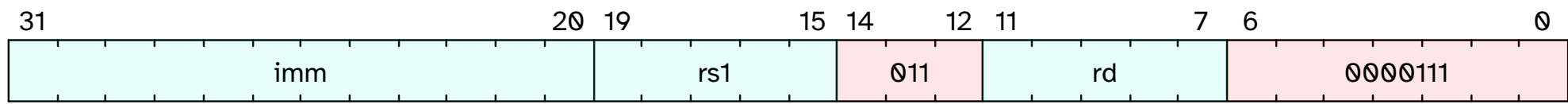
C.128. fld

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.128.1. Encoding



C.128.2. Description

No description available.

C.128.3. Access

M	S	U
Always	Always	Always

C.128.4. Decode Variables

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

C.128.5. IDL Operation

C.128.6. Exceptions

This instruction does not generate synchronous exceptions.

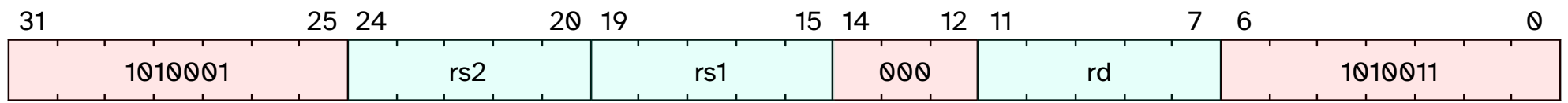
C.129. fle.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.129.1. Encoding



C.129.2. Description

No description available.

C.129.3. Access

M	S	U
Always	Always	Always

C.129.4. Decode Variables

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

C.129.5. IDL Operation

C.129.6. Exceptions

This instruction does not generate synchronous exceptions.

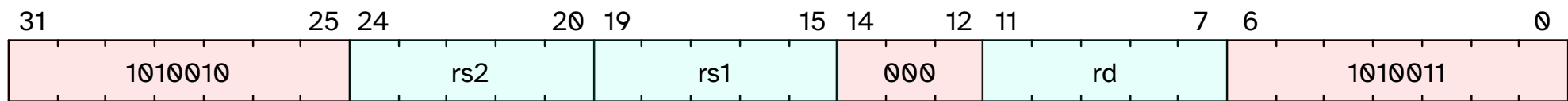
C.130. fle.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.130.1. Encoding



C.130.2. Description

No description available.

C.130.3. Access

M	S	U
Always	Always	Always

C.130.4. Decode Variables

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

C.130.5. IDL Operation

C.130.6. Exceptions

This instruction does not generate synchronous exceptions.

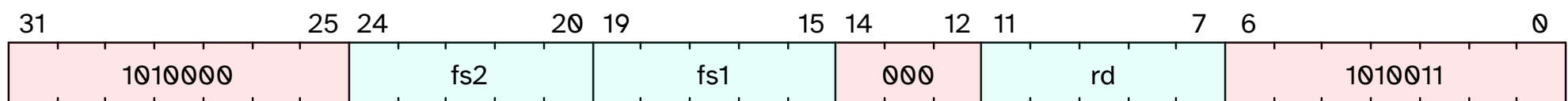
C.131. fle.s

Single-precision floating-point less than or equal

This instruction is defined by:

- F, version $\geq$ F@2.2.0

C.131.1. Encoding



C.131.2. Description

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

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

Positive zero and negative zero are considered equal.

C.131.3. Access

M	S	U
Always	Always	Always

C.131.4. Decode Variables

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

C.131.5. IDL Operation

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

C.131.6. 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.131.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

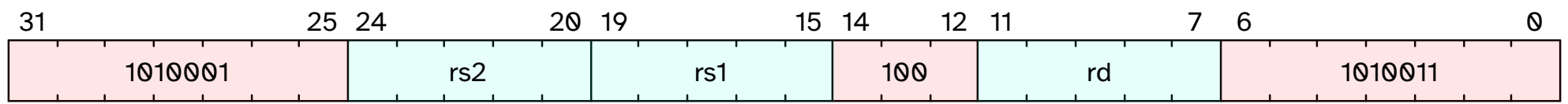
C.132. fleq.d

No synopsis available

This instruction is defined by:

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

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

C.132.5. IDL Operation

C.132.6. Exceptions

This instruction does not generate synchronous exceptions.

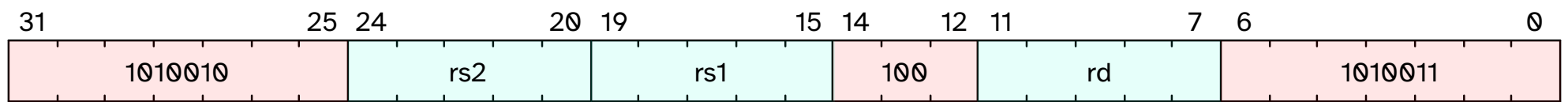
C.133. fleq.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.133.1. Encoding



C.133.2. Description

No description available.

C.133.3. Access

M	S	U
Always	Always	Always

C.133.4. Decode Variables

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

C.133.5. IDL Operation

C.133.6. Exceptions

This instruction does not generate synchronous exceptions.

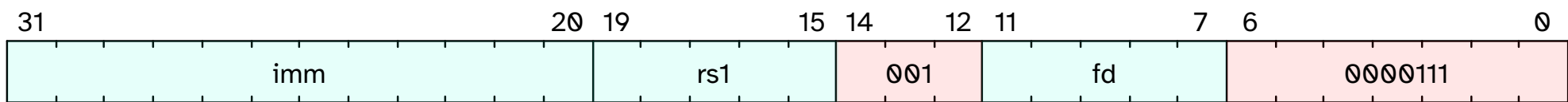
C.134. flh

Half-precision floating-point load

This instruction is defined by:

- anyOf:
 - Zfh, version >= Zfh@1.0.0
 - Zfhmin, version >= Zfhmin@1.0.0
 - Zhinx, version >= Zhinx@1.0.0

C.134.1. Encoding



C.134.2. Description

The `flh` instruction loads a single-precision floating-point value from memory at address `rs1 + imm` into floating-point register `rd`.

`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.134.3. Access

M	S	U
Always	Always	Always

C.134.4. Decode Variables

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

C.134.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $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.134.6. 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.134.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

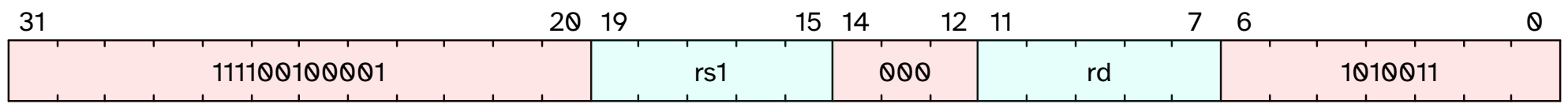
C.135. fli.d

No synopsis available

This instruction is defined by:

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

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

C.135.5. IDL Operation

C.135.6. Exceptions

This instruction does not generate synchronous exceptions.

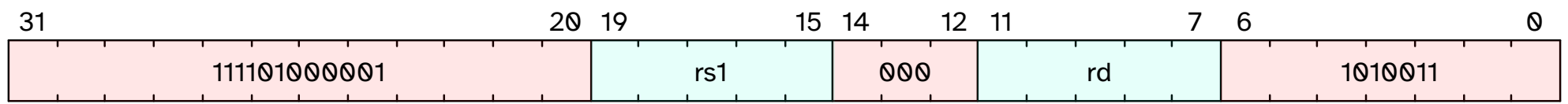
C.136. fli.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.136.1. Encoding



C.136.2. Description

No description available.

C.136.3. Access

M	S	U
Always	Always	Always

C.136.4. Decode Variables

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

C.136.5. IDL Operation

C.136.6. Exceptions

This instruction does not generate synchronous exceptions.

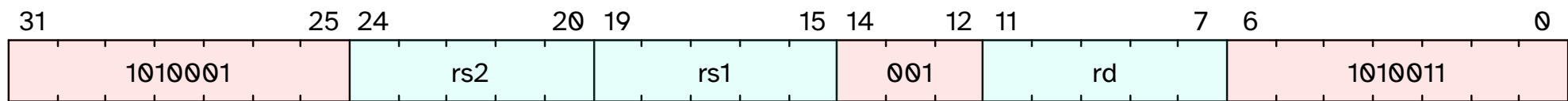
C.137. flt.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.137.1. Encoding



C.137.2. Description

No description available.

C.137.3. Access

M	S	U
Always	Always	Always

C.137.4. Decode Variables

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

C.137.5. IDL Operation

C.137.6. Exceptions

This instruction does not generate synchronous exceptions.

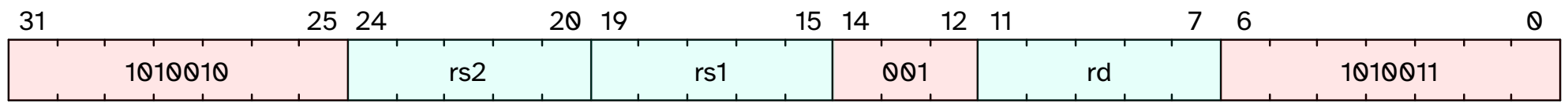
C.138. flt.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.138.1. Encoding



C.138.2. Description

No description available.

C.138.3. Access

M	S	U
Always	Always	Always

C.138.4. Decode Variables

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

C.138.5. IDL Operation

C.138.6. Exceptions

This instruction does not generate synchronous exceptions.

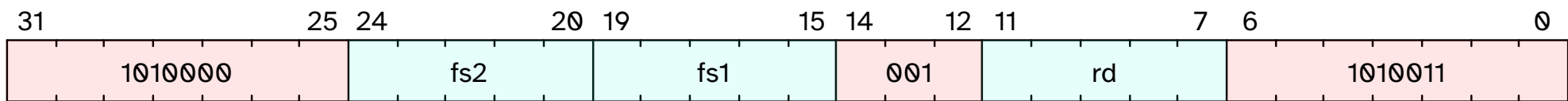
C.139. fts.s

Single-precision floating-point less than

This instruction is defined by:

- F, version >= F@2.2.0

C.139.1. Encoding



C.139.2. Description

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

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

C.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> rd = $encoding[11:7];
```

C.139.5. IDL Operation

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

C.139.6. 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.139.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

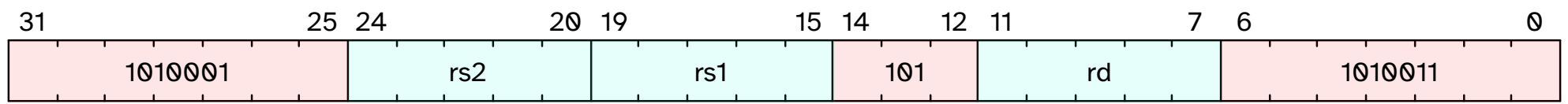
C.140. fltq.d

No synopsis available

This instruction is defined by:

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

C.140.1. Encoding



C.140.2. Description

No description available.

C.140.3. Access

M	S	U
Always	Always	Always

C.140.4. Decode Variables

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

C.140.5. IDL Operation

C.140.6. Exceptions

This instruction does not generate synchronous exceptions.

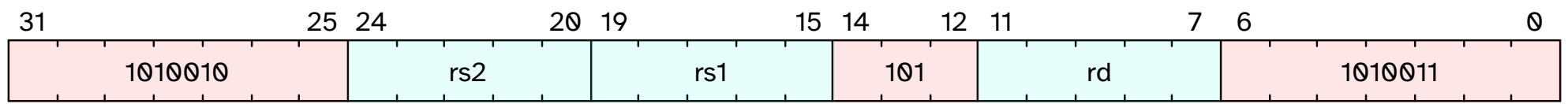
C.141. fltq.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.141.1. Encoding



C.141.2. Description

No description available.

C.141.3. Access

M	S	U
Always	Always	Always

C.141.4. Decode Variables

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

C.141.5. IDL Operation

C.141.6. Exceptions

This instruction does not generate synchronous exceptions.

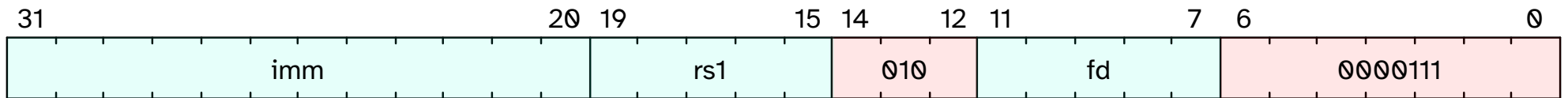
C.142. flw

Single-precision floating-point load

This instruction is defined by:

- F, version $\geq$ F@2.2.0

C.142.1. Encoding



C.142.2. Description

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

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

C.142.3. Access

M	S	U
Always	Always	Always

C.142.4. Decode Variables

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

C.142.5. IDL Operation

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

C.142.6. 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.142.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

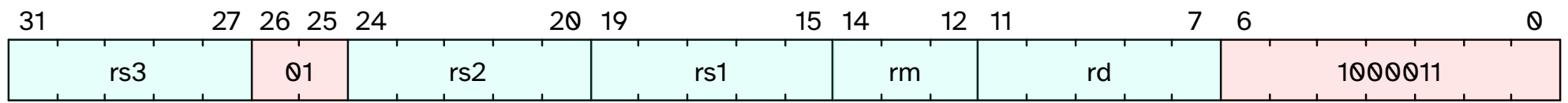
C.143. fmadd.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.143.1. Encoding



C.143.2. Description

No description available.

C.143.3. Access

M	S	U
Always	Always	Always

C.143.4. Decode Variables

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

C.143.5. IDL Operation

C.143.6. Exceptions

This instruction does not generate synchronous exceptions.

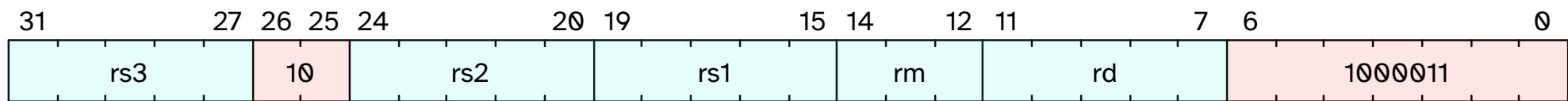
C.144. fmadd.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.144.1. Encoding



C.144.2. Description

No description available.

C.144.3. Access

M	S	U
Always	Always	Always

C.144.4. Decode Variables

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

C.144.5. IDL Operation

C.144.6. Exceptions

This instruction does not generate synchronous exceptions.

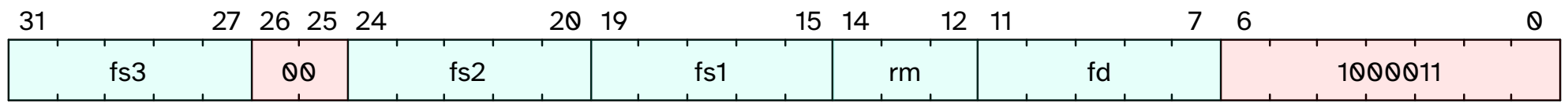
C.145. fmadd.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

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> 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.145.5. IDL Operation

C.145.6. 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),
          FMADD_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.145.7. Exceptions

This instruction does not generate synchronous exceptions.

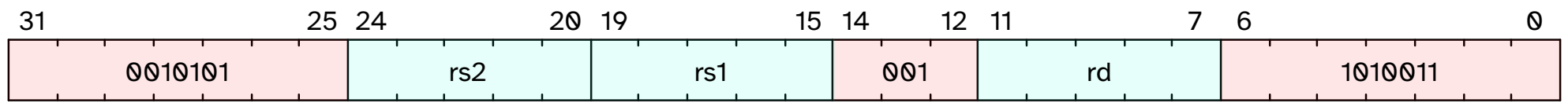
C.146. fmax.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.146.1. Encoding



C.146.2. Description

No description available.

C.146.3. Access

M	S	U
Always	Always	Always

C.146.4. Decode Variables

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

C.146.5. IDL Operation

C.146.6. Exceptions

This instruction does not generate synchronous exceptions.

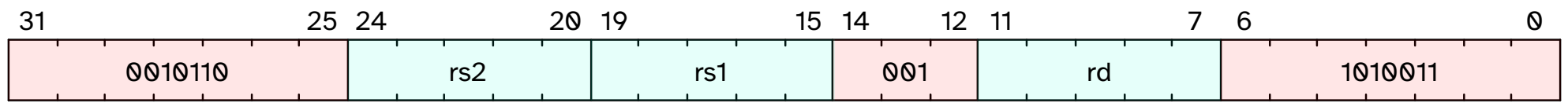
C.147. fmax.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.147.1. Encoding



C.147.2. Description

No description available.

C.147.3. Access

M	S	U
Always	Always	Always

C.147.4. Decode Variables

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

C.147.5. IDL Operation

C.147.6. Exceptions

This instruction does not generate synchronous exceptions.

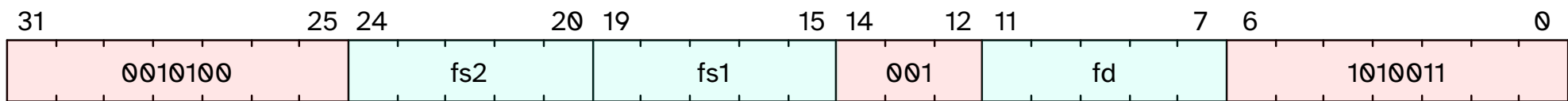
C.148. fmax.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.148.1. Encoding



C.148.2. Description

No description available.

C.148.3. Access

M	S	U
Always	Always	Always

C.148.4. Decode Variables

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

C.148.5. IDL Operation

C.148.6. 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.148.7. Exceptions

This instruction does not generate synchronous exceptions.

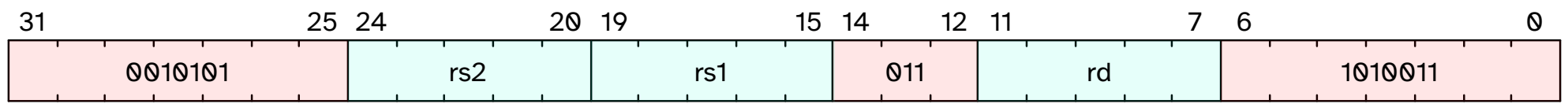
C.149. fmaxm.d

No synopsis available

This instruction is defined by:

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

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

C.149.5. IDL Operation

C.149.6. Exceptions

This instruction does not generate synchronous exceptions.

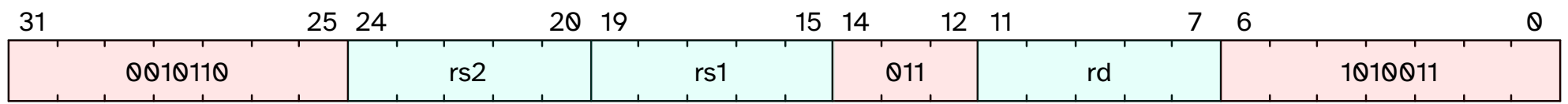
C.150. fmaxm.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.150.1. Encoding



C.150.2. Description

No description available.

C.150.3. Access

M	S	U
Always	Always	Always

C.150.4. Decode Variables

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

C.150.5. IDL Operation

C.150.6. Exceptions

This instruction does not generate synchronous exceptions.

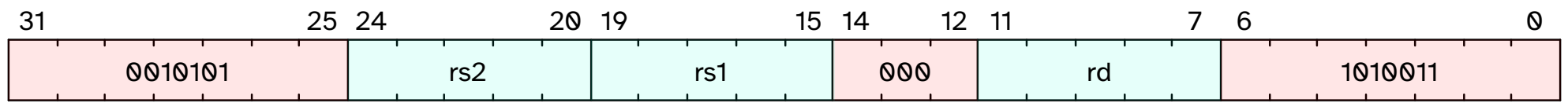
C.151. fmin.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.151.1. Encoding



C.151.2. Description

No description available.

C.151.3. Access

M	S	U
Always	Always	Always

C.151.4. Decode Variables

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

C.151.5. IDL Operation

C.151.6. Exceptions

This instruction does not generate synchronous exceptions.

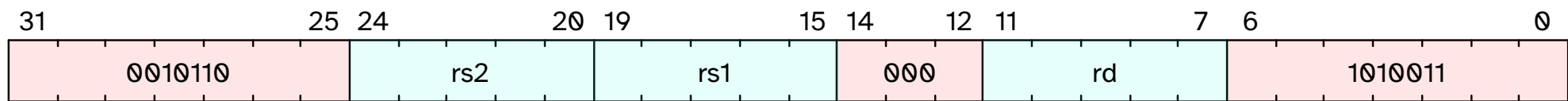
C.152. fmin.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

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

C.152.5. IDL Operation

C.152.6. Exceptions

This instruction does not generate synchronous exceptions.

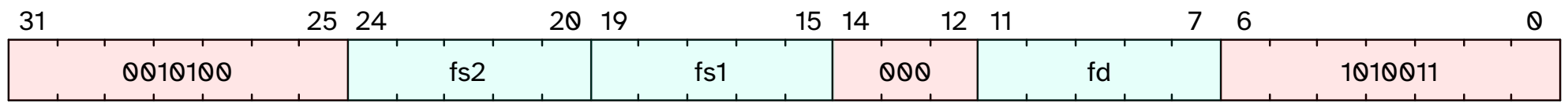
C.153. fmin.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.153.1. Encoding



C.153.2. Description

No description available.

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

C.153.5. IDL Operation

C.153.6. 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.153.7. Exceptions

This instruction does not generate synchronous exceptions.

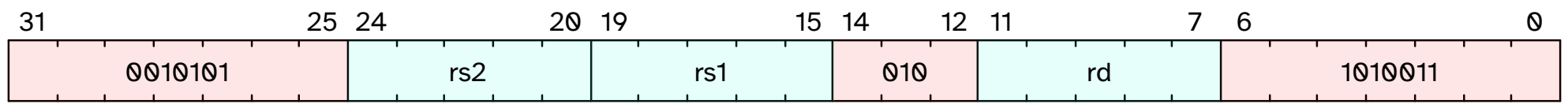
C.154. fminm.d

No synopsis available

This instruction is defined by:

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

C.154.1. Encoding



C.154.2. Description

No description available.

C.154.3. Access

M	S	U
Always	Always	Always

C.154.4. Decode Variables

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

C.154.5. IDL Operation

C.154.6. Exceptions

This instruction does not generate synchronous exceptions.

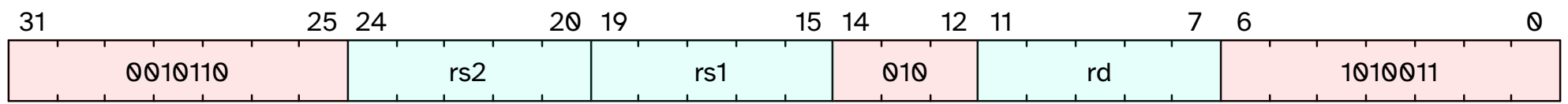
C.155. fminm.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.155.1. Encoding



C.155.2. Description

No description available.

C.155.3. Access

M	S	U
Always	Always	Always

C.155.4. Decode Variables

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

C.155.5. IDL Operation

C.155.6. Exceptions

This instruction does not generate synchronous exceptions.

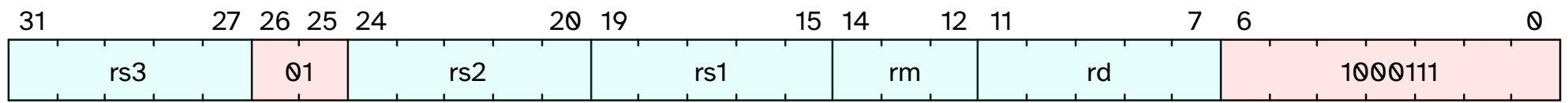
C.156. fmsub.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.156.1. Encoding



C.156.2. Description

No description available.

C.156.3. Access

M	S	U
Always	Always	Always

C.156.4. Decode Variables

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

C.156.5. IDL Operation

C.156.6. Exceptions

This instruction does not generate synchronous exceptions.

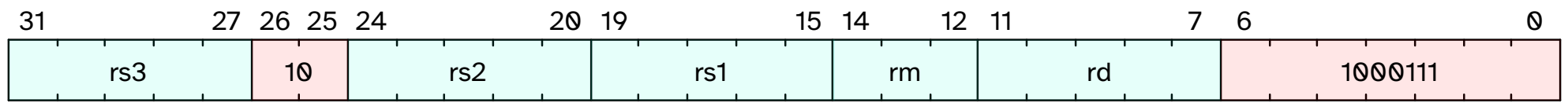
C.157. fmsub.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.157.1. Encoding



C.157.2. Description

No description available.

C.157.3. Access

M	S	U
Always	Always	Always

C.157.4. Decode Variables

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

C.157.5. IDL Operation

C.157.6. Exceptions

This instruction does not generate synchronous exceptions.

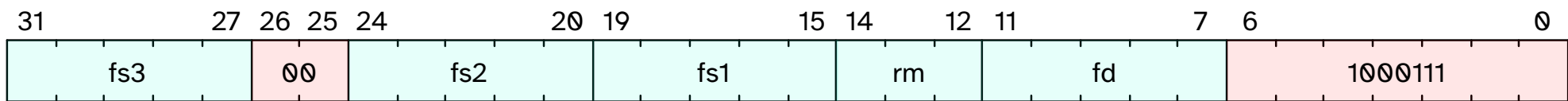
C.158. fmsub.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.158.1. Encoding



C.158.2. Description

No description available.

C.158.3. Access

M	S	U
Always	Always	Always

C.158.4. 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.158.5. IDL Operation

C.158.6. 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),
          FMADD_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.158.7. Exceptions

This instruction does not generate synchronous exceptions.

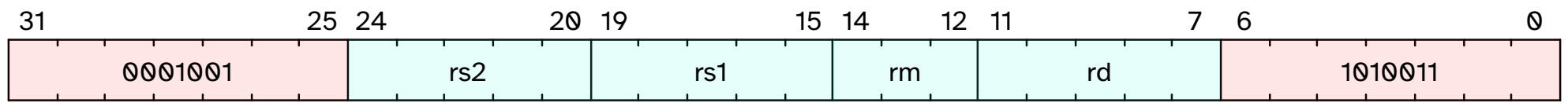
C.159. fmul.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.159.1. Encoding



C.159.2. Description

No description available.

C.159.3. Access

M	S	U
Always	Always	Always

C.159.4. Decode Variables

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

C.159.5. IDL Operation

C.159.6. Exceptions

This instruction does not generate synchronous exceptions.

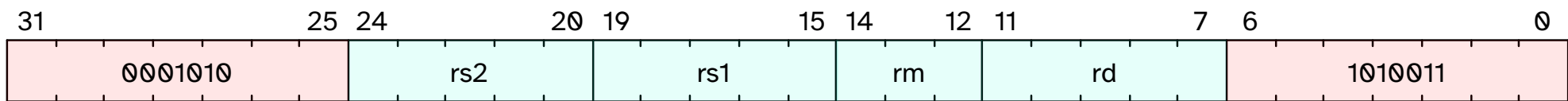
C.160. fmulh

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.160.1. Encoding



C.160.2. Description

No description available.

C.160.3. Access

M	S	U
Always	Always	Always

C.160.4. Decode Variables

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

C.160.5. IDL Operation

C.160.6. Exceptions

This instruction does not generate synchronous exceptions.

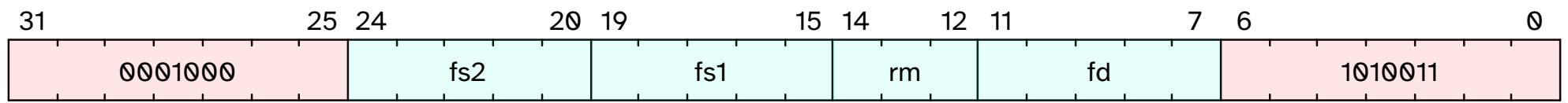
C.161. fmul.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.161.1. Encoding



C.161.2. Description

No description available.

C.161.3. Access

M	S	U
Always	Always	Always

C.161.4. 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.161.5. IDL Operation

C.161.6. 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.161.7. Exceptions

This instruction does not generate synchronous exceptions.

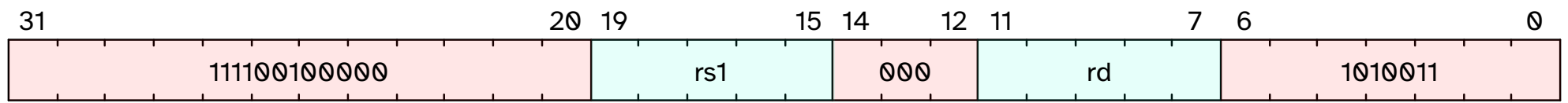
C.162. fmv.d.x

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.162.1. Encoding



C.162.2. Description

No description available.

C.162.3. Access

M	S	U
Always	Always	Always

C.162.4. Decode Variables

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

C.162.5. IDL Operation

C.162.6. Exceptions

This instruction does not generate synchronous exceptions.

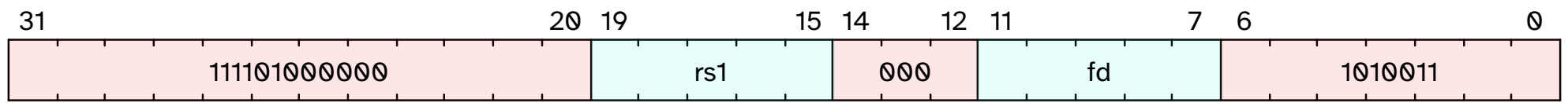
C.163. fmv.h.x

Half-precision floating-point move from integer

This instruction is defined by:

- F, version >= F@2.2.0

C.163.1. Encoding



C.163.2. Description

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

C.163.3. Access

M	S	U
Always	Always	Always

C.163.4. Decode Variables

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

C.163.5. IDL Operation

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

C.163.6. 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.163.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

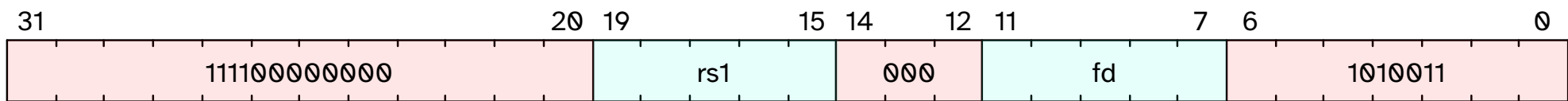
C.164. fmv.w.x

Single-precision floating-point move from integer

This instruction is defined by:

- F, version >= F@2.2.0

C.164.1. Encoding



C.164.2. Description

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

C.164.3. Access

M	S	U
Always	Always	Always

C.164.4. Decode Variables

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

C.164.5. IDL Operation

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

C.164.6. 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.164.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

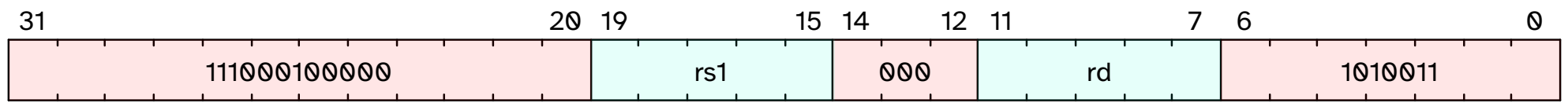
C.165. fmv.x.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.165.1. Encoding



C.165.2. Description

No description available.

C.165.3. Access

M	S	U
Always	Always	Always

C.165.4. Decode Variables

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

C.165.5. IDL Operation

C.165.6. Exceptions

This instruction does not generate synchronous exceptions.

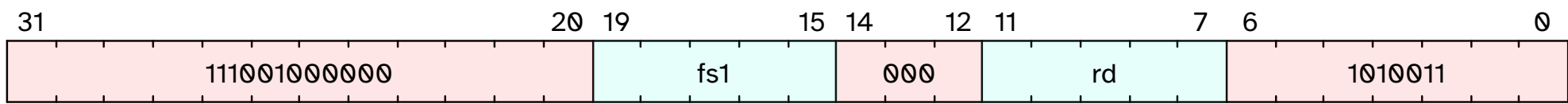
C.166. fmv.x.h

Move half-precision value from floating-point to integer register

This instruction is defined by:

- anyOf:
 - Zfh, version >= Zfh@1.0.0
 - Zfhmin, version >= Zfhmin@1.0.0
 - Zhinx, version >= Zhinx@1.0.0

C.166.1. Encoding



C.166.2. Description

Moves the half-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 16 bits of integer register rd.

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.166.3. Access

M	S	U
Always	Always	Always

C.166.4. Decode Variables

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

C.166.5. IDL Operation

```
check_f_ok($encoding);
X[rd] = sext(f[fs1][15:0], 16);
```

C.166.6. 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.166.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

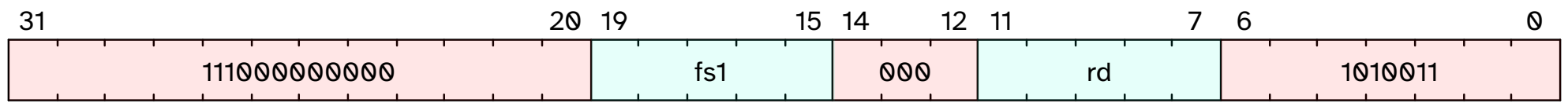
C.167. fmv.x.w

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

This instruction is defined by:

- F, version >= F@2.2.0

C.167.1. Encoding



C.167.2. Description

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

C.167.3. Access

M	S	U
Always	Always	Always

C.167.4. Decode Variables

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

C.167.5. IDL Operation

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

C.167.6. 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.167.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

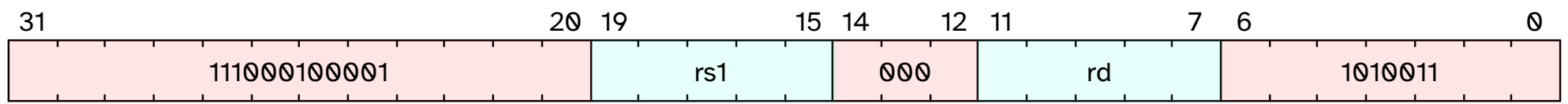
C.168. fmvh.x.d

No synopsis available

This instruction is defined by:

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

C.168.1. Encoding



C.168.2. Description

No description available.

C.168.3. Access

M	S	U
Always	Always	Always

C.168.4. Decode Variables

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

C.168.5. IDL Operation

C.168.6. Exceptions

This instruction does not generate synchronous exceptions.

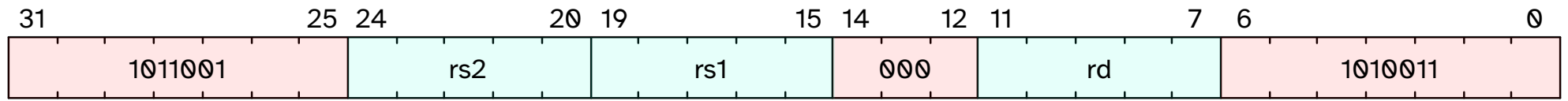
C.169. fmv.d.x

No synopsis available

This instruction is defined by:

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

C.169.1. Encoding



C.169.2. Description

No description available.

C.169.3. Access

M	S	U
Always	Always	Always

C.169.4. Decode Variables

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

C.169.5. IDL Operation

C.169.6. Exceptions

This instruction does not generate synchronous exceptions.

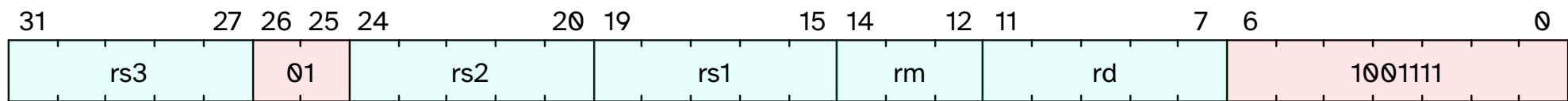
C.170. fnmadd.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.170.1. Encoding



C.170.2. Description

No description available.

C.170.3. Access

M	S	U
Always	Always	Always

C.170.4. Decode Variables

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

C.170.5. IDL Operation

C.170.6. Exceptions

This instruction does not generate synchronous exceptions.

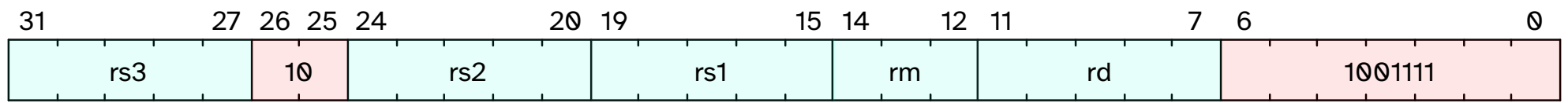
C.171. fnmadd.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.171.1. Encoding



C.171.2. Description

No description available.

C.171.3. Access

M	S	U
Always	Always	Always

C.171.4. Decode Variables

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

C.171.5. IDL Operation

C.171.6. Exceptions

This instruction does not generate synchronous exceptions.

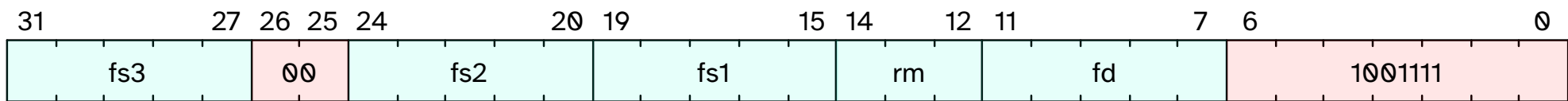
C.172. fnmadd.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.172.1. Encoding



C.172.2. Description

No description available.

C.172.3. Access

M	S	U
Always	Always	Always

C.172.4. 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.172.5. IDL Operation

C.172.6. 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),
          FMADD_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.172.7. Exceptions

This instruction does not generate synchronous exceptions.

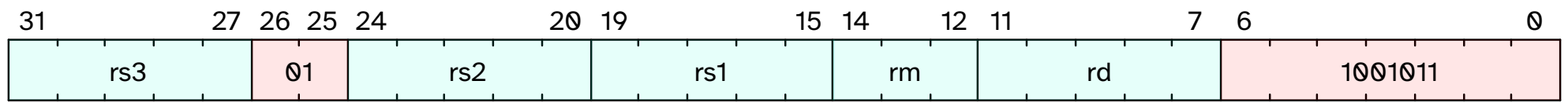
C.173. fnmsub.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.173.1. Encoding



C.173.2. Description

No description available.

C.173.3. Access

M	S	U
Always	Always	Always

C.173.4. Decode Variables

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

C.173.5. IDL Operation

C.173.6. Exceptions

This instruction does not generate synchronous exceptions.

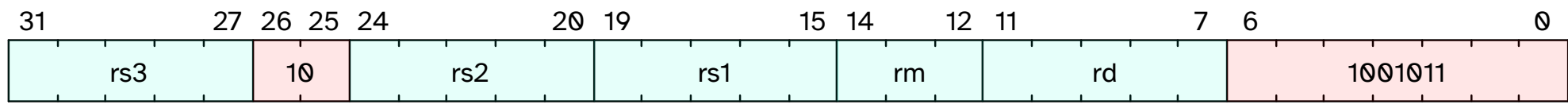
C.174. fnmsub.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.174.1. Encoding



C.174.2. Description

No description available.

C.174.3. Access

M	S	U
Always	Always	Always

C.174.4. Decode Variables

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

C.174.5. IDL Operation

C.174.6. Exceptions

This instruction does not generate synchronous exceptions.

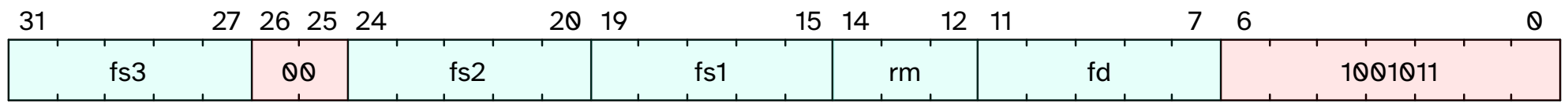
C.175. fnmsub.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.175.1. Encoding



C.175.2. Description

No description available.

C.175.3. Access

M	S	U
Always	Always	Always

C.175.4. 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.175.5. IDL Operation

C.175.6. 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),
          FMADD_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.175.7. Exceptions

This instruction does not generate synchronous exceptions.

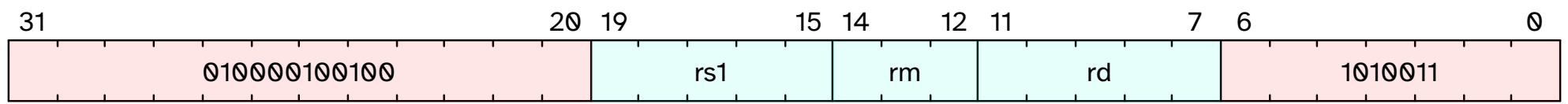
C.176. fround.d

No synopsis available

This instruction is defined by:

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

C.176.1. Encoding



C.176.2. Description

No description available.

C.176.3. Access

M	S	U
Always	Always	Always

C.176.4. Decode Variables

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

C.176.5. IDL Operation

C.176.6. Exceptions

This instruction does not generate synchronous exceptions.

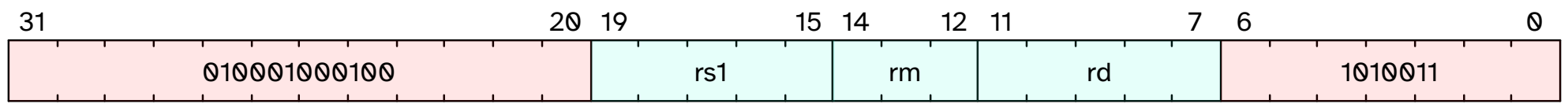
C.177. fround.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.177.1. Encoding



C.177.2. Description

No description available.

C.177.3. Access

M	S	U
Always	Always	Always

C.177.4. Decode Variables

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

C.177.5. IDL Operation

C.177.6. Exceptions

This instruction does not generate synchronous exceptions.

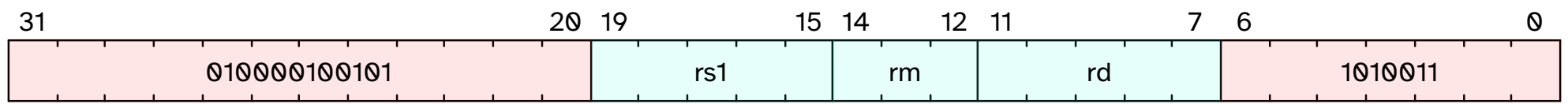
C.178. froundnx.d

No synopsis available

This instruction is defined by:

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

C.178.1. Encoding



C.178.2. Description

No description available.

C.178.3. Access

M	S	U
Always	Always	Always

C.178.4. Decode Variables

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

C.178.5. IDL Operation

C.178.6. Exceptions

This instruction does not generate synchronous exceptions.

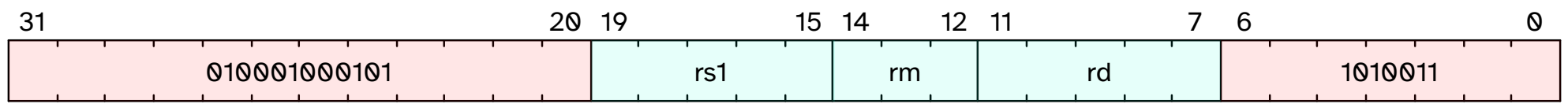
C.179. froundnx.h

No synopsis available

This instruction is defined by:

- allOf:
 - Zfa, version >= Zfa@1.0.0
 - Zfh, version >= Zfh@1.0.0

C.179.1. Encoding



C.179.2. Description

No description available.

C.179.3. Access

M	S	U
Always	Always	Always

C.179.4. Decode Variables

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

C.179.5. IDL Operation

C.179.6. Exceptions

This instruction does not generate synchronous exceptions.

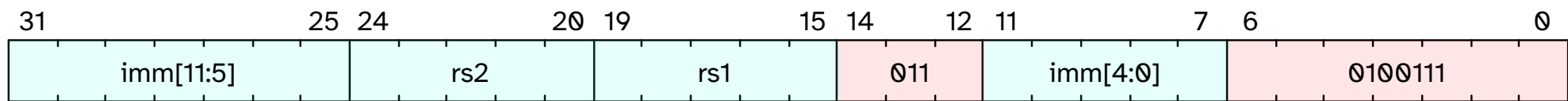
C.180. fsd

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.180.1. Encoding



C.180.2. Description

No description available.

C.180.3. Access

M	S	U
Always	Always	Always

C.180.4. Decode Variables

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

C.180.5. IDL Operation

C.180.6. Exceptions

This instruction does not generate synchronous exceptions.

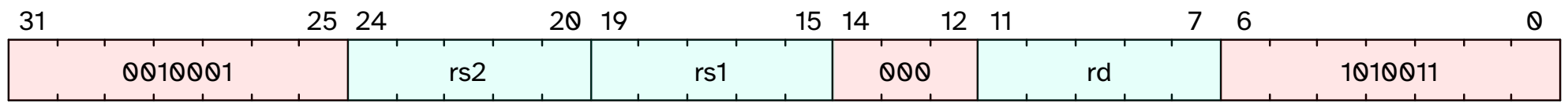
C.181. fsgnj.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.181.1. Encoding



C.181.2. Description

No description available.

C.181.3. Access

M	S	U
Always	Always	Always

C.181.4. Decode Variables

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

C.181.5. IDL Operation

C.181.6. Exceptions

This instruction does not generate synchronous exceptions.

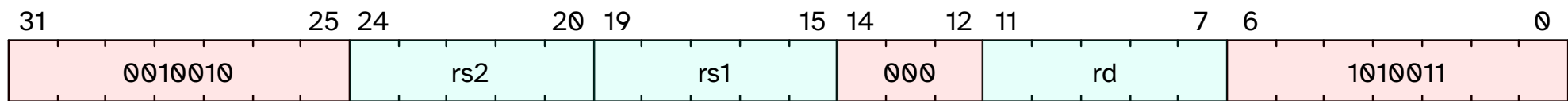
C.182. fsgnj.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.182.1. Encoding



C.182.2. Description

No description available.

C.182.3. Access

M	S	U
Always	Always	Always

C.182.4. Decode Variables

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

C.182.5. IDL Operation

C.182.6. Exceptions

This instruction does not generate synchronous exceptions.

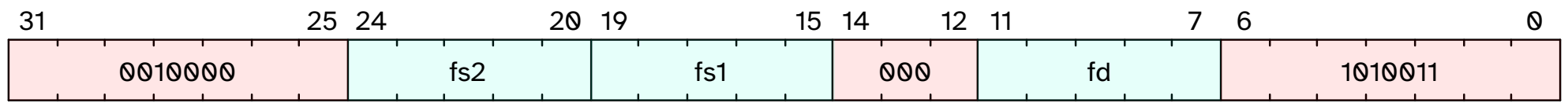
C.183. fsgnj.s

Single-precision sign inject

This instruction is defined by:

- F, version >= F@2.2.0

C.183.1. Encoding



C.183.2. Description

Writes *fd* with sign bit of *fs2* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.183.3. Access

M	S	U
Always	Always	Always

C.183.4. Decode Variables

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

C.183.5. 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.183.6. 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.183.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

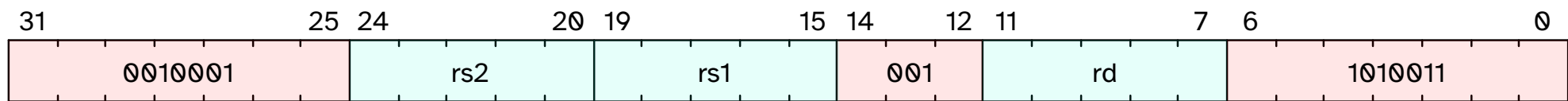
C.184. fsgnjn.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.184.1. Encoding



C.184.2. Description

No description available.

C.184.3. Access

M	S	U
Always	Always	Always

C.184.4. Decode Variables

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

C.184.5. IDL Operation

C.184.6. Exceptions

This instruction does not generate synchronous exceptions.

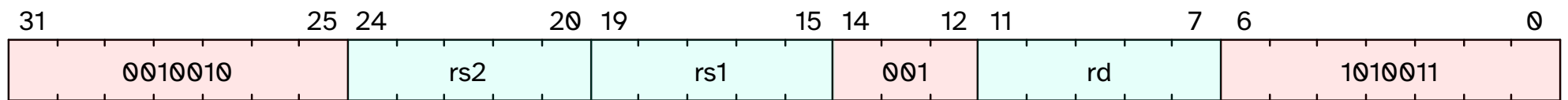
C.185. fsgnjn.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.185.1. Encoding



C.185.2. Description

No description available.

C.185.3. Access

M	S	U
Always	Always	Always

C.185.4. Decode Variables

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

C.185.5. IDL Operation

C.185.6. Exceptions

This instruction does not generate synchronous exceptions.

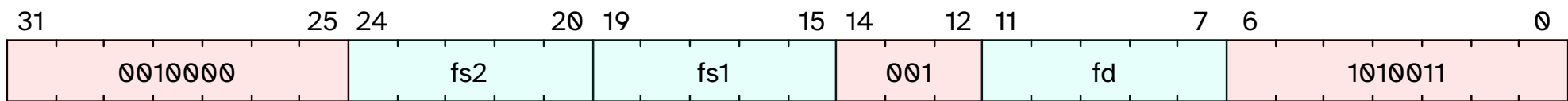
C.186. fsgnjn.s

Single-precision sign inject negate

This instruction is defined by:

- F, version >= F@2.2.0

C.186.1. Encoding



C.186.2. Description

Writes *fd* with the opposite of the sign bit of *fs2* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.186.3. Access

M	S	U
Always	Always	Always

C.186.4. Decode Variables

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

C.186.5. 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.186.6. 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.186.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

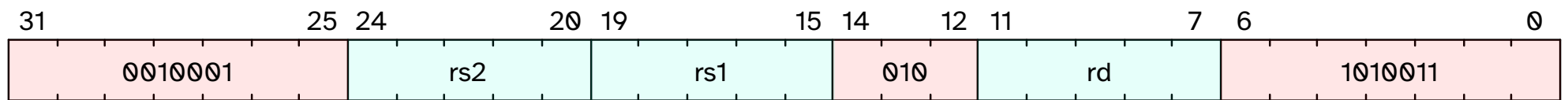
C.187. fsgnjx.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.187.1. Encoding



C.187.2. Description

No description available.

C.187.3. Access

M	S	U
Always	Always	Always

C.187.4. Decode Variables

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

C.187.5. IDL Operation

C.187.6. Exceptions

This instruction does not generate synchronous exceptions.

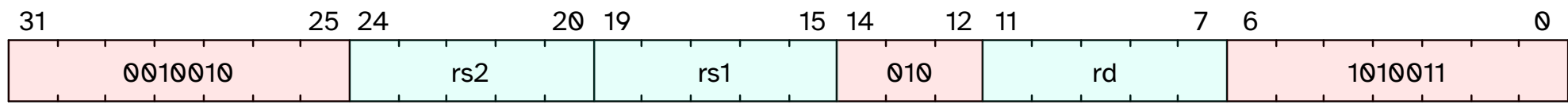
C.188. fsgnjx.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.188.1. Encoding



C.188.2. Description

No description available.

C.188.3. Access

M	S	U
Always	Always	Always

C.188.4. Decode Variables

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

C.188.5. IDL Operation

C.188.6. Exceptions

This instruction does not generate synchronous exceptions.

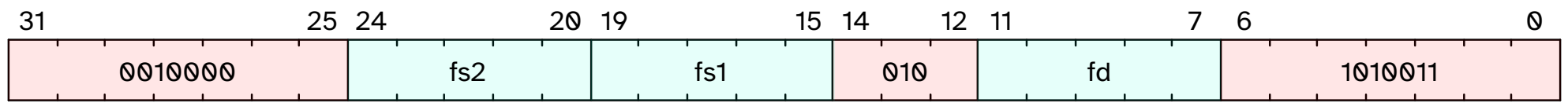
C.189. fsgnjx.s

Single-precision sign inject exclusive or

This instruction is defined by:

- F, version >= F@2.2.0

C.189.1. Encoding



C.189.2. Description

Writes *fd* with the xor of the sign bits of *fs2* and *fs1* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.189.3. Access

M	S	U
Always	Always	Always

C.189.4. Decode Variables

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

C.189.5. 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.189.6. 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.189.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

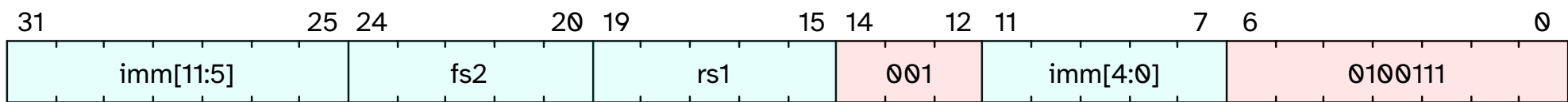
C.190. fsh

Half-precision floating-point store

This instruction is defined by:

- anyOf:
 - Zfh, version >= Zfh@1.0.0
 - Zfhmin, version >= Zfhmin@1.0.0
 - Zhinx, version >= Zhinx@1.0.0

C.190.1. Encoding



C.190.2. Description

The `fsh` instruction stores a half-precision floating-point value from register *rd* to memory at address *rs1* + *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 *rs2*.

`fsh` is only guaranteed to execute atomically if the effective address is naturally aligned.

C.190.3. Access

M	S	U
Always	Always	Always

C.190.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs1 = $encoding[19:15];
Bits<5> fs2 = $encoding[24:20];
```

C.190.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $signed(imm);
Bits<16> hp_value = f[fs2][15:0];
write_memory<16>(virtual_address, hp_value, $encoding);
```

C.190.6. 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) {
```

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

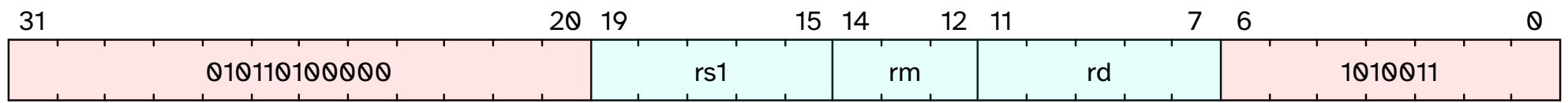
C.191. fsqrt.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.191.1. Encoding



C.191.2. Description

No description available.

C.191.3. Access

M	S	U
Always	Always	Always

C.191.4. Decode Variables

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

C.191.5. IDL Operation

C.191.6. Exceptions

This instruction does not generate synchronous exceptions.

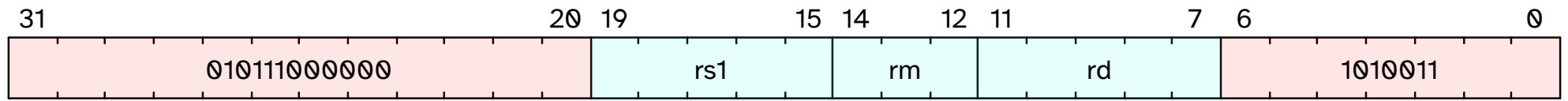
C.192. fsqrt.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.192.1. Encoding



C.192.2. Description

No description available.

C.192.3. Access

M	S	U
Always	Always	Always

C.192.4. Decode Variables

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

C.192.5. IDL Operation

C.192.6. Exceptions

This instruction does not generate synchronous exceptions.

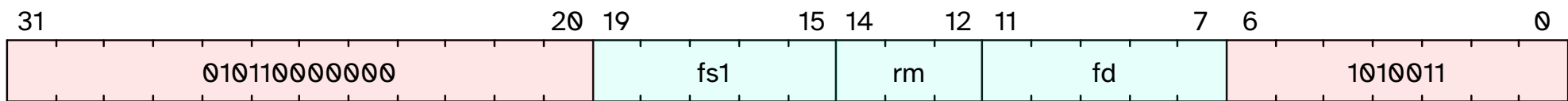
C.193. fsqrt.s

No synopsis available

This instruction is defined by:

- F, version >= F@2.2.0

C.193.1. Encoding



C.193.2. Description

No description available.

C.193.3. Access

M	S	U
Always	Always	Always

C.193.4. Decode Variables

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

C.193.5. IDL Operation

C.193.6. 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.193.7. Exceptions

This instruction does not generate synchronous exceptions.

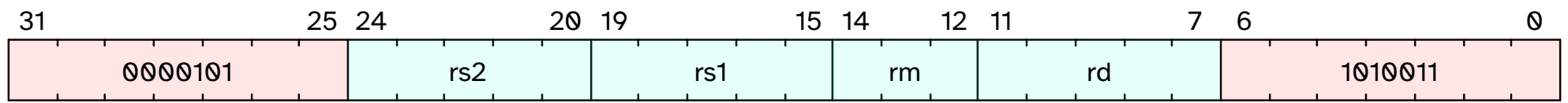
C.194. fsub.d

No synopsis available

This instruction is defined by:

- D, version >= D@2.2.0

C.194.1. Encoding



C.194.2. Description

No description available.

C.194.3. Access

M	S	U
Always	Always	Always

C.194.4. Decode Variables

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

C.194.5. IDL Operation

C.194.6. Exceptions

This instruction does not generate synchronous exceptions.

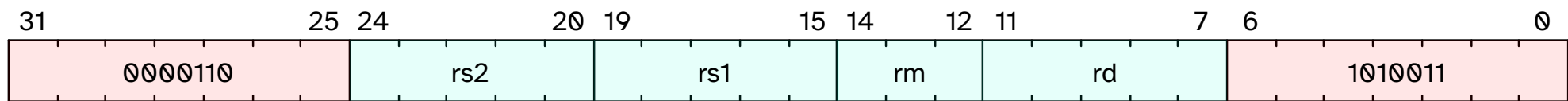
C.195. fsub.h

No synopsis available

This instruction is defined by:

- Zfh, version >= Zfh@1.0.0

C.195.1. Encoding



C.195.2. Description

No description available.

C.195.3. Access

M	S	U
Always	Always	Always

C.195.4. Decode Variables

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

C.195.5. IDL Operation

C.195.6. Exceptions

This instruction does not generate synchronous exceptions.

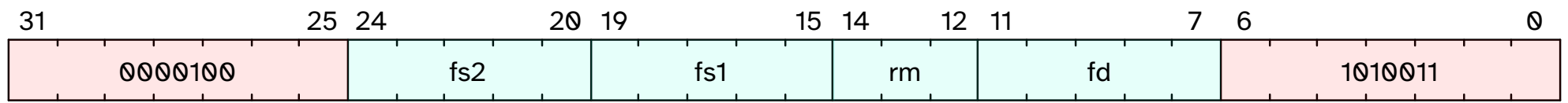
C.196. fsub.s

Single-precision floating-point subtraction

This instruction is defined by:

- F, version >= F@2.2.0

C.196.1. Encoding



C.196.2. Description

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

C.196.3. Access

M	S	U
Always	Always	Always

C.196.4. 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.196.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode mode = rm_to_mode(rm, $encoding);
X[fd] = f32_sub(X[fs1], X[fs2], mode);
```

C.196.6. 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.196.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

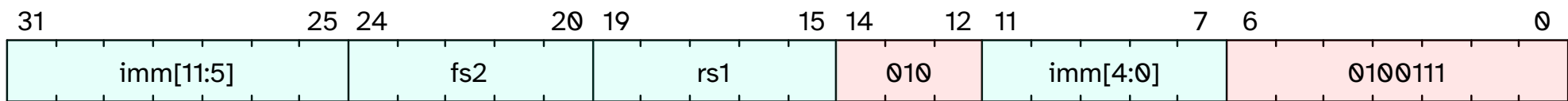
C.197. fsw

Single-precision floating-point store

This instruction is defined by:

- F, version >= F@2.2.0

C.197.1. Encoding



C.197.2. Description

The `fsw` instruction stores a single-precision floating-point value in `fs2` to memory at address `rs1 + imm`.

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

C.197.3. Access

M	S	U
Always	Always	Always

C.197.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

C.197.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $signed(imm);
write_memory<32>(virtual_address, f[fs2][31:0], $encoding);
```

C.197.6. 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_SAM0_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.197.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

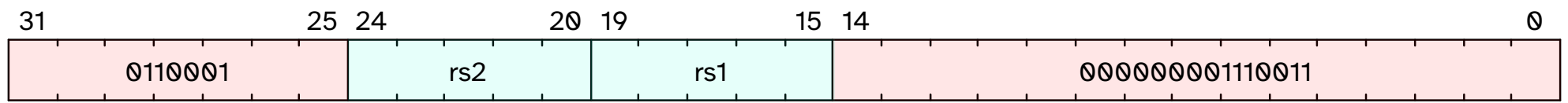
C.198. hfence.gvma

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.198.1. Encoding



C.198.2. Description

No description available.

C.198.3. Access

M	S	U
Always	Always	Always

C.198.4. Decode Variables

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

C.198.5. IDL Operation

C.198.6. Exceptions

This instruction does not generate synchronous exceptions.

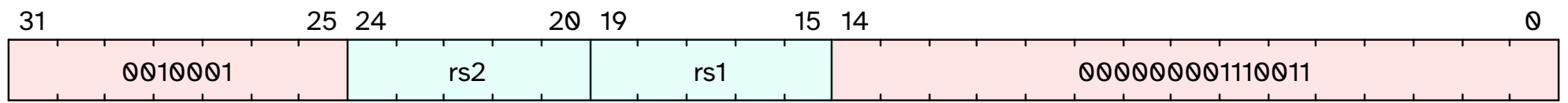
C.199. hfence.vma

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.199.1. Encoding



C.199.2. Description

No description available.

C.199.3. Access

M	S	U
Always	Always	Always

C.199.4. Decode Variables

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

C.199.5. IDL Operation

C.199.6. Exceptions

This instruction does not generate synchronous exceptions.

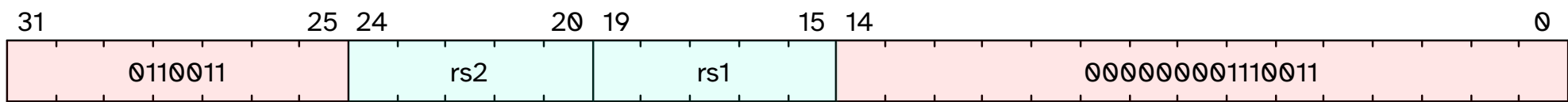
C.200. hinval.gvma

Invalidate cached address translations

This instruction is defined by:

- allOf:
 - Svinval, version >= Svinval@1.0.0
 - H, version >= H@1.0.0

C.200.1. Encoding



C.200.2. Description

hinval.gvma has the same semantics as [sINVAL.VMA](#) except that it combines with [sfence.w.inval](#) and [sfence.inval.ir](#) to replace [hfence.gvma](#) and uses VMID instead of ASID.

C.200.3. Access

M	S	U
Always	Sometimes	Never

C.200.4. Decode Variables

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

C.200.5. IDL Operation

```
XReg gpa = X[rs1];
Bits<VMID_WIDTH> vmid = X[rs2][VMID_WIDTH - 1:0];
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (mstatus.TVM == 1 && mode() == PrivilegeMode::S) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if ((mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU)) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
vma_type.gstage = true;
if ((rs1 == 0) && (rs2 == 0)) {
  vma_type.global = true;
  invalidate_translations(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
  vma_type.single_vmid = true;
  vma_type.vmid = vmid;
  invalidate_translations(vma_type);
} else if ((rs1 != 0) && (rs2 == 0)) {
  if (canonical_gpaddr?(gpa)) {
    vma_type.single_gpaddr = true;
    vma_type.gpaddr = gpa;
    invalidate_translations(vma_type);
  }
} else {
  if (canonical_gpaddr?(gpa)) {
    vma_type.single_vmid = true;
    vma_type.vmid = vmid;
    vma_type.single_gpaddr = true;
    vma_type.gpaddr = gpa;
    invalidate_translations(vma_type);
  }
}
```

C.200.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

DRAFT

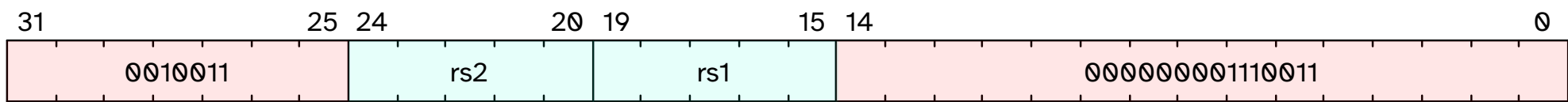
C.201. hinval.vvma

Invalidate cached address translations

This instruction is defined by:

- allOf:
 - Svinval, version >= Svinval@1.0.0
 - H, version >= H@1.0.0

C.201.1. Encoding



C.201.2. Description

hinval.vvma has the same semantics as sinval.vvma except that it combines with sfence.w.inval and sfence.inval.ir to replace hfence.vvma.

C.201.3. Access

M	S	U
Always	Always	Never

C.201.4. Decode Variables

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

C.201.5. IDL Operation

```
XReg vaddr = X[rs1];
Bits<ASID_WIDTH> asid = X[rs2][ASID_WIDTH - 1:0];
Bits<VMID_WIDTH> vmid = hgatp.VMID;
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if ((misa.H == 1) && (mode() == PrivilegeMode::VS || mode() == PrivilegeMode::VU)) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
vma_type.vsmode = true;
vma_type.single_vmid = true;
vma_type.vmid = vmid;
if ((rs1 == 0) && (rs2 == 0)) {
    vma_type.global = true;
    invalidate_translations(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    invalidate_translations(vma_type);
} else if ((rs1 != 0) && (rs2 == 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.201.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

DRAFT

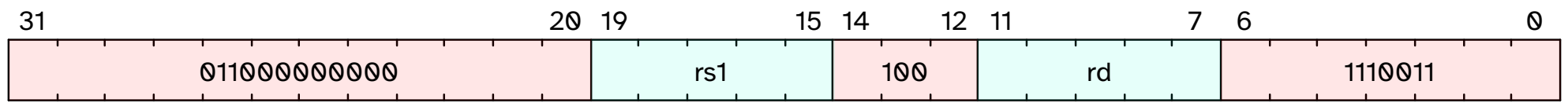
C.202. hlv.b

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.202.1. Encoding



C.202.2. Description

No description available.

C.202.3. Access

M	S	U
Always	Always	Always

C.202.4. Decode Variables

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

C.202.5. IDL Operation

C.202.6. Exceptions

This instruction does not generate synchronous exceptions.

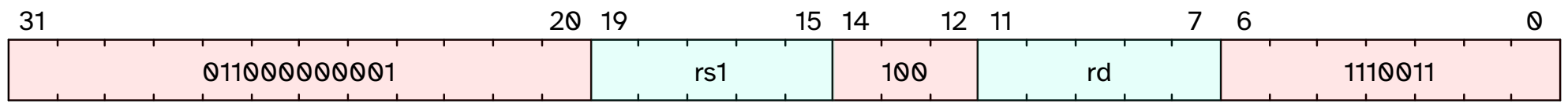
C.203. hlv.bu

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.203.1. Encoding



C.203.2. Description

No description available.

C.203.3. Access

M	S	U
Always	Always	Always

C.203.4. Decode Variables

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

C.203.5. IDL Operation

C.203.6. Exceptions

This instruction does not generate synchronous exceptions.

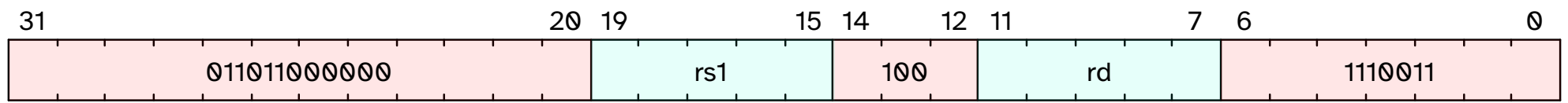
C.204. hlv.d

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.204.1. Encoding



C.204.2. Description

No description available.

C.204.3. Access

M	S	U
Always	Always	Always

C.204.4. Decode Variables

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

C.204.5. IDL Operation

C.204.6. Exceptions

This instruction does not generate synchronous exceptions.

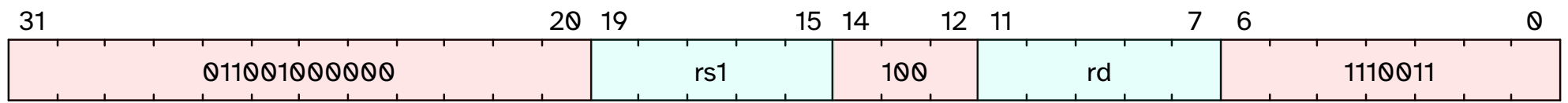
C.205. hlv.h

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.205.1. Encoding



C.205.2. Description

No description available.

C.205.3. Access

M	S	U
Always	Always	Always

C.205.4. Decode Variables

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

C.205.5. IDL Operation

C.205.6. Exceptions

This instruction does not generate synchronous exceptions.

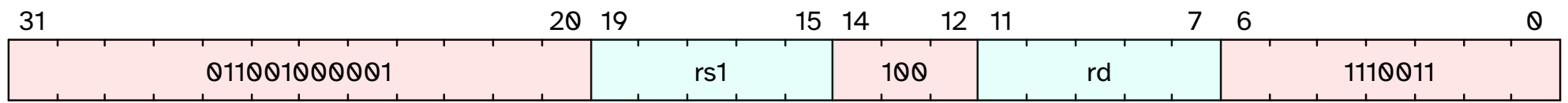
C.206. hlv.hu

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.206.1. Encoding



C.206.2. Description

No description available.

C.206.3. Access

M	S	U
Always	Always	Always

C.206.4. Decode Variables

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

C.206.5. IDL Operation

C.206.6. Exceptions

This instruction does not generate synchronous exceptions.

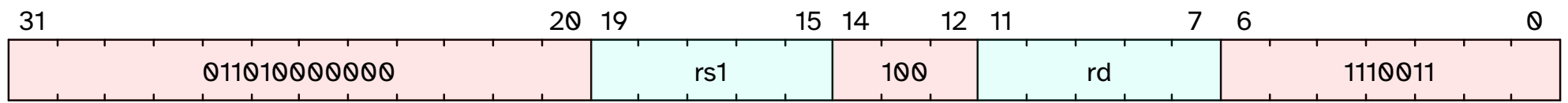
C.207. hlv.w

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.207.1. Encoding



C.207.2. Description

No description available.

C.207.3. Access

M	S	U
Always	Always	Always

C.207.4. Decode Variables

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

C.207.5. IDL Operation

C.207.6. Exceptions

This instruction does not generate synchronous exceptions.

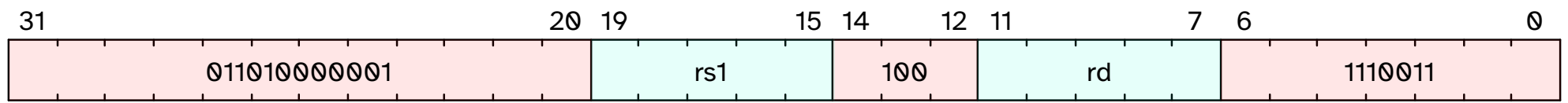
C.208. hlv.wu

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.208.1. Encoding



C.208.2. Description

No description available.

C.208.3. Access

M	S	U
Always	Always	Always

C.208.4. Decode Variables

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

C.208.5. IDL Operation

C.208.6. Exceptions

This instruction does not generate synchronous exceptions.

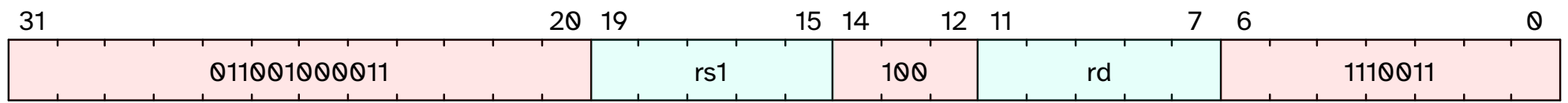
C.209. hlvx.hu

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.209.1. Encoding



C.209.2. Description

No description available.

C.209.3. Access

M	S	U
Always	Always	Always

C.209.4. Decode Variables

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

C.209.5. IDL Operation

C.209.6. Exceptions

This instruction does not generate synchronous exceptions.

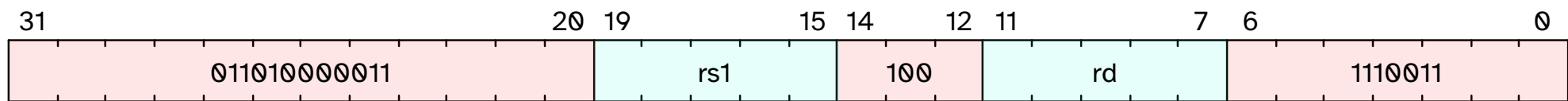
C.210. hlvx.wu

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.210.1. Encoding



C.210.2. Description

No description available.

C.210.3. Access

M	S	U
Always	Always	Always

C.210.4. Decode Variables

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

C.210.5. IDL Operation

C.210.6. Exceptions

This instruction does not generate synchronous exceptions.

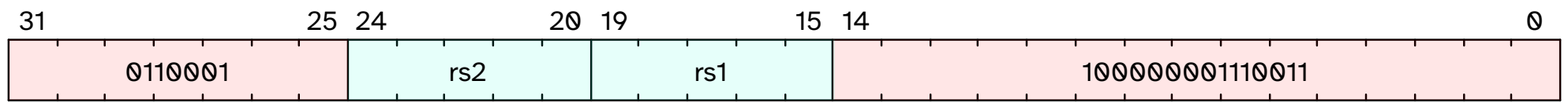
C.211. hsv.b

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.211.1. Encoding



C.211.2. Description

No description available.

C.211.3. Access

M	S	U
Always	Always	Always

C.211.4. Decode Variables

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

C.211.5. IDL Operation

C.211.6. Exceptions

This instruction does not generate synchronous exceptions.

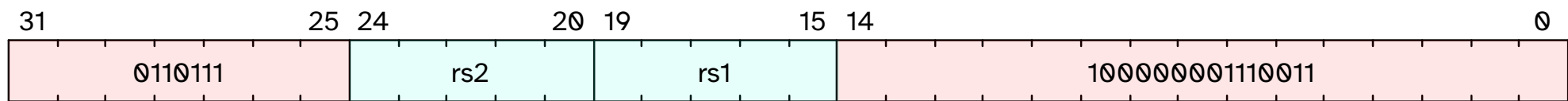
C.212. hsv.d

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.212.1. Encoding



C.212.2. Description

No description available.

C.212.3. Access

M	S	U
Always	Always	Always

C.212.4. Decode Variables

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

C.212.5. IDL Operation

C.212.6. Exceptions

This instruction does not generate synchronous exceptions.

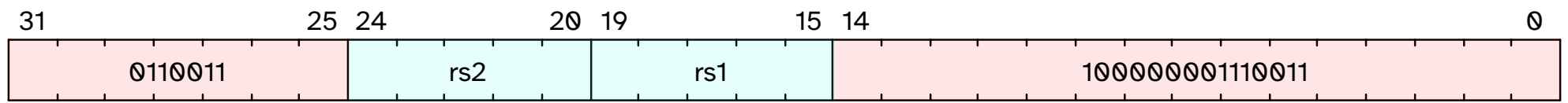
C.213. hsv.h

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.213.1. Encoding



C.213.2. Description

No description available.

C.213.3. Access

M	S	U
Always	Always	Always

C.213.4. Decode Variables

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

C.213.5. IDL Operation

C.213.6. Exceptions

This instruction does not generate synchronous exceptions.

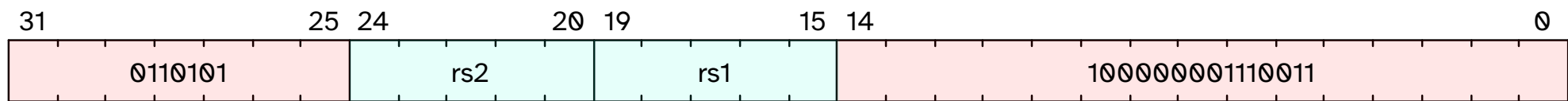
C.214. hsv.w

No synopsis available

This instruction is defined by:

- H, version >= H@1.0.0

C.214.1. Encoding



C.214.2. Description

No description available.

C.214.3. Access

M	S	U
Always	Always	Always

C.214.4. Decode Variables

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

C.214.5. IDL Operation

C.214.6. Exceptions

This instruction does not generate synchronous exceptions.

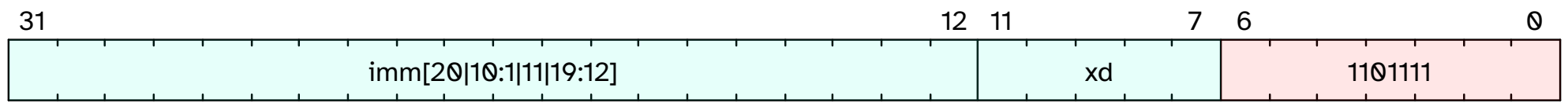
C.215. jal

Jump and link

This instruction is defined by:

- I, version >= I@2.1.0

C.215.1. Encoding



C.215.2. Description

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

C.215.3. Access

M	S	U
Always	Always	Always

C.215.4. 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.215.5. IDL Operation

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

C.215.6. 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.215.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

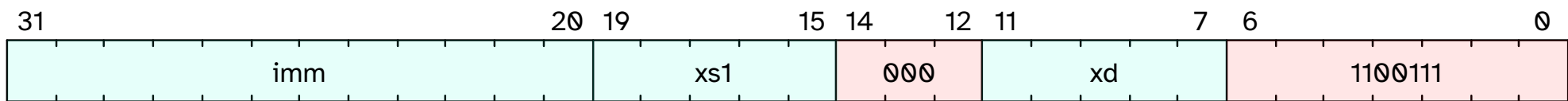
C.216. jalr

Jump and link register

This instruction is defined by:

- I, version >= I@2.1.0

C.216.1. Encoding



C.216.2. 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.216.3. Access

M	S	U
Always	Always	Always

C.216.4. Decode Variables

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

C.216.5. IDL Operation

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

C.216.6. 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.216.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

DRAFT

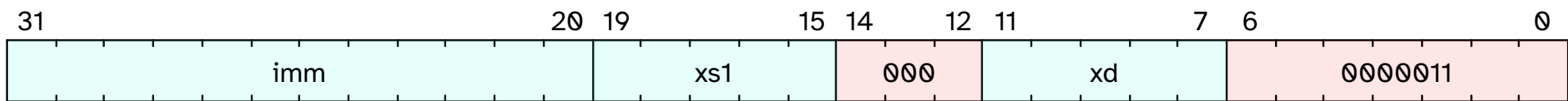
C.217. lb

Load byte

This instruction is defined by:

- I, version >= I@2.1.0

C.217.1. Encoding



C.217.2. 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.217.3. Access

M	S	U
Always	Always	Always

C.217.4. Decode Variables

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

C.217.5. IDL Operation

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

C.217.6. 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.217.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

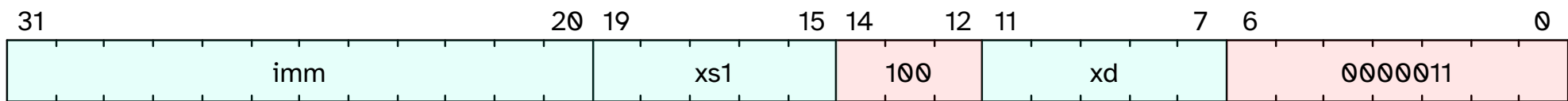
C.218. lbu

Load byte unsigned

This instruction is defined by:

- I, version >= I@2.1.0

C.218.1. Encoding



C.218.2. 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.218.3. Access

M	S	U
Always	Always	Always

C.218.4. Decode Variables

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

C.218.5. IDL Operation

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

C.218.6. 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.218.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

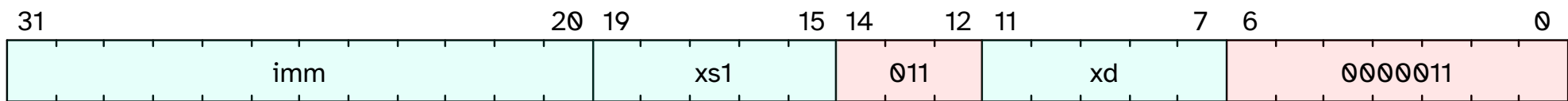
C.219. ld

Load doubleword

This instruction is defined by:

- I, version >= I@2.1.0

C.219.1. Encoding



C.219.2. Description

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

C.219.3. Access

M	S	U
Always	Always	Always

C.219.4. Decode Variables

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

C.219.5. IDL Operation

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

C.219.6. 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.219.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

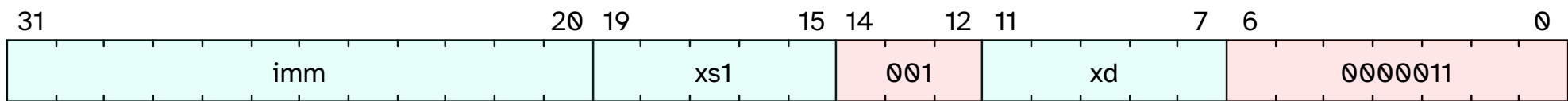
C.220. lh

Load halfword

This instruction is defined by:

- I, version >= I@2.1.0

C.220.1. Encoding



C.220.2. 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.220.3. Access

M	S	U
Always	Always	Always

C.220.4. Decode Variables

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

C.220.5. IDL Operation

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

C.220.6. 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.220.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

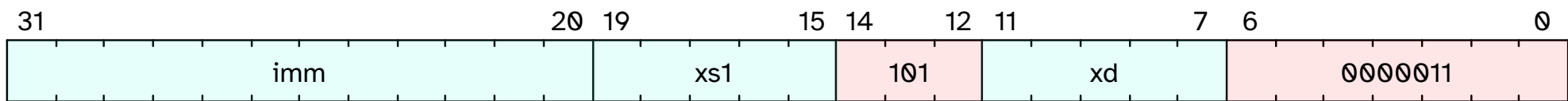
C.221. lhu

Load halfword unsigned

This instruction is defined by:

- I, version >= I@2.1.0

C.221.1. Encoding



C.221.2. 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.221.3. Access

M	S	U
Always	Always	Always

C.221.4. Decode Variables

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

C.221.5. IDL Operation

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

C.221.6. 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.221.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

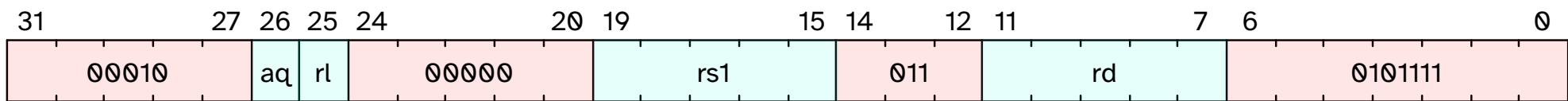
C.222. lr.d

Load reserved doubleword

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

C.222.1. Encoding



C.222.2. Description

Loads a word from the address in rs1, places the value in rd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

The address in rs1 must be 8-byte aligned.

If the address is not naturally aligned, a LoadAddressMisaligned exception or an LoadAccessFault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC’s address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

C.222.3. Access

M	S	U
Always	Always	Always

C.222.4. Decode Variables

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

C.222.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is misaligned");
  }
}
```

```

}
}
X[rd] = load_reserved<32>(virtual_address, aq, rl, $encoding);

```

C.222.6. Sail Operation

```

{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Extensions might perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), Read(Data), width) {
      Ext_DataAddr_Error(e) => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        let aligned : bool =
          /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
           * to treat them as valid here; otherwise we'd need to throw an internal_error.
           */
          match width {
            BYTE    => true,
            HALF    => vaddr[0..0] == 0b0,
            WORD    => vaddr[1..0] == 0b00,
            DOUBLE  => vaddr[2..0] == 0b000
          };
        /* "LR faults like a normal load, even though it's in the AMO major opcode space."
         * - Andrew Waterman, isa-dev, 10 Jul 2018.
         */
        if not(aligned)
        then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
        else match translateAddr(vaddr, Read(Data)) {
          TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) =>
            match (width, sizeof(xlen)) {
              (BYTE, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 1, aq, aq & rl, true), false),
              (HALF, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 2, aq, aq & rl, true), false),
              (WORD, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 4, aq, aq & rl, true), false),
              (DOUBLE, 64) => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 8, aq, aq & rl, true), false),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            }
        }
      }
    }
  }
} else {
  handle_illegal();
  RETIRE_FAIL
}
}

```

C.222.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

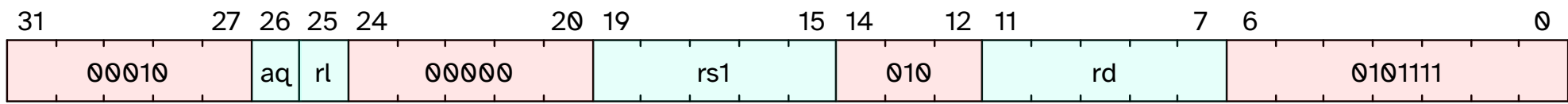
C.223. lr.w

Load reserved word

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

C.223.1. Encoding



C.223.2. Description

Loads a word from the address in rs1, places the sign-extended value in rd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

<%- if MXLEN == 64 -%> The 32-bit load result is sign-extended to 64-bits. <%- end -%>

The address in rs1 must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words).

If the address is not naturally aligned, a LoadAddressMisaligned exception or an LoadAccessFault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC’s address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

C.223.3. Access

M	S	U
Always	Always	Always

C.223.4. Decode Variables

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

C.223.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
```

```

    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address
    is misaligned");
}
}
XReg load_value = load_reserved<32>(virtual_address, aq, rl, $encoding);
if (xlen() == 64) {
    X[rd] = load_value;
} else {
    X[rd] = sext(load_value[31:0], 32);
}

```

C.223.6. Sail Operation

```

{
    if extension("A") then {
        /* Get the address, X(rs1) (no offset).
        * Extensions might perform additional checks on address validity.
        */
        match ext_data_get_addr(rs1, zeros(), Read(Data), width) {
            Ext_DataAddr_Error(e) => { ext_handle_data_check_error(e); RETIRE_FAIL },
            Ext_DataAddr_OK(vaddr) => {
                let aligned : bool =
                    /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
                    * to treat them as valid here; otherwise we'd need to throw an internal_error.
                    */
                    match width {
                        BYTE    => true,
                        HALF    => vaddr[0..0] == 0b0,
                        WORD    => vaddr[1..0] == 0b00,
                        DOUBLE  => vaddr[2..0] == 0b000
                    };
                /* "LR faults like a normal load, even though it's in the AMO major opcode space."
                * - Andrew Waterman, isa-dev, 10 Jul 2018.
                */
                if not(aligned)
                then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
                else match translateAddr(vaddr, Read(Data)) {
                    TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                    TR_Address(addr, _) =>
                        match (width, sizeof(xlen)) {
                            (BYTE, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 1, aq, aq & rl, true), false),
                            (HALF, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 2, aq, aq & rl, true), false),
                            (WORD, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 4, aq, aq & rl, true), false),
                            (DOUBLE, 64) => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 8, aq, aq & rl, true), false),
                            -             => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                        }
                    -
                }
            }
        }
    }
} else {
    handle_illegal();
    RETIRE_FAIL
}
}

```

C.223.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

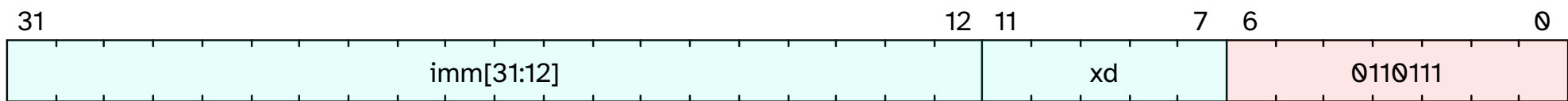
C.224. lui

Load upper immediate

This instruction is defined by:

- I, version >= I@2.1.0

C.224.1. Encoding



C.224.2. Description

Load the zero-extended imm into xd.

C.224.3. Access

M	S	U
Always	Always	Always

C.224.4. Decode Variables

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

C.224.5. IDL Operation

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

C.224.6. Sail Operation

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

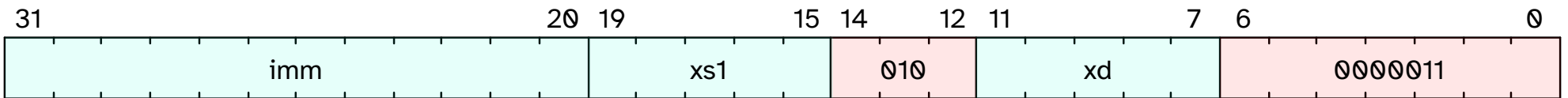
C.224.7. Exceptions

This instruction does not generate synchronous exceptions.

C.225. lw**Load word**

This instruction is defined by:

- I, version $\geq$ I@2.1.0

C.225.1. Encoding**C.225.2. 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.225.3. Access

M	S	U
Always	Always	Always

C.225.4. Decode Variables

```

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

```

C.225.5. IDL Operation

```

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

```

C.225.6. 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.225.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

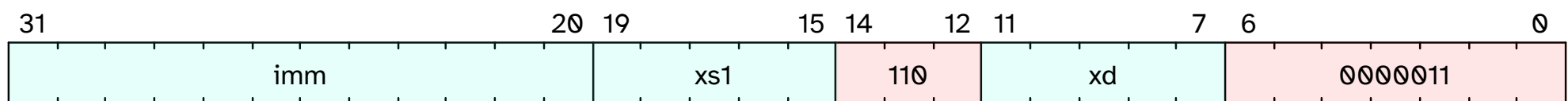
C.226. lwu

Load word unsigned

This instruction is defined by:

- I, version $\geq$ I@2.1.0

C.226.1. Encoding



C.226.2. Description

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

C.226.3. Access

M	S	U
Always	Always	Always

C.226.4. Decode Variables

```

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

```

C.226.5. IDL Operation

```

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

```

C.226.6. 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.226.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

DRAFT

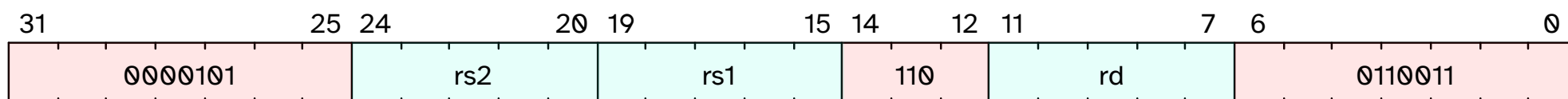
C.227. max

Maximum

This instruction is defined by:

- Zbb, version $\geq$ Zbb@1.0.0

C.227.1. Encoding



C.227.2. Description

This instruction 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.227.3. Access

M	S	U
Always	Always	Always

C.227.4. Decode Variables

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

C.227.5. IDL Operation

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

C.227.6. 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.227.7. Exceptions

This instruction may result in the following synchronous exceptions:

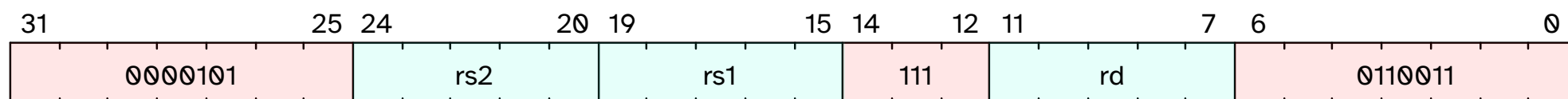
- IllegalInstruction

DRAFT

C.228. maxu**Unsigned maximum**

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.228.1. Encoding**C.228.2. Description**

This instruction returns the larger of two unsigned integers.

C.228.3. Access

M	S	U
Always	Always	Always

C.228.4. Decode Variables

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

C.228.5. IDL Operation

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

C.228.6. 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.228.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

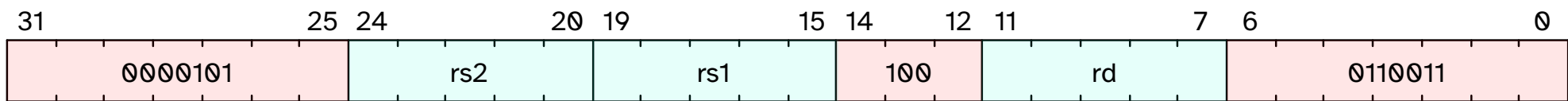
C.229. min

Minimum

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.229.1. Encoding



C.229.2. Description

This instruction returns the smaller of two signed integers.

C.229.3. Access

M	S	U
Always	Always	Always

C.229.4. Decode Variables

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

C.229.5. IDL Operation

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

C.229.6. 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.229.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

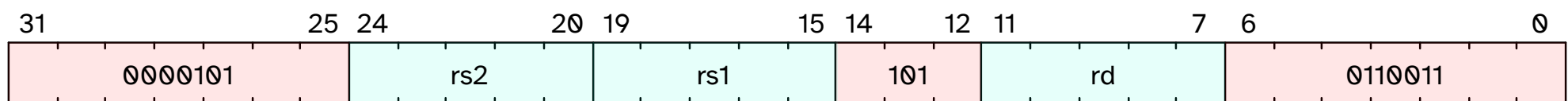
C.230. minu

Unsigned minimum

This instruction is defined by:

- Zbb, version $\geq$ Zbb@1.0.0

C.230.1. Encoding



C.230.2. Description

This instruction returns the smaller of two unsigned integers.

C.230.3. Access

M	S	U
Always	Always	Always

C.230.4. Decode Variables

```

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

```

C.230.5. IDL Operation

```

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

```

C.230.6. Sail Operation

```

{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISC_V_ANDN => rs1_val & ~(rs2_val),
    RISC_V_ORN  => rs1_val | ~(rs2_val),
    RISC_V_XNOR => ~(rs1_val ^ rs2_val),
    RISC_V_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val), signed(rs2_val))),
    RISC_V_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISC_V_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val), signed(rs2_val))),
    RISC_V_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISC_V_ROL  => if sizeof(xlen) == 32
                    then rs1_val <<< rs2_val[4..0]
                    else rs1_val <<< rs2_val[5..0],
    RISC_V_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.230.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

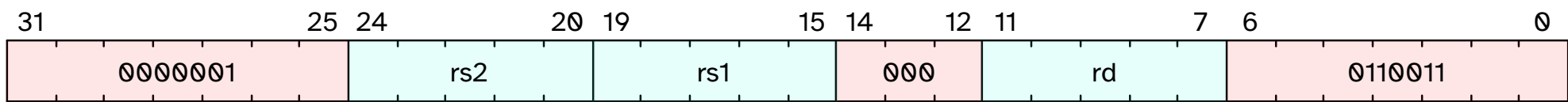
C.231. mul

Signed multiply

This instruction is defined by:

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

C.231.1. Encoding



C.231.2. Description

MUL performs an XLEN-bitxXLEN-bit multiplication of rs1 by rs2 and places the lower XLEN bits in the destination register. Any overflow is thrown away.



If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2 (source register specifiers must be in same order and rdh cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

C.231.3. Access

M	S	U
Always	Always	Always

C.231.4. Decode Variables

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

C.231.5. IDL Operation

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

C.231.6. 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.231.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

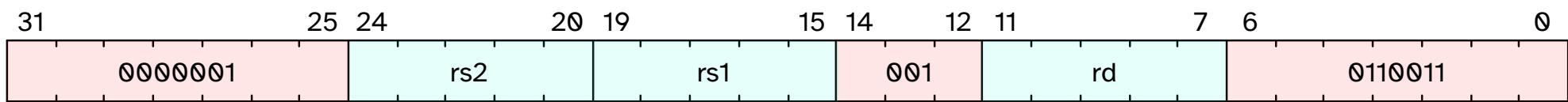
C.232. mulh

Signed multiply high

This instruction is defined by:

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

C.232.1. Encoding



C.232.2. Description

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

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

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

Microarchitectures may look for that sequence and fuse the operations.

C.232.3. Access

M	S	U
Always	Always	Always

C.232.4. Decode Variables

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

C.232.5. IDL Operation

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

C.232.6. 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.232.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

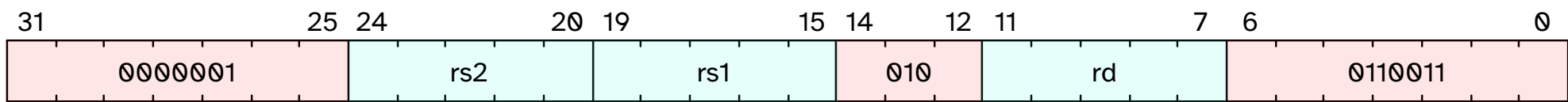
C.233. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

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

C.233.1. Encoding



C.233.2. Description

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

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

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

Microarchitectures may look for that sequence and fuse the operations.

C.233.3. Access

M	S	U
Always	Always	Always

C.233.4. Decode Variables

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

C.233.5. IDL Operation

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

C.233.6. 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.233.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

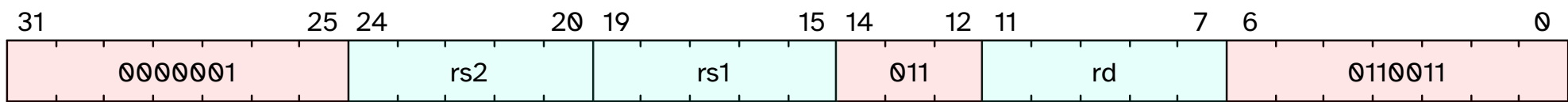
C.234. mulhu

Unsigned multiply high

This instruction is defined by:

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

C.234.1. Encoding



C.234.2. Description

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

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

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

Microarchitectures may look for that sequence and fuse the operations.

C.234.3. Access

M	S	U
Always	Always	Always

C.234.4. Decode Variables

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

C.234.5. IDL Operation

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

C.234.6. 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.234.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

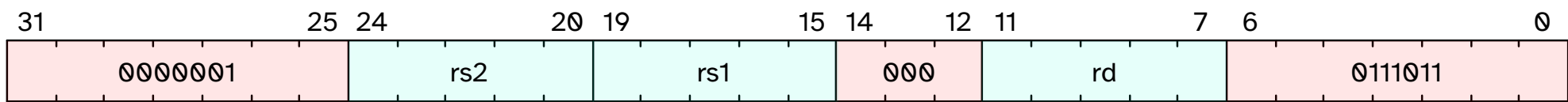
C.235. mulw

Signed 32-bit multiply

This instruction is defined by:

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

C.235.1. Encoding



C.235.2. Description

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

Any overflow is thrown away.

i

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

C.235.3. Access

M	S	U
Always	Always	Always

C.235.4. Decode Variables

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

C.235.5. IDL Operation

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

C.235.6. Sail Operation

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

C.235.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

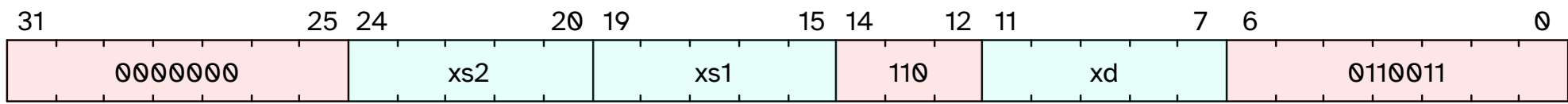
C.236. or

Or

This instruction is defined by:

- I, version >= I@2.1.0

C.236.1. Encoding



C.236.2. Description

Or xs1 with xs2, and store the result in xd

C.236.3. Access

M	S	U
Always	Always	Always

C.236.4. Decode Variables

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

C.236.5. IDL Operation

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

C.236.6. 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.236.7. Exceptions

This instruction does not generate synchronous exceptions.

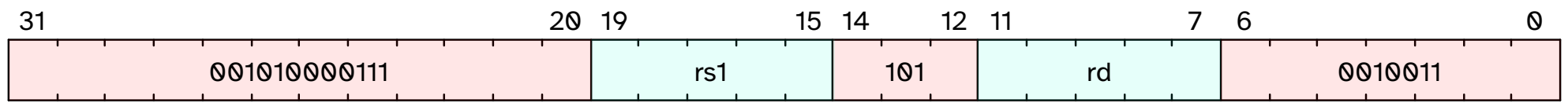
C.237. orc.b

Bitware OR-combine, byte granule

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.237.1. Encoding



C.237.2. Description

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

C.237.3. Access

M	S	U
Always	Always	Always

C.237.4. Decode Variables

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

C.237.5. IDL Operation

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

C.237.6. 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.237.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

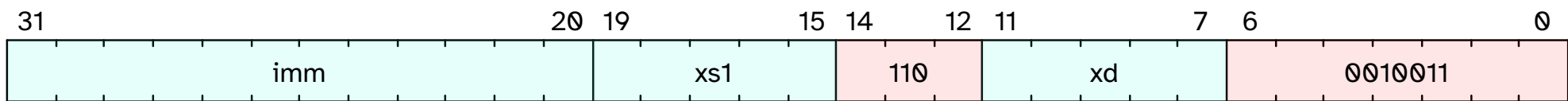
C.238. ori

Or immediate

This instruction is defined by:

- I, version >= I@2.1.0

C.238.1. Encoding



C.238.2. Description

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

C.238.3. Access

M	S	U
Always	Always	Always

C.238.4. Decode Variables

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

C.238.5. IDL Operation

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

C.238.6. 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.238.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

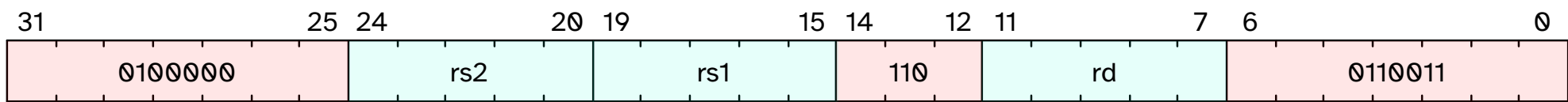
C.239. orn

OR with inverted operand

This instruction is defined by:

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

C.239.1. Encoding



C.239.2. Description

This instruction performs the bitwise logical OR operation between rs1 and the bitwise inversion of rs2.

C.239.3. Access

M	S	U
Always	Always	Always

C.239.4. Decode Variables

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

C.239.5. IDL Operation

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

C.239.6. 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.239.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

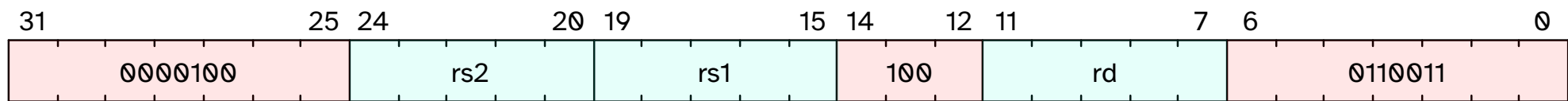
C.240. pack

No synopsis available

This instruction is defined by:

- Zbkb, version >= Zbkb@1.0.0

C.240.1. Encoding



C.240.2. Description

No description available.

C.240.3. Access

M	S	U
Always	Always	Always

C.240.4. Decode Variables

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

C.240.5. IDL Operation

C.240.6. Exceptions

This instruction does not generate synchronous exceptions.

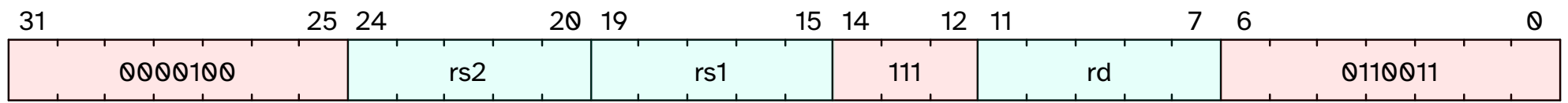
C.241. packh

No synopsis available

This instruction is defined by:

- Zbkb, version >= Zbkb@1.0.0

C.241.1. Encoding



C.241.2. Description

No description available.

C.241.3. Access

M	S	U
Always	Always	Always

C.241.4. Decode Variables

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

C.241.5. IDL Operation

C.241.6. Exceptions

This instruction does not generate synchronous exceptions.

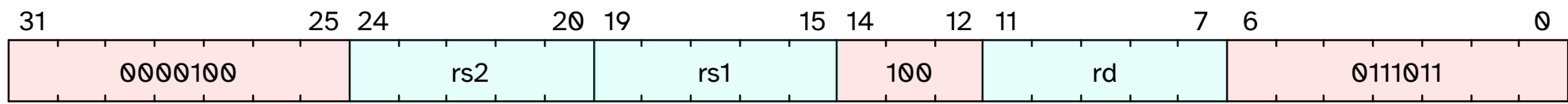
C.242. packw

No synopsis available

This instruction is defined by:

- Zbkb, version >= Zbkb@1.0.0

C.242.1. Encoding



C.242.2. Description

No description available.

C.242.3. Access

M	S	U
Always	Always	Always

C.242.4. Decode Variables

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

C.242.5. IDL Operation

C.242.6. Exceptions

This instruction does not generate synchronous exceptions.

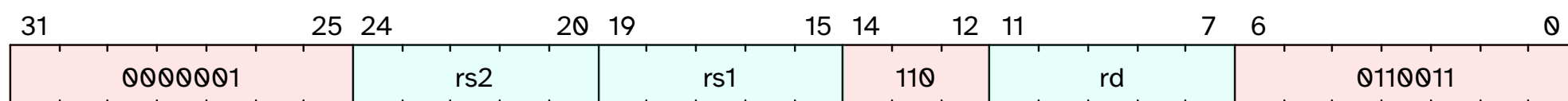
C.243. rem

Signed remainder

This instruction is defined by:

- M, version $\geq$ M@2.0.0

C.243.1. Encoding



C.243.2. Description

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

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

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

C.243.3. Access

M	S	U
Always	Always	Always

C.243.4. Decode Variables

```

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

```

C.243.5. IDL Operation

```

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

```

C.243.6. 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.243.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

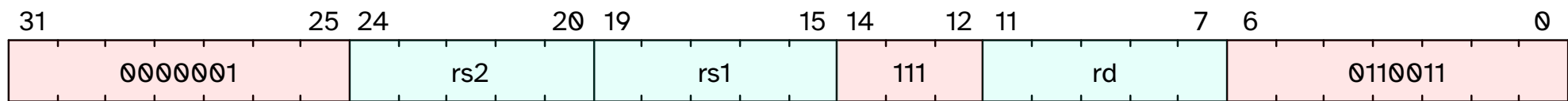
C.244. remu

Unsigned remainder

This instruction is defined by:

- M, version >= M@2.0.0

C.244.1. Encoding



C.244.2. Description

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

C.244.3. Access

M	S	U
Always	Always	Always

C.244.4. Decode Variables

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

C.244.5. IDL Operation

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

C.244.6. 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.244.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

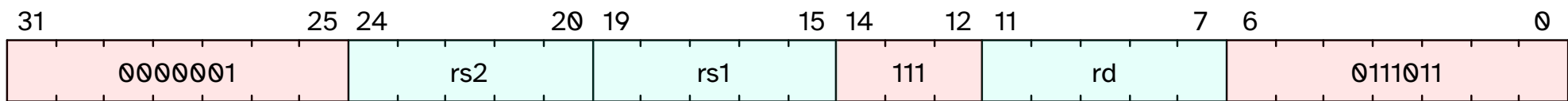
C.245. remuw

Unsigned 32-bit remainder

This instruction is defined by:

- M, version >= M@2.0.0

C.245.1. Encoding



C.245.2. Description

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

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

C.245.3. Access

M	S	U
Always	Always	Always

C.245.4. Decode Variables

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

C.245.5. IDL Operation

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

C.245.6. Sail Operation

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

C.245.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

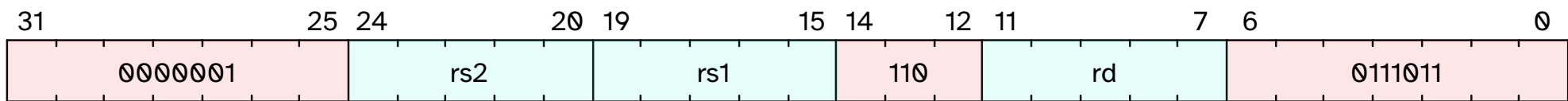
C.246. remw

Signed 32-bit remainder

This instruction is defined by:

- M, version >= M@2.0.0

C.246.1. Encoding



C.246.2. Description

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

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

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

C.246.3. Access

M	S	U
Always	Always	Always

C.246.4. Decode Variables

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

C.246.5. IDL Operation

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

C.246.6. Sail Operation

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

```
}
```

C.246.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

C.247. rev8

Byte-reverse register (RV64 encoding)

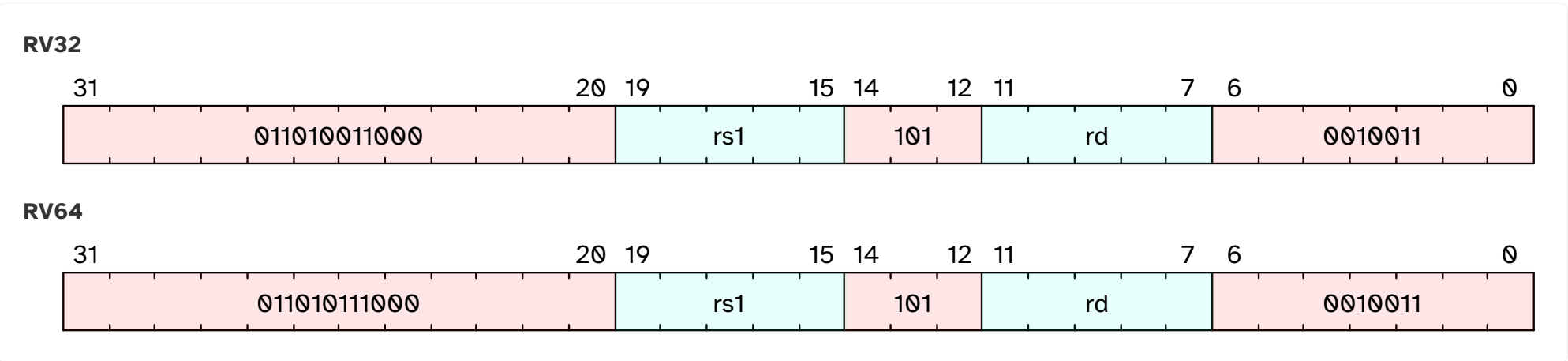
This instruction is defined by:

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

C.247.1. Encoding



This instruction has different encodings in RV32 and RV64.



C.247.2. Description

This instruction reverses the order of the bytes in rs1.



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



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

C.247.3. Access

M	S	U
Always	Always	Always

C.247.4. Decode Variables

RV32

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

RV64

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

C.247.5. IDL Operation

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

C.247.6. 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.247.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

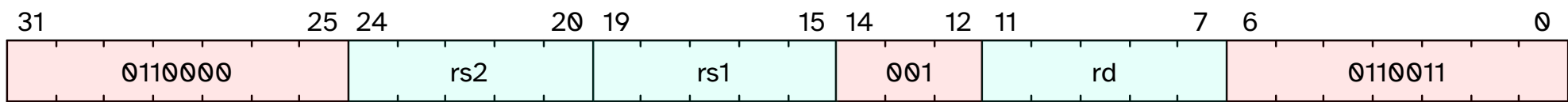
C.248. rol

Rotate left (Register)

This instruction is defined by:

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

C.248.1. Encoding



C.248.2. Description

This instruction performs a rotate left of rs1 by the amount in least-significant log2(XLEN) bits of rs2.

C.248.3. Access

M	S	U
Always	Always	Always

C.248.4. Decode Variables

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

C.248.5. IDL Operation

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

C.248.6. 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.248.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

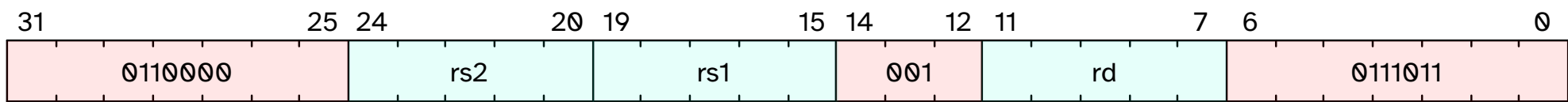
C.249. rolw

Rotate left word (Register)

This instruction is defined by:

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

C.249.1. Encoding



C.249.2. Description

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

C.249.3. Access

M	S	U
Always	Always	Always

C.249.4. Decode Variables

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

C.249.5. IDL Operation

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

C.249.6. Sail Operation

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

C.249.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

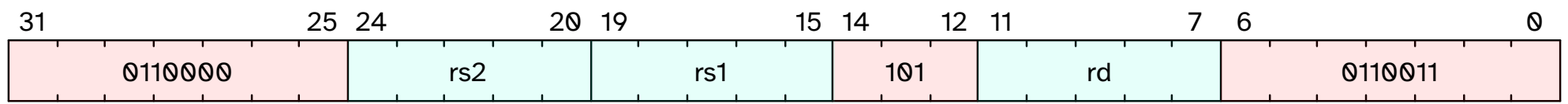
C.250. ror

Rotate right (Register)

This instruction is defined by:

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

C.250.1. Encoding



C.250.2. Description

This instruction performs a rotate right of rs1 by the amount in least-significant log2(XLEN) bits of rs2.

C.250.3. Access

M	S	U
Always	Always	Always

C.250.4. Decode Variables

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

C.250.5. IDL Operation

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

C.250.6. 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.250.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

C.251. rori

Rotate right (Immediate)

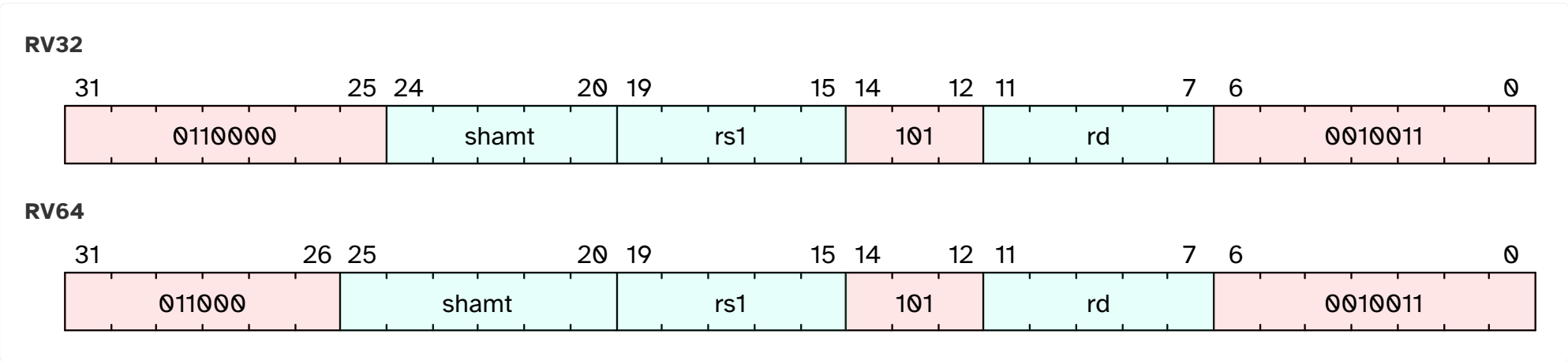
This instruction is defined by:

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

C.251.1. Encoding



This instruction has different encodings in RV32 and RV64.



C.251.2. Description

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

C.251.3. Access

M	S	U
Always	Always	Always

C.251.4. Decode Variables

RV32

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

RV64

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

C.251.5. IDL Operation

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

C.251.6. 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.251.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

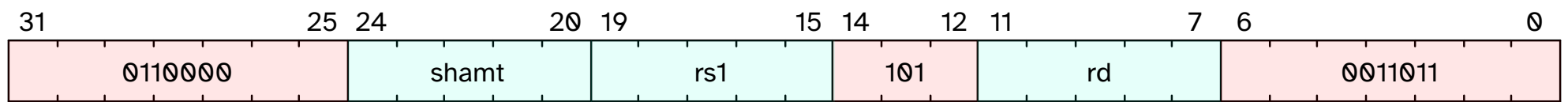
C.252. roriw

Rotate right word (Immediate)

This instruction is defined by:

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

C.252.1. Encoding



C.252.2. Description

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

C.252.3. Access

M	S	U
Always	Always	Always

C.252.4. Decode Variables

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

C.252.5. IDL Operation

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

C.252.6. Sail Operation

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

C.252.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

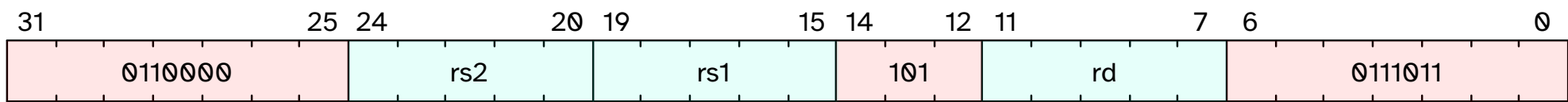
C.253. rorw

Rotate right word (Register)

This instruction is defined by:

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

C.253.1. Encoding



C.253.2. Description

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

C.253.3. Access

M	S	U
Always	Always	Always

C.253.4. Decode Variables

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

C.253.5. IDL Operation

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

C.253.6. Sail Operation

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

C.253.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

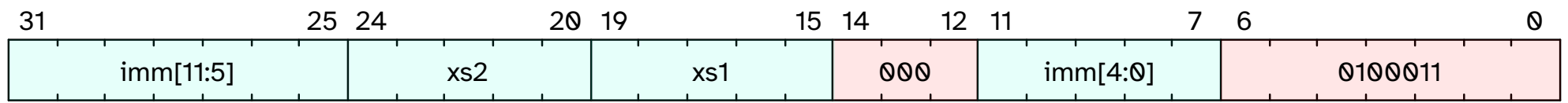
C.254. sb

Store byte

This instruction is defined by:

- I, version >= I@2.1.0

C.254.1. Encoding



C.254.2. Description

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

C.254.3. Access

M	S	U
Always	Always	Always

C.254.4. Decode Variables

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

C.254.5. IDL Operation

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

C.254.6. 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.254.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

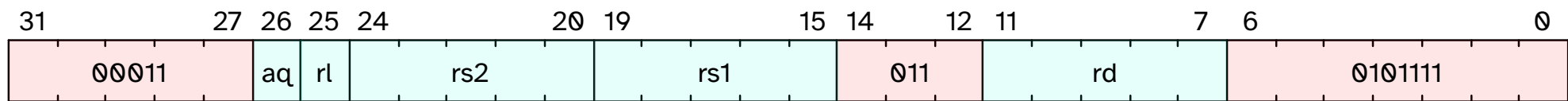
C.255. sc.d

Store conditional doubleword

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

C.255.1. Encoding



C.255.2. Description

sc.d conditionally writes a doubleword in rs2 to the address in rs1: the sc.d succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.d succeeds, the instruction writes the doubleword in rs2 to memory, and it writes zero to rd. If the sc.d fails, the instruction does not write to memory, and it writes a nonzero value to rd. For the purposes of memory protection, a failed sc.d may be treated like a store. Regardless of success or failure, executing an sc.d instruction invalidates any reservation held by this hart.

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in rs1 must be naturally aligned to the size of the operand (i.e., eight-byte aligned). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.



Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC’s address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.



The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- during a preemptive context switch, and
- if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.

The invalidation of a hart’s reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.



The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and

after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.

If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

C.255.3. Access

M	S	U
Always	Always	Always

C.255.4. Decode Variables

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

C.255.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is misaligned");
  }
}
Boolean success = store_conditional<64>(virtual_address, value, aq, rl, $encoding);
X[rd] = success ? 0 : 1;
```

C.255.6. Sail Operation

```
{
  if speculate_conditional () == false then {
    /* should only happen in rmem
     * rmem: allow SC to fail very early
     */
    X(rd) = zero_extend(0b1); RETIRE_SUCCESS
  } else {
    if extension("A") then {
      /* normal non-rmem case
       * rmem: SC is allowed to succeed (but might fail later)
       */
      /* Get the address, X(rs1) (no offset).
       * Extensions might perform additional checks on address validity.
       */
      match ext_data_get_addr(rs1, zeros(), Write(Data), width) {
        Ext_DataAddr_Error(e) => { ext_handle_data_check_error(e); RETIRE_FAIL },
        Ext_DataAddr_OK(vaddr) => {
          let aligned : bool =
            /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
             * to treat them as valid here; otherwise we'd need to throw an internal_error.
             */
            match width {
              BYTE   => true,
              HALF   => vaddr[0..0] == 0b0,
              WORD    => vaddr[1..0] == 0b00,
```

This instruction may result in the following synchronous exceptions:

- RVA22 Profile Release | © RISC-V International

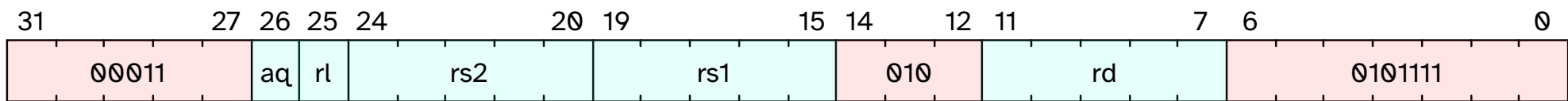
C.256. sc.w

Store conditional word

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

C.256.1. Encoding



C.256.2. Description

sc.w conditionally writes a word in rs2 to the address in rs1: the sc.w succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.w succeeds, the instruction writes the word in rs2 to memory, and it writes zero to rd. If the sc.w fails, the instruction does not write to memory, and it writes a nonzero value to rd. For the purposes of memory protection, a failed sc.w may be treated like a store. Regardless of success or failure, executing an sc.w instruction invalidates any reservation held by this hart.

<%- if MXLEN == 64 -%>

If a value other than 0 or 1 is defined as a result for sc.w, the value will before sign-extended into rd. <%- end -%>

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in rs1 must be naturally aligned to the size of the operand (i.e., eight-byte aligned for doublewords and four-byte aligned for words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC’s address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- during a preemptive context switch, and
- if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.

The invalidation of a hart’s reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given

release semantics by by setting the *rl* bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the *aq* bit on the LR instruction, and setting the *rl* bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.

If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

C.256.3. Access

M	S	U
Always	Always	Always

C.256.4. Decode Variables

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

C.256.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is misaligned");
  }
}
Boolean success = store_conditional<32>(virtual_address, value, aq, rl, $encoding);
X[rd] = success ? 0 : 1;
```

C.256.6. Sail Operation

```
{
  if speculate_conditional () == false then {
    /* should only happen in rmem
     * rmem: allow SC to fail very early
     */
    X(rd) = zero_extend(0b1); RETIRE_SUCCESS
  } else {
    if extension("A") then {
      /* normal non-rmem case
       * rmem: SC is allowed to succeed (but might fail later)
       */
      /* Get the address, X(rs1) (no offset).
       * Extensions might perform additional checks on address validity.
       */
      match ext_data_get_addr(rs1, zeros(), Write(Data), width) {
        Ext_DataAddr_Error(e) => { ext_handle_data_check_error(e); RETIRE_FAIL },
        Ext_DataAddr_OK(vaddr) => {
          let aligned : bool =
            /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
             * to treat them as valid here; otherwise we'd need to throw an internal_error.
             */
            match width {
```



```

        BYTE    => true,
        HALF    => vaddr[0..0] == 0b0,
        WORD     => vaddr[1..0] == 0b00,
        DOUBLE   => vaddr[2..0] == 0b000
    };
    if not(aligned)
    then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
    else {
        if match_reservation(vaddr) == false then {
            /* cannot happen in rmem */
            X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS
        } else {
            match translateAddr(vaddr, Write(Data)) { /* Write and ReadWrite are equivalent here:
                                                         * both result in a SAMO exception */
            TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            TR_Address(addr, _) => {
                let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
                    (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
                    (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
                };
                match (eares) {
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                    MemValue(_) => {
                        rs2_val = X(rs2);
                        let res : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                            (BYTE, _)    => mem_write_value(addr, 1, rs2_val[7..0], aq & rl, rl, true),
                            (HALF, _)    => mem_write_value(addr, 2, rs2_val[15..0], aq & rl, rl, true),
                            (WORD, _)    => mem_write_value(addr, 4, rs2_val[31..0], aq & rl, rl, true),
                            (DOUBLE, 64) => mem_write_value(addr, 8, rs2_val, aq & rl, rl, true),
                            _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
                        };
                        match (res) {
                            MemValue(true)  => { X(rd) = zero_extend(0b0); cancel_reservation(); RETIRE_SUCCESS },
                            MemValue(false) => { X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS },
                            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                        }
                    }
                }
            }
        }
    }
} else {
    handle_illegal();
    RETIRE_FAIL
}
}
}

```

C.256.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

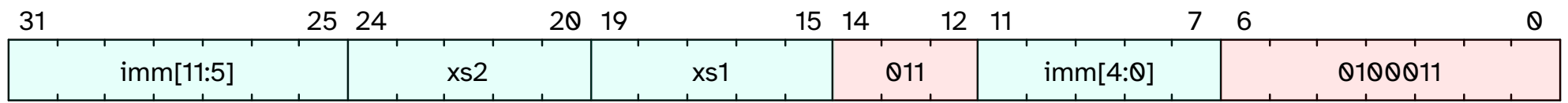
C.257. sd

Store doubleword

This instruction is defined by:

- I, version >= I@2.1.0

C.257.1. Encoding



C.257.2. Description

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

C.257.3. Access

M	S	U
Always	Always	Always

C.257.4. Decode Variables

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

C.257.5. IDL Operation

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

C.257.6. 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.257.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

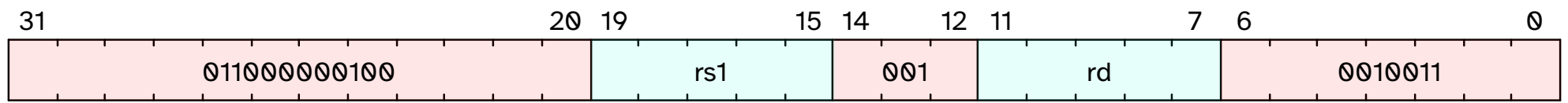
C.258. sext.b

Sign-extend byte

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.258.1. Encoding



C.258.2. Description

This instruction 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.258.3. Access

M	S	U
Always	Always	Always

C.258.4. Decode Variables

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

C.258.5. IDL Operation

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

C.258.6. 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.258.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

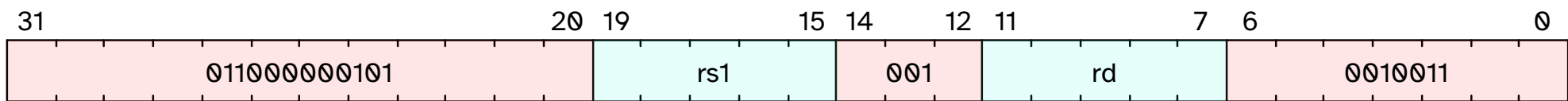
C.259. sext.h

Sign-extend halfword

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

C.259.1. Encoding



C.259.2. Description

This instruction 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.259.3. Access

M	S	U
Always	Always	Always

C.259.4. Decode Variables

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

C.259.5. IDL Operation

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

C.259.6. 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.259.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

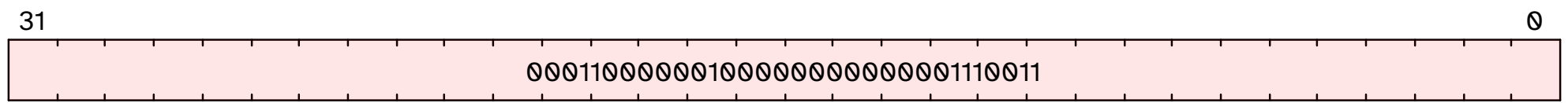
C.260. sfence.inval.ir

Order implicit page table reads after invalidation

This instruction is defined by:

- Svinval, version >= Svinval@1.0.0

C.260.1. Encoding



C.260.2. 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.260.3. Access

M	S	U
Always	Sometimes	Never

C.260.4. Decode Variables

C.260.5. IDL Operation

```
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (misa.H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
vma_type.global = true;
vma_type.smode = true;
if (misa.H == 1) {
    vma_type.vsmode = true;
    vma_type.gstage = true;
}
order_pgtbl_reads_after_vma_fence(vma_type);
```

C.260.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.

In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with rs1=x0. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.

Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.

This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

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

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

The following common situations typically require executing an SFENCE.VMA instruction:

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

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



A future extension could redefine ASIDs to be global across the SEE, enabling such options as shared translation caches and hardware support for broadcast TLB shutdown. However, as Oses have evolved to significantly reduce the scope of TLB shutdowns 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.261.3. Access

M	S	U
Always	Always	Never

C.261.4. Decode Variables

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

C.261.5. IDL Operation

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



```

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

```

C.261.6. Sail Operation

```

{
    let addr : option(xlenbits) = if rs1 == 0b000000 then None() else Some(X(rs1));
    let asid : option(xlenbits) = if rs2 == 0b000000 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.261.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

DRAFT

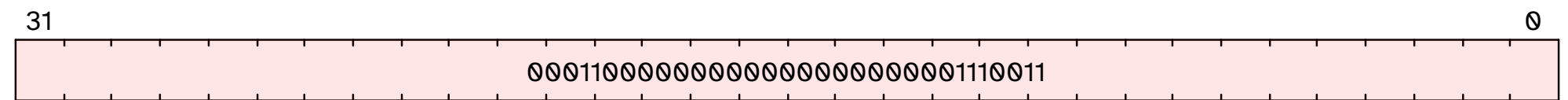
C.262. sfence.w inval

Order writes before sfence

This instruction is defined by:

- Svinval, version $\geq$ Svinval@1.0.0

C.262.1. Encoding



C.262.2. Description

The `s fence.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.262.3. Access

M	S	U
Always	Sometimes	Never

C.262.4. Decode Variables

C.262.5. IDL Operation

```

if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (misa.H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
vma_type.global = true;
vma_type.smode = true;
if (misa.H == 1) {
    vma_type.vsmode = true;
    vma_type.gstage = true;
}
order_pgtbl_writes_before_vma_fence(vma_type);

```

C.262.6. Exceptions

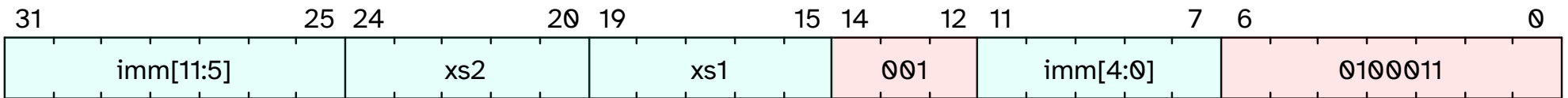
This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

C.263. sh**Store halfword**

This instruction is defined by:

- I, version >= I@2.1.0

C.263.1. Encoding**C.263.2. Description**

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

C.263.3. Access

M	S	U
Always	Always	Always

C.263.4. Decode Variables

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

C.263.5. IDL Operation

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

C.263.6. 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.263.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

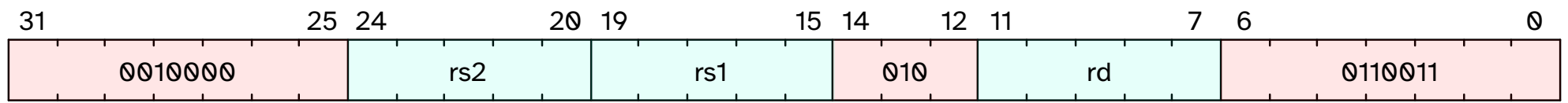
C.264. sh1add

Shift left by 1 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.264.1. Encoding



C.264.2. Description

This instruction shifts rs1 to the left by 1 bit and adds it to rs2.

C.264.3. Access

M	S	U
Always	Always	Always

C.264.4. Decode Variables

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

C.264.5. IDL Operation

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

C.264.6. 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.264.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

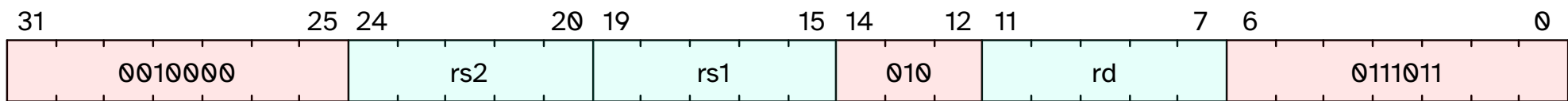
C.265. sh1add.uw

Shift unsigned word left by 1 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.265.1. Encoding



C.265.2. Description

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

C.265.3. Access

M	S	U
Always	Always	Always

C.265.4. Decode Variables

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

C.265.5. IDL Operation

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

C.265.6. Sail Operation

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

C.265.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

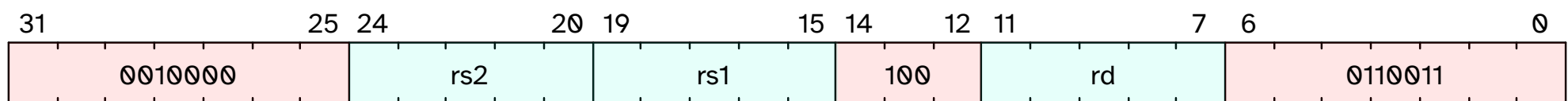
C.266. sh2add

Shift left by 2 and add

This instruction is defined by:

- Zba, version $\geq$ Zba@1.0.0

C.266.1. Encoding



C.266.2. Description

This instruction shifts rs1 to the left by 2 places and adds it to rs2.

C.266.3. Access

M	S	U
Always	Always	Always

C.266.4. Decode Variables

```

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

```

C.266.5. IDL Operation

```

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

```

C.266.6. Sail Operation

```

{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISC_V_SH1ADD => 0b01,
    RISC_V_SH2ADD => 0b10,
    RISC_V_SH3ADD => 0b11
  };
  let result : xlenbits = (rs1_val << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}

```

C.266.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

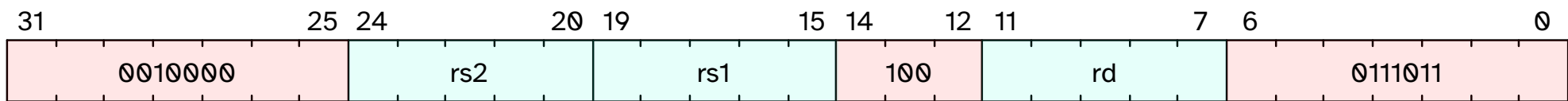
C.267. sh2add.uw

Shift unsigned word left by 2 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.267.1. Encoding



C.267.2. Description

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

C.267.3. Access

M	S	U
Always	Always	Always

C.267.4. Decode Variables

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

C.267.5. IDL Operation

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

C.267.6. Sail Operation

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

C.267.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

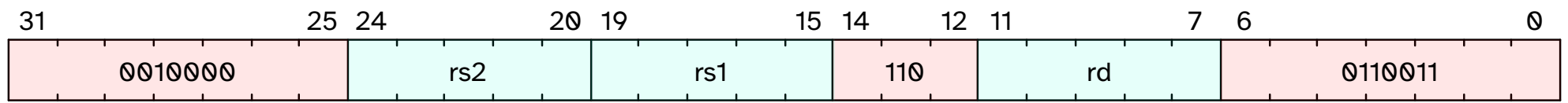
C.268. sh3add

Shift left by 3 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.268.1. Encoding



C.268.2. Description

This instruction shifts rs1 to the left by 3 places and adds it to rs2.

C.268.3. Access

M	S	U
Always	Always	Always

C.268.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.268.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 3);
```

C.268.6. 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.268.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

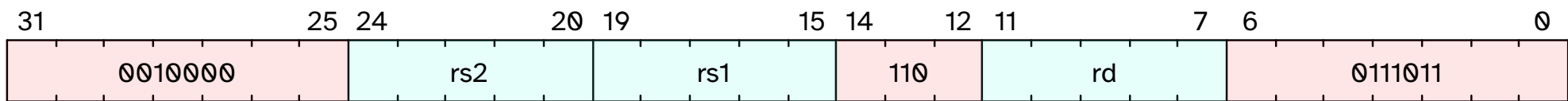
C.269. sh3add.uw

Shift unsigned word left by 3 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.269.1. Encoding



C.269.2. Description

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 3 places.

C.269.3. Access

M	S	U
Always	Always	Always

C.269.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.269.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 3);
```

C.269.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_ADDUW    => 0b00,
    RISCV_SH1ADDUW => 0b01,
    RISCV_SH2ADDUW => 0b10,
    RISCV_SH3ADDUW => 0b11
  };
  let result : xlenbits = (zero_extend(rs1_val[31..0]) << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

C.269.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

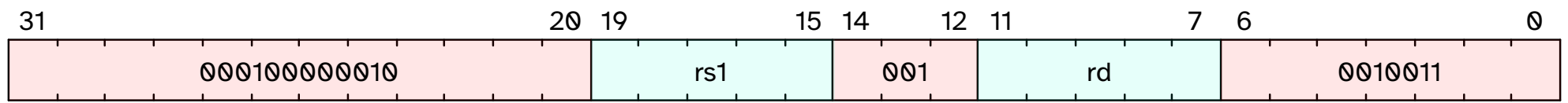
C.270. sha256sig0

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.270.1. Encoding



C.270.2. Description

No description available.

C.270.3. Access

M	S	U
Always	Always	Always

C.270.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.270.5. IDL Operation

C.270.6. Exceptions

This instruction does not generate synchronous exceptions.

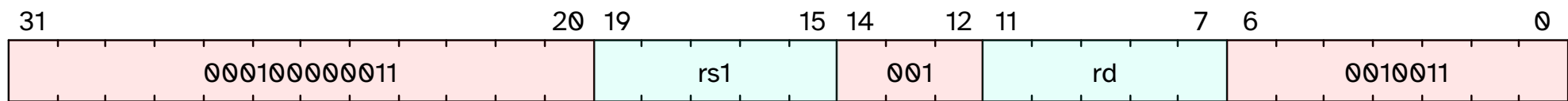
C.271. sha256sig1

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.271.1. Encoding



C.271.2. Description

No description available.

C.271.3. Access

M	S	U
Always	Always	Always

C.271.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.271.5. IDL Operation

C.271.6. Exceptions

This instruction does not generate synchronous exceptions.

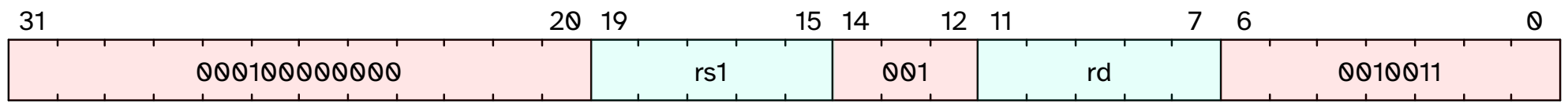
C.272. sha256sum0

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.272.1. Encoding



C.272.2. Description

No description available.

C.272.3. Access

M	S	U
Always	Always	Always

C.272.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.272.5. IDL Operation

C.272.6. Exceptions

This instruction does not generate synchronous exceptions.

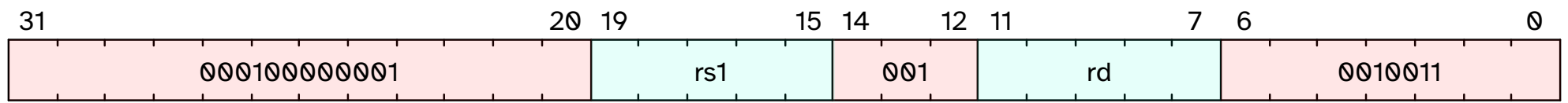
C.273. sha256sum1

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.273.1. Encoding



C.273.2. Description

No description available.

C.273.3. Access

M	S	U
Always	Always	Always

C.273.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.273.5. IDL Operation

C.273.6. Exceptions

This instruction does not generate synchronous exceptions.

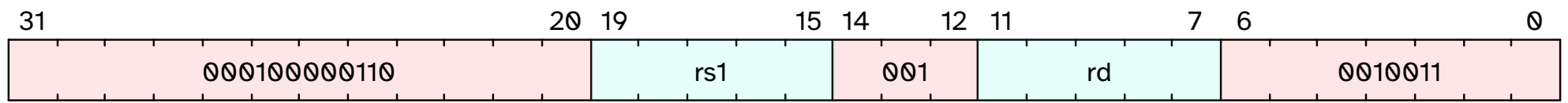
C.274. sha512sig0

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.274.1. Encoding



C.274.2. Description

No description available.

C.274.3. Access

M	S	U
Always	Always	Always

C.274.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.274.5. IDL Operation

C.274.6. Exceptions

This instruction does not generate synchronous exceptions.

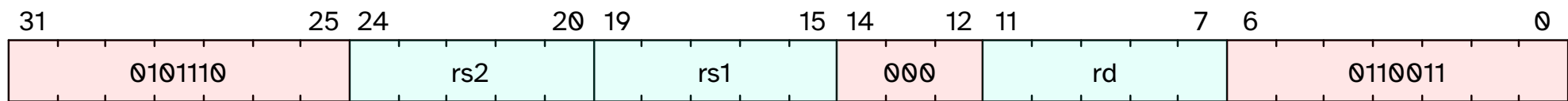
C.275. sha512sig0h

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.275.1. Encoding



C.275.2. Description

No description available.

C.275.3. Access

M	S	U
Always	Always	Always

C.275.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.275.5. IDL Operation

C.275.6. Exceptions

This instruction does not generate synchronous exceptions.

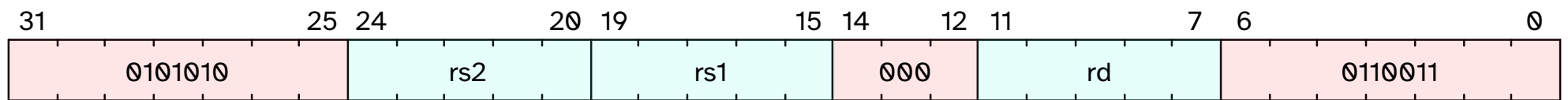
C.276. sha512sig0l

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.276.1. Encoding



C.276.2. Description

No description available.

C.276.3. Access

M	S	U
Always	Always	Always

C.276.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.276.5. IDL Operation

C.276.6. Exceptions

This instruction does not generate synchronous exceptions.

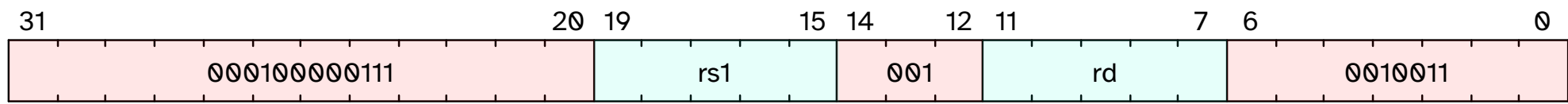
C.277. sha512sig1

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.277.1. Encoding



C.277.2. Description

No description available.

C.277.3. Access

M	S	U
Always	Always	Always

C.277.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.277.5. IDL Operation

C.277.6. Exceptions

This instruction does not generate synchronous exceptions.

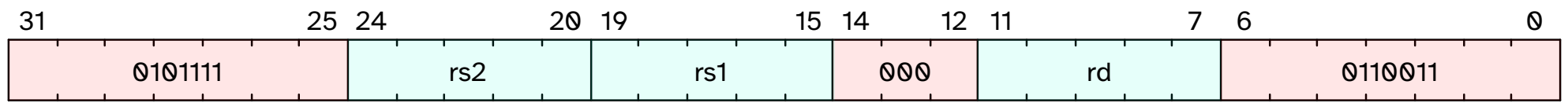
C.278. sha512sig1h

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.278.1. Encoding



C.278.2. Description

No description available.

C.278.3. Access

M	S	U
Always	Always	Always

C.278.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.278.5. IDL Operation

C.278.6. Exceptions

This instruction does not generate synchronous exceptions.

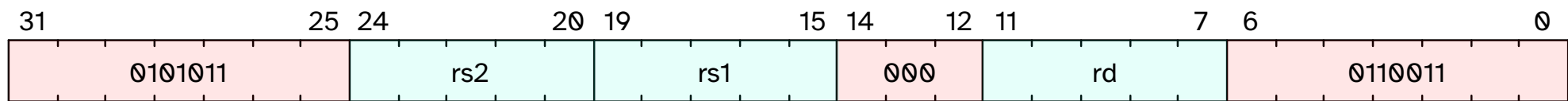
C.279. sha512sig1l

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.279.1. Encoding



C.279.2. Description

No description available.

C.279.3. Access

M	S	U
Always	Always	Always

C.279.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.279.5. IDL Operation

C.279.6. Exceptions

This instruction does not generate synchronous exceptions.

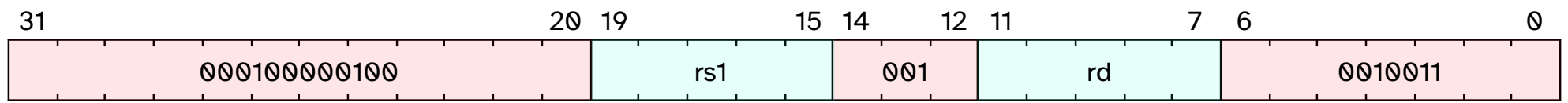
C.280. sha512sum0

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.280.1. Encoding



C.280.2. Description

No description available.

C.280.3. Access

M	S	U
Always	Always	Always

C.280.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.280.5. IDL Operation

C.280.6. Exceptions

This instruction does not generate synchronous exceptions.

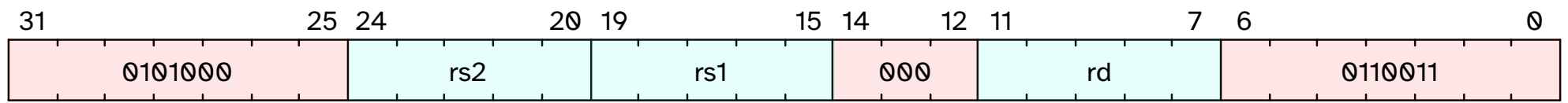
C.281. sha512sum0r

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.281.1. Encoding



C.281.2. Description

No description available.

C.281.3. Access

M	S	U
Always	Always	Always

C.281.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.281.5. IDL Operation

C.281.6. Exceptions

This instruction does not generate synchronous exceptions.

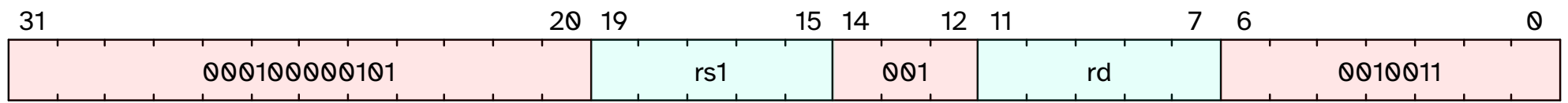
C.282. sha512sum1

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.282.1. Encoding



C.282.2. Description

No description available.

C.282.3. Access

M	S	U
Always	Always	Always

C.282.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.282.5. IDL Operation

C.282.6. Exceptions

This instruction does not generate synchronous exceptions.

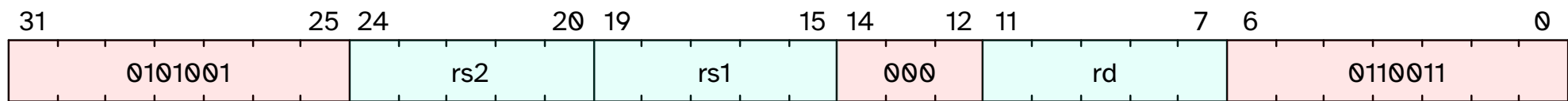
C.283. sha512sum1r

No synopsis available

This instruction is defined by:

- Zknh, version >= Zknh@1.0.0

C.283.1. Encoding



C.283.2. Description

No description available.

C.283.3. Access

M	S	U
Always	Always	Always

C.283.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.283.5. IDL Operation

C.283.6. Exceptions

This instruction does not generate synchronous exceptions.

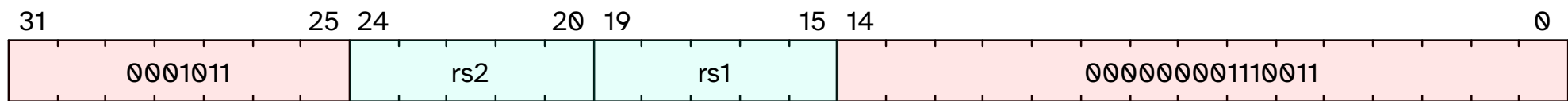
C.284. sinval.vma

Invalidate cached address translations

This instruction is defined by:

- Svinval, version >= Svinval@1.0.0

C.284.1. Encoding



C.284.2. Description

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.

C.284.3. Access

M	S	U
Always	Sometimes	Never

C.284.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

C.284.5. IDL Operation

```
XReg vaddr = X[rs1];
Bits<ASID_WIDTH> asid = X[rs2][ASID_WIDTH - 1:0];
if (mstatus.TVM == 1 && mode() == PrivilegeMode::S) || (mode() == PrivilegeMode::VS) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (misa.H == 1 && hstatus.VTVM == 1 && mode() == PrivilegeMode::VS) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
if (mode() == PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (misa.H == 1 && mode() == PrivilegeMode::VU) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
if (misa.H == 1 && mode() == PrivilegeMode::VS) {
    vma_type.vsmode = true;
    vma_type.single_vmid = true;
    vma_type.vmid = hgatp.VMID;
} else {
    vma_type.smode = true;
}
if ((rs1 == 0) && (rs2 == 0)) {
    vma_type.global = true;
    invalidate_translations(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    invalidate_translations(vma_type);
} else if ((rs1 != 0) && (rs2 == 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.284.6. Exceptions

This instruction may result in the following synchronous exceptions:

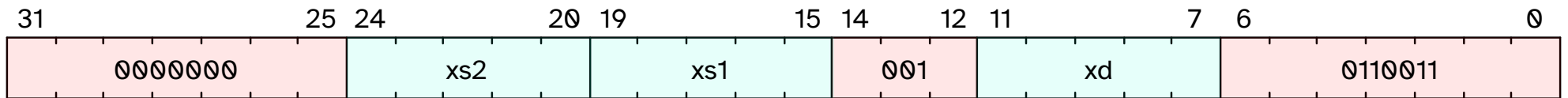
- IllegalInstruction
- VirtualInstruction

DRAFT

C.285. sll**Shift left logical**

This instruction is defined by:

- I, version >= I@2.1.0

C.285.1. Encoding**C.285.2. Description**

Shift the value in `xs1` left by the value in the lower 6 bits of `xs2`, and store the result in `xd`.

C.285.3. Access

M	S	U
Always	Always	Always

C.285.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.285.5. IDL Operation

```
if (xlen() == 64) {
  X[xd] = X[xs1] << X[xs2][5:0];
} else {
  X[xd] = X[xs1] << X[xs2][4:0];
}
```

C.285.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISC_V_ADD => xs1_val + xs2_val,
    RISC_V_SLT => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISC_V_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISC_V_AND => xs1_val & xs2_val,
    RISC_V_OR => xs1_val | xs2_val,
    RISC_V_XOR => xs1_val ^ xs2_val,
    RISC_V_SLL => if sizeof(xlen) == 32
      then xs1_val << (xs2_val[4..0])
      else xs1_val << (xs2_val[5..0]),
    RISC_V_SRL => if sizeof(xlen) == 32
      then xs1_val >> (xs2_val[4..0])
      else xs1_val >> (xs2_val[5..0]),
    RISC_V_SUB => xs1_val - xs2_val,
    RISC_V_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.285.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

C.286. slli

Shift left logical immediate

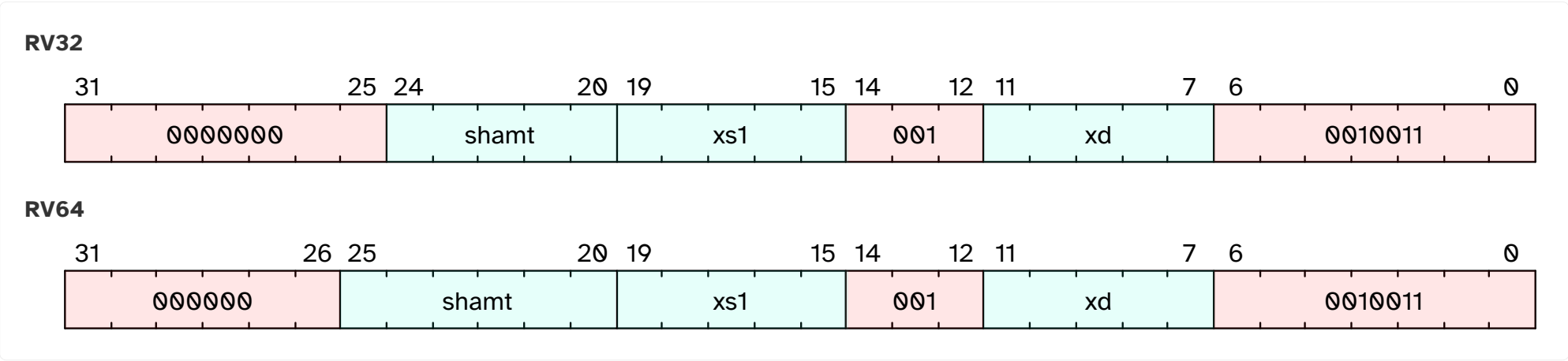
This instruction is defined by:

- I, version >= I@2.1.0

C.286.1. Encoding



This instruction has different encodings in RV32 and RV64.



C.286.2. Description

Shift the value in xs1 left by shamt, and store the result in xd

C.286.3. Access

M	S	U
Always	Always	Always

C.286.4. 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.286.5. IDL Operation

```
X[xd] = X[xs1] << shamt;
```

C.286.6. 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.286.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

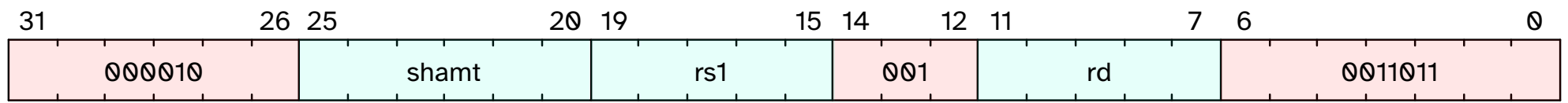
C.287. slli.uw

Shift left unsigned word (Immediate)

This instruction is defined by:

- Zba, version >= Zba@1.0.0

C.287.1. Encoding



C.287.2. Description

This instruction takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate.



This instruction is the same as [slli](#) with `zext.w` performed on `rs1` before shifting.

C.287.3. Access

M	S	U
Always	Always	Always

C.287.4. Decode Variables

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.287.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs1][31:0] << shamt;
```

C.287.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let result : xlenbits = zero_extend(rs1_val[31..0]) << shamt;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

C.287.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

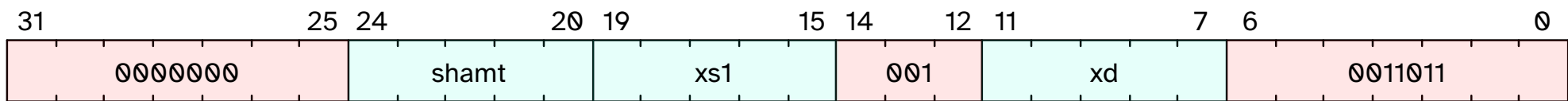
C.288. slliw

Shift left logical immediate word

This instruction is defined by:

- I, version >= I@2.1.0

C.288.1. Encoding



C.288.2. Description

Shift the 32-bit value in xs1 left by shamt, and store the sign-extended result in xd

C.288.3. Access

M	S	U
Always	Always	Always

C.288.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.288.5. IDL Operation

```
X[xd] = sext(X[xs1] << shamt, 31);
```

C.288.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

C.288.7. Exceptions

This instruction does not generate synchronous exceptions.

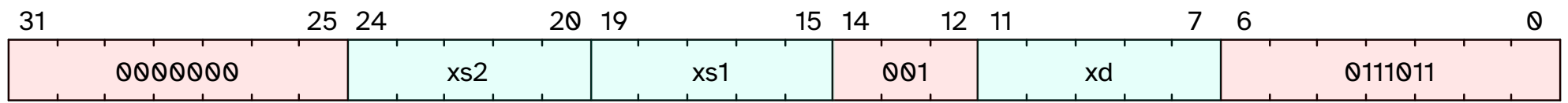
C.289. sllw

Shift left logical word

This instruction is defined by:

- I, version >= I@2.1.0

C.289.1. Encoding



C.289.2. Description

Shift the 32-bit value in xs1 left by the value in the lower 5 bits of xs2, and store the sign-extended result in xd.

C.289.3. Access

M	S	U
Always	Always	Always

C.289.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.289.5. IDL Operation

```
X[xd] = sext(X[xs1] << X[xs2][4:0], 31);
```

C.289.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISC_V_ADDW => xs1_val + xs2_val,
    RISC_V_SUBW => xs1_val - xs2_val,
    RISC_V_SLLW => xs1_val << (xs2_val[4..0]),
    RISC_V_SRLW => xs1_val >> (xs2_val[4..0]),
    RISC_V_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

C.289.7. Exceptions

This instruction does not generate synchronous exceptions.

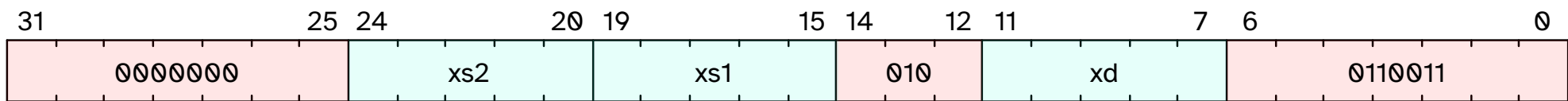
C.290. slt

Set on less than

This instruction is defined by:

- I, version >= I@2.1.0

C.290.1. Encoding



C.290.2. 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.290.3. Access

M	S	U
Always	Always	Always

C.290.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.290.5. IDL Operation

```
XReg src1 = X[xs1];
XReg src2 = X[xs2];
X[xd] = ($signed(src1) < $signed(src2)) ? '1' : '0';
```

C.290.6. 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.290.7. Exceptions

This instruction does not generate synchronous exceptions.

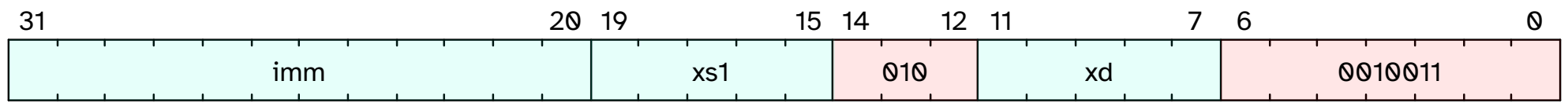
C.291. slti

Set on less than immediate

This instruction is defined by:

- I, version >= I@2.1.0

C.291.1. Encoding



C.291.2. 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.291.3. Access

M	S	U
Always	Always	Always

C.291.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.291.5. IDL Operation

```
X[xd] = ($signed(X[xs1]) < $signed(imm)) ? '1' : '0';
```

C.291.6. 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.291.7. Exceptions

This instruction does not generate synchronous exceptions.

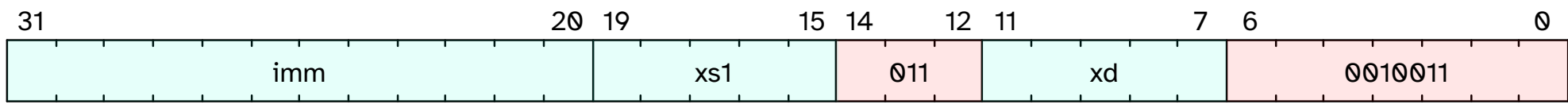
C.292. sltiu

Set on less than immediate unsigned

This instruction is defined by:

- I, version >= I@2.1.0

C.292.1. Encoding



C.292.2. 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.

i

sltiu xd, xs1, 1 sets xd to 1 if xs1 equals zero, otherwise sets xd to 0 (assembler pseudoinstruction SEQZ xd, rs).

C.292.3. Access

M	S	U
Always	Always	Always

C.292.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.292.5. IDL Operation

```
Bits<MXLEN> sign_extend_imm = $signed(imm);
X[xd] = (X[xs1] < sign_extend_imm) ? 1 : 0;
```

C.292.6. 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.292.7. Exceptions

This instruction does not generate synchronous exceptions.

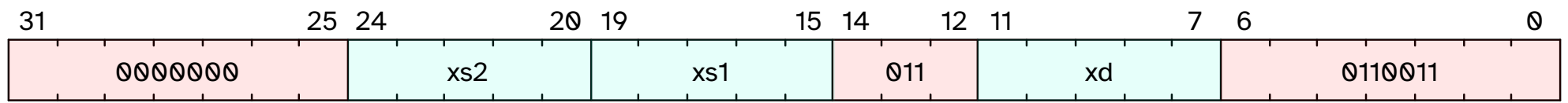
C.293. sltu

Set on less than unsigned

This instruction is defined by:

- I, version >= I@2.1.0

C.293.1. Encoding



C.293.2. 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.293.3. Access

M	S	U
Always	Always	Always

C.293.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.293.5. IDL Operation

```
X[xd] = (X[xs1] < X[xs2]) ? 1 : 0;
```

C.293.6. 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.293.7. Exceptions

This instruction does not generate synchronous exceptions.

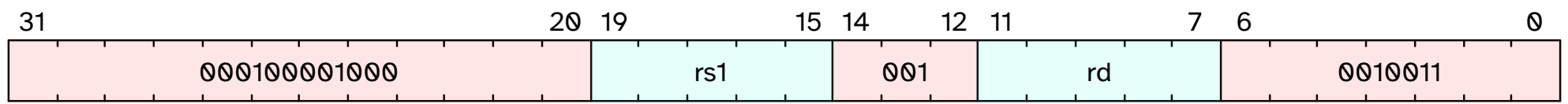
C.294. sm3p0

No synopsis available

This instruction is defined by:

- anyOf:
 - Zks, version >= Zks@1.0.0
 - Zksh, version >= Zksh@1.0.0

C.294.1. Encoding



C.294.2. Description

No description available.

C.294.3. Access

M	S	U
Always	Always	Always

C.294.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.294.5. IDL Operation

C.294.6. Exceptions

This instruction does not generate synchronous exceptions.

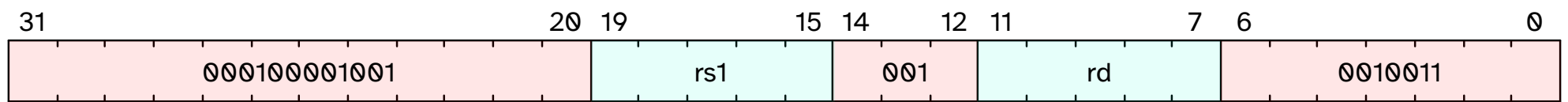
C.295. sm3p1

No synopsis available

This instruction is defined by:

- anyOf:
 - Zks, version >= Zks@1.0.0
 - Zksh, version >= Zksh@1.0.0

C.295.1. Encoding



C.295.2. Description

No description available.

C.295.3. Access

M	S	U
Always	Always	Always

C.295.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.295.5. IDL Operation

C.295.6. Exceptions

This instruction does not generate synchronous exceptions.

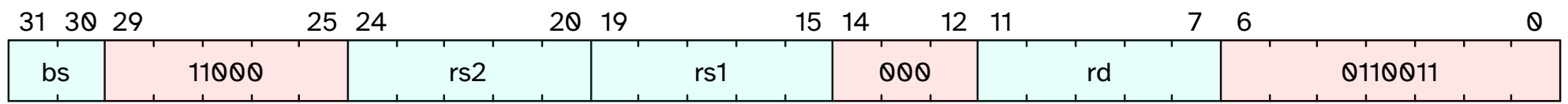
C.296. sm4ed

No synopsis available

This instruction is defined by:

- anyOf:
 - Zks, version >= Zks@1.0.0
 - Zksed, version >= Zksed@1.0.0

C.296.1. Encoding



C.296.2. Description

No description available.

C.296.3. Access

M	S	U
Always	Always	Always

C.296.4. Decode Variables

```
Bits<2> bs = $encoding[31:30];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.296.5. IDL Operation

C.296.6. Exceptions

This instruction does not generate synchronous exceptions.

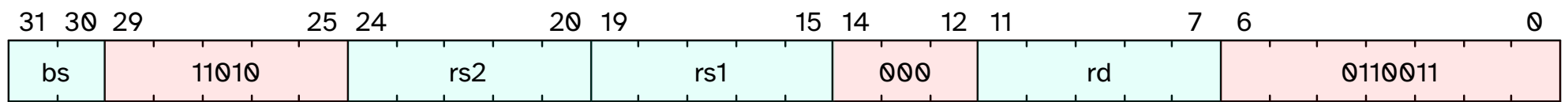
C.297. sm4ks

No synopsis available

This instruction is defined by:

- anyOf:
 - Zks, version >= Zks@1.0.0
 - Zksed, version >= Zksed@1.0.0

C.297.1. Encoding



C.297.2. Description

No description available.

C.297.3. Access

M	S	U
Always	Always	Always

C.297.4. Decode Variables

```
Bits<2> bs = $encoding[31:30];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.297.5. IDL Operation

C.297.6. Exceptions

This instruction does not generate synchronous exceptions.

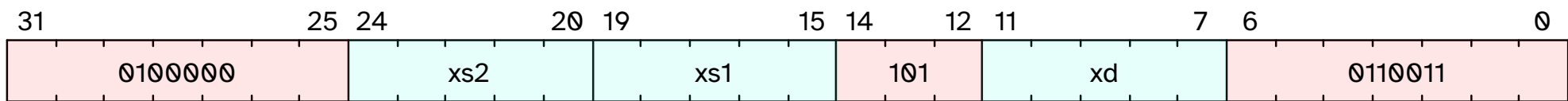
C.298. sra

Shift right arithmetic

This instruction is defined by:

- I, version >= I@2.1.0

C.298.1. Encoding



C.298.2. 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.298.3. Access

M	S	U
Always	Always	Always

C.298.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.298.5. IDL Operation

```
if (xlen() == 64) {
  X[xd] = X[xs1] >>> X[xs2][5:0];
} else {
  X[xd] = X[xs1] >>> X[xs2][4:0];
}
```

C.298.6. 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.298.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

C.299. srai

Shift right arithmetic immediate

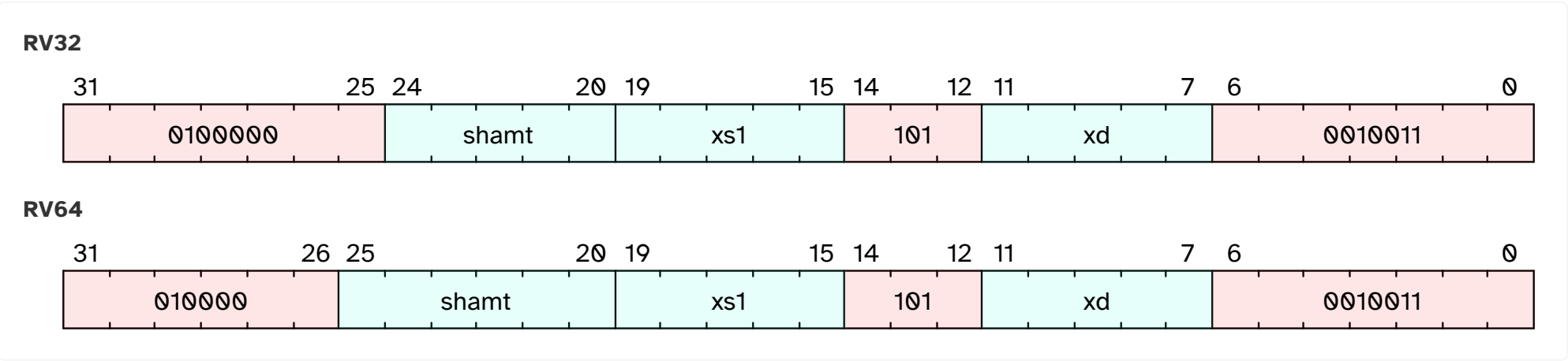
This instruction is defined by:

- I, version >= I@2.1.0

C.299.1. Encoding



This instruction has different encodings in RV32 and RV64.



C.299.2. 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.299.3. Access

M	S	U
Always	Always	Always

C.299.4. 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.299.5. IDL Operation

```
X[xd] = X[xs1] >>> shamt;
```

C.299.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISC_V_SLLI => if sizeof(xlen) == 32
      then xs1_val << shamt[4..0]
      else xs1_val << shamt,
    RISC_V_SRLI => if sizeof(xlen) == 32
      then xs1_val >> shamt[4..0]
      else xs1_val >> shamt,
    RISC_V_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.299.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

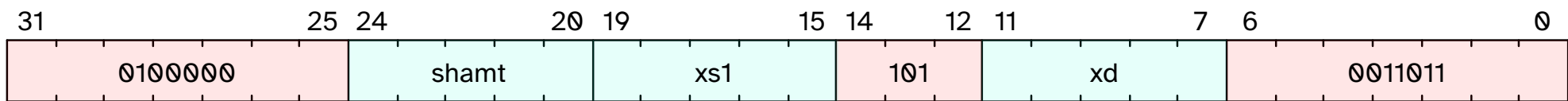
C.300. sraiw

Shift right arithmetic immediate word

This instruction is defined by:

- I, version >= I@2.1.0

C.300.1. Encoding



C.300.2. Description

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the 32-bit value in xs1 right by shamt, and store the sign-extended result in xd.

C.300.3. Access

M	S	U
Always	Always	Always

C.300.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.300.5. IDL Operation

```
XReg operand = sext(X[xs1], 31);
X[xd] = sext(operand >>> shamt, 31);
```

C.300.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

C.300.7. Exceptions

This instruction does not generate synchronous exceptions.

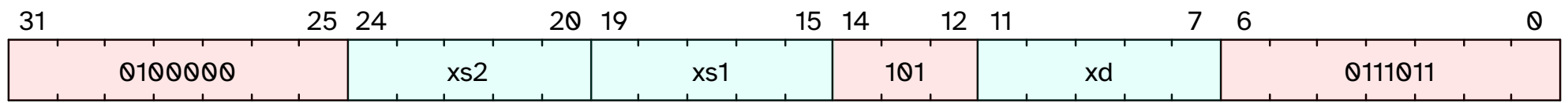
C.301. sraw

Shift right arithmetic word

This instruction is defined by:

- I, version >= I@2.1.0

C.301.1. Encoding



C.301.2. Description

Arithmetic shift the 32-bit value in xs1 right by the value in the lower 5 bits of xs2, and store the sign-extended result in xd.

C.301.3. Access

M	S	U
Always	Always	Always

C.301.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.301.5. IDL Operation

```
XReg operand1 = sext(X[xs1], 31);
X[xd] = sext(operand1 >>> X[xs2][4:0], 31);
```

C.301.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISC_V_ADDW => xs1_val + xs2_val,
    RISC_V_SUBW => xs1_val - xs2_val,
    RISC_V_SLLW => xs1_val << (xs2_val[4..0]),
    RISC_V_SRLW => xs1_val >> (xs2_val[4..0]),
    RISC_V_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

C.301.7. Exceptions

This instruction does not generate synchronous exceptions.

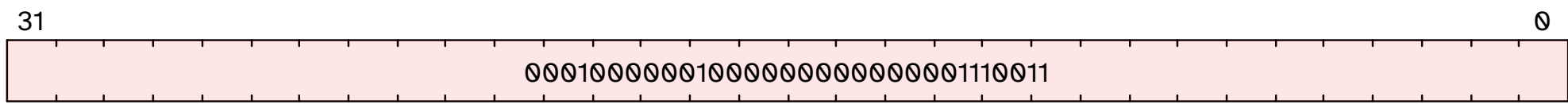
C.302. sret

Supervisor Exception Return

This instruction is defined by:

- S, version >= S@1.11.0

C.302.1. Encoding



C.302.2. Description

Returns from an exception.

When `sret` is allowed to execute, its behavior depends on whether or not the current privilege mode is virtualized.

When the current privilege mode is (H)S-mode or M-mode

`sret` sets `hstatus` = 0, `mstatus.SPP` = 0, `mstatus.SIE` = `mstatus.SPIE`, and `mstatus.SPIE` = 1, changes the privilege mode according to the table below, and then jumps to the address in `sepc`.

Table 14. 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 15. 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.302.3. 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.302.4. Decode Variables

C.302.5. IDL Operation

```
if (implemented?(ExtensionName::H)) {
  if (mstatus.TSR == 1'b0 && hstatus.VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  }
}
```

```

    } else if (mode() == PrivilegeMode::VU) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
} else if (mstatus.TSR == 1'b0 && hstatus.VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode::VS) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
} else if (mstatus.TSR == 1'b1 && hstatus.VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
} else if (mstatus.TSR == 1'b1 && hstatus.VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode::VS) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
}
} else {
    if (mode() != PrivilegeMode::U) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
if (!virtual_mode?()) {
    if (implemented?(ExtensionName::H)) {
        if (hstatus.SPV == 1'b1) {
            if (mstatus.SPP == 2'b01) {
                set_mode(PrivilegeMode::VS);
            } else if (mstatus.SPP == 2'b00) {
                set_mode(PrivilegeMode::VU);
            }
        }
        hstatus.SPV = 0;
    }
    mstatus.SIE = mstatus.SPIE;
    mstatus.SPIE = 1;
    mstatus.SPP = 2'b00;
    $pc = $bits(sepc);
} else {
    if (mstatus.TSR == 1'b1) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
    vsstatus.SPP = 0;
    vsstatus.SIE = vsstatus.SPIE;
    vsstatus.SPIE = 1;
    $pc = $bits(vsepc);
}
}

```

C.302.6. 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.302.7. Exceptions

This instruction may result in the following synchronous exceptions:

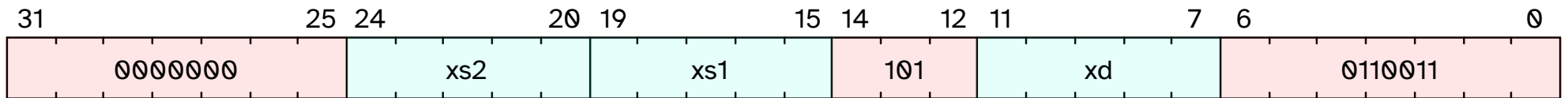
- IllegalInstruction
- VirtualInstruction

DRAFT

C.303. srl**Shift right logical**

This instruction is defined by:

- I, version >= I@2.1.0

C.303.1. Encoding**C.303.2. Description**

Logical shift the value in xs1 right by the value in the lower bits of xs2, and store the result in xd.

C.303.3. Access

M	S	U
Always	Always	Always

C.303.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.303.5. IDL Operation

```
if (xlen() == 64) {
  X[xd] = X[xs1] >> X[xs2][5:0];
} else {
  X[xd] = X[xs1] >> X[xs2][4:0];
}
```

C.303.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISC_V_ADD => xs1_val + xs2_val,
    RISC_V_SLT => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISC_V_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISC_V_AND => xs1_val & xs2_val,
    RISC_V_OR => xs1_val | xs2_val,
    RISC_V_XOR => xs1_val ^ xs2_val,
    RISC_V_SLL => if sizeof(xlen) == 32
      then xs1_val << (xs2_val[4..0])
      else xs1_val << (xs2_val[5..0]),
    RISC_V_SRL => if sizeof(xlen) == 32
      then xs1_val >> (xs2_val[4..0])
      else xs1_val >> (xs2_val[5..0]),
    RISC_V_SUB => xs1_val - xs2_val,
    RISC_V_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.303.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

C.304. srli

Shift right logical immediate

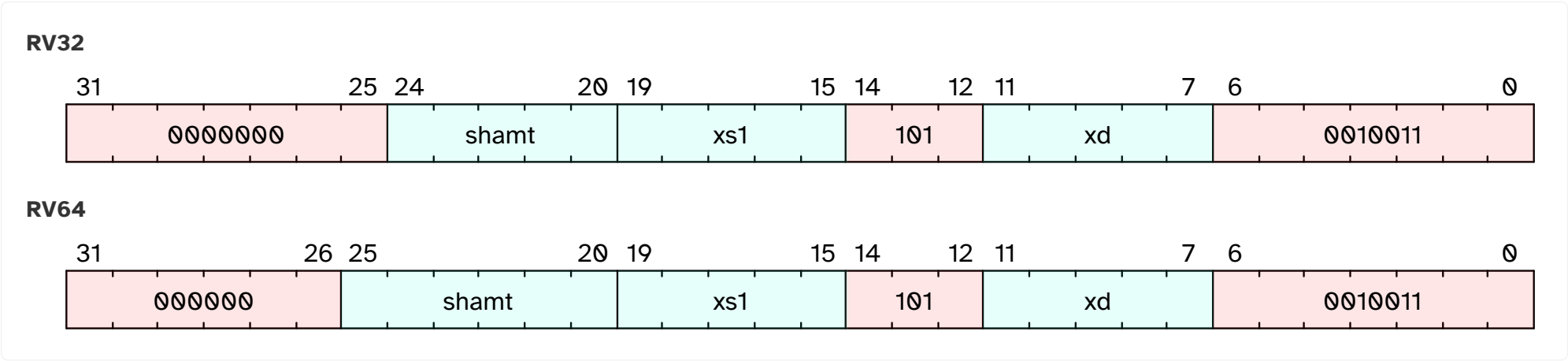
This instruction is defined by:

- I, version >= I@2.1.0

C.304.1. Encoding



This instruction has different encodings in RV32 and RV64.



C.304.2. Description

Shift the value in xs1 right by shamt, and store the result in xd

C.304.3. Access

M	S	U
Always	Always	Always

C.304.4. 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.304.5. IDL Operation

```
X[xd] = X[xs1] >> shamt;
```

C.304.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISC_V_SLLI => if sizeof(xlen) == 32
      then xs1_val << shamt[4..0]
      else xs1_val << shamt,
    RISC_V_SRLI => if sizeof(xlen) == 32
      then xs1_val >> shamt[4..0]
      else xs1_val >> shamt,
    RISC_V_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.304.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

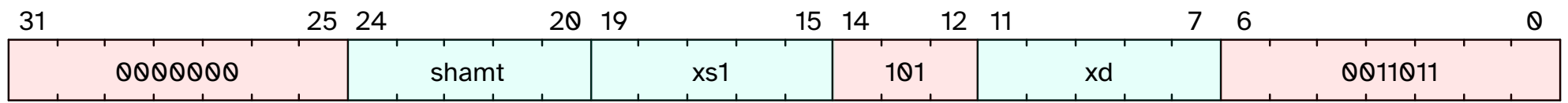
C.305. srlw

Shift right logical immediate word

This instruction is defined by:

- I, version >= I@2.1.0

C.305.1. Encoding



C.305.2. Description

Shift the 32-bit value in xs1 right by shamt, and store the sign-extended result in xd

C.305.3. Access

M	S	U
Always	Always	Always

C.305.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.305.5. IDL Operation

```
XReg operand = X[xs1][31:0];
X[xd] = sext(operand >> shamt, 31);
```

C.305.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

C.305.7. Exceptions

This instruction does not generate synchronous exceptions.

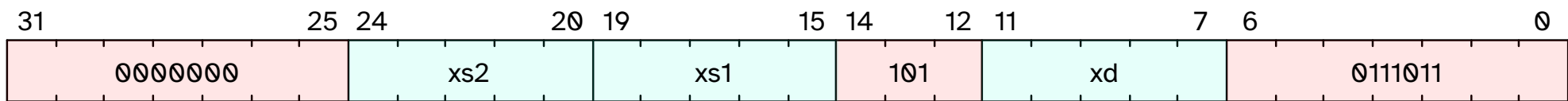
C.306. srlw

Shift right logical word

This instruction is defined by:

- I, version >= I@2.1.0

C.306.1. Encoding



C.306.2. Description

Logical shift the 32-bit value in xs1 right by the value in the lower 5 bits of xs2, and store the sign-extended result in xd.

C.306.3. Access

M	S	U
Always	Always	Always

C.306.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.306.5. IDL Operation

```
X[xd] = sext(X[xs1][31:0] >> X[xs2][4:0], 31);
```

C.306.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISC_V_ADDW => xs1_val + xs2_val,
    RISC_V_SUBW => xs1_val - xs2_val,
    RISC_V_SLLW => xs1_val << (xs2_val[4..0]),
    RISC_V_SRLW => xs1_val >> (xs2_val[4..0]),
    RISC_V_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

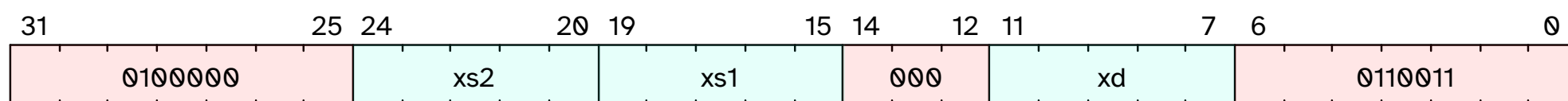
C.306.7. Exceptions

This instruction does not generate synchronous exceptions.

C.307. sub**Subtract**

This instruction is defined by:

- I, version >= I@2.1.0

C.307.1. Encoding**C.307.2. Description**

Subtract the value in xs2 from xs1, and store the result in xd

C.307.3. Access

M	S	U
Always	Always	Always

C.307.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.307.5. IDL Operation

```
XReg t0 = X[xs1];
XReg t1 = X[xs2];
X[xd] = t0 - t1;
```

C.307.6. 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.307.7. Exceptions

This instruction does not generate synchronous exceptions.

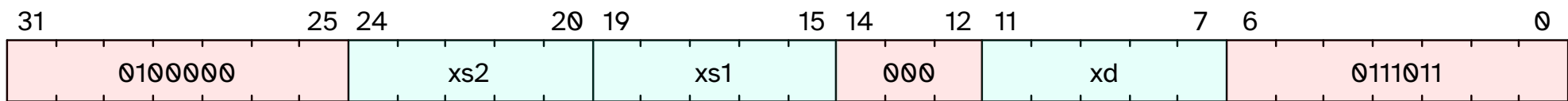
C.308. subw

Subtract word

This instruction is defined by:

- I, version >= I@2.1.0

C.308.1. Encoding



C.308.2. Description

Subtract the 32-bit values in xs2 from xs1, and store the sign-extended result in xd

C.308.3. Access

M	S	U
Always	Always	Always

C.308.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.308.5. IDL Operation

```
Bits<32> t0 = X[xs1][31:0];
Bits<32> t1 = X[xs2][31:0];
X[xd] = sext(t0 - t1, 31);
```

C.308.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISC_V_ADDW => xs1_val + xs2_val,
    RISC_V_SUBW => xs1_val - xs2_val,
    RISC_V_SLLW => xs1_val << (xs2_val[4..0]),
    RISC_V_SRLW => xs1_val >> (xs2_val[4..0]),
    RISC_V_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

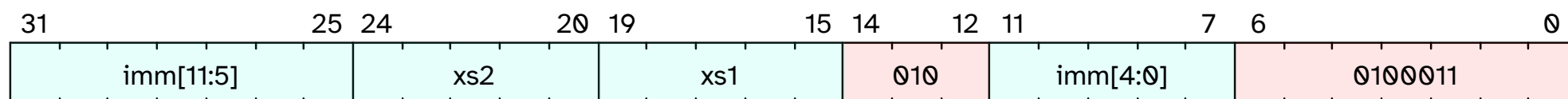
C.308.7. Exceptions

This instruction does not generate synchronous exceptions.

C.309. sw**Store word**

This instruction is defined by:

- I, version $\geq$ I@2.1.0

C.309.1. Encoding**C.309.2. Description**

Store 32 bits of data from register xs2 to an address formed by adding xs1 to a signed offset.

C.309.3. Access

M	S	U
Always	Always	Always

C.309.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

C.309.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<32>(virtual_address, X[xs2][31:0], $encoding);
```

C.309.6. 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.309.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

DRAFT

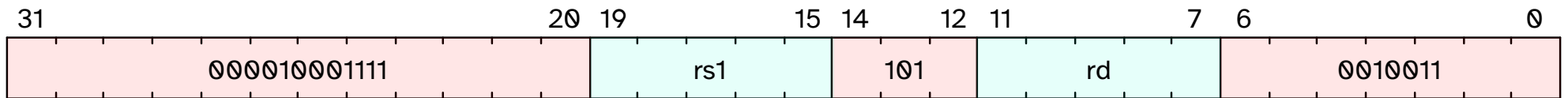
C.310. unzip

Bit deinterleave

This instruction is defined by:

- Zbkb, version $\geq$ Zbkb@1.0.0

C.310.1. Encoding



C.310.2. Description

This instruction gathers bits from the high and low halves of the source word into odd/even bit positions in the destination word. It is the inverse of the zip instruction. This instruction is available only on RV32.

C.310.3. Access

M	S	U
Always	Always	Always

C.310.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.310.5. IDL Operation

```
XReg input = X[rs1];
XReg output = 0;
for (U32 i = 0; i < (xlen() / 2 - 1); i = i + 1) {
    output[i] = input[2 * i];
    output[i + xlen() / 2] = input[2 * i + 1];
}
X[rd] = output;
```

C.310.6. Sail Operation

```
foreach (i from 0 to xlen/2-1) {
    X(rd)[i] = X(rs1)[2*i];
    X(rd)[i+xlen/2] = X(rs1)[2*i+1];
}
```

C.310.7. Exceptions

This instruction does not generate synchronous exceptions.

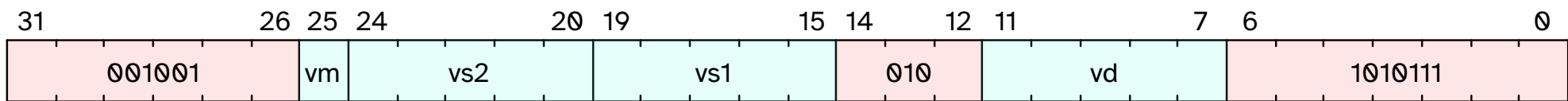
C.311. vaadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.311.1. Encoding



C.311.2. Description

No description available.

C.311.3. Access

M	S	U
Always	Always	Always

C.311.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.311.5. IDL Operation

C.311.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.311.7. Exceptions

This instruction does not generate synchronous exceptions.

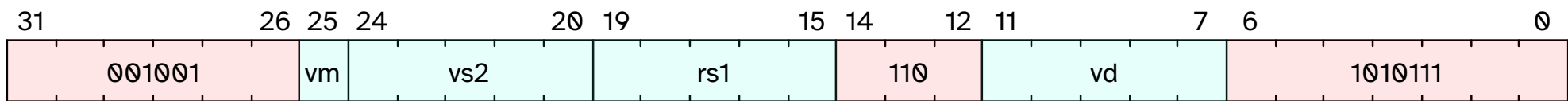
C.312. vaadd.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.312.1. Encoding



C.312.2. Description

No description available.

C.312.3. Access

M	S	U
Always	Always	Always

C.312.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.312.5. IDL Operation

C.312.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU      => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB       => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP   => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL        => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU      => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU     => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU       => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV        => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU       => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM        => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.312.7. Exceptions

This instruction does not generate synchronous exceptions.

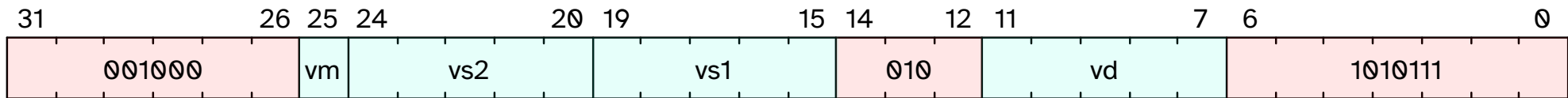
C.313. vaaddu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.313.1. Encoding



C.313.2. Description

No description available.

C.313.3. Access

M	S	U
Always	Always	Always

C.313.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.313.5. IDL Operation

C.313.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.313.7. Exceptions

This instruction does not generate synchronous exceptions.

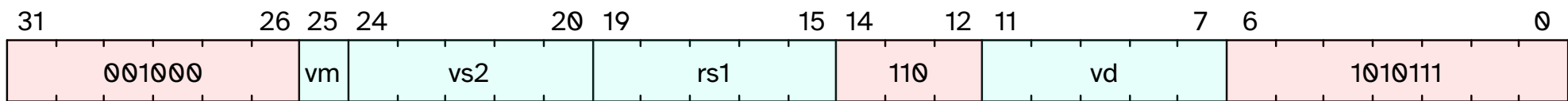
C.314. vaaddu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.314.1. Encoding



C.314.2. Description

No description available.

C.314.3. Access

M	S	U
Always	Always	Always

C.314.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.314.5. IDL Operation

C.314.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU      => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD        => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU     => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU     => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM      => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.314.7. Exceptions

This instruction does not generate synchronous exceptions.

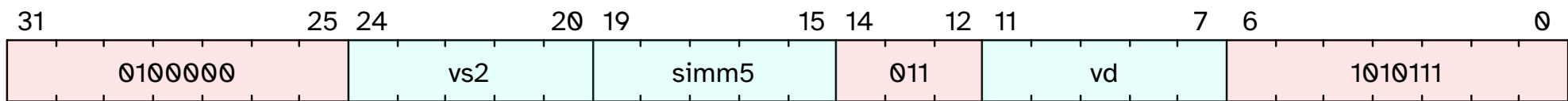
C.315. vadc.vim

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.315.1. Encoding



C.315.2. Description

No description available.

C.315.3. Access

M	S	U
Always	Always	Always

C.315.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.315.5. IDL Operation

C.315.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  /* for bypassing normal masking in init_masked_result */
  vec_trues : vector('n, dec, bool) = undefined;
  foreach (i from 0 to (num_elem - 1)) {
    vec_trues[i] = true
  };

  let vm_val  : vector('n, dec, bool)    = read_vmask_carry(num_elem, 0b0, 0b000000);
  let imm_val : bits('m)                 = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vec_trues);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VIMS_VADC    => to_bits(SEW, unsigned(vs2_val[i]) + unsigned(imm_val) + unsigned(bool_to_bits(vm_val[i])))
      }
    }
  };
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.315.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

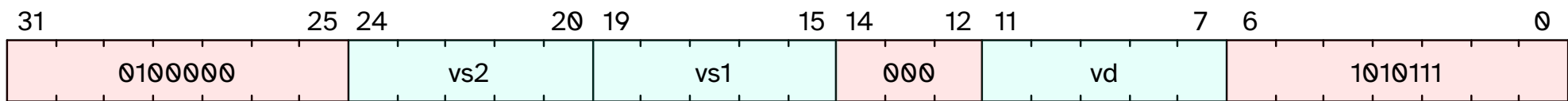
C.316. vadc.vvm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.316.1. Encoding



C.316.2. Description

No description available.

C.316.3. Access

M	S	U
Always	Always	Always

C.316.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.316.5. IDL Operation

C.316.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  /* for bypassing normal masking in init_masked_result */
  vec_trues : vector('n, dec, bool) = undefined;
  foreach (i from 0 to (num_elem - 1)) {
    vec_trues[i] = true
  };

  let vm_val  : vector('n, dec, bool)    = read_vmask_carry(num_elem, 0b0, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vec_trues);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VVMS_VADC    => to_bits(SEW, unsigned(vs2_val[i]) + unsigned(vs1_val[i]) + unsigned(bool_to_bits(vm_val[i]))),
        VVMS_VSBC    => to_bits(SEW, unsigned(vs2_val[i]) - unsigned(vs1_val[i]) - unsigned(bool_to_bits(vm_val[i])))
      }
    }
  };
}
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.316.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

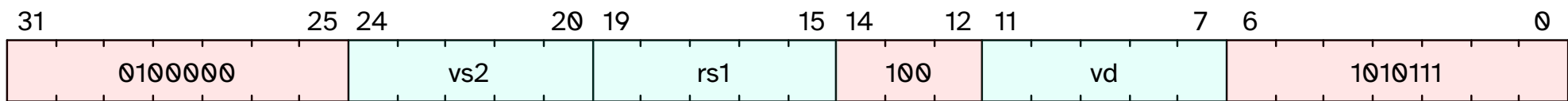
C.317. vadc.vxm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.317.1. Encoding



C.317.2. Description

No description available.

C.317.3. Access

M	S	U
Always	Always	Always

C.317.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.317.5. IDL Operation

C.317.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  /* for bypassing normal masking in init_masked_result */
  vec_trues : vector('n, dec, bool) = undefined;
  foreach (i from 0 to (num_elem - 1)) {
    vec_trues[i] = true
  };

  let vm_val  : vector('n, dec, bool)      = read_vmask_carry(num_elem, 0b0, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vec_trues);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VXMS_VADC    => to_bits(SEW, unsigned(vs2_val[i]) + unsigned(rs1_val) + unsigned(bool_to_bits(vm_val[i]))),
        VXMS_VSBC    => to_bits(SEW, unsigned(vs2_val[i]) - unsigned(rs1_val) - unsigned(bool_to_bits(vm_val[i])))
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.317.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

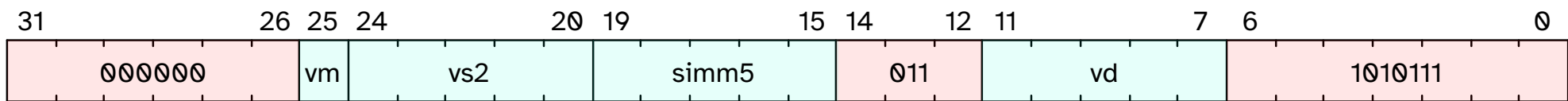
C.318. vadd.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.318.1. Encoding



C.318.2. Description

No description available.

C.318.3. Access

M	S	U
Always	Always	Always

C.318.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.318.5. IDL Operation

C.318.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD    => vs2_val[i] + imm_val,
        VI_VRSUB   => imm_val - vs2_val[i],
        VI_VAND     => vs2_val[i] & imm_val,
        VI_VOR      => vs2_val[i] | imm_val,
        VI_VXOR     => vs2_val[i] ^ imm_val,
        VI_VSADDU   => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD    => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL     => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.318.7. Exceptions

This instruction does not generate synchronous exceptions.

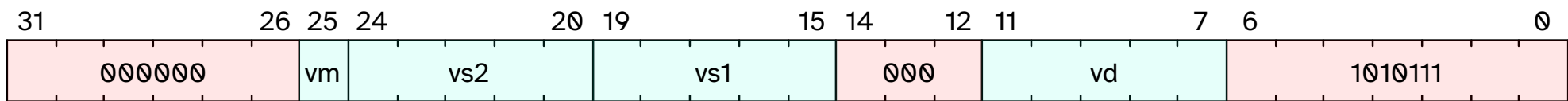
C.319. vadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.319.1. Encoding



C.319.2. Description

No description available.

C.319.3. Access

M	S	U
Always	Always	Always

C.319.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.319.5. IDL Operation

C.319.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.319.7. Exceptions

This instruction does not generate synchronous exceptions.

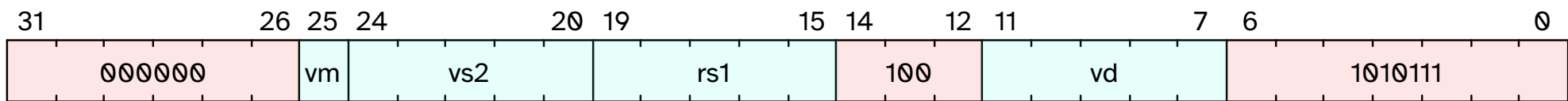
C.320. vadd.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.320.1. Encoding



C.320.2. Description

No description available.

C.320.3. Access

M	S	U
Always	Always	Always

C.320.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.320.5. IDL Operation

C.320.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD  => vs2_val[i] + rs1_val,
        VX_VSUB  => vs2_val[i] - rs1_val,
        VX_VRSUB => rs1_val - vs2_val[i],
        VX_VAND  => vs2_val[i] & rs1_val,
        VX_VOR   => vs2_val[i] | rs1_val,
        VX_VXOR  => vs2_val[i] ^ rs1_val,
        VX_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.320.7. Exceptions

This instruction does not generate synchronous exceptions.

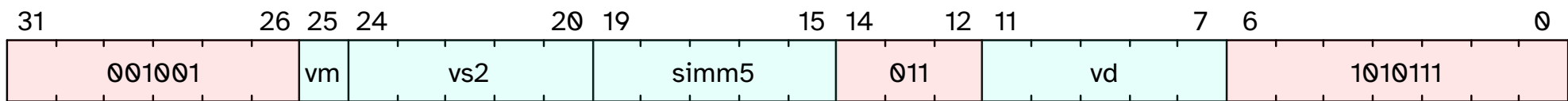
C.321. vand.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.321.1. Encoding



C.321.2. Description

No description available.

C.321.3. Access

M	S	U
Always	Always	Always

C.321.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.321.5. IDL Operation

C.321.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD    => vs2_val[i] + imm_val,
        VI_VRSUB   => imm_val - vs2_val[i],
        VI_VAND     => vs2_val[i] & imm_val,
        VI_VOR      => vs2_val[i] | imm_val,
        VI_VXOR     => vs2_val[i] ^ imm_val,
        VI_VSADDU   => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD    => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL     => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.321.7. Exceptions

This instruction does not generate synchronous exceptions.

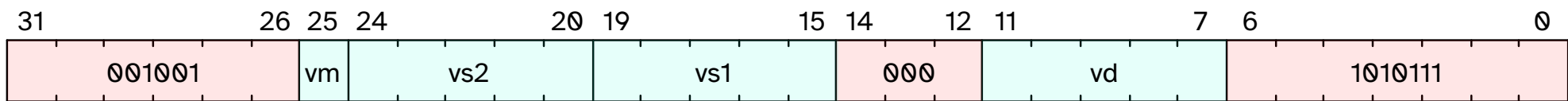
C.322. vand.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.322.1. Encoding



C.322.2. Description

No description available.

C.322.3. Access

M	S	U
Always	Always	Always

C.322.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.322.5. IDL Operation

C.322.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.322.7. Exceptions

This instruction does not generate synchronous exceptions.

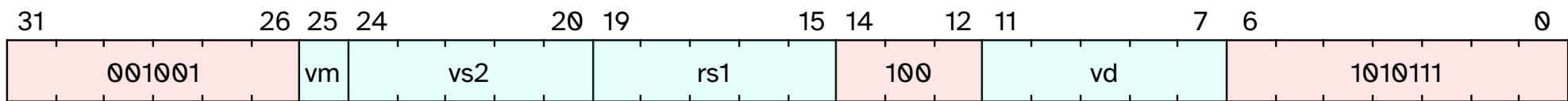
C.323. vand.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.323.1. Encoding



C.323.2. Description

No description available.

C.323.3. Access

M	S	U
Always	Always	Always

C.323.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.323.5. IDL Operation

C.323.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.323.7. Exceptions

This instruction does not generate synchronous exceptions.

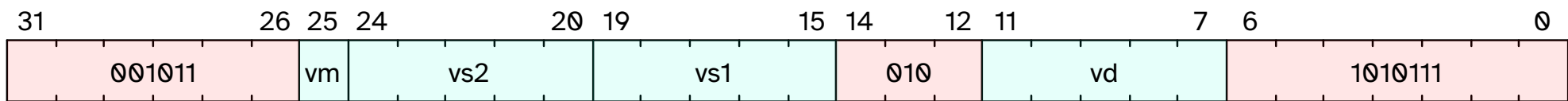
C.324. vasub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.324.1. Encoding



C.324.2. Description

No description available.

C.324.3. Access

M	S	U
Always	Always	Always

C.324.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.324.5. IDL Operation

C.324.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.324.7. Exceptions

This instruction does not generate synchronous exceptions.

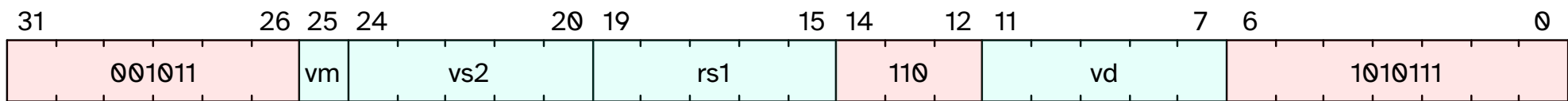
C.325. vasub.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.325.1. Encoding



C.325.2. Description

No description available.

C.325.3. Access

M	S	U
Always	Always	Always

C.325.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.325.5. IDL Operation

C.325.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD      => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.325.7. Exceptions

This instruction does not generate synchronous exceptions.

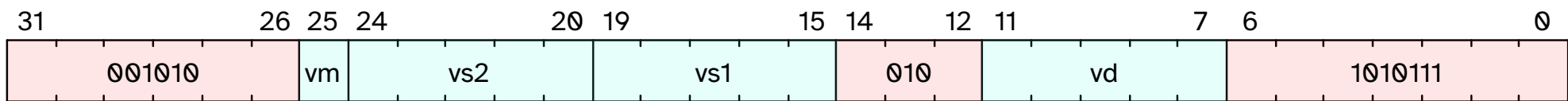
C.326. vasubu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.326.1. Encoding



C.326.2. Description

No description available.

C.326.3. Access

M	S	U
Always	Always	Always

C.326.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.326.5. IDL Operation

C.326.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.326.7. Exceptions

This instruction does not generate synchronous exceptions.

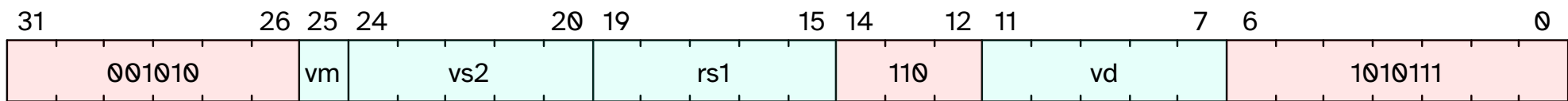
C.327. vasubu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.327.1. Encoding



C.327.2. Description

No description available.

C.327.3. Access

M	S	U
Always	Always	Always

C.327.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.327.5. IDL Operation

C.327.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU      => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD       => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.327.7. Exceptions

This instruction does not generate synchronous exceptions.

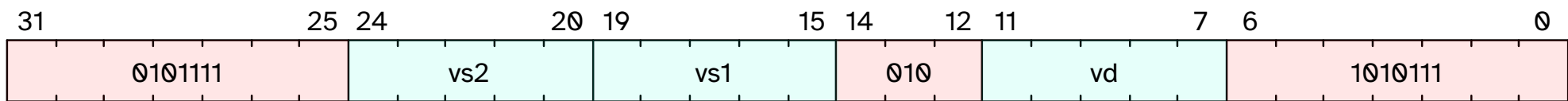
C.328. vcompress.vm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.328.1. Encoding



C.328.2. Description

No description available.

C.328.3. Access

M	S	U
Always	Always	Always

C.328.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.328.5. IDL Operation

C.328.6. Sail Operation

```
{
  let start_element = get_start_element();
  let end_element = get_end_element();
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  /* vcompress should always be executed with a vstart of 0 */
  if start_element != 0 | vs1 == vd | vs2 == vd | illegal_vd_unmasked()
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  /* body elements */
  vd_idx : nat = 0;
  foreach (i from 0 to (num_elem - 1)) {
    if i <= end_element then {
      if vs1_val[i] then {
        let 'p = vd_idx;
        assert('p < 'n);
        result['p] = vs2_val[i];
        vd_idx = vd_idx + 1;
      }
    }
  };
  /* tail elements */
  if vd_idx < num_elem then {
```

```

let tail_ag : agtype = get_vtype_vta();
let 'p = vd_idx;
foreach (i from 'p to (num_elem - 1)) {
  result[i] = match tail_ag {
    UNDISTURBED => vd_val[i],
    AGNOSTIC    => vd_val[i] /* TODO: configuration support */
  }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.328.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

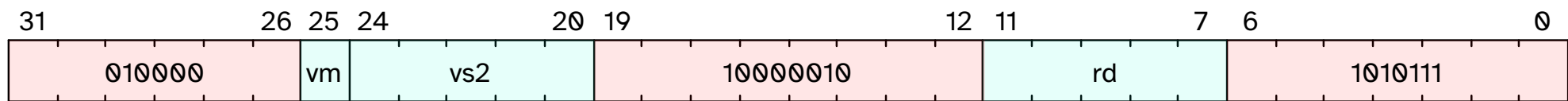
C.329. vcpop.m

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.329.1. Encoding



C.329.2. Description

No description available.

C.329.3. Access

M	S	U
Always	Always	Always

C.329.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rd = $encoding[11:7];
```

C.329.5. IDL Operation

C.329.6. Exceptions

This instruction does not generate synchronous exceptions.

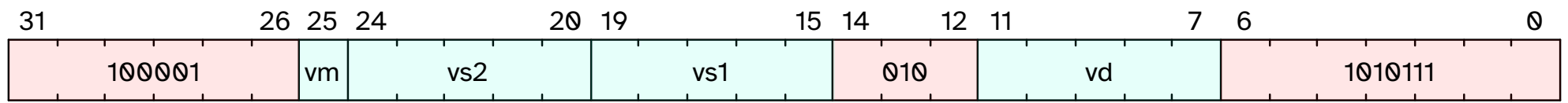
C.330. vdiv.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.330.1. Encoding



C.330.2. Description

No description available.

C.330.3. Access

M	S	U
Always	Always	Always

C.330.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.330.5. IDL Operation

C.330.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.330.7. Exceptions

This instruction does not generate synchronous exceptions.

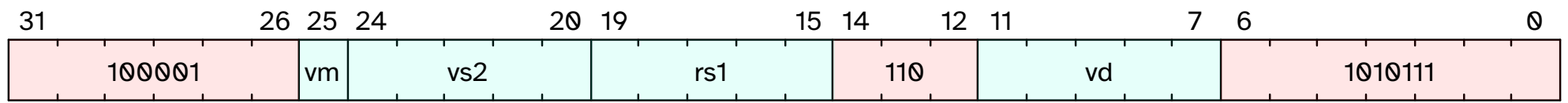
C.331. vdiv.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.331.1. Encoding



C.331.2. Description

No description available.

C.331.3. Access

M	S	U
Always	Always	Always

C.331.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.331.5. IDL Operation

C.331.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```



```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.331.7. Exceptions

This instruction does not generate synchronous exceptions.

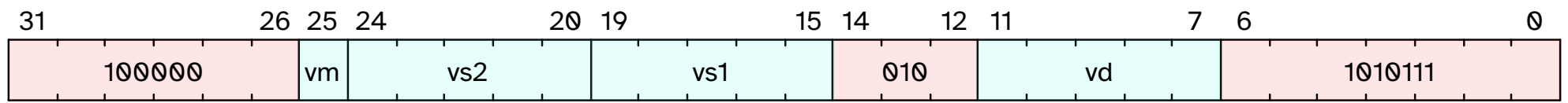
C.332. vdivu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.332.1. Encoding



C.332.2. Description

No description available.

C.332.3. Access

M	S	U
Always	Always	Always

C.332.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.332.5. IDL Operation

C.332.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```



```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.332.7. Exceptions

This instruction does not generate synchronous exceptions.

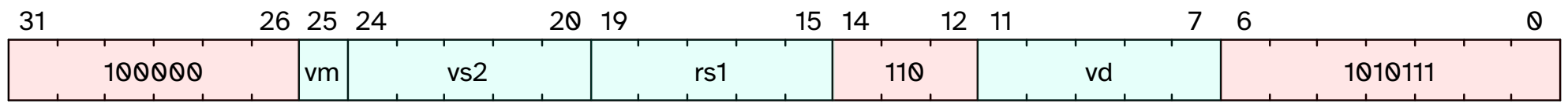
C.333. vdivu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.333.1. Encoding



C.333.2. Description

No description available.

C.333.3. Access

M	S	U
Always	Always	Always

C.333.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.333.5. IDL Operation

C.333.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.333.7. Exceptions

This instruction does not generate synchronous exceptions.

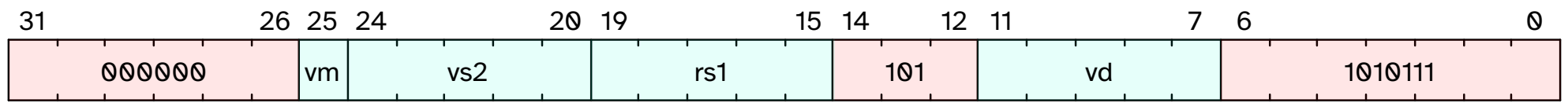
C.334. vfadd.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.334.1. Encoding



C.334.2. Description

No description available.

C.334.3. Access

M	S	U
Always	Always	Always

C.334.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.334.5. IDL Operation

C.334.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```
VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.334.7. Exceptions

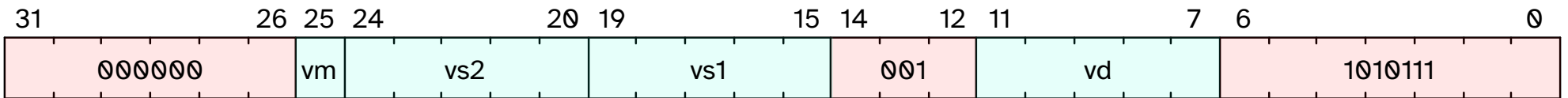
This instruction does not generate synchronous exceptions.

DRAFT

C.335. vfadd.vv**No synopsis available**

This instruction is defined by:

- V, version >= V@1.0.0

C.335.1. Encoding**C.335.2. Description**

No description available.

C.335.3. Access

M	S	U
Always	Always	Always

C.335.4. Decode Variables

```

Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];

```

C.335.5. IDL Operation**C.335.6. Sail Operation**

```

{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],

```



```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.335.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

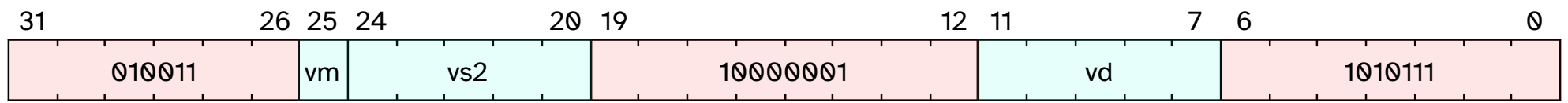
C.336. vfcclass.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.336.1. Encoding



C.336.2. Description

No description available.

C.336.3. Access

M	S	U
Always	Always	Always

C.336.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.336.5. IDL Operation

C.336.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary1 {
        FVV_VSQRT      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16Sqrt(rm_3b, vs2_val[i]),
            32 => riscv_f32Sqrt(rm_3b, vs2_val[i]),
            64 => riscv_f64Sqrt(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FVV_VRSQRT7    => {
```



```

        let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16Rsqrte7(rm_3b, vs2_val[i]),
            32 => riscv_f32Rsqrte7(rm_3b, vs2_val[i]),
            64 => riscv_f64Rsqrte7(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
FVV_VREC7 => {
    let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
        16 => riscv_f16Recip7(rm_3b, vs2_val[i]),
        32 => riscv_f32Recip7(rm_3b, vs2_val[i]),
        64 => riscv_f64Recip7(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FVV_VCLASS => fp_class(vs2_val[i])
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.336.7. Exceptions

This instruction does not generate synchronous exceptions.

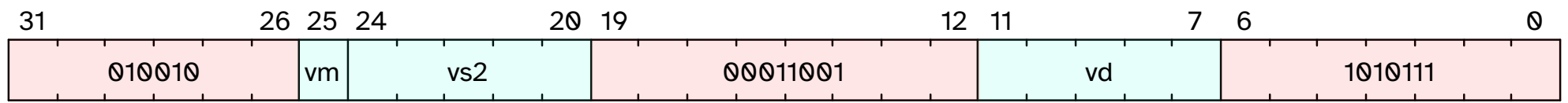
C.337. vfcvt.f.x.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.337.1. Encoding



C.337.2. Description

No description available.

C.337.3. Access

M	S	U
Always	Always	Always

C.337.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.337.5. IDL Operation

C.337.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary0 {
        FV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi32(rm_3b, vs2_val[i]),
            64 => riscv_f64ToUi64(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FV_CVT_X_F      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
```

```

        16 => riscv_f16ToI16(rm_3b, vs2_val[i]),
        32 => riscv_f32ToI32(rm_3b, vs2_val[i]),
        64 => riscv_f64ToI64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_XU    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
        32 => riscv_ui32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_ui64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_X    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
        32 => riscv_i32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_i64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_XU_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToUi16(0b001, vs2_val[i]),
        32 => riscv_f32ToUi32(0b001, vs2_val[i]),
        64 => riscv_f64ToUi64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_X_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToI16(0b001, vs2_val[i]),
        32 => riscv_f32ToI32(0b001, vs2_val[i]),
        64 => riscv_f64ToI64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
}
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.337.7. Exceptions

This instruction does not generate synchronous exceptions.

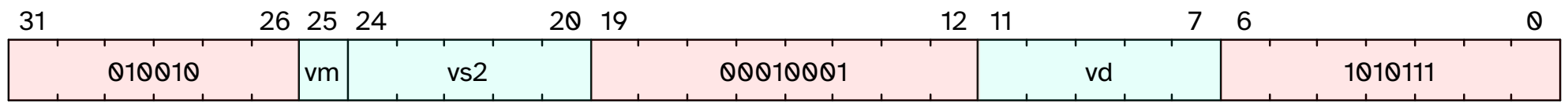
C.338. vfcvt.f.xu.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.338.1. Encoding



C.338.2. Description

No description available.

C.338.3. Access

M	S	U
Always	Always	Always

C.338.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.338.5. IDL Operation

C.338.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary0 {
        FV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi32(rm_3b, vs2_val[i]),
            64 => riscv_f64ToUi64(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FV_CVT_X_F      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
```

```

        16 => riscv_f16ToI16(rm_3b, vs2_val[i]),
        32 => riscv_f32ToI32(rm_3b, vs2_val[i]),
        64 => riscv_f64ToI64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_XU    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
        32 => riscv_ui32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_ui64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_X    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
        32 => riscv_i32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_i64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_XU_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToUi16(0b001, vs2_val[i]),
        32 => riscv_f32ToUi32(0b001, vs2_val[i]),
        64 => riscv_f64ToUi64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_X_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToI16(0b001, vs2_val[i]),
        32 => riscv_f32ToI32(0b001, vs2_val[i]),
        64 => riscv_f64ToI64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
}
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.338.7. Exceptions

This instruction does not generate synchronous exceptions.

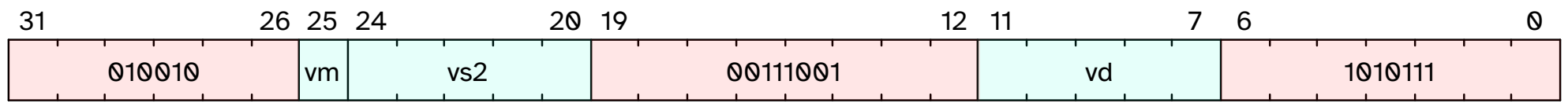
C.339. vfcvt.rtz.x.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.339.1. Encoding



C.339.2. Description

No description available.

C.339.3. Access

M	S	U
Always	Always	Always

C.339.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.339.5. IDL Operation

C.339.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary0 {
        FV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi32(rm_3b, vs2_val[i]),
            64 => riscv_f64ToUi64(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FV_CVT_X_F      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
```



```

        16 => riscv_f16ToI16(rm_3b, vs2_val[i]),
        32 => riscv_f32ToI32(rm_3b, vs2_val[i]),
        64 => riscv_f64ToI64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_XU    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
        32 => riscv_ui32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_ui64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_X    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
        32 => riscv_i32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_i64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_XU_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToUi16(0b001, vs2_val[i]),
        32 => riscv_f32ToUi32(0b001, vs2_val[i]),
        64 => riscv_f64ToUi64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_X_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToI16(0b001, vs2_val[i]),
        32 => riscv_f32ToI32(0b001, vs2_val[i]),
        64 => riscv_f64ToI64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
}
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.339.7. Exceptions

This instruction does not generate synchronous exceptions.

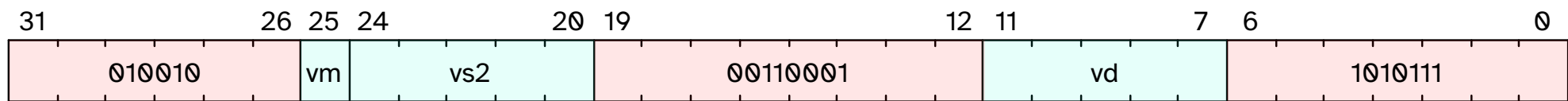
C.340. vfcvt.rtz.xu.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.340.1. Encoding



C.340.2. Description

No description available.

C.340.3. Access

M	S	U
Always	Always	Always

C.340.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.340.5. IDL Operation

C.340.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary0 {
        FV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi32(rm_3b, vs2_val[i]),
            64 => riscv_f64ToUi64(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FV_CVT_X_F      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
```



```

        16 => riscv_f16ToI16(rm_3b, vs2_val[i]),
        32 => riscv_f32ToI32(rm_3b, vs2_val[i]),
        64 => riscv_f64ToI64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_XU    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
        32 => riscv_ui32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_ui64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_X    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
        32 => riscv_i32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_i64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_XU_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToUi16(0b001, vs2_val[i]),
        32 => riscv_f32ToUi32(0b001, vs2_val[i]),
        64 => riscv_f64ToUi64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_X_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToI16(0b001, vs2_val[i]),
        32 => riscv_f32ToI32(0b001, vs2_val[i]),
        64 => riscv_f64ToI64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
}
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.340.7. Exceptions

This instruction does not generate synchronous exceptions.

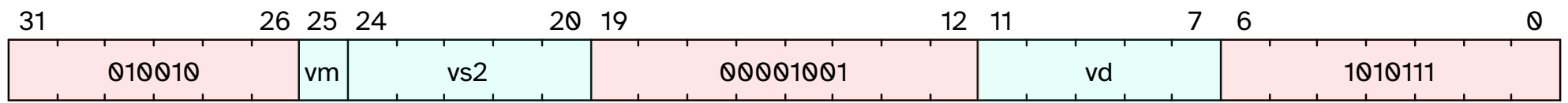
C.341. vfcvt.x.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.341.1. Encoding



C.341.2. Description

No description available.

C.341.3. Access

M	S	U
Always	Always	Always

C.341.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.341.5. IDL Operation

C.341.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary0 {
        FV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi32(rm_3b, vs2_val[i]),
            64 => riscv_f64ToUi64(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FV_CVT_X_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
```

```

        16 => riscv_f16ToI16(rm_3b, vs2_val[i]),
        32 => riscv_f32ToI32(rm_3b, vs2_val[i]),
        64 => riscv_f64ToI64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_XU    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
        32 => riscv_ui32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_ui64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_X    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
        32 => riscv_i32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_i64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_XU_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToUi16(0b001, vs2_val[i]),
        32 => riscv_f32ToUi32(0b001, vs2_val[i]),
        64 => riscv_f64ToUi64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_X_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToI16(0b001, vs2_val[i]),
        32 => riscv_f32ToI32(0b001, vs2_val[i]),
        64 => riscv_f64ToI64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
}
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.341.7. Exceptions

This instruction does not generate synchronous exceptions.

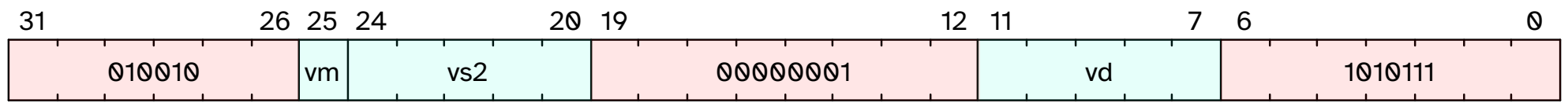
C.342. vfcvt.xu.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.342.1. Encoding



C.342.2. Description

No description available.

C.342.3. Access

M	S	U
Always	Always	Always

C.342.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.342.5. IDL Operation

C.342.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary0 {
        FV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi32(rm_3b, vs2_val[i]),
            64 => riscv_f64ToUi64(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FV_CVT_X_F      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
```

```

        16 => riscv_f16ToI16(rm_3b, vs2_val[i]),
        32 => riscv_f32ToI32(rm_3b, vs2_val[i]),
        64 => riscv_f64ToI64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_XU    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
        32 => riscv_ui32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_ui64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_F_X    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
        32 => riscv_i32ToF32(rm_3b, vs2_val[i]),
        64 => riscv_i64ToF64(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_XU_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToUi16(0b001, vs2_val[i]),
        32 => riscv_f32ToUi32(0b001, vs2_val[i]),
        64 => riscv_f64ToUi64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FV_CVT_RTZ_X_F => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16ToI16(0b001, vs2_val[i]),
        32 => riscv_f32ToI32(0b001, vs2_val[i]),
        64 => riscv_f64ToI64(0b001, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
}
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.342.7. Exceptions

This instruction does not generate synchronous exceptions.

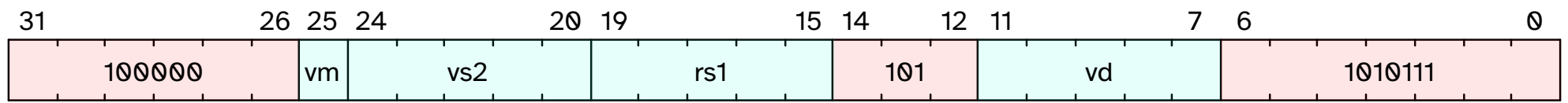
C.343. vfddiv.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.343.1. Encoding



C.343.2. Description

No description available.

C.343.3. Access

M	S	U
Always	Always	Always

C.343.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.343.5. IDL Operation

C.343.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```



```
VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJJ     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJJX    => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.343.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

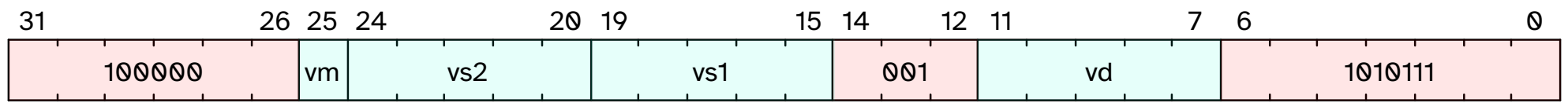
C.344. vfddiv.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.344.1. Encoding



C.344.2. Description

No description available.

C.344.3. Access

M	S	U
Always	Always	Always

C.344.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.344.5. IDL Operation

C.344.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```



```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.344.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

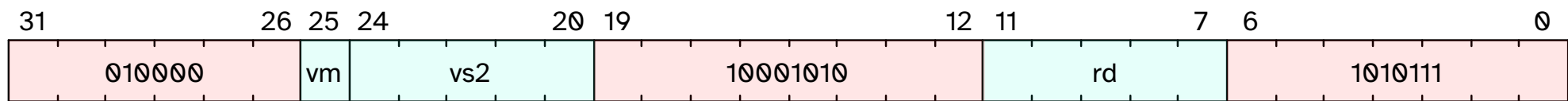
C.345. vfirst.m

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.345.1. Encoding



C.345.2. Description

No description available.

C.345.3. Access

M	S	U
Always	Always	Always

C.345.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rd = $encoding[11:7];
```

C.345.5. IDL Operation

C.345.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() | not(assert_vstart(0)) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, 0, vs2_val, vm_val);

  index : int = -1;
  foreach (i from 0 to (num_elem - 1)) {
    if index == -1 then {
      if mask[i] & vs2_val[i] then index = i;
    };
  };

  X(rd) = to_bits(sizeof(xlen), index);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.345.7. Exceptions

This instruction does not generate synchronous exceptions.

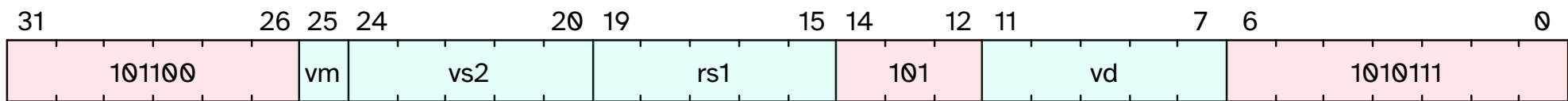
C.346. vfmacc.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.346.1. Encoding



C.346.2. Description

No description available.

C.346.3. Access

M	S	U
Always	Always	Always

C.346.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.346.5. IDL Operation

C.346.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC    => fp_muloadd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC   => fp_nmulo_sub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC    => fp_mulo_sub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC   => fp_nmulo_add(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD    => fp_muloadd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD   => fp_nmulo_sub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB    => fp_mulo_sub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB   => fp_nmulo_add(rm_3b, rs1_val, vd_val[i], vs2_val[i])
      }
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.346.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

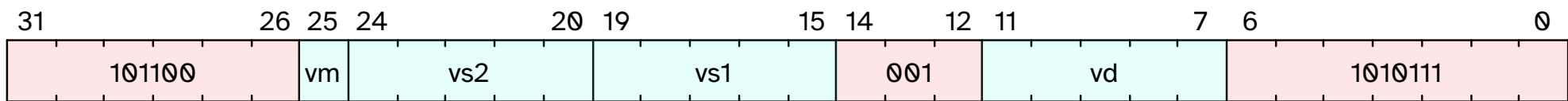
C.347. vfmaccc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.347.1. Encoding



C.347.2. Description

No description available.

C.347.3. Access

M	S	U
Always	Always	Always

C.347.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.347.5. IDL Operation

C.347.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_muloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmulo_sub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mulo_sub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmulo_add(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD     => fp_muloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD    => fp_nmulo_sub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB     => fp_mulo_sub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB    => fp_nmulo_add(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.347.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

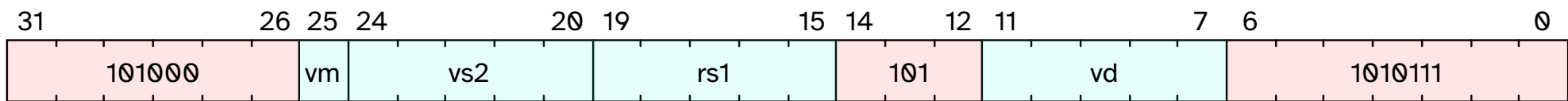
C.348. vfmadd.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.348.1. Encoding



C.348.2. Description

No description available.

C.348.3. Access

M	S	U
Always	Always	Always

C.348.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.348.5. IDL Operation

C.348.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC  => fp_mU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC => fp_nmulsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC  => fp_mU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC => fp_nmuladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD  => fp_mU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD => fp_nmulsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB  => fp_mU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB => fp_nmuladd(rm_3b, rs1_val, vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.348.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

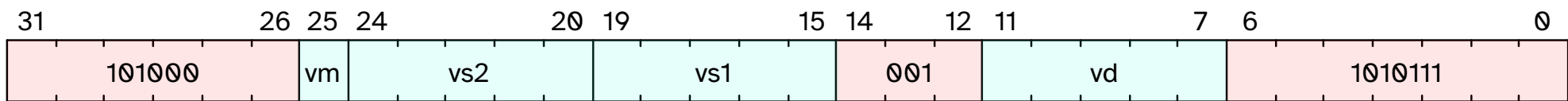
C.349. vfmadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.349.1. Encoding



C.349.2. Description

No description available.

C.349.3. Access

M	S	U
Always	Always	Always

C.349.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.349.5. IDL Operation

C.349.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_mUadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmulsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mUsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmuladd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD    => fp_mUadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD   => fp_nmulsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB    => fp_mUsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB   => fp_nmuladd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```

    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.349.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

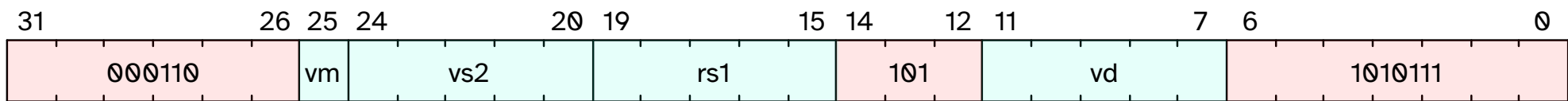
C.350. vfmax.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.350.1. Encoding



C.350.2. Description

No description available.

C.350.3. Access

M	S	U
Always	Always	Always

C.350.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.350.5. IDL Operation

C.350.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.350.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

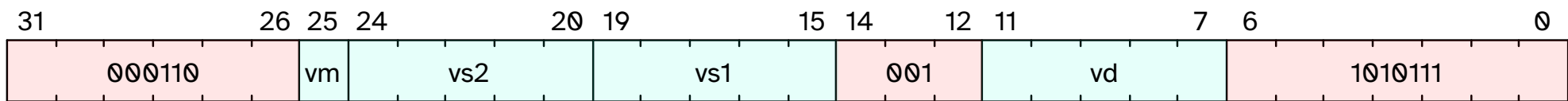
C.351. vfmax.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.351.1. Encoding



C.351.2. Description

No description available.

C.351.3. Access

M	S	U
Always	Always	Always

C.351.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.351.5. IDL Operation

C.351.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.351.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

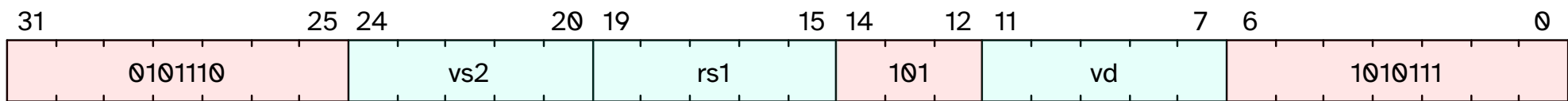
C.352. vfmerge.vfm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.352.1. Encoding



C.352.2. Description

No description available.

C.352.3. Access

M	S	U
Always	Always	Always

C.352.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.352.5. IDL Operation

C.352.6. Sail Operation

```
{
  let rm_3b      = fcsr.FRM();
  let start_element = get_start_element();
  let end_element  = get_end_element();
  let SEW         = get_sew();
  let LMUL_pow    = get_lmUL_pow();
  let num_elem    = get_num_elem(LMUL_pow, SEW); /* max(VLMAX,VLEN/SEW)) */
  let real_num_elem = if LMUL_pow >= 0 then num_elem else num_elem / (0 - LMUL_pow); /* VLMAX */

  if illegal_fp_vd_masked(vd, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, 0b000000);
  let rs1_val : bits('m)                   = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  let tail_ag : agtype = get_vtype_vta();
  foreach (i from 0 to (num_elem - 1)) {
    if i < start_element then {
      result[i] = vd_val[i]
    } else if i > end_element | i >= real_num_elem then {
      result[i] = match tail_ag {
        UNDISTURBED => vd_val[i],
        AGNOSTIC    => vd_val[i] /* TODO: configuration support */
      }
    } else {
      /* the merge operates on all body elements */
      result[i] = if vm_val[i] then rs1_val else vs2_val[i]
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.352.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

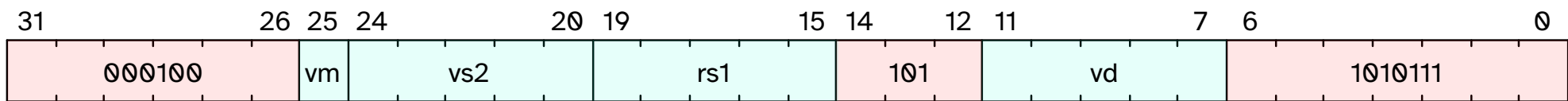
C.353. vfmin.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.353.1. Encoding



C.353.2. Description

No description available.

C.353.3. Access

M	S	U
Always	Always	Always

C.353.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.353.5. IDL Operation

C.353.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.353.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

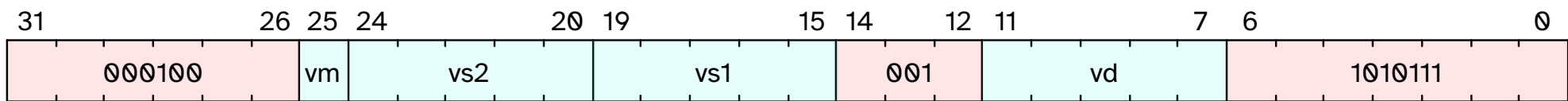
C.354. vfmin.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.354.1. Encoding



C.354.2. Description

No description available.

C.354.3. Access

M	S	U
Always	Always	Always

C.354.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.354.5. IDL Operation

C.354.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.354.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

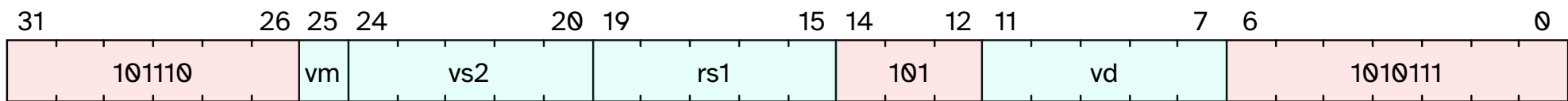
C.355. vfmsac.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.355.1. Encoding



C.355.2. Description

No description available.

C.355.3. Access

M	S	U
Always	Always	Always

C.355.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.355.5. IDL Operation

C.355.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC    => fp_mU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC   => fp_nmulsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC    => fp_mU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC   => fp_nmuladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD    => fp_mU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD   => fp_nmulsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB    => fp_mU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB   => fp_nmuladd(rm_3b, rs1_val, vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.355.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

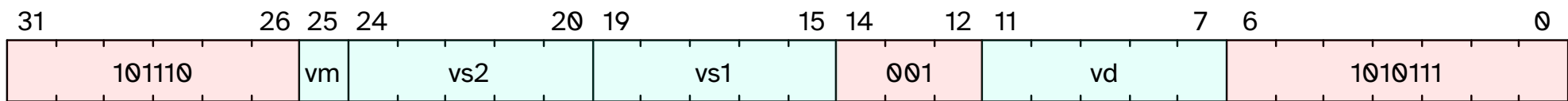
C.356. vfmsac.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.356.1. Encoding



C.356.2. Description

No description available.

C.356.3. Access

M	S	U
Always	Always	Always

C.356.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.356.5. IDL Operation

C.356.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_muloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmulo_sub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mulo_sub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmulo_add(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD    => fp_muloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD   => fp_nmulo_sub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB    => fp_mulo_sub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB   => fp_nmulo_add(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
      }
    }
  }
}
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.356.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

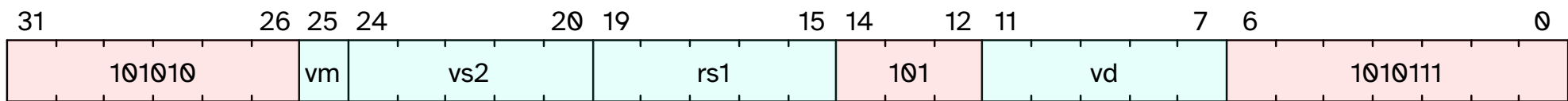
C.357. vfmsub.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.357.1. Encoding



C.357.2. Description

No description available.

C.357.3. Access

M	S	U
Always	Always	Always

C.357.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.357.5. IDL Operation

C.357.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC    => fp_mU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC   => fp_nmU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC    => fp_mU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC   => fp_nmU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD    => fp_mU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD   => fp_nmU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB    => fp_mU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB   => fp_nmU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.357.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

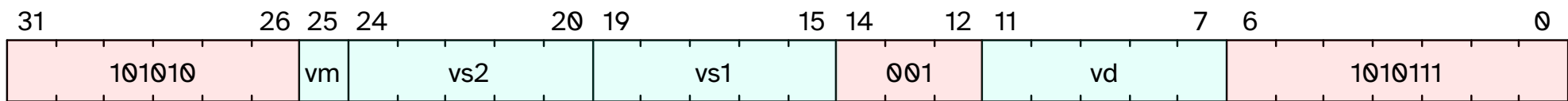
C.358. vfmsub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.358.1. Encoding



C.358.2. Description

No description available.

C.358.3. Access

M	S	U
Always	Always	Always

C.358.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.358.5. IDL Operation

C.358.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_mU_ladd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmulsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mU_lsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmuladd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD    => fp_mU_ladd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD   => fp_nmulsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB    => fp_mU_lsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB   => fp_nmuladd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.358.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

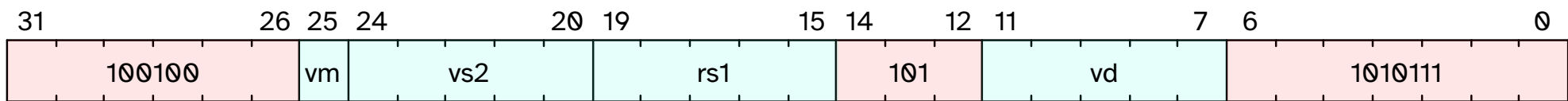
C.359. vfmul.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.359.1. Encoding



C.359.2. Description

No description available.

C.359.3. Access

M	S	U
Always	Always	Always

C.359.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.359.5. IDL Operation

C.359.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.359.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

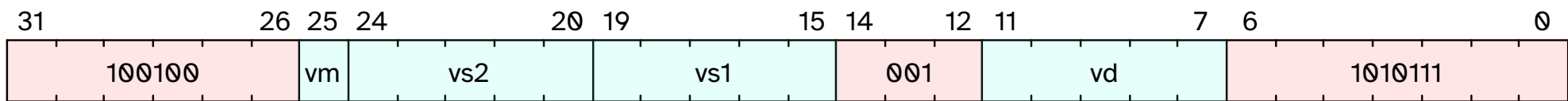
C.360. vfmul.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.360.1. Encoding



C.360.2. Description

No description available.

C.360.3. Access

M	S	U
Always	Always	Always

C.360.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.360.5. IDL Operation

C.360.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.360.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

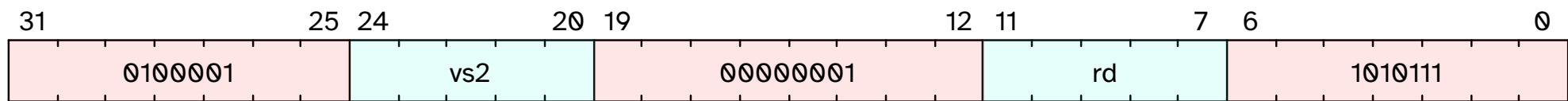
C.361. vfmv.fs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.361.1. Encoding



C.361.2. Description

No description available.

C.361.3. Access

M	S	U
Always	Always	Always

C.361.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rd = $encoding[11:7];
```

C.361.5. IDL Operation

C.361.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let num_elem = get_num_elem(0, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) | SEW > sizeof(flen)
  then { handle_illegal(); return RETIRE_FAIL };
  assert(num_elem > 0 & SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, 0, vs2);
  match 'm {
    16 => F_H(rd) = vs2_val[0],
    32 => F_S(rd) = vs2_val[0],
    64 => F_D(rd) = vs2_val[0]
  };
  vstart = zeros();

  RETIRE_SUCCESS
}
```

C.361.7. Exceptions

This instruction does not generate synchronous exceptions.

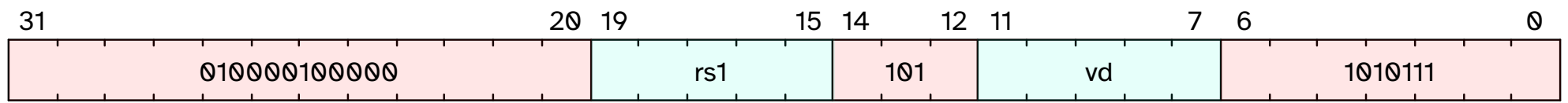
C.362. vfmv.s.f

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.362.1. Encoding



C.362.2. Description

No description available.

C.362.3. Access

M	S	U
Always	Always	Always

C.362.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.362.5. IDL Operation

C.362.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let num_elem = get_num_elem(0, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(num_elem > 0 & SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b1, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, 0, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, 0, vd_val, vm_val);

  /* one body element */
  if mask[0] then result[0] = rs1_val;

  /* others treated as tail elements */
  let tail_ag : agtype = get_vtype_vta();
  foreach (i from 1 to (num_elem - 1)) {
    result[i] = match tail_ag {
      UNDISTURBED => vd_val[i],
      AGNOSTIC    => vd_val[i] /* TODO: configuration support */
    }
  };

  write_vreg(num_elem, SEW, 0, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

```
}

```

C.362.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

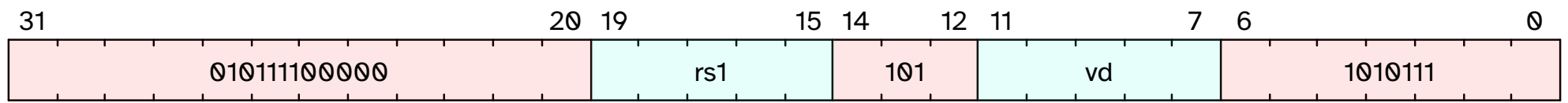
C.363. vfmv.v.f

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.363.1. Encoding



C.363.2. Description

No description available.

C.363.3. Access

M	S	U
Always	Always	Always

C.363.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.363.5. IDL Operation

C.363.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let rs1_val : bits('m)          = get_scalar_fp(rs1, 'm);
  let vm_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b1, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then result[i] = rs1_val
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.363.7. Exceptions

This instruction does not generate synchronous exceptions.

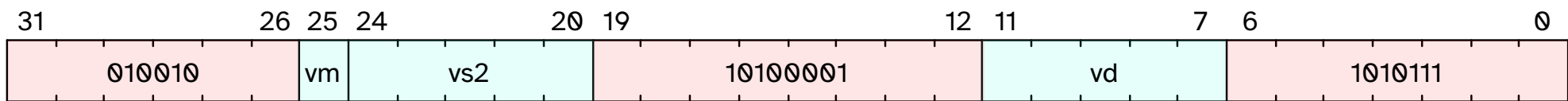
C.364. vfncvt.f.f.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.364.1. Encoding



C.364.2. Description

No description available.

C.364.3. Access

M	S	U
Always	Always	Always

C.364.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.364.5. IDL Operation

C.364.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };

```

```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```


C.364.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

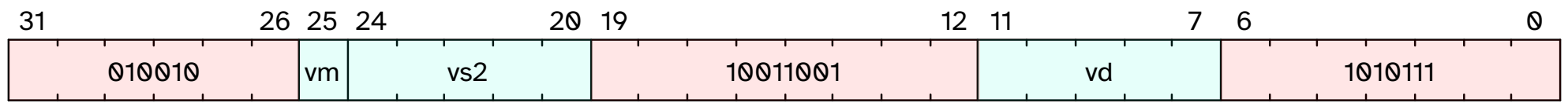
C.365. vfncvt.f.x.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.365.1. Encoding



C.365.2. Description

No description available.

C.365.3. Access

M	S	U
Always	Always	Always

C.365.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.365.5. IDL Operation

C.365.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };
        }
      }
    }
  }
}
```



```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.365.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

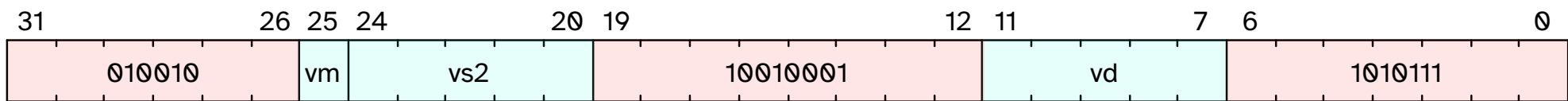
C.366. vfncvt.f.xu.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.366.1. Encoding



C.366.2. Description

No description available.

C.366.3. Access

M	S	U
Always	Always	Always

C.366.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.366.5. IDL Operation

C.366.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };
        }
      }
    }
  }
}
```

```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.366.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

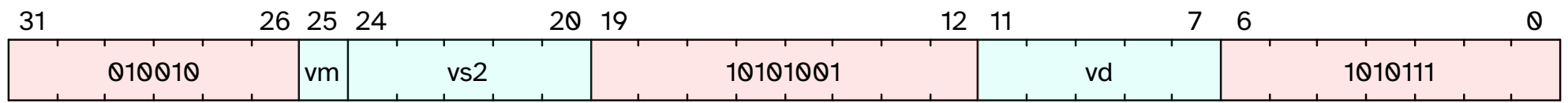
C.367. vfncvt.rod.f.f.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.367.1. Encoding



C.367.2. Description

No description available.

C.367.3. Access

M	S	U
Always	Always	Always

C.367.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.367.5. IDL Operation

C.367.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };
        }
      }
    }
  }
}
```



```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.367.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

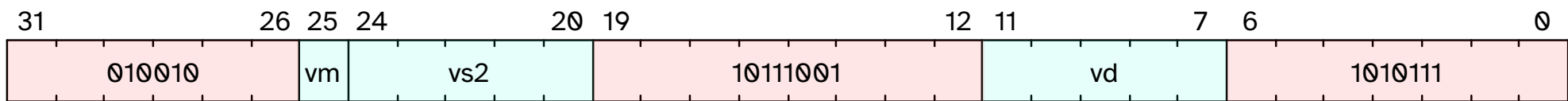
C.368. vfncvt.rtz.x.f.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.368.1. Encoding



C.368.2. Description

No description available.

C.368.3. Access

M	S	U
Always	Always	Always

C.368.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.368.5. IDL Operation

C.368.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };
        }
      }
    }
  }
}
```

```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.368.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

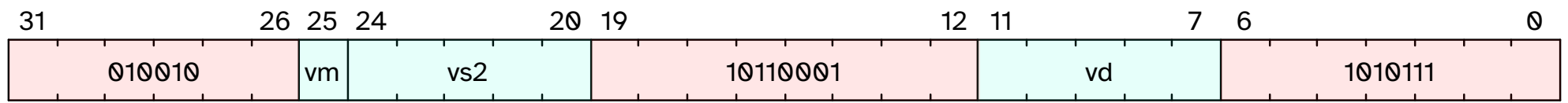
C.369. vfncvt.rtz.xu.f.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.369.1. Encoding



C.369.2. Description

No description available.

C.369.3. Access

M	S	U
Always	Always	Always

C.369.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.369.5. IDL Operation

C.369.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };
        }
      }
    }
  }
}
```

```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.369.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

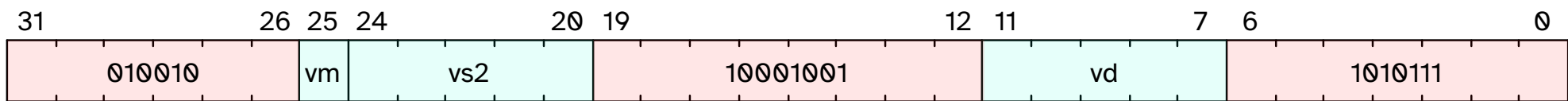
C.370. vfncvt.x.f.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.370.1. Encoding



C.370.2. Description

No description available.

C.370.3. Access

M	S	U
Always	Always	Always

C.370.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.370.5. IDL Operation

C.370.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };

```

```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```


C.370.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

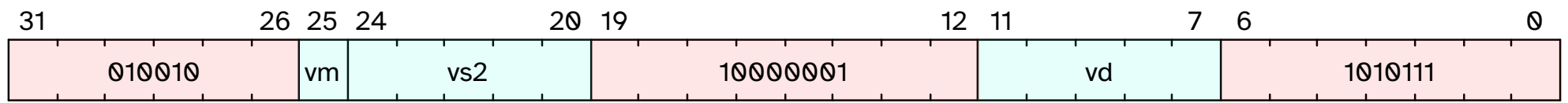
C.371. vfncvt.xu.f.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.371.1. Encoding



C.371.2. Description

No description available.

C.371.3. Access

M	S	U
Always	Always	Always

C.371.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.371.5. IDL Operation

C.371.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfnnunary0 {
        FNV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            8  => riscv_f16ToUi8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToUi16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToUi32(rm_3b, vs2_val[i])
          };
        }
      }
    }
  }
}
```

```

        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(rm_3b, vs2_val[i]),
            16 => riscv_f32ToI16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToI32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_ui32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_ui64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_i32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_i64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(rm_3b, vs2_val[i]),
            32 => riscv_f64ToF32(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_ROD_F_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f32ToF16(0b110, vs2_val[i]),
            32 => riscv_f64ToF32(0b110, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToUi8(0b001, vs2_val[i]),
            16 => riscv_f32ToUi16(0b001, vs2_val[i]),
            32 => riscv_f64ToUi32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FNV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            8 => riscv_f16ToI8(0b001, vs2_val[i]),
            16 => riscv_f32ToI16(0b001, vs2_val[i]),
            32 => riscv_f64ToI32(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.371.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

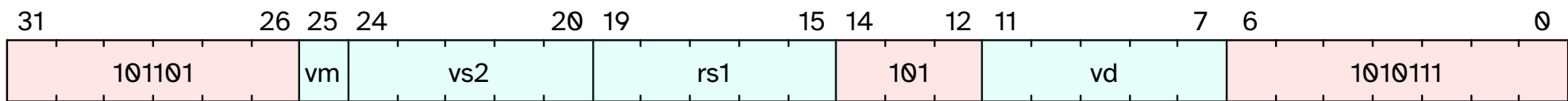
C.372. vfnmacc.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.372.1. Encoding



C.372.2. Description

No description available.

C.372.3. Access

M	S	U
Always	Always	Always

C.372.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.372.5. IDL Operation

C.372.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC    => fp_mU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC   => fp_nmulsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC    => fp_mU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC   => fp_nmuladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD    => fp_mU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD   => fp_nmulsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB    => fp_mU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB   => fp_nmuladd(rm_3b, rs1_val, vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.372.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

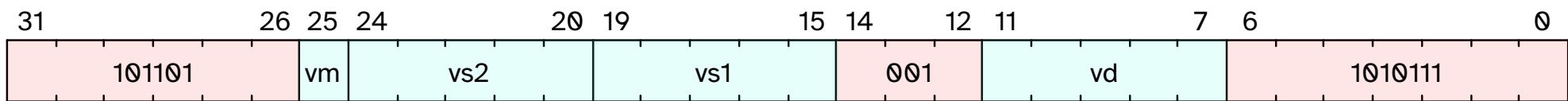
C.373. vfmacc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.373.1. Encoding



C.373.2. Description

No description available.

C.373.3. Access

M	S	U
Always	Always	Always

C.373.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.373.5. IDL Operation

C.373.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_mUadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmUsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mUsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmUadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD    => fp_mUadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD   => fp_nmUsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB    => fp_mUsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB   => fp_nmUadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.373.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

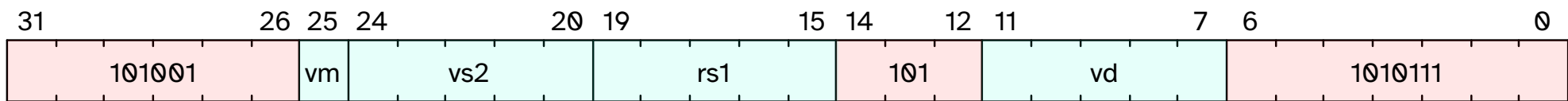
C.374. vfnmadd.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.374.1. Encoding



C.374.2. Description

No description available.

C.374.3. Access

M	S	U
Always	Always	Always

C.374.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.374.5. IDL Operation

C.374.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC  => fp_mU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC => fp_nmU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC  => fp_mU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC => fp_nmU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD  => fp_mU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD => fp_nmU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB  => fp_mU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB => fp_nmU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.374.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

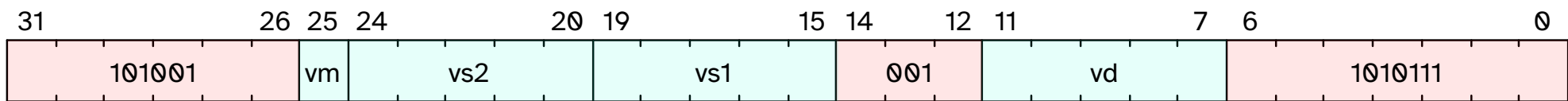
C.375. vfnmadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.375.1. Encoding



C.375.2. Description

No description available.

C.375.3. Access

M	S	U
Always	Always	Always

C.375.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.375.5. IDL Operation

C.375.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMADD => fp_muloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNADD => fp_nmuloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC => fp_muloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC => fp_nmuloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD => fp_muloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNADD => fp_nmuloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB => fp_muloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB => fp_nmuloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.375.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

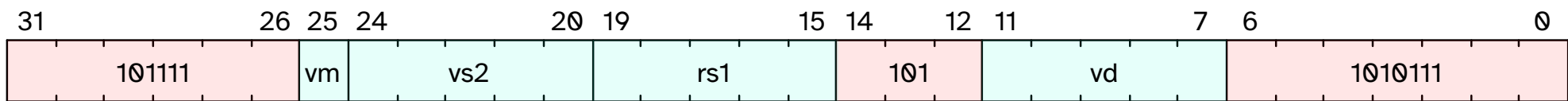
C.376. vfnmsac.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.376.1. Encoding



C.376.2. Description

No description available.

C.376.3. Access

M	S	U
Always	Always	Always

C.376.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.376.5. IDL Operation

C.376.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC    => fp_muloadd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC   => fp_nmulo_sub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC    => fp_mulo_sub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC   => fp_nmulo_add(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD    => fp_muloadd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMAADD  => fp_nmulo_sub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB    => fp_mulo_sub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB   => fp_nmulo_add(rm_3b, rs1_val, vd_val[i], vs2_val[i])
      }
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.376.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

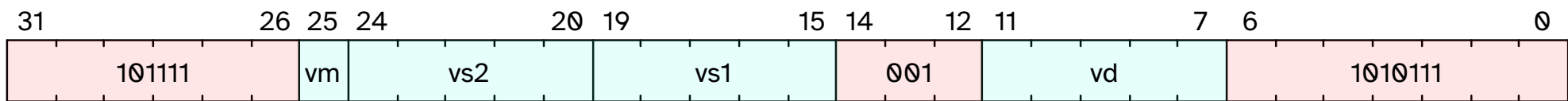
C.377. vfnmsac.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.377.1. Encoding



C.377.2. Description

No description available.

C.377.3. Access

M	S	U
Always	Always	Always

C.377.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.377.5. IDL Operation

C.377.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_mUadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmulsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mUsub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmuladd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD    => fp_mUadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD   => fp_nmulsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB    => fp_mUsub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB   => fp_nmuladd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.377.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

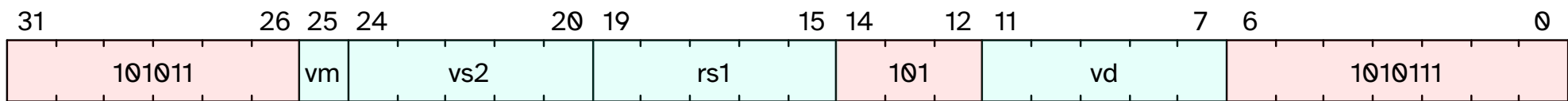
C.378. vfnmsub.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.378.1. Encoding



C.378.2. Description

No description available.

C.378.3. Access

M	S	U
Always	Always	Always

C.378.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.378.5. IDL Operation

C.378.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VMACC  => fp_mU_ladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMACC => fp_nmulsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMSAC  => fp_mU_lsub(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VNMSAC => fp_nmuladd(rm_3b, rs1_val, vs2_val[i], vd_val[i]),
        VF_VMADD  => fp_mU_ladd(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMADD => fp_nmulsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VMSUB  => fp_mU_lsub(rm_3b, rs1_val, vd_val[i], vs2_val[i]),
        VF_VNMSUB => fp_nmuladd(rm_3b, rs1_val, vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.378.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

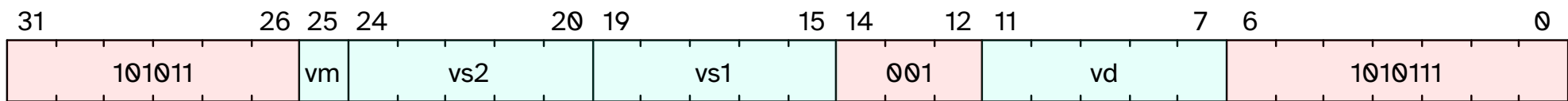
C.379. vfnmsub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.379.1. Encoding



C.379.2. Description

No description available.

C.379.3. Access

M	S	U
Always	Always	Always

C.379.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.379.5. IDL Operation

C.379.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VMACC    => fp_muloadd(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMACC   => fp_nmulo_sub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMSAC    => fp_mulo_sub(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VNMSAC   => fp_nmulo_add(rm_3b, vs1_val[i], vs2_val[i], vd_val[i]),
        FVV_VMADD    => fp_muloadd(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMADD   => fp_nmulo_sub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VMSUB    => fp_mulo_sub(rm_3b, vs1_val[i], vd_val[i], vs2_val[i]),
        FVV_VNMSUB   => fp_nmulo_add(rm_3b, vs1_val[i], vd_val[i], vs2_val[i])
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.379.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

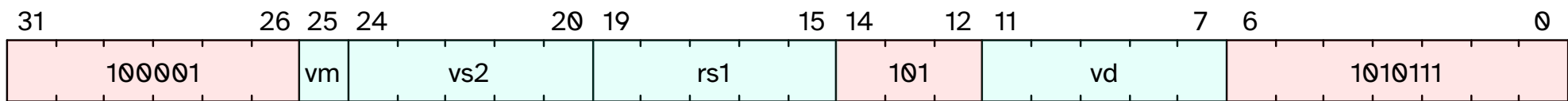
C.380. vfrdiv.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.380.1. Encoding



C.380.2. Description

No description available.

C.380.3. Access

M	S	U
Always	Always	Always

C.380.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.380.5. IDL Operation

C.380.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.380.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

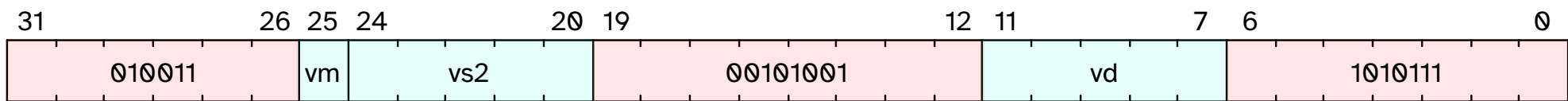
C.381. vfrec7.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.381.1. Encoding



C.381.2. Description

No description available.

C.381.3. Access

M	S	U
Always	Always	Always

C.381.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.381.5. IDL Operation

C.381.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary1 {
        FVV_VSQRT      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16Sqrt(rm_3b, vs2_val[i]),
            32 => riscv_f32Sqrt(rm_3b, vs2_val[i]),
            64 => riscv_f64Sqrt(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FVV_VRSQRT7     => {
```

```

        let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
          16 => riscv_f16Rsqrte7(rm_3b, vs2_val[i]),
          32 => riscv_f32Rsqrte7(rm_3b, vs2_val[i]),
          64 => riscv_f64Rsqrte7(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
      },
    FVV_VREC7 => {
      let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
        16 => riscv_f16Recip7(rm_3b, vs2_val[i]),
        32 => riscv_f32Recip7(rm_3b, vs2_val[i]),
        64 => riscv_f64Recip7(rm_3b, vs2_val[i])
      };
      accrue_fflags(fflags);
      elem
    },
    FVV_VCLASS => fp_class(vs2_val[i])
  }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.381.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

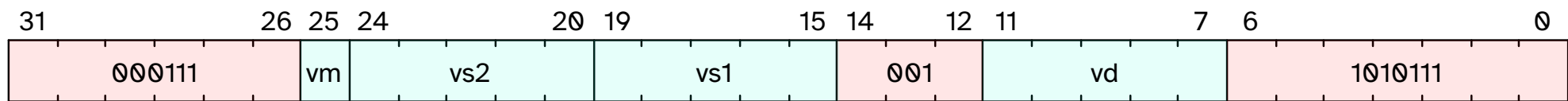
C.382. vfredmax.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.382.1. Encoding



C.382.2. Description

No description available.

C.382.3. Access

M	S	U
Always	Always	Always

C.382.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.382.5. IDL Operation

C.382.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);

  if funct6 == FVV_VFWREDOSUM | funct6 == FVV_VFWREDUSUM then
    process_rfvv_widen(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
  else
    process_rfvv_single(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
}
```

C.382.7. Exceptions

This instruction does not generate synchronous exceptions.

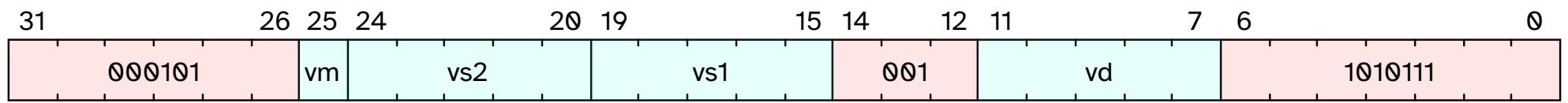
C.383. vfredmin.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.383.1. Encoding



C.383.2. Description

No description available.

C.383.3. Access

M	S	U
Always	Always	Always

C.383.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.383.5. IDL Operation

C.383.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);

  if funct6 == FVV_VFWREDOSUM | funct6 == FVV_VFWREDUSUM then
    process_rfvv_widen(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
  else
    process_rfvv_single(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
}
```

C.383.7. Exceptions

This instruction does not generate synchronous exceptions.

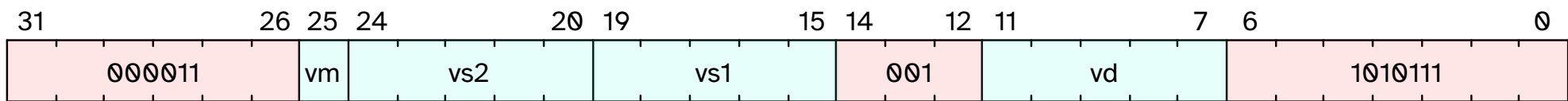
C.384. vfredosum.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.384.1. Encoding



C.384.2. Description

No description available.

C.384.3. Access

M	S	U
Always	Always	Always

C.384.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.384.5. IDL Operation

C.384.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);

  if funct6 == FVV_VFWREDOSUM | funct6 == FVV_VFWREDUSUM then
    process_rfvv_widen(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
  else
    process_rfvv_single(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
}
```

C.384.7. Exceptions

This instruction does not generate synchronous exceptions.

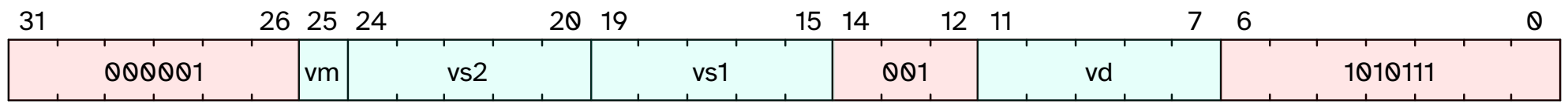
C.385. vfredusum.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.385.1. Encoding



C.385.2. Description

No description available.

C.385.3. Access

M	S	U
Always	Always	Always

C.385.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.385.5. IDL Operation

C.385.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);

  if funct6 == FVV_VFWREDOSUM | funct6 == FVV_VFWREDUSUM then
    process_rfvv_widen(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
  else
    process_rfvv_single(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
}
```

C.385.7. Exceptions

This instruction does not generate synchronous exceptions.

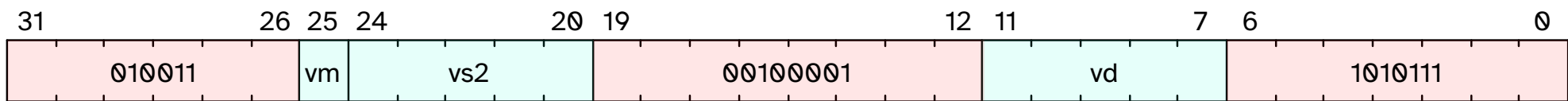
C.386. vfrsqr7.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.386.1. Encoding



C.386.2. Description

No description available.

C.386.3. Access

M	S	U
Always	Always	Always

C.386.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.386.5. IDL Operation

C.386.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary1 {
        FVV_VSQRT      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16Sqrt(rm_3b, vs2_val[i]),
            32 => riscv_f32Sqrt(rm_3b, vs2_val[i]),
            64 => riscv_f64Sqrt(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FVV_VRSQRT7    => {
```

```

        let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
          16 => riscv_f16Rsqrte7(rm_3b, vs2_val[i]),
          32 => riscv_f32Rsqrte7(rm_3b, vs2_val[i]),
          64 => riscv_f64Rsqrte7(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
      },
    FVV_VREC7 => {
      let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
        16 => riscv_f16Recip7(rm_3b, vs2_val[i]),
        32 => riscv_f32Recip7(rm_3b, vs2_val[i]),
        64 => riscv_f64Recip7(rm_3b, vs2_val[i])
      };
      accrue_fflags(fflags);
      elem
    },
    FVV_VCLASS => fp_class(vs2_val[i])
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.386.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

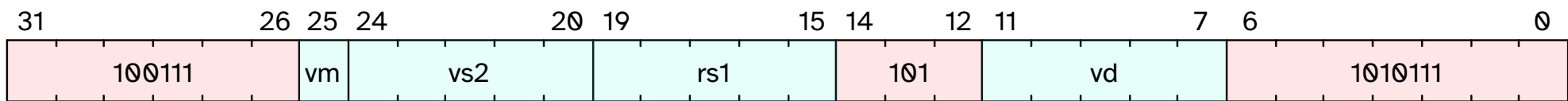
C.387. vfrsub.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.387.1. Encoding



C.387.2. Description

No description available.

C.387.3. Access

M	S	U
Always	Always	Always

C.387.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.387.5. IDL Operation

C.387.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.387.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

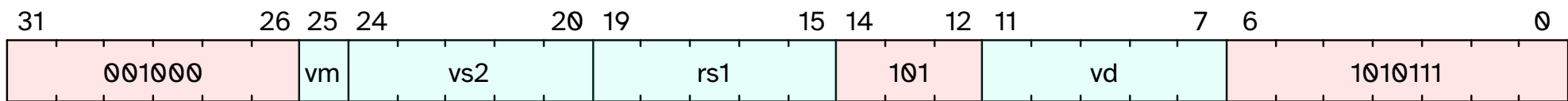
C.388. vfgsnj.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.388.1. Encoding



C.388.2. Description

No description available.

C.388.3. Access

M	S	U
Always	Always	Always

C.388.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.388.5. IDL Operation

C.388.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.388.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

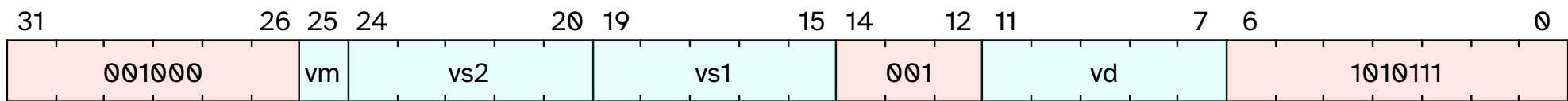
C.389. vfsgnj.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.389.1. Encoding



C.389.2. Description

No description available.

C.389.3. Access

M	S	U
Always	Always	Always

C.389.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.389.5. IDL Operation

C.389.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.389.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

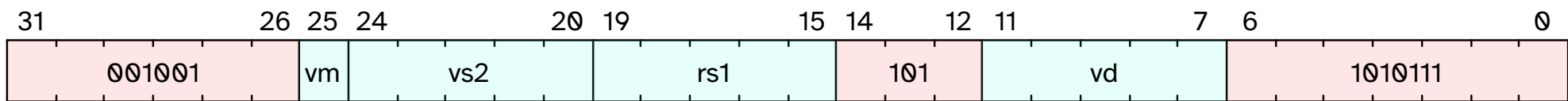
C.390. vfsgnjn.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.390.1. Encoding



C.390.2. Description

No description available.

C.390.3. Access

M	S	U
Always	Always	Always

C.390.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.390.5. IDL Operation

C.390.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.390.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

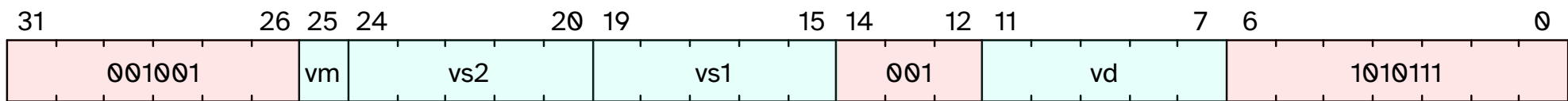
C.391. vfsgnjn.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.391.1. Encoding



C.391.2. Description

No description available.

C.391.3. Access

M	S	U
Always	Always	Always

C.391.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.391.5. IDL Operation

C.391.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.391.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

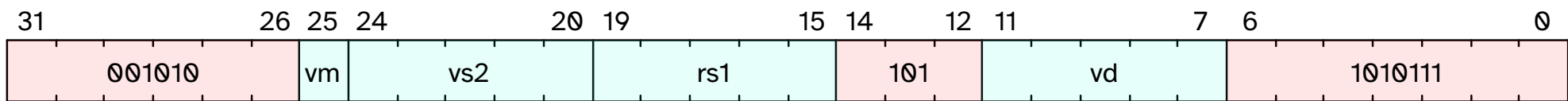
C.392. vfgsnjx.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.392.1. Encoding



C.392.2. Description

No description available.

C.392.3. Access

M	S	U
Always	Always	Always

C.392.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.392.5. IDL Operation

C.392.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.392.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

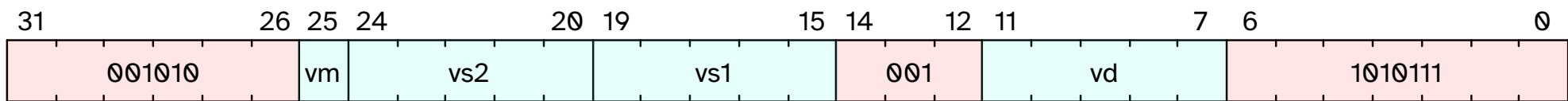
C.393. vfgsnjx.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.393.1. Encoding



C.393.2. Description

No description available.

C.393.3. Access

M	S	U
Always	Always	Always

C.393.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.393.5. IDL Operation

C.393.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.393.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

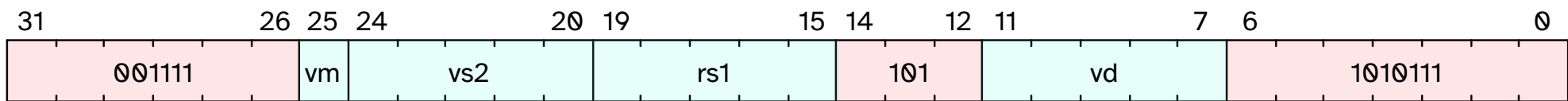
C.394. vfslide1down.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.394.1. Encoding



C.394.2. Description

No description available.

C.394.3. Access

M	S	U
Always	Always	Always

C.394.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.394.5. IDL Operation

C.394.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.394.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

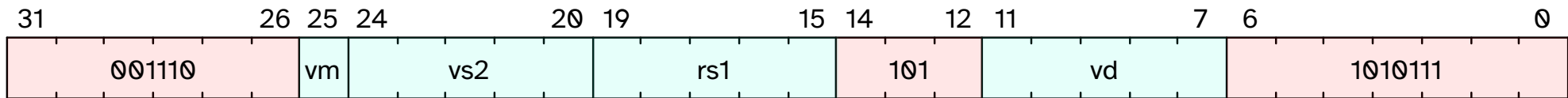
C.395. vfslide1up.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.395.1. Encoding



C.395.2. Description

No description available.

C.395.3. Access

M	S	U
Always	Always	Always

C.395.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.395.5. IDL Operation

C.395.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN    => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.395.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

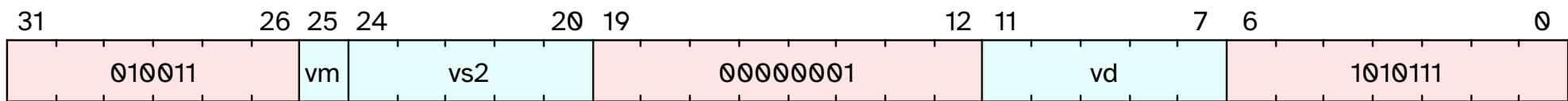
C.396. vfsqrt.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.396.1. Encoding



C.396.2. Description

No description available.

C.396.3. Access

M	S	U
Always	Always	Always

C.396.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.396.5. IDL Operation

C.396.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfunary1 {
        FVV_VSQRT      => {
          let (fflags, elem) : (bits_fflags, bits('m)) = match 'm {
            16 => riscv_f16Sqrt(rm_3b, vs2_val[i]),
            32 => riscv_f32Sqrt(rm_3b, vs2_val[i]),
            64 => riscv_f64Sqrt(rm_3b, vs2_val[i])
          };
          accrue_fflags(fflags);
          elem
        },
        FVV_VRSQRT7     => {
```

```

        let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
            16 => riscv_f16Rsqrte7(rm_3b, vs2_val[i]),
            32 => riscv_f32Rsqrte7(rm_3b, vs2_val[i]),
            64 => riscv_f64Rsqrte7(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
FVV_VREC7    => {
    let (fflags, elem) : (bits_fflags, bits('m')) = match 'm {
        16 => riscv_f16Recip7(rm_3b, vs2_val[i]),
        32 => riscv_f32Recip7(rm_3b, vs2_val[i]),
        64 => riscv_f64Recip7(rm_3b, vs2_val[i])
    };
    accrue_fflags(fflags);
    elem
},
FVV_VCLASS   => fp_class(vs2_val[i])
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.396.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

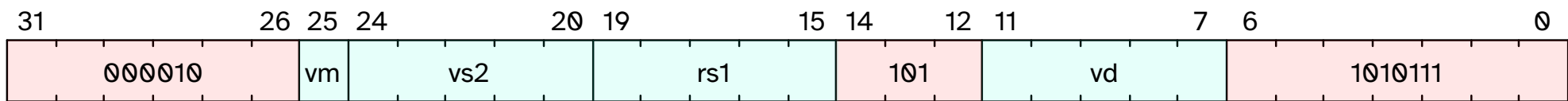
C.397. vsub.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.397.1. Encoding



C.397.2. Description

No description available.

C.397.3. Access

M	S	U
Always	Always	Always

C.397.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.397.5. IDL Operation

C.397.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VF_VADD      => fp_add(rm_3b, vs2_val[i], rs1_val),
        VF_VSUB      => fp_sub(rm_3b, vs2_val[i], rs1_val),
        VF_VRSUB     => fp_sub(rm_3b, rs1_val, vs2_val[i]),
        VF_VMIN      => fp_min(vs2_val[i], rs1_val),
        VF_VMAX      => fp_max(vs2_val[i], rs1_val),
        VF_VMUL      => fp_mul(rm_3b, vs2_val[i], rs1_val),
        VF_VDIV      => fp_div(rm_3b, vs2_val[i], rs1_val),
        VF_VRDIV     => fp_div(rm_3b, rs1_val, vs2_val[i]),
```

```

VF_VSGNJ      => [rs1_val['m - 1]] @ vs2_val[i][('m - 2)..0],
VF_VSGNJN     => (0b1 ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSGNJX     => ([vs2_val[i]['m - 1]] ^ [rs1_val['m - 1]]) @ vs2_val[i][('m - 2)..0],
VF_VSLIDE1UP  => {
    if vs2 == vd then { handle_illegal(); return RETIRE_FAIL };
    if i == 0 then rs1_val else vs2_val[i - 1]
},
VF_VSLIDE1DOWN => {
    let last_elem = get_end_element();
    assert(last_elem < num_elem);
    if i < last_elem then vs2_val[i + 1] else rs1_val
}

}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.397.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

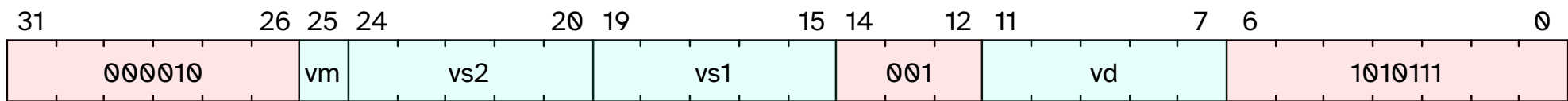
C.398. vsub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.398.1. Encoding



C.398.2. Description

No description available.

C.398.3. Access

M	S	U
Always	Always	Always

C.398.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.398.5. IDL Operation

C.398.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_normal(vd, vm, SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSUB    => fp_sub(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VMIN    => fp_min(vs2_val[i], vs1_val[i]),
        FVV_VMAX    => fp_max(vs2_val[i], vs1_val[i]),
        FVV_VMUL    => fp_mul(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VDIV    => fp_div(rm_3b, vs2_val[i], vs1_val[i]),
        FVV_VSGNJ   => [vs1_val[i]['m - 1]] @ vs2_val[i][('m - 2)..0],
        FVV_VSGNJN  => (0b1 ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0],
```

```

        FVV_VSGNJX    => ([vs2_val[i]['m - 1]] ^ [vs1_val[i]['m - 1]]) @ vs2_val[i][('m - 2)..0]
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.398.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

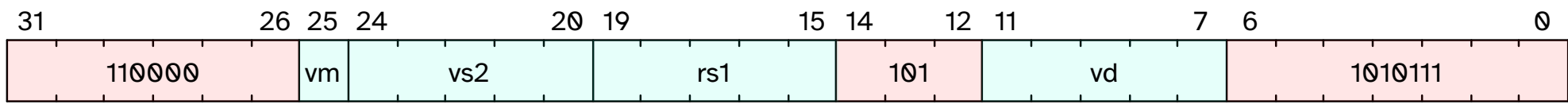
C.399. vfwadd.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.399.1. Encoding



C.399.2. Description

No description available.

C.399.3. Access

M	S	U
Always	Always	Always

C.399.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.399.5. IDL Operation

C.399.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VADD    => fp_add(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val)),
        FWVF_VSUB    => fp_sub(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val)),
        FWVF_VMUL    => fp_mul(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val))
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.399.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

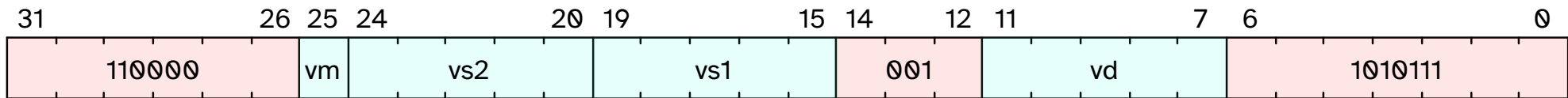
C.400. vfwadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.400.1. Encoding



C.400.2. Description

No description available.

C.400.3. Access

M	S	U
Always	Always	Always

C.400.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.400.5. IDL Operation

C.400.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i])),
        FVV_VSUB    => fp_sub(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i])),
```

```

        FWVW_VMUL    => fp_mul(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.400.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

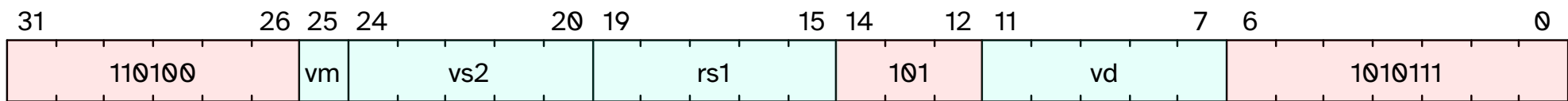
C.401. vfwadd.wf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.401.1. Encoding



C.401.2. Description

No description available.

C.401.3. Access

M	S	U
Always	Always	Always

C.401.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.401.5. IDL Operation

C.401.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen)
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWF_VADD    => fp_add(rm_3b, vs2_val[i], fp_widen(rs1_val)),
        FWF_VSUB    => fp_sub(rm_3b, vs2_val[i], fp_widen(rs1_val))
      }
    }
  }
}
```

```
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.401.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

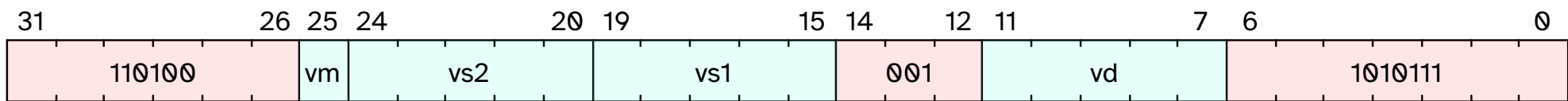
C.402. vfwadd.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.402.1. Encoding



C.402.2. Description

No description available.

C.402.3. Access

M	S	U
Always	Always	Always

C.402.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.402.5. IDL Operation

C.402.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWV_VADD    => fp_add(rm_3b, vs2_val[i], fp_widen(vs1_val[i])),
        FWV_VSUB    => fp_sub(rm_3b, vs2_val[i], fp_widen(vs1_val[i]))
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.402.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

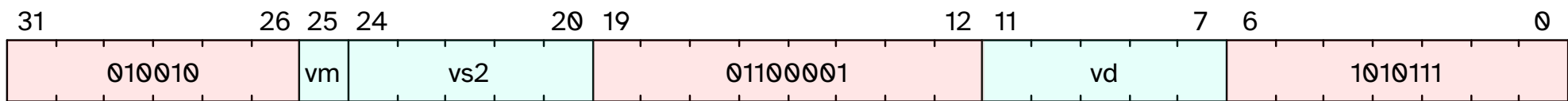
C.403. vfwcvt.f.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.403.1. Encoding



C.403.2. Description

No description available.

C.403.3. Access

M	S	U
Always	Always	Always

C.403.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.403.5. IDL Operation

C.403.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.403.7. Exceptions

This instruction does not generate synchronous exceptions.

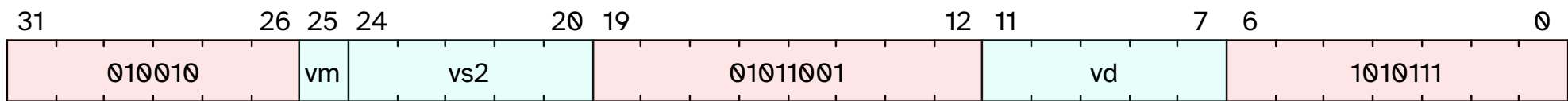
C.404. vfwcvt.f.x.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.404.1. Encoding



C.404.2. Description

No description available.

C.404.3. Access

M	S	U
Always	Always	Always

C.404.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.404.5. IDL Operation

C.404.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.404.7. Exceptions

This instruction does not generate synchronous exceptions.

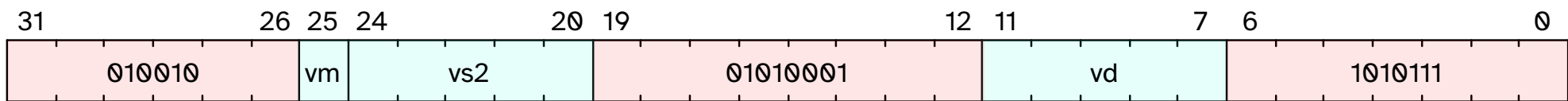
C.405. vfwcvt.f.xu.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.405.1. Encoding



C.405.2. Description

No description available.

C.405.3. Access

M	S	U
Always	Always	Always

C.405.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.405.5. IDL Operation

C.405.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.405.7. Exceptions

This instruction does not generate synchronous exceptions.

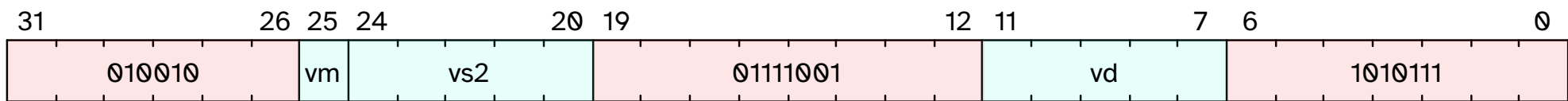
C.406. vfwcvt.rtz.x.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.406.1. Encoding



C.406.2. Description

No description available.

C.406.3. Access

M	S	U
Always	Always	Always

C.406.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.406.5. IDL Operation

C.406.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask       : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.406.7. Exceptions

This instruction does not generate synchronous exceptions.

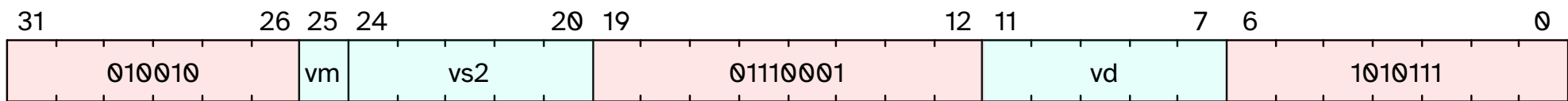
C.407. vfwcvt.rtz.xu.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.407.1. Encoding



C.407.2. Description

No description available.

C.407.3. Access

M	S	U
Always	Always	Always

C.407.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.407.5. IDL Operation

C.407.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.407.7. Exceptions

This instruction does not generate synchronous exceptions.

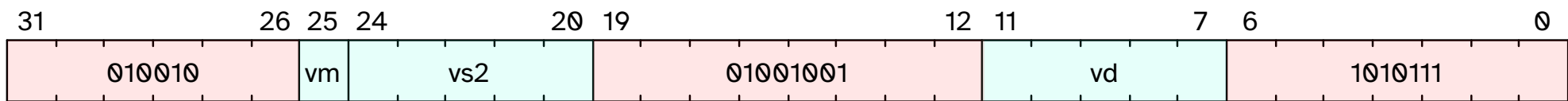
C.408. vfwcvt.x.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.408.1. Encoding



C.408.2. Description

No description available.

C.408.3. Access

M	S	U
Always	Always	Always

C.408.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.408.5. IDL Operation

C.408.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.408.7. Exceptions

This instruction does not generate synchronous exceptions.

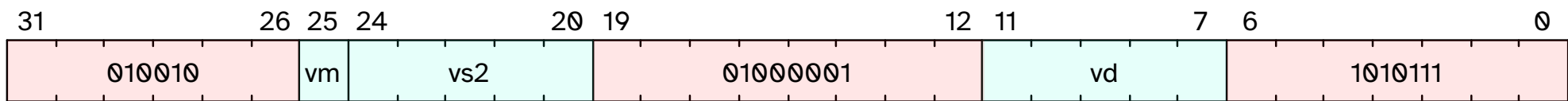
C.409. vfwcvt.xu.f.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.409.1. Encoding



C.409.2. Description

No description available.

C.409.3. Access

M	S	U
Always	Always	Always

C.409.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.409.5. IDL Operation

C.409.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 8 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match vfwunary0 {
        FWV_CVT_XU_F    => {
          let (fflags, elem) : (bits_fflags, bits('o)) = match 'm {
            8  => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToUi64(rm_3b, vs2_val[i])
          }
        }
      }
    }
  }
}
```

```

        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToI64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_XU => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_ui32ToF16(rm_3b, zero_extend(vs2_val[i])),
            16 => riscv_ui32ToF32(rm_3b, zero_extend(vs2_val[i])),
            32 => riscv_ui32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_X => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => riscv_i32ToF16(rm_3b, sign_extend(vs2_val[i])),
            16 => riscv_i32ToF32(rm_3b, sign_extend(vs2_val[i])),
            32 => riscv_i32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_F_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToF32(rm_3b, vs2_val[i]),
            32 => riscv_f32ToF64(rm_3b, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_XU_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToUi32(0b001, vs2_val[i]),
            32 => riscv_f32ToUi64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    },
    FWV_CVT_RTZ_X_F => {
        let (fflags, elem) : (bits_fflags, bits('o')) = match 'm {
            8 => { handle_illegal(); return RETIRE_FAIL },
            16 => riscv_f16ToI32(0b001, vs2_val[i]),
            32 => riscv_f32ToI64(0b001, vs2_val[i])
        };
        accrue_fflags(fflags);
        elem
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.409.7. Exceptions

This instruction does not generate synchronous exceptions.

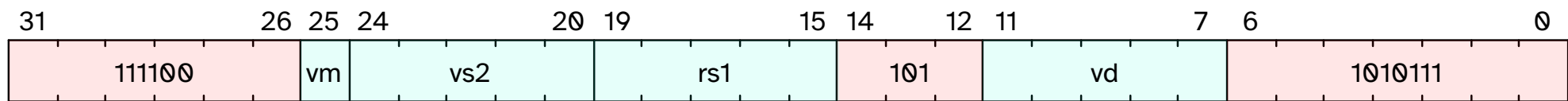
C.410. vfwmacc.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.410.1. Encoding



C.410.2. Description

No description available.

C.410.3. Access

M	S	U
Always	Always	Always

C.410.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.410.5. IDL Operation

C.410.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VMACC => fp_muloadd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VNMACC => fp_nmulo_sub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VMSAC  => fp_mulo_sub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVF_VNMSAC => fp_nmuladd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.410.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

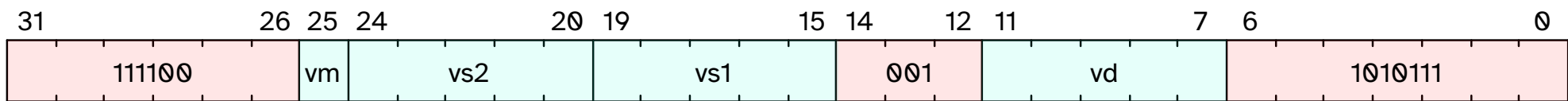
C.411. vfwmaccc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.411.1. Encoding



C.411.2. Description

No description available.

C.411.3. Access

M	S	U
Always	Always	Always

C.411.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.411.5. IDL Operation

C.411.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWV_VMACC => fp_muloadd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWV_VNMACC => fp_nmulo_sub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWV_VMSAC    => fp_mulsub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWV_VNMSAC   => fp_nmuladd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.411.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

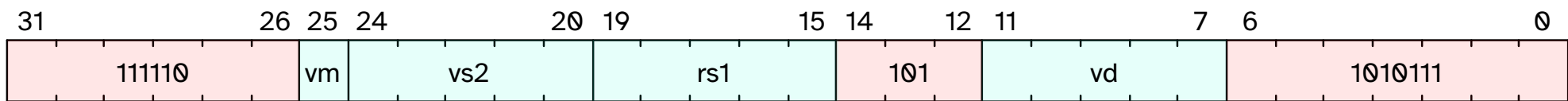
C.412. vfwmsac.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.412.1. Encoding



C.412.2. Description

No description available.

C.412.3. Access

M	S	U
Always	Always	Always

C.412.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.412.5. IDL Operation

C.412.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VMACC => fp_mU_ladd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VNMACC => fp_nmU_lsub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VMSAC  => fp_mU_lsub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVF_VNMSAC => fp_nmuladd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.412.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

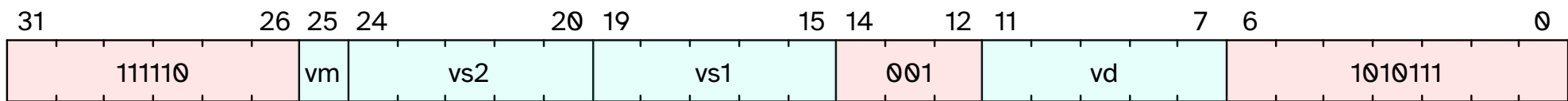
C.413. vfwmsac.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.413.1. Encoding



C.413.2. Description

No description available.

C.413.3. Access

M	S	U
Always	Always	Always

C.413.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.413.5. IDL Operation

C.413.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVV_VMACC => fp_mU_ladd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWVV_VNMACC => fp_nmU_lsub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVW_VMSAC    => fp_mulsub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWVW_VNMSAC   => fp_nmuladd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.413.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

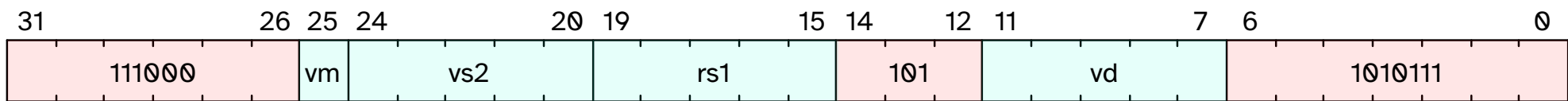
C.414. vfwmul.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.414.1. Encoding



C.414.2. Description

No description available.

C.414.3. Access

M	S	U
Always	Always	Always

C.414.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.414.5. IDL Operation

C.414.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VADD    => fp_add(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val)),
        FWVF_VSUB    => fp_sub(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val)),
        FWVF_VMUL    => fp_mul(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val))
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.414.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

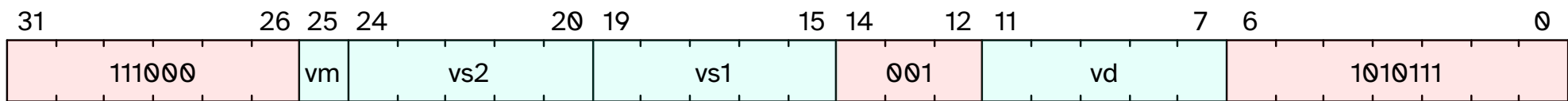
C.415. vfwmul.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.415.1. Encoding



C.415.2. Description

No description available.

C.415.3. Access

M	S	U
Always	Always	Always

C.415.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.415.5. IDL Operation

C.415.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWWV_VADD    => fp_add(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i])),
        FWWV_VSUB    => fp_sub(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i])),
```

```

        FWVW_VMUL    => fp_mul(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.415.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

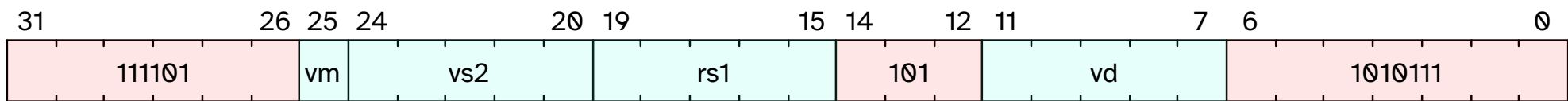
C.416. vfwnmacc.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.416.1. Encoding



C.416.2. Description

No description available.

C.416.3. Access

M	S	U
Always	Always	Always

C.416.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.416.5. IDL Operation

C.416.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VMACC  => fp_muladd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VNMACC => fp_nmULsub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VMSAC  => fp_mULsub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVF_VNMSAC => fp_nmuladd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.416.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

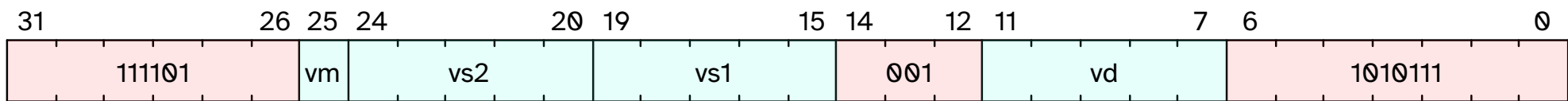
C.417. vfwnmacc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.417.1. Encoding



C.417.2. Description

No description available.

C.417.3. Access

M	S	U
Always	Always	Always

C.417.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.417.5. IDL Operation

C.417.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVV_VMACC => fp_mU_ladd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWVV_VNMACC => fp_nmU_lsub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVV_VMSAC    => fp_mulsub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWVV_VNMSAC   => fp_nmuladd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.417.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

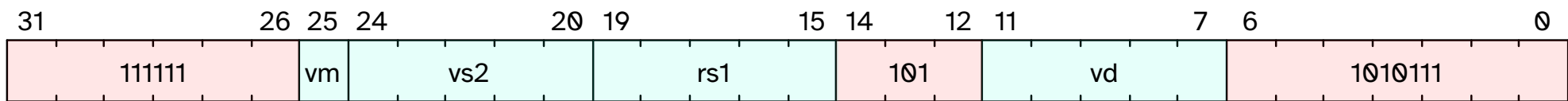
C.418. vfwnmsac.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.418.1. Encoding



C.418.2. Description

No description available.

C.418.3. Access

M	S	U
Always	Always	Always

C.418.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.418.5. IDL Operation

C.418.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VMACC => fp_muloadd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VNMACC => fp_nmulo_sub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
        FWVF_VMSAC  => fp_mulo_sub(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVF_VNMSAC => fp_nmuladd(rm_3b, fp_widen(rs1_val), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.418.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

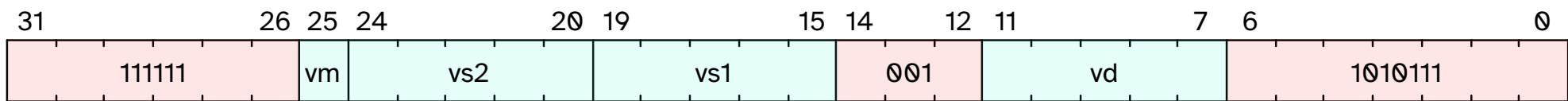
C.419. vfwnmsac.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.419.1. Encoding



C.419.2. Description

No description available.

C.419.3. Access

M	S	U
Always	Always	Always

C.419.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.419.5. IDL Operation

C.419.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVWV_VMACC => fp_muloadd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FVWV_VNMACC => fp_nmulo_sub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
```

```

        FWVW_VMSAC    => fp_mulsub(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i]),
        FWVW_VNMSAC   => fp_nmuladd(rm_3b, fp_widen(vs1_val[i]), fp_widen(vs2_val[i]), vd_val[i])
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.419.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

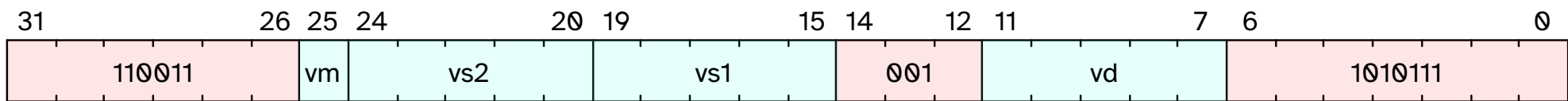
C.420. vfwredosum.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.420.1. Encoding



C.420.2. Description

No description available.

C.420.3. Access

M	S	U
Always	Always	Always

C.420.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.420.5. IDL Operation

C.420.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);

  if funct6 == FVV_VFWREDOSUM | funct6 == FVV_VFWREDUSUM then
    process_rfVV_widen(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
  else
    process_rfVV_single(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
}
```

C.420.7. Exceptions

This instruction does not generate synchronous exceptions.

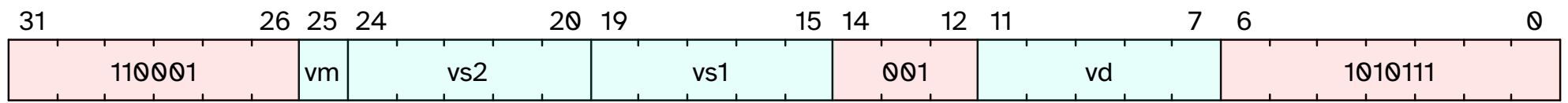
C.421. vfwredusum.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.421.1. Encoding



C.421.2. Description

No description available.

C.421.3. Access

M	S	U
Always	Always	Always

C.421.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.421.5. IDL Operation

C.421.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);

  if funct6 == FVV_VFWREDOSUM | funct6 == FVV_VFWREDUSUM then
    process_rfvv_widen(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
  else
    process_rfvv_single(funct6, vm, vs2, vs1, vd, num_elem_vs, SEW, LMUL_pow)
}
```

C.421.7. Exceptions

This instruction does not generate synchronous exceptions.

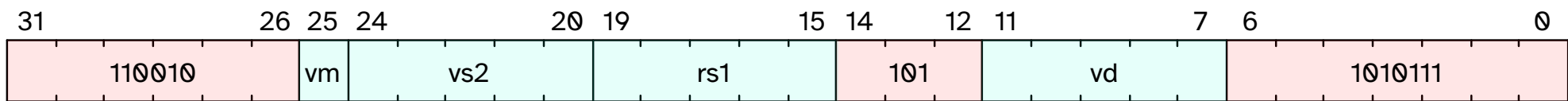
C.422. vfwsb.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.422.1. Encoding



C.422.2. Description

No description available.

C.422.3. Access

M	S	U
Always	Always	Always

C.422.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.422.5. IDL Operation

C.422.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWVF_VADD    => fp_add(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val)),
        FWVF_VSUB    => fp_sub(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val)),
        FWVF_VMUL    => fp_mul(rm_3b, fp_widen(vs2_val[i]), fp_widen(rs1_val))
```

```
    }  
  }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.422.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

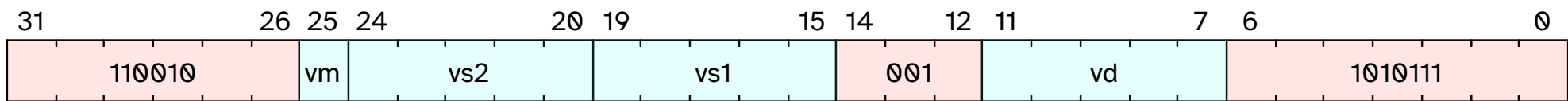
C.423. vfwsb.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.423.1. Encoding



C.423.2. Description

No description available.

C.423.3. Access

M	S	U
Always	Always	Always

C.423.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.423.5. IDL Operation

C.423.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FVV_VADD    => fp_add(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i])),
        FVV_VSUB    => fp_sub(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i])),
```

```

        FWVVMUL    => fp_mul(rm_3b, fp_widen(vs2_val[i]), fp_widen(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.423.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

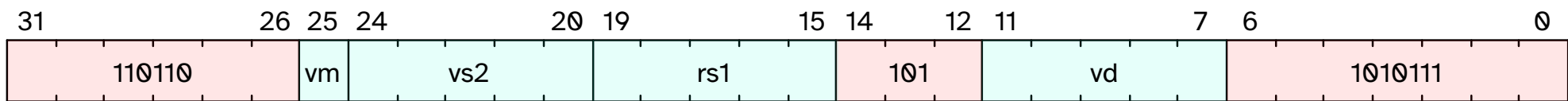
C.424. vfwsb.wf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.424.1. Encoding



C.424.2. Description

No description available.

C.424.3. Access

M	S	U
Always	Always	Always

C.424.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.424.5. IDL Operation

C.424.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen)
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWF_VADD    => fp_add(rm_3b, vs2_val[i], fp_widen(rs1_val)),
        FWF_VSUB    => fp_sub(rm_3b, vs2_val[i], fp_widen(rs1_val))
      }
    }
  }
}
```

```
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.424.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

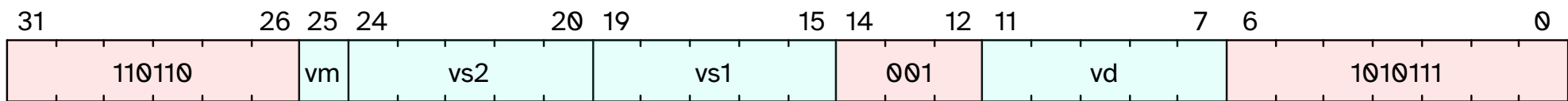
C.425. vfwsb.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.425.1. Encoding



C.425.2. Description

No description available.

C.425.3. Access

M	S	U
Always	Always	Always

C.425.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.425.5. IDL Operation

C.425.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_fp_variable_width(vd, vm, SEW, rm_3b, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW >= 16 & SEW_widen <= 64);

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        FWV_VADD    => fp_add(rm_3b, vs2_val[i], fp_widen(vs1_val[i])),
        FWV_VSUB    => fp_sub(rm_3b, vs2_val[i], fp_widen(vs1_val[i]))
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.425.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

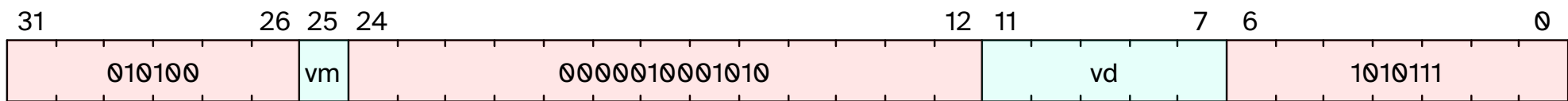
C.426. vid.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.426.1. Encoding



C.426.2. Description

No description available.

C.426.3. Access

M	S	U
Always	Always	Always

C.426.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vd = $encoding[11:7];
```

C.426.5. IDL Operation

C.426.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then result[i] = to_bits(SEW, i)
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.426.7. Exceptions

This instruction does not generate synchronous exceptions.

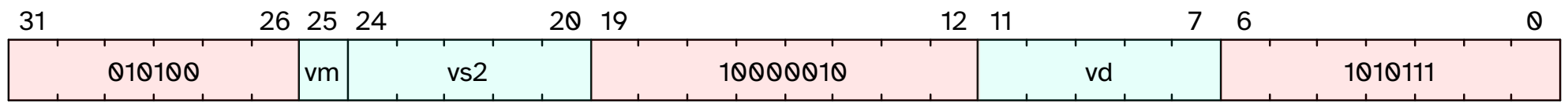
C.427. viota.m

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.427.1. Encoding



C.427.2. Description

No description available.

C.427.3. Access

M	S	U
Always	Always	Always

C.427.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.427.5. IDL Operation

C.427.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) | not(assert_vstart(0)) | vd == vs2
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  sum : int = 0;
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = to_bits(SEW, sum);
      if vs2_val[i] then sum = sum + 1
    }
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.427.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

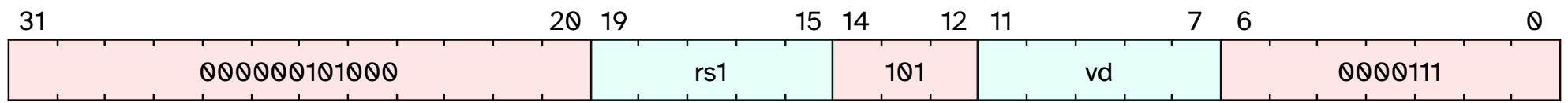
C.428. vl1re16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.428.1. Encoding



C.428.2. Description

No description available.

C.428.3. Access

M	S	U
Always	Always	Always

C.428.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.428.5. IDL Operation

C.428.6. Exceptions

This instruction does not generate synchronous exceptions.

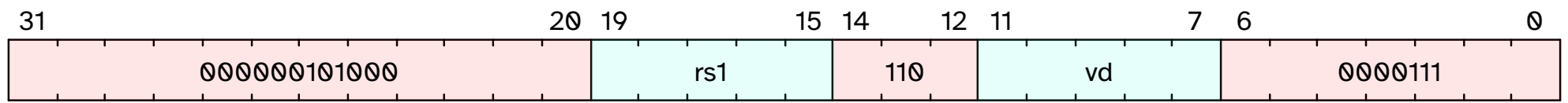
C.429. vl1re32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.429.1. Encoding



C.429.2. Description

No description available.

C.429.3. Access

M	S	U
Always	Always	Always

C.429.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.429.5. IDL Operation

C.429.6. Exceptions

This instruction does not generate synchronous exceptions.

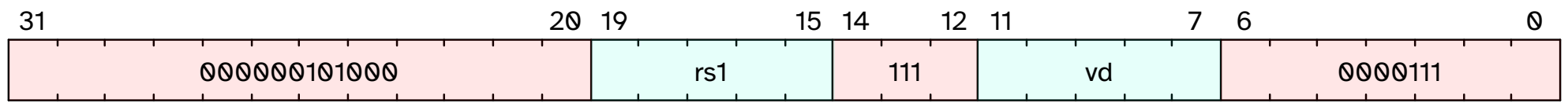
C.430. vl1re64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.430.1. Encoding



C.430.2. Description

No description available.

C.430.3. Access

M	S	U
Always	Always	Always

C.430.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.430.5. IDL Operation

C.430.6. Exceptions

This instruction does not generate synchronous exceptions.

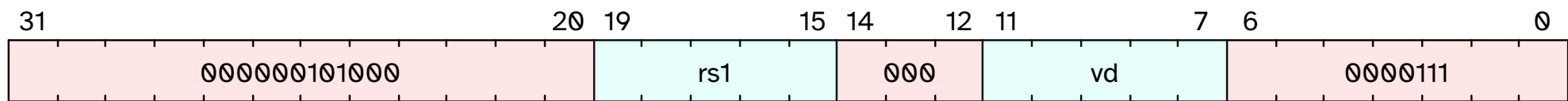
C.431. vl1re8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.431.1. Encoding



C.431.2. Description

No description available.

C.431.3. Access

M	S	U
Always	Always	Always

C.431.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.431.5. IDL Operation

C.431.6. Exceptions

This instruction does not generate synchronous exceptions.

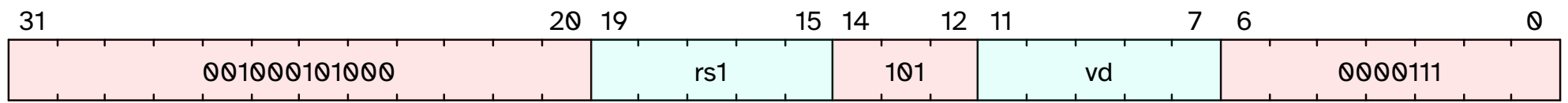
C.432. vl2re16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.432.1. Encoding



C.432.2. Description

No description available.

C.432.3. Access

M	S	U
Always	Always	Always

C.432.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.432.5. IDL Operation

C.432.6. Exceptions

This instruction does not generate synchronous exceptions.

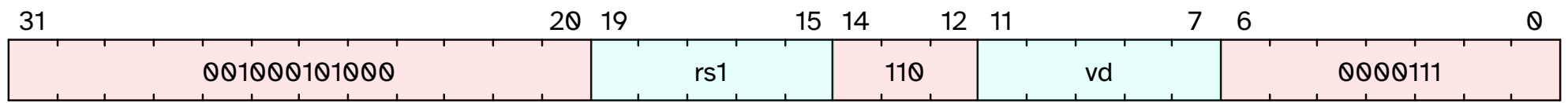
C.433. vl2re32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.433.1. Encoding



C.433.2. Description

No description available.

C.433.3. Access

M	S	U
Always	Always	Always

C.433.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.433.5. IDL Operation

C.433.6. Exceptions

This instruction does not generate synchronous exceptions.

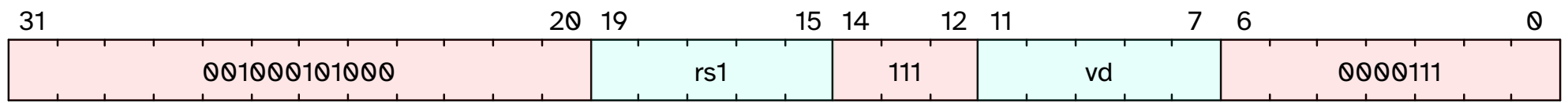
C.434. vl2re64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.434.1. Encoding



C.434.2. Description

No description available.

C.434.3. Access

M	S	U
Always	Always	Always

C.434.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.434.5. IDL Operation

C.434.6. Exceptions

This instruction does not generate synchronous exceptions.

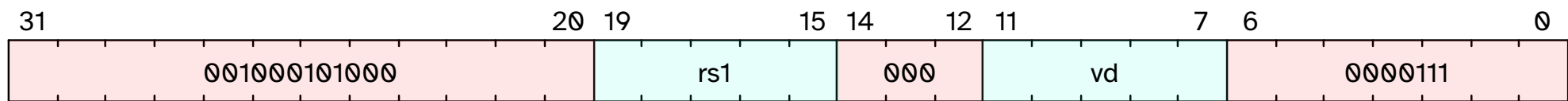
C.435. vl2re8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.435.1. Encoding



C.435.2. Description

No description available.

C.435.3. Access

M	S	U
Always	Always	Always

C.435.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.435.5. IDL Operation

C.435.6. Exceptions

This instruction does not generate synchronous exceptions.

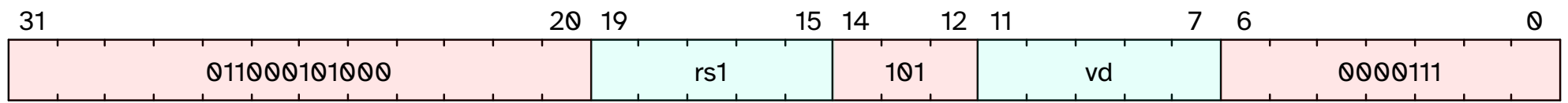
C.436. vl4re16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.436.1. Encoding



C.436.2. Description

No description available.

C.436.3. Access

M	S	U
Always	Always	Always

C.436.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.436.5. IDL Operation

C.436.6. Exceptions

This instruction does not generate synchronous exceptions.

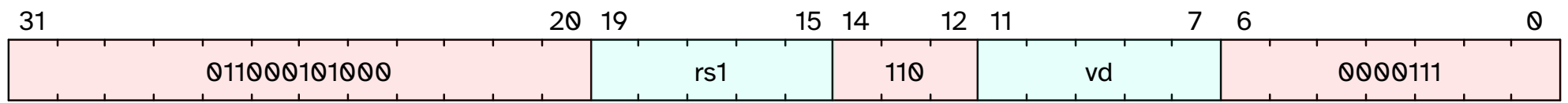
C.437. vl4re32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.437.1. Encoding



C.437.2. Description

No description available.

C.437.3. Access

M	S	U
Always	Always	Always

C.437.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.437.5. IDL Operation

C.437.6. Exceptions

This instruction does not generate synchronous exceptions.

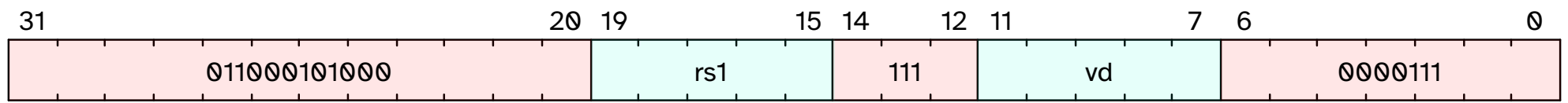
C.438. vl4re64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.438.1. Encoding



C.438.2. Description

No description available.

C.438.3. Access

M	S	U
Always	Always	Always

C.438.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.438.5. IDL Operation

C.438.6. Exceptions

This instruction does not generate synchronous exceptions.

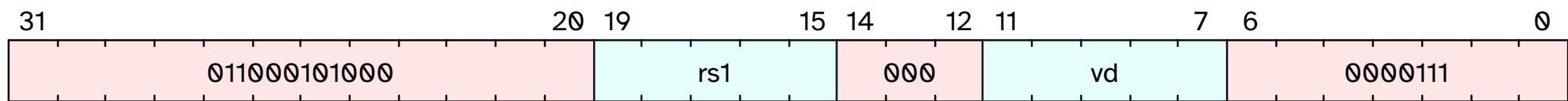
C.439. vl4re8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.439.1. Encoding



C.439.2. Description

No description available.

C.439.3. Access

M	S	U
Always	Always	Always

C.439.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.439.5. IDL Operation

C.439.6. Exceptions

This instruction does not generate synchronous exceptions.

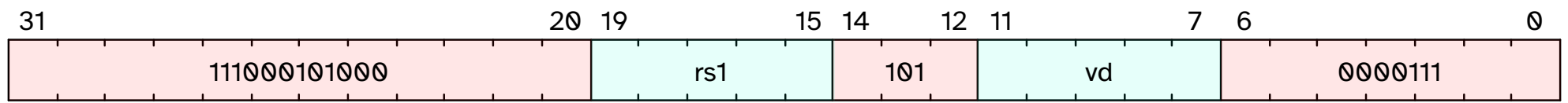
C.440. vl8re16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.440.1. Encoding



C.440.2. Description

No description available.

C.440.3. Access

M	S	U
Always	Always	Always

C.440.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.440.5. IDL Operation

C.440.6. Exceptions

This instruction does not generate synchronous exceptions.

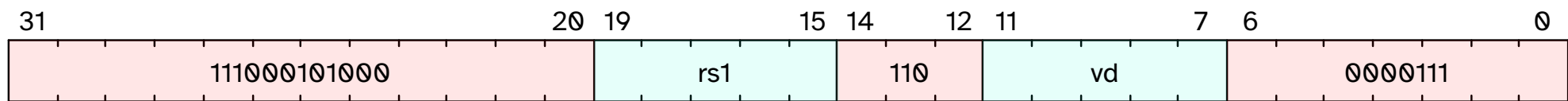
C.441. vl8re32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.441.1. Encoding



C.441.2. Description

No description available.

C.441.3. Access

M	S	U
Always	Always	Always

C.441.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.441.5. IDL Operation

C.441.6. Exceptions

This instruction does not generate synchronous exceptions.

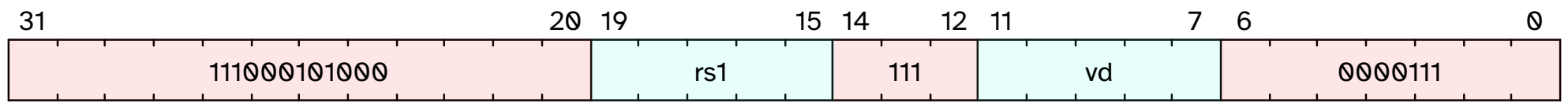
C.442. vl8re64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.442.1. Encoding



C.442.2. Description

No description available.

C.442.3. Access

M	S	U
Always	Always	Always

C.442.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.442.5. IDL Operation

C.442.6. Exceptions

This instruction does not generate synchronous exceptions.

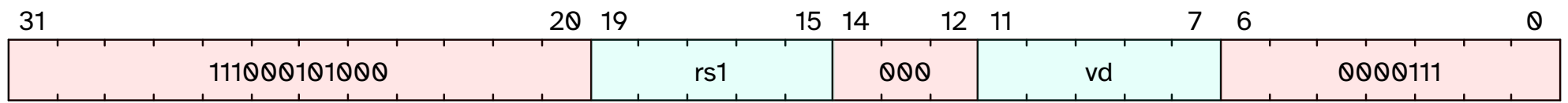
C.443. vl8re8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.443.1. Encoding



C.443.2. Description

No description available.

C.443.3. Access

M	S	U
Always	Always	Always

C.443.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.443.5. IDL Operation

C.443.6. Exceptions

This instruction does not generate synchronous exceptions.

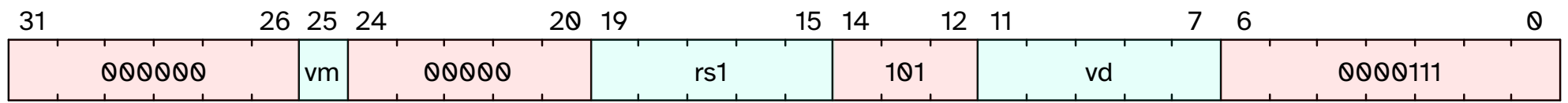
C.444. vle16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.444.1. Encoding



C.444.2. Description

No description available.

C.444.3. Access

M	S	U
Always	Always	Always

C.444.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.444.5. IDL Operation

C.444.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW); /* # of element of each register group */
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlseg(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.444.7. Exceptions

This instruction does not generate synchronous exceptions.

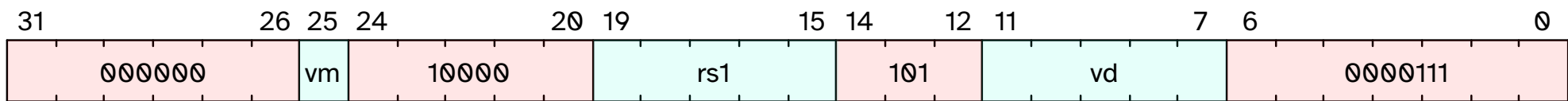
C.445. vle16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.445.1. Encoding



C.445.2. Description

No description available.

C.445.3. Access

M	S	U
Always	Always	Always

C.445.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.445.5. IDL Operation

C.445.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsegff(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.445.7. Exceptions

This instruction does not generate synchronous exceptions.

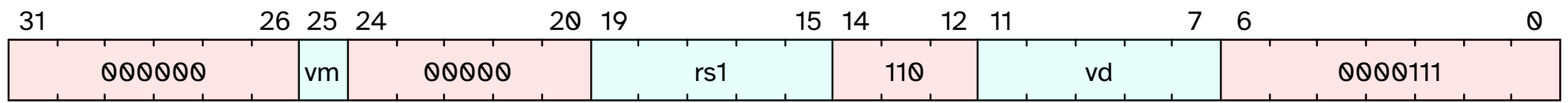
C.446. vle32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.446.1. Encoding



C.446.2. Description

No description available.

C.446.3. Access

M	S	U
Always	Always	Always

C.446.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.446.5. IDL Operation

C.446.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW); /* # of element of each register group */
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlseg(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.446.7. Exceptions

This instruction does not generate synchronous exceptions.

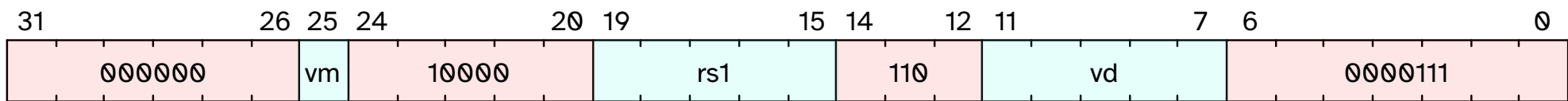
C.447. vle32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.447.1. Encoding



C.447.2. Description

No description available.

C.447.3. Access

M	S	U
Always	Always	Always

C.447.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.447.5. IDL Operation

C.447.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsegff(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.447.7. Exceptions

This instruction does not generate synchronous exceptions.

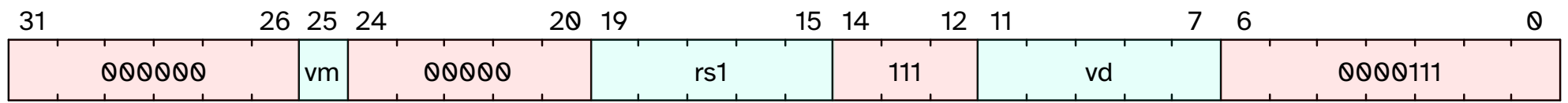
C.448. vle64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.448.1. Encoding



C.448.2. Description

No description available.

C.448.3. Access

M	S	U
Always	Always	Always

C.448.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.448.5. IDL Operation

C.448.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW); /* # of element of each register group */
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlseg(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.448.7. Exceptions

This instruction does not generate synchronous exceptions.

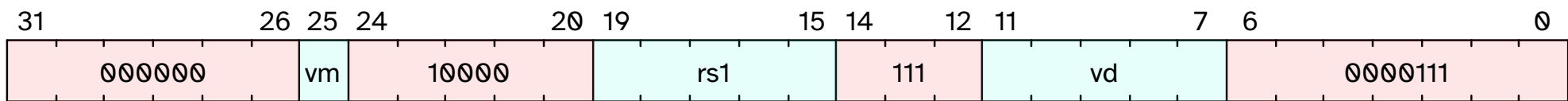
C.449. vle64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.449.1. Encoding



C.449.2. Description

No description available.

C.449.3. Access

M	S	U
Always	Always	Always

C.449.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.449.5. IDL Operation

C.449.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsegff(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.449.7. Exceptions

This instruction does not generate synchronous exceptions.

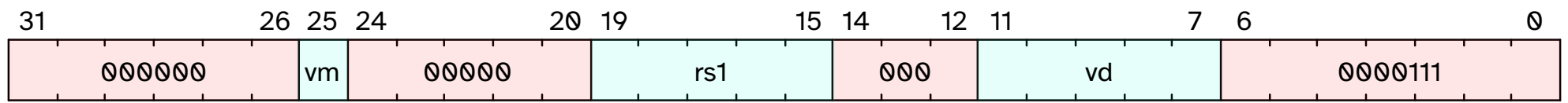
C.450. vle8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.450.1. Encoding



C.450.2. Description

No description available.

C.450.3. Access

M	S	U
Always	Always	Always

C.450.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.450.5. IDL Operation

C.450.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW); /* # of element of each register group */
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlseg(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.450.7. Exceptions

This instruction does not generate synchronous exceptions.

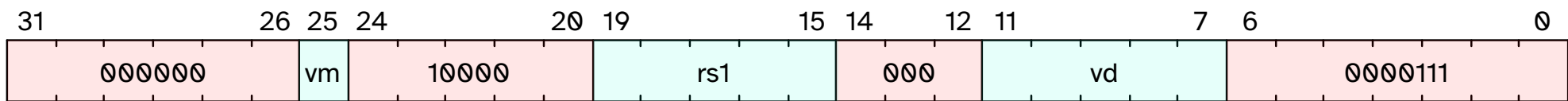
C.451. vle8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.451.1. Encoding



C.451.2. Description

No description available.

C.451.3. Access

M	S	U
Always	Always	Always

C.451.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.451.5. IDL Operation

C.451.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsegff(nf_int, vm, vd, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.451.7. Exceptions

This instruction does not generate synchronous exceptions.

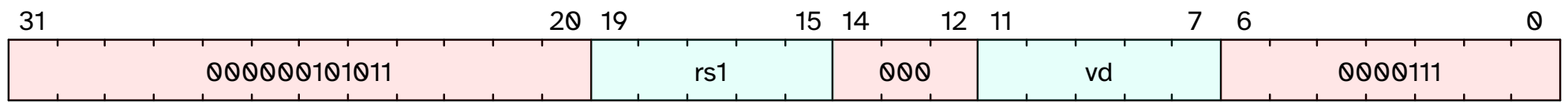
C.452. vlm.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.452.1. Encoding



C.452.2. Description

No description available.

C.452.3. Access

M	S	U
Always	Always	Always

C.452.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.452.5. IDL Operation

C.452.6. Sail Operation

```
{
  let EEW = 8;
  let EMUL_pow = 0;
  let vl_val = unsigned(vl);
  let evl : int = if vl_val % 8 == 0 then vl_val / 8 else vl_val / 8 + 1; /* the effective vector length is
evl=ceil(vl/8) */
  let num_elem = get_num_elem(EMUL_pow, EEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  assert(evl >= 0);
  process_vm(vd_or_vs3, rs1, num_elem, evl, op)
}
```

C.452.7. Exceptions

This instruction does not generate synchronous exceptions.

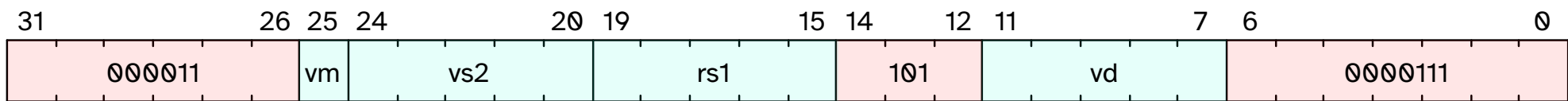
C.453. vloxei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.453.1. Encoding



C.453.2. Description

No description available.

C.453.3. Access

M	S	U
Always	Always	Always

C.453.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.453.5. IDL Operation

C.453.6. Sail Operation

```
{
  let EEW_index_pow = vewidth_pow(width);
  let EEW_index_bytes = vewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmul_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 3)
}
```

C.453.7. Exceptions

This instruction does not generate synchronous exceptions.

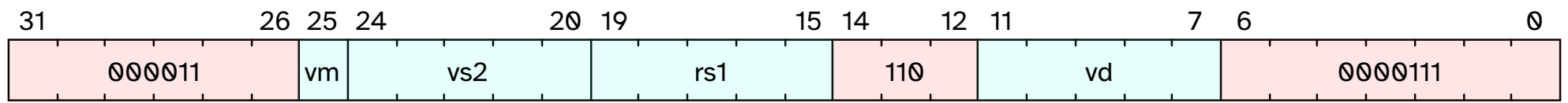
C.454. vloxei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.454.1. Encoding



C.454.2. Description

No description available.

C.454.3. Access

M	S	U
Always	Always	Always

C.454.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.454.5. IDL Operation

C.454.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 3)
}
```

C.454.7. Exceptions

This instruction does not generate synchronous exceptions.

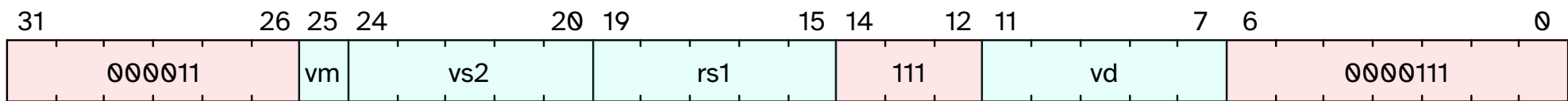
C.455. vloxei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.455.1. Encoding



C.455.2. Description

No description available.

C.455.3. Access

M	S	U
Always	Always	Always

C.455.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.455.5. IDL Operation

C.455.6. Sail Operation

```
{
  let EEW_index_pow = vewidth_pow(width);
  let EEW_index_bytes = vewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 3)
}
```

C.455.7. Exceptions

This instruction does not generate synchronous exceptions.

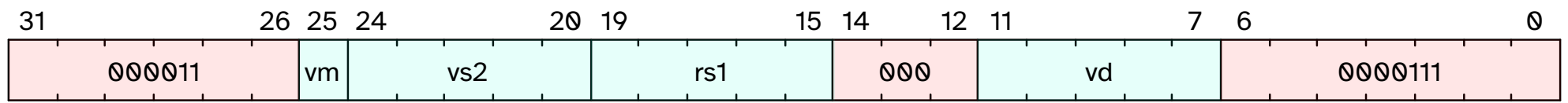
C.456. vloxei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.456.1. Encoding



C.456.2. Description

No description available.

C.456.3. Access

M	S	U
Always	Always	Always

C.456.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.456.5. IDL Operation

C.456.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 3)
}
```

C.456.7. Exceptions

This instruction does not generate synchronous exceptions.

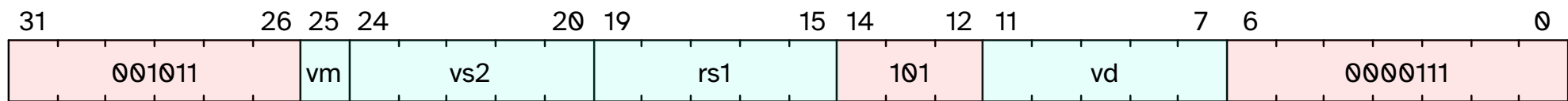
C.457. vloxseg2ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.457.1. Encoding



C.457.2. Description

No description available.

C.457.3. Access

M	S	U
Always	Always	Always

C.457.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.457.5. IDL Operation

C.457.6. Exceptions

This instruction does not generate synchronous exceptions.

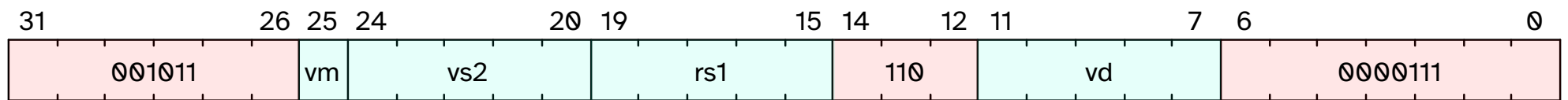
C.458. vloxseg2ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.458.1. Encoding



C.458.2. Description

No description available.

C.458.3. Access

M	S	U
Always	Always	Always

C.458.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.458.5. IDL Operation

C.458.6. Exceptions

This instruction does not generate synchronous exceptions.

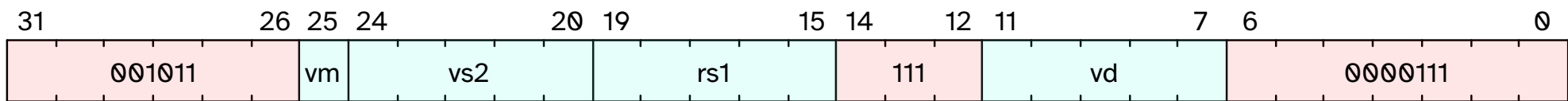
C.459. vloxseg2ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.459.1. Encoding



C.459.2. Description

No description available.

C.459.3. Access

M	S	U
Always	Always	Always

C.459.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.459.5. IDL Operation

C.459.6. Exceptions

This instruction does not generate synchronous exceptions.

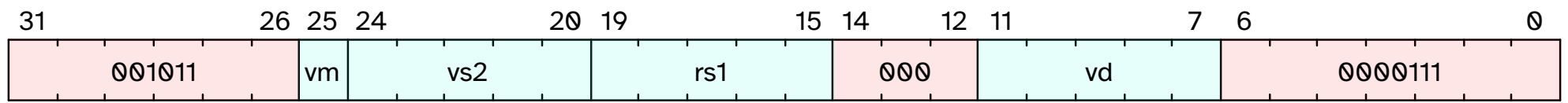
C.460. vloxseg2ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.460.1. Encoding



C.460.2. Description

No description available.

C.460.3. Access

M	S	U
Always	Always	Always

C.460.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.460.5. IDL Operation

C.460.6. Exceptions

This instruction does not generate synchronous exceptions.

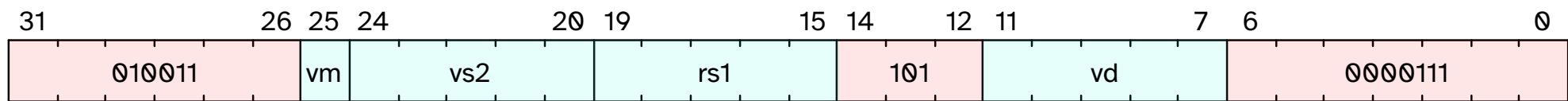
C.461. vloxseg3ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.461.1. Encoding



C.461.2. Description

No description available.

C.461.3. Access

M	S	U
Always	Always	Always

C.461.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.461.5. IDL Operation

C.461.6. Exceptions

This instruction does not generate synchronous exceptions.

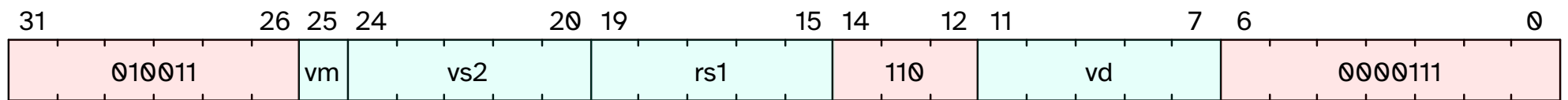
C.462. vloxseg3ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.462.1. Encoding



C.462.2. Description

No description available.

C.462.3. Access

M	S	U
Always	Always	Always

C.462.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.462.5. IDL Operation

C.462.6. Exceptions

This instruction does not generate synchronous exceptions.

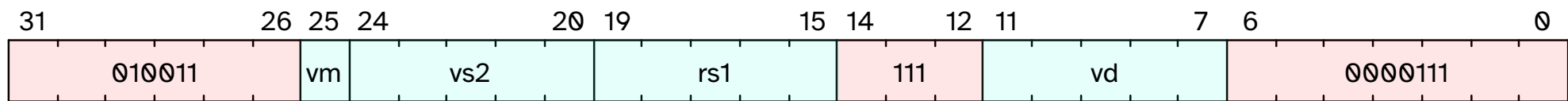
C.463. vloxseg3ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.463.1. Encoding



C.463.2. Description

No description available.

C.463.3. Access

M	S	U
Always	Always	Always

C.463.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.463.5. IDL Operation

C.463.6. Exceptions

This instruction does not generate synchronous exceptions.

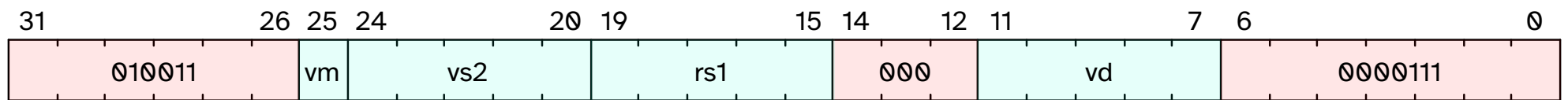
C.464. vloxseg3ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.464.1. Encoding



C.464.2. Description

No description available.

C.464.3. Access

M	S	U
Always	Always	Always

C.464.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.464.5. IDL Operation

C.464.6. Exceptions

This instruction does not generate synchronous exceptions.

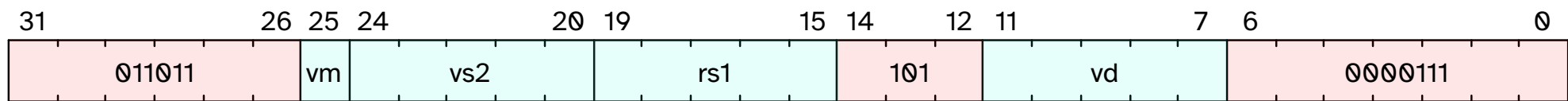
C.465. vloxseg4ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.465.1. Encoding



C.465.2. Description

No description available.

C.465.3. Access

M	S	U
Always	Always	Always

C.465.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.465.5. IDL Operation

C.465.6. Exceptions

This instruction does not generate synchronous exceptions.

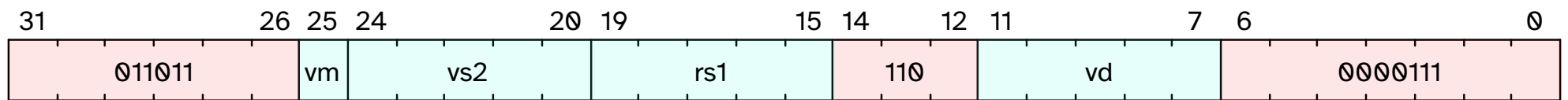
C.466. vloxseg4ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.466.1. Encoding



C.466.2. Description

No description available.

C.466.3. Access

M	S	U
Always	Always	Always

C.466.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.466.5. IDL Operation

C.466.6. Exceptions

This instruction does not generate synchronous exceptions.

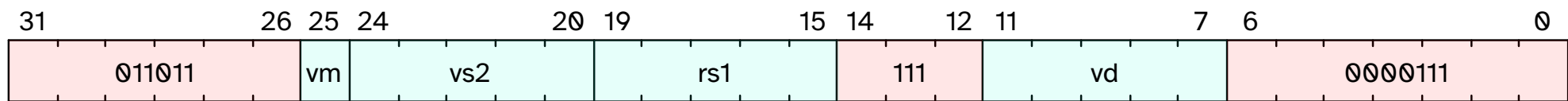
C.467. vloxseg4ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.467.1. Encoding



C.467.2. Description

No description available.

C.467.3. Access

M	S	U
Always	Always	Always

C.467.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.467.5. IDL Operation

C.467.6. Exceptions

This instruction does not generate synchronous exceptions.

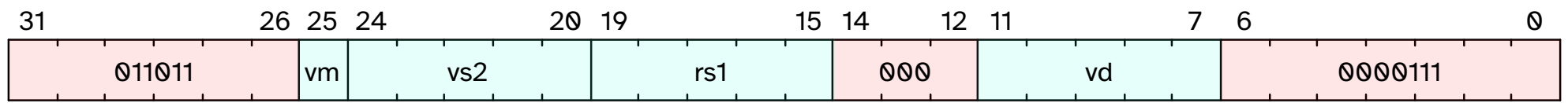
C.468. vloxseg4ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.468.1. Encoding



C.468.2. Description

No description available.

C.468.3. Access

M	S	U
Always	Always	Always

C.468.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.468.5. IDL Operation

C.468.6. Exceptions

This instruction does not generate synchronous exceptions.

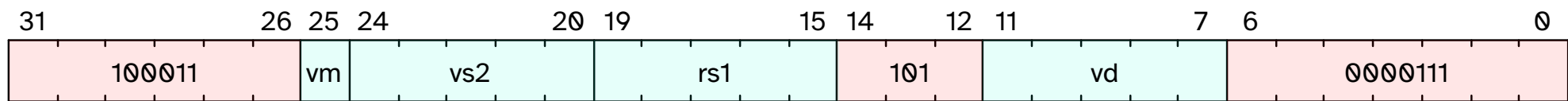
C.469. vloxseg5ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.469.1. Encoding



C.469.2. Description

No description available.

C.469.3. Access

M	S	U
Always	Always	Always

C.469.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.469.5. IDL Operation

C.469.6. Exceptions

This instruction does not generate synchronous exceptions.

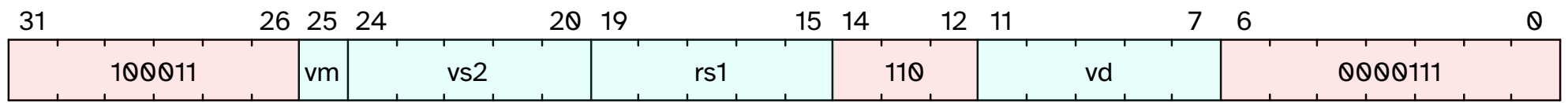
C.470. vloxseg5ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.470.1. Encoding



C.470.2. Description

No description available.

C.470.3. Access

M	S	U
Always	Always	Always

C.470.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.470.5. IDL Operation

C.470.6. Exceptions

This instruction does not generate synchronous exceptions.

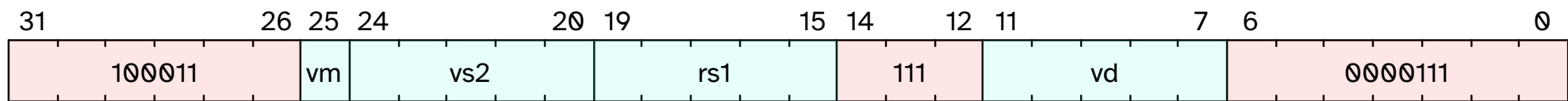
C.471. vloxseg5ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.471.1. Encoding



C.471.2. Description

No description available.

C.471.3. Access

M	S	U
Always	Always	Always

C.471.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.471.5. IDL Operation

C.471.6. Exceptions

This instruction does not generate synchronous exceptions.

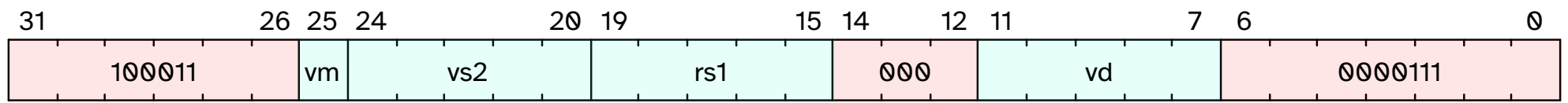
C.472. vloxseg5ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.472.1. Encoding



C.472.2. Description

No description available.

C.472.3. Access

M	S	U
Always	Always	Always

C.472.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.472.5. IDL Operation

C.472.6. Exceptions

This instruction does not generate synchronous exceptions.

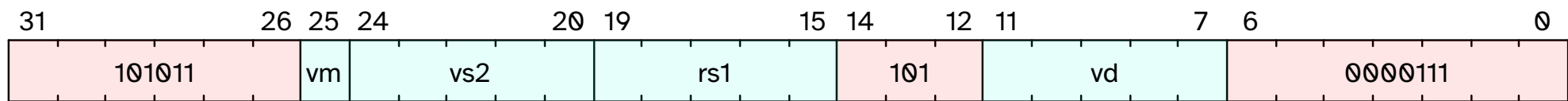
C.473. vloxseg6ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.473.1. Encoding



C.473.2. Description

No description available.

C.473.3. Access

M	S	U
Always	Always	Always

C.473.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.473.5. IDL Operation

C.473.6. Exceptions

This instruction does not generate synchronous exceptions.

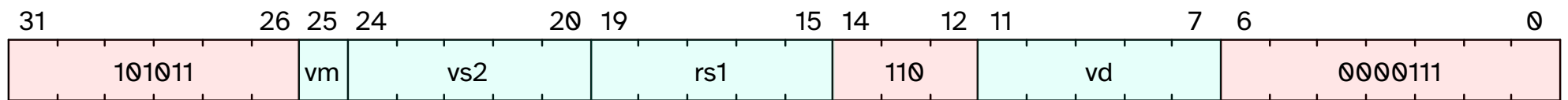
C.474. vloxseg6ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.474.1. Encoding



C.474.2. Description

No description available.

C.474.3. Access

M	S	U
Always	Always	Always

C.474.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.474.5. IDL Operation

C.474.6. Exceptions

This instruction does not generate synchronous exceptions.

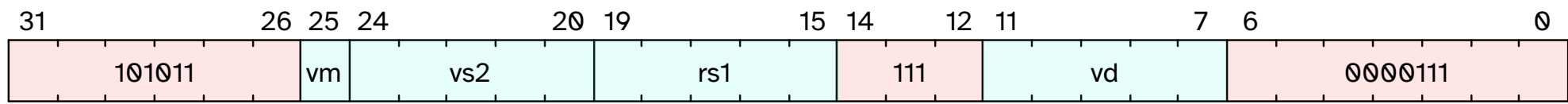
C.475. vloxseg6ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.475.1. Encoding



C.475.2. Description

No description available.

C.475.3. Access

M	S	U
Always	Always	Always

C.475.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.475.5. IDL Operation

C.475.6. Exceptions

This instruction does not generate synchronous exceptions.

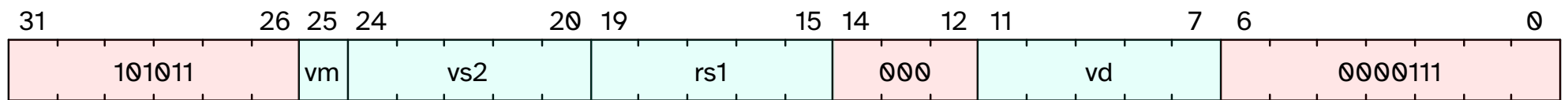
C.476. vloxseg6ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.476.1. Encoding



C.476.2. Description

No description available.

C.476.3. Access

M	S	U
Always	Always	Always

C.476.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.476.5. IDL Operation

C.476.6. Exceptions

This instruction does not generate synchronous exceptions.

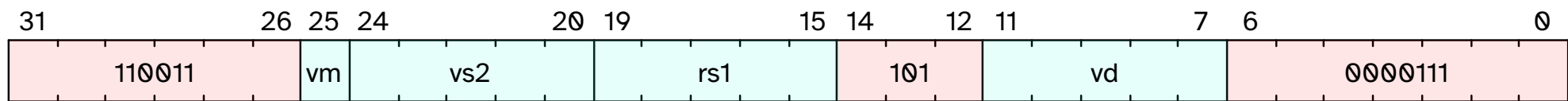
C.477. vloxseg7ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.477.1. Encoding



C.477.2. Description

No description available.

C.477.3. Access

M	S	U
Always	Always	Always

C.477.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.477.5. IDL Operation

C.477.6. Exceptions

This instruction does not generate synchronous exceptions.

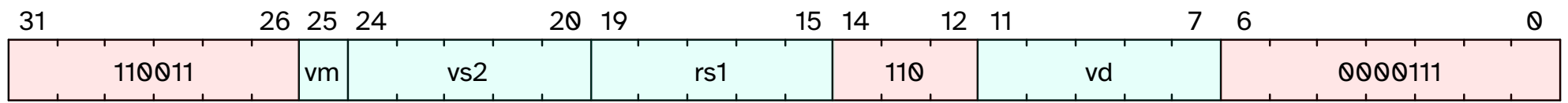
C.478. vloxseg7ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.478.1. Encoding



C.478.2. Description

No description available.

C.478.3. Access

M	S	U
Always	Always	Always

C.478.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.478.5. IDL Operation

C.478.6. Exceptions

This instruction does not generate synchronous exceptions.

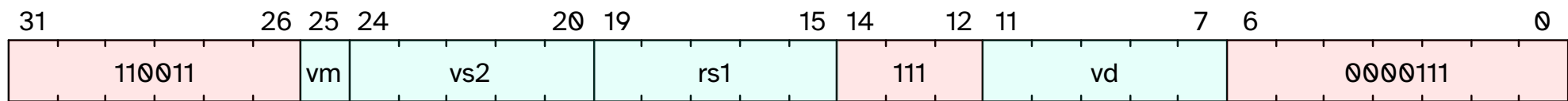
C.479. vloxseg7ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.479.1. Encoding



C.479.2. Description

No description available.

C.479.3. Access

M	S	U
Always	Always	Always

C.479.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.479.5. IDL Operation

C.479.6. Exceptions

This instruction does not generate synchronous exceptions.

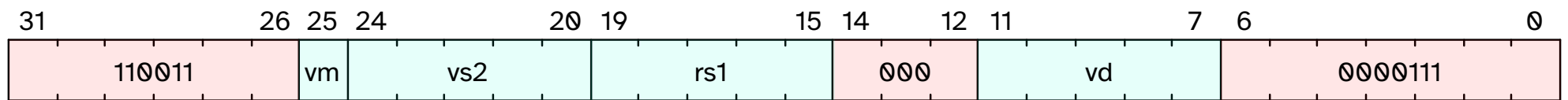
C.480. vloxseg7ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.480.1. Encoding



C.480.2. Description

No description available.

C.480.3. Access

M	S	U
Always	Always	Always

C.480.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.480.5. IDL Operation

C.480.6. Exceptions

This instruction does not generate synchronous exceptions.

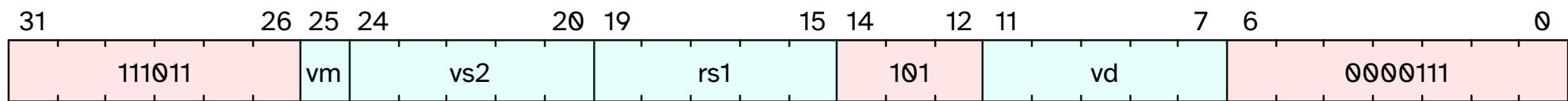
C.481. vloxseg8ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.481.1. Encoding



C.481.2. Description

No description available.

C.481.3. Access

M	S	U
Always	Always	Always

C.481.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.481.5. IDL Operation

C.481.6. Exceptions

This instruction does not generate synchronous exceptions.

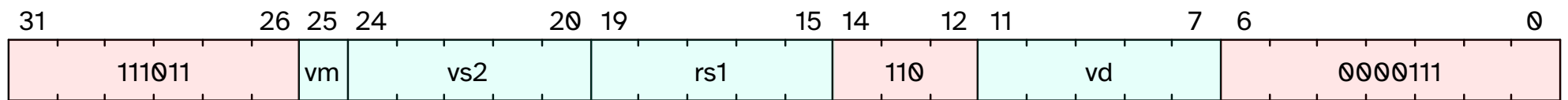
C.482. vloxseg8ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.482.1. Encoding



C.482.2. Description

No description available.

C.482.3. Access

M	S	U
Always	Always	Always

C.482.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.482.5. IDL Operation

C.482.6. Exceptions

This instruction does not generate synchronous exceptions.

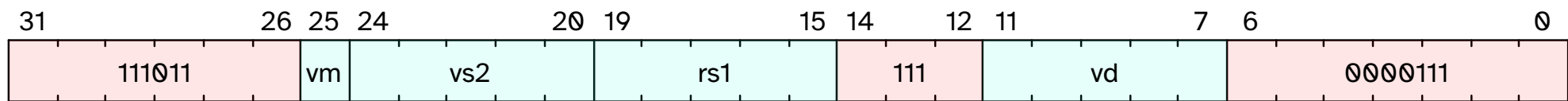
C.483. vloxseg8ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.483.1. Encoding



C.483.2. Description

No description available.

C.483.3. Access

M	S	U
Always	Always	Always

C.483.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.483.5. IDL Operation

C.483.6. Exceptions

This instruction does not generate synchronous exceptions.

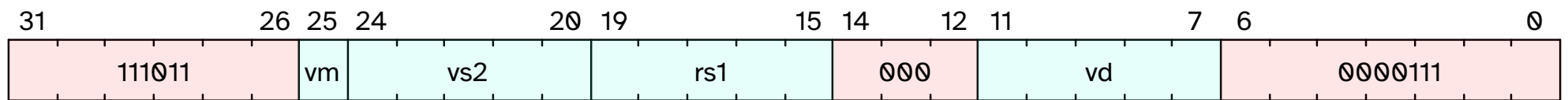
C.484. vloxseg8ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.484.1. Encoding



C.484.2. Description

No description available.

C.484.3. Access

M	S	U
Always	Always	Always

C.484.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.484.5. IDL Operation

C.484.6. Exceptions

This instruction does not generate synchronous exceptions.

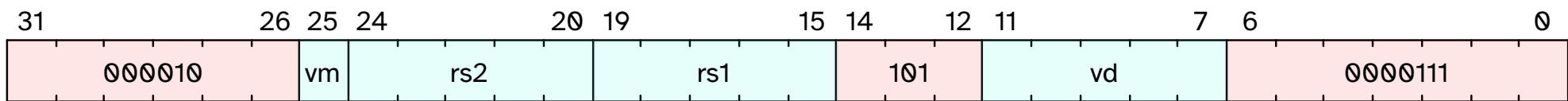
C.485. vlse16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.485.1. Encoding



C.485.2. Description

No description available.

C.485.3. Access

M	S	U
Always	Always	Always

C.485.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.485.5. IDL Operation

C.485.6. Sail Operation

```
{
  let load_width_bytes = vewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmul_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsseg(nf_int, vm, vd, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.485.7. Exceptions

This instruction does not generate synchronous exceptions.

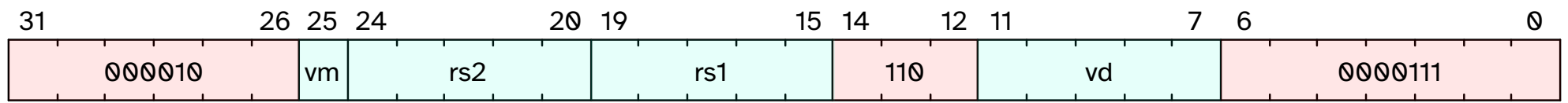
C.486. vlse32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.486.1. Encoding



C.486.2. Description

No description available.

C.486.3. Access

M	S	U
Always	Always	Always

C.486.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.486.5. IDL Operation

C.486.6. Sail Operation

```
{
  let load_width_bytes = vewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmulo_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsseg(nf_int, vm, vd, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.486.7. Exceptions

This instruction does not generate synchronous exceptions.

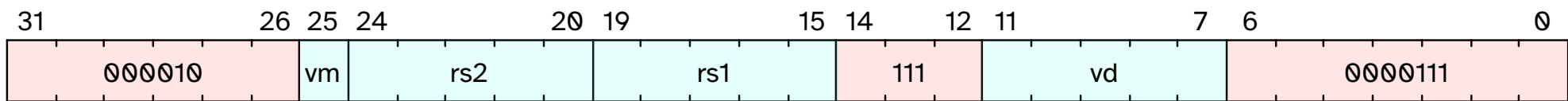
C.487. vlse64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.487.1. Encoding



C.487.2. Description

No description available.

C.487.3. Access

M	S	U
Always	Always	Always

C.487.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.487.5. IDL Operation

C.487.6. Sail Operation

```
{
  let load_width_bytes = vewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmul_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsseg(nf_int, vm, vd, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.487.7. Exceptions

This instruction does not generate synchronous exceptions.

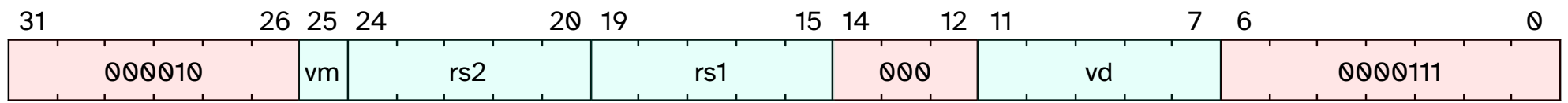
C.488. vlse8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.488.1. Encoding



C.488.2. Description

No description available.

C.488.3. Access

M	S	U
Always	Always	Always

C.488.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.488.5. IDL Operation

C.488.6. Sail Operation

```
{
  let load_width_bytes = vewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_load(vd, vm, nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vlsseg(nf_int, vm, vd, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.488.7. Exceptions

This instruction does not generate synchronous exceptions.

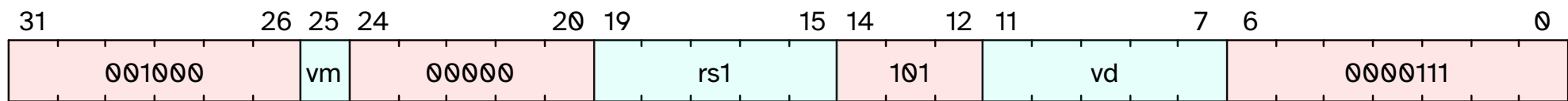
C.489. vlseg2e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.489.1. Encoding



C.489.2. Description

No description available.

C.489.3. Access

M	S	U
Always	Always	Always

C.489.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.489.5. IDL Operation

C.489.6. Exceptions

This instruction does not generate synchronous exceptions.

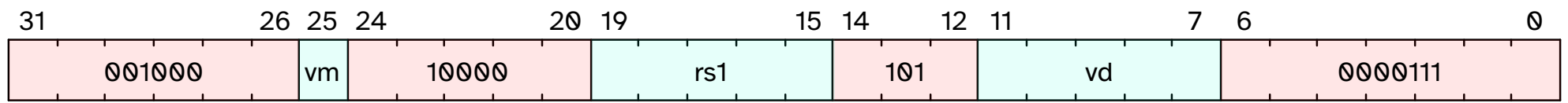
C.490. vlseg2e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.490.1. Encoding



C.490.2. Description

No description available.

C.490.3. Access

M	S	U
Always	Always	Always

C.490.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.490.5. IDL Operation

C.490.6. Exceptions

This instruction does not generate synchronous exceptions.

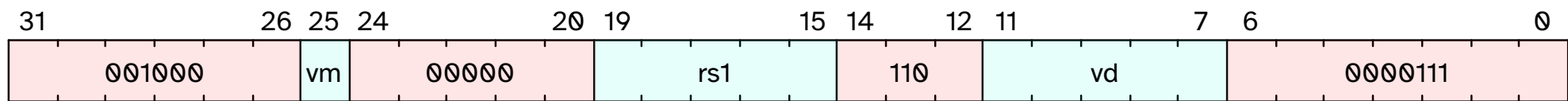
C.491. vlseg2e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.491.1. Encoding



C.491.2. Description

No description available.

C.491.3. Access

M	S	U
Always	Always	Always

C.491.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.491.5. IDL Operation

C.491.6. Exceptions

This instruction does not generate synchronous exceptions.

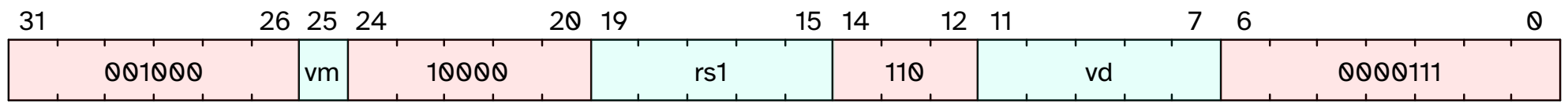
C.492. vlseg2e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.492.1. Encoding



C.492.2. Description

No description available.

C.492.3. Access

M	S	U
Always	Always	Always

C.492.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.492.5. IDL Operation

C.492.6. Exceptions

This instruction does not generate synchronous exceptions.

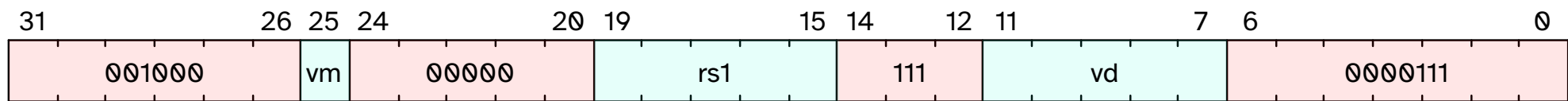
C.493. vlseg2e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.493.1. Encoding



C.493.2. Description

No description available.

C.493.3. Access

M	S	U
Always	Always	Always

C.493.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.493.5. IDL Operation

C.493.6. Exceptions

This instruction does not generate synchronous exceptions.

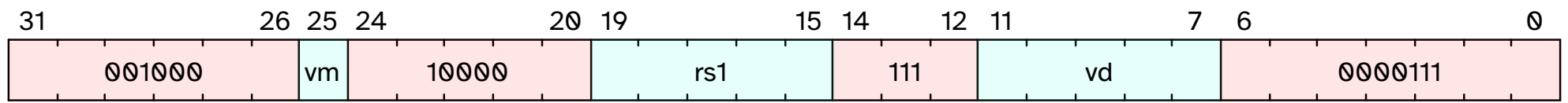
C.494. vlseg2e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.494.1. Encoding



C.494.2. Description

No description available.

C.494.3. Access

M	S	U
Always	Always	Always

C.494.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.494.5. IDL Operation

C.494.6. Exceptions

This instruction does not generate synchronous exceptions.

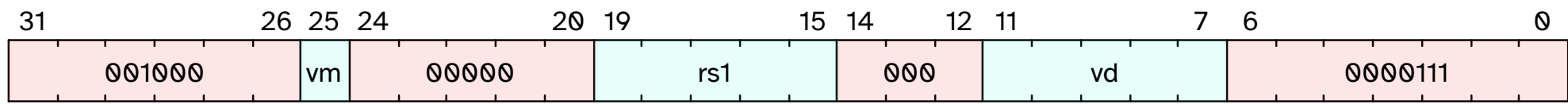
C.495. vlseg2e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.495.1. Encoding



C.495.2. Description

No description available.

C.495.3. Access

M	S	U
Always	Always	Always

C.495.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.495.5. IDL Operation

C.495.6. Exceptions

This instruction does not generate synchronous exceptions.

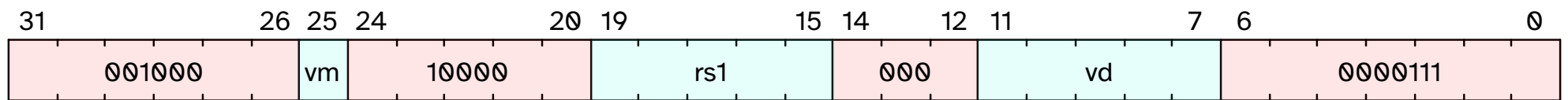
C.496. vlseg2e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.496.1. Encoding



C.496.2. Description

No description available.

C.496.3. Access

M	S	U
Always	Always	Always

C.496.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.496.5. IDL Operation

C.496.6. Exceptions

This instruction does not generate synchronous exceptions.

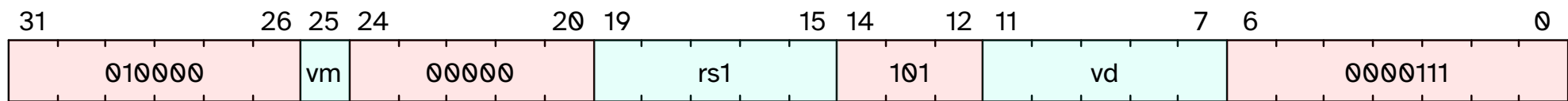
C.497. vlseg3e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.497.1. Encoding



C.497.2. Description

No description available.

C.497.3. Access

M	S	U
Always	Always	Always

C.497.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.497.5. IDL Operation

C.497.6. Exceptions

This instruction does not generate synchronous exceptions.

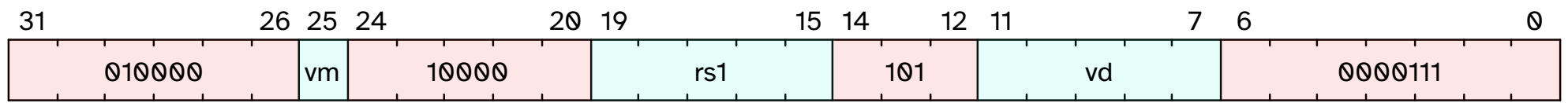
C.498. vlseg3e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.498.1. Encoding



C.498.2. Description

No description available.

C.498.3. Access

M	S	U
Always	Always	Always

C.498.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.498.5. IDL Operation

C.498.6. Exceptions

This instruction does not generate synchronous exceptions.

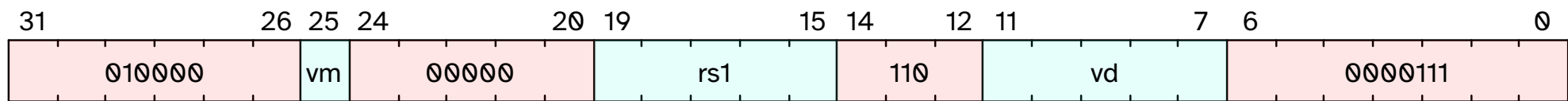
C.499. vlseg3e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.499.1. Encoding



C.499.2. Description

No description available.

C.499.3. Access

M	S	U
Always	Always	Always

C.499.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.499.5. IDL Operation

C.499.6. Exceptions

This instruction does not generate synchronous exceptions.

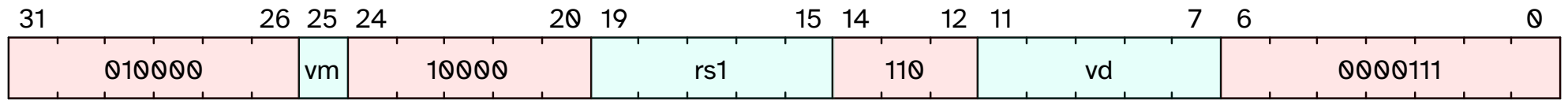
C.500. vlseg3e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.500.1. Encoding



C.500.2. Description

No description available.

C.500.3. Access

M	S	U
Always	Always	Always

C.500.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.500.5. IDL Operation

C.500.6. Exceptions

This instruction does not generate synchronous exceptions.

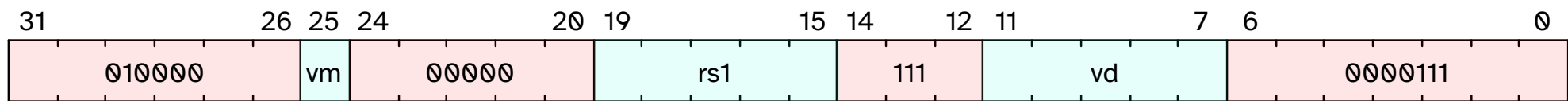
C.501. vlseg3e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.501.1. Encoding



C.501.2. Description

No description available.

C.501.3. Access

M	S	U
Always	Always	Always

C.501.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.501.5. IDL Operation

C.501.6. Exceptions

This instruction does not generate synchronous exceptions.

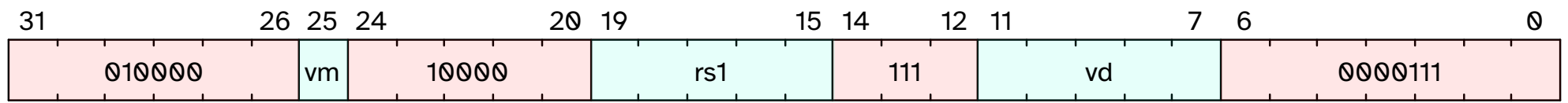
C.502. vlseg3e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.502.1. Encoding



C.502.2. Description

No description available.

C.502.3. Access

M	S	U
Always	Always	Always

C.502.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.502.5. IDL Operation

C.502.6. Exceptions

This instruction does not generate synchronous exceptions.

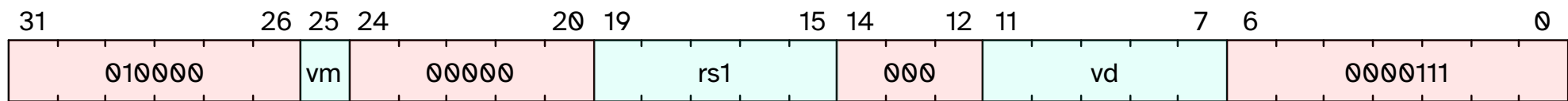
C.503. vlseg3e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.503.1. Encoding



C.503.2. Description

No description available.

C.503.3. Access

M	S	U
Always	Always	Always

C.503.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.503.5. IDL Operation

C.503.6. Exceptions

This instruction does not generate synchronous exceptions.

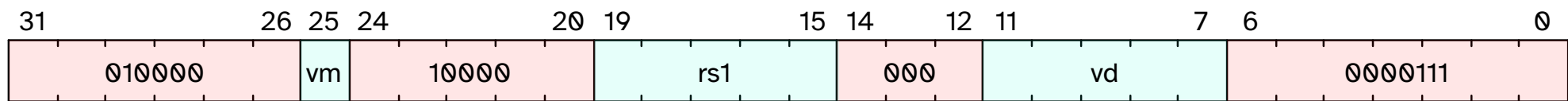
C.504. vlseg3e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.504.1. Encoding



C.504.2. Description

No description available.

C.504.3. Access

M	S	U
Always	Always	Always

C.504.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.504.5. IDL Operation

C.504.6. Exceptions

This instruction does not generate synchronous exceptions.

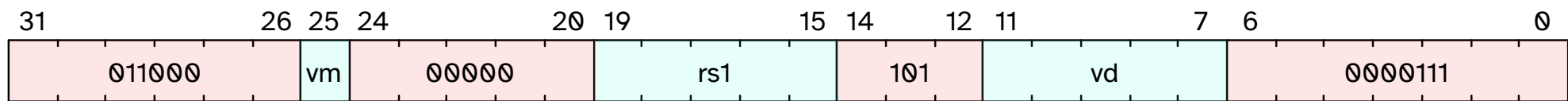
C.505. vlseg4e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.505.1. Encoding



C.505.2. Description

No description available.

C.505.3. Access

M	S	U
Always	Always	Always

C.505.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.505.5. IDL Operation

C.505.6. Exceptions

This instruction does not generate synchronous exceptions.

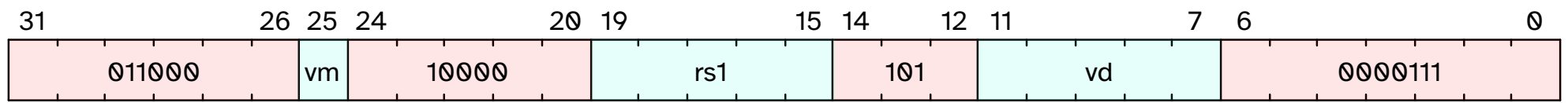
C.506. vlseg4e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.506.1. Encoding



C.506.2. Description

No description available.

C.506.3. Access

M	S	U
Always	Always	Always

C.506.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.506.5. IDL Operation

C.506.6. Exceptions

This instruction does not generate synchronous exceptions.

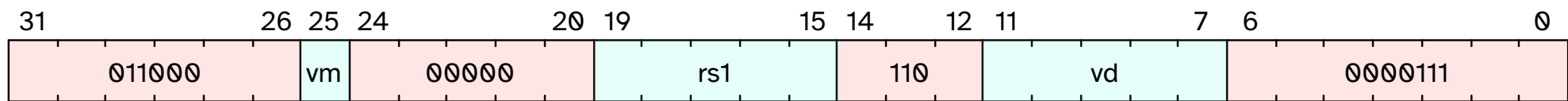
C.507. vlseg4e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.507.1. Encoding



C.507.2. Description

No description available.

C.507.3. Access

M	S	U
Always	Always	Always

C.507.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.507.5. IDL Operation

C.507.6. Exceptions

This instruction does not generate synchronous exceptions.

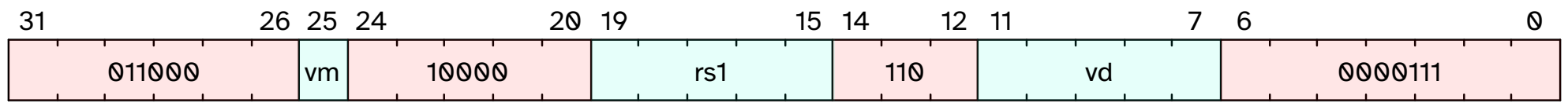
C.508. vlseg4e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.508.1. Encoding



C.508.2. Description

No description available.

C.508.3. Access

M	S	U
Always	Always	Always

C.508.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.508.5. IDL Operation

C.508.6. Exceptions

This instruction does not generate synchronous exceptions.

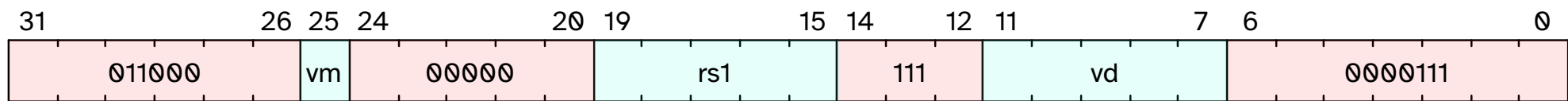
C.509. vlseg4e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.509.1. Encoding



C.509.2. Description

No description available.

C.509.3. Access

M	S	U
Always	Always	Always

C.509.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.509.5. IDL Operation

C.509.6. Exceptions

This instruction does not generate synchronous exceptions.

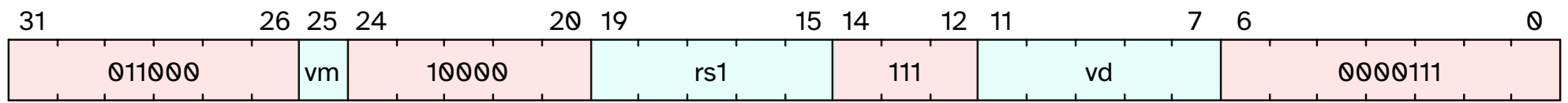
C.510. vlseg4e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.510.1. Encoding



C.510.2. Description

No description available.

C.510.3. Access

M	S	U
Always	Always	Always

C.510.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.510.5. IDL Operation

C.510.6. Exceptions

This instruction does not generate synchronous exceptions.

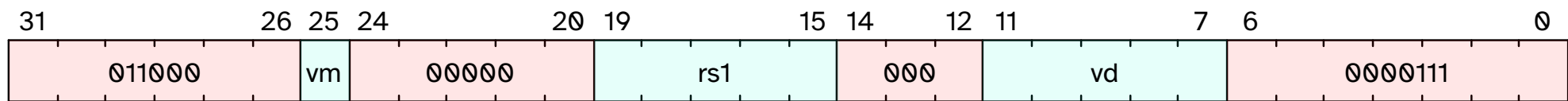
C.511. vlseg4e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.511.1. Encoding



C.511.2. Description

No description available.

C.511.3. Access

M	S	U
Always	Always	Always

C.511.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.511.5. IDL Operation

C.511.6. Exceptions

This instruction does not generate synchronous exceptions.

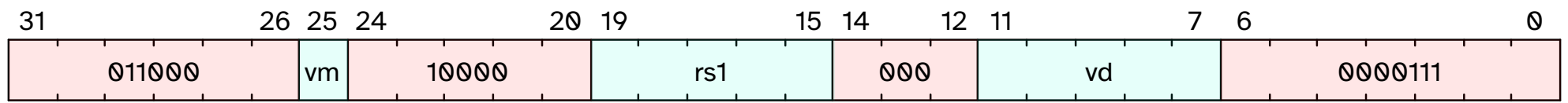
C.512. vlseg4e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.512.1. Encoding



C.512.2. Description

No description available.

C.512.3. Access

M	S	U
Always	Always	Always

C.512.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.512.5. IDL Operation

C.512.6. Exceptions

This instruction does not generate synchronous exceptions.

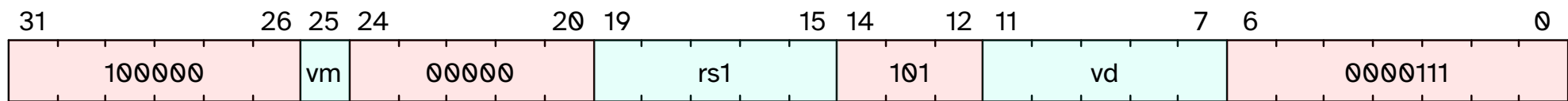
C.513. vlseg5e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.513.1. Encoding



C.513.2. Description

No description available.

C.513.3. Access

M	S	U
Always	Always	Always

C.513.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.513.5. IDL Operation

C.513.6. Exceptions

This instruction does not generate synchronous exceptions.

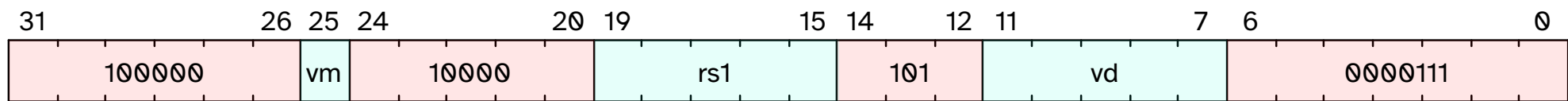
C.514. vlseg5e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.514.1. Encoding



C.514.2. Description

No description available.

C.514.3. Access

M	S	U
Always	Always	Always

C.514.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.514.5. IDL Operation

C.514.6. Exceptions

This instruction does not generate synchronous exceptions.

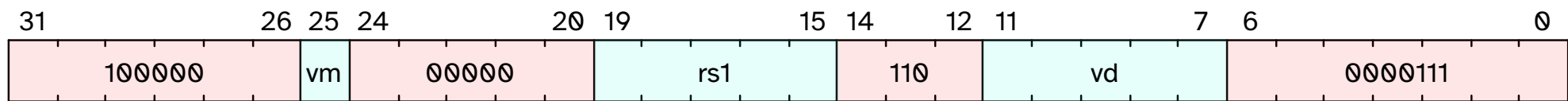
C.515. vlseg5e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.515.1. Encoding



C.515.2. Description

No description available.

C.515.3. Access

M	S	U
Always	Always	Always

C.515.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.515.5. IDL Operation

C.515.6. Exceptions

This instruction does not generate synchronous exceptions.

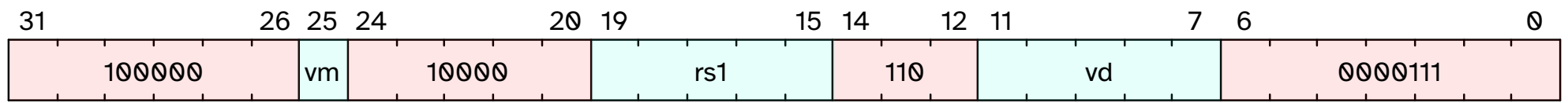
C.516. vlseg5e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.516.1. Encoding



C.516.2. Description

No description available.

C.516.3. Access

M	S	U
Always	Always	Always

C.516.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.516.5. IDL Operation

C.516.6. Exceptions

This instruction does not generate synchronous exceptions.

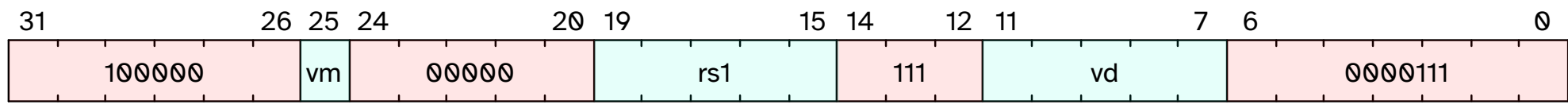
C.517. vlseg5e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.517.1. Encoding



C.517.2. Description

No description available.

C.517.3. Access

M	S	U
Always	Always	Always

C.517.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.517.5. IDL Operation

C.517.6. Exceptions

This instruction does not generate synchronous exceptions.

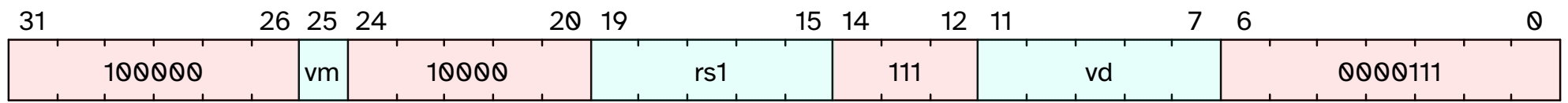
C.518. vlseg5e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.518.1. Encoding



C.518.2. Description

No description available.

C.518.3. Access

M	S	U
Always	Always	Always

C.518.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.518.5. IDL Operation

C.518.6. Exceptions

This instruction does not generate synchronous exceptions.

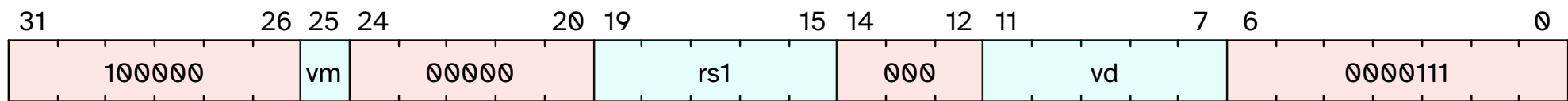
C.519. vlseg5e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.519.1. Encoding



C.519.2. Description

No description available.

C.519.3. Access

M	S	U
Always	Always	Always

C.519.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.519.5. IDL Operation

C.519.6. Exceptions

This instruction does not generate synchronous exceptions.

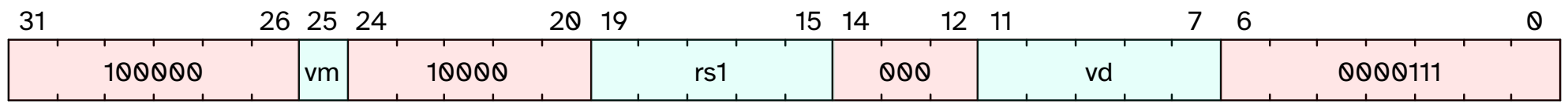
C.520. vlseg5e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.520.1. Encoding



C.520.2. Description

No description available.

C.520.3. Access

M	S	U
Always	Always	Always

C.520.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.520.5. IDL Operation

C.520.6. Exceptions

This instruction does not generate synchronous exceptions.

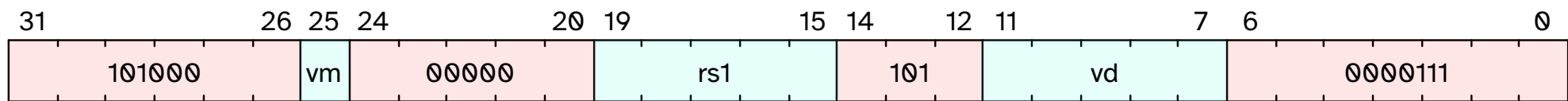
C.521. vlseg6e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.521.1. Encoding



C.521.2. Description

No description available.

C.521.3. Access

M	S	U
Always	Always	Always

C.521.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.521.5. IDL Operation

C.521.6. Exceptions

This instruction does not generate synchronous exceptions.

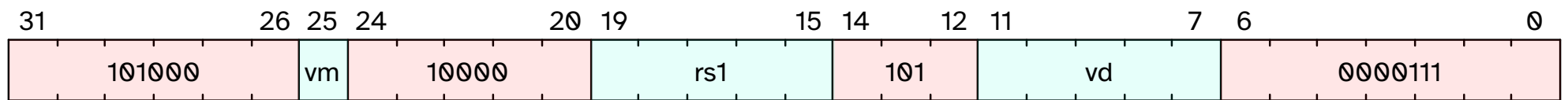
C.522. vlseg6e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.522.1. Encoding



C.522.2. Description

No description available.

C.522.3. Access

M	S	U
Always	Always	Always

C.522.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.522.5. IDL Operation

C.522.6. Exceptions

This instruction does not generate synchronous exceptions.

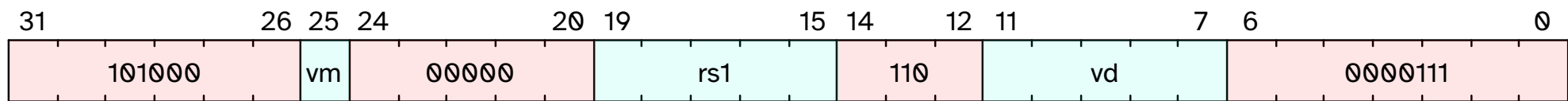
C.523. vlseg6e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.523.1. Encoding



C.523.2. Description

No description available.

C.523.3. Access

M	S	U
Always	Always	Always

C.523.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.523.5. IDL Operation

C.523.6. Exceptions

This instruction does not generate synchronous exceptions.

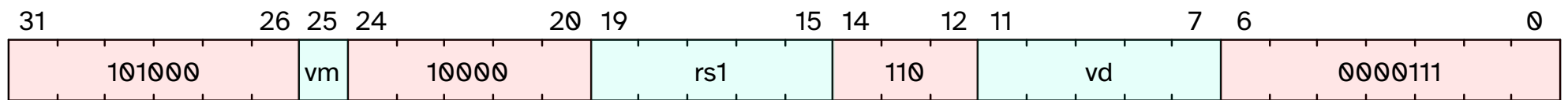
C.524. vlseg6e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.524.1. Encoding



C.524.2. Description

No description available.

C.524.3. Access

M	S	U
Always	Always	Always

C.524.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.524.5. IDL Operation

C.524.6. Exceptions

This instruction does not generate synchronous exceptions.

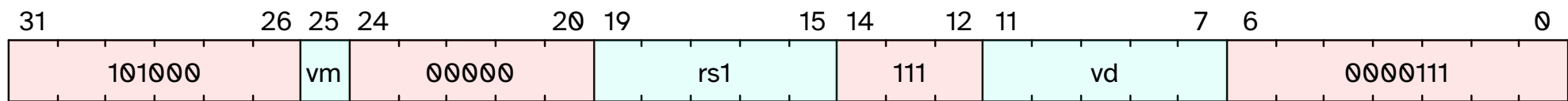
C.525. vlseg6e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.525.1. Encoding



C.525.2. Description

No description available.

C.525.3. Access

M	S	U
Always	Always	Always

C.525.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.525.5. IDL Operation

C.525.6. Exceptions

This instruction does not generate synchronous exceptions.

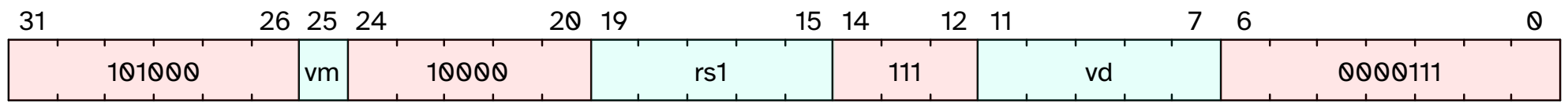
C.526. vlseg6e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.526.1. Encoding



C.526.2. Description

No description available.

C.526.3. Access

M	S	U
Always	Always	Always

C.526.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.526.5. IDL Operation

C.526.6. Exceptions

This instruction does not generate synchronous exceptions.

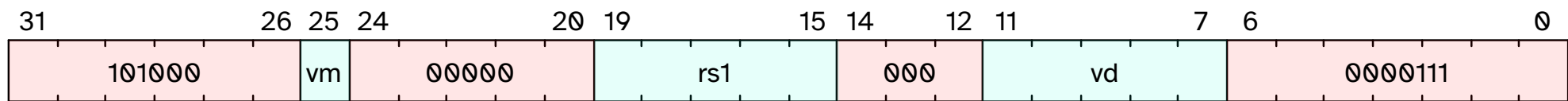
C.527. vlseg6e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.527.1. Encoding



C.527.2. Description

No description available.

C.527.3. Access

M	S	U
Always	Always	Always

C.527.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.527.5. IDL Operation

C.527.6. Exceptions

This instruction does not generate synchronous exceptions.

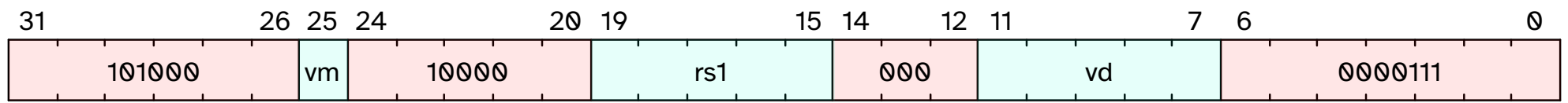
C.528. vlseg6e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.528.1. Encoding



C.528.2. Description

No description available.

C.528.3. Access

M	S	U
Always	Always	Always

C.528.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.528.5. IDL Operation

C.528.6. Exceptions

This instruction does not generate synchronous exceptions.

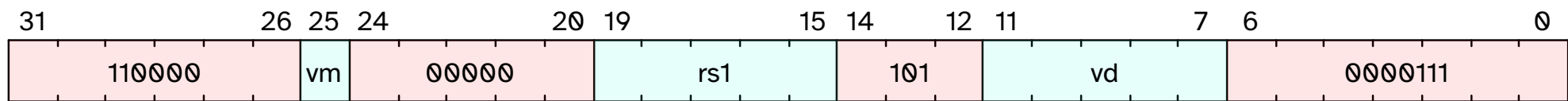
C.529. vlseg7e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.529.1. Encoding



C.529.2. Description

No description available.

C.529.3. Access

M	S	U
Always	Always	Always

C.529.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.529.5. IDL Operation

C.529.6. Exceptions

This instruction does not generate synchronous exceptions.

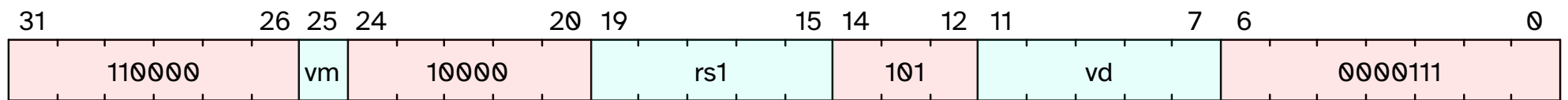
C.530. vlseg7e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.530.1. Encoding



C.530.2. Description

No description available.

C.530.3. Access

M	S	U
Always	Always	Always

C.530.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.530.5. IDL Operation

C.530.6. Exceptions

This instruction does not generate synchronous exceptions.

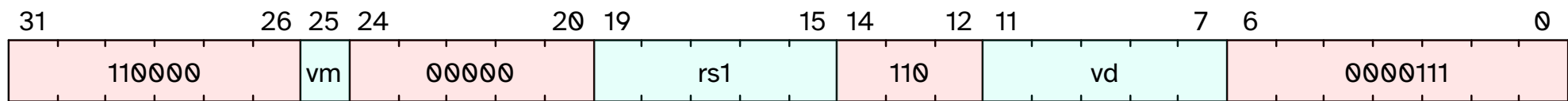
C.531. vlseg7e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.531.1. Encoding



C.531.2. Description

No description available.

C.531.3. Access

M	S	U
Always	Always	Always

C.531.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.531.5. IDL Operation

C.531.6. Exceptions

This instruction does not generate synchronous exceptions.

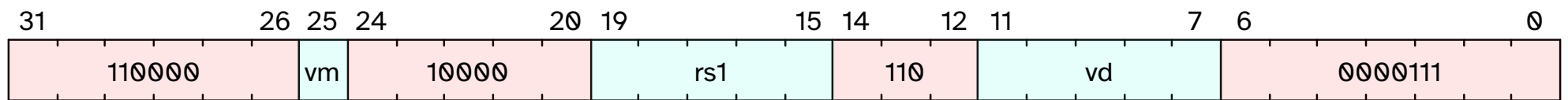
C.532. vlseg7e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.532.1. Encoding



C.532.2. Description

No description available.

C.532.3. Access

M	S	U
Always	Always	Always

C.532.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.532.5. IDL Operation

C.532.6. Exceptions

This instruction does not generate synchronous exceptions.

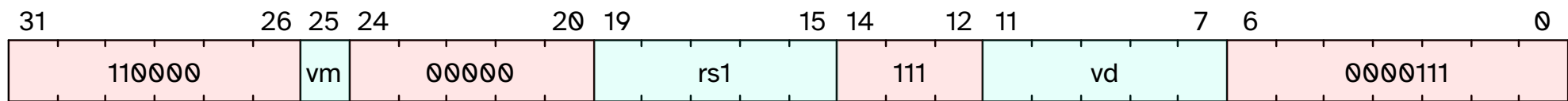
C.533. vlseg7e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.533.1. Encoding



C.533.2. Description

No description available.

C.533.3. Access

M	S	U
Always	Always	Always

C.533.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.533.5. IDL Operation

C.533.6. Exceptions

This instruction does not generate synchronous exceptions.

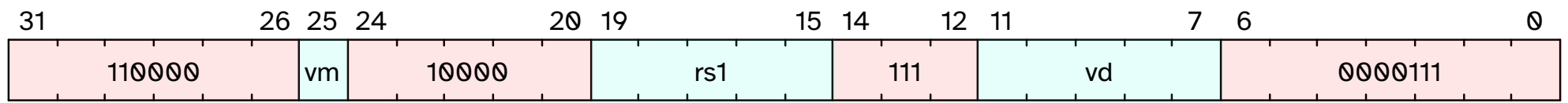
C.534. vlseg7e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.534.1. Encoding



C.534.2. Description

No description available.

C.534.3. Access

M	S	U
Always	Always	Always

C.534.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.534.5. IDL Operation

C.534.6. Exceptions

This instruction does not generate synchronous exceptions.

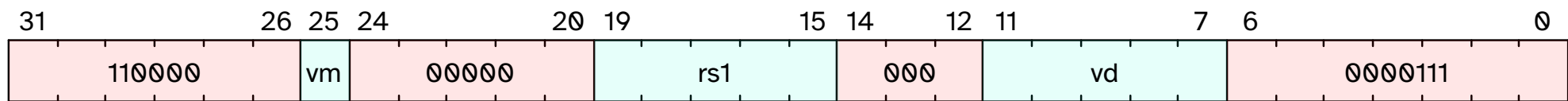
C.535. vlseg7e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.535.1. Encoding



C.535.2. Description

No description available.

C.535.3. Access

M	S	U
Always	Always	Always

C.535.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.535.5. IDL Operation

C.535.6. Exceptions

This instruction does not generate synchronous exceptions.

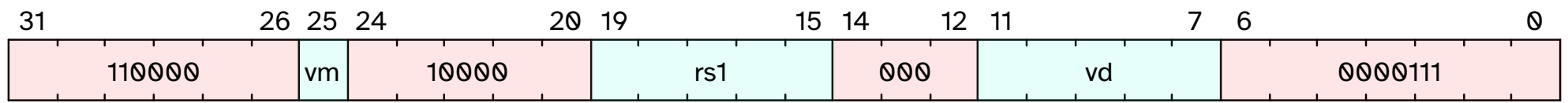
C.536. vlseg7e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.536.1. Encoding



C.536.2. Description

No description available.

C.536.3. Access

M	S	U
Always	Always	Always

C.536.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.536.5. IDL Operation

C.536.6. Exceptions

This instruction does not generate synchronous exceptions.

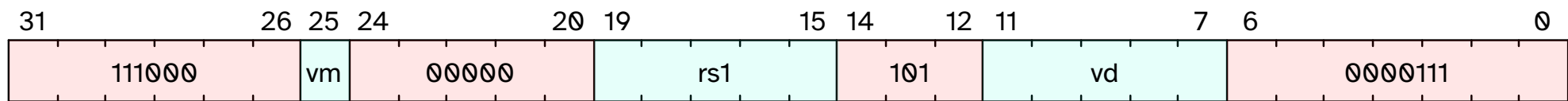
C.537. vlseg8e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.537.1. Encoding



C.537.2. Description

No description available.

C.537.3. Access

M	S	U
Always	Always	Always

C.537.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.537.5. IDL Operation

C.537.6. Exceptions

This instruction does not generate synchronous exceptions.

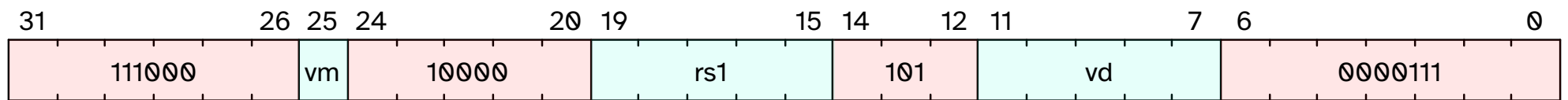
C.538. vlseg8e16ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.538.1. Encoding



C.538.2. Description

No description available.

C.538.3. Access

M	S	U
Always	Always	Always

C.538.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.538.5. IDL Operation

C.538.6. Exceptions

This instruction does not generate synchronous exceptions.

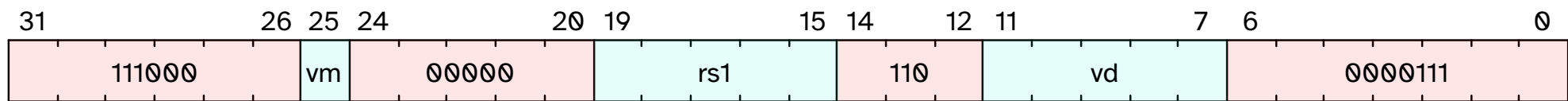
C.539. vlseg8e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.539.1. Encoding



C.539.2. Description

No description available.

C.539.3. Access

M	S	U
Always	Always	Always

C.539.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.539.5. IDL Operation

C.539.6. Exceptions

This instruction does not generate synchronous exceptions.

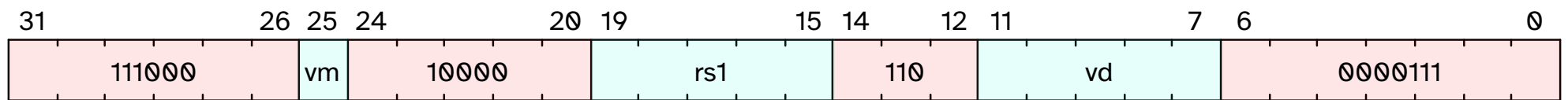
C.540. vlseg8e32ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.540.1. Encoding



C.540.2. Description

No description available.

C.540.3. Access

M	S	U
Always	Always	Always

C.540.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.540.5. IDL Operation

C.540.6. Exceptions

This instruction does not generate synchronous exceptions.

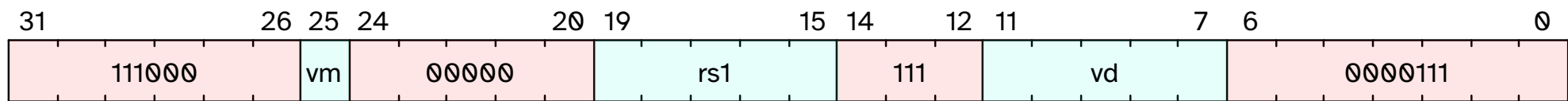
C.541. vlseg8e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.541.1. Encoding



C.541.2. Description

No description available.

C.541.3. Access

M	S	U
Always	Always	Always

C.541.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.541.5. IDL Operation

C.541.6. Exceptions

This instruction does not generate synchronous exceptions.

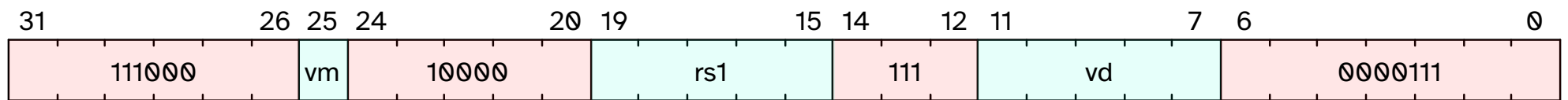
C.542. vlseg8e64ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.542.1. Encoding



C.542.2. Description

No description available.

C.542.3. Access

M	S	U
Always	Always	Always

C.542.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.542.5. IDL Operation

C.542.6. Exceptions

This instruction does not generate synchronous exceptions.

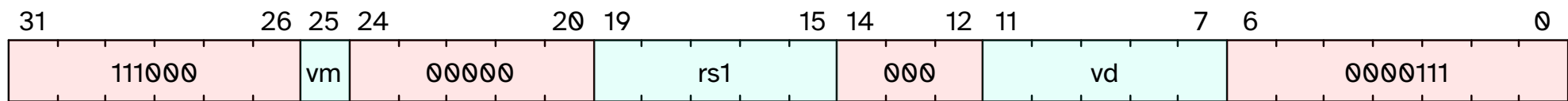
C.543. vlseg8e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.543.1. Encoding



C.543.2. Description

No description available.

C.543.3. Access

M	S	U
Always	Always	Always

C.543.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.543.5. IDL Operation

C.543.6. Exceptions

This instruction does not generate synchronous exceptions.

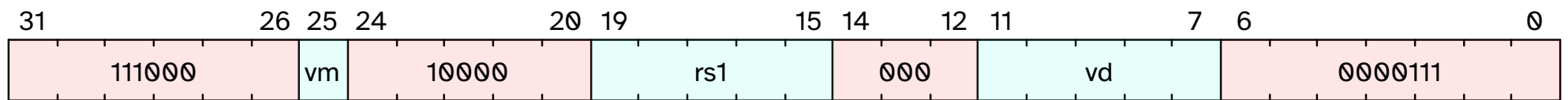
C.544. vlseg8e8ff.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.544.1. Encoding



C.544.2. Description

No description available.

C.544.3. Access

M	S	U
Always	Always	Always

C.544.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.544.5. IDL Operation

C.544.6. Exceptions

This instruction does not generate synchronous exceptions.

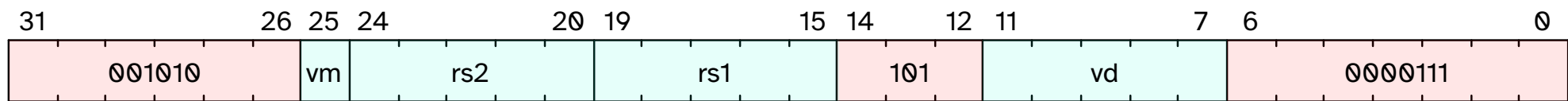
C.545. vlsseg2e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.545.1. Encoding



C.545.2. Description

No description available.

C.545.3. Access

M	S	U
Always	Always	Always

C.545.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.545.5. IDL Operation

C.545.6. Exceptions

This instruction does not generate synchronous exceptions.

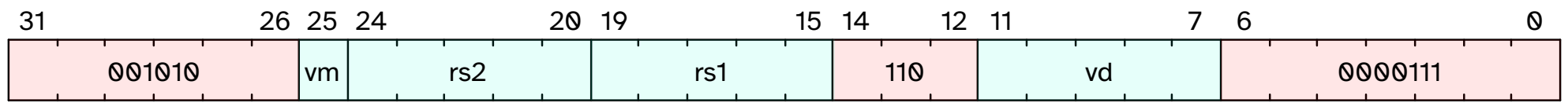
C.546. vlsseg2e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.546.1. Encoding



C.546.2. Description

No description available.

C.546.3. Access

M	S	U
Always	Always	Always

C.546.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.546.5. IDL Operation

C.546.6. Exceptions

This instruction does not generate synchronous exceptions.

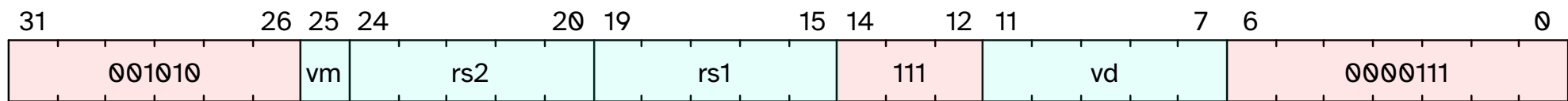
C.547. vlsseg2e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.547.1. Encoding



C.547.2. Description

No description available.

C.547.3. Access

M	S	U
Always	Always	Always

C.547.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.547.5. IDL Operation

C.547.6. Exceptions

This instruction does not generate synchronous exceptions.

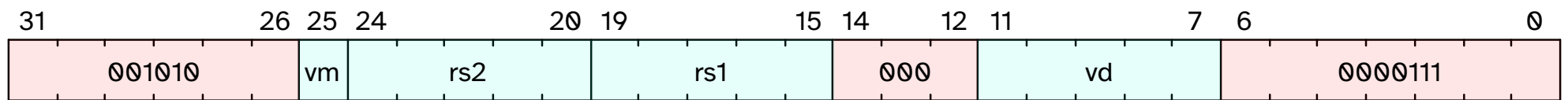
C.548. vlsseg2e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.548.1. Encoding



C.548.2. Description

No description available.

C.548.3. Access

M	S	U
Always	Always	Always

C.548.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.548.5. IDL Operation

C.548.6. Exceptions

This instruction does not generate synchronous exceptions.

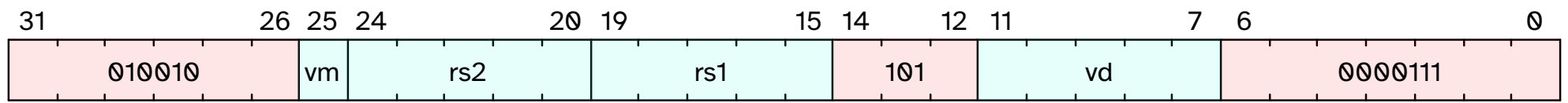
C.549. vlsseg3e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.549.1. Encoding



C.549.2. Description

No description available.

C.549.3. Access

M	S	U
Always	Always	Always

C.549.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.549.5. IDL Operation

C.549.6. Exceptions

This instruction does not generate synchronous exceptions.

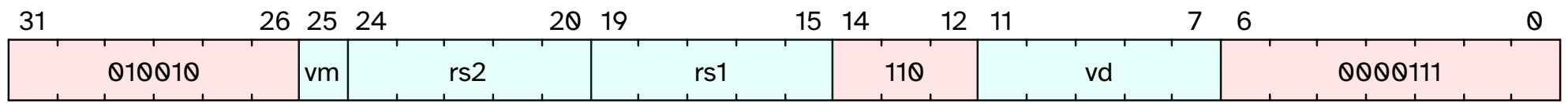
C.550. vlsseg3e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.550.1. Encoding



C.550.2. Description

No description available.

C.550.3. Access

M	S	U
Always	Always	Always

C.550.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.550.5. IDL Operation

C.550.6. Exceptions

This instruction does not generate synchronous exceptions.

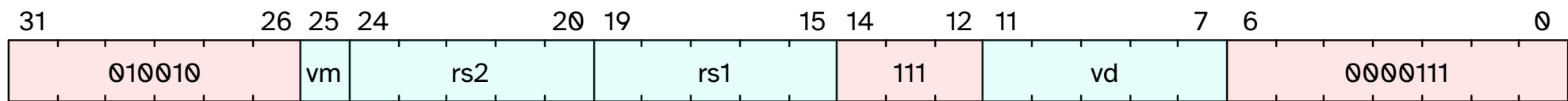
C.551. vlsseg3e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.551.1. Encoding



C.551.2. Description

No description available.

C.551.3. Access

M	S	U
Always	Always	Always

C.551.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.551.5. IDL Operation

C.551.6. Exceptions

This instruction does not generate synchronous exceptions.

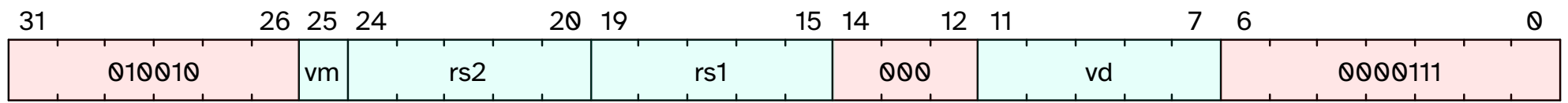
C.552. vlsseg3e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.552.1. Encoding



C.552.2. Description

No description available.

C.552.3. Access

M	S	U
Always	Always	Always

C.552.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.552.5. IDL Operation

C.552.6. Exceptions

This instruction does not generate synchronous exceptions.

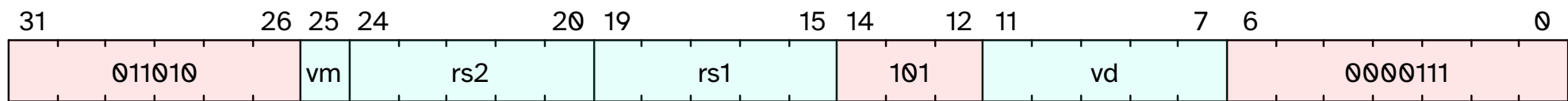
C.553. vlsseg4e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.553.1. Encoding



C.553.2. Description

No description available.

C.553.3. Access

M	S	U
Always	Always	Always

C.553.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.553.5. IDL Operation

C.553.6. Exceptions

This instruction does not generate synchronous exceptions.

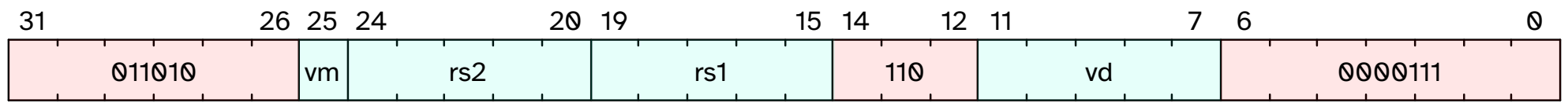
C.554. vlsseg4e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.554.1. Encoding



C.554.2. Description

No description available.

C.554.3. Access

M	S	U
Always	Always	Always

C.554.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.554.5. IDL Operation

C.554.6. Exceptions

This instruction does not generate synchronous exceptions.

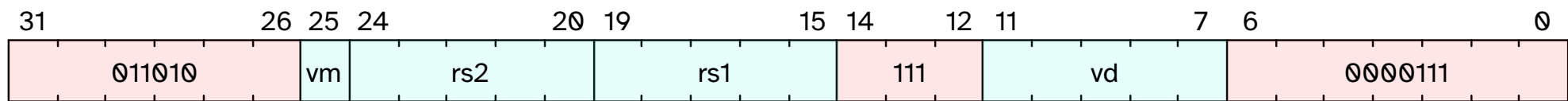
C.555. vlsseg4e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.555.1. Encoding



C.555.2. Description

No description available.

C.555.3. Access

M	S	U
Always	Always	Always

C.555.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.555.5. IDL Operation

C.555.6. Exceptions

This instruction does not generate synchronous exceptions.

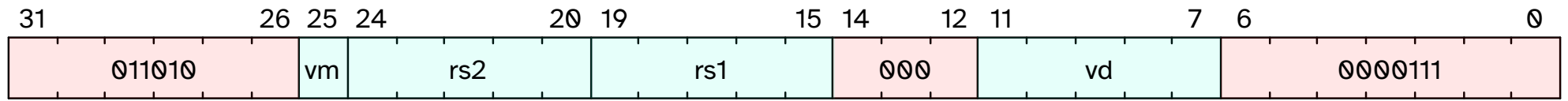
C.556. vlsseg4e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.556.1. Encoding



C.556.2. Description

No description available.

C.556.3. Access

M	S	U
Always	Always	Always

C.556.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.556.5. IDL Operation

C.556.6. Exceptions

This instruction does not generate synchronous exceptions.

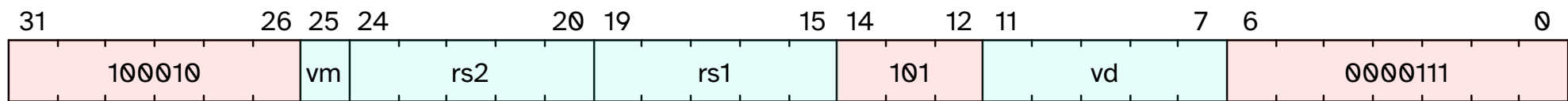
C.557. vlsseg5e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.557.1. Encoding



C.557.2. Description

No description available.

C.557.3. Access

M	S	U
Always	Always	Always

C.557.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.557.5. IDL Operation

C.557.6. Exceptions

This instruction does not generate synchronous exceptions.

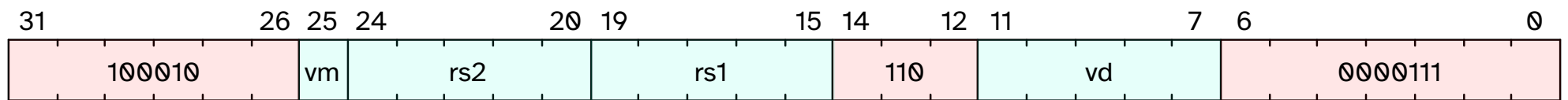
C.558. vlsseg5e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.558.1. Encoding



C.558.2. Description

No description available.

C.558.3. Access

M	S	U
Always	Always	Always

C.558.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.558.5. IDL Operation

C.558.6. Exceptions

This instruction does not generate synchronous exceptions.

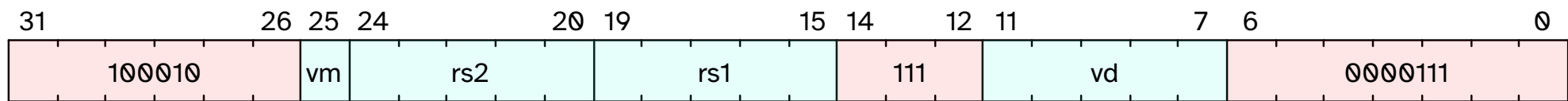
C.559. vlsseg5e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.559.1. Encoding



C.559.2. Description

No description available.

C.559.3. Access

M	S	U
Always	Always	Always

C.559.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.559.5. IDL Operation

C.559.6. Exceptions

This instruction does not generate synchronous exceptions.

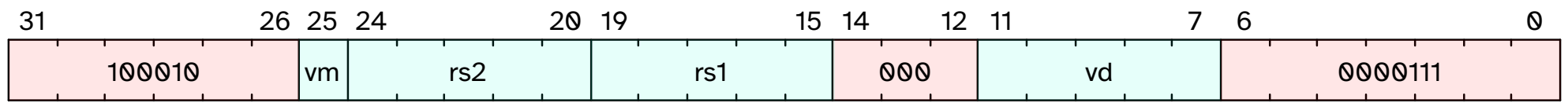
C.560. vlsseg5e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.560.1. Encoding



C.560.2. Description

No description available.

C.560.3. Access

M	S	U
Always	Always	Always

C.560.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.560.5. IDL Operation

C.560.6. Exceptions

This instruction does not generate synchronous exceptions.

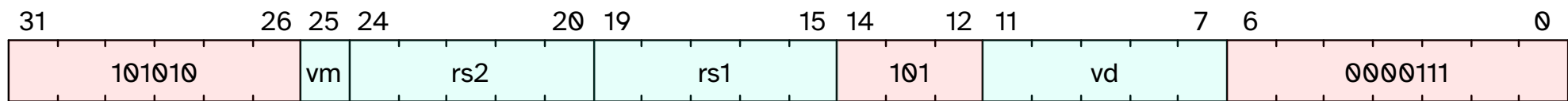
C.561. vlsseg6e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.561.1. Encoding



C.561.2. Description

No description available.

C.561.3. Access

M	S	U
Always	Always	Always

C.561.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.561.5. IDL Operation

C.561.6. Exceptions

This instruction does not generate synchronous exceptions.

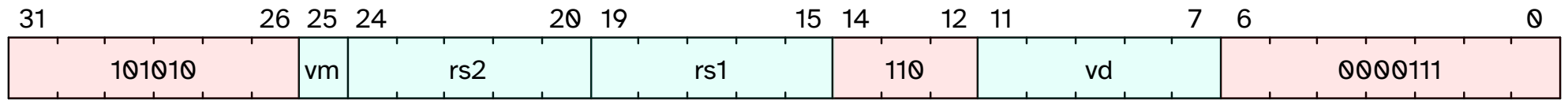
C.562. vlsseg6e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.562.1. Encoding



C.562.2. Description

No description available.

C.562.3. Access

M	S	U
Always	Always	Always

C.562.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.562.5. IDL Operation

C.562.6. Exceptions

This instruction does not generate synchronous exceptions.

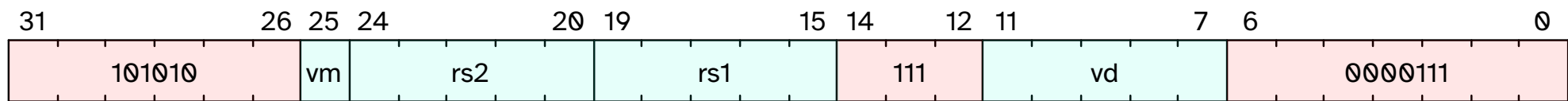
C.563. vlsseg6e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.563.1. Encoding



C.563.2. Description

No description available.

C.563.3. Access

M	S	U
Always	Always	Always

C.563.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.563.5. IDL Operation

C.563.6. Exceptions

This instruction does not generate synchronous exceptions.

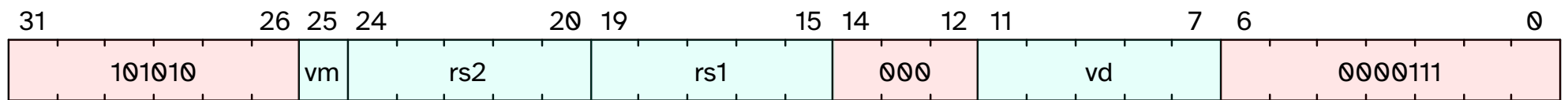
C.564. vlsseg6e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.564.1. Encoding



C.564.2. Description

No description available.

C.564.3. Access

M	S	U
Always	Always	Always

C.564.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.564.5. IDL Operation

C.564.6. Exceptions

This instruction does not generate synchronous exceptions.

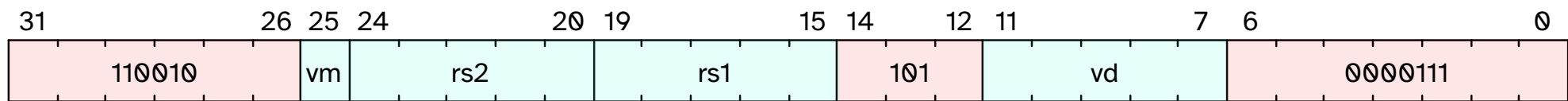
C.565. vlsseg7e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.565.1. Encoding



C.565.2. Description

No description available.

C.565.3. Access

M	S	U
Always	Always	Always

C.565.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.565.5. IDL Operation

C.565.6. Exceptions

This instruction does not generate synchronous exceptions.

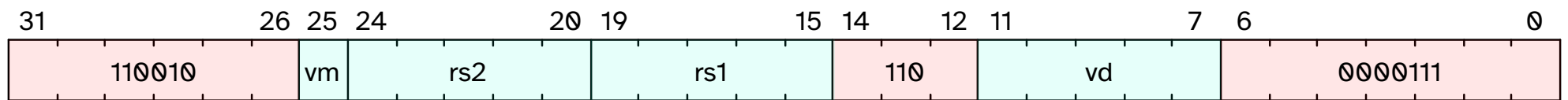
C.566. vlsseg7e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.566.1. Encoding



C.566.2. Description

No description available.

C.566.3. Access

M	S	U
Always	Always	Always

C.566.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.566.5. IDL Operation

C.566.6. Exceptions

This instruction does not generate synchronous exceptions.

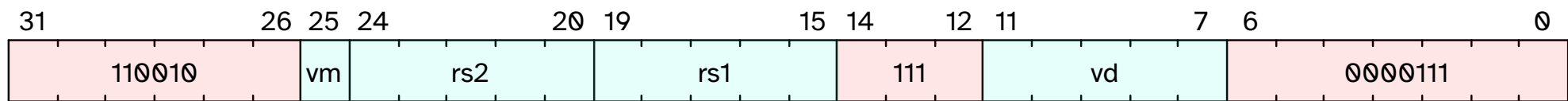
C.567. vlsseg7e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.567.1. Encoding



C.567.2. Description

No description available.

C.567.3. Access

M	S	U
Always	Always	Always

C.567.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.567.5. IDL Operation

C.567.6. Exceptions

This instruction does not generate synchronous exceptions.

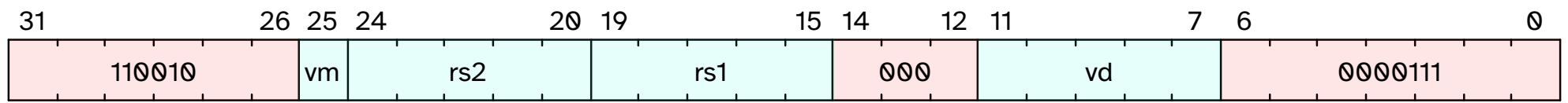
C.568. vlsseg7e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.568.1. Encoding



C.568.2. Description

No description available.

C.568.3. Access

M	S	U
Always	Always	Always

C.568.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.568.5. IDL Operation

C.568.6. Exceptions

This instruction does not generate synchronous exceptions.

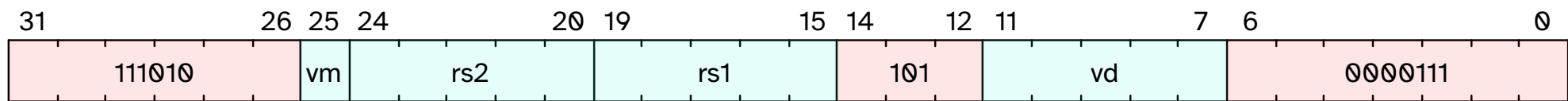
C.569. vlsseg8e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.569.1. Encoding



C.569.2. Description

No description available.

C.569.3. Access

M	S	U
Always	Always	Always

C.569.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.569.5. IDL Operation

C.569.6. Exceptions

This instruction does not generate synchronous exceptions.

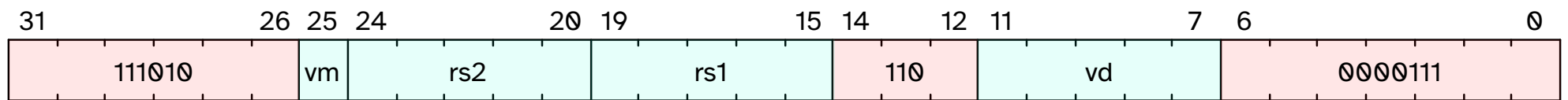
C.570. vlsseg8e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.570.1. Encoding



C.570.2. Description

No description available.

C.570.3. Access

M	S	U
Always	Always	Always

C.570.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.570.5. IDL Operation

C.570.6. Exceptions

This instruction does not generate synchronous exceptions.

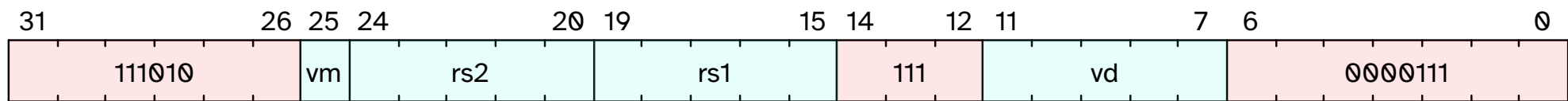
C.571. vlsseg8e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.571.1. Encoding



C.571.2. Description

No description available.

C.571.3. Access

M	S	U
Always	Always	Always

C.571.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.571.5. IDL Operation

C.571.6. Exceptions

This instruction does not generate synchronous exceptions.

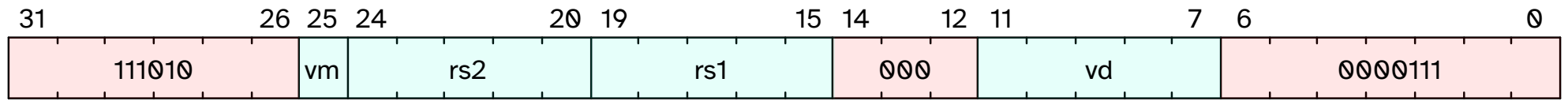
C.572. vlsseg8e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.572.1. Encoding



C.572.2. Description

No description available.

C.572.3. Access

M	S	U
Always	Always	Always

C.572.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.572.5. IDL Operation

C.572.6. Exceptions

This instruction does not generate synchronous exceptions.

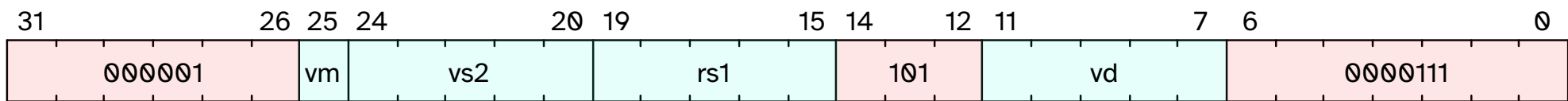
C.573. vluxe16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.573.1. Encoding



C.573.2. Description

No description available.

C.573.3. Access

M	S	U
Always	Always	Always

C.573.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.573.5. IDL Operation

C.573.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.573.7. Exceptions

This instruction does not generate synchronous exceptions.

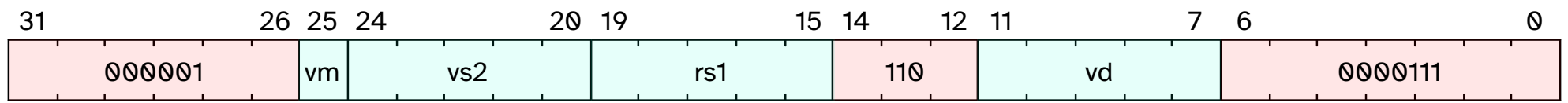
C.574. vluxeï32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.574.1. Encoding



C.574.2. Description

No description available.

C.574.3. Access

M	S	U
Always	Always	Always

C.574.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.574.5. IDL Operation

C.574.6. Sail Operation

```
{
  let EEW_index_pow = vewidth_pow(width);
  let EEW_index_bytes = vewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.574.7. Exceptions

This instruction does not generate synchronous exceptions.

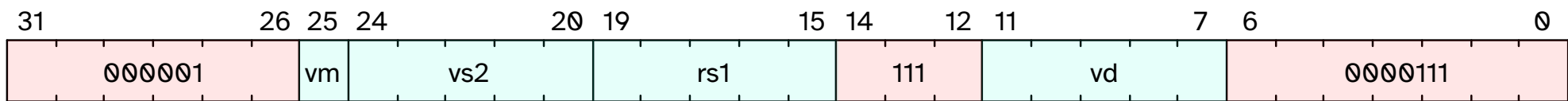
C.575. vluxei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.575.1. Encoding



C.575.2. Description

No description available.

C.575.3. Access

M	S	U
Always	Always	Always

C.575.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.575.5. IDL Operation

C.575.6. Sail Operation

```
{
  let EEW_index_pow = vlewidth_pow(width);
  let EEW_index_bytes = vlewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.575.7. Exceptions

This instruction does not generate synchronous exceptions.

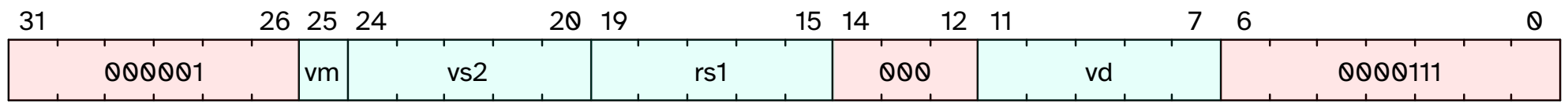
C.576. vluxei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.576.1. Encoding



C.576.2. Description

No description available.

C.576.3. Access

M	S	U
Always	Always	Always

C.576.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.576.5. IDL Operation

C.576.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8);
  let nf_int = nfields_int(nf);

  if illegal_indexed_load(vd, vm, nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal();
return RETIRE_FAIL };

  process_vlxseg(nf_int, vm, vd, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.576.7. Exceptions

This instruction does not generate synchronous exceptions.

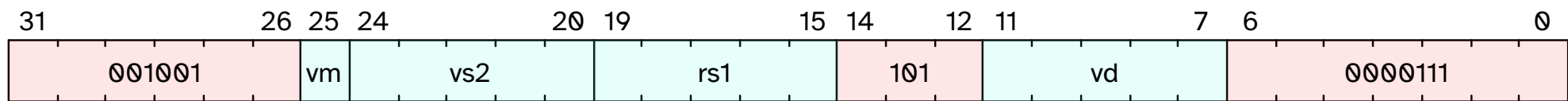
C.577. vluxseg2ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.577.1. Encoding



C.577.2. Description

No description available.

C.577.3. Access

M	S	U
Always	Always	Always

C.577.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.577.5. IDL Operation

C.577.6. Exceptions

This instruction does not generate synchronous exceptions.

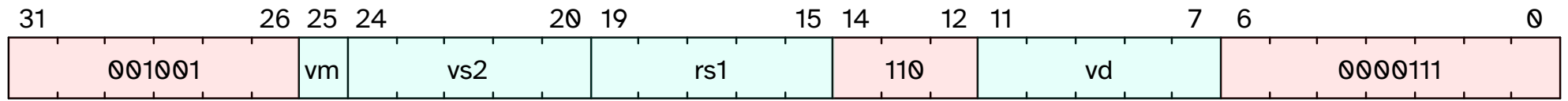
C.578. vluxseg2ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.578.1. Encoding



C.578.2. Description

No description available.

C.578.3. Access

M	S	U
Always	Always	Always

C.578.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.578.5. IDL Operation

C.578.6. Exceptions

This instruction does not generate synchronous exceptions.

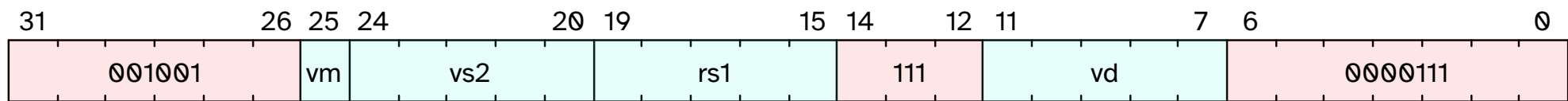
C.579. vluxseg2ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.579.1. Encoding



C.579.2. Description

No description available.

C.579.3. Access

M	S	U
Always	Always	Always

C.579.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.579.5. IDL Operation

C.579.6. Exceptions

This instruction does not generate synchronous exceptions.

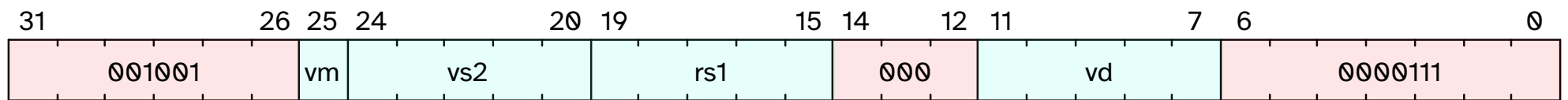
C.580. vluxseg2ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.580.1. Encoding



C.580.2. Description

No description available.

C.580.3. Access

M	S	U
Always	Always	Always

C.580.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.580.5. IDL Operation

C.580.6. Exceptions

This instruction does not generate synchronous exceptions.

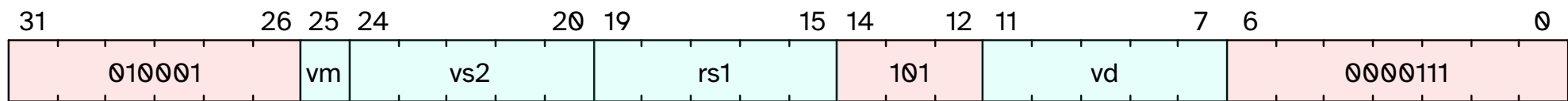
C.581. vluxseg3ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.581.1. Encoding



C.581.2. Description

No description available.

C.581.3. Access

M	S	U
Always	Always	Always

C.581.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.581.5. IDL Operation

C.581.6. Exceptions

This instruction does not generate synchronous exceptions.

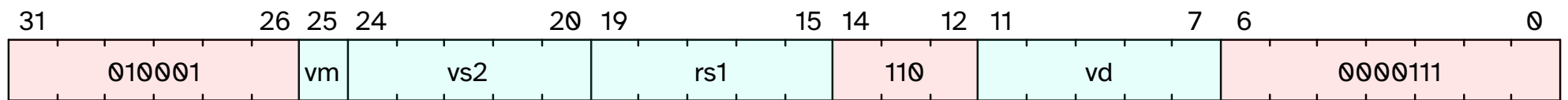
C.582. vluxseg3ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.582.1. Encoding



C.582.2. Description

No description available.

C.582.3. Access

M	S	U
Always	Always	Always

C.582.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.582.5. IDL Operation

C.582.6. Exceptions

This instruction does not generate synchronous exceptions.

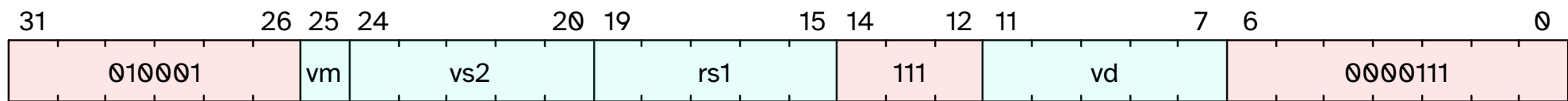
C.583. vluxseg3ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.583.1. Encoding



C.583.2. Description

No description available.

C.583.3. Access

M	S	U
Always	Always	Always

C.583.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.583.5. IDL Operation

C.583.6. Exceptions

This instruction does not generate synchronous exceptions.

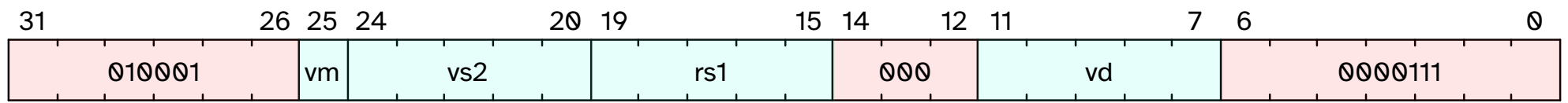
C.584. vluxseg3ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.584.1. Encoding



C.584.2. Description

No description available.

C.584.3. Access

M	S	U
Always	Always	Always

C.584.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.584.5. IDL Operation

C.584.6. Exceptions

This instruction does not generate synchronous exceptions.

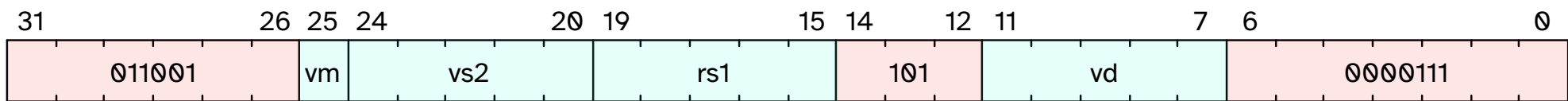
C.585. vluxseg4ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.585.1. Encoding



C.585.2. Description

No description available.

C.585.3. Access

M	S	U
Always	Always	Always

C.585.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.585.5. IDL Operation

C.585.6. Exceptions

This instruction does not generate synchronous exceptions.

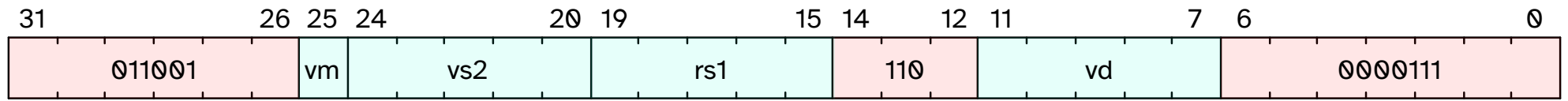
C.586. vluxseg4ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.586.1. Encoding



C.586.2. Description

No description available.

C.586.3. Access

M	S	U
Always	Always	Always

C.586.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.586.5. IDL Operation

C.586.6. Exceptions

This instruction does not generate synchronous exceptions.

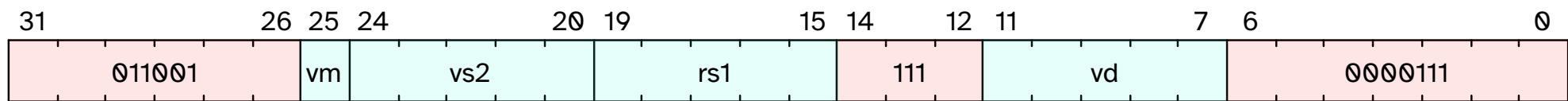
C.587. vluxseg4ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.587.1. Encoding



C.587.2. Description

No description available.

C.587.3. Access

M	S	U
Always	Always	Always

C.587.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.587.5. IDL Operation

C.587.6. Exceptions

This instruction does not generate synchronous exceptions.

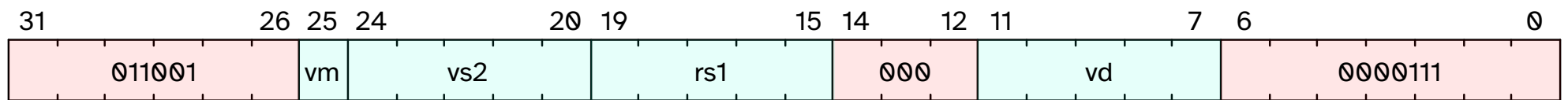
C.588. vluxseg4ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.588.1. Encoding



C.588.2. Description

No description available.

C.588.3. Access

M	S	U
Always	Always	Always

C.588.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.588.5. IDL Operation

C.588.6. Exceptions

This instruction does not generate synchronous exceptions.

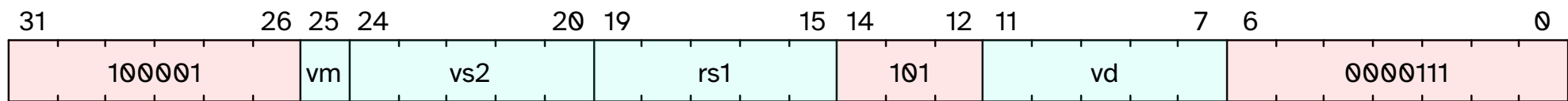
C.589. vluxseg5ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.589.1. Encoding



C.589.2. Description

No description available.

C.589.3. Access

M	S	U
Always	Always	Always

C.589.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.589.5. IDL Operation

C.589.6. Exceptions

This instruction does not generate synchronous exceptions.

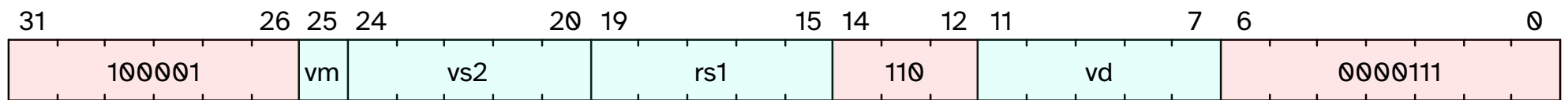
C.590. vluxseg5ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.590.1. Encoding



C.590.2. Description

No description available.

C.590.3. Access

M	S	U
Always	Always	Always

C.590.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.590.5. IDL Operation

C.590.6. Exceptions

This instruction does not generate synchronous exceptions.

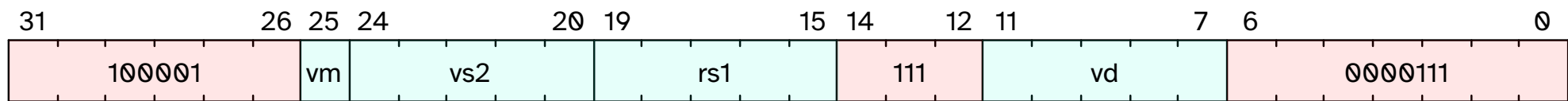
C.591. vluxseg5ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.591.1. Encoding



C.591.2. Description

No description available.

C.591.3. Access

M	S	U
Always	Always	Always

C.591.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.591.5. IDL Operation

C.591.6. Exceptions

This instruction does not generate synchronous exceptions.

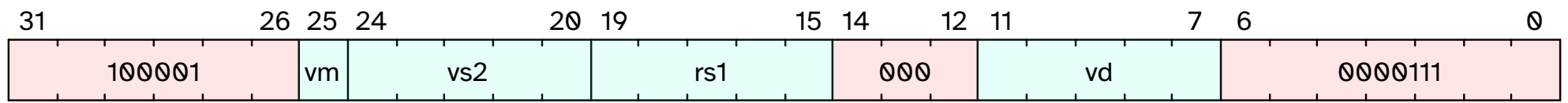
C.592. vluxseg5ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.592.1. Encoding



C.592.2. Description

No description available.

C.592.3. Access

M	S	U
Always	Always	Always

C.592.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.592.5. IDL Operation

C.592.6. Exceptions

This instruction does not generate synchronous exceptions.

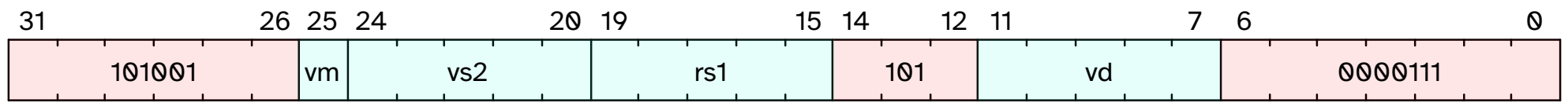
C.593. vluxseg6ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.593.1. Encoding



C.593.2. Description

No description available.

C.593.3. Access

M	S	U
Always	Always	Always

C.593.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.593.5. IDL Operation

C.593.6. Exceptions

This instruction does not generate synchronous exceptions.

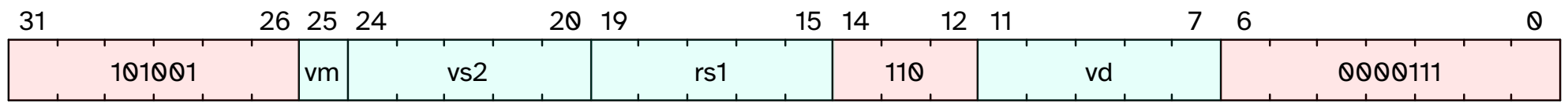
C.594. vluxseg6ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.594.1. Encoding



C.594.2. Description

No description available.

C.594.3. Access

M	S	U
Always	Always	Always

C.594.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.594.5. IDL Operation

C.594.6. Exceptions

This instruction does not generate synchronous exceptions.

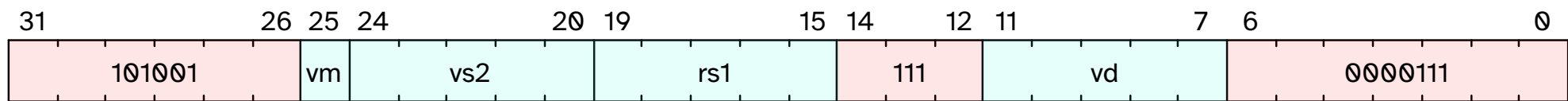
C.595. vluxseg6ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.595.1. Encoding



C.595.2. Description

No description available.

C.595.3. Access

M	S	U
Always	Always	Always

C.595.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.595.5. IDL Operation

C.595.6. Exceptions

This instruction does not generate synchronous exceptions.

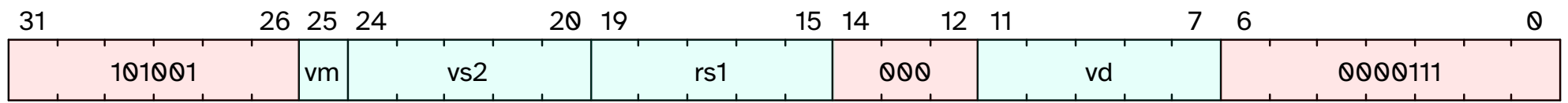
C.596. vluxseg6ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.596.1. Encoding



C.596.2. Description

No description available.

C.596.3. Access

M	S	U
Always	Always	Always

C.596.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.596.5. IDL Operation

C.596.6. Exceptions

This instruction does not generate synchronous exceptions.

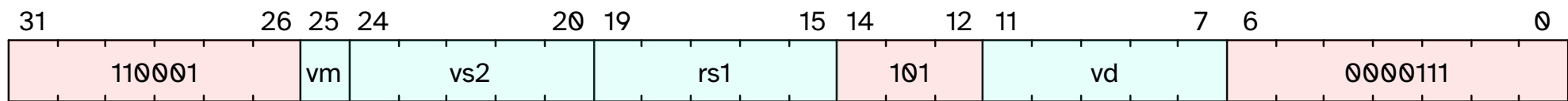
C.597. vluxseg7ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.597.1. Encoding



C.597.2. Description

No description available.

C.597.3. Access

M	S	U
Always	Always	Always

C.597.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.597.5. IDL Operation

C.597.6. Exceptions

This instruction does not generate synchronous exceptions.

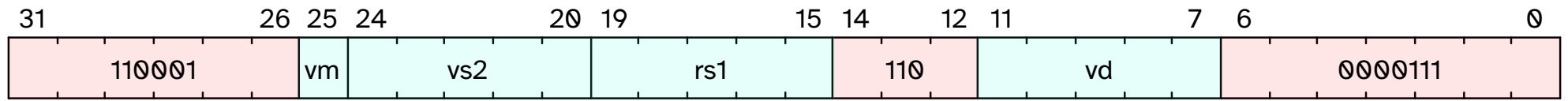
C.598. vluxseg7ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.598.1. Encoding



C.598.2. Description

No description available.

C.598.3. Access

M	S	U
Always	Always	Always

C.598.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.598.5. IDL Operation

C.598.6. Exceptions

This instruction does not generate synchronous exceptions.

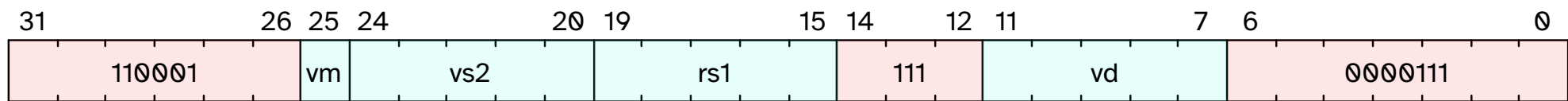
C.599. vluxseg7ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.599.1. Encoding



C.599.2. Description

No description available.

C.599.3. Access

M	S	U
Always	Always	Always

C.599.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.599.5. IDL Operation

C.599.6. Exceptions

This instruction does not generate synchronous exceptions.

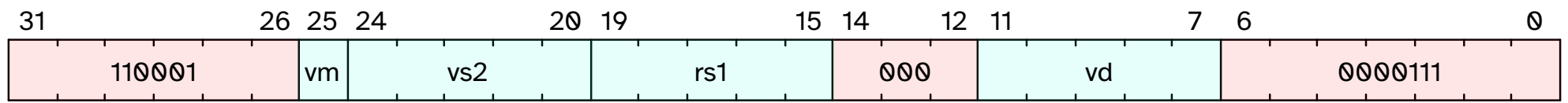
C.600. vluxseg7ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.600.1. Encoding



C.600.2. Description

No description available.

C.600.3. Access

M	S	U
Always	Always	Always

C.600.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.600.5. IDL Operation

C.600.6. Exceptions

This instruction does not generate synchronous exceptions.

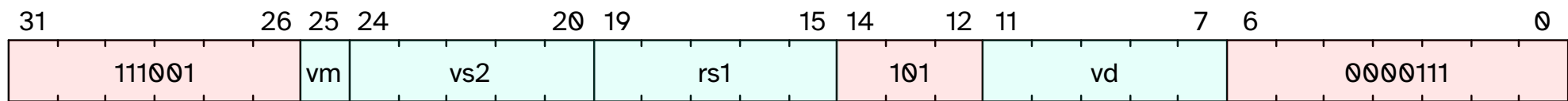
C.601. vluxseg8ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.601.1. Encoding



C.601.2. Description

No description available.

C.601.3. Access

M	S	U
Always	Always	Always

C.601.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.601.5. IDL Operation

C.601.6. Exceptions

This instruction does not generate synchronous exceptions.

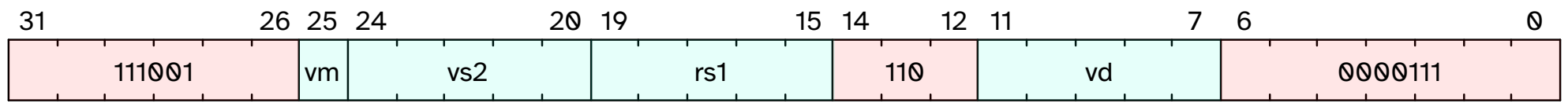
C.602. vluxseg8ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.602.1. Encoding



C.602.2. Description

No description available.

C.602.3. Access

M	S	U
Always	Always	Always

C.602.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.602.5. IDL Operation

C.602.6. Exceptions

This instruction does not generate synchronous exceptions.

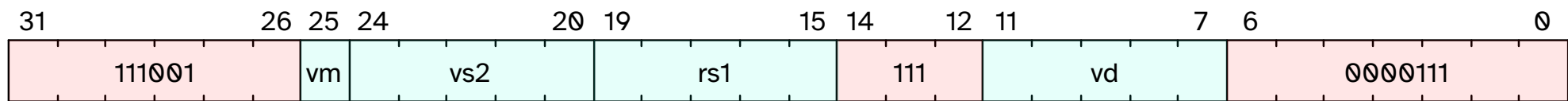
C.603. vluxseg8ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.603.1. Encoding



C.603.2. Description

No description available.

C.603.3. Access

M	S	U
Always	Always	Always

C.603.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.603.5. IDL Operation

C.603.6. Exceptions

This instruction does not generate synchronous exceptions.

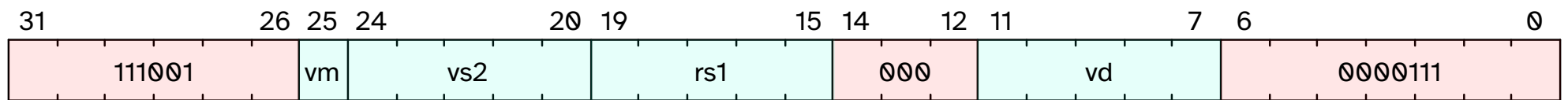
C.604. vluxseg8ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.604.1. Encoding



C.604.2. Description

No description available.

C.604.3. Access

M	S	U
Always	Always	Always

C.604.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.604.5. IDL Operation

C.604.6. Exceptions

This instruction does not generate synchronous exceptions.

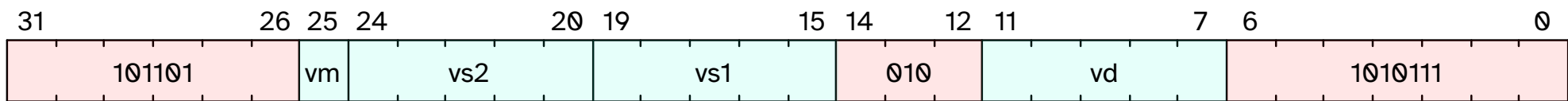
C.605. vmacc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.605.1. Encoding



C.605.2. Description

No description available.

C.605.3. Access

M	S	U
Always	Always	Always

C.605.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.605.5. IDL Operation

C.605.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VMACC => get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0) + vd_val[i],
        MVV_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0),
        MVV_VMADD  => get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0) + vs2_val[i],
        MVV_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.605.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

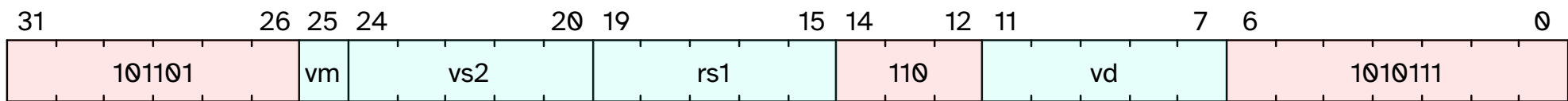
C.606. vmacc.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.606.1. Encoding



C.606.2. Description

No description available.

C.606.3. Access

M	S	U
Always	Always	Always

C.606.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.606.5. IDL Operation

C.606.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VMACC => get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0) + vd_val[i],
        MVX_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0),
        MVX_VMADD  => get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0) + vs2_val[i],
        MVX_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.606.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

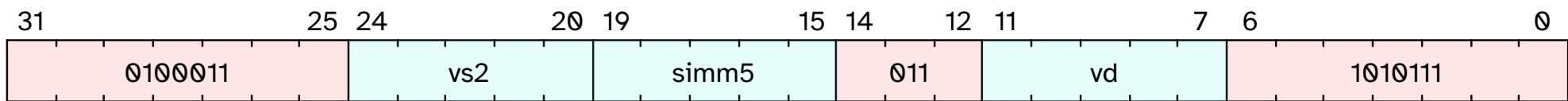
C.607. vmadc.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.607.1. Encoding



C.607.2. Description

No description available.

C.607.3. Access

M	S	U
Always	Always	Always

C.607.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.607.5. IDL Operation

C.607.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let imm_val : bits('m)          = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VIMC_VMADC => unsigned(vs2_val[i]) + unsigned(imm_val) > 2 ^ SEW - 1
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.607.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

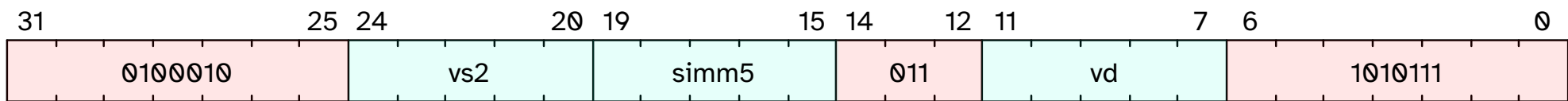
C.608. vmadc.vim

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.608.1. Encoding



C.608.2. Description

No description available.

C.608.3. Access

M	S	U
Always	Always	Always

C.608.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.608.5. IDL Operation

C.608.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask_carry(num_elem, 0b0, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VIM_VMACD => unsigned(vs2_val[i]) + unsigned(imm_val) + unsigned(bool_to_bits(vm_val[i])) > 2 ^ SEW - 1
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.608.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

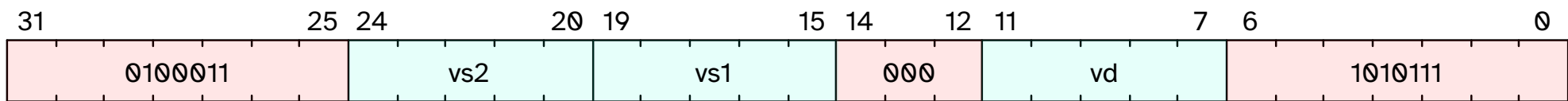
C.609. vmadc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.609.1. Encoding



C.609.2. Description

No description available.

C.609.3. Access

M	S	U
Always	Always	Always

C.609.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.609.5. IDL Operation

C.609.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVMC_VMADC => unsigned(vs2_val[i]) + unsigned(vs1_val[i]) > 2 ^ SEW - 1,
        VVMC_VMSBC => unsigned(vs2_val[i]) - unsigned(vs1_val[i]) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.609.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

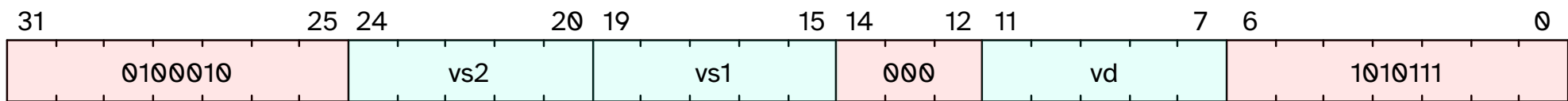
C.610. vmadc.vvm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.610.1. Encoding



C.610.2. Description

No description available.

C.610.3. Access

M	S	U
Always	Always	Always

C.610.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.610.5. IDL Operation

C.610.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask_carry(num_elem, 0b0, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVM_VMACDC => unsigned(vs2_val[i]) + unsigned(vs1_val[i]) + unsigned(bool_to_bits(vm_val[i])) > 2 ^ SEW - 1,
        VVM_VMSBC  => unsigned(vs2_val[i]) - unsigned(vs1_val[i]) - unsigned(bool_to_bits(vm_val[i])) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.610.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

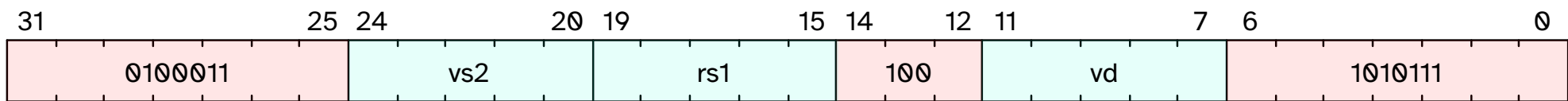
C.611. vmadc.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.611.1. Encoding



C.611.2. Description

No description available.

C.611.3. Access

M	S	U
Always	Always	Always

C.611.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.611.5. IDL Operation

C.611.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let rs1_val : bits('m)           = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXMC_VMADC => unsigned(vs2_val[i]) + unsigned(rs1_val) > 2 ^ SEW - 1,
        VXMC_VMSBC => unsigned(vs2_val[i]) - unsigned(rs1_val) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.611.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

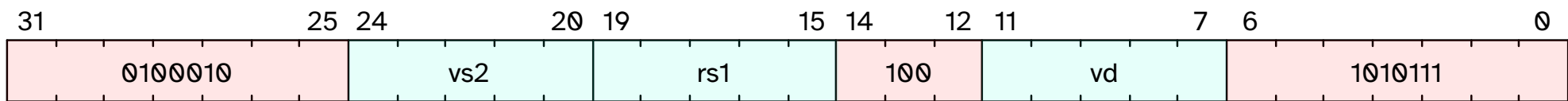
C.612. vmadc.vxm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.612.1. Encoding



C.612.2. Description

No description available.

C.612.3. Access

M	S	U
Always	Always	Always

C.612.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.612.5. IDL Operation

C.612.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask_carry(num_elem, 0b0, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXM_VMADC => unsigned(vs2_val[i]) + unsigned(rs1_val) + unsigned(bool_to_bits(vm_val[i])) > 2 ^ SEW - 1,
        VXM_VMSBC => unsigned(vs2_val[i]) - unsigned(rs1_val) - unsigned(bool_to_bits(vm_val[i])) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.612.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

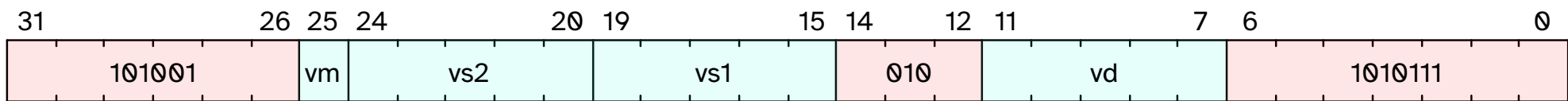
C.613. vmadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.613.1. Encoding



C.613.2. Description

No description available.

C.613.3. Access

M	S	U
Always	Always	Always

C.613.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.613.5. IDL Operation

C.613.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VMACC => get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0) + vd_val[i],
        MVV_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0),
        MVV_VMADD  => get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0) + vs2_val[i],
        MVV_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.613.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

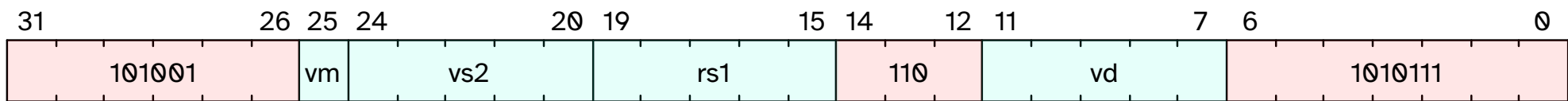
C.614. vmadd.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.614.1. Encoding



C.614.2. Description

No description available.

C.614.3. Access

M	S	U
Always	Always	Always

C.614.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.614.5. IDL Operation

C.614.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VMACC => get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0) + vd_val[i],
        MVX_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0),
        MVX_VMADD  => get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0) + vs2_val[i],
        MVX_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.614.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

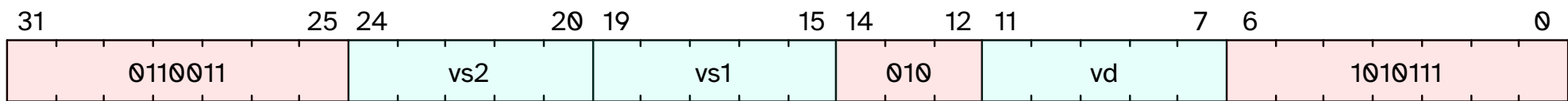
C.615. vmand.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.615.1. Encoding



C.615.2. Description

No description available.

C.615.3. Access

M	S	U
Always	Always	Always

C.615.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.615.5. IDL Operation

C.615.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, 0, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MM_VMAND      => vs2_val[i] & vs1_val[i],
        MM_VMNAND     => not(vs2_val[i] & vs1_val[i]),
        MM_VMANDNOT    => vs2_val[i] & not(vs1_val[i]),
        MM_VMXOR      => vs2_val[i] != vs1_val[i],
        MM_VMOR       => vs2_val[i] | vs1_val[i],
        MM_VMNOR      => not(vs2_val[i] | vs1_val[i]),
        MM_VMORNOT     => vs2_val[i] | not(vs1_val[i]),
        MM_VMXNOR     => vs2_val[i] == vs1_val[i]
      }
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.615.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

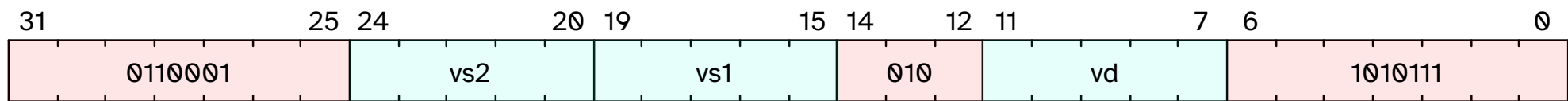
C.616. vmandn.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.616.1. Encoding



C.616.2. Description

No description available.

C.616.3. Access

M	S	U
Always	Always	Always

C.616.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.616.5. IDL Operation

C.616.6. Exceptions

This instruction does not generate synchronous exceptions.

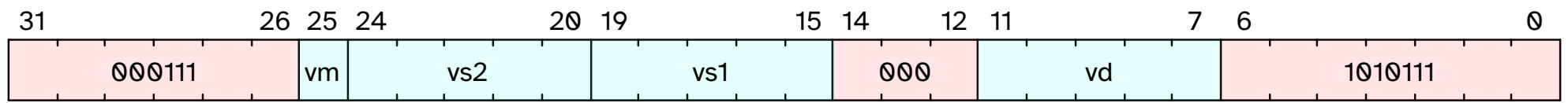
C.617. vmax.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.617.1. Encoding



C.617.2. Description

No description available.

C.617.3. Access

M	S	U
Always	Always	Always

C.617.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.617.5. IDL Operation

C.617.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```



```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

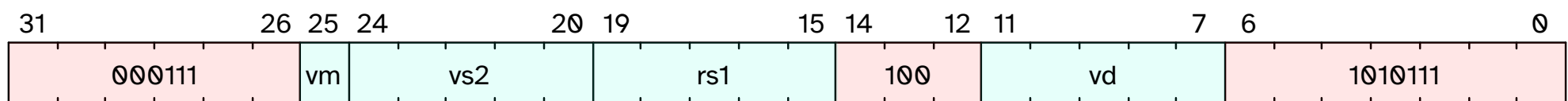
C.617.7. Exceptions

This instruction does not generate synchronous exceptions.

C.618. vmax.vx**No synopsis available**

This instruction is defined by:

- V, version >= V@1.0.0

C.618.1. Encoding**C.618.2. Description**

No description available.

C.618.3. Access

M	S	U
Always	Always	Always

C.618.4. Decode Variables

```

Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];

```

C.618.5. IDL Operation**C.618.6. Sail Operation**

```

{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()

```



```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.618.7. Exceptions

This instruction does not generate synchronous exceptions.

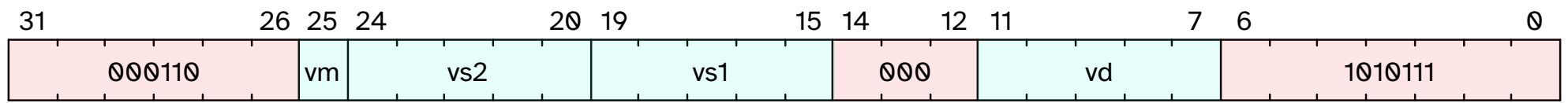
C.619. vmaxu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.619.1. Encoding



C.619.2. Description

No description available.

C.619.3. Access

M	S	U
Always	Always	Always

C.619.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.619.5. IDL Operation

C.619.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU    => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD     => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.619.7. Exceptions

This instruction does not generate synchronous exceptions.

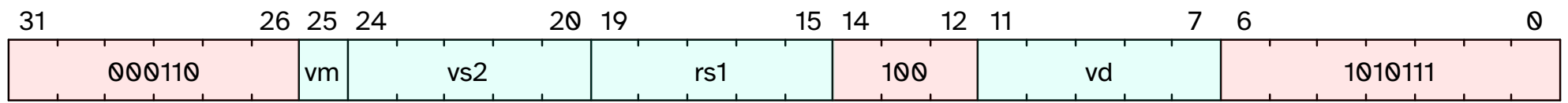
C.620. vmaxu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.620.1. Encoding



C.620.2. Description

No description available.

C.620.3. Access

M	S	U
Always	Always	Always

C.620.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.620.5. IDL Operation

C.620.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.620.7. Exceptions

This instruction does not generate synchronous exceptions.

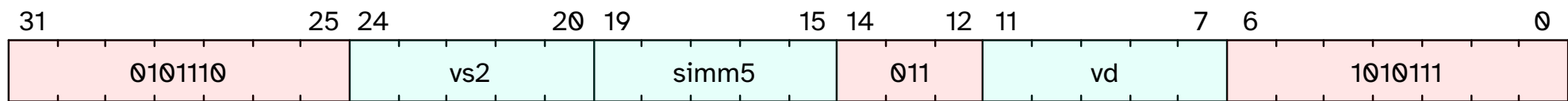
C.621. vmerge.vim

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.621.1. Encoding



C.621.2. Description

No description available.

C.621.3. Access

M	S	U
Always	Always	Always

C.621.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.621.5. IDL Operation

C.621.6. Sail Operation

```
{
  let start_element = get_start_element();
  let end_element   = get_end_element();
  let SEW           = get_sew();
  let LMUL_pow      = get_lmUL_pow();
  let num_elem      = get_num_elem(LMUL_pow, SEW); /* max(VLMAX,VLEN/SEW)) */
  let real_num_elem = if LMUL_pow >= 0 then num_elem else num_elem / (0 - LMUL_pow); /* VLMAX */

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, 0b000000);
  let imm_val : bits('m)                 = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  let tail_ag : agtype = get_vtype_vta();
  foreach (i from 0 to (num_elem - 1)) {
    if i < start_element then {
      result[i] = vd_val[i]
    } else if i > end_element | i >= real_num_elem then {
      result[i] = match tail_ag {
        UNDISTURBED => vd_val[i],
        AGNOSTIC    => vd_val[i] /* TODO: configuration support */
      }
    } else {
      /* the merge operates on all body elements */
      result[i] = if vm_val[i] then imm_val else vs2_val[i]
    }
  }
};
```



```
    write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
    vstart = zeros();  
    RETIRE_SUCCESS  
}
```

C.621.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

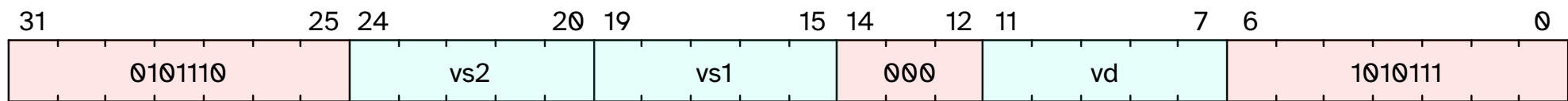
C.622. vmerge.vvm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.622.1. Encoding



C.622.2. Description

No description available.

C.622.3. Access

M	S	U
Always	Always	Always

C.622.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.622.5. IDL Operation

C.622.6. Sail Operation

```
{
  let start_element = get_start_element();
  let end_element   = get_end_element();
  let SEW           = get_sew();
  let LMUL_pow      = get_lmUL_pow();
  let num_elem      = get_num_elem(LMUL_pow, SEW); /* max(VLMAX,VLEN/SEW)) */
  let real_num_elem = if LMUL_pow >= 0 then num_elem else num_elem / (0 - LMUL_pow); /* VLMAX */

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  let tail_ag : agtype = get_vtype_vta();
  foreach (i from 0 to (num_elem - 1)) {
    if i < start_element then {
      result[i] = vd_val[i]
    } else if i > end_element | i >= real_num_elem then {
      result[i] = match tail_ag {
        UNDISTURBED => vd_val[i],
        AGNOSTIC    => vd_val[i] /* TODO: configuration support */
      }
    } else {
      /* the merge operates on all body elements */
      result[i] = if vm_val[i] then vs1_val[i] else vs2_val[i]
    }
  }
};
```



```
    write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
    vstart = zeros();  
    RETIRE_SUCCESS  
}
```

C.622.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

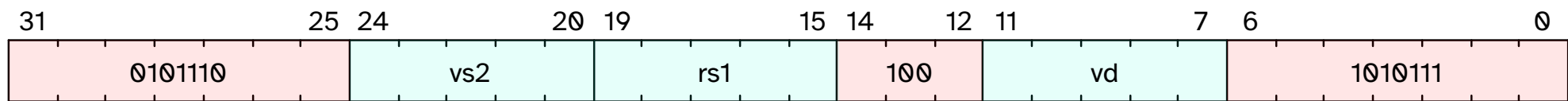
C.623. vmerge.vxm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.623.1. Encoding



C.623.2. Description

No description available.

C.623.3. Access

M	S	U
Always	Always	Always

C.623.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.623.5. IDL Operation

C.623.6. Sail Operation

```
{
  let start_element = get_start_element();
  let end_element   = get_end_element();
  let SEW           = get_sew();
  let LMUL_pow      = get_lmUL_pow();
  let num_elem      = get_num_elem(LMUL_pow, SEW); /* max(VLMAX,VLEN/SEW)) */
  let real_num_elem = if LMUL_pow >= 0 then num_elem else num_elem / (0 - LMUL_pow); /* VLMAX */

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  let tail_ag : agtype = get_vtype_vta();
  foreach (i from 0 to (num_elem - 1)) {
    if i < start_element then {
      result[i] = vd_val[i]
    } else if i > end_element | i >= real_num_elem then {
      result[i] = match tail_ag {
        UNDISTURBED => vd_val[i],
        AGNOSTIC    => vd_val[i] /* TODO: configuration support */
      }
    } else {
      /* the merge operates on all body elements */
      result[i] = if vm_val[i] then rs1_val else vs2_val[i]
    }
  }
};
```

```
    write_vreg(num_elem, SEW, LMUL_pow, vd, result);
    vstart = zeros();
    RETIRE_SUCCESS
}
```

C.623.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

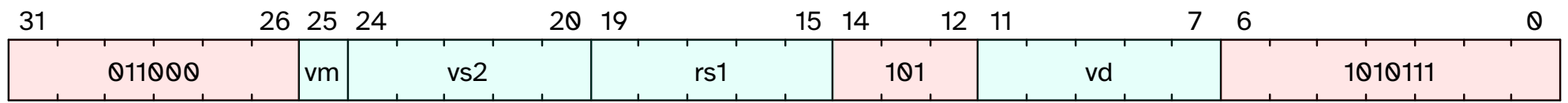
C.624. vmfeq.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.624.1. Encoding



C.624.2. Description

No description available.

C.624.3. Access

M	S	U
Always	Always	Always

C.624.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.624.5. IDL Operation

C.624.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VFM_VMFEQ => fp_eq(vs2_val[i], rs1_val),
        VFM_VMFNE => ~(fp_eq(vs2_val[i], rs1_val)),
        VFM_VMFLE => fp_le(vs2_val[i], rs1_val),
        VFM_VMFLT => fp_lt(vs2_val[i], rs1_val),
        VFM_VMFGE => fp_ge(vs2_val[i], rs1_val),
        VFM_VMFGT => fp_gt(vs2_val[i], rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.624.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

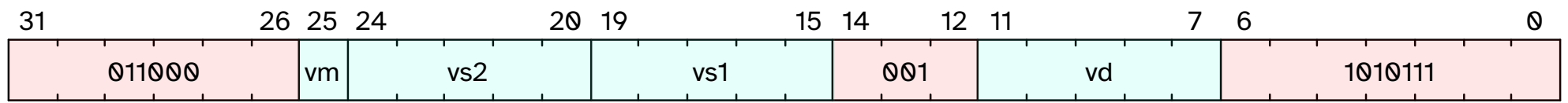
C.625. vmfeq.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.625.1. Encoding



C.625.2. Description

No description available.

C.625.3. Access

M	S	U
Always	Always	Always

C.625.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.625.5. IDL Operation

C.625.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        FVVM_VMFEQ    => fp_eq(vs2_val[i], vs1_val[i]),
        FVVM_VMFNE    => ~(fp_eq(vs2_val[i], vs1_val[i])),
        FVVM_VMFLE    => fp_le(vs2_val[i], vs1_val[i]),
        FVVM_VMFLT    => fp_lt(vs2_val[i], vs1_val[i])
      };
      result[i] = res
    }
  }
};
```



```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.625.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

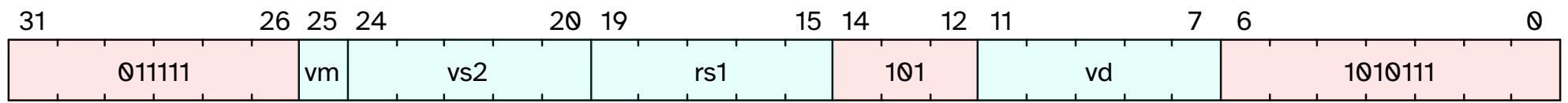
C.626. vmfge.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.626.1. Encoding



C.626.2. Description

No description available.

C.626.3. Access

M	S	U
Always	Always	Always

C.626.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.626.5. IDL Operation

C.626.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VFM_VMFEQ => fp_eq(vs2_val[i], rs1_val),
        VFM_VMFNE => ~(fp_eq(vs2_val[i], rs1_val)),
        VFM_VMFLE => fp_le(vs2_val[i], rs1_val),
        VFM_VMFLT => fp_lt(vs2_val[i], rs1_val),
        VFM_VMFGE => fp_ge(vs2_val[i], rs1_val),
        VFM_VMFGT => fp_gt(vs2_val[i], rs1_val)
      };
      result[i] = res
    }
  }
}
```



```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.626.7. Exceptions

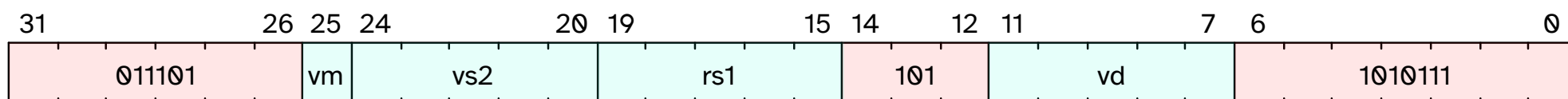
This instruction does not generate synchronous exceptions.

DRAFT

C.627. vmfgt.vf**No synopsis available**

This instruction is defined by:

- V, version >= V@1.0.0

C.627.1. Encoding**C.627.2. Description**

No description available.

C.627.3. Access

M	S	U
Always	Always	Always

C.627.4. Decode Variables

```

Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];

```

C.627.5. IDL Operation**C.627.6. Sail Operation**

```

{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VFM_VMFEQ => fp_eq(vs2_val[i], rs1_val),
        VFM_VMFNE => ~(fp_eq(vs2_val[i], rs1_val)),
        VFM_VMFLE => fp_le(vs2_val[i], rs1_val),
        VFM_VMFLT => fp_lt(vs2_val[i], rs1_val),
        VFM_VMFGE => fp_ge(vs2_val[i], rs1_val),
        VFM_VMFGT => fp_gt(vs2_val[i], rs1_val)
      };
      result[i] = res
    }
  }
}

```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.627.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

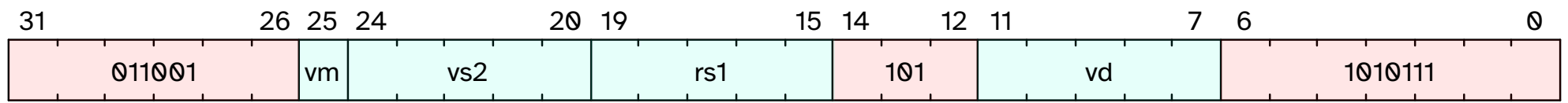
C.628. vmfle.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.628.1. Encoding



C.628.2. Description

No description available.

C.628.3. Access

M	S	U
Always	Always	Always

C.628.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.628.5. IDL Operation

C.628.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VFM_VMFEQ => fp_eq(vs2_val[i], rs1_val),
        VFM_VMFNE => ~(fp_eq(vs2_val[i], rs1_val)),
        VFM_VMFLE => fp_le(vs2_val[i], rs1_val),
        VFM_VMFLT => fp_lt(vs2_val[i], rs1_val),
        VFM_VMFGE => fp_ge(vs2_val[i], rs1_val),
        VFM_VMFGT => fp_gt(vs2_val[i], rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.628.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

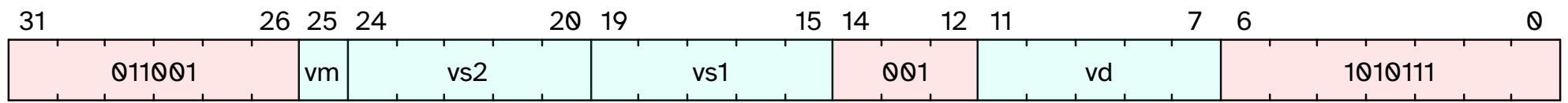
C.629. vmfle.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.629.1. Encoding



C.629.2. Description

No description available.

C.629.3. Access

M	S	U
Always	Always	Always

C.629.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.629.5. IDL Operation

C.629.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        FVVM_VMFEQ => fp_eq(vs2_val[i], vs1_val[i]),
        FVVM_VMFNE => ~(fp_eq(vs2_val[i], vs1_val[i])),
        FVVM_VMFLE => fp_le(vs2_val[i], vs1_val[i]),
        FVVM_VMFLT => fp_lt(vs2_val[i], vs1_val[i])
      };
      result[i] = res
    }
  }
};
```



```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.629.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

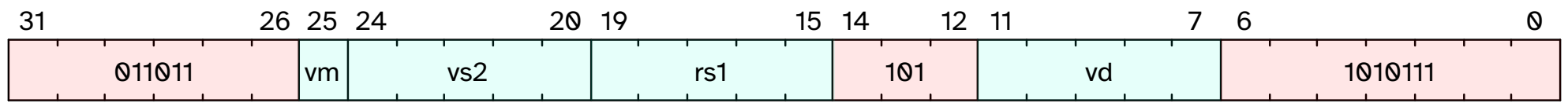
C.630. vmflt.vf

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.630.1. Encoding



C.630.2. Description

No description available.

C.630.3. Access

M	S	U
Always	Always	Always

C.630.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.630.5. IDL Operation

C.630.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VFM_VMFEQ => fp_eq(vs2_val[i], rs1_val),
        VFM_VMFNE => ~(fp_eq(vs2_val[i], rs1_val)),
        VFM_VMFLE => fp_le(vs2_val[i], rs1_val),
        VFM_VMFLT => fp_lt(vs2_val[i], rs1_val),
        VFM_VMFGE => fp_ge(vs2_val[i], rs1_val),
        VFM_VMFGT => fp_gt(vs2_val[i], rs1_val)
      };
      result[i] = res
    }
  }
}
```



```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.630.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

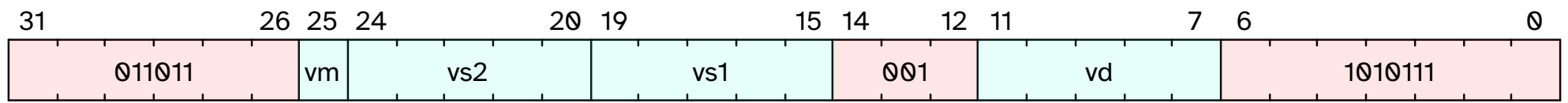
C.631. vmflt.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.631.1. Encoding



C.631.2. Description

No description available.

C.631.3. Access

M	S	U
Always	Always	Always

C.631.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.631.5. IDL Operation

C.631.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        FVVM_VMFEQ => fp_eq(vs2_val[i], vs1_val[i]),
        FVVM_VMFNE => ~(fp_eq(vs2_val[i], vs1_val[i])),
        FVVM_VMFLE => fp_le(vs2_val[i], vs1_val[i]),
        FVVM_VMFLT => fp_lt(vs2_val[i], vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
    write_vmask(num_elem, vd, result);
    vstart = zeros();
    RETIRE_SUCCESS
}
```

C.631.7. Exceptions

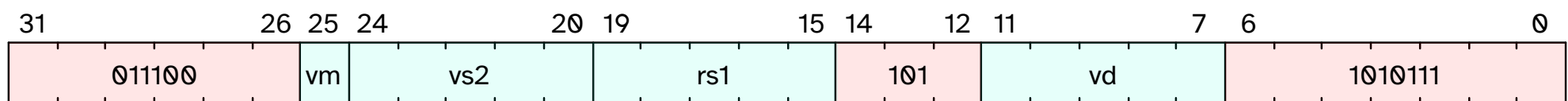
This instruction does not generate synchronous exceptions.

DRAFT

C.632. vmfne.vf**No synopsis available**

This instruction is defined by:

- V, version >= V@1.0.0

C.632.1. Encoding**C.632.2. Description**

No description available.

C.632.3. Access

M	S	U
Always	Always	Always

C.632.4. Decode Variables

```

Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];

```

C.632.5. IDL Operation**C.632.6. Sail Operation**

```

{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar_fp(rs1, 'm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VFM_VMFEQ => fp_eq(vs2_val[i], rs1_val),
        VFM_VMFNE => ~(fp_eq(vs2_val[i], rs1_val)),
        VFM_VMFLE => fp_le(vs2_val[i], rs1_val),
        VFM_VMFLT => fp_lt(vs2_val[i], rs1_val),
        VFM_VMFGE => fp_ge(vs2_val[i], rs1_val),
        VFM_VMFGT => fp_gt(vs2_val[i], rs1_val)
      };
      result[i] = res
    }
  }
}

```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.632.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

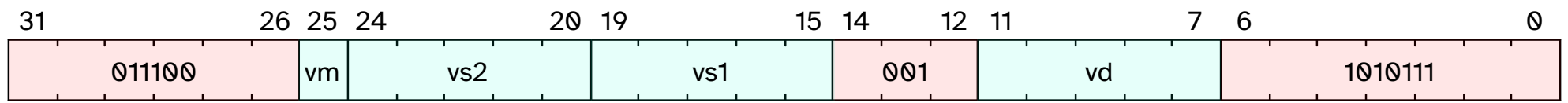
C.633. vmfne.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.633.1. Encoding



C.633.2. Description

No description available.

C.633.3. Access

M	S	U
Always	Always	Always

C.633.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.633.5. IDL Operation

C.633.6. Sail Operation

```
{
  let rm_3b    = fcsr.FRM();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_fp_vd_unmasked(SEW, rm_3b) then { handle_illegal(); return RETIRE_FAIL };
  assert(SEW != 8);

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        FVVM_VMFEQ => fp_eq(vs2_val[i], vs1_val[i]),
        FVVM_VMFNE => ~(fp_eq(vs2_val[i], vs1_val[i])),
        FVVM_VMFLE => fp_le(vs2_val[i], vs1_val[i]),
        FVVM_VMFLT => fp_lt(vs2_val[i], vs1_val[i])
      };
      result[i] = res
    }
  }
};
```



```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.633.7. Exceptions

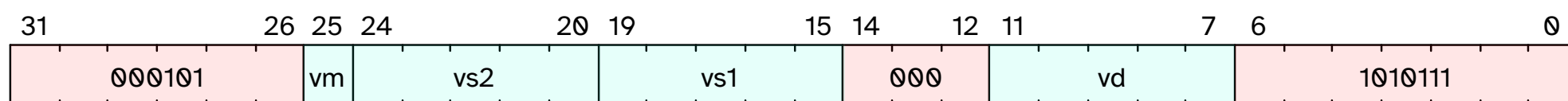
This instruction does not generate synchronous exceptions.

DRAFT

C.634. vmin.vv**No synopsis available**

This instruction is defined by:

- V, version >= V@1.0.0

C.634.1. Encoding**C.634.2. Description**

No description available.

C.634.3. Access

M	S	U
Always	Always	Always

C.634.4. Decode Variables

```

Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];

```

C.634.5. IDL Operation**C.634.6. Sail Operation**

```

{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU    => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD     => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {

```



```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.634.7. Exceptions

This instruction does not generate synchronous exceptions.

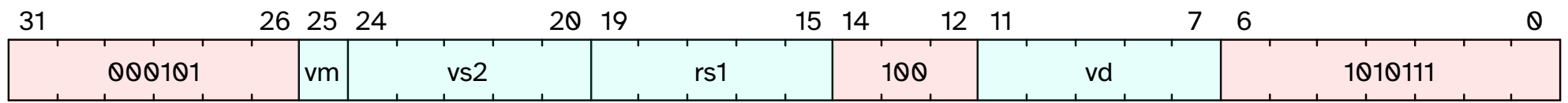
C.635. vmin.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.635.1. Encoding



C.635.2. Description

No description available.

C.635.3. Access

M	S	U
Always	Always	Always

C.635.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.635.5. IDL Operation

C.635.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.635.7. Exceptions

This instruction does not generate synchronous exceptions.

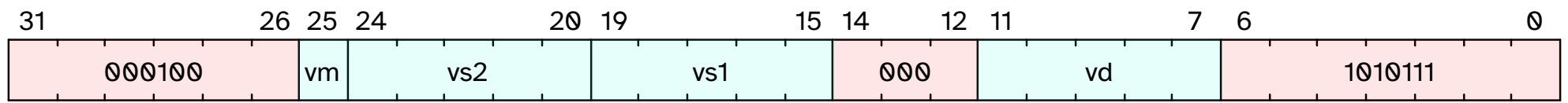
C.636. vminu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.636.1. Encoding



C.636.2. Description

No description available.

C.636.3. Access

M	S	U
Always	Always	Always

C.636.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.636.5. IDL Operation

C.636.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.636.7. Exceptions

This instruction does not generate synchronous exceptions.

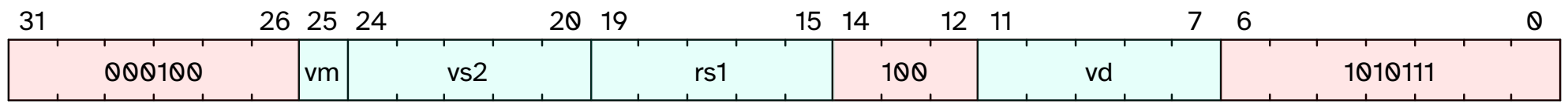
C.637. vminu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.637.1. Encoding



C.637.2. Description

No description available.

C.637.3. Access

M	S	U
Always	Always	Always

C.637.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.637.5. IDL Operation

C.637.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```



```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.637.7. Exceptions

This instruction does not generate synchronous exceptions.

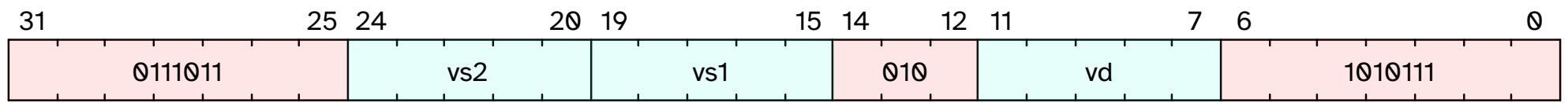
C.638. vmnand.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.638.1. Encoding



C.638.2. Description

No description available.

C.638.3. Access

M	S	U
Always	Always	Always

C.638.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.638.5. IDL Operation

C.638.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, 0, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MM_VMAND      => vs2_val[i] & vs1_val[i],
        MM_VMNAND     => not(vs2_val[i] & vs1_val[i]),
        MM_VMANDNOT   => vs2_val[i] & not(vs1_val[i]),
        MM_VMXOR      => vs2_val[i] != vs1_val[i],
        MM_VMOR       => vs2_val[i] | vs1_val[i],
        MM_VMNOR      => not(vs2_val[i] | vs1_val[i]),
        MM_VMORNOT    => vs2_val[i] | not(vs1_val[i]),
        MM_VMXNOR     => vs2_val[i] == vs1_val[i]
      }
    }
  }
};
```



```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.638.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

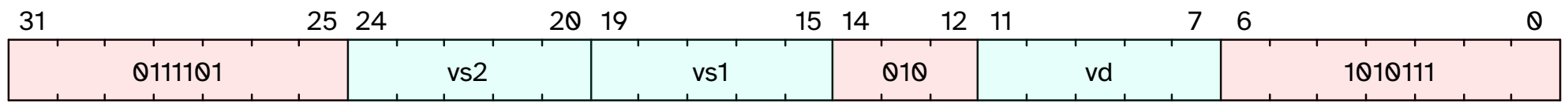
C.639. vmnor.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.639.1. Encoding



C.639.2. Description

No description available.

C.639.3. Access

M	S	U
Always	Always	Always

C.639.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.639.5. IDL Operation

C.639.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, 0, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MM_VMAND      => vs2_val[i] & vs1_val[i],
        MM_VMNAND     => not(vs2_val[i] & vs1_val[i]),
        MM_VMANDNOT   => vs2_val[i] & not(vs1_val[i]),
        MM_VMXOR      => vs2_val[i] != vs1_val[i],
        MM_VMOR       => vs2_val[i] | vs1_val[i],
        MM_VMNOR      => not(vs2_val[i] | vs1_val[i]),
        MM_VMORNOT    => vs2_val[i] | not(vs1_val[i]),
        MM_VMXNOR     => vs2_val[i] == vs1_val[i]
      }
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.639.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

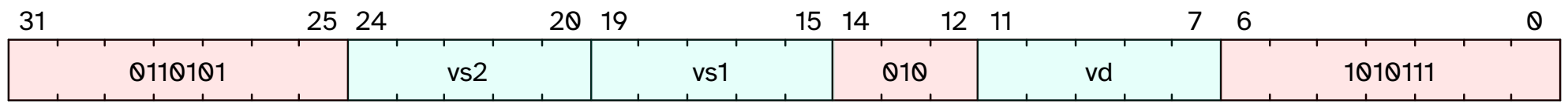
C.640. vmor.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.640.1. Encoding



C.640.2. Description

No description available.

C.640.3. Access

M	S	U
Always	Always	Always

C.640.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.640.5. IDL Operation

C.640.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, 0, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MM_VMAND      => vs2_val[i] & vs1_val[i],
        MM_VMNAND     => not(vs2_val[i] & vs1_val[i]),
        MM_VMANDNOT   => vs2_val[i] & not(vs1_val[i]),
        MM_VMXOR      => vs2_val[i] != vs1_val[i],
        MM_VMOR       => vs2_val[i] | vs1_val[i],
        MM_VMNOR      => not(vs2_val[i] | vs1_val[i]),
        MM_VMORNOT    => vs2_val[i] | not(vs1_val[i]),
        MM_VMXNOR     => vs2_val[i] == vs1_val[i]
      }
    }
  }
};
```

```
write_vmask(num_elem, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.640.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

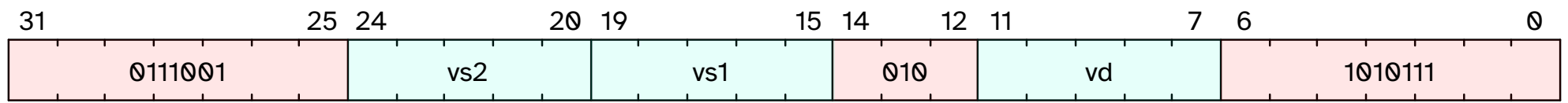
C.641. vmorn.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.641.1. Encoding



C.641.2. Description

No description available.

C.641.3. Access

M	S	U
Always	Always	Always

C.641.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.641.5. IDL Operation

C.641.6. Exceptions

This instruction does not generate synchronous exceptions.

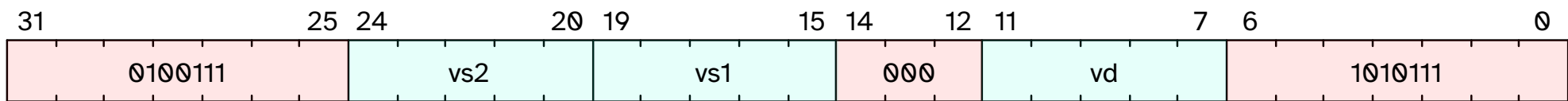
C.642. vmsbc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.642.1. Encoding



C.642.2. Description

No description available.

C.642.3. Access

M	S	U
Always	Always	Always

C.642.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.642.5. IDL Operation

C.642.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVMC_VMADC => unsigned(vs2_val[i]) + unsigned(vs1_val[i]) > 2 ^ SEW - 1,
        VVMC_VMSBC => unsigned(vs2_val[i]) - unsigned(vs1_val[i]) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.642.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

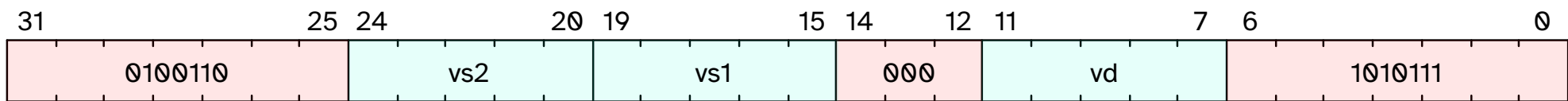
C.643. vmsbc.vvm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.643.1. Encoding



C.643.2. Description

No description available.

C.643.3. Access

M	S	U
Always	Always	Always

C.643.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.643.5. IDL Operation

C.643.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask_carry(num_elem, 0b0, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVM_VMADC => unsigned(vs2_val[i]) + unsigned(vs1_val[i]) + unsigned(bool_to_bits(vm_val[i])) > 2 ^ SEW - 1,
        VVM_VMSBC => unsigned(vs2_val[i]) - unsigned(vs1_val[i]) - unsigned(bool_to_bits(vm_val[i])) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.643.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

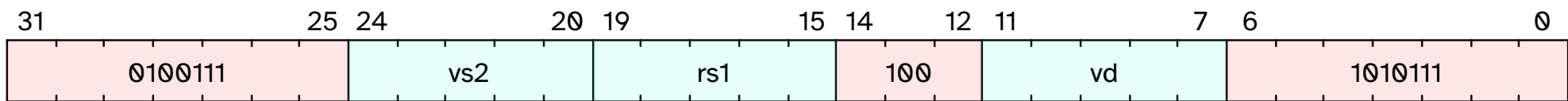
C.644. vmsbc.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.644.1. Encoding



C.644.2. Description

No description available.

C.644.3. Access

M	S	U
Always	Always	Always

C.644.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.644.5. IDL Operation

C.644.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let rs1_val : bits('m)          = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXMC_VMADC => unsigned(vs2_val[i]) + unsigned(rs1_val) > 2 ^ SEW - 1,
        VXMC_VMSBC => unsigned(vs2_val[i]) - unsigned(rs1_val) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.644.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

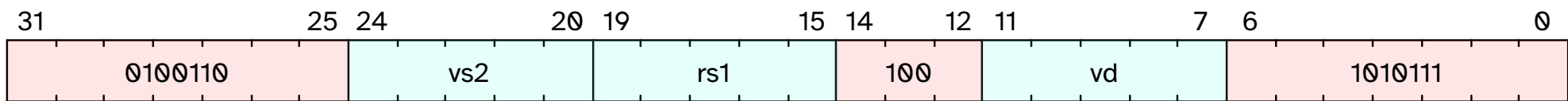
C.645. vmsbc.vxm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.645.1. Encoding



C.645.2. Description

No description available.

C.645.3. Access

M	S	U
Always	Always	Always

C.645.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.645.5. IDL Operation

C.645.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask_carry(num_elem, 0b0, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, LMUL_pow, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXM_VMADC => unsigned(vs2_val[i]) + unsigned(rs1_val) + unsigned(bool_to_bits(vm_val[i])) > 2 ^ SEW - 1,
        VXM_VMSBC => unsigned(vs2_val[i]) - unsigned(rs1_val) - unsigned(bool_to_bits(vm_val[i])) < 0
      };
      result[i] = res
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.645.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

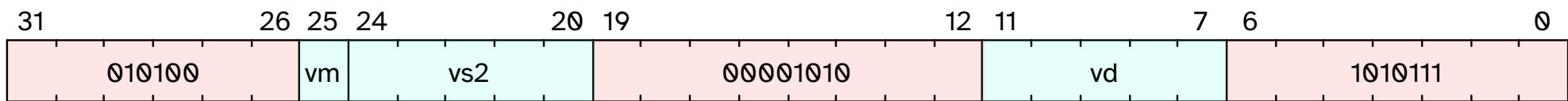
C.646. vmsbf.m

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.646.1. Encoding



C.646.2. Description

No description available.

C.646.3. Access

M	S	U
Always	Always	Always

C.646.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.646.5. IDL Operation

C.646.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_normal(vd, vm) | not(assert_vstart(0)) | vd == vs2
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, 0, vd_val, vm_val);

  found_elem : bool = false;
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      if vs2_val[i] then found_elem = true;
      result[i] = if found_elem then false else true
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.646.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

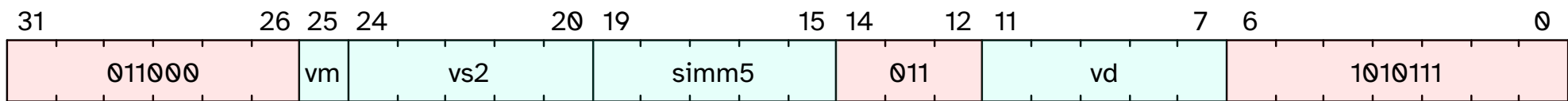
C.647. vmseq.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.647.1. Encoding



C.647.2. Description

No description available.

C.647.3. Access

M	S	U
Always	Always	Always

C.647.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.647.5. IDL Operation

C.647.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VICMP_VMSEQ  => vs2_val[i] == imm_val,
        VICMP_VMSNE  => vs2_val[i] != imm_val,
        VICMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(imm_val),
        VICMP_VMSLE  => signed(vs2_val[i]) <= signed(imm_val),
        VICMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(imm_val),
        VICMP_VMSGT  => signed(vs2_val[i]) > signed(imm_val)
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.647.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

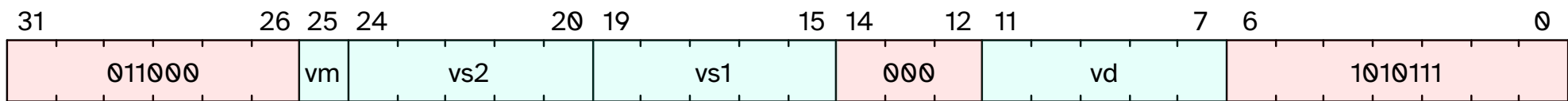
C.648. vmseq.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.648.1. Encoding



C.648.2. Description

No description available.

C.648.3. Access

M	S	U
Always	Always	Always

C.648.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.648.5. IDL Operation

C.648.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVCMP_VMSEQ => vs2_val[i] == vs1_val[i],
        VVCMP_VMSNE => vs2_val[i] != vs1_val[i],
        VVCMP_VMSLTU => unsigned(vs2_val[i]) < unsigned(vs1_val[i]),
        VVCMP_VMSLT  => signed(vs2_val[i]) < signed(vs1_val[i]),
        VVCMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(vs1_val[i]),
        VVCMP_VMSLE  => signed(vs2_val[i]) <= signed(vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.648.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

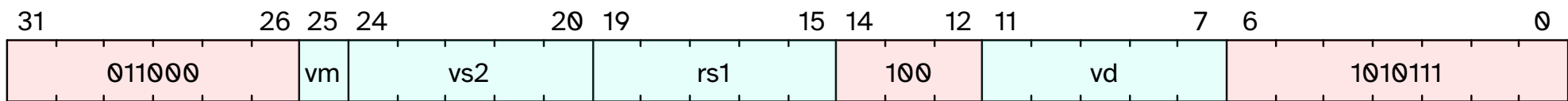
C.649. vmseq.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.649.1. Encoding



C.649.2. Description

No description available.

C.649.3. Access

M	S	U
Always	Always	Always

C.649.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.649.5. IDL Operation

C.649.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.649.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

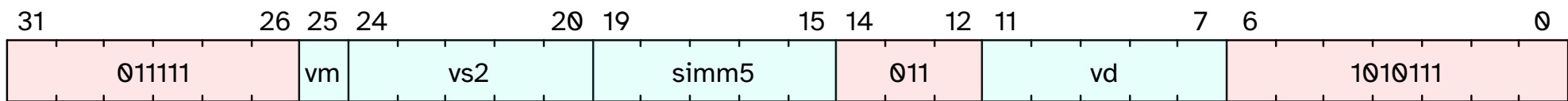
C.650. vmsgt.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.650.1. Encoding



C.650.2. Description

No description available.

C.650.3. Access

M	S	U
Always	Always	Always

C.650.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.650.5. IDL Operation

C.650.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VICMP_VMSEQ => vs2_val[i] == imm_val,
        VICMP_VMSNE => vs2_val[i] != imm_val,
        VICMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(imm_val),
        VICMP_VMSLE  => signed(vs2_val[i]) <= signed(imm_val),
        VICMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(imm_val),
        VICMP_VMSGT  => signed(vs2_val[i]) > signed(imm_val)
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.650.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

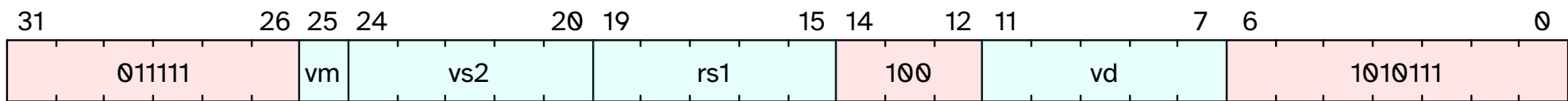
C.651. vmsgt.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.651.1. Encoding



C.651.2. Description

No description available.

C.651.3. Access

M	S	U
Always	Always	Always

C.651.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.651.5. IDL Operation

C.651.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.651.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

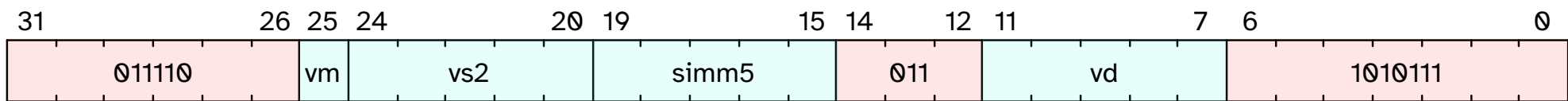
C.652. vmsgtu.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.652.1. Encoding



C.652.2. Description

No description available.

C.652.3. Access

M	S	U
Always	Always	Always

C.652.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.652.5. IDL Operation

C.652.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m))   = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VICMP_VMSEQ  => vs2_val[i] == imm_val,
        VICMP_VMSNE  => vs2_val[i] != imm_val,
        VICMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(imm_val),
        VICMP_VMSLE  => signed(vs2_val[i]) <= signed(imm_val),
        VICMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(imm_val),
        VICMP_VMSGT  => signed(vs2_val[i]) > signed(imm_val)
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.652.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

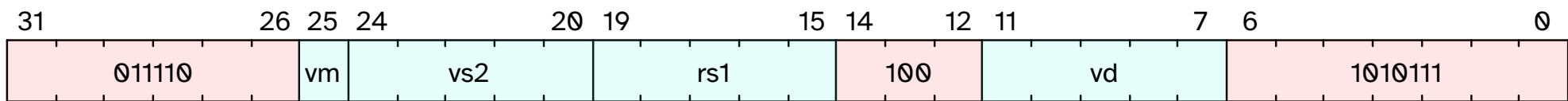
C.653. vmsgtu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.653.1. Encoding



C.653.2. Description

No description available.

C.653.3. Access

M	S	U
Always	Always	Always

C.653.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.653.5. IDL Operation

C.653.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.653.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

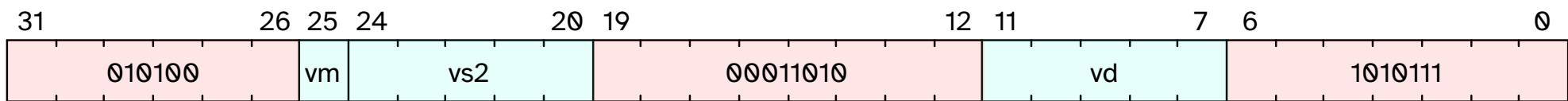
C.654. vmsif.m

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.654.1. Encoding



C.654.2. Description

No description available.

C.654.3. Access

M	S	U
Always	Always	Always

C.654.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.654.5. IDL Operation

C.654.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_normal(vd, vm) | not(assert_vstart(0)) | vd == vs2
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, 0, vd_val, vm_val);

  found_elem : bool = false;
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = if found_elem then false else true;
      if vs2_val[i] then found_elem = true
    }
  };

  write_vmask(num_elem, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.654.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

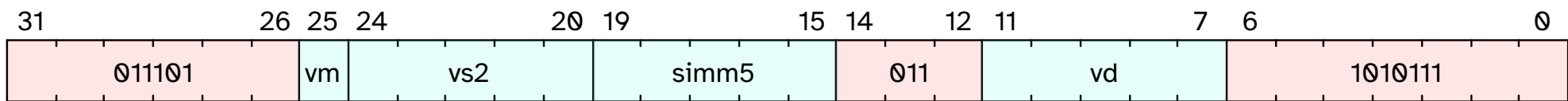
C.655. vmsle.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.655.1. Encoding



C.655.2. Description

No description available.

C.655.3. Access

M	S	U
Always	Always	Always

C.655.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.655.5. IDL Operation

C.655.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VICMP_VMSEQ => vs2_val[i] == imm_val,
        VICMP_VMSNE => vs2_val[i] != imm_val,
        VICMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(imm_val),
        VICMP_VMSLE  => signed(vs2_val[i]) <= signed(imm_val),
        VICMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(imm_val),
        VICMP_VMSGT  => signed(vs2_val[i]) > signed(imm_val)
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.655.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

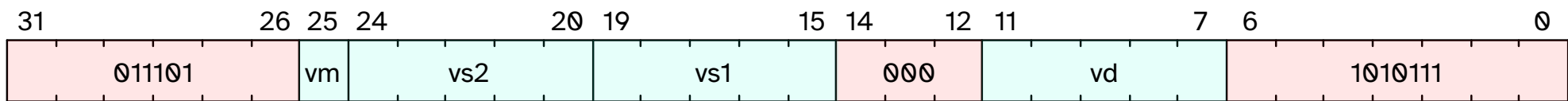
C.656. vmsle.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.656.1. Encoding



C.656.2. Description

No description available.

C.656.3. Access

M	S	U
Always	Always	Always

C.656.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.656.5. IDL Operation

C.656.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVCMP_VMSEQ => vs2_val[i] == vs1_val[i],
        VVCMP_VMSNE => vs2_val[i] != vs1_val[i],
        VVCMP_VMSLTU => unsigned(vs2_val[i]) < unsigned(vs1_val[i]),
        VVCMP_VMSLT  => signed(vs2_val[i]) < signed(vs1_val[i]),
        VVCMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(vs1_val[i]),
        VVCMP_VMSLE  => signed(vs2_val[i]) <= signed(vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.656.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

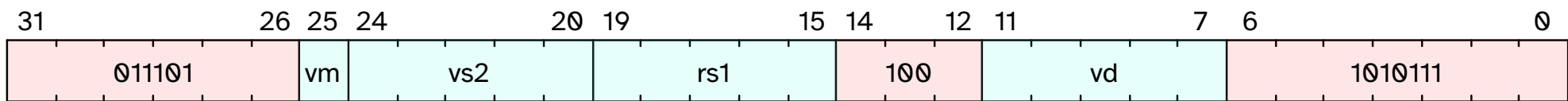
C.657. vmsle.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.657.1. Encoding



C.657.2. Description

No description available.

C.657.3. Access

M	S	U
Always	Always	Always

C.657.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.657.5. IDL Operation

C.657.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.657.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

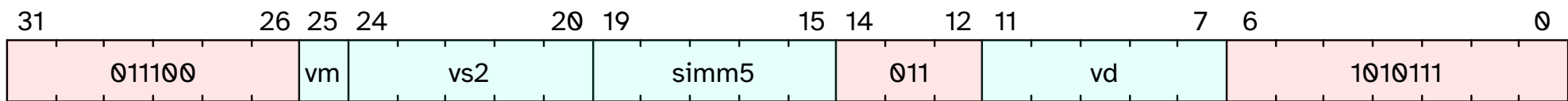
C.658. vmsleu.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.658.1. Encoding



C.658.2. Description

No description available.

C.658.3. Access

M	S	U
Always	Always	Always

C.658.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.658.5. IDL Operation

C.658.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VICMP_VMSEQ => vs2_val[i] == imm_val,
        VICMP_VMSNE => vs2_val[i] != imm_val,
        VICMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(imm_val),
        VICMP_VMSLE  => signed(vs2_val[i]) <= signed(imm_val),
        VICMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(imm_val),
        VICMP_VMSGT  => signed(vs2_val[i]) > signed(imm_val)
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.658.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

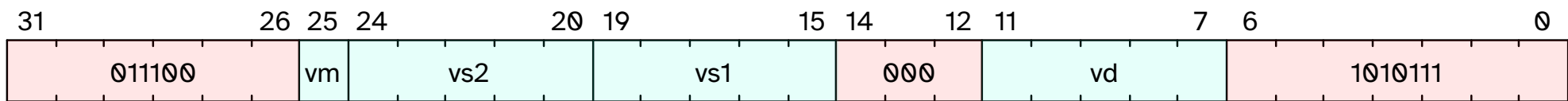
C.659. vmsleu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.659.1. Encoding



C.659.2. Description

No description available.

C.659.3. Access

M	S	U
Always	Always	Always

C.659.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.659.5. IDL Operation

C.659.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVCMP_VMSEQ  => vs2_val[i] == vs1_val[i],
        VVCMP_VMSNE  => vs2_val[i] != vs1_val[i],
        VVCMP_VMSLTU => unsigned(vs2_val[i]) < unsigned(vs1_val[i]),
        VVCMP_VMSLT  => signed(vs2_val[i]) < signed(vs1_val[i]),
        VVCMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(vs1_val[i]),
        VVCMP_VMSLE  => signed(vs2_val[i]) <= signed(vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.659.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

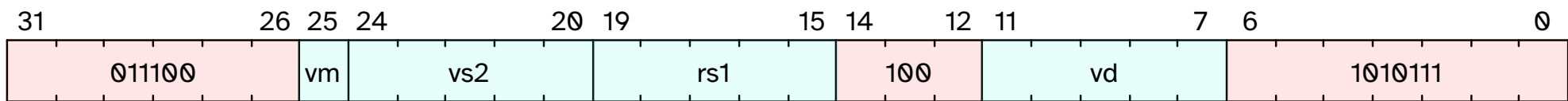
C.660. vmsleu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.660.1. Encoding



C.660.2. Description

No description available.

C.660.3. Access

M	S	U
Always	Always	Always

C.660.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.660.5. IDL Operation

C.660.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.660.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

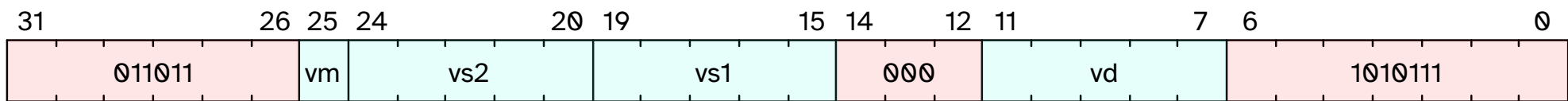
C.661. vmslt.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.661.1. Encoding



C.661.2. Description

No description available.

C.661.3. Access

M	S	U
Always	Always	Always

C.661.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.661.5. IDL Operation

C.661.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVCMP_VMSEQ    => vs2_val[i] == vs1_val[i],
        VVCMP_VMSNE    => vs2_val[i] != vs1_val[i],
        VVCMP_VMSLTU    => unsigned(vs2_val[i]) < unsigned(vs1_val[i]),
        VVCMP_VMSLT     => signed(vs2_val[i]) < signed(vs1_val[i]),
        VVCMP_VMSLEU    => unsigned(vs2_val[i]) <= unsigned(vs1_val[i]),
        VVCMP_VMSLE     => signed(vs2_val[i]) <= signed(vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.661.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

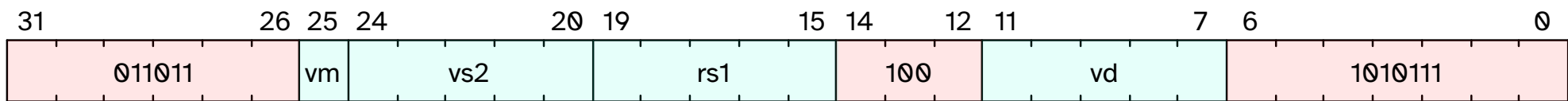
C.662. vmslt.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.662.1. Encoding



C.662.2. Description

No description available.

C.662.3. Access

M	S	U
Always	Always	Always

C.662.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.662.5. IDL Operation

C.662.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.662.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

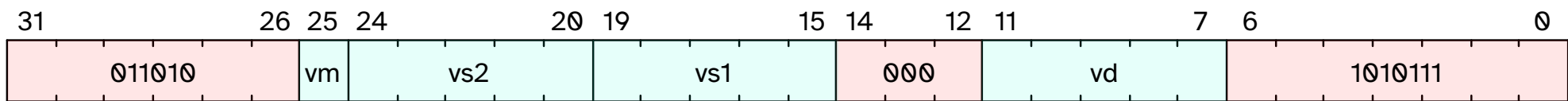
C.663. vmsltu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.663.1. Encoding



C.663.2. Description

No description available.

C.663.3. Access

M	S	U
Always	Always	Always

C.663.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.663.5. IDL Operation

C.663.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVCMP_VMSEQ  => vs2_val[i] == vs1_val[i],
        VVCMP_VMSNE  => vs2_val[i] != vs1_val[i],
        VVCMP_VMSLTU => unsigned(vs2_val[i]) < unsigned(vs1_val[i]),
        VVCMP_VMSLT  => signed(vs2_val[i]) < signed(vs1_val[i]),
        VVCMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(vs1_val[i]),
        VVCMP_VMSLE  => signed(vs2_val[i]) <= signed(vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.663.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

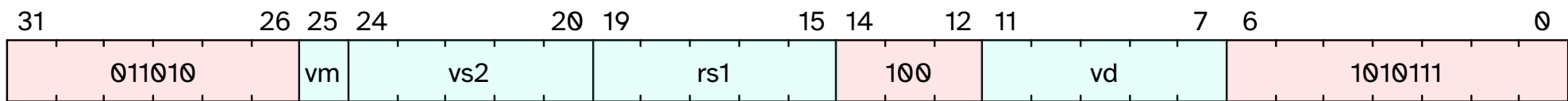
C.664. vmsltu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.664.1. Encoding



C.664.2. Description

No description available.

C.664.3. Access

M	S	U
Always	Always	Always

C.664.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.664.5. IDL Operation

C.664.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ    => vs2_val[i] == rs1_val,
        VXCMP_VMSNE    => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU   => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT    => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU   => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE    => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU   => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT    => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.664.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

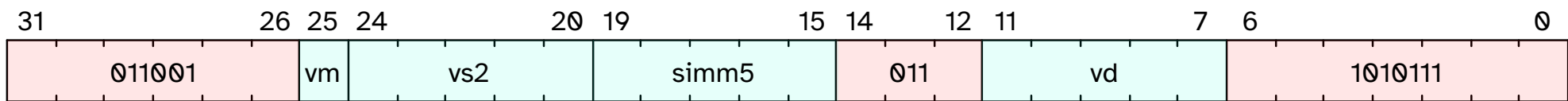
C.665. vmsne.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.665.1. Encoding



C.665.2. Description

No description available.

C.665.3. Access

M	S	U
Always	Always	Always

C.665.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.665.5. IDL Operation

C.665.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)      = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VICMP_VMSEQ => vs2_val[i] == imm_val,
        VICMP_VMSNE => vs2_val[i] != imm_val,
        VICMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(imm_val),
        VICMP_VMSLE  => signed(vs2_val[i]) <= signed(imm_val),
        VICMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(imm_val),
        VICMP_VMSGT  => signed(vs2_val[i]) > signed(imm_val)
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.665.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

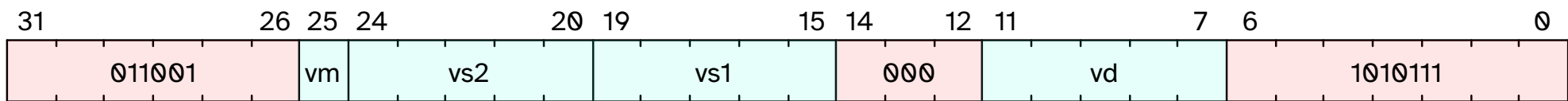
C.666. vmsne.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.666.1. Encoding



C.666.2. Description

No description available.

C.666.3. Access

M	S	U
Always	Always	Always

C.666.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.666.5. IDL Operation

C.666.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool)    = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VVCMP_VMSEQ => vs2_val[i] == vs1_val[i],
        VVCMP_VMSNE => vs2_val[i] != vs1_val[i],
        VVCMP_VMSLTU => unsigned(vs2_val[i]) < unsigned(vs1_val[i]),
        VVCMP_VMSLT  => signed(vs2_val[i]) < signed(vs1_val[i]),
        VVCMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(vs1_val[i]),
        VVCMP_VMSLE  => signed(vs2_val[i]) <= signed(vs1_val[i])
      };
      result[i] = res
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.666.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

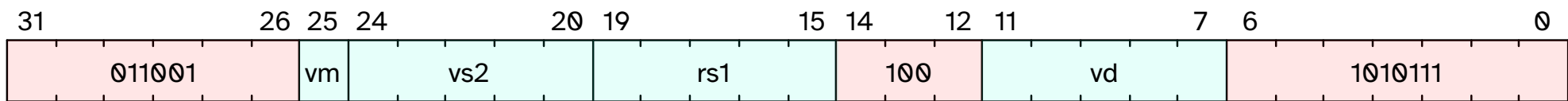
C.667. vmsne.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.667.1. Encoding



C.667.2. Description

No description available.

C.667.3. Access

M	S	U
Always	Always	Always

C.667.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.667.5. IDL Operation

C.667.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b0, vd);
  let result  : vector('n, dec, bool)      = undefined;
  let mask    : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let res : bool = match funct6 {
        VXCMP_VMSEQ => vs2_val[i] == rs1_val,
        VXCMP_VMSNE => vs2_val[i] != rs1_val,
        VXCMP_VMSLTU => unsigned(vs2_val[i]) < unsigned(rs1_val),
        VXCMP_VMSLT  => signed(vs2_val[i]) < signed(rs1_val),
        VXCMP_VMSLEU => unsigned(vs2_val[i]) <= unsigned(rs1_val),
        VXCMP_VMSLE  => signed(vs2_val[i]) <= signed(rs1_val),
        VXCMP_VMSGTU => unsigned(vs2_val[i]) > unsigned(rs1_val),
        VXCMP_VMSGT  => signed(vs2_val[i]) > signed(rs1_val)
      };
      result[i] = res
    }
  }
}
```

```
    }  
};  
  
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.667.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

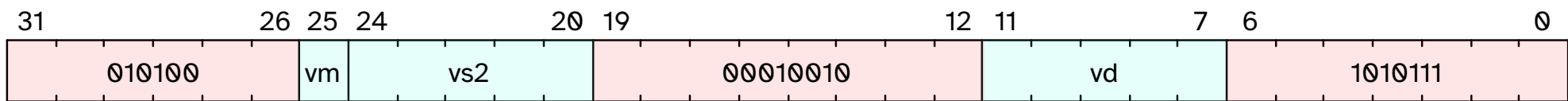
C.668. vmsof.m

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.668.1. Encoding



C.668.2. Description

No description available.

C.668.3. Access

M	S	U
Always	Always	Always

C.668.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.668.5. IDL Operation

C.668.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_normal(vd, vm) | not(assert_vstart(0)) | vd == vs2
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_cmp(num_elem, SEW, 0, vd_val, vm_val);

  found_elem : bool = false;
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      if vs2_val[i] & not(found_elem) then {
        result[i] = true;
        found_elem = true
      } else {
        result[i] = false
      }
    }
  }
};

write_vmask(num_elem, vd, result);
vstart = zeros();
```

```
RETIRE_SUCCESS
```

```
}
```

C.668.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

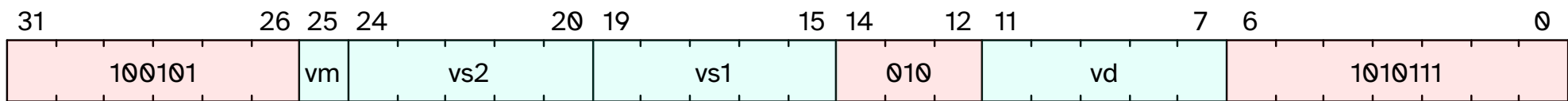
C.669. vmul.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.669.1. Encoding



C.669.2. Description

No description available.

C.669.3. Access

M	S	U
Always	Always	Always

C.669.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.669.5. IDL Operation

C.669.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.669.7. Exceptions

This instruction does not generate synchronous exceptions.

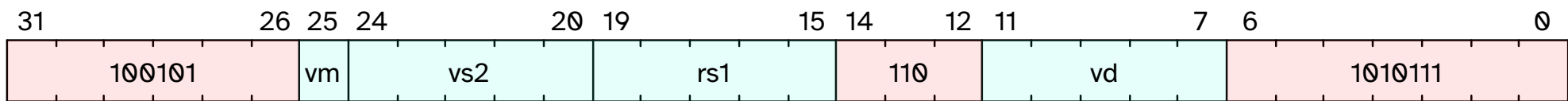
C.670. vmul.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.670.1. Encoding



C.670.2. Description

No description available.

C.670.3. Access

M	S	U
Always	Always	Always

C.670.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.670.5. IDL Operation

C.670.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU      => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD       => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.670.7. Exceptions

This instruction does not generate synchronous exceptions.

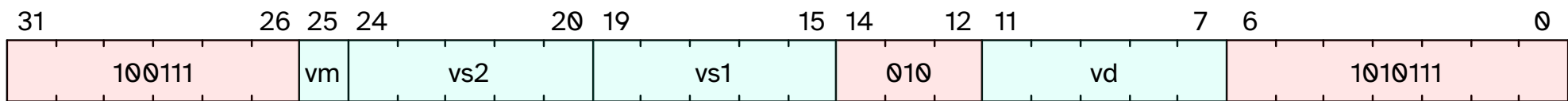
C.671. vmulh.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.671.1. Encoding



C.671.2. Description

No description available.

C.671.3. Access

M	S	U
Always	Always	Always

C.671.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.671.5. IDL Operation

C.671.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));
        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.671.7. Exceptions

This instruction does not generate synchronous exceptions.

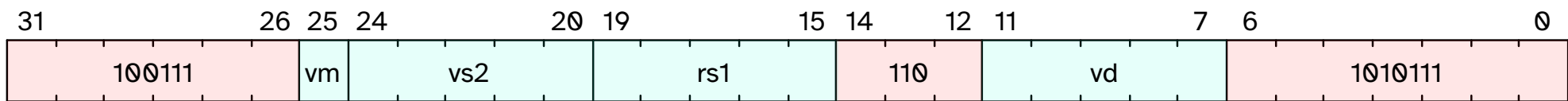
C.672. vmulh.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.672.1. Encoding



C.672.2. Description

No description available.

C.672.3. Access

M	S	U
Always	Always	Always

C.672.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.672.5. IDL Operation

C.672.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.672.7. Exceptions

This instruction does not generate synchronous exceptions.

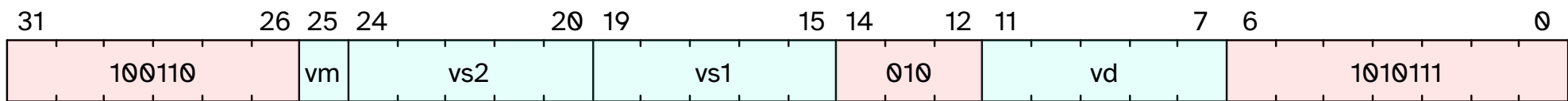
C.673. vmulhsu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.673.1. Encoding



C.673.2. Description

No description available.

C.673.3. Access

M	S	U
Always	Always	Always

C.673.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.673.5. IDL Operation

C.673.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));
        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.673.7. Exceptions

This instruction does not generate synchronous exceptions.

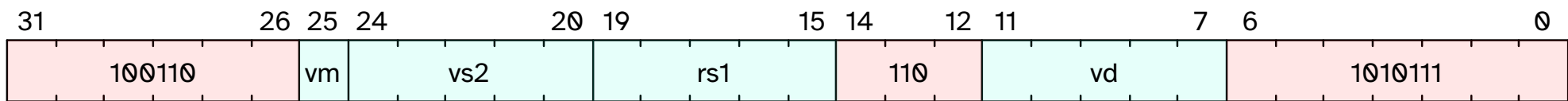
C.674. vmulhsu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.674.1. Encoding



C.674.2. Description

No description available.

C.674.3. Access

M	S	U
Always	Always	Always

C.674.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.674.5. IDL Operation

C.674.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU     => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU     => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM      => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.674.7. Exceptions

This instruction does not generate synchronous exceptions.

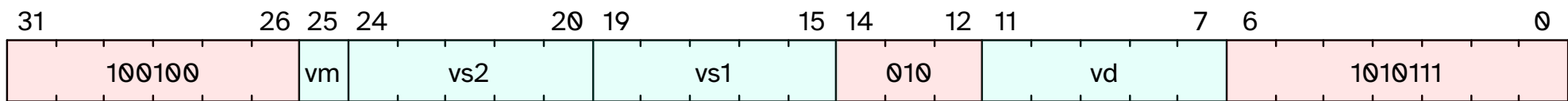
C.675. vmulhu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.675.1. Encoding



C.675.2. Description

No description available.

C.675.3. Access

M	S	U
Always	Always	Always

C.675.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.675.5. IDL Operation

C.675.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.675.7. Exceptions

This instruction does not generate synchronous exceptions.

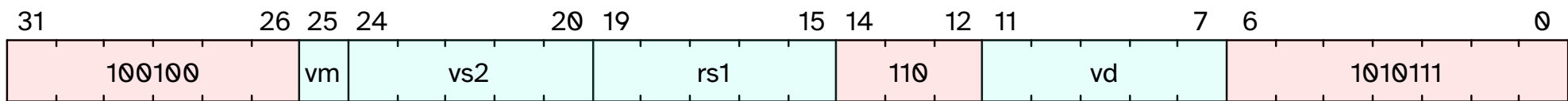
C.676. vmulhu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.676.1. Encoding



C.676.2. Description

No description available.

C.676.3. Access

M	S	U
Always	Always	Always

C.676.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.676.5. IDL Operation

C.676.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU      => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD        => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU     => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU     => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM      => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.676.7. Exceptions

This instruction does not generate synchronous exceptions.

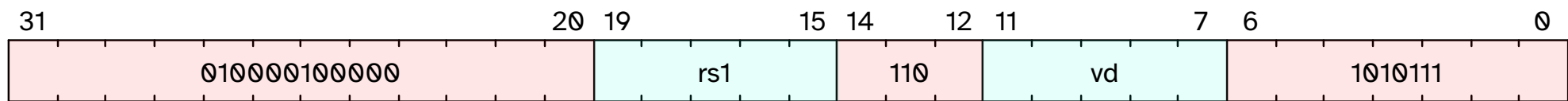
C.677. vmv.s.x

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.677.1. Encoding



C.677.2. Description

No description available.

C.677.3. Access

M	S	U
Always	Always	Always

C.677.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.677.5. IDL Operation

C.677.6. Sail Operation

```
{
  let SEW      = get_sew();
  let num_elem = get_num_elem(0, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  assert(num_elem > 0);
  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, 0b1, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, 'm);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, 0, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, 0, vd_val, vm_val);

  /* one body element */
  if mask[0] then result[0] = rs1_val;

  /* others treated as tail elements */
  let tail_ag : agtype = get_vtype_vta();
  foreach (i from 1 to (num_elem - 1)) {
    result[i] = match tail_ag {
      UNDISTURBED => vd_val[i],
      AGNOSTIC    => vd_val[i] /* TODO: configuration support */
    }
  };

  write_vreg(num_elem, SEW, 0, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.677.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

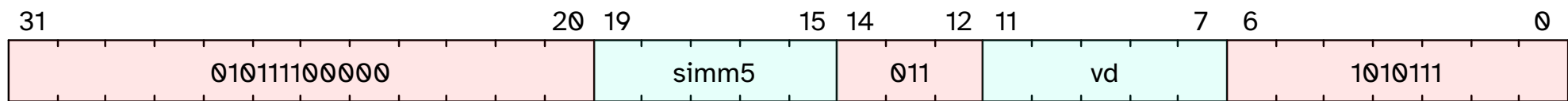
C.678. vmv.v.i

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.678.1. Encoding



C.678.2. Description

No description available.

C.678.3. Access

M	S	U
Always	Always	Always

C.678.4. Decode Variables

```
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.678.5. IDL Operation

C.678.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, 0b1, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then result[i] = imm_val
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.678.7. Exceptions

This instruction does not generate synchronous exceptions.

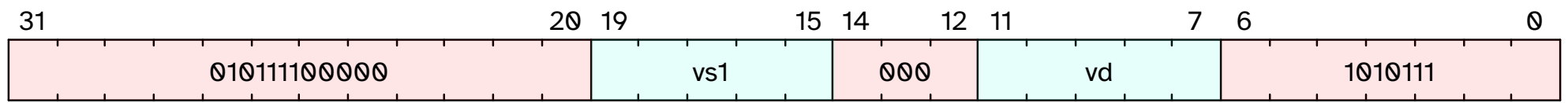
C.679. vmv.v.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.679.1. Encoding



C.679.2. Description

No description available.

C.679.3. Access

M	S	U
Always	Always	Always

C.679.4. Decode Variables

```
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.679.5. IDL Operation

C.679.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b1, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then result[i] = vs1_val[i]
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.679.7. Exceptions

This instruction does not generate synchronous exceptions.

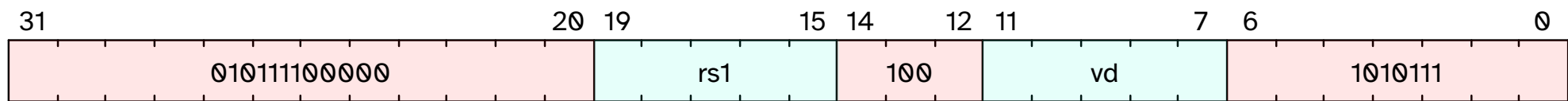
C.680. vmv.v.x

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.680.1. Encoding



C.680.2. Description

No description available.

C.680.3. Access

M	S	U
Always	Always	Always

C.680.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.680.5. IDL Operation

C.680.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let rs1_val : bits('m)          = get_scalar(rs1, 'm);
  let vm_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b1, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then result[i] = rs1_val
  };

  write_vreg(num_elem, SEW, LMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.680.7. Exceptions

This instruction does not generate synchronous exceptions.

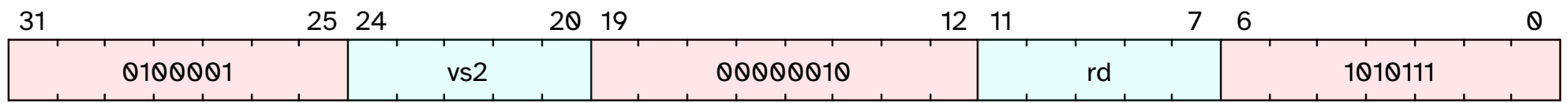
C.681. vmv.x.s

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.681.1. Encoding



C.681.2. Description

No description available.

C.681.3. Access

M	S	U
Always	Always	Always

C.681.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rd = $encoding[11:7];
```

C.681.5. IDL Operation

C.681.6. Sail Operation

```
{
  let SEW      = get_sew();
  let num_elem = get_num_elem(0, SEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  assert(num_elem > 0);
  let 'n = num_elem;
  let 'm = SEW;

  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, 0, vs2);
  X(rd) = if sizeof(xlen) < SEW then slice(vs2_val[0], 0, sizeof(xlen))
          else if sizeof(xlen) > SEW then sign_extend(vs2_val[0])
          else vs2_val[0];
  vstart = zeros();

  RETIRE_SUCCESS
}
```

C.681.7. Exceptions

This instruction does not generate synchronous exceptions.

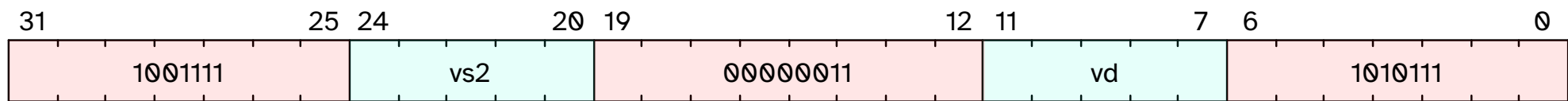
C.682. vmv1r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.682.1. Encoding



C.682.2. Description

No description available.

C.682.3. Access

M	S	U
Always	Always	Always

C.682.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.682.5. IDL Operation

C.682.6. Sail Operation

```
{
  let start_element = get_start_element();
  let SEW      = get_sew();
  let imm_val = unsigned(zero_extend(sizeof(xlen), simm));
  let EMUL    = imm_val + 1;

  if not(EMUL == 1 | EMUL == 2 | EMUL == 4 | EMUL == 8) then { handle_illegal(); return RETIRE_FAIL };

  let EMUL_pow = log2(EMUL);
  let num_elem = get_num_elem(EMUL_pow, SEW);
  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b1, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  foreach (i from 0 to (num_elem - 1)) {
    result[i] = if i < start_element then vd_val[i] else vs2_val[i]
  };

  write_vreg(num_elem, SEW, EMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.682.7. Exceptions

This instruction does not generate synchronous exceptions.

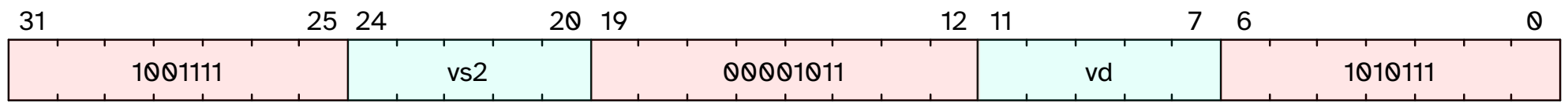
C.683. vmv2r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.683.1. Encoding



C.683.2. Description

No description available.

C.683.3. Access

M	S	U
Always	Always	Always

C.683.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.683.5. IDL Operation

C.683.6. Sail Operation

```
{
  let start_element = get_start_element();
  let SEW      = get_sew();
  let imm_val = unsigned(zero_extend(sizeof(xlen), simm));
  let EMUL    = imm_val + 1;

  if not(EMUL == 1 | EMUL == 2 | EMUL == 4 | EMUL == 8) then { handle_illegal(); return RETIRE_FAIL };

  let EMUL_pow = log2(EMUL);
  let num_elem = get_num_elem(EMUL_pow, SEW);
  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b1, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  foreach (i from 0 to (num_elem - 1)) {
    result[i] = if i < start_element then vd_val[i] else vs2_val[i]
  };

  write_vreg(num_elem, SEW, EMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.683.7. Exceptions

This instruction does not generate synchronous exceptions.

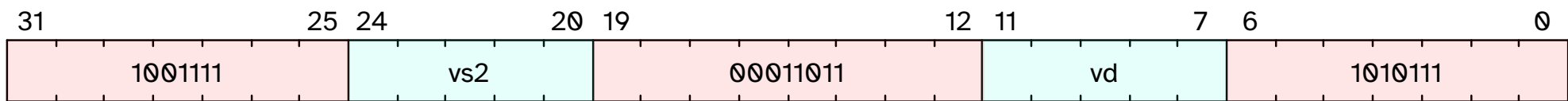
C.684. vmv4r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.684.1. Encoding



C.684.2. Description

No description available.

C.684.3. Access

M	S	U
Always	Always	Always

C.684.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.684.5. IDL Operation

C.684.6. Sail Operation

```
{
  let start_element = get_start_element();
  let SEW      = get_sew();
  let imm_val = unsigned(zero_extend(sizeof(xlen), simm));
  let EMUL     = imm_val + 1;

  if not(EMUL == 1 | EMUL == 2 | EMUL == 4 | EMUL == 8) then { handle_illegal(); return RETIRE_FAIL };

  let EMUL_pow = log2(EMUL);
  let num_elem = get_num_elem(EMUL_pow, SEW);
  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b1, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  foreach (i from 0 to (num_elem - 1)) {
    result[i] = if i < start_element then vd_val[i] else vs2_val[i]
  };

  write_vreg(num_elem, SEW, EMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.684.7. Exceptions

This instruction does not generate synchronous exceptions.

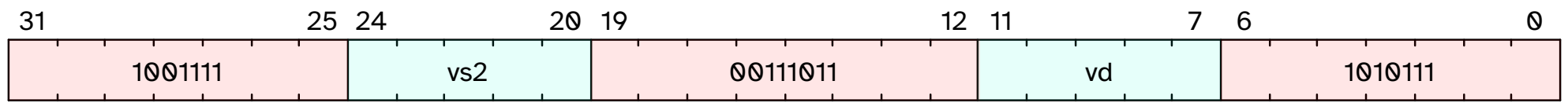
C.685. vmv8r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.685.1. Encoding



C.685.2. Description

No description available.

C.685.3. Access

M	S	U
Always	Always	Always

C.685.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.685.5. IDL Operation

C.685.6. Sail Operation

```
{
  let start_element = get_start_element();
  let SEW      = get_sew();
  let imm_val = unsigned(zero_extend(sizeof(xlen), simm));
  let EMUL    = imm_val + 1;

  if not(EMUL == 1 | EMUL == 2 | EMUL == 4 | EMUL == 8) then { handle_illegal(); return RETIRE_FAIL };

  let EMUL_pow = log2(EMUL);
  let num_elem = get_num_elem(EMUL_pow, SEW);
  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, 0b1, 0b000000);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, EMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;

  foreach (i from 0 to (num_elem - 1)) {
    result[i] = if i < start_element then vd_val[i] else vs2_val[i]
  };

  write_vreg(num_elem, SEW, EMUL_pow, vd, result);
  vstart = zeros();
  RETIRE_SUCCESS
}
```

C.685.7. Exceptions

This instruction does not generate synchronous exceptions.

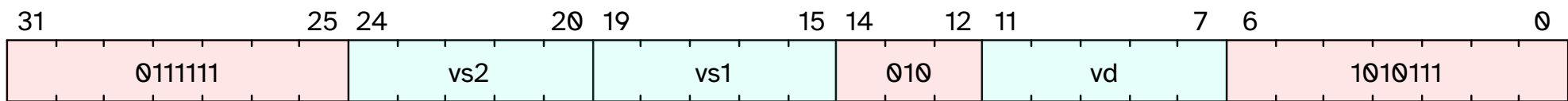
C.686. vmxnor.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.686.1. Encoding



C.686.2. Description

No description available.

C.686.3. Access

M	S	U
Always	Always	Always

C.686.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.686.5. IDL Operation

C.686.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, 0, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MM_VMAND      => vs2_val[i] & vs1_val[i],
        MM_VMNAND     => not(vs2_val[i] & vs1_val[i]),
        MM_VMANDNOT   => vs2_val[i] & not(vs1_val[i]),
        MM_VMXOR      => vs2_val[i] != vs1_val[i],
        MM_VMOR       => vs2_val[i] | vs1_val[i],
        MM_VMNOR      => not(vs2_val[i] | vs1_val[i]),
        MM_VMORNOT    => vs2_val[i] | not(vs1_val[i]),
        MM_VMXNOR     => vs2_val[i] == vs1_val[i]
      }
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.686.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

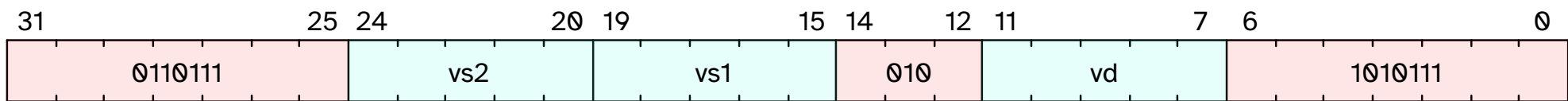
C.687. vmxor.mm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.687.1. Encoding



C.687.2. Description

No description available.

C.687.3. Access

M	S	U
Always	Always	Always

C.687.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.687.5. IDL Operation

C.687.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = unsigned(vlenb) * 8;

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vs1_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs1);
  let vs2_val : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vs2);
  let vd_val  : vector('n, dec, bool) = read_vmask(num_elem, 0b0, vd);
  result      : vector('n, dec, bool) = undefined;
  mask        : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result_carry(num_elem, SEW, 0, vd_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MM_VMAND      => vs2_val[i] & vs1_val[i],
        MM_VMNAND     => not(vs2_val[i] & vs1_val[i]),
        MM_VMANDNOT   => vs2_val[i] & not(vs1_val[i]),
        MM_VMXOR      => vs2_val[i] != vs1_val[i],
        MM_VMOR       => vs2_val[i] | vs1_val[i],
        MM_VMNOR      => not(vs2_val[i] | vs1_val[i]),
        MM_VMORNOT    => vs2_val[i] | not(vs1_val[i]),
        MM_VMXNOR     => vs2_val[i] == vs1_val[i]
      }
    }
  }
};
```

```
write_vmask(num_elem, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.687.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

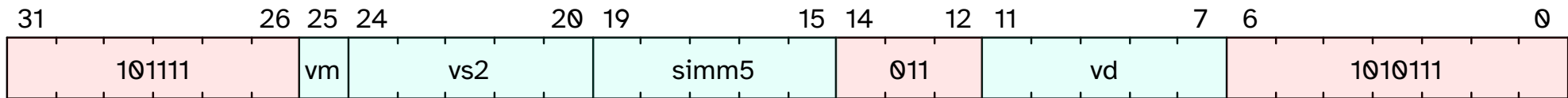
C.688. vnclip.wi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.688.1. Encoding



C.688.2. Description

No description available.

C.688.3. Access

M	S	U
Always	Always	Always

C.688.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.688.5. IDL Operation

C.688.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let shift_amount = get_shift_amount(imm_val, SEW_widen);
      let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
      result[i] = match funct6 {
        NI_VNCLIPU => {
          let result_wide = (vs2_val[i] >> shift_amount) + zero_extend('o, rounding_incr);
```

```

        unsigned_saturation('m, result_wide)
    },
    NI_VNCLIP => {
        let v_double : bits('m * 4) = sign_extend(vs2_val[i]);
        let result_wide = slice(v_double >> shift_amount, 0, 'o) + zero_extend('o, rounding_incr);
        signed_saturation('m, result_wide)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.688.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

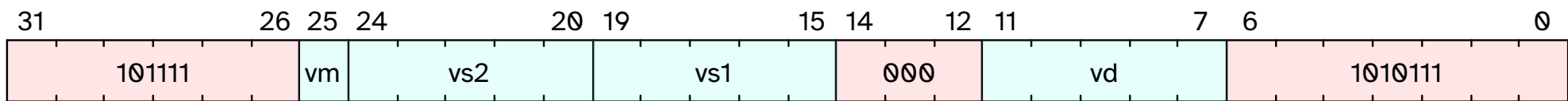
C.689. vnclip.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.689.1. Encoding



C.689.2. Description

No description available.

C.689.3. Access

M	S	U
Always	Always	Always

C.689.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.689.5. IDL Operation

C.689.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('m)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let shift_amount = get_shift_amount(vs1_val[i], SEW_widen);
      let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
      result[i] = match funct6 {
        NV_VNCLIPU => {
          let result_wide = (vs2_val[i] >> shift_amount) + zero_extend('o, rounding_incr);
```

```

        unsigned_saturation('m, result_wide);
    },
    NV_VNCLIP => {
        let v_double : bits('m * 4) = sign_extend(vs2_val[i]);
        let result_wide = slice(v_double >> shift_amount, 0, 'o) + zero_extend('o, rounding_incr);
        signed_saturation('m, result_wide);
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.689.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

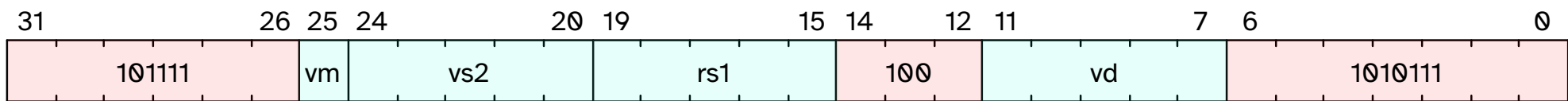
C.690. vnclip.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.690.1. Encoding



C.690.2. Description

No description available.

C.690.3. Access

M	S	U
Always	Always	Always

C.690.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.690.5. IDL Operation

C.690.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('m)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let shift_amount = get_shift_amount(rs1_val, SEW_widen);
      let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
      result[i] = match funct6 {
        NX_VNCLIPU => {
          let result_wide = (vs2_val[i] >> shift_amount) + zero_extend('o, rounding_incr);
```

```

        unsigned_saturation('m, result_wide)
    },
    NX_VNCLIP => {
        let v_double : bits('m * 4) = sign_extend(vs2_val[i]);
        let result_wide = slice(v_double >> shift_amount, 0, 'o) + zero_extend('o, rounding_incr);
        signed_saturation('m, result_wide)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.690.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

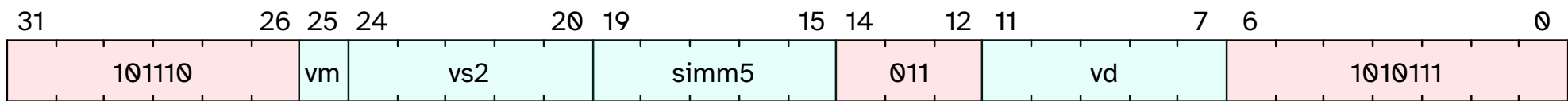
C.691. vnclipu.wi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.691.1. Encoding



C.691.2. Description

No description available.

C.691.3. Access

M	S	U
Always	Always	Always

C.691.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.691.5. IDL Operation

C.691.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('m)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let shift_amount = get_shift_amount(imm_val, SEW_widen);
      let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
      result[i] = match funct6 {
        NI_VNCLIPU => {
          let result_wide = (vs2_val[i] >> shift_amount) + zero_extend('o, rounding_incr);
```

```

        unsigned_saturation('m, result_wide)
    },
    NI_VNCLIP => {
        let v_double : bits('m * 4) = sign_extend(vs2_val[i]);
        let result_wide = slice(v_double >> shift_amount, 0, 'o) + zero_extend('o, rounding_incr);
        signed_saturation('m, result_wide)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.691.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

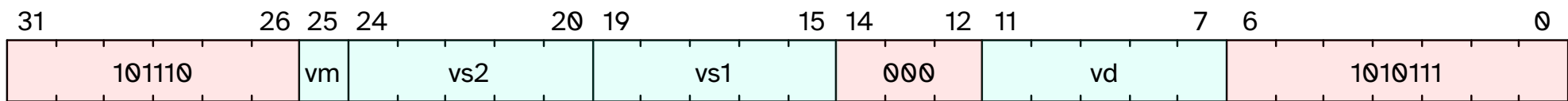
C.692. vnclipu.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.692.1. Encoding



C.692.2. Description

No description available.

C.692.3. Access

M	S	U
Always	Always	Always

C.692.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.692.5. IDL Operation

C.692.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('m)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let shift_amount = get_shift_amount(vs1_val[i], SEW_widen);
      let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
      result[i] = match funct6 {
        NV_VNCLIPU => {
          let result_wide = (vs2_val[i] >> shift_amount) + zero_extend('o, rounding_incr);
```

```

        unsigned_saturation('m, result_wide);
    },
    NV_VNCLIP => {
        let v_double : bits('m * 4) = sign_extend(vs2_val[i]);
        let result_wide = slice(v_double >> shift_amount, 0, 'o) + zero_extend('o, rounding_incr);
        signed_saturation('m, result_wide);
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.692.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

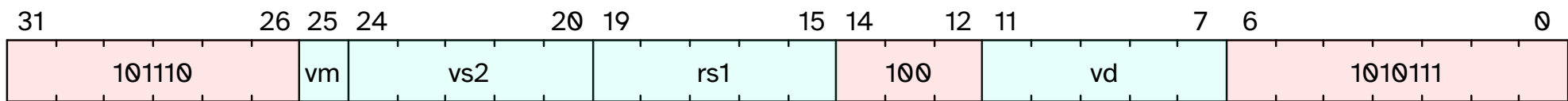
C.693. vnclipu.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.693.1. Encoding



C.693.2. Description

No description available.

C.693.3. Access

M	S	U
Always	Always	Always

C.693.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.693.5. IDL Operation

C.693.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      let shift_amount = get_shift_amount(rs1_val, SEW_widen);
      let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
      result[i] = match funct6 {
        NX_VNCLIPU => {
          let result_wide = (vs2_val[i] >> shift_amount) + zero_extend('o, rounding_incr);
```

```

        unsigned_saturation('m, result_wide)
    },
    NX_VNCLIP => {
        let v_double : bits('m * 4) = sign_extend(vs2_val[i]);
        let result_wide = slice(v_double >> shift_amount, 0, 'o) + zero_extend('o, rounding_incr);
        signed_saturation('m, result_wide)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.693.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

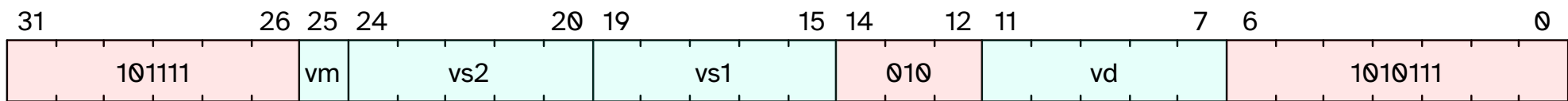
C.694. vnmsac.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.694.1. Encoding



C.694.2. Description

No description available.

C.694.3. Access

M	S	U
Always	Always	Always

C.694.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.694.5. IDL Operation

C.694.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VMACC => get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0) + vd_val[i],
        MVV_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0),
        MVV_VMADD  => get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0) + vs2_val[i],
        MVV_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.694.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

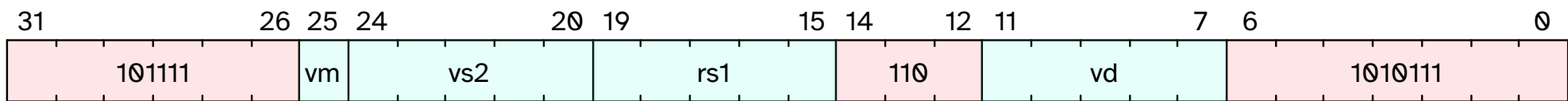
C.695. vnmsac.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.695.1. Encoding



C.695.2. Description

No description available.

C.695.3. Access

M	S	U
Always	Always	Always

C.695.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.695.5. IDL Operation

C.695.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VMACC => get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0) + vd_val[i],
        MVX_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0),
        MVX_VMADD  => get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0) + vs2_val[i],
        MVX_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

```
RETIRE_SUCCESS
```

```
}
```

C.695.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

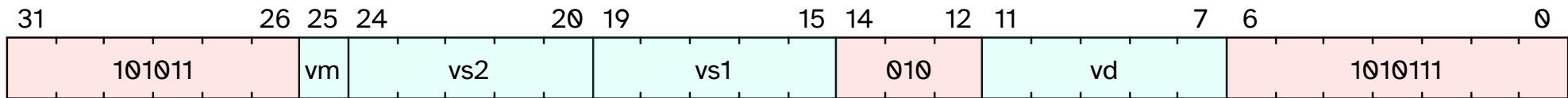
C.696. vnmsub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.696.1. Encoding



C.696.2. Description

No description available.

C.696.3. Access

M	S	U
Always	Always	Always

C.696.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.696.5. IDL Operation

C.696.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VMACC => get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0) + vd_val[i],
        MVV_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vs2_val[i]), 0),
        MVV_VMADD  => get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0) + vs2_val[i],
        MVV_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(vs1_val[i]) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.696.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

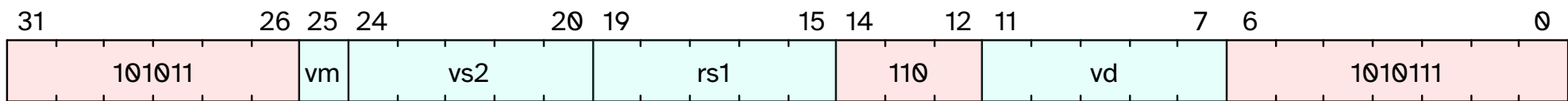
C.697. vnmsub.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.697.1. Encoding



C.697.2. Description

No description available.

C.697.3. Access

M	S	U
Always	Always	Always

C.697.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.697.5. IDL Operation

C.697.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VMACC => get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0) + vd_val[i],
        MVX_VNMSAC => vd_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vs2_val[i]), 0),
        MVX_VMADD  => get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0) + vs2_val[i],
        MVX_VNMSUB => vs2_val[i] - get_slice_int(SEW, signed(rs1_val) * signed(vd_val[i]), 0)
      }
    }
  }
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
```

RETIRE_SUCCESS

}

C.697.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

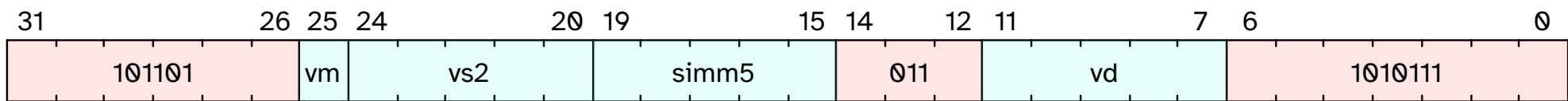
C.698. vnsra.wi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.698.1. Encoding



C.698.2. Description

No description available.

C.698.3. Access

M	S	U
Always	Always	Always

C.698.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.698.5. IDL Operation

C.698.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        NIS_VNSRL => {
          let shift_amount = get_shift_amount(imm_val, SEW_widen);
          slice(vs2_val[i] >> shift_amount, 0, SEW)
        },

```

```

    NIS_VNSRA => {
        let shift_amount = get_shift_amount(imm_val, SEW_widen);
        let v_double : bits('o * 2) = sign_extend(vs2_val[i]);
        let arith_shifted : bits('o) = slice(v_double >> shift_amount, 0, SEW_widen);
        slice(arith_shifted, 0, SEW)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.698.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

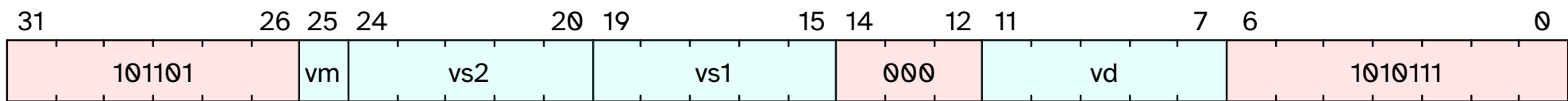
C.699. vnsra.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.699.1. Encoding



C.699.2. Description

No description available.

C.699.3. Access

M	S	U
Always	Always	Always

C.699.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.699.5. IDL Operation

C.699.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        NVS_VNSRL => {
          let shift_amount = get_shift_amount(vs1_val[i], SEW_widen);
          slice(vs2_val[i] >> shift_amount, 0, SEW)
        },
      }
    }
  }
}
```

```

    NVS_VNSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW_widen);
        let v_double : bits('o * 2) = sign_extend(vs2_val[i]);
        let arith_shifted : bits('o) = slice(v_double >> shift_amount, 0, SEW_widen);
        slice(arith_shifted, 0, SEW)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.699.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

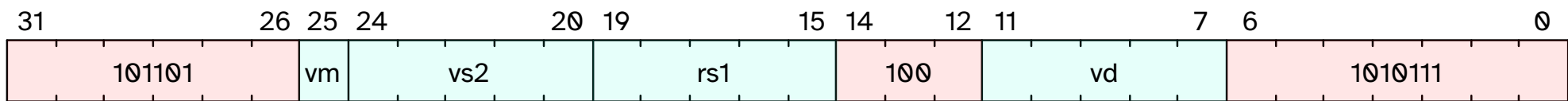
C.700. vnsra.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.700.1. Encoding



C.700.2. Description

No description available.

C.700.3. Access

M	S	U
Always	Always	Always

C.700.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.700.5. IDL Operation

C.700.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        NXS_VNSRL => {
          let shift_amount = get_shift_amount(rs1_val, SEW_widen);
          slice(vs2_val[i] >> shift_amount, 0, SEW)
        },
      },
    }
  }
}
```

```

    NXS_VNSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW_widen);
        let v_double : bits('o * 2) = sign_extend(vs2_val[i]);
        let arith_shifted : bits('o) = slice(v_double >> shift_amount, 0, SEW_widen);
        slice(arith_shifted, 0, SEW)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.700.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

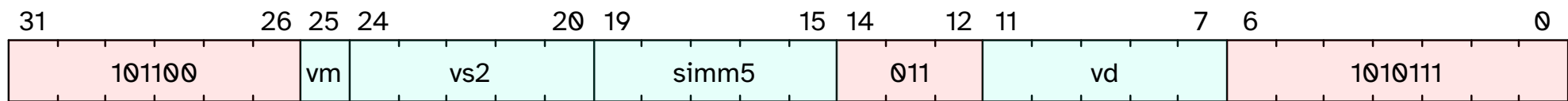
C.701. vnsrl.wi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.701.1. Encoding



C.701.2. Description

No description available.

C.701.3. Access

M	S	U
Always	Always	Always

C.701.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.701.5. IDL Operation

C.701.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        NIS_VNSRL => {
          let shift_amount = get_shift_amount(imm_val, SEW_widen);
          slice(vs2_val[i] >> shift_amount, 0, SEW)
        },
      },
    }
  }
}
```

```

    NIS_VNSRA => {
        let shift_amount = get_shift_amount(imm_val, SEW_widen);
        let v_double : bits('o * 2) = sign_extend(vs2_val[i]);
        let arith_shifted : bits('o) = slice(v_double >> shift_amount, 0, SEW_widen);
        slice(arith_shifted, 0, SEW)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.701.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

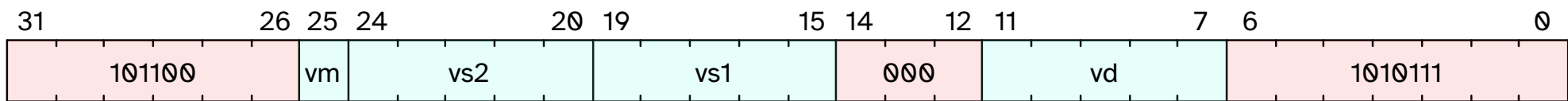
C.702. vnsrl.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.702.1. Encoding



C.702.2. Description

No description available.

C.702.3. Access

M	S	U
Always	Always	Always

C.702.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.702.5. IDL Operation

C.702.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        NVS_VNSRL => {
          let shift_amount = get_shift_amount(vs1_val[i], SEW_widen);
          slice(vs2_val[i] >> shift_amount, 0, SEW)
        },
      };
    }
  }
}
```

```

    NVS_VNSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW_widen);
        let v_double : bits('o * 2) = sign_extend(vs2_val[i]);
        let arith_shifted : bits('o) = slice(v_double >> shift_amount, 0, SEW_widen);
        slice(arith_shifted, 0, SEW)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.702.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

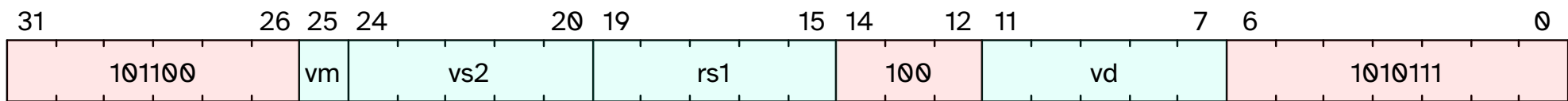
C.703. vnsrl.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.703.1. Encoding



C.703.2. Description

No description available.

C.703.3. Access

M	S	U
Always	Always	Always

C.703.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.703.5. IDL Operation

C.703.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_widen, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW_widen <= 64);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        NXS_VNSRL => {
          let shift_amount = get_shift_amount(rs1_val, SEW_widen);
          slice(vs2_val[i] >> shift_amount, 0, SEW)
        },
      },
    }
  }
}
```

```

    NXS_VNSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW_widen);
        let v_double : bits('o * 2) = sign_extend(vs2_val[i]);
        let arith_shifted : bits('o) = slice(v_double >> shift_amount, 0, SEW_widen);
        slice(arith_shifted, 0, SEW)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.703.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

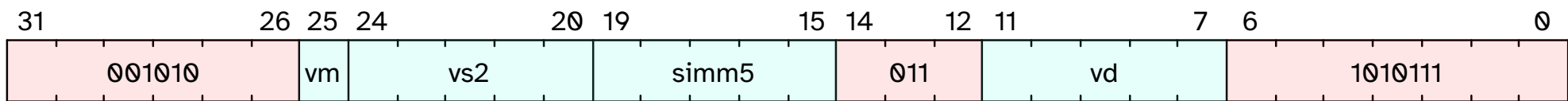
C.704. vor.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.704.1. Encoding



C.704.2. Description

No description available.

C.704.3. Access

M	S	U
Always	Always	Always

C.704.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.704.5. IDL Operation

C.704.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD  => vs2_val[i] + imm_val,
        VI_VRSUB => imm_val - vs2_val[i],
        VI_VAND  => vs2_val[i] & imm_val,
        VI_VOR   => vs2_val[i] | imm_val,
        VI_VXOR  => vs2_val[i] ^ imm_val,
        VI_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL   => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.704.7. Exceptions

This instruction does not generate synchronous exceptions.

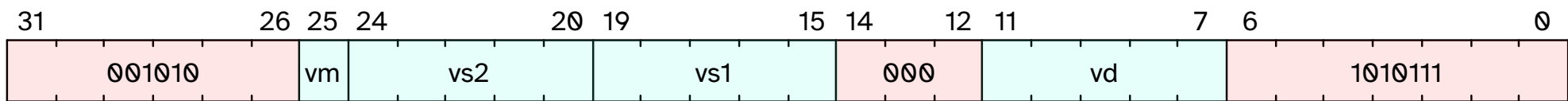
C.705. vor.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.705.1. Encoding



C.705.2. Description

No description available.

C.705.3. Access

M	S	U
Always	Always	Always

C.705.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.705.5. IDL Operation

C.705.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.705.7. Exceptions

This instruction does not generate synchronous exceptions.

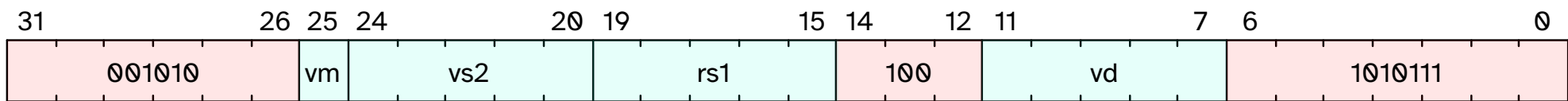
C.706. vor.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.706.1. Encoding



C.706.2. Description

No description available.

C.706.3. Access

M	S	U
Always	Always	Always

C.706.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.706.5. IDL Operation

C.706.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.706.7. Exceptions

This instruction does not generate synchronous exceptions.

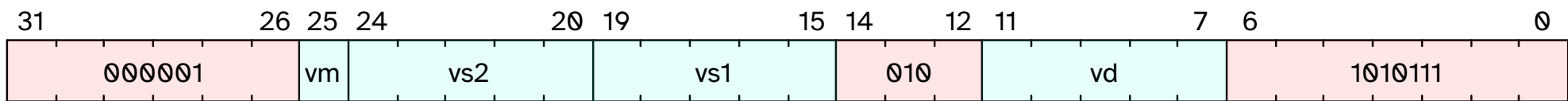
C.707. vredand.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.707.1. Encoding



C.707.2. Description

No description available.

C.707.3. Access

M	S	U
Always	Always	Always

C.707.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.707.5. IDL Operation

C.707.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.707.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

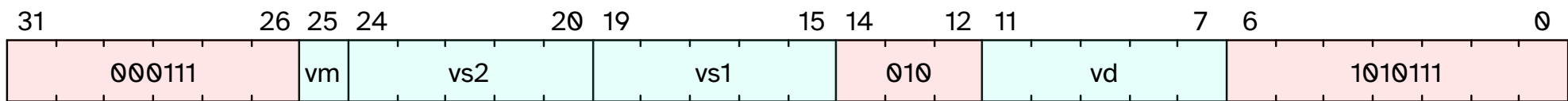
C.708. vredmax.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.708.1. Encoding



C.708.2. Description

No description available.

C.708.3. Access

M	S	U
Always	Always	Always

C.708.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.708.5. IDL Operation

C.708.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.708.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

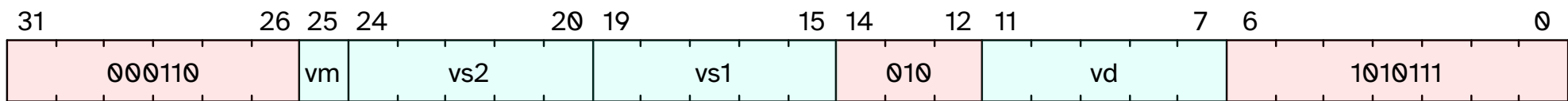
C.709. vredmaxu.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.709.1. Encoding



C.709.2. Description

No description available.

C.709.3. Access

M	S	U
Always	Always	Always

C.709.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.709.5. IDL Operation

C.709.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.709.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

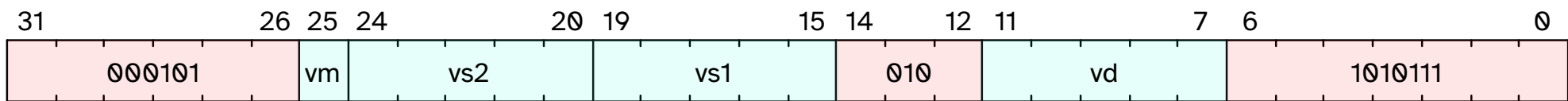
C.710. vredmin.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.710.1. Encoding



C.710.2. Description

No description available.

C.710.3. Access

M	S	U
Always	Always	Always

C.710.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.710.5. IDL Operation

C.710.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.710.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

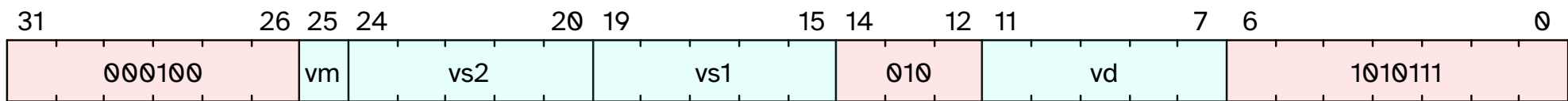
C.711. vredminu.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.711.1. Encoding



C.711.2. Description

No description available.

C.711.3. Access

M	S	U
Always	Always	Always

C.711.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.711.5. IDL Operation

C.711.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.711.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

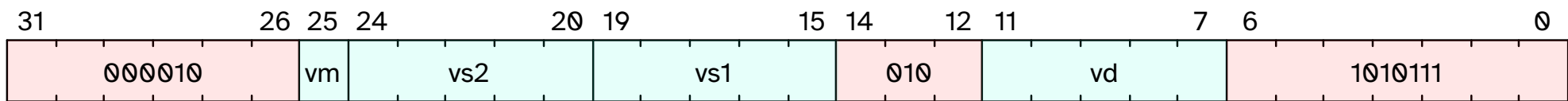
C.712. vredor.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.712.1. Encoding



C.712.2. Description

No description available.

C.712.3. Access

M	S	U
Always	Always	Always

C.712.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.712.5. IDL Operation

C.712.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.712.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

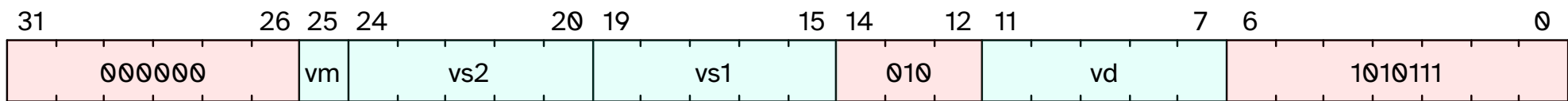
C.713. vredsum.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.713.1. Encoding



C.713.2. Description

No description available.

C.713.3. Access

M	S	U
Always	Always	Always

C.713.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.713.5. IDL Operation

C.713.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.713.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

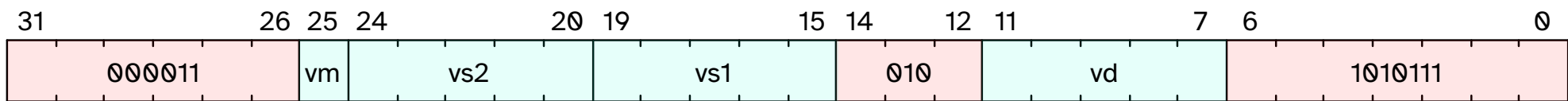
C.714. vredxor.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.714.1. Encoding



C.714.2. Description

No description available.

C.714.3. Access

M	S	U
Always	Always	Always

C.714.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.714.5. IDL Operation

C.714.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW); /* vd regardless of LMUL setting */

  if illegal_reduction() then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val  : vector('d, dec, bits('m)) = read_vreg(num_elem_vd, SEW, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)     = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('m) = read_single_element(SEW, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      sum = match funct6 {
        MVV_VREDSUM  => sum + vs2_val[i],
        MVV_VREDAND  => sum & vs2_val[i],
        MVV_VREDOR   => sum | vs2_val[i],
        MVV_VREDXOR  => sum ^ vs2_val[i],
        MVV_VREDMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(sum))),
        MVV_VREDMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(sum))),
        MVV_VREDMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(sum)))
      }
    }
  }
}
```

```
    }  
};  
  
write_single_element(SEW, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.714.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

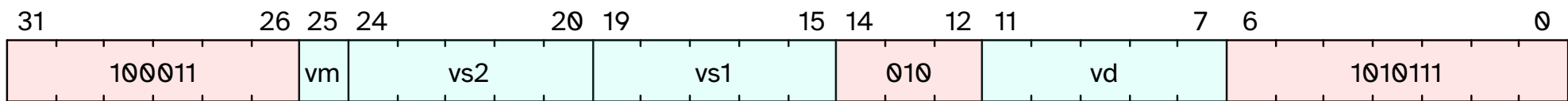
C.715. vrem.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.715.1. Encoding



C.715.2. Description

No description available.

C.715.3. Access

M	S	U
Always	Always	Always

C.715.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.715.5. IDL Operation

C.715.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));
        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.715.7. Exceptions

This instruction does not generate synchronous exceptions.

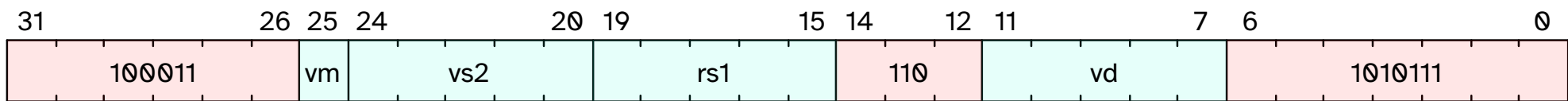
C.716. vrem.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.716.1. Encoding



C.716.2. Description

No description available.

C.716.3. Access

M	S	U
Always	Always	Always

C.716.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.716.5. IDL Operation

C.716.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU      => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD        => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.716.7. Exceptions

This instruction does not generate synchronous exceptions.

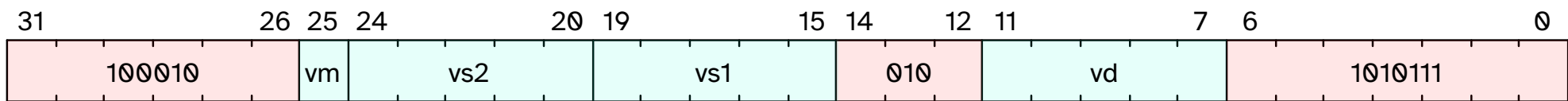
C.717. vremu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.717.1. Encoding



C.717.2. Description

No description available.

C.717.3. Access

M	S	U
Always	Always	Always

C.717.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.717.5. IDL Operation

C.717.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVV_VAADDU => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVV_VAADD  => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i]);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
```

```

    MVV_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm') + zero_extend('m, rounding_incr)
    },
    MVV_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i]);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm') + zero_extend('m, rounding_incr)
    },
    MVV_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), 0),
    MVV_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(vs1_val[i]), SEW),
    MVV_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(vs1_val[i]), SEW),
    MVV_VDIVU     => {
        let q : int = if unsigned(vs1_val[i]) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(vs1_val[i]));
        to_bits(SEW, q)
    },
    MVV_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(vs1_val[i]) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(vs1_val[i]));
        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVV_VREMU     => {
        let r : int = if unsigned(vs1_val[i]) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVV_VREM      => {
        let r : int = if signed(vs1_val[i]) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(vs1_val[i]));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.717.7. Exceptions

This instruction does not generate synchronous exceptions.

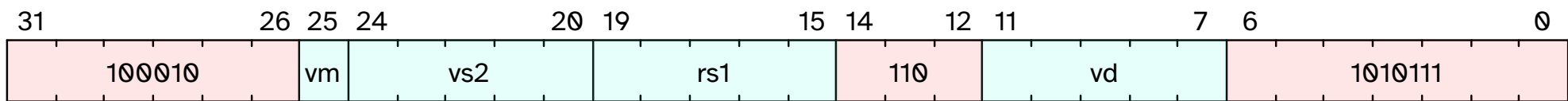
C.718. vremu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.718.1. Encoding



C.718.2. Description

No description available.

C.718.3. Access

M	S	U
Always	Always	Always

C.718.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.718.5. IDL Operation

C.718.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.718.7. Exceptions

This instruction does not generate synchronous exceptions.

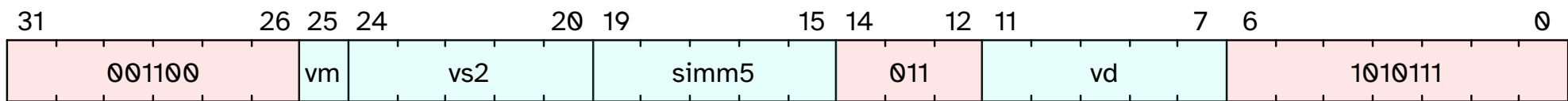
C.719. vrgather.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.719.1. Encoding



C.719.2. Description

No description available.

C.719.3. Access

M	S	U
Always	Always	Always

C.719.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.719.5. IDL Operation

C.719.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let imm_val : nat = unsigned(zero_extend(sizeof(xlen), simm));
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result : vector('n, dec, bits('m)) = undefined;
  mask : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VSLIDEUP => {
          if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
          if i >= imm_val then vs2_val[i - imm_val] else vd_val[i]
        },
        VI_VSLIDEDOWN => {
          let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
          assert(VLMAX > 0 & VLMAX <= 'n);
          if i + imm_val < VLMAX then vs2_val[i + imm_val] else zeros()
```

```

        },
    VI_VRGATHER => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX > 0 & VLMAX <= 'n');
        if imm_val < VLMAX then vs2_val[imm_val] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.719.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

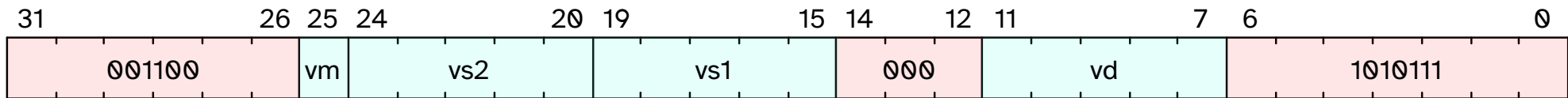
C.720. vrgather.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.720.1. Encoding



C.720.2. Description

No description available.

C.720.3. Access

M	S	U
Always	Always	Always

C.720.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.720.5. IDL Operation

C.720.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.720.7. Exceptions

This instruction does not generate synchronous exceptions.

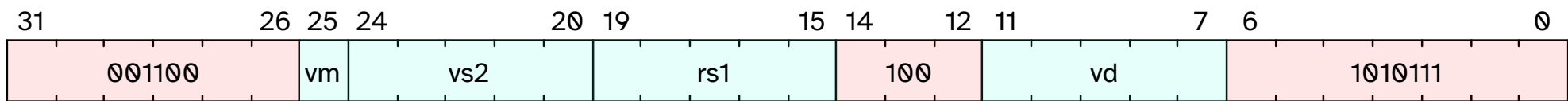
C.721. vrgather.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.721.1. Encoding



C.721.2. Description

No description available.

C.721.3. Access

M	S	U
Always	Always	Always

C.721.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.721.5. IDL Operation

C.721.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : nat                  = unsigned(X(rs1));
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VSLIDEUP => {
          if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
          if i >= rs1_val then vs2_val[i - rs1_val] else vd_val[i]
        },
        VX_VSLIDEDOWN => {
          let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
          assert(VLMAX > 0 & VLMAX <= 'n);
          if i + rs1_val < VLMAX then vs2_val[i + rs1_val] else zeros()
```

```

    },
    VX_VRGATHER => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX > 0 & VLMAX <= 'n');
        if rs1_val < VLMAX then vs2_val[rs1_val] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.721.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

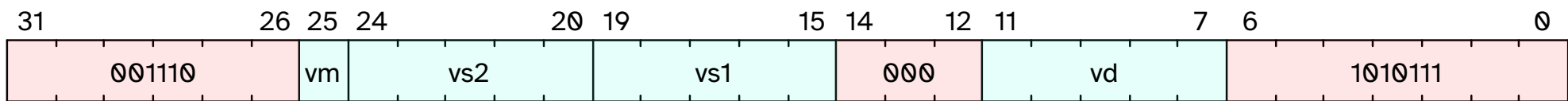
C.722. vrgatherei16.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.722.1. Encoding



C.722.2. Description

No description available.

C.722.3. Access

M	S	U
Always	Always	Always

C.722.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.722.5. IDL Operation

C.722.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.722.7. Exceptions

This instruction does not generate synchronous exceptions.

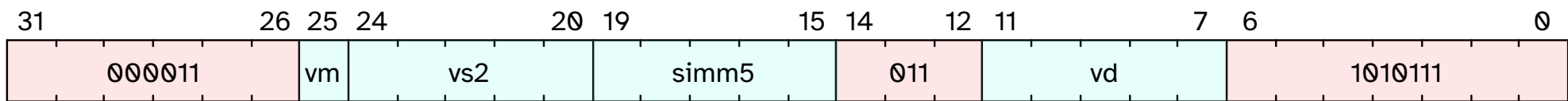
C.723. vrsb.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.723.1. Encoding



C.723.2. Description

No description available.

C.723.3. Access

M	S	U
Always	Always	Always

C.723.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.723.5. IDL Operation

C.723.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD  => vs2_val[i] + imm_val,
        VI_VRSUB => imm_val - vs2_val[i],
        VI_VAND  => vs2_val[i] & imm_val,
        VI_VOR   => vs2_val[i] | imm_val,
        VI_VXOR  => vs2_val[i] ^ imm_val,
        VI_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL  => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.723.7. Exceptions

This instruction does not generate synchronous exceptions.

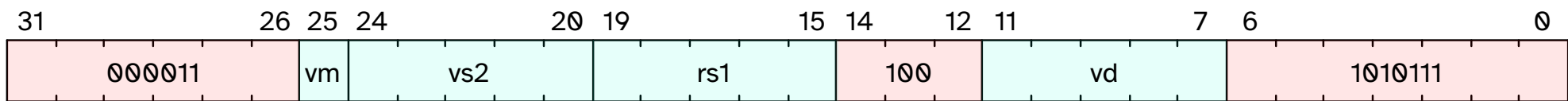
C.724. vrsub.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.724.1. Encoding



C.724.2. Description

No description available.

C.724.3. Access

M	S	U
Always	Always	Always

C.724.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.724.5. IDL Operation

C.724.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.724.7. Exceptions

This instruction does not generate synchronous exceptions.

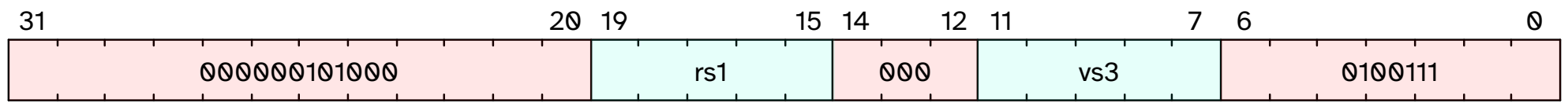
C.725. vs1r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.725.1. Encoding



C.725.2. Description

No description available.

C.725.3. Access

M	S	U
Always	Always	Always

C.725.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.725.5. IDL Operation

C.725.6. Exceptions

This instruction does not generate synchronous exceptions.

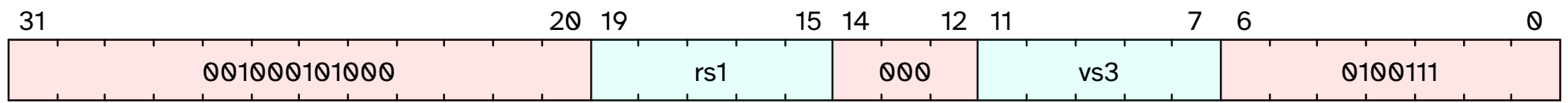
C.726. vs2r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.726.1. Encoding



C.726.2. Description

No description available.

C.726.3. Access

M	S	U
Always	Always	Always

C.726.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.726.5. IDL Operation

C.726.6. Exceptions

This instruction does not generate synchronous exceptions.

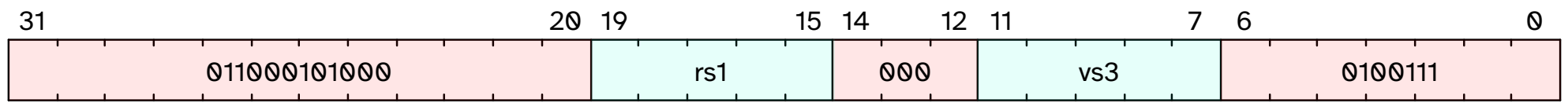
C.727. vs4r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.727.1. Encoding



C.727.2. Description

No description available.

C.727.3. Access

M	S	U
Always	Always	Always

C.727.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.727.5. IDL Operation

C.727.6. Exceptions

This instruction does not generate synchronous exceptions.

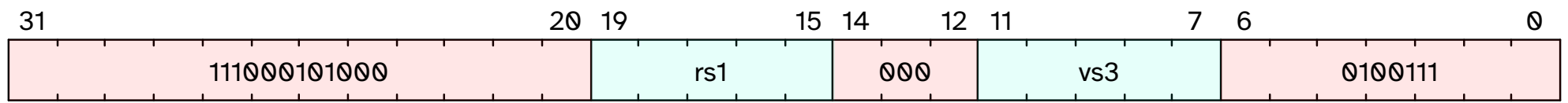
C.728. vs8r.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.728.1. Encoding



C.728.2. Description

No description available.

C.728.3. Access

M	S	U
Always	Always	Always

C.728.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.728.5. IDL Operation

C.728.6. Exceptions

This instruction does not generate synchronous exceptions.

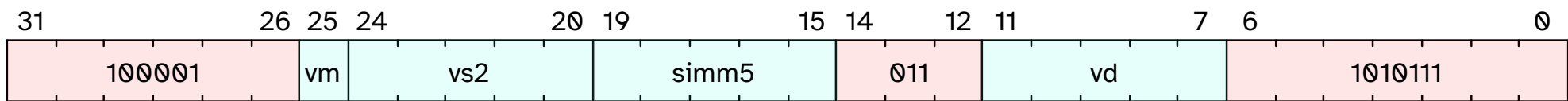
C.729. vsadd.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.729.1. Encoding



C.729.2. Description

No description available.

C.729.3. Access

M	S	U
Always	Always	Always

C.729.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.729.5. IDL Operation

C.729.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD    => vs2_val[i] + imm_val,
        VI_VRSUB   => imm_val - vs2_val[i],
        VI_VAND     => vs2_val[i] & imm_val,
        VI_VOR      => vs2_val[i] | imm_val,
        VI_VXOR     => vs2_val[i] ^ imm_val,
        VI_VSADDU   => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD    => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL     => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}

};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.729.7. Exceptions

This instruction does not generate synchronous exceptions.

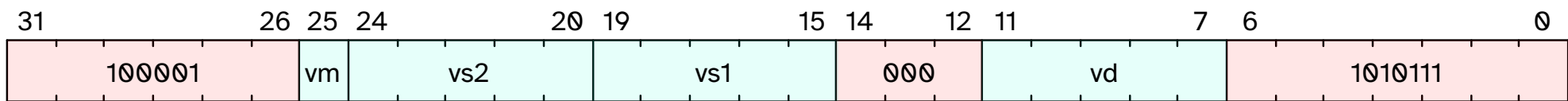
C.730. vsadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.730.1. Encoding



C.730.2. Description

No description available.

C.730.3. Access

M	S	U
Always	Always	Always

C.730.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.730.5. IDL Operation

C.730.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmul_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.730.7. Exceptions

This instruction does not generate synchronous exceptions.

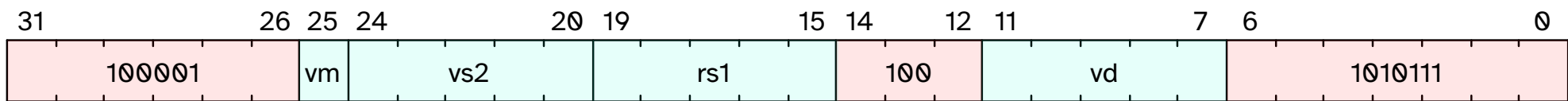
C.731. vsadd.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.731.1. Encoding



C.731.2. Description

No description available.

C.731.3. Access

M	S	U
Always	Always	Always

C.731.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.731.5. IDL Operation

C.731.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.731.7. Exceptions

This instruction does not generate synchronous exceptions.

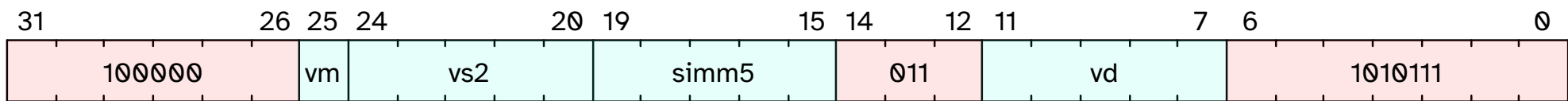
C.732. vsaddu.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.732.1. Encoding



C.732.2. Description

No description available.

C.732.3. Access

M	S	U
Always	Always	Always

C.732.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.732.5. IDL Operation

C.732.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD    => vs2_val[i] + imm_val,
        VI_VRSUB   => imm_val - vs2_val[i],
        VI_VAND     => vs2_val[i] & imm_val,
        VI_VOR      => vs2_val[i] | imm_val,
        VI_VXOR     => vs2_val[i] ^ imm_val,
        VI_VSADDU   => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD    => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL     => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}

};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.732.7. Exceptions

This instruction does not generate synchronous exceptions.

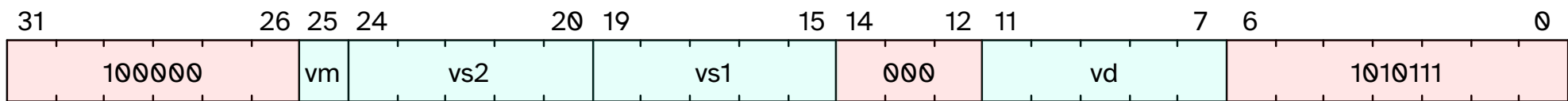
C.733. vsaddu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.733.1. Encoding



C.733.2. Description

No description available.

C.733.3. Access

M	S	U
Always	Always	Always

C.733.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.733.5. IDL Operation

C.733.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.733.7. Exceptions

This instruction does not generate synchronous exceptions.

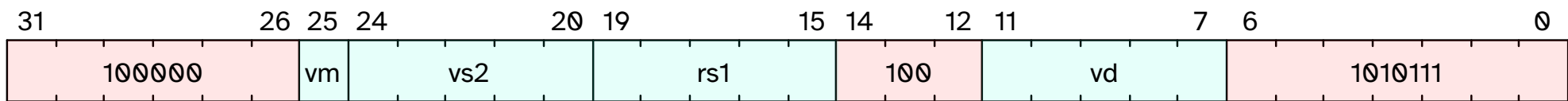
C.734. vsaddu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.734.1. Encoding



C.734.2. Description

No description available.

C.734.3. Access

M	S	U
Always	Always	Always

C.734.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.734.5. IDL Operation

C.734.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.734.7. Exceptions

This instruction does not generate synchronous exceptions.

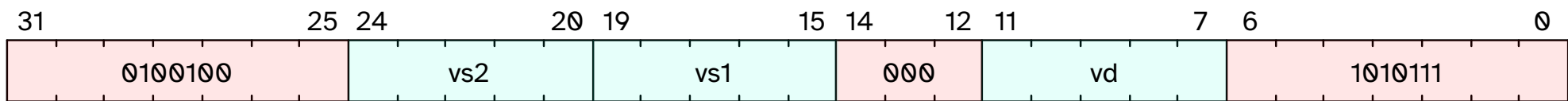
C.735. vsbc.vvm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.735.1. Encoding



C.735.2. Description

No description available.

C.735.3. Access

M	S	U
Always	Always	Always

C.735.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.735.5. IDL Operation

C.735.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  /* for bypassing normal masking in init_masked_result */
  vec_trues : vector('n, dec, bool) = undefined;
  foreach (i from 0 to (num_elem - 1)) {
    vec_trues[i] = true
  };

  let vm_val  : vector('n, dec, bool)      = read_vmask_carry(num_elem, 0b0, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vec_trues);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VVMS_VADC    => to_bits(SEW, unsigned(vs2_val[i]) + unsigned(vs1_val[i]) + unsigned(bool_to_bits(vm_val[i]))),
        VVMS_VSBC    => to_bits(SEW, unsigned(vs2_val[i]) - unsigned(vs1_val[i]) - unsigned(bool_to_bits(vm_val[i])))
      }
    }
  };
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.735.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

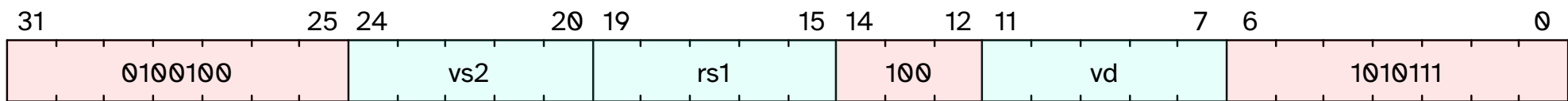
C.736. vsbc.vxm

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.736.1. Encoding



C.736.2. Description

No description available.

C.736.3. Access

M	S	U
Always	Always	Always

C.736.4. Decode Variables

```
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.736.5. IDL Operation

C.736.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_vd_masked(vd) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  /* for bypassing normal masking in init_masked_result */
  vec_trues : vector('n, dec, bool) = undefined;
  foreach (i from 0 to (num_elem - 1)) {
    vec_trues[i] = true
  };

  let vm_val  : vector('n, dec, bool)    = read_vmask_carry(num_elem, 0b0, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vec_trues);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VXMS_VADC    => to_bits(SEW, unsigned(vs2_val[i]) + unsigned(rs1_val) + unsigned(bool_to_bits(vm_val[i]))),
        VXMS_VSBC    => to_bits(SEW, unsigned(vs2_val[i]) - unsigned(rs1_val) - unsigned(bool_to_bits(vm_val[i])))
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.736.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

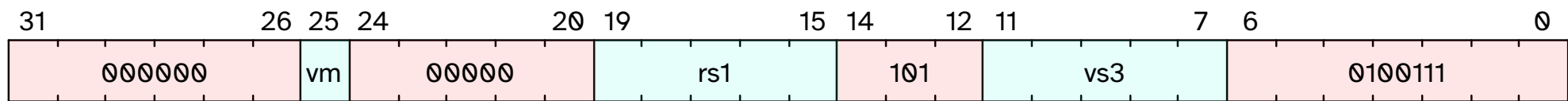
C.737. vse16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.737.1. Encoding



C.737.2. Description

No description available.

C.737.3. Access

M	S	U
Always	Always	Always

C.737.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.737.5. IDL Operation

C.737.6. Sail Operation

```
{
  let load_width_bytes = vewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsseg(nf_int, vm, vs3, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.737.7. Exceptions

This instruction does not generate synchronous exceptions.

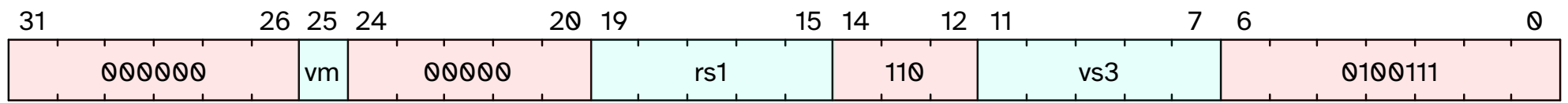
C.738. vse32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.738.1. Encoding



C.738.2. Description

No description available.

C.738.3. Access

M	S	U
Always	Always	Always

C.738.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.738.5. IDL Operation

C.738.6. Sail Operation

```
{
  let load_width_bytes = vwidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vwidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsseg(nf_int, vm, vs3, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.738.7. Exceptions

This instruction does not generate synchronous exceptions.

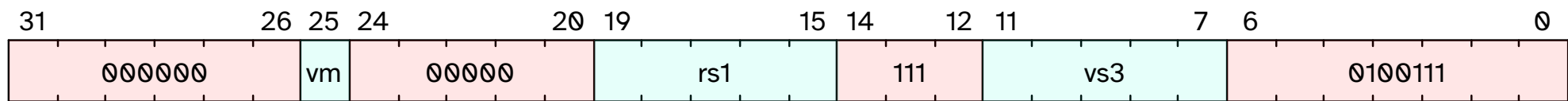
C.739. vse64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.739.1. Encoding



C.739.2. Description

No description available.

C.739.3. Access

M	S	U
Always	Always	Always

C.739.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.739.5. IDL Operation

C.739.6. Sail Operation

```
{
  let load_width_bytes = vwidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vwidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsseg(nf_int, vm, vs3, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.739.7. Exceptions

This instruction does not generate synchronous exceptions.

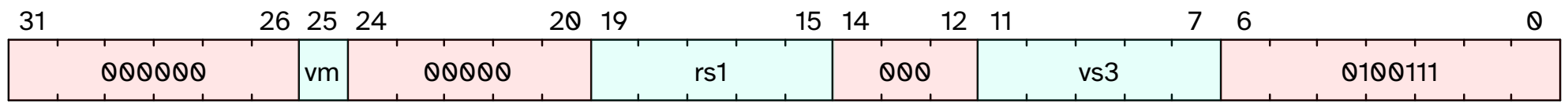
C.740. vse8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.740.1. Encoding



C.740.2. Description

No description available.

C.740.3. Access

M	S	U
Always	Always	Always

C.740.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.740.5. IDL Operation

C.740.6. Sail Operation

```
{
  let load_width_bytes = vwidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vwidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsseg(nf_int, vm, vs3, load_width_bytes, rs1, EMUL_pow, num_elem)
}
```

C.740.7. Exceptions

This instruction does not generate synchronous exceptions.

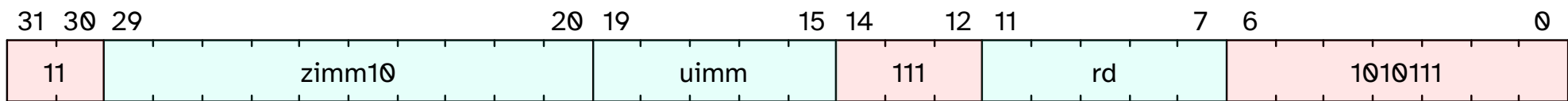
C.741. vsetivli

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.741.1. Encoding



C.741.2. Description

No description available.

C.741.3. Access

M	S	U
Always	Always	Always

C.741.4. Decode Variables

```
Bits<10> zimm10 = $encoding[29:20];
Bits<5> uimm = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.741.5. IDL Operation

C.741.6. Sail Operation

```
{
  let VLEN_pow      = get_vlen_pow();
  let ELEN_pow      = get_elen_pow();
  let LMUL_pow_ori  = get_lmUL_pow();
  let SEW_pow_ori   = get_sew_pow();
  let ratio_pow_ori = SEW_pow_ori - LMUL_pow_ori;

  /* set vtype */
  vtype->bits() = 0b0 @ zeros(sizeof(xlen) - 9) @ ma @ ta @ sew @ lmUL;

  /* check legal SEW and LMUL and calculate VLMAX */
  let LMUL_pow_new = get_lmUL_pow();
  let SEW_pow_new  = get_sew_pow();
  if SEW_pow_new > LMUL_pow_new + ELEN_pow then {
    /* Note: Implementations can set vill or trap if the vtype setting is not supported.
     * TODO: configuration support for both solutions
     */
    vtype->bits() = 0b1 @ zeros(sizeof(xlen) - 1); /* set vtype.vill */
    vl = zeros();
    print_reg("CSR vtype <- " ^ BitStr(vtype.bits()));
    print_reg("CSR vl <- " ^ BitStr(vl));
    return RETIRE_SUCCESS
  };
  let VLMAX = int_power(2, VLEN_pow + LMUL_pow_new - SEW_pow_new);
  let AVL   = unsigned(uimm); /* AVL is encoded as 5-bit zero-extended imm in the rs1 field */

  /* set vl according to VLMAX and AVL */
  vl = if AVL <= VLMAX then to_bits(sizeof(xlen), AVL)
        else if AVL < 2 * VLMAX then to_bits(sizeof(xlen), (AVL + 1) / 2)
        else to_bits(sizeof(xlen), VLMAX);
  /* Note: ceil(AVL / 2) <= vl <= VLMAX when VLMAX < AVL < (2 * VLMAX)
   * TODO: configuration support for either using ceil(AVL / 2) or VLMAX
   */
}
```

```
X(rd) = vl;  
print_reg("CSR vtype <- " ^ BitStr(vtype.bits()));  
print_reg("CSR vl <- " ^ BitStr(vl));  
  
/* reset vstart to 0 */  
vstart = zeros();  
print_reg("CSR vstart <- " ^ BitStr(vstart));  
  
RETIRE_SUCCESS  
}
```

C.741.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

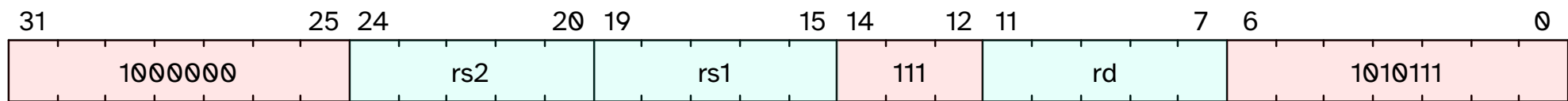
C.742. vsetvl

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.742.1. Encoding



C.742.2. Description

No description available.

C.742.3. Access

M	S	U
Always	Always	Always

C.742.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.742.5. IDL Operation

C.742.6. Exceptions

This instruction does not generate synchronous exceptions.

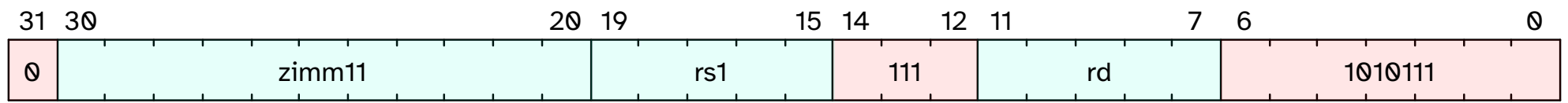
C.743. vsetvli

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.743.1. Encoding



C.743.2. Description

No description available.

C.743.3. Access

M	S	U
Always	Always	Always

C.743.4. Decode Variables

```
Bits<11> zimm11 = $encoding[30:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.743.5. IDL Operation

C.743.6. Sail Operation

```
{
  let VLEN_pow      = get_vlen_pow();
  let ELEN_pow      = get_elen_pow();
  let LMUL_pow_ori  = get_lmul_pow();
  let SEW_pow_ori   = get_sew_pow();
  let ratio_pow_ori = SEW_pow_ori - LMUL_pow_ori;

  /* set vtype */
  match op {
    VSETVLI => {
      vtype->bits() = 0b0 @ zeros(sizeof(xlen) - 9) @ ma @ ta @ sew @ lmul
    },
    VSETVL  => {
      let rs2 : regidx = sew[1 .. 0] @ lmul;
      vtype->bits() = X(rs2)
    }
  };

  /* check legal SEW and LMUL and calculate VLMAX */
  let LMUL_pow_new = get_lmul_pow();
  let SEW_pow_new  = get_sew_pow();
  if SEW_pow_new > LMUL_pow_new + ELEN_pow then {
    /* Note: Implementations can set vill or trap if the vtype setting is not supported.
     * TODO: configuration support for both solutions
     */
    vtype->bits() = 0b1 @ zeros(sizeof(xlen) - 1); /* set vtype.vill */
    vl = zeros();
    print_reg("CSR vtype <- " ^ BitStr(vtype.bits()));
    print_reg("CSR vl <- " ^ BitStr(vl));
    return RETIRE_SUCCESS
  };
  let VLMAX = int_power(2, VLEN_pow + LMUL_pow_new - SEW_pow_new);
```



```

/* set vl according to VLMAX and AVL */
if (rs1 != 0b000000) then { /* normal stripmining */
    let rs1_val = X(rs1);
    let AVL = unsigned(rs1_val);
    vl = if AVL <= VLMAX then to_bits(sizeof(xlen), AVL)
        else if AVL < 2 * VLMAX then to_bits(sizeof(xlen), (AVL + 1) / 2)
        else to_bits(sizeof(xlen), VLMAX);
    /* Note: ceil(AVL / 2) <= vl <= VLMAX when VLMAX < AVL < (2 * VLMAX)
     * TODO: configuration support for either using ceil(AVL / 2) or VLMAX
     */
    X(rd) = vl;
} else if (rd != 0b000000) then { /* set vl to VLMAX */
    let AVL = unsigned(ones(sizeof(xlen)));
    vl = to_bits(sizeof(xlen), VLMAX);
    X(rd) = vl;
} else { /* keep existing vl */
    let AVL = unsigned(vl);
    let ratio_pow_new = SEW_pow_new - LMUL_pow_new;
    if (ratio_pow_new != ratio_pow_ori) then {
        /* Note: Implementations can set vill or trap if the vtype setting is not supported.
         * TODO: configuration support for both solutions
         */
        vtype->bits() = 0b1 @ zeros(sizeof(xlen) - 1); /* set vtype.vill */
        vl = zeros();
    }
};
print_reg("CSR vtype <- " ^ BitStr(vtype.bits()));
print_reg("CSR vl <- " ^ BitStr(vl));

/* reset vstart to 0 */
vstart = zeros();
print_reg("CSR vstart <- " ^ BitStr(vstart));

RETIRE_SUCCESS
}

```

C.743.7. Exceptions

This instruction does not generate synchronous exceptions.

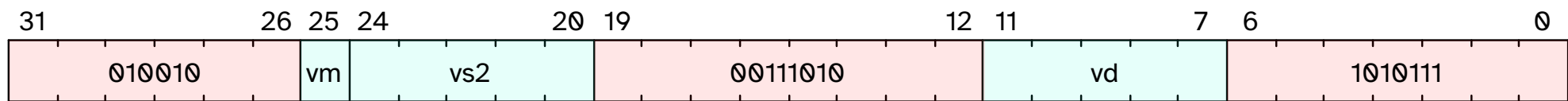
C.744. vsxt.vf2

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.744.1. Encoding



C.744.2. Description

No description available.

C.744.3. Access

M	S	U
Always	Always	Always

C.744.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.744.5. IDL Operation

C.744.6. Sail Operation

```
{
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_half = SEW / 2;
  let LMUL_pow_half = LMUL_pow - 1;

  if illegal_variable_width(vd, vm, SEW_half, LMUL_pow_half) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_half, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_half;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_half, LMUL_pow_half, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW > SEW_half);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VEXT2_ZVF2 => zero_extend(vs2_val[i]),
        VEXT2_SVF2 => sign_extend(vs2_val[i])
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.744.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

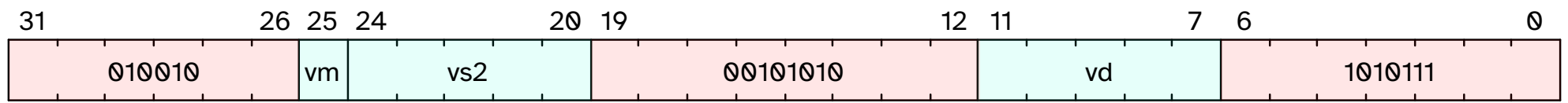
C.745. vsex.t.vf4

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.745.1. Encoding



C.745.2. Description

No description available.

C.745.3. Access

M	S	U
Always	Always	Always

C.745.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.745.5. IDL Operation

C.745.6. Sail Operation

```
{
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_quart = SEW / 4;
  let LMUL_pow_quart = LMUL_pow - 2;

  if illegal_variable_width(vd, vm, SEW_quart, LMUL_pow_quart) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_quart, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_quart;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_quart, LMUL_pow_quart, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW > SEW_quart);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VEXT4_ZVF4 => zero_extend(vs2_val[i]),
        VEXT4_SVF4 => sign_extend(vs2_val[i])
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.745.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

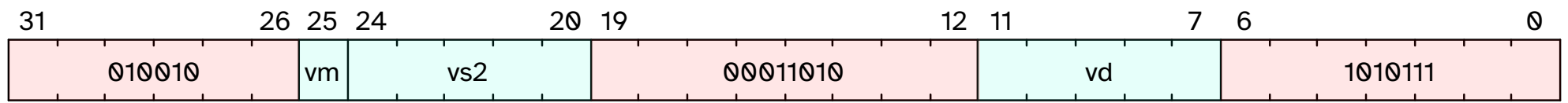
C.746. vsex.vf8

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.746.1. Encoding



C.746.2. Description

No description available.

C.746.3. Access

M	S	U
Always	Always	Always

C.746.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.746.5. IDL Operation

C.746.6. Sail Operation

```
{
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_eighth = SEW / 8;
  let LMUL_pow_eighth = LMUL_pow - 3;

  if illegal_variable_width(vd, vm, SEW_eighth, LMUL_pow_eighth) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_eighth, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_eighth;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_eighth, LMUL_pow_eighth, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW > SEW_eighth);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VEXT8_ZVF8 => zero_extend(vs2_val[i]),
        VEXT8_SVF8 => sign_extend(vs2_val[i])
      }
    }
  }
};
```



```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.746.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

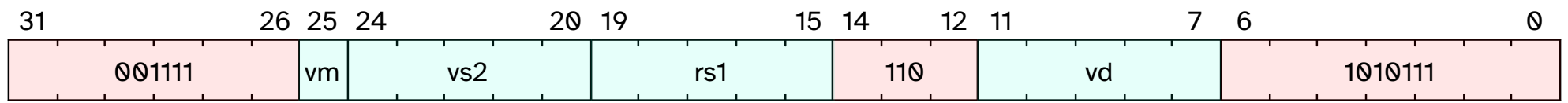
C.747. vslide1down.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.747.1. Encoding



C.747.2. Description

No description available.

C.747.3. Access

M	S	U
Always	Always	Always

C.747.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.747.5. IDL Operation

C.747.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```



```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP  => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL       => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU     => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU    => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU      => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV       => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU      => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM       => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.747.7. Exceptions

This instruction does not generate synchronous exceptions.

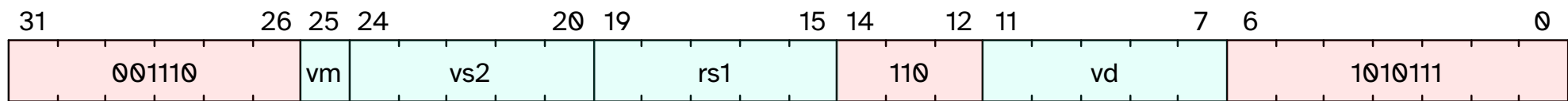
C.748. vslide1up.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.748.1. Encoding



C.748.2. Description

No description available.

C.748.3. Access

M	S	U
Always	Always	Always

C.748.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.748.5. IDL Operation

C.748.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        MVX_VAADDU    => {
          let result_add = zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
        MVX_VAADD     => {
          let result_add = sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val);
          let rounding_incr = get_fixed_rounding_incr(result_add, 1);
          slice(result_add >> 1, 0, 'm) + zero_extend('m, rounding_incr)
        },
      }
    }
  }
}
```

```

    MVX_VASUBU    => {
        let result_sub = zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VASUB     => {
        let result_sub = sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val);
        let rounding_incr = get_fixed_rounding_incr(result_sub, 1);
        slice(result_sub >> 1, 0, 'm) + zero_extend('m, rounding_incr)
    },
    MVX_VSLIDE1UP => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        if i == 0 then rs1_val else vs2_val[i - 1]
    },
    MVX_VSLIDE1DOWN => {
        let last_elem = get_end_element();
        assert(last_elem < num_elem);
        if i < last_elem then vs2_val[i + 1] else rs1_val
    },
    MVX_VMUL      => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), 0),
    MVX_VMULH     => get_slice_int(SEW, signed(vs2_val[i]) * signed(rs1_val), SEW),
    MVX_VMULHU    => get_slice_int(SEW, unsigned(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VMULHSU   => get_slice_int(SEW, signed(vs2_val[i]) * unsigned(rs1_val), SEW),
    MVX_VDIVU     => {
        let q : int = if unsigned(rs1_val) == 0 then -1 else quot_round_zero(unsigned(vs2_val[i]),
unsigned(rs1_val));
        to_bits(SEW, q)
    },
    MVX_VDIV      => {
        let elem_max : int = 2 ^ (SEW - 1) - 1;
        let elem_min : int = 0 - 2 ^ (SEW - 1);
        let q : int = if signed(rs1_val) == 0 then -1 else quot_round_zero(signed(vs2_val[i]),
signed(rs1_val));

        /* check for signed overflow */
        let q' : int = if q > elem_max then elem_min else q;
        to_bits(SEW, q')
    },
    MVX_VREMU     => {
        let r : int = if unsigned(rs1_val) == 0 then unsigned(vs2_val[i]) else
rem_round_zero(unsigned(vs2_val[i]), unsigned(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    },
    MVX_VREM      => {
        let r : int = if signed(rs1_val) == 0 then signed(vs2_val[i]) else
rem_round_zero(signed(vs2_val[i]), signed(rs1_val));
        /* signed overflow case returns zero naturally as required due to -1 divisor */
        to_bits(SEW, r)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.748.7. Exceptions

This instruction does not generate synchronous exceptions.

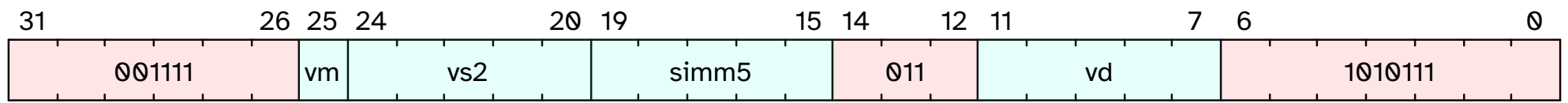
C.749. vslidedown.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.749.1. Encoding



C.749.2. Description

No description available.

C.749.3. Access

M	S	U
Always	Always	Always

C.749.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.749.5. IDL Operation

C.749.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let imm_val : nat = unsigned(zero_extend(sizeof(xlen), simm));
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result : vector('n, dec, bits('m)) = undefined;
  mask : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VSLIDEUP => {
          if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
          if i >= imm_val then vs2_val[i - imm_val] else vd_val[i]
        },
        VI_VSLIDEDOWN => {
          let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
          assert(VLMAX > 0 & VLMAX <= 'n);
          if i + imm_val < VLMAX then vs2_val[i + imm_val] else zeros()
        }
      }
    }
  }
}
```

```

    },
    VI_VRGATHER    => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX > 0 & VLMAX <= 'n');
        if imm_val < VLMAX then vs2_val[imm_val] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.749.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

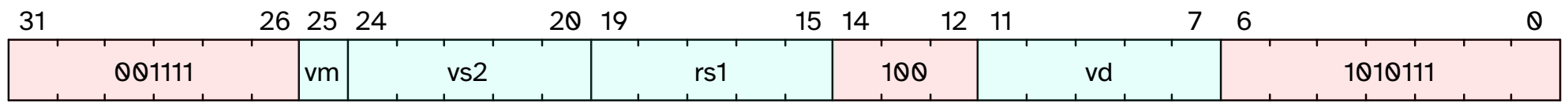
C.750. vslidedown.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.750.1. Encoding



C.750.2. Description

No description available.

C.750.3. Access

M	S	U
Always	Always	Always

C.750.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.750.5. IDL Operation

C.750.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : nat = unsigned(X(rs1));
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result : vector('n, dec, bits('m)) = undefined;
  mask : vector('n, dec, bool) = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VSLIDEUP => {
          if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
          if i >= rs1_val then vs2_val[i - rs1_val] else vd_val[i]
        },
        VX_VSLIDEDOWN => {
          let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
          assert(VLMAX > 0 & VLMAX <= 'n);
          if i + rs1_val < VLMAX then vs2_val[i + rs1_val] else zeros()
```



```

    },
    VX_VRGATHER    => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX > 0 & VLMAX <= 'n');
        if rs1_val < VLMAX then vs2_val[rs1_val] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.750.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

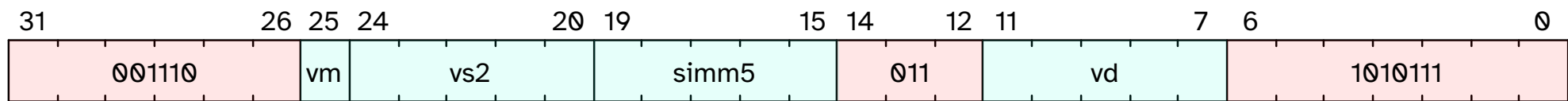
C.751. vslideup.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.751.1. Encoding



C.751.2. Description

No description available.

C.751.3. Access

M	S	U
Always	Always	Always

C.751.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.751.5. IDL Operation

C.751.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : nat                     = unsigned(zero_extend(sizeof(xlen), simm));
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VSLIDEUP => {
          if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
          if i >= imm_val then vs2_val[i - imm_val] else vd_val[i]
        },
        VI_VSLIDEDOWN => {
          let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
          assert(VLMAX > 0 & VLMAX <= 'n);
          if i + imm_val < VLMAX then vs2_val[i + imm_val] else zeros()
        }
      }
    }
  }
}
```



```

    },
    VI_VRGATHER    => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX > 0 & VLMAX <= 'n');
        if imm_val < VLMAX then vs2_val[imm_val] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.751.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

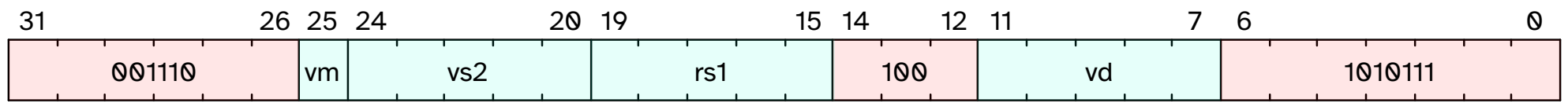
C.752. vslideup.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.752.1. Encoding



C.752.2. Description

No description available.

C.752.3. Access

M	S	U
Always	Always	Always

C.752.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.752.5. IDL Operation

C.752.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : nat                     = unsigned(X(rs1));
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VSLIDEUP => {
          if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
          if i >= rs1_val then vs2_val[i - rs1_val] else vd_val[i]
        },
        VX_VSLIDEDOWN => {
          let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
          assert(VLMAX > 0 & VLMAX <= 'n);
          if i + rs1_val < VLMAX then vs2_val[i + rs1_val] else zeros()
        }
      }
    }
  }
}
```

```

    },
    VX_VRGATHER    => {
        if (vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX > 0 & VLMAX <= 'n');
        if rs1_val < VLMAX then vs2_val[rs1_val] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.752.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

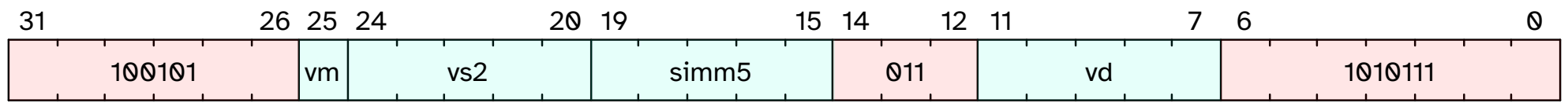
C.753. vsll.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.753.1. Encoding



C.753.2. Description

No description available.

C.753.3. Access

M	S	U
Always	Always	Always

C.753.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.753.5. IDL Operation

C.753.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD  => vs2_val[i] + imm_val,
        VI_VRSUB => imm_val - vs2_val[i],
        VI_VAND  => vs2_val[i] & imm_val,
        VI_VOR   => vs2_val[i] | imm_val,
        VI_VXOR  => vs2_val[i] ^ imm_val,
        VI_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL  => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}

};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}
```

C.753.7. Exceptions

This instruction does not generate synchronous exceptions.

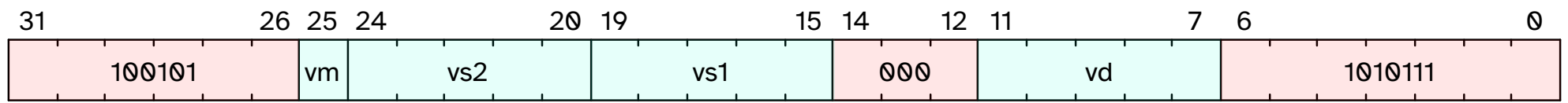
C.754. vsll.w

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.754.1. Encoding



C.754.2. Description

No description available.

C.754.3. Access

M	S	U
Always	Always	Always

C.754.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.754.5. IDL Operation

C.754.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU    => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD     => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```



```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.754.7. Exceptions

This instruction does not generate synchronous exceptions.

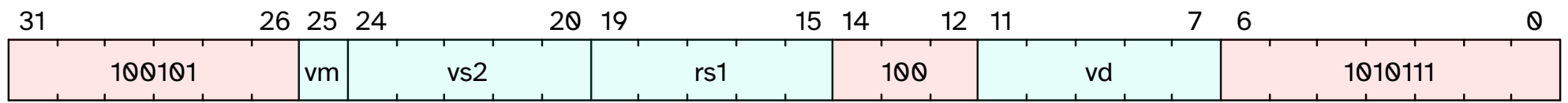
C.755. vsll.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.755.1. Encoding



C.755.2. Description

No description available.

C.755.3. Access

M	S	U
Always	Always	Always

C.755.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.755.5. IDL Operation

C.755.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```



```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN  => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX  => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.755.7. Exceptions

This instruction does not generate synchronous exceptions.

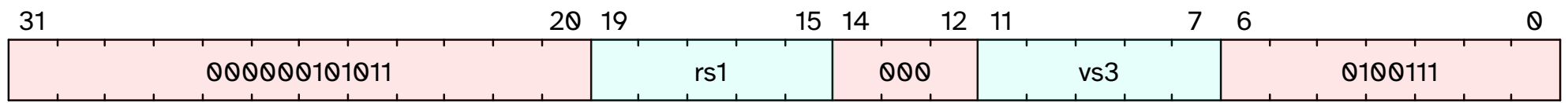
C.756. vsm.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.756.1. Encoding



C.756.2. Description

No description available.

C.756.3. Access

M	S	U
Always	Always	Always

C.756.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.756.5. IDL Operation

C.756.6. Sail Operation

```
{
  let EEW = 8;
  let EMUL_pow = 0;
  let vl_val = unsigned(vl);
  let evl : int = if vl_val % 8 == 0 then vl_val / 8 else vl_val / 8 + 1; /* the effective vector length is
evl=ceil(vl/8) */
  let num_elem = get_num_elem(EMUL_pow, EEW);

  if illegal_vd_unmasked() then { handle_illegal(); return RETIRE_FAIL };

  assert(evl >= 0);
  process_vm(vd_or_vs3, rs1, num_elem, evl, op)
}
```

C.756.7. Exceptions

This instruction does not generate synchronous exceptions.

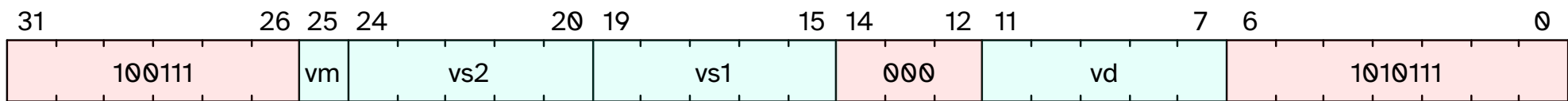
C.757. vsmul.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.757.1. Encoding



C.757.2. Description

No description available.

C.757.3. Access

M	S	U
Always	Always	Always

C.757.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.757.5. IDL Operation

C.757.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.757.7. Exceptions

This instruction does not generate synchronous exceptions.

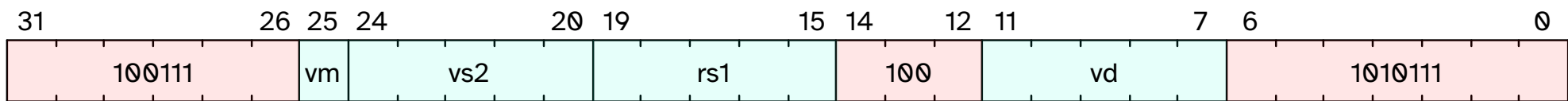
C.758. vsmul.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.758.1. Encoding



C.758.2. Description

No description available.

C.758.3. Access

M	S	U
Always	Always	Always

C.758.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.758.5. IDL Operation

C.758.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.758.7. Exceptions

This instruction does not generate synchronous exceptions.

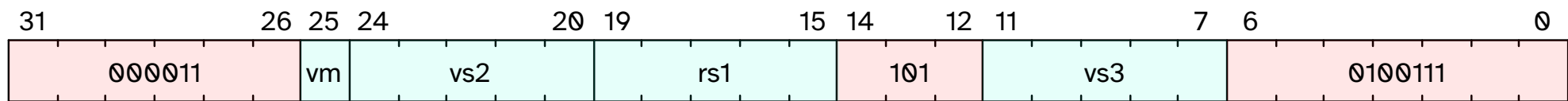
C.759. vsoxei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.759.1. Encoding



C.759.2. Description

No description available.

C.759.3. Access

M	S	U
Always	Always	Always

C.759.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.759.5. IDL Operation

C.759.6. Sail Operation

```
{
  let EEW_index_pow = vlewidth_pow(width);
  let EEW_index_bytes = vlewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.759.7. Exceptions

This instruction does not generate synchronous exceptions.

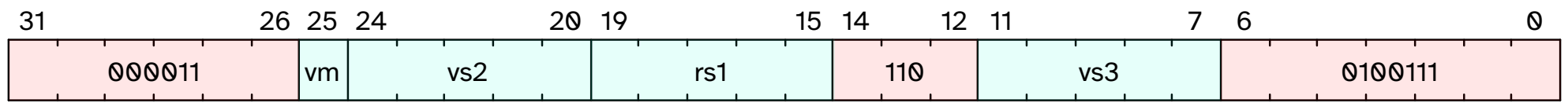
C.760. vsoxei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.760.1. Encoding



C.760.2. Description

No description available.

C.760.3. Access

M	S	U
Always	Always	Always

C.760.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.760.5. IDL Operation

C.760.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.760.7. Exceptions

This instruction does not generate synchronous exceptions.

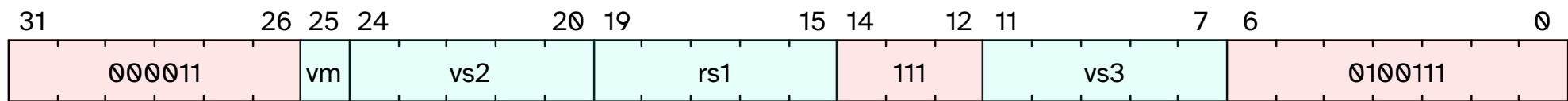
C.761. vsoxei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.761.1. Encoding



C.761.2. Description

No description available.

C.761.3. Access

M	S	U
Always	Always	Always

C.761.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.761.5. IDL Operation

C.761.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.761.7. Exceptions

This instruction does not generate synchronous exceptions.

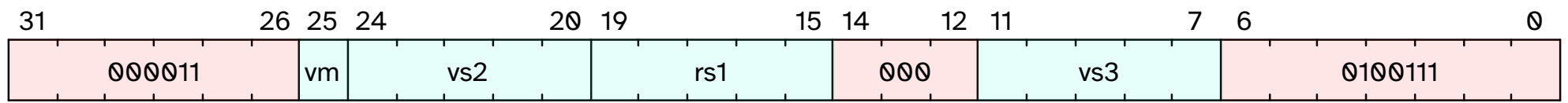
C.762. vsoxei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.762.1. Encoding



C.762.2. Description

No description available.

C.762.3. Access

M	S	U
Always	Always	Always

C.762.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.762.5. IDL Operation

C.762.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.762.7. Exceptions

This instruction does not generate synchronous exceptions.

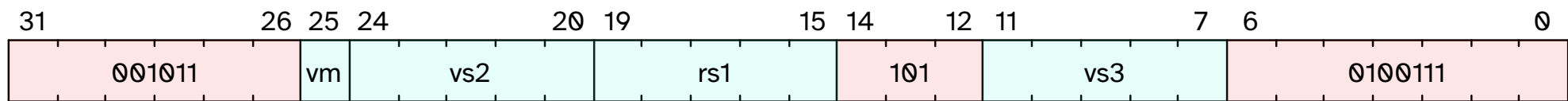
C.763. vsoxseg2ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.763.1. Encoding



C.763.2. Description

No description available.

C.763.3. Access

M	S	U
Always	Always	Always

C.763.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.763.5. IDL Operation

C.763.6. Exceptions

This instruction does not generate synchronous exceptions.

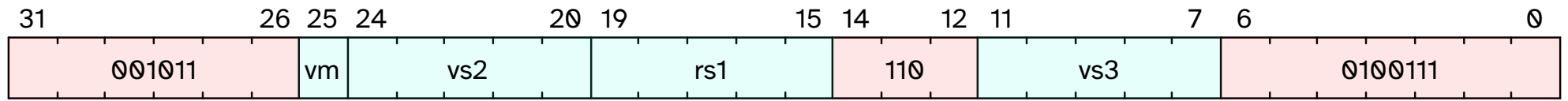
C.764. vsoxseg2ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.764.1. Encoding



C.764.2. Description

No description available.

C.764.3. Access

M	S	U
Always	Always	Always

C.764.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.764.5. IDL Operation

C.764.6. Exceptions

This instruction does not generate synchronous exceptions.

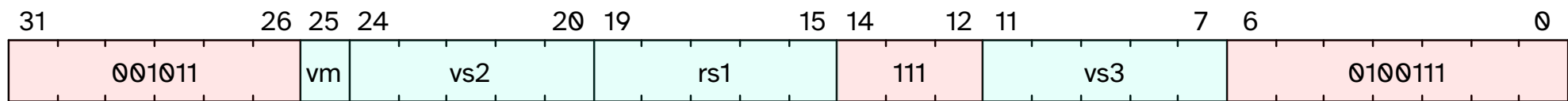
C.765. vsoxseg2ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.765.1. Encoding



C.765.2. Description

No description available.

C.765.3. Access

M	S	U
Always	Always	Always

C.765.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.765.5. IDL Operation

C.765.6. Exceptions

This instruction does not generate synchronous exceptions.

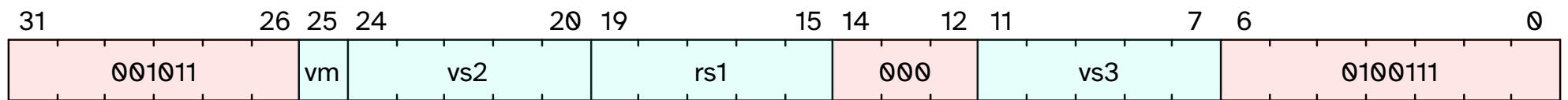
C.766. vsoxseg2ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.766.1. Encoding



C.766.2. Description

No description available.

C.766.3. Access

M	S	U
Always	Always	Always

C.766.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.766.5. IDL Operation

C.766.6. Exceptions

This instruction does not generate synchronous exceptions.

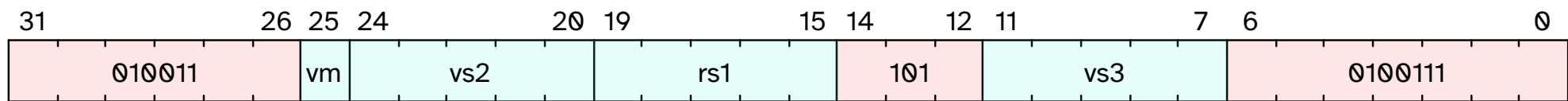
C.767. vsoxseg3ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.767.1. Encoding



C.767.2. Description

No description available.

C.767.3. Access

M	S	U
Always	Always	Always

C.767.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.767.5. IDL Operation

C.767.6. Exceptions

This instruction does not generate synchronous exceptions.

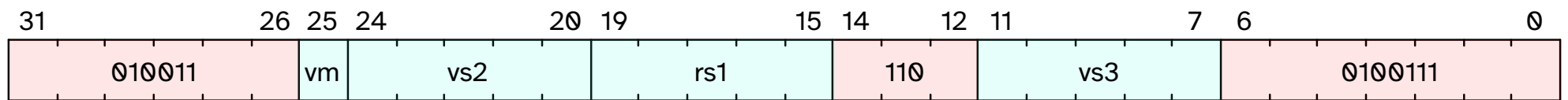
C.768. vsoxseg3ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.768.1. Encoding



C.768.2. Description

No description available.

C.768.3. Access

M	S	U
Always	Always	Always

C.768.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.768.5. IDL Operation

C.768.6. Exceptions

This instruction does not generate synchronous exceptions.

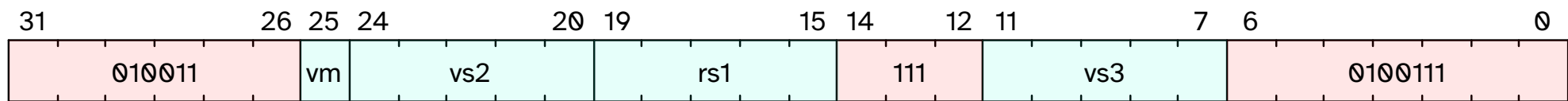
C.769. vsoxseg3ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.769.1. Encoding



C.769.2. Description

No description available.

C.769.3. Access

M	S	U
Always	Always	Always

C.769.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.769.5. IDL Operation

C.769.6. Exceptions

This instruction does not generate synchronous exceptions.

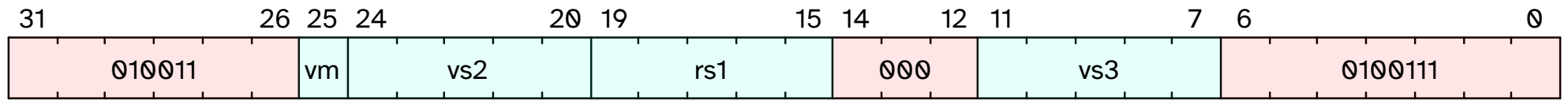
C.770. vsoxseg3ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.770.1. Encoding



C.770.2. Description

No description available.

C.770.3. Access

M	S	U
Always	Always	Always

C.770.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.770.5. IDL Operation

C.770.6. Exceptions

This instruction does not generate synchronous exceptions.

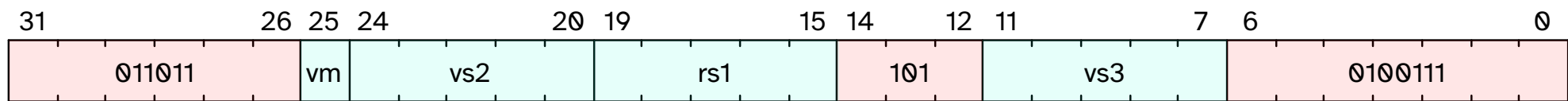
C.771. vsoxseg4ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.771.1. Encoding



C.771.2. Description

No description available.

C.771.3. Access

M	S	U
Always	Always	Always

C.771.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.771.5. IDL Operation

C.771.6. Exceptions

This instruction does not generate synchronous exceptions.

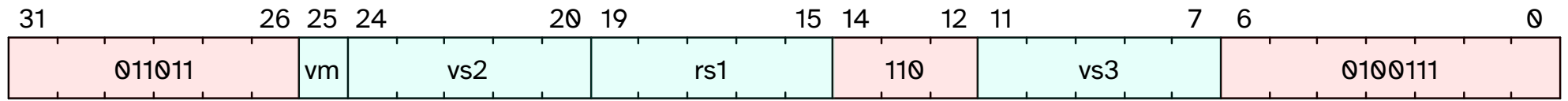
C.772. vsoxseg4ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.772.1. Encoding



C.772.2. Description

No description available.

C.772.3. Access

M	S	U
Always	Always	Always

C.772.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.772.5. IDL Operation

C.772.6. Exceptions

This instruction does not generate synchronous exceptions.

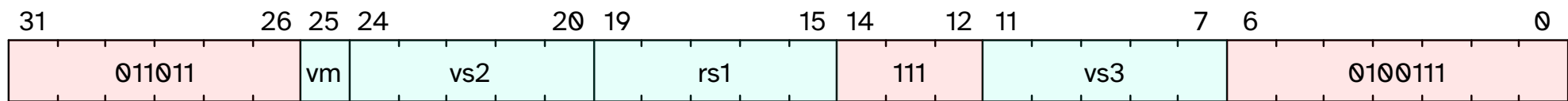
C.773. vsoxseg4ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.773.1. Encoding



C.773.2. Description

No description available.

C.773.3. Access

M	S	U
Always	Always	Always

C.773.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.773.5. IDL Operation

C.773.6. Exceptions

This instruction does not generate synchronous exceptions.

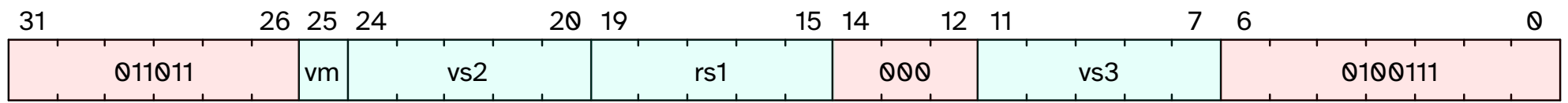
C.774. vsoxseg4ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.774.1. Encoding



C.774.2. Description

No description available.

C.774.3. Access

M	S	U
Always	Always	Always

C.774.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.774.5. IDL Operation

C.774.6. Exceptions

This instruction does not generate synchronous exceptions.

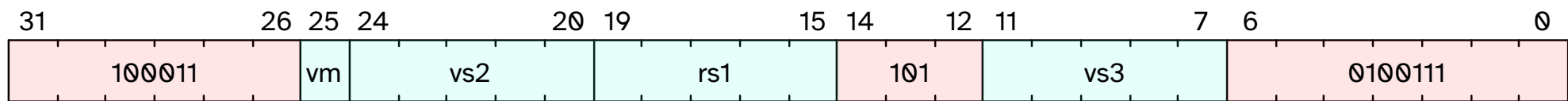
C.775. vsoxseg5ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.775.1. Encoding



C.775.2. Description

No description available.

C.775.3. Access

M	S	U
Always	Always	Always

C.775.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.775.5. IDL Operation

C.775.6. Exceptions

This instruction does not generate synchronous exceptions.

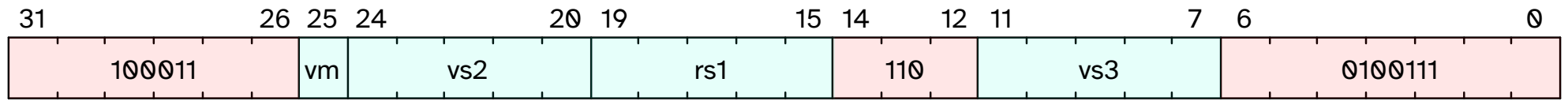
C.776. vsoxseg5ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.776.1. Encoding



C.776.2. Description

No description available.

C.776.3. Access

M	S	U
Always	Always	Always

C.776.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.776.5. IDL Operation

C.776.6. Exceptions

This instruction does not generate synchronous exceptions.

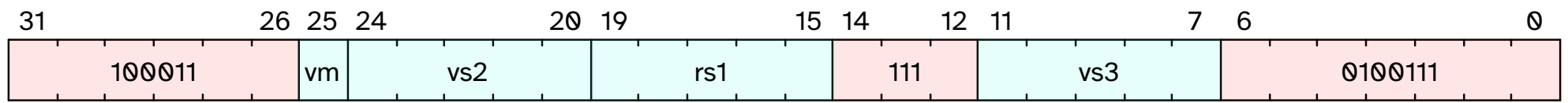
C.777. vsoxseg5ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.777.1. Encoding



C.777.2. Description

No description available.

C.777.3. Access

M	S	U
Always	Always	Always

C.777.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.777.5. IDL Operation

C.777.6. Exceptions

This instruction does not generate synchronous exceptions.

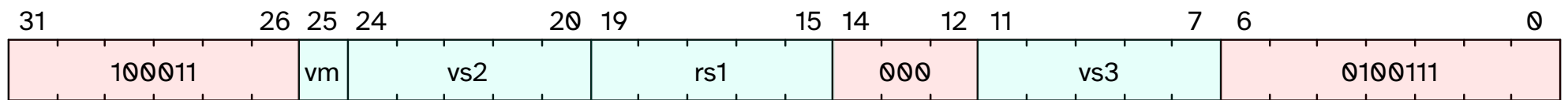
C.778. vsoxseg5ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.778.1. Encoding



C.778.2. Description

No description available.

C.778.3. Access

M	S	U
Always	Always	Always

C.778.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.778.5. IDL Operation

C.778.6. Exceptions

This instruction does not generate synchronous exceptions.

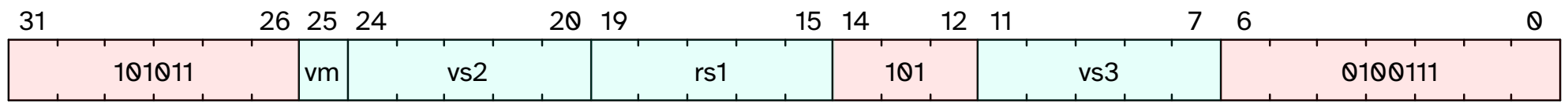
C.779. vsoxseg6ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.779.1. Encoding



C.779.2. Description

No description available.

C.779.3. Access

M	S	U
Always	Always	Always

C.779.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.779.5. IDL Operation

C.779.6. Exceptions

This instruction does not generate synchronous exceptions.

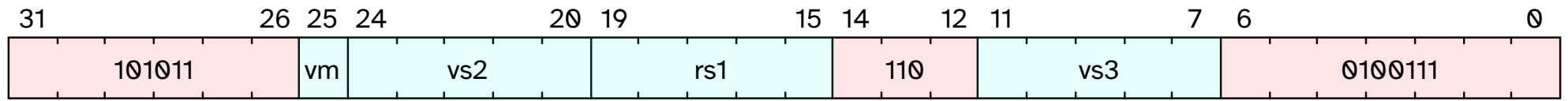
C.780. vsoxseg6ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.780.1. Encoding



C.780.2. Description

No description available.

C.780.3. Access

M	S	U
Always	Always	Always

C.780.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.780.5. IDL Operation

C.780.6. Exceptions

This instruction does not generate synchronous exceptions.

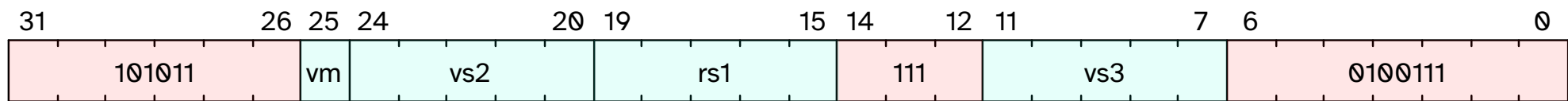
C.781. vsoxseg6ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.781.1. Encoding



C.781.2. Description

No description available.

C.781.3. Access

M	S	U
Always	Always	Always

C.781.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.781.5. IDL Operation

C.781.6. Exceptions

This instruction does not generate synchronous exceptions.

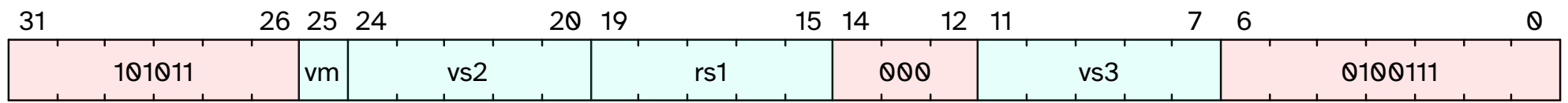
C.782. vsoxseg6ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.782.1. Encoding



C.782.2. Description

No description available.

C.782.3. Access

M	S	U
Always	Always	Always

C.782.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.782.5. IDL Operation

C.782.6. Exceptions

This instruction does not generate synchronous exceptions.

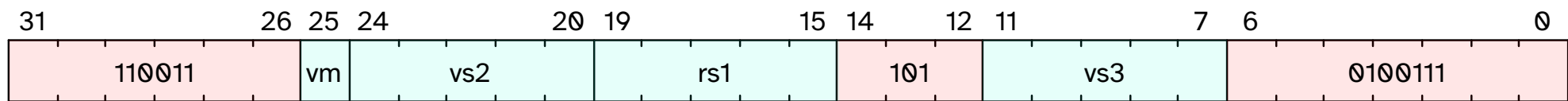
C.783. vsoxseg7ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.783.1. Encoding



C.783.2. Description

No description available.

C.783.3. Access

M	S	U
Always	Always	Always

C.783.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.783.5. IDL Operation

C.783.6. Exceptions

This instruction does not generate synchronous exceptions.

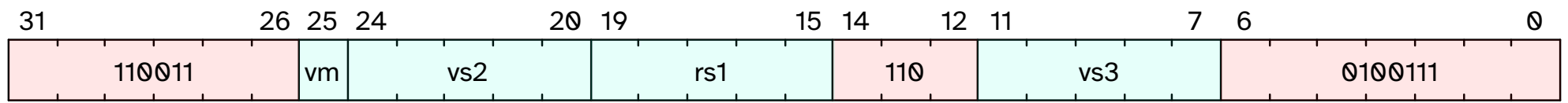
C.784. vsoxseg7ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.784.1. Encoding



C.784.2. Description

No description available.

C.784.3. Access

M	S	U
Always	Always	Always

C.784.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.784.5. IDL Operation

C.784.6. Exceptions

This instruction does not generate synchronous exceptions.

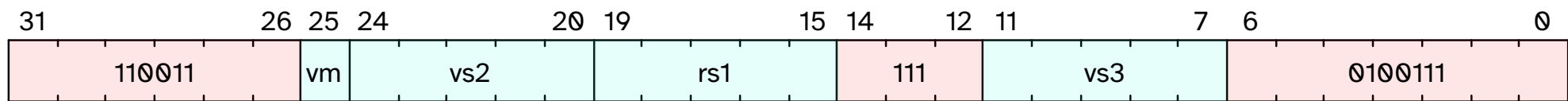
C.785. vsoxseg7ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.785.1. Encoding



C.785.2. Description

No description available.

C.785.3. Access

M	S	U
Always	Always	Always

C.785.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.785.5. IDL Operation

C.785.6. Exceptions

This instruction does not generate synchronous exceptions.

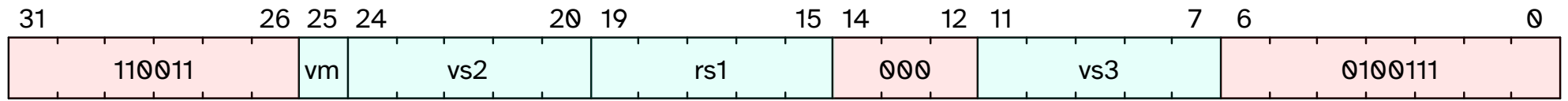
C.786. vsoxseg7ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.786.1. Encoding



C.786.2. Description

No description available.

C.786.3. Access

M	S	U
Always	Always	Always

C.786.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.786.5. IDL Operation

C.786.6. Exceptions

This instruction does not generate synchronous exceptions.

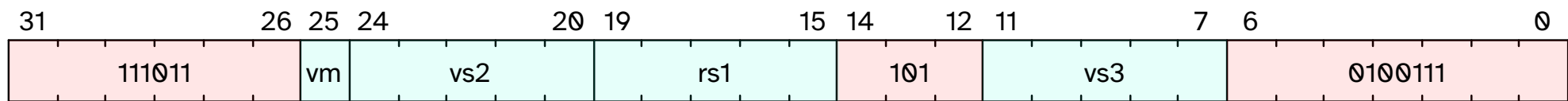
C.787. vsoxseg8ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.787.1. Encoding



C.787.2. Description

No description available.

C.787.3. Access

M	S	U
Always	Always	Always

C.787.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.787.5. IDL Operation

C.787.6. Exceptions

This instruction does not generate synchronous exceptions.

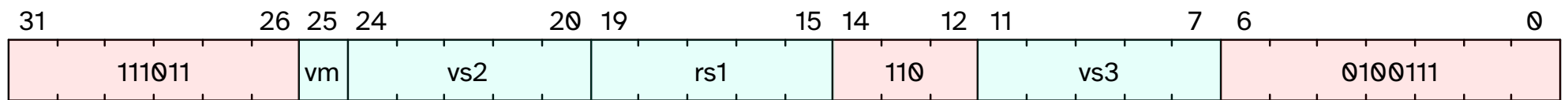
C.788. vsoxseg8ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.788.1. Encoding



C.788.2. Description

No description available.

C.788.3. Access

M	S	U
Always	Always	Always

C.788.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.788.5. IDL Operation

C.788.6. Exceptions

This instruction does not generate synchronous exceptions.

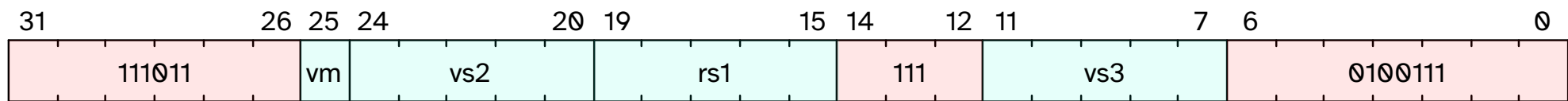
C.789. vsoxseg8ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.789.1. Encoding



C.789.2. Description

No description available.

C.789.3. Access

M	S	U
Always	Always	Always

C.789.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.789.5. IDL Operation

C.789.6. Exceptions

This instruction does not generate synchronous exceptions.

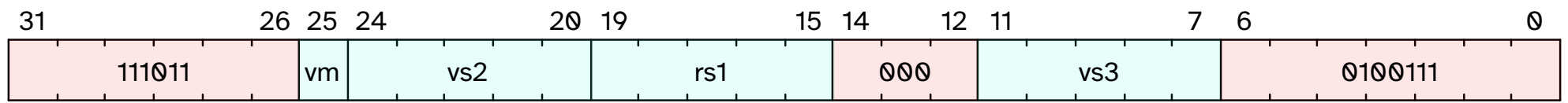
C.790. vsoxseg8ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.790.1. Encoding



C.790.2. Description

No description available.

C.790.3. Access

M	S	U
Always	Always	Always

C.790.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.790.5. IDL Operation

C.790.6. Exceptions

This instruction does not generate synchronous exceptions.

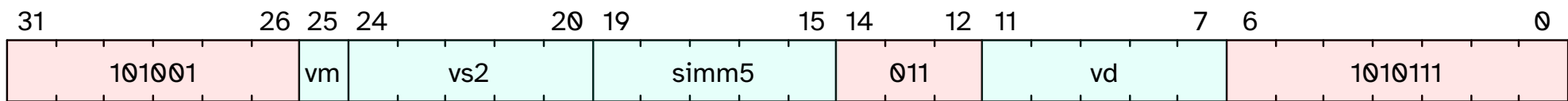
C.791. vsra.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.791.1. Encoding



C.791.2. Description

No description available.

C.791.3. Access

M	S	U
Always	Always	Always

C.791.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.791.5. IDL Operation

C.791.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD  => vs2_val[i] + imm_val,
        VI_VRSUB => imm_val - vs2_val[i],
        VI_VAND  => vs2_val[i] & imm_val,
        VI_VOR   => vs2_val[i] | imm_val,
        VI_VXOR  => vs2_val[i] ^ imm_val,
        VI_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL  => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.791.7. Exceptions

This instruction does not generate synchronous exceptions.

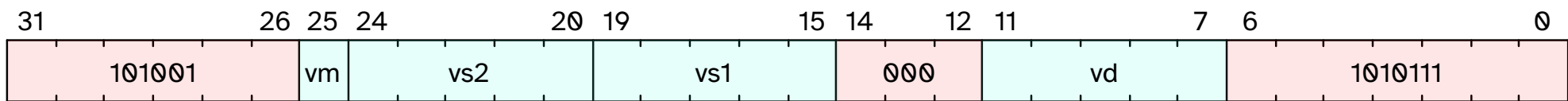
C.792. vsra.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.792.1. Encoding



C.792.2. Description

No description available.

C.792.3. Access

M	S	U
Always	Always	Always

C.792.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.792.5. IDL Operation

C.792.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);
        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.792.7. Exceptions

This instruction does not generate synchronous exceptions.

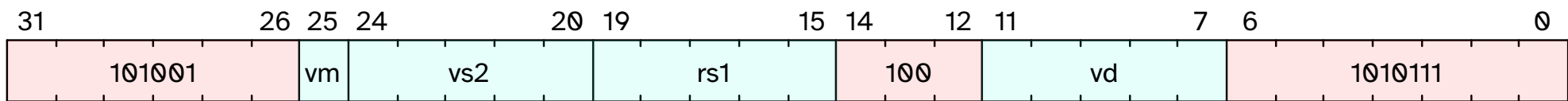
C.793. vsra.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.793.1. Encoding



C.793.2. Description

No description available.

C.793.3. Access

M	S	U
Always	Always	Always

C.793.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.793.5. IDL Operation

C.793.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.793.7. Exceptions

This instruction does not generate synchronous exceptions.

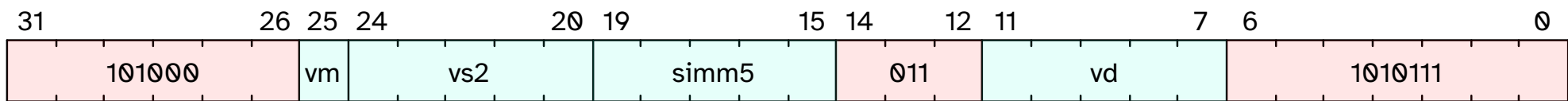
C.794. vsrl.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.794.1. Encoding



C.794.2. Description

No description available.

C.794.3. Access

M	S	U
Always	Always	Always

C.794.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.794.5. IDL Operation

C.794.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m))   = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m))   = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m))   = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD    => vs2_val[i] + imm_val,
        VI_VRSUB   => imm_val - vs2_val[i],
        VI_VAND     => vs2_val[i] & imm_val,
        VI_VOR      => vs2_val[i] | imm_val,
        VI_VXOR     => vs2_val[i] ^ imm_val,
        VI_VSADDU   => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD    => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL     => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.794.7. Exceptions

This instruction does not generate synchronous exceptions.

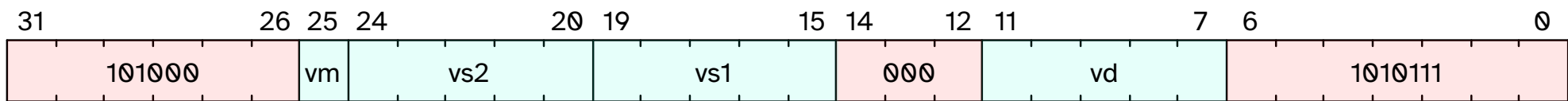
C.795. vsrl.vw

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.795.1. Encoding



C.795.2. Description

No description available.

C.795.3. Access

M	S	U
Always	Always	Always

C.795.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.795.5. IDL Operation

C.795.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.795.7. Exceptions

This instruction does not generate synchronous exceptions.

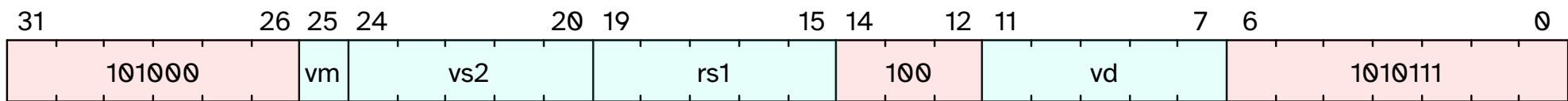
C.796. vsrl.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.796.1. Encoding



C.796.2. Description

No description available.

C.796.3. Access

M	S	U
Always	Always	Always

C.796.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.796.5. IDL Operation

C.796.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.796.7. Exceptions

This instruction does not generate synchronous exceptions.

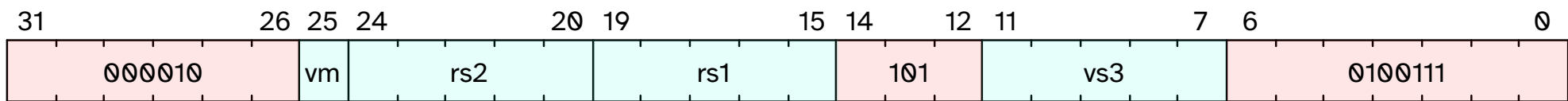
C.797. vsse16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.797.1. Encoding



C.797.2. Description

No description available.

C.797.3. Access

M	S	U
Always	Always	Always

C.797.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.797.5. IDL Operation

C.797.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vssseg(nf_int, vm, vs3, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.797.7. Exceptions

This instruction does not generate synchronous exceptions.

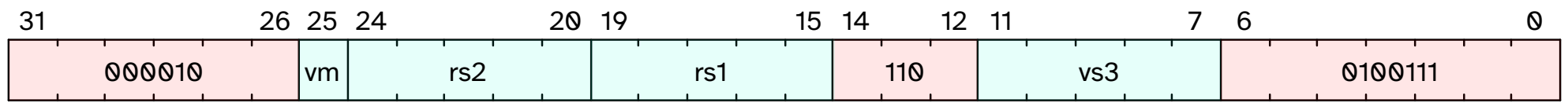
C.798. vsse32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.798.1. Encoding



C.798.2. Description

No description available.

C.798.3. Access

M	S	U
Always	Always	Always

C.798.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.798.5. IDL Operation

C.798.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vssseg(nf_int, vm, vs3, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.798.7. Exceptions

This instruction does not generate synchronous exceptions.

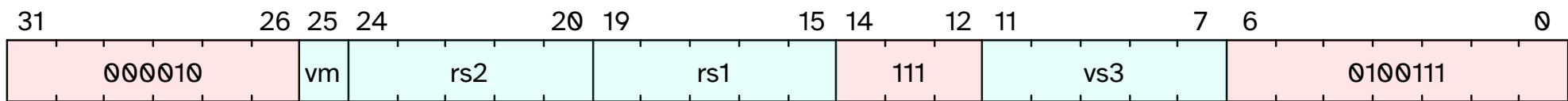
C.799. vsse64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.799.1. Encoding



C.799.2. Description

No description available.

C.799.3. Access

M	S	U
Always	Always	Always

C.799.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.799.5. IDL Operation

C.799.6. Sail Operation

```
{
  let load_width_bytes = vlewidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vlewidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmUL_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vssseg(nf_int, vm, vs3, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.799.7. Exceptions

This instruction does not generate synchronous exceptions.

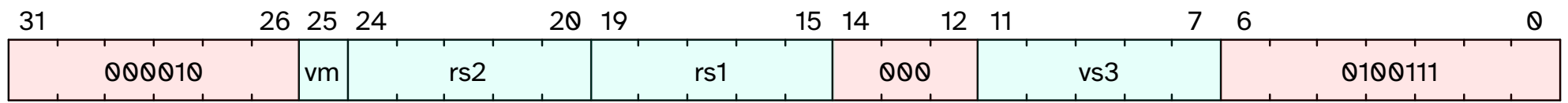
C.800. vsse8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.800.1. Encoding



C.800.2. Description

No description available.

C.800.3. Access

M	S	U
Always	Always	Always

C.800.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.800.5. IDL Operation

C.800.6. Sail Operation

```
{
  let load_width_bytes = vwidth_bytesnumber(width);
  let EEW = load_width_bytes * 8;
  let EEW_pow = vwidth_pow(width);
  let SEW_pow = get_sew_pow();
  let LMUL_pow = get_lmul_pow();
  let EMUL_pow = EEW_pow - SEW_pow + LMUL_pow;
  let num_elem = get_num_elem(EMUL_pow, EEW);
  let nf_int = nfields_int(nf);

  if illegal_store(nf_int, EEW, EMUL_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vssseg(nf_int, vm, vs3, load_width_bytes, rs1, rs2, EMUL_pow, num_elem)
}
```

C.800.7. Exceptions

This instruction does not generate synchronous exceptions.

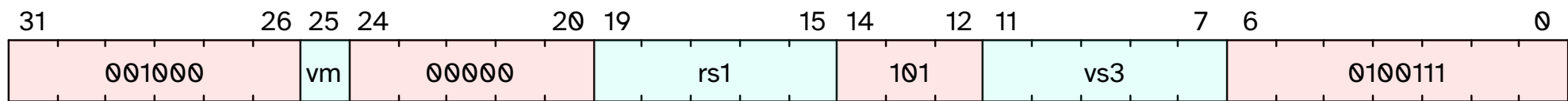
C.801. vsseg2e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.801.1. Encoding



C.801.2. Description

No description available.

C.801.3. Access

M	S	U
Always	Always	Always

C.801.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.801.5. IDL Operation

C.801.6. Exceptions

This instruction does not generate synchronous exceptions.

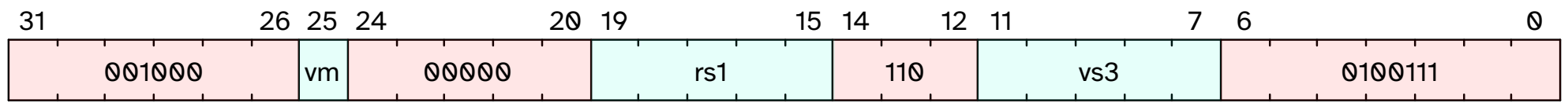
C.802. vsseg2e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.802.1. Encoding



C.802.2. Description

No description available.

C.802.3. Access

M	S	U
Always	Always	Always

C.802.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.802.5. IDL Operation

C.802.6. Exceptions

This instruction does not generate synchronous exceptions.

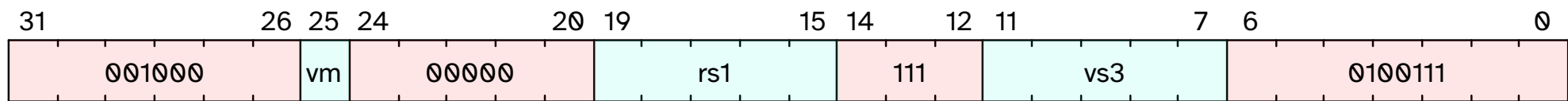
C.803. vsseg2e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.803.1. Encoding



C.803.2. Description

No description available.

C.803.3. Access

M	S	U
Always	Always	Always

C.803.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.803.5. IDL Operation

C.803.6. Exceptions

This instruction does not generate synchronous exceptions.

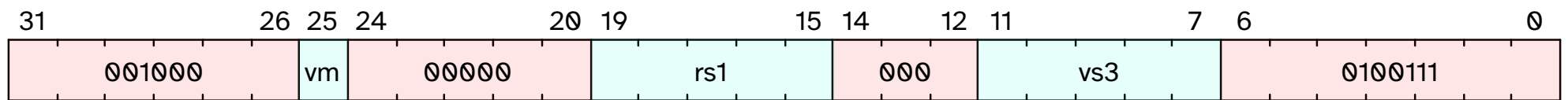
C.804. vsseg2e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.804.1. Encoding



C.804.2. Description

No description available.

C.804.3. Access

M	S	U
Always	Always	Always

C.804.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.804.5. IDL Operation

C.804.6. Exceptions

This instruction does not generate synchronous exceptions.

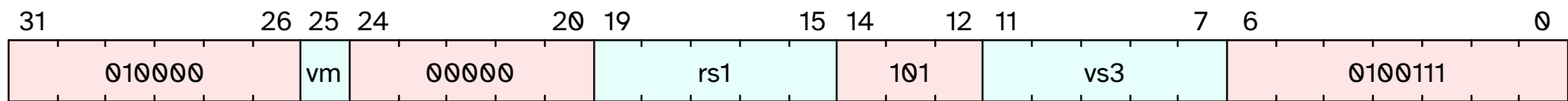
C.805. vsseg3e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.805.1. Encoding



C.805.2. Description

No description available.

C.805.3. Access

M	S	U
Always	Always	Always

C.805.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.805.5. IDL Operation

C.805.6. Exceptions

This instruction does not generate synchronous exceptions.

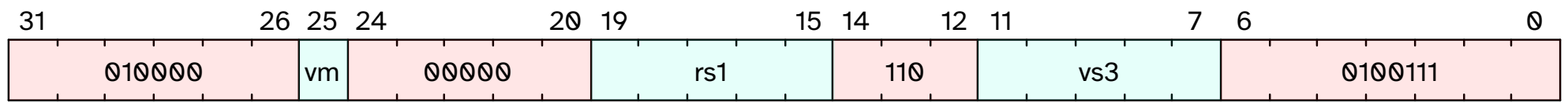
C.806. vsseg3e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.806.1. Encoding



C.806.2. Description

No description available.

C.806.3. Access

M	S	U
Always	Always	Always

C.806.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.806.5. IDL Operation

C.806.6. Exceptions

This instruction does not generate synchronous exceptions.

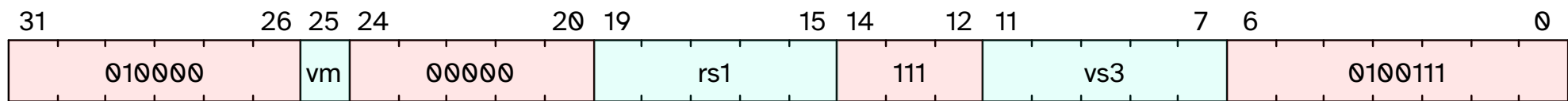
C.807. vsseg3e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.807.1. Encoding



C.807.2. Description

No description available.

C.807.3. Access

M	S	U
Always	Always	Always

C.807.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.807.5. IDL Operation

C.807.6. Exceptions

This instruction does not generate synchronous exceptions.

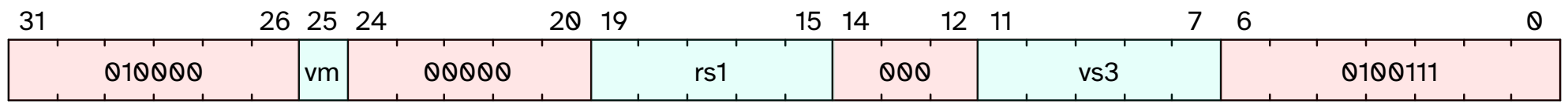
C.808. vsseg3e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.808.1. Encoding



C.808.2. Description

No description available.

C.808.3. Access

M	S	U
Always	Always	Always

C.808.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.808.5. IDL Operation

C.808.6. Exceptions

This instruction does not generate synchronous exceptions.

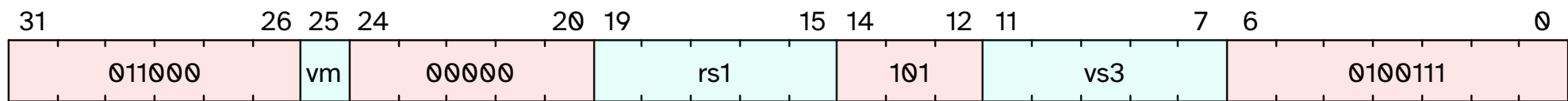
C.809. vsseg4e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.809.1. Encoding



C.809.2. Description

No description available.

C.809.3. Access

M	S	U
Always	Always	Always

C.809.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.809.5. IDL Operation

C.809.6. Exceptions

This instruction does not generate synchronous exceptions.

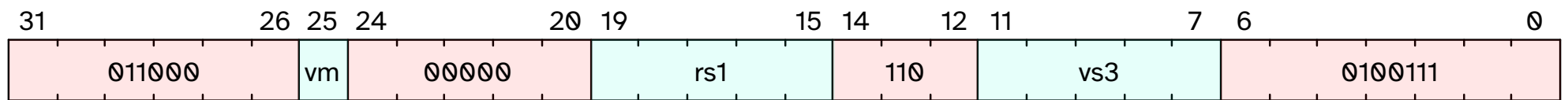
C.810. vsseg4e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.810.1. Encoding



C.810.2. Description

No description available.

C.810.3. Access

M	S	U
Always	Always	Always

C.810.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.810.5. IDL Operation

C.810.6. Exceptions

This instruction does not generate synchronous exceptions.

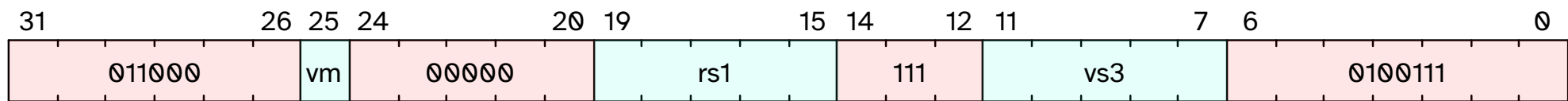
C.811. vsseg4e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.811.1. Encoding



C.811.2. Description

No description available.

C.811.3. Access

M	S	U
Always	Always	Always

C.811.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.811.5. IDL Operation

C.811.6. Exceptions

This instruction does not generate synchronous exceptions.

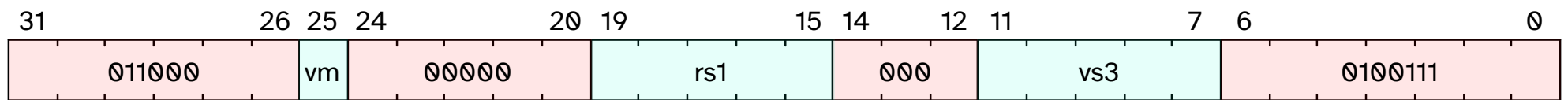
C.812. vsseg4e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.812.1. Encoding



C.812.2. Description

No description available.

C.812.3. Access

M	S	U
Always	Always	Always

C.812.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.812.5. IDL Operation

C.812.6. Exceptions

This instruction does not generate synchronous exceptions.

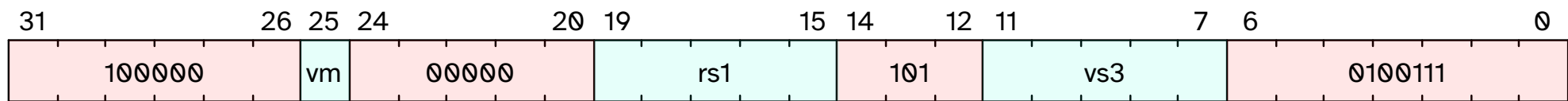
C.813. vsseg5e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.813.1. Encoding



C.813.2. Description

No description available.

C.813.3. Access

M	S	U
Always	Always	Always

C.813.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.813.5. IDL Operation

C.813.6. Exceptions

This instruction does not generate synchronous exceptions.

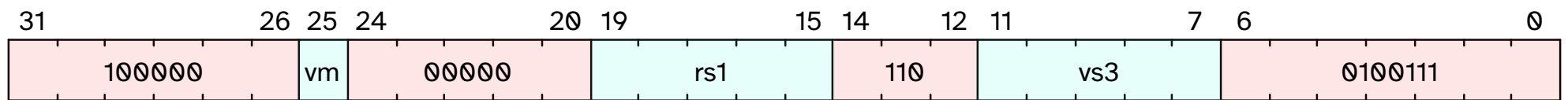
C.814. vsseg5e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.814.1. Encoding



C.814.2. Description

No description available.

C.814.3. Access

M	S	U
Always	Always	Always

C.814.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.814.5. IDL Operation

C.814.6. Exceptions

This instruction does not generate synchronous exceptions.

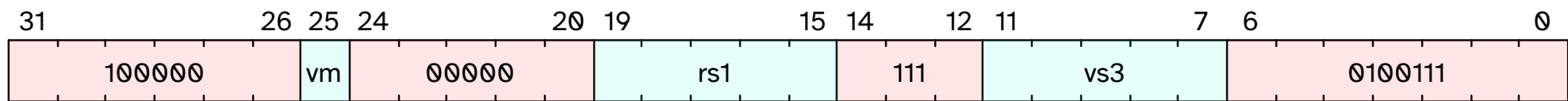
C.815. vsseg5e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.815.1. Encoding



C.815.2. Description

No description available.

C.815.3. Access

M	S	U
Always	Always	Always

C.815.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.815.5. IDL Operation

C.815.6. Exceptions

This instruction does not generate synchronous exceptions.

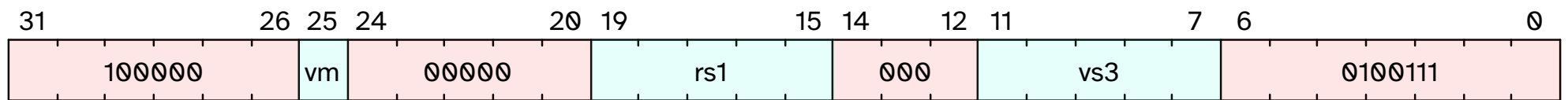
C.816. vsseg5e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.816.1. Encoding



C.816.2. Description

No description available.

C.816.3. Access

M	S	U
Always	Always	Always

C.816.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.816.5. IDL Operation

C.816.6. Exceptions

This instruction does not generate synchronous exceptions.

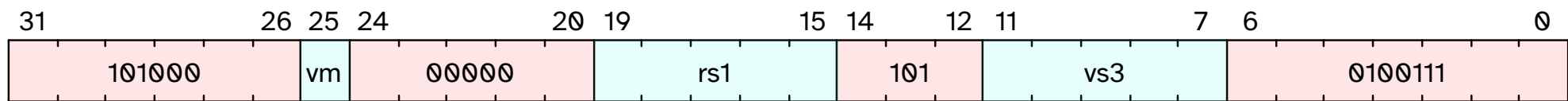
C.817. vsseg6e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.817.1. Encoding



C.817.2. Description

No description available.

C.817.3. Access

M	S	U
Always	Always	Always

C.817.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.817.5. IDL Operation

C.817.6. Exceptions

This instruction does not generate synchronous exceptions.

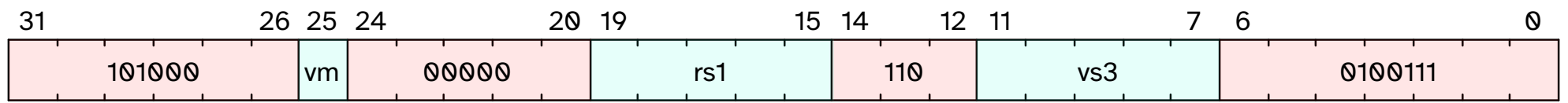
C.818. vsseg6e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.818.1. Encoding



C.818.2. Description

No description available.

C.818.3. Access

M	S	U
Always	Always	Always

C.818.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.818.5. IDL Operation

C.818.6. Exceptions

This instruction does not generate synchronous exceptions.

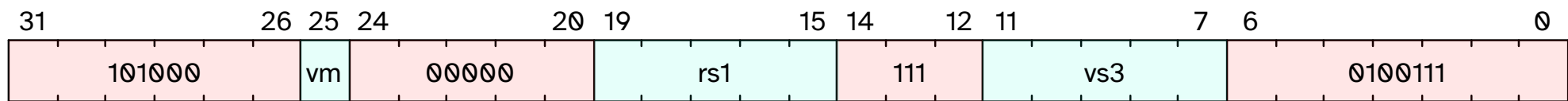
C.819. vsseg6e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.819.1. Encoding



C.819.2. Description

No description available.

C.819.3. Access

M	S	U
Always	Always	Always

C.819.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.819.5. IDL Operation

C.819.6. Exceptions

This instruction does not generate synchronous exceptions.

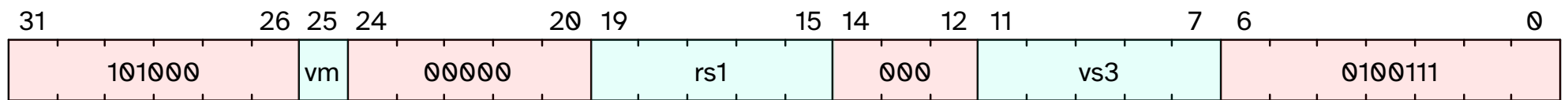
C.820. vsseg6e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.820.1. Encoding



C.820.2. Description

No description available.

C.820.3. Access

M	S	U
Always	Always	Always

C.820.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.820.5. IDL Operation

C.820.6. Exceptions

This instruction does not generate synchronous exceptions.

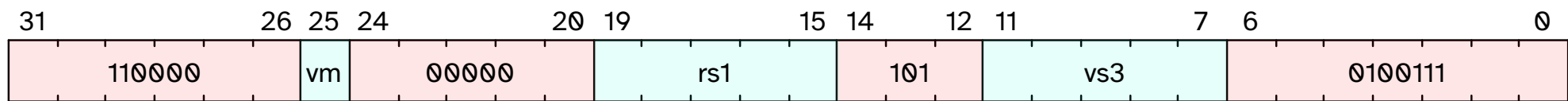
C.821. vsseg7e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.821.1. Encoding



C.821.2. Description

No description available.

C.821.3. Access

M	S	U
Always	Always	Always

C.821.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.821.5. IDL Operation

C.821.6. Exceptions

This instruction does not generate synchronous exceptions.

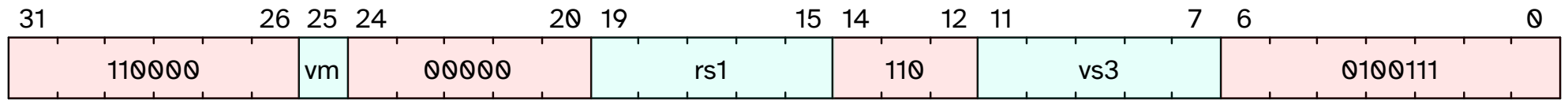
C.822. vsseg7e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.822.1. Encoding



C.822.2. Description

No description available.

C.822.3. Access

M	S	U
Always	Always	Always

C.822.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.822.5. IDL Operation

C.822.6. Exceptions

This instruction does not generate synchronous exceptions.

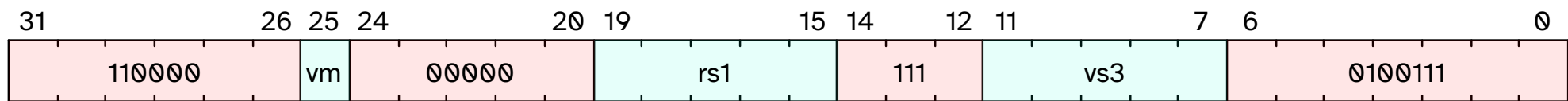
C.823. vsseg7e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.823.1. Encoding



C.823.2. Description

No description available.

C.823.3. Access

M	S	U
Always	Always	Always

C.823.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.823.5. IDL Operation

C.823.6. Exceptions

This instruction does not generate synchronous exceptions.

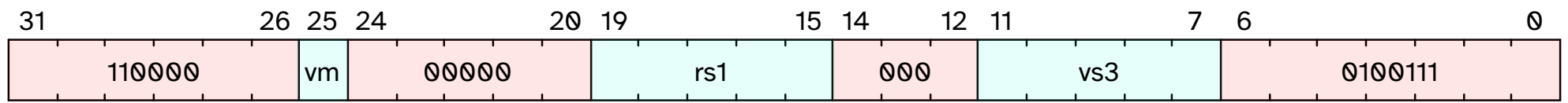
C.824. vsseg7e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.824.1. Encoding



C.824.2. Description

No description available.

C.824.3. Access

M	S	U
Always	Always	Always

C.824.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.824.5. IDL Operation

C.824.6. Exceptions

This instruction does not generate synchronous exceptions.

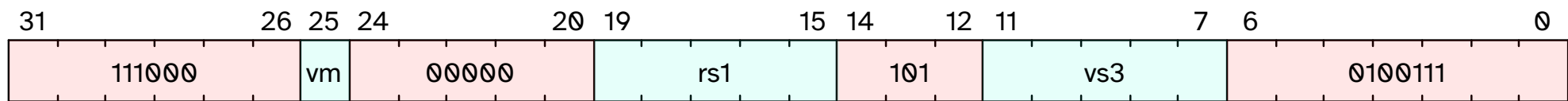
C.825. vsseg8e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.825.1. Encoding



C.825.2. Description

No description available.

C.825.3. Access

M	S	U
Always	Always	Always

C.825.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.825.5. IDL Operation

C.825.6. Exceptions

This instruction does not generate synchronous exceptions.

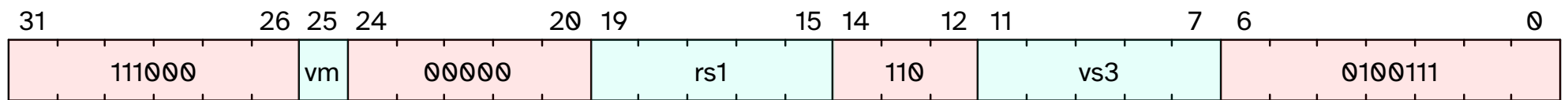
C.826. vsseg8e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.826.1. Encoding



C.826.2. Description

No description available.

C.826.3. Access

M	S	U
Always	Always	Always

C.826.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.826.5. IDL Operation

C.826.6. Exceptions

This instruction does not generate synchronous exceptions.

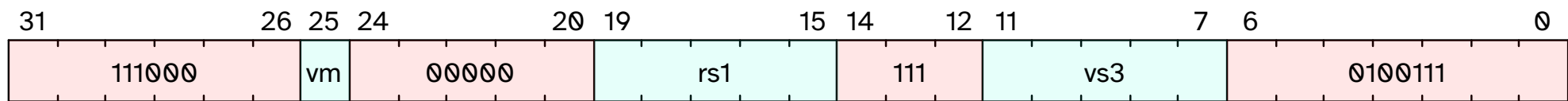
C.827. vsseg8e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.827.1. Encoding



C.827.2. Description

No description available.

C.827.3. Access

M	S	U
Always	Always	Always

C.827.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.827.5. IDL Operation

C.827.6. Exceptions

This instruction does not generate synchronous exceptions.

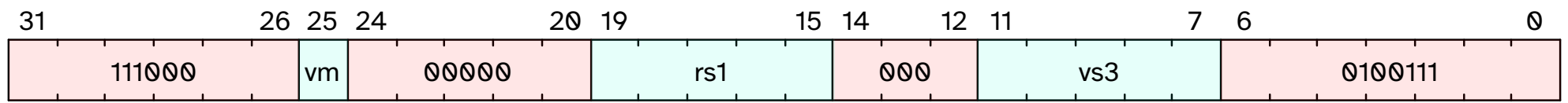
C.828. vsseg8e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.828.1. Encoding



C.828.2. Description

No description available.

C.828.3. Access

M	S	U
Always	Always	Always

C.828.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.828.5. IDL Operation

C.828.6. Exceptions

This instruction does not generate synchronous exceptions.

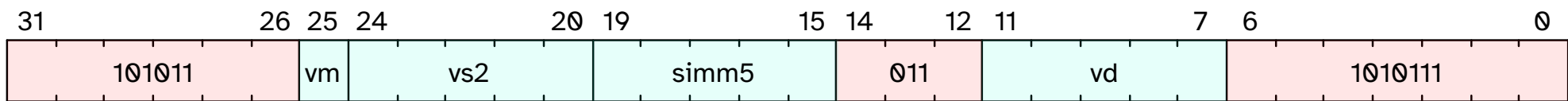
C.829. vssra.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.829.1. Encoding



C.829.2. Description

No description available.

C.829.3. Access

M	S	U
Always	Always	Always

C.829.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.829.5. IDL Operation

C.829.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                  = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD    => vs2_val[i] + imm_val,
        VI_VRSUB   => imm_val - vs2_val[i],
        VI_VAND     => vs2_val[i] & imm_val,
        VI_VOR      => vs2_val[i] | imm_val,
        VI_VXOR     => vs2_val[i] ^ imm_val,
        VI_VSADDU   => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD    => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL     => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}

};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.829.7. Exceptions

This instruction does not generate synchronous exceptions.

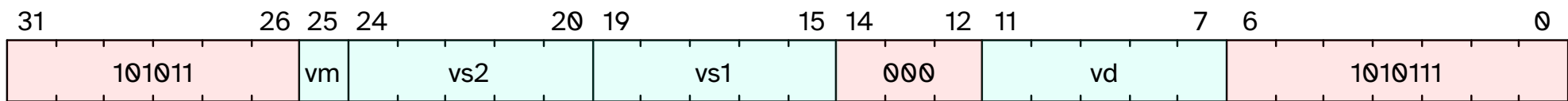
C.830. vssra.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.830.1. Encoding



C.830.2. Description

No description available.

C.830.3. Access

M	S	U
Always	Always	Always

C.830.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.830.5. IDL Operation

C.830.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.830.7. Exceptions

This instruction does not generate synchronous exceptions.

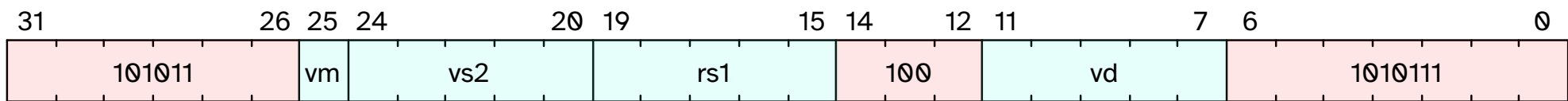
C.831. vssra.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.831.1. Encoding



C.831.2. Description

No description available.

C.831.3. Access

M	S	U
Always	Always	Always

C.831.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.831.5. IDL Operation

C.831.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.831.7. Exceptions

This instruction does not generate synchronous exceptions.

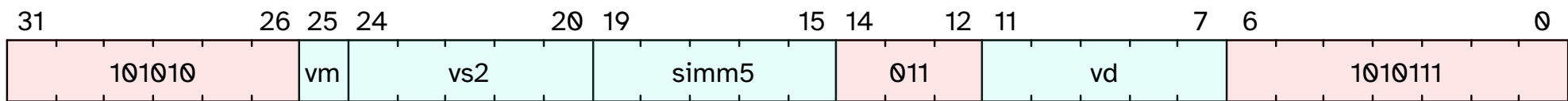
C.832. vssrl.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.832.1. Encoding



C.832.2. Description

No description available.

C.832.3. Access

M	S	U
Always	Always	Always

C.832.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.832.5. IDL Operation

C.832.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                   = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD  => vs2_val[i] + imm_val,
        VI_VRSUB => imm_val - vs2_val[i],
        VI_VAND  => vs2_val[i] & imm_val,
        VI_VOR   => vs2_val[i] | imm_val,
        VI_VXOR  => vs2_val[i] ^ imm_val,
        VI_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL   => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.832.7. Exceptions

This instruction does not generate synchronous exceptions.

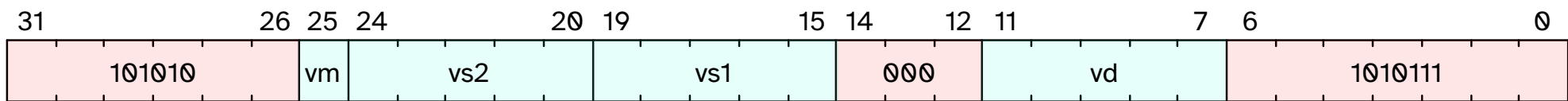
C.833. vssrl.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.833.1. Encoding



C.833.2. Description

No description available.

C.833.3. Access

M	S	U
Always	Always	Always

C.833.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.833.5. IDL Operation

C.833.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.833.7. Exceptions

This instruction does not generate synchronous exceptions.

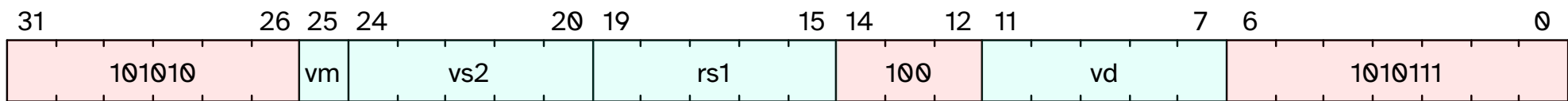
C.834. vssrLvx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.834.1. Encoding



C.834.2. Description

No description available.

C.834.3. Access

M	S	U
Always	Always	Always

C.834.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.834.5. IDL Operation

C.834.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.834.7. Exceptions

This instruction does not generate synchronous exceptions.

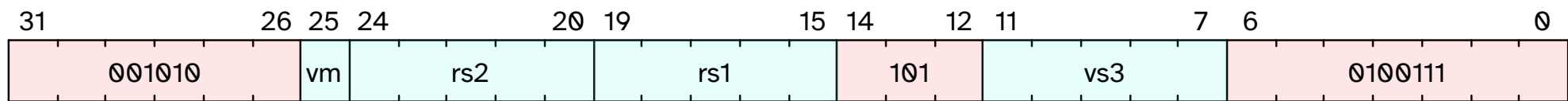
C.835. vssseg2e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.835.1. Encoding



C.835.2. Description

No description available.

C.835.3. Access

M	S	U
Always	Always	Always

C.835.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.835.5. IDL Operation

C.835.6. Exceptions

This instruction does not generate synchronous exceptions.

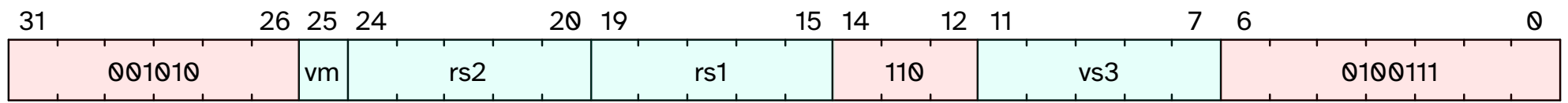
C.836. vssseg2e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.836.1. Encoding



C.836.2. Description

No description available.

C.836.3. Access

M	S	U
Always	Always	Always

C.836.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.836.5. IDL Operation

C.836.6. Exceptions

This instruction does not generate synchronous exceptions.

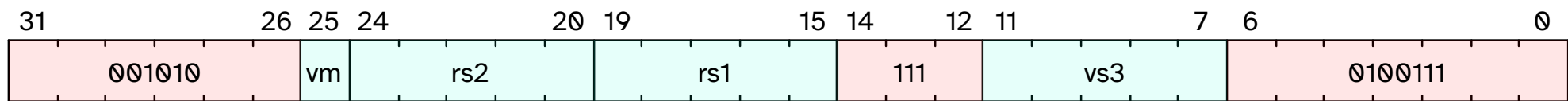
C.837. vssseg2e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.837.1. Encoding



C.837.2. Description

No description available.

C.837.3. Access

M	S	U
Always	Always	Always

C.837.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.837.5. IDL Operation

C.837.6. Exceptions

This instruction does not generate synchronous exceptions.

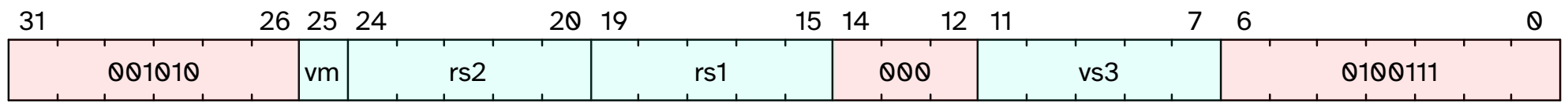
C.838. vssseg2e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.838.1. Encoding



C.838.2. Description

No description available.

C.838.3. Access

M	S	U
Always	Always	Always

C.838.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.838.5. IDL Operation

C.838.6. Exceptions

This instruction does not generate synchronous exceptions.

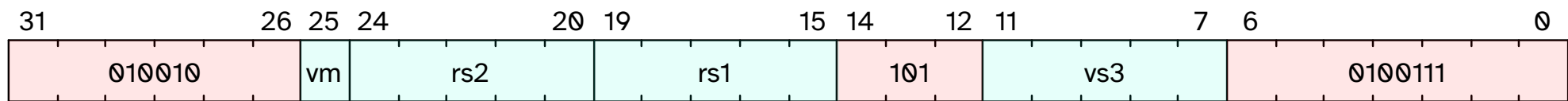
C.839. vssseg3e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.839.1. Encoding



C.839.2. Description

No description available.

C.839.3. Access

M	S	U
Always	Always	Always

C.839.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.839.5. IDL Operation

C.839.6. Exceptions

This instruction does not generate synchronous exceptions.

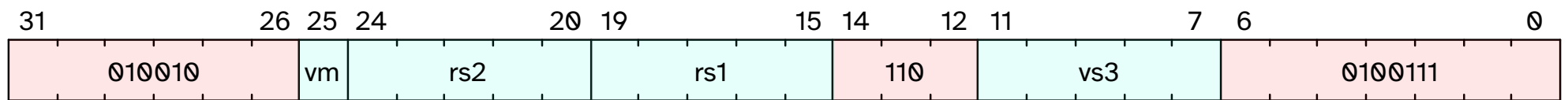
C.840. vssseg3e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.840.1. Encoding



C.840.2. Description

No description available.

C.840.3. Access

M	S	U
Always	Always	Always

C.840.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.840.5. IDL Operation

C.840.6. Exceptions

This instruction does not generate synchronous exceptions.

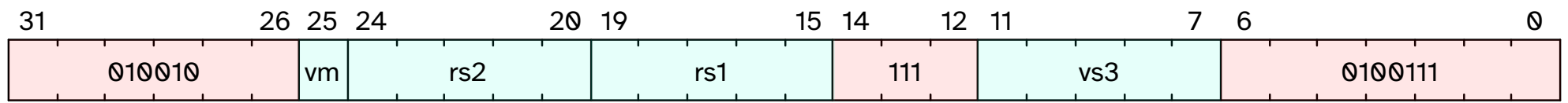
C.841. vssseg3e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.841.1. Encoding



C.841.2. Description

No description available.

C.841.3. Access

M	S	U
Always	Always	Always

C.841.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.841.5. IDL Operation

C.841.6. Exceptions

This instruction does not generate synchronous exceptions.

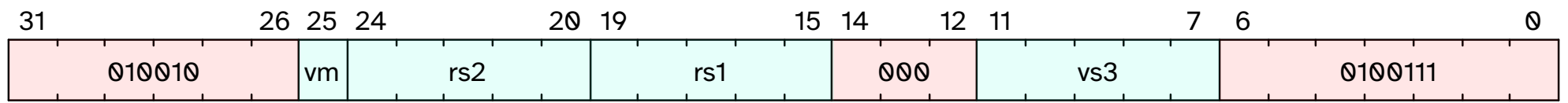
C.842. vssseg3e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.842.1. Encoding



C.842.2. Description

No description available.

C.842.3. Access

M	S	U
Always	Always	Always

C.842.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.842.5. IDL Operation

C.842.6. Exceptions

This instruction does not generate synchronous exceptions.

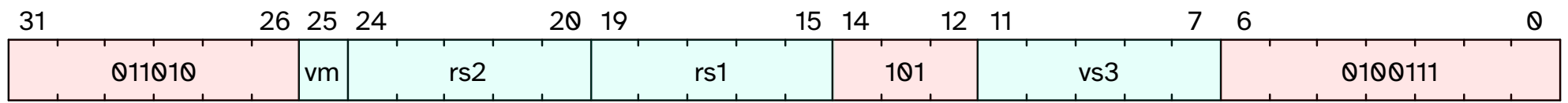
C.843. vssseg4e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.843.1. Encoding



C.843.2. Description

No description available.

C.843.3. Access

M	S	U
Always	Always	Always

C.843.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.843.5. IDL Operation

C.843.6. Exceptions

This instruction does not generate synchronous exceptions.

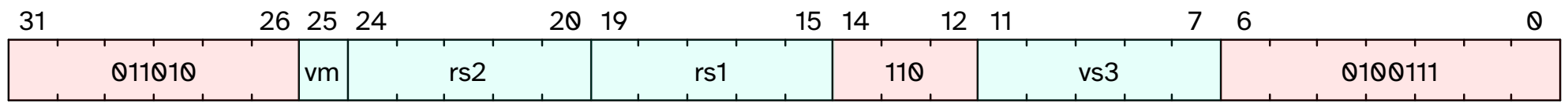
C.844. vssseg4e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.844.1. Encoding



C.844.2. Description

No description available.

C.844.3. Access

M	S	U
Always	Always	Always

C.844.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.844.5. IDL Operation

C.844.6. Exceptions

This instruction does not generate synchronous exceptions.

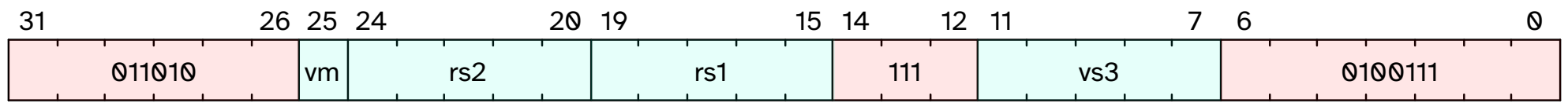
C.845. vssseg4e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.845.1. Encoding



C.845.2. Description

No description available.

C.845.3. Access

M	S	U
Always	Always	Always

C.845.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.845.5. IDL Operation

C.845.6. Exceptions

This instruction does not generate synchronous exceptions.

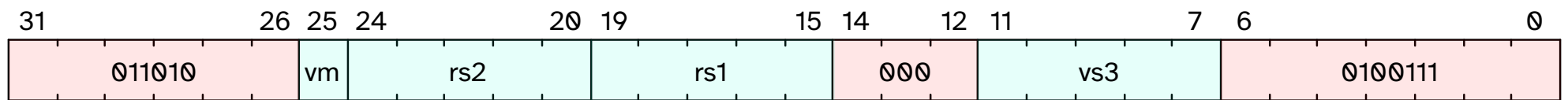
C.846. vssseg4e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.846.1. Encoding



C.846.2. Description

No description available.

C.846.3. Access

M	S	U
Always	Always	Always

C.846.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.846.5. IDL Operation

C.846.6. Exceptions

This instruction does not generate synchronous exceptions.

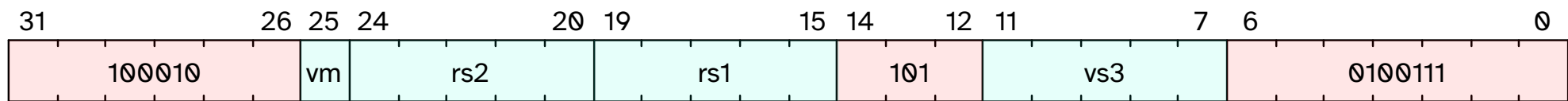
C.847. vssseg5e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.847.1. Encoding



C.847.2. Description

No description available.

C.847.3. Access

M	S	U
Always	Always	Always

C.847.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.847.5. IDL Operation

C.847.6. Exceptions

This instruction does not generate synchronous exceptions.

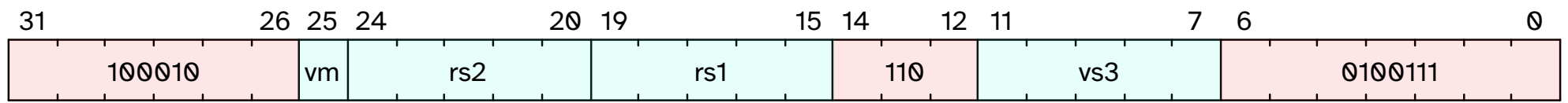
C.848. vssseg5e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.848.1. Encoding



C.848.2. Description

No description available.

C.848.3. Access

M	S	U
Always	Always	Always

C.848.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.848.5. IDL Operation

C.848.6. Exceptions

This instruction does not generate synchronous exceptions.

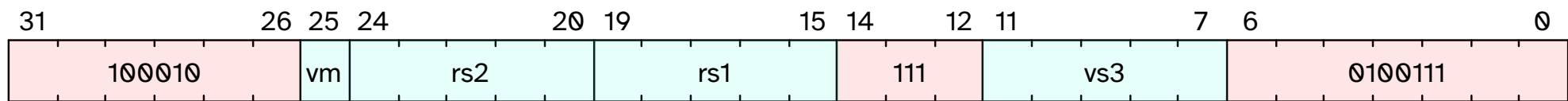
C.849. vssseg5e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.849.1. Encoding



C.849.2. Description

No description available.

C.849.3. Access

M	S	U
Always	Always	Always

C.849.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.849.5. IDL Operation

C.849.6. Exceptions

This instruction does not generate synchronous exceptions.

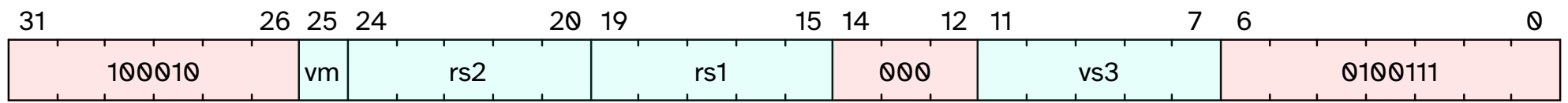
C.850. vssseg5e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.850.1. Encoding



C.850.2. Description

No description available.

C.850.3. Access

M	S	U
Always	Always	Always

C.850.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.850.5. IDL Operation

C.850.6. Exceptions

This instruction does not generate synchronous exceptions.

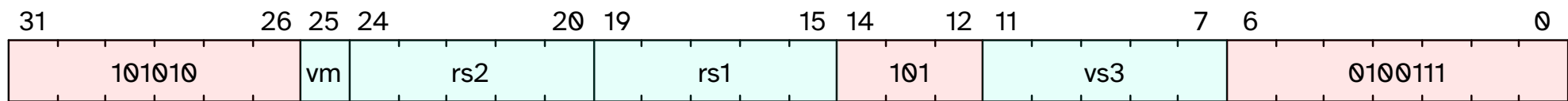
C.851. vssseg6e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.851.1. Encoding



C.851.2. Description

No description available.

C.851.3. Access

M	S	U
Always	Always	Always

C.851.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.851.5. IDL Operation

C.851.6. Exceptions

This instruction does not generate synchronous exceptions.

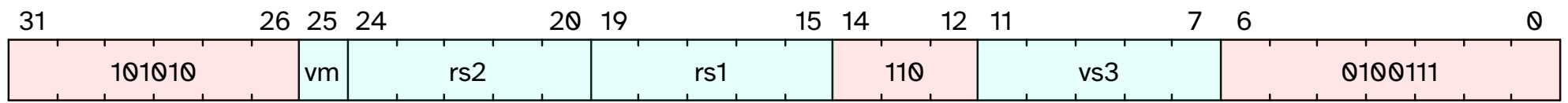
C.852. vssseg6e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.852.1. Encoding



C.852.2. Description

No description available.

C.852.3. Access

M	S	U
Always	Always	Always

C.852.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.852.5. IDL Operation

C.852.6. Exceptions

This instruction does not generate synchronous exceptions.

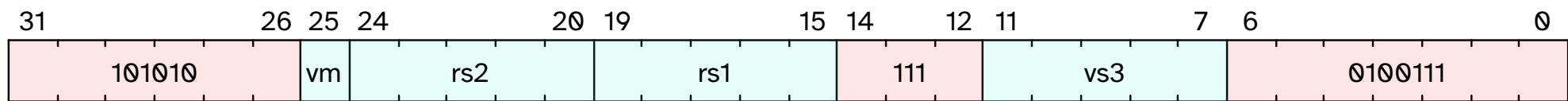
C.853. vssseg6e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.853.1. Encoding



C.853.2. Description

No description available.

C.853.3. Access

M	S	U
Always	Always	Always

C.853.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.853.5. IDL Operation

C.853.6. Exceptions

This instruction does not generate synchronous exceptions.

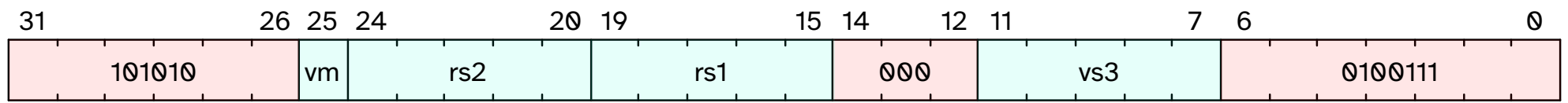
C.854. vssseg6e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.854.1. Encoding



C.854.2. Description

No description available.

C.854.3. Access

M	S	U
Always	Always	Always

C.854.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.854.5. IDL Operation

C.854.6. Exceptions

This instruction does not generate synchronous exceptions.

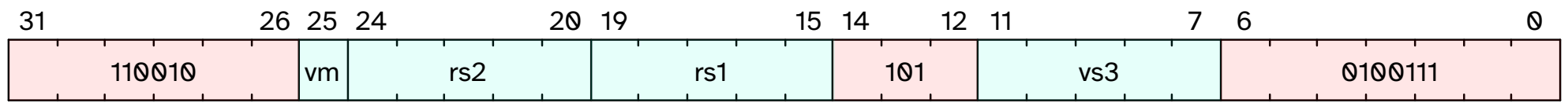
C.855. vssseg7e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.855.1. Encoding



C.855.2. Description

No description available.

C.855.3. Access

M	S	U
Always	Always	Always

C.855.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.855.5. IDL Operation

C.855.6. Exceptions

This instruction does not generate synchronous exceptions.

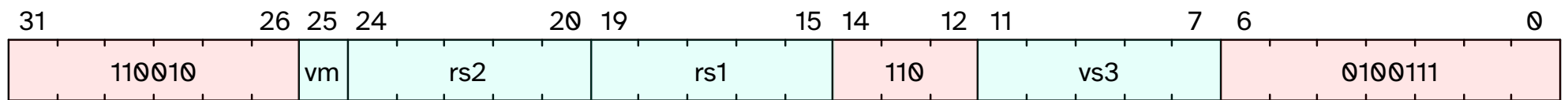
C.856. vssseg7e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.856.1. Encoding



C.856.2. Description

No description available.

C.856.3. Access

M	S	U
Always	Always	Always

C.856.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.856.5. IDL Operation

C.856.6. Exceptions

This instruction does not generate synchronous exceptions.

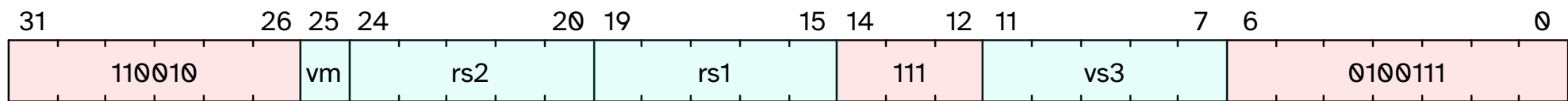
C.857. vssseg7e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.857.1. Encoding



C.857.2. Description

No description available.

C.857.3. Access

M	S	U
Always	Always	Always

C.857.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.857.5. IDL Operation

C.857.6. Exceptions

This instruction does not generate synchronous exceptions.

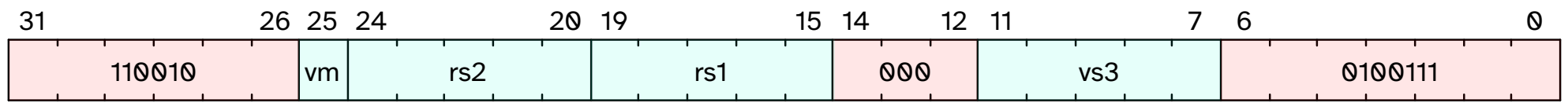
C.858. vssseg7e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.858.1. Encoding



C.858.2. Description

No description available.

C.858.3. Access

M	S	U
Always	Always	Always

C.858.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.858.5. IDL Operation

C.858.6. Exceptions

This instruction does not generate synchronous exceptions.

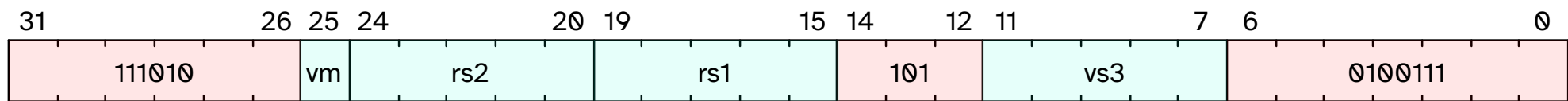
C.859. vssseg8e16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.859.1. Encoding



C.859.2. Description

No description available.

C.859.3. Access

M	S	U
Always	Always	Always

C.859.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.859.5. IDL Operation

C.859.6. Exceptions

This instruction does not generate synchronous exceptions.

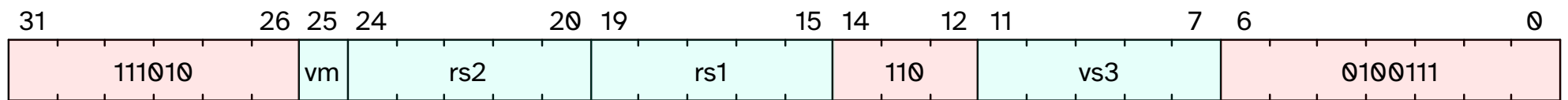
C.860. vssseg8e32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.860.1. Encoding



C.860.2. Description

No description available.

C.860.3. Access

M	S	U
Always	Always	Always

C.860.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.860.5. IDL Operation

C.860.6. Exceptions

This instruction does not generate synchronous exceptions.

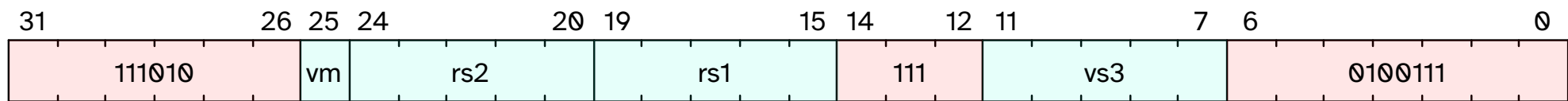
C.861. vssseg8e64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.861.1. Encoding



C.861.2. Description

No description available.

C.861.3. Access

M	S	U
Always	Always	Always

C.861.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.861.5. IDL Operation

C.861.6. Exceptions

This instruction does not generate synchronous exceptions.

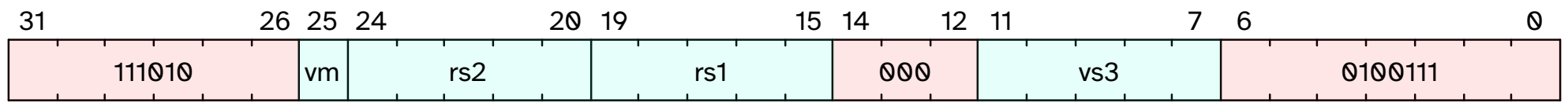
C.862. vssseg8e8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.862.1. Encoding



C.862.2. Description

No description available.

C.862.3. Access

M	S	U
Always	Always	Always

C.862.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.862.5. IDL Operation

C.862.6. Exceptions

This instruction does not generate synchronous exceptions.

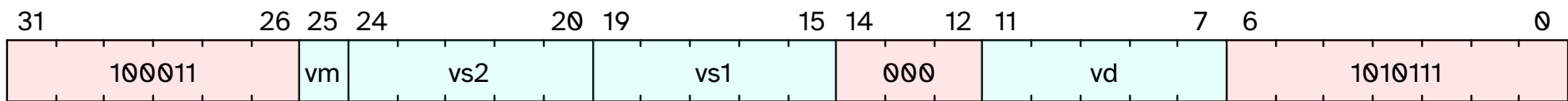
C.863. vssub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.863.1. Encoding



C.863.2. Description

No description available.

C.863.3. Access

M	S	U
Always	Always	Always

C.863.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.863.5. IDL Operation

C.863.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.863.7. Exceptions

This instruction does not generate synchronous exceptions.

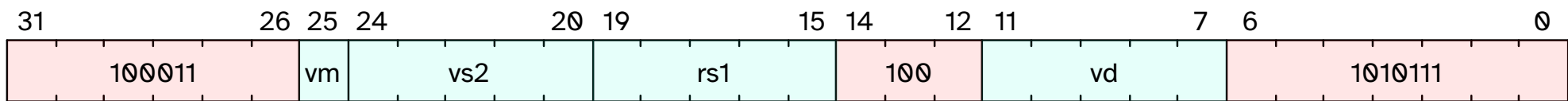
C.864. vssub.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.864.1. Encoding



C.864.2. Description

No description available.

C.864.3. Access

M	S	U
Always	Always	Always

C.864.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.864.5. IDL Operation

C.864.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.864.7. Exceptions

This instruction does not generate synchronous exceptions.

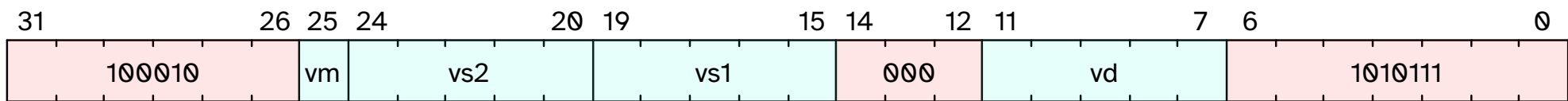
C.865. vssubu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.865.1. Encoding



C.865.2. Description

No description available.

C.865.3. Access

M	S	U
Always	Always	Always

C.865.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.865.5. IDL Operation

C.865.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW      = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.865.7. Exceptions

This instruction does not generate synchronous exceptions.

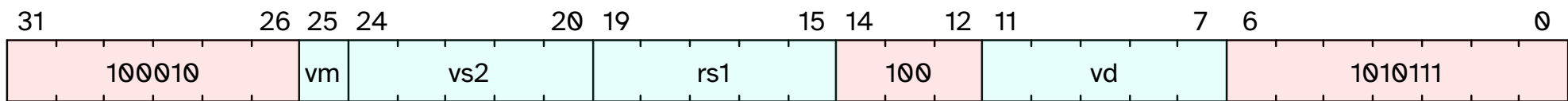
C.866. vssubu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.866.1. Encoding



C.866.2. Description

No description available.

C.866.3. Access

M	S	U
Always	Always	Always

C.866.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.866.5. IDL Operation

C.866.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.866.7. Exceptions

This instruction does not generate synchronous exceptions.

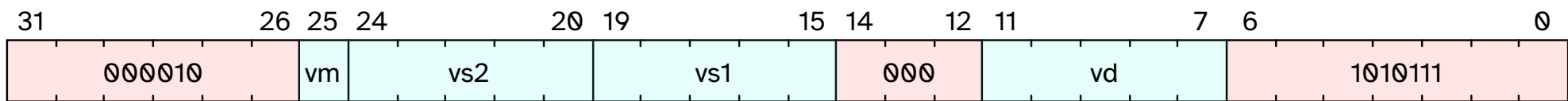
C.867. vsub.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.867.1. Encoding



C.867.2. Description

No description available.

C.867.3. Access

M	S	U
Always	Always	Always

C.867.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.867.5. IDL Operation

C.867.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmulo_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU     => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD      => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU     => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.867.7. Exceptions

This instruction does not generate synchronous exceptions.

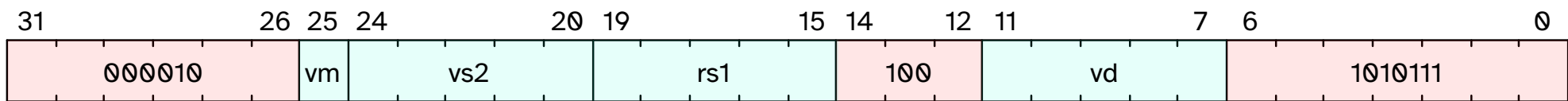
C.868. vsub.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.868.1. Encoding



C.868.2. Description

No description available.

C.868.3. Access

M	S	U
Always	Always	Always

C.868.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.868.5. IDL Operation

C.868.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.868.7. Exceptions

This instruction does not generate synchronous exceptions.

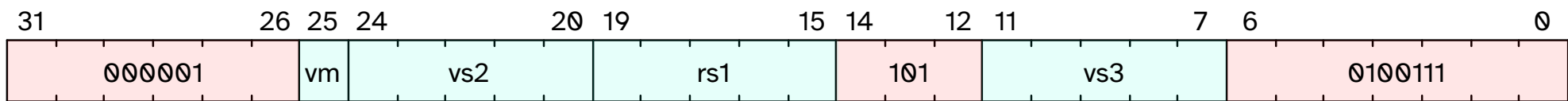
C.869. vsuxei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.869.1. Encoding



C.869.2. Description

No description available.

C.869.3. Access

M	S	U
Always	Always	Always

C.869.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.869.5. IDL Operation

C.869.6. Sail Operation

```
{
  let EEW_index_pow = vlewidth_pow(width);
  let EEW_index_bytes = vlewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.869.7. Exceptions

This instruction does not generate synchronous exceptions.

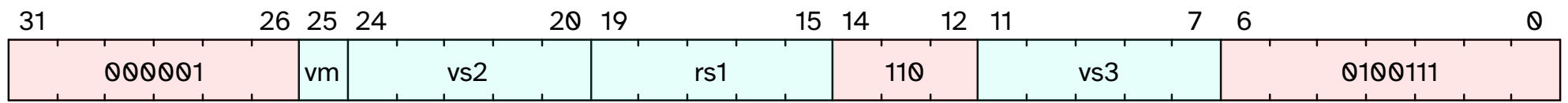
C.870. vsuxei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.870.1. Encoding



C.870.2. Description

No description available.

C.870.3. Access

M	S	U
Always	Always	Always

C.870.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.870.5. IDL Operation

C.870.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.870.7. Exceptions

This instruction does not generate synchronous exceptions.

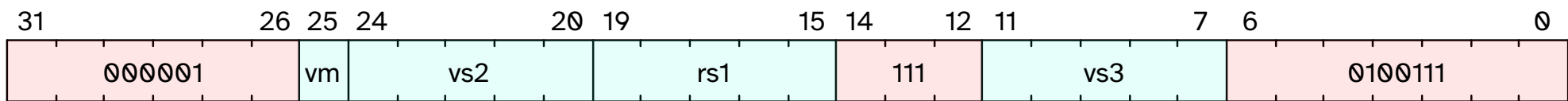
C.871. vsuxei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.871.1. Encoding



C.871.2. Description

No description available.

C.871.3. Access

M	S	U
Always	Always	Always

C.871.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.871.5. IDL Operation

C.871.6. Sail Operation

```
{
  let EEW_index_pow = vlength_pow(width);
  let EEW_index_bytes = vlength_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.871.7. Exceptions

This instruction does not generate synchronous exceptions.

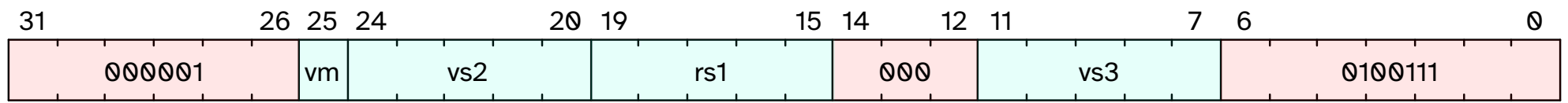
C.872. vsuxei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.872.1. Encoding



C.872.2. Description

No description available.

C.872.3. Access

M	S	U
Always	Always	Always

C.872.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.872.5. IDL Operation

C.872.6. Sail Operation

```
{
  let EEW_index_pow = vlewidth_pow(width);
  let EEW_index_bytes = vlewidth_bytesnumber(width);
  let EEW_data_pow = get_sew_pow();
  let EEW_data_bytes = get_sew_bytes();
  let EMUL_data_pow = get_lmUL_pow();
  let EMUL_index_pow = EEW_index_pow - EEW_data_pow + EMUL_data_pow;
  let num_elem = get_num_elem(EMUL_data_pow, EEW_data_bytes * 8); /* number of data and indices are the same */
  let nf_int = nfields_int(nf);

  if illegal_indexed_store(nf_int, EEW_index_bytes * 8, EMUL_index_pow, EMUL_data_pow) then { handle_illegal(); return RETIRE_FAIL };

  process_vsxseg(nf_int, vm, vs3, EEW_index_bytes, EEW_data_bytes, EMUL_index_pow, EMUL_data_pow, rs1, vs2, num_elem, 1)
}
```

C.872.7. Exceptions

This instruction does not generate synchronous exceptions.

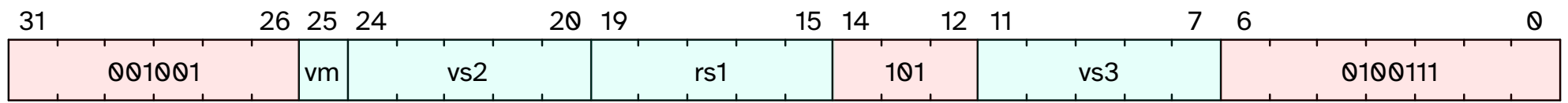
C.873. vsuxseg2ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.873.1. Encoding



C.873.2. Description

No description available.

C.873.3. Access

M	S	U
Always	Always	Always

C.873.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.873.5. IDL Operation

C.873.6. Exceptions

This instruction does not generate synchronous exceptions.

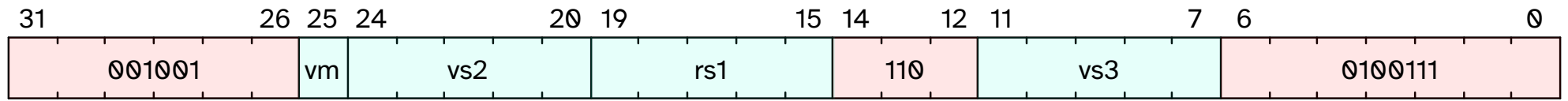
C.874. vsuxseg2ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.874.1. Encoding



C.874.2. Description

No description available.

C.874.3. Access

M	S	U
Always	Always	Always

C.874.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.874.5. IDL Operation

C.874.6. Exceptions

This instruction does not generate synchronous exceptions.

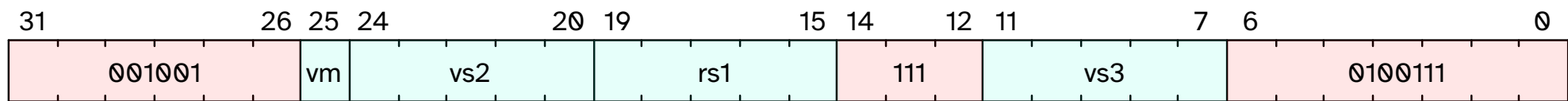
C.875. vsuxseg2ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.875.1. Encoding



C.875.2. Description

No description available.

C.875.3. Access

M	S	U
Always	Always	Always

C.875.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.875.5. IDL Operation

C.875.6. Exceptions

This instruction does not generate synchronous exceptions.

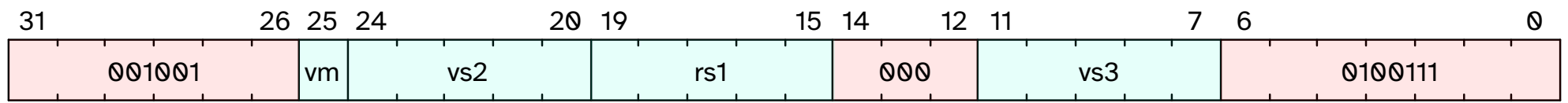
C.876. vsuxseg2ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.876.1. Encoding



C.876.2. Description

No description available.

C.876.3. Access

M	S	U
Always	Always	Always

C.876.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.876.5. IDL Operation

C.876.6. Exceptions

This instruction does not generate synchronous exceptions.

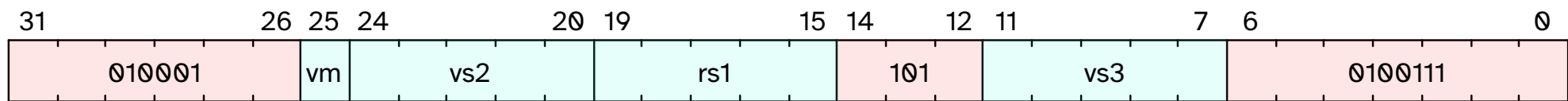
C.877. vsuxseg3ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.877.1. Encoding



C.877.2. Description

No description available.

C.877.3. Access

M	S	U
Always	Always	Always

C.877.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.877.5. IDL Operation

C.877.6. Exceptions

This instruction does not generate synchronous exceptions.

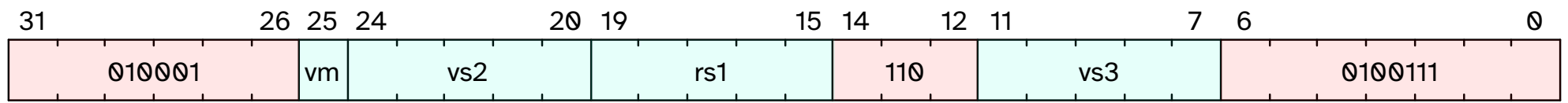
C.878. vsuxseg3ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.878.1. Encoding



C.878.2. Description

No description available.

C.878.3. Access

M	S	U
Always	Always	Always

C.878.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.878.5. IDL Operation

C.878.6. Exceptions

This instruction does not generate synchronous exceptions.

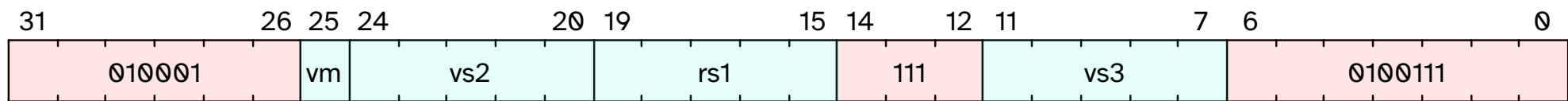
C.879. vsuxseg3ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.879.1. Encoding



C.879.2. Description

No description available.

C.879.3. Access

M	S	U
Always	Always	Always

C.879.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.879.5. IDL Operation

C.879.6. Exceptions

This instruction does not generate synchronous exceptions.

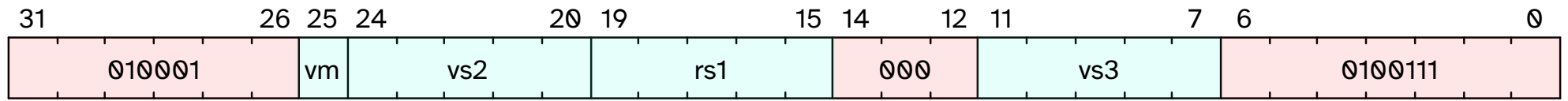
C.880. vsuxseg3ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.880.1. Encoding



C.880.2. Description

No description available.

C.880.3. Access

M	S	U
Always	Always	Always

C.880.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.880.5. IDL Operation

C.880.6. Exceptions

This instruction does not generate synchronous exceptions.

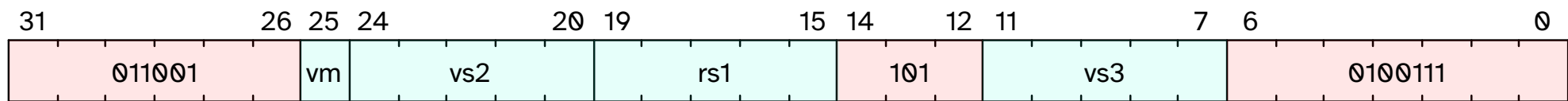
C.881. vsuxseg4ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.881.1. Encoding



C.881.2. Description

No description available.

C.881.3. Access

M	S	U
Always	Always	Always

C.881.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.881.5. IDL Operation

C.881.6. Exceptions

This instruction does not generate synchronous exceptions.

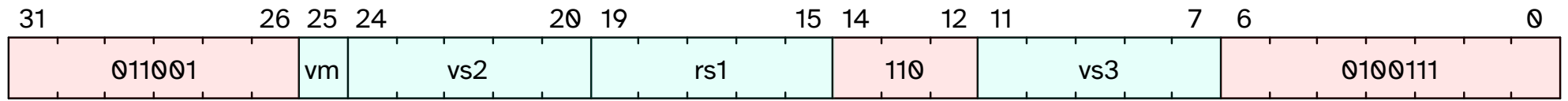
C.882. vsuxseg4ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.882.1. Encoding



C.882.2. Description

No description available.

C.882.3. Access

M	S	U
Always	Always	Always

C.882.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.882.5. IDL Operation

C.882.6. Exceptions

This instruction does not generate synchronous exceptions.

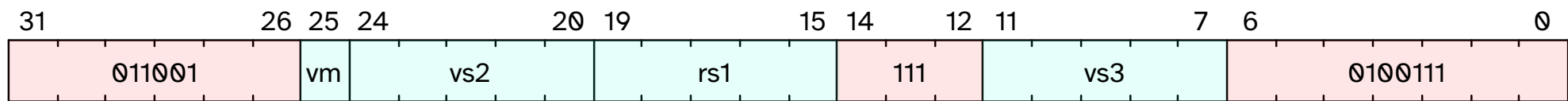
C.883. vsuxseg4ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.883.1. Encoding



C.883.2. Description

No description available.

C.883.3. Access

M	S	U
Always	Always	Always

C.883.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.883.5. IDL Operation

C.883.6. Exceptions

This instruction does not generate synchronous exceptions.

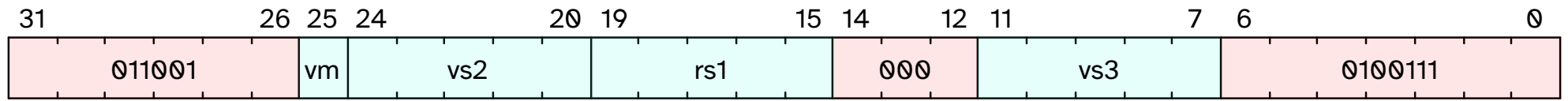
C.884. vsuxseg4ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.884.1. Encoding



C.884.2. Description

No description available.

C.884.3. Access

M	S	U
Always	Always	Always

C.884.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.884.5. IDL Operation

C.884.6. Exceptions

This instruction does not generate synchronous exceptions.

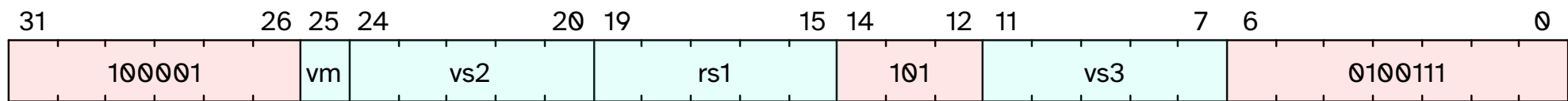
C.885. vsuxseg5ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.885.1. Encoding



C.885.2. Description

No description available.

C.885.3. Access

M	S	U
Always	Always	Always

C.885.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.885.5. IDL Operation

C.885.6. Exceptions

This instruction does not generate synchronous exceptions.

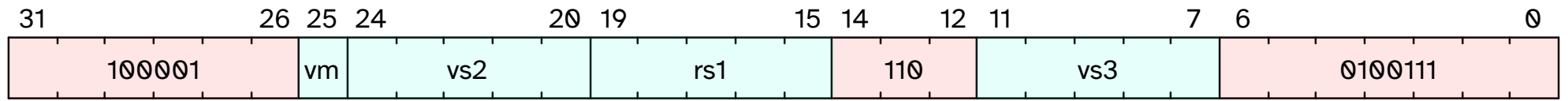
C.886. vsuxseg5ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.886.1. Encoding



C.886.2. Description

No description available.

C.886.3. Access

M	S	U
Always	Always	Always

C.886.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.886.5. IDL Operation

C.886.6. Exceptions

This instruction does not generate synchronous exceptions.

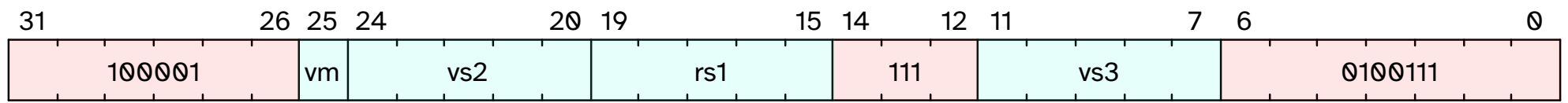
C.887. vsuxseg5ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.887.1. Encoding



C.887.2. Description

No description available.

C.887.3. Access

M	S	U
Always	Always	Always

C.887.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.887.5. IDL Operation

C.887.6. Exceptions

This instruction does not generate synchronous exceptions.

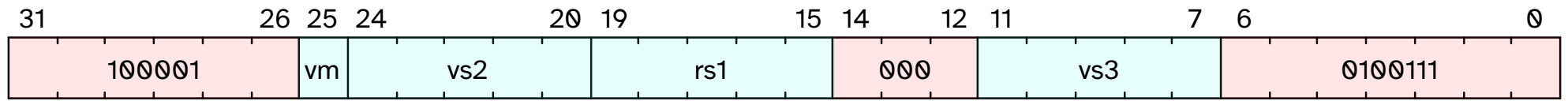
C.888. vsuxseg5ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.888.1. Encoding



C.888.2. Description

No description available.

C.888.3. Access

M	S	U
Always	Always	Always

C.888.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.888.5. IDL Operation

C.888.6. Exceptions

This instruction does not generate synchronous exceptions.

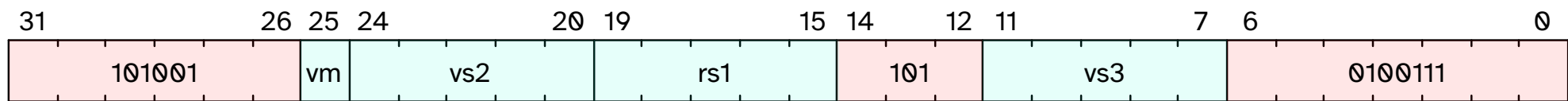
C.889. vsuxseg6ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.889.1. Encoding



C.889.2. Description

No description available.

C.889.3. Access

M	S	U
Always	Always	Always

C.889.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.889.5. IDL Operation

C.889.6. Exceptions

This instruction does not generate synchronous exceptions.

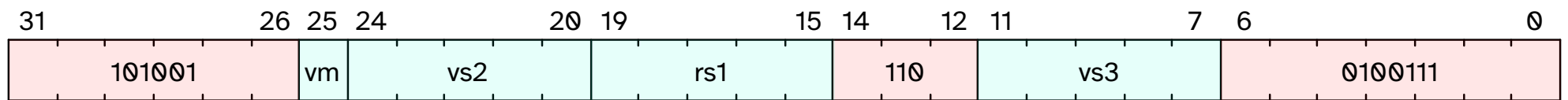
C.890. vsuxseg6ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.890.1. Encoding



C.890.2. Description

No description available.

C.890.3. Access

M	S	U
Always	Always	Always

C.890.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.890.5. IDL Operation

C.890.6. Exceptions

This instruction does not generate synchronous exceptions.

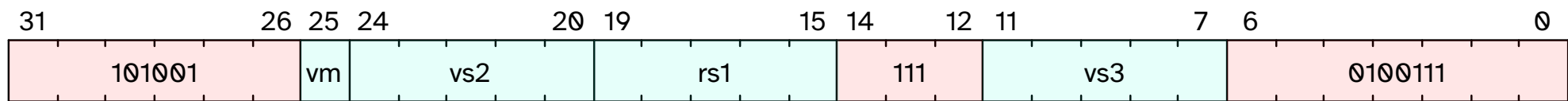
C.891. vsuxseg6ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.891.1. Encoding



C.891.2. Description

No description available.

C.891.3. Access

M	S	U
Always	Always	Always

C.891.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.891.5. IDL Operation

C.891.6. Exceptions

This instruction does not generate synchronous exceptions.

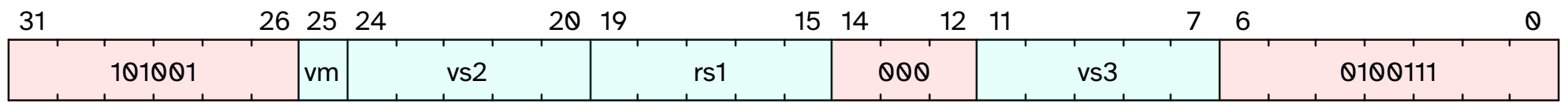
C.892. vsuxseg6ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.892.1. Encoding



C.892.2. Description

No description available.

C.892.3. Access

M	S	U
Always	Always	Always

C.892.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.892.5. IDL Operation

C.892.6. Exceptions

This instruction does not generate synchronous exceptions.

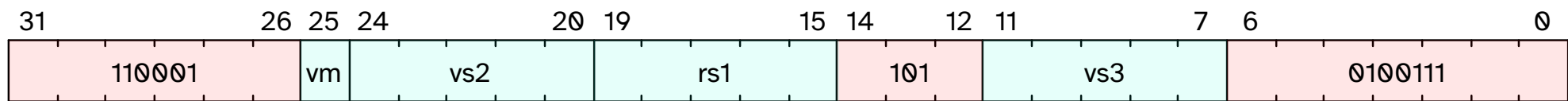
C.893. vsuxseg7ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.893.1. Encoding



C.893.2. Description

No description available.

C.893.3. Access

M	S	U
Always	Always	Always

C.893.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.893.5. IDL Operation

C.893.6. Exceptions

This instruction does not generate synchronous exceptions.

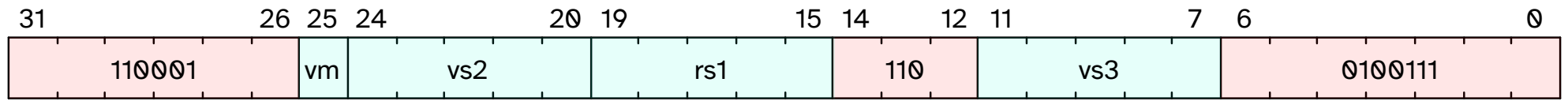
C.894. vsuxseg7ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.894.1. Encoding



C.894.2. Description

No description available.

C.894.3. Access

M	S	U
Always	Always	Always

C.894.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.894.5. IDL Operation

C.894.6. Exceptions

This instruction does not generate synchronous exceptions.

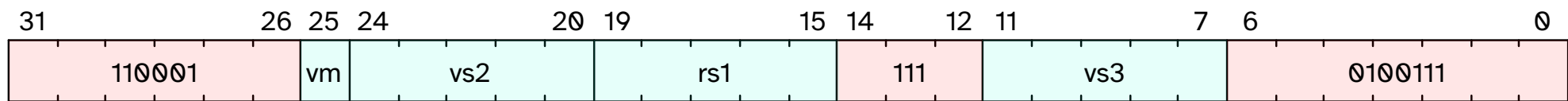
C.895. vsuxseg7ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.895.1. Encoding



C.895.2. Description

No description available.

C.895.3. Access

M	S	U
Always	Always	Always

C.895.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.895.5. IDL Operation

C.895.6. Exceptions

This instruction does not generate synchronous exceptions.

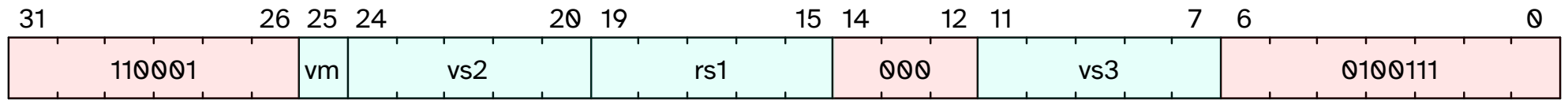
C.896. vsuxseg7ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.896.1. Encoding



C.896.2. Description

No description available.

C.896.3. Access

M	S	U
Always	Always	Always

C.896.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.896.5. IDL Operation

C.896.6. Exceptions

This instruction does not generate synchronous exceptions.

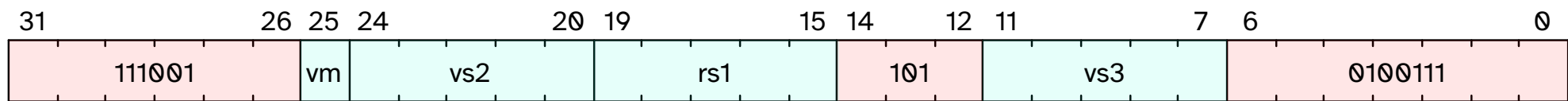
C.897. vsuxseg8ei16.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.897.1. Encoding



C.897.2. Description

No description available.

C.897.3. Access

M	S	U
Always	Always	Always

C.897.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.897.5. IDL Operation

C.897.6. Exceptions

This instruction does not generate synchronous exceptions.

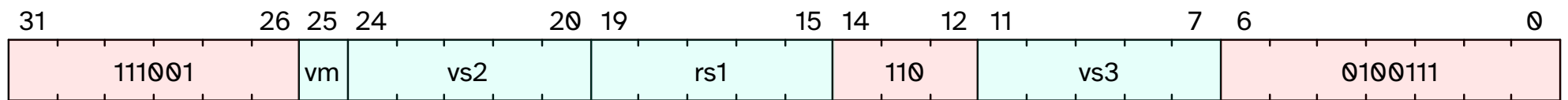
C.898. vsuxseg8ei32.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.898.1. Encoding



C.898.2. Description

No description available.

C.898.3. Access

M	S	U
Always	Always	Always

C.898.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.898.5. IDL Operation

C.898.6. Exceptions

This instruction does not generate synchronous exceptions.

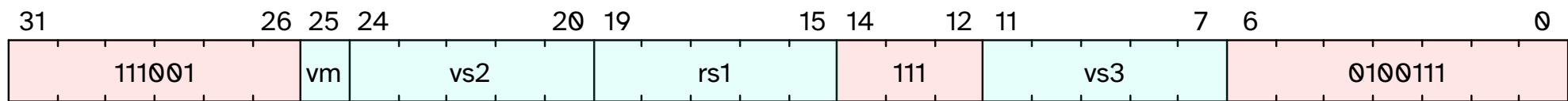
C.899. vsuxseg8ei64.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.899.1. Encoding



C.899.2. Description

No description available.

C.899.3. Access

M	S	U
Always	Always	Always

C.899.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.899.5. IDL Operation

C.899.6. Exceptions

This instruction does not generate synchronous exceptions.

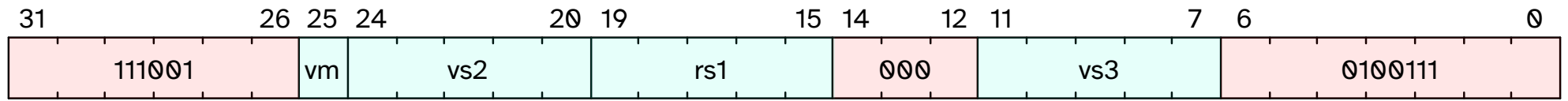
C.900. vsuxseg8ei8.v

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.900.1. Encoding



C.900.2. Description

No description available.

C.900.3. Access

M	S	U
Always	Always	Always

C.900.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vs3 = $encoding[11:7];
```

C.900.5. IDL Operation

C.900.6. Exceptions

This instruction does not generate synchronous exceptions.

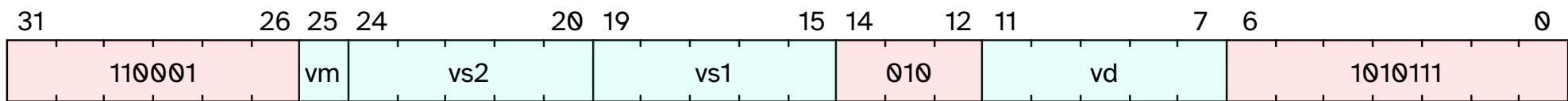
C.901. vwadd.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.901.1. Encoding



C.901.2. Description

No description available.

C.901.3. Access

M	S	U
Always	Always	Always

C.901.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.901.5. IDL Operation

C.901.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD  => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB  => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU  => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU  => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VWMUL    => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VWMULU   => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VWMULSU  => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.901.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

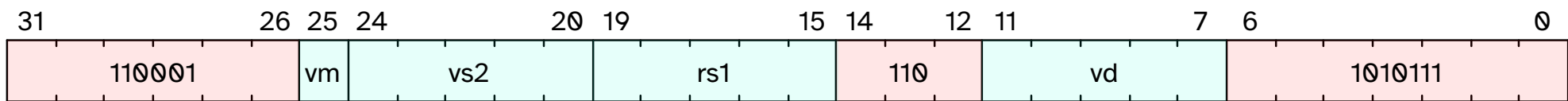
C.902. vwadd.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.902.1. Encoding



C.902.2. Description

No description available.

C.902.3. Access

M	S	U
Always	Always	Always

C.902.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.902.5. IDL Operation

C.902.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVX_VADD  => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        WVX_VSUB  => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        WVX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        WVX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        WVX_VVMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVX_VVMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVX_VVMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.902.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

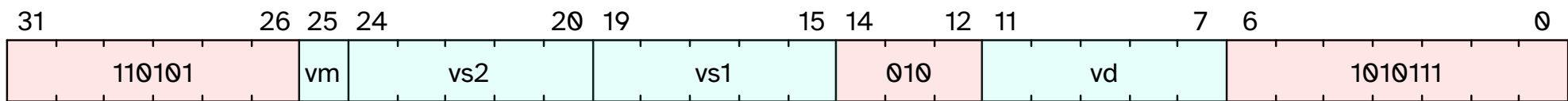
C.903. vwadd.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.903.1. Encoding



C.903.2. Description

No description available.

C.903.3. Access

M	S	U
Always	Always	Always

C.903.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.903.5. IDL Operation

C.903.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WV_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WV_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WV_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WV_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i]))
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.903.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

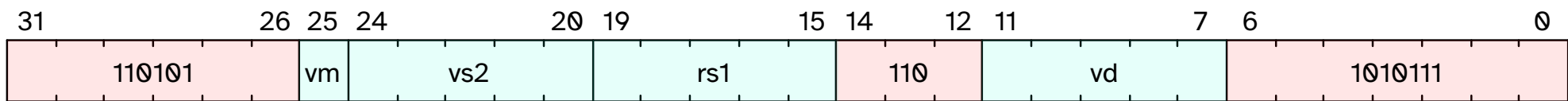
C.904. vwadd.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.904.1. Encoding



C.904.2. Description

No description available.

C.904.3. Access

M	S	U
Always	Always	Always

C.904.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.904.5. IDL Operation

C.904.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen)
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        WX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        WX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        WX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val))
      }
    }
  }
}
```

```
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.904.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

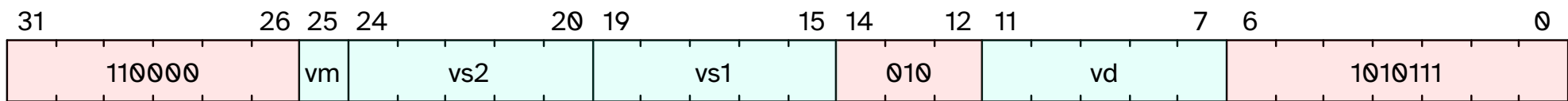
C.905. vwaddu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.905.1. Encoding



C.905.2. Description

No description available.

C.905.3. Access

M	S	U
Always	Always	Always

C.905.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.905.5. IDL Operation

C.905.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD  => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB  => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU  => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU  => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VWMUL    => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VWMULU   => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VWMULSU  => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.905.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

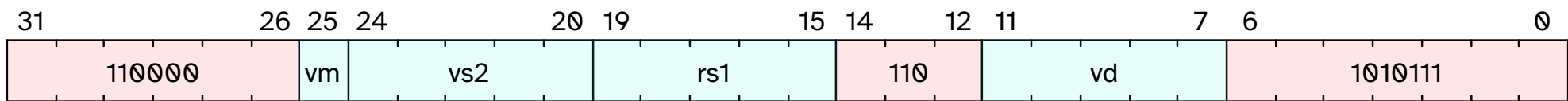
C.906. vwaddu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.906.1. Encoding



C.906.2. Description

No description available.

C.906.3. Access

M	S	U
Always	Always	Always

C.906.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.906.5. IDL Operation

C.906.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVX_VADD  => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        WVX_VSUB  => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        WVX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        WVX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        WVX_VVMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVX_VVMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVX_VVMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.906.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

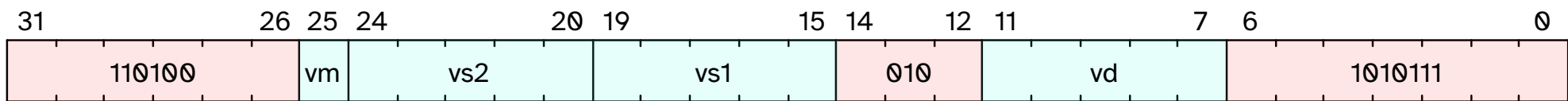
C.907. vwaddu.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.907.1. Encoding



C.907.2. Description

No description available.

C.907.3. Access

M	S	U
Always	Always	Always

C.907.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.907.5. IDL Operation

C.907.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WV_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WV_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WV_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WV_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i]))
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.907.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

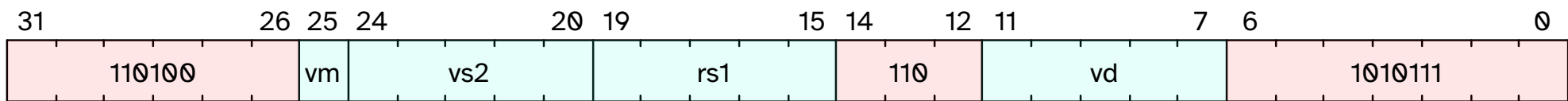
C.908. vwaddu.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.908.1. Encoding



C.908.2. Description

No description available.

C.908.3. Access

M	S	U
Always	Always	Always

C.908.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.908.5. IDL Operation

C.908.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen)
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                   = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        WX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        WX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        WX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val))
      }
    }
  }
}
```

```
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.908.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

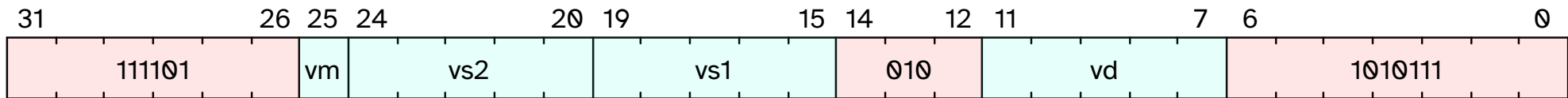
C.909. vwmacc.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.909.1. Encoding



C.909.2. Description

No description available.

C.909.3. Access

M	S	U
Always	Always	Always

C.909.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.909.5. IDL Operation

C.909.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVV_VWMACC  => to_bits(SEW_widen, signed(vs1_val[i]) * signed(vs2_val[i])) + vd_val[i],
        WMVV_VWMACCU => to_bits(SEW_widen, unsigned(vs1_val[i]) * unsigned(vs2_val[i])) + vd_val[i],
        WMVV_VWMACCSU => to_bits(SEW_widen, signed(vs1_val[i]) * unsigned(vs2_val[i]))+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.909.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

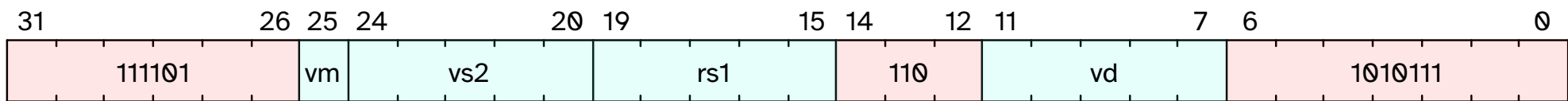
C.910. vwmacc.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.910.1. Encoding



C.910.2. Description

No description available.

C.910.3. Access

M	S	U
Always	Always	Always

C.910.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.910.5. IDL Operation

C.910.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVX_VWMAccU => (to_bits(SEW_widen, unsigned(rs1_val) * unsigned(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMAcc  => (to_bits(SEW_widen, signed(rs1_val) * signed(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMAccU => (to_bits(SEW_widen, unsigned(rs1_val) * signed(vs2_val[i])) )+ vd_val[i],
        WMVX_VWMAccSU => (to_bits(SEW_widen, signed(rs1_val) * unsigned(vs2_val[i])) )+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.910.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

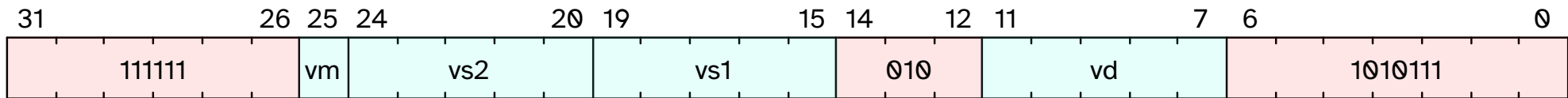
C.911. vwmaccsu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.911.1. Encoding



C.911.2. Description

No description available.

C.911.3. Access

M	S	U
Always	Always	Always

C.911.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.911.5. IDL Operation

C.911.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVV_VWMAcc    => to_bits(SEW_widen, signed(vs1_val[i]) * signed(vs2_val[i])) + vd_val[i],
        WMVV_VWMAccU   => to_bits(SEW_widen, unsigned(vs1_val[i]) * unsigned(vs2_val[i])) + vd_val[i],
        WMVV_VWMAccSU => to_bits(SEW_widen, signed(vs1_val[i]) * unsigned(vs2_val[i]))+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.911.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

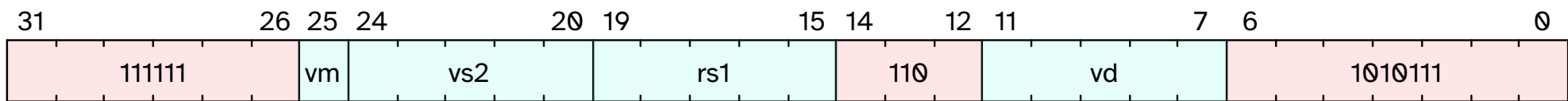
C.912. vwmaccsu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.912.1. Encoding



C.912.2. Description

No description available.

C.912.3. Access

M	S	U
Always	Always	Always

C.912.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.912.5. IDL Operation

C.912.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVX_VWMACCU => (to_bits(SEW_widen, unsigned(rs1_val) * unsigned(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMACC  => (to_bits(SEW_widen, signed(rs1_val) * signed(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMACCU => (to_bits(SEW_widen, unsigned(rs1_val) * signed(vs2_val[i])) )+ vd_val[i],
        WMVX_VWMACCSU => (to_bits(SEW_widen, signed(rs1_val) * unsigned(vs2_val[i])) )+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.912.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

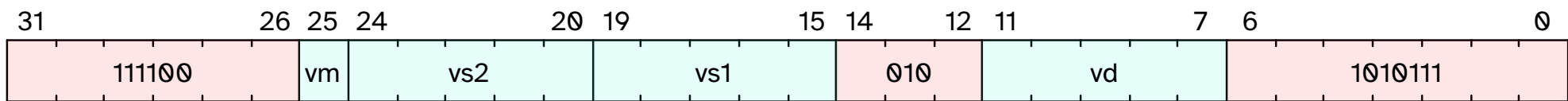
C.913. vwmaccu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.913.1. Encoding



C.913.2. Description

No description available.

C.913.3. Access

M	S	U
Always	Always	Always

C.913.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.913.5. IDL Operation

C.913.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVV_VWMAcc  => to_bits(SEW_widen, signed(vs1_val[i]) * signed(vs2_val[i])) + vd_val[i],
        WMVV_VWMAccU => to_bits(SEW_widen, unsigned(vs1_val[i]) * unsigned(vs2_val[i])) + vd_val[i],
        WMVV_VWMAccSU => to_bits(SEW_widen, signed(vs1_val[i]) * unsigned(vs2_val[i]))+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.913.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

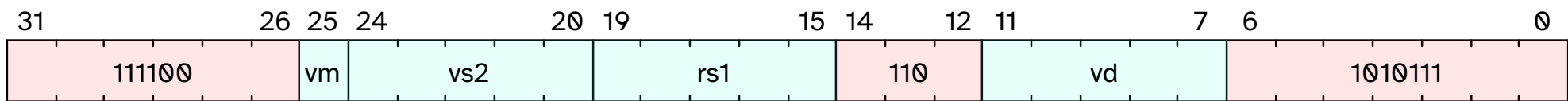
C.914. vwmaccu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.914.1. Encoding



C.914.2. Description

No description available.

C.914.3. Access

M	S	U
Always	Always	Always

C.914.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.914.5. IDL Operation

C.914.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVX_VWMACCU => (to_bits(SEW_widen, unsigned(rs1_val) * unsigned(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMACC  => (to_bits(SEW_widen, signed(rs1_val) * signed(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMACCU => (to_bits(SEW_widen, unsigned(rs1_val) * signed(vs2_val[i])) )+ vd_val[i],
        WMVX_VWMACCSU => (to_bits(SEW_widen, signed(rs1_val) * unsigned(vs2_val[i])) )+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.914.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

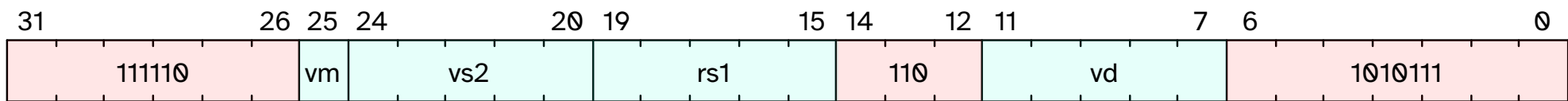
C.915. vwmaccus.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.915.1. Encoding



C.915.2. Description

No description available.

C.915.3. Access

M	S	U
Always	Always	Always

C.915.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.915.5. IDL Operation

C.915.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WMVX_VWMACCU => (to_bits(SEW_widen, unsigned(rs1_val) * unsigned(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMACC  => (to_bits(SEW_widen, signed(rs1_val) * signed(vs2_val[i])) ) + vd_val[i],
        WMVX_VWMACCU => (to_bits(SEW_widen, unsigned(rs1_val) * signed(vs2_val[i])) )+ vd_val[i],
        WMVX_VWMACCSU => (to_bits(SEW_widen, signed(rs1_val) * unsigned(vs2_val[i])) )+ vd_val[i]
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.915.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

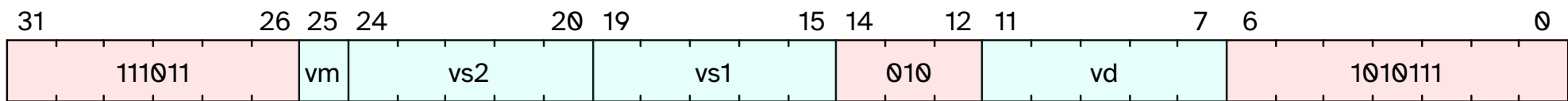
C.916. vwmul.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.916.1. Encoding



C.916.2. Description

No description available.

C.916.3. Access

M	S	U
Always	Always	Always

C.916.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.916.5. IDL Operation

C.916.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD    => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB    => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU   => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU   => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VWMUL    => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VWMULU   => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VWMULSU  => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.916.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

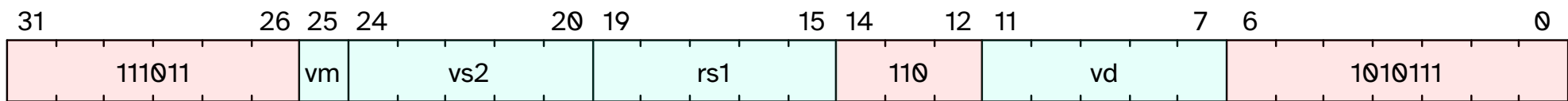
C.917. vwmul.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.917.1. Encoding



C.917.2. Description

No description available.

C.917.3. Access

M	S	U
Always	Always	Always

C.917.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.917.5. IDL Operation

C.917.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        W VX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        W VX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        W VX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        W VX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        W VX_VWMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVX_VWMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVX_VWMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.917.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

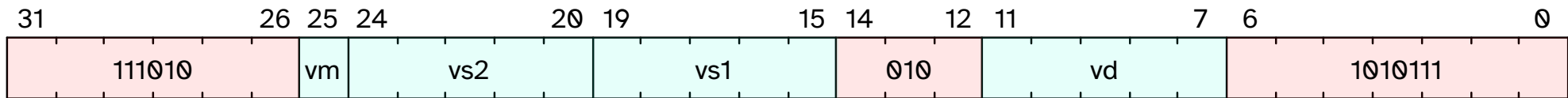
C.918. vwmulsu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.918.1. Encoding



C.918.2. Description

No description available.

C.918.3. Access

M	S	U
Always	Always	Always

C.918.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.918.5. IDL Operation

C.918.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD    => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB    => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU   => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU   => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VWMUL   => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VWMULU  => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VWMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.918.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

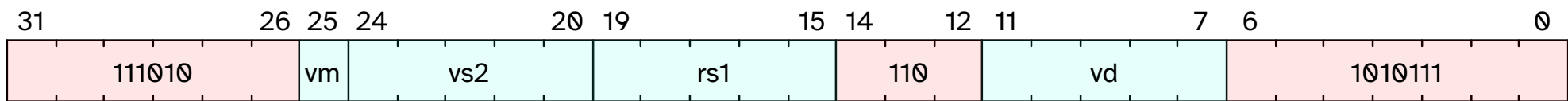
C.919. vwmulsu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.919.1. Encoding



C.919.2. Description

No description available.

C.919.3. Access

M	S	U
Always	Always	Always

C.919.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.919.5. IDL Operation

C.919.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        W VX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        W VX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        W VX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        W VX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        W VX_VWMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVX_VVMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVX_VWMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.919.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

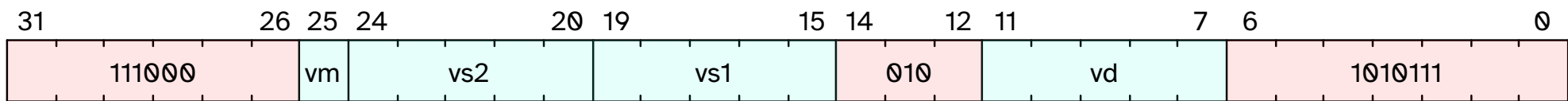
C.920. vwmulu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.920.1. Encoding



C.920.2. Description

No description available.

C.920.3. Access

M	S	U
Always	Always	Always

C.920.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.920.5. IDL Operation

C.920.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)    = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD    => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB    => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU   => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU   => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VVMUL   => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VVMULU  => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VVMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.920.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

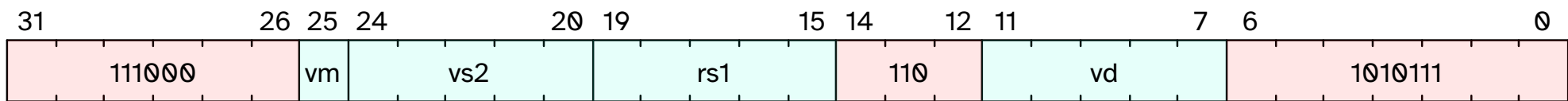
C.921. vwmulu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.921.1. Encoding



C.921.2. Description

No description available.

C.921.3. Access

M	S	U
Always	Always	Always

C.921.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.921.5. IDL Operation

C.921.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        W VX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        W VX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        W VX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        W VX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        W VX_VWMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.921.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

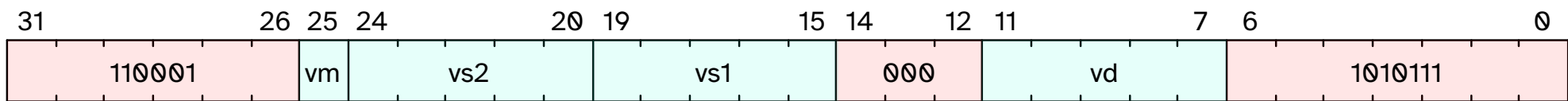
C.922. vwredsum.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.922.1. Encoding



C.922.2. Description

No description available.

C.922.3. Access

M	S	U
Always	Always	Always

C.922.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.922.5. IDL Operation

C.922.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW_widen); /* vd regardless of LMUL setting */

  if illegal_reduction_widen(SEW_widen, LMUL_pow_widen) then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val : vector('d, dec, bits('o)) = read_vreg(num_elem_vd, SEW_widen, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)      = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('o) = read_single_element(SEW_widen, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      let elem : bits('o) = match funct6 {
        IVV_VWREDSUMU => to_bits(SEW_widen, unsigned(vs2_val[i])),
        IVV_VWREDSUM  => to_bits(SEW_widen, signed(vs2_val[i]))
      };
      sum = sum + elem
    }
  }
};
```

```
write_single_element(SEW_widen, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.922.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

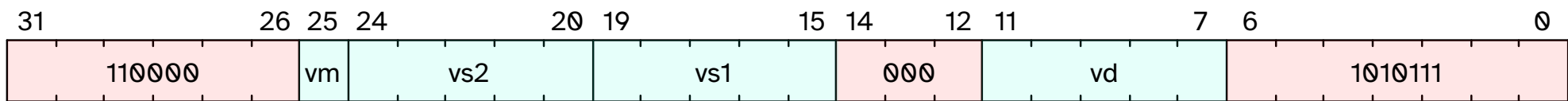
C.923. vwredsumu.vs

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.923.1. Encoding



C.923.2. Description

No description available.

C.923.3. Access

M	S	U
Always	Always	Always

C.923.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.923.5. IDL Operation

C.923.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let SEW_widen    = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;
  let num_elem_vs = get_num_elem(LMUL_pow, SEW);
  let num_elem_vd = get_num_elem(0, SEW_widen); /* vd regardless of LMUL setting */

  if illegal_reduction_widen(SEW_widen, LMUL_pow_widen) then { handle_illegal(); return RETIRE_FAIL };

  if unsigned(vl) == 0 then return RETIRE_SUCCESS; /* if vl=0, no operation is performed */

  let 'n = num_elem_vs;
  let 'd = num_elem_vd;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem_vs, vm, 0b000000);
  let vd_val  : vector('d, dec, bits('o)) = read_vreg(num_elem_vd, SEW_widen, 0, vd);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem_vs, SEW, LMUL_pow, vs2);
  let mask    : vector('n, dec, bool)    = init_masked_source(num_elem_vs, LMUL_pow, vm_val);

  sum : bits('o) = read_single_element(SEW_widen, 0, vs1); /* vs1 regardless of LMUL setting */
  foreach (i from 0 to (num_elem_vs - 1)) {
    if mask[i] then {
      let elem : bits('o) = match funct6 {
        IVV_VWREDSUMU => to_bits(SEW_widen, unsigned(vs2_val[i])),
        IVV_VWREDSUM  => to_bits(SEW_widen, signed(vs2_val[i]))
      };
      sum = sum + elem
    }
  }
};
```

```
write_single_element(SEW_widen, 0, vd, sum);  
/* other elements in vd are treated as tail elements, currently remain unchanged */  
/* TODO: configuration support for agnostic behavior */  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.923.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

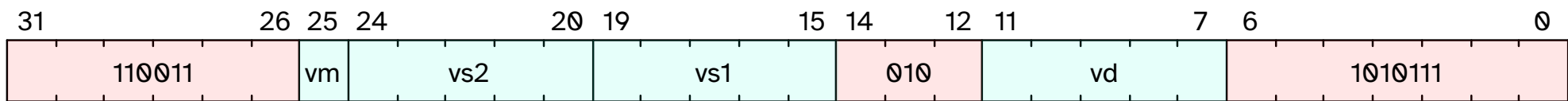
C.924. vwsb.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.924.1. Encoding



C.924.2. Description

No description available.

C.924.3. Access

M	S	U
Always	Always	Always

C.924.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.924.5. IDL Operation

C.924.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD    => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB    => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU    => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU    => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VWMUL   => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VWMULU  => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VWMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.924.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

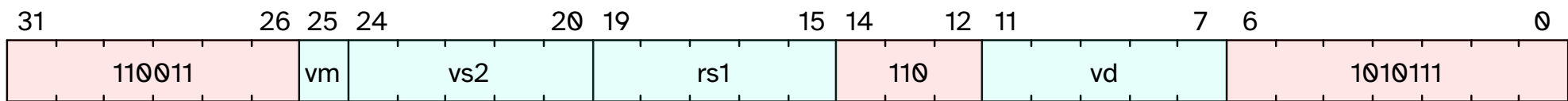
C.925. vwsb.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.925.1. Encoding



C.925.2. Description

No description available.

C.925.3. Access

M	S	U
Always	Always	Always

C.925.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.925.5. IDL Operation

C.925.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask       : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        W VX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        W VX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        W VX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        W VX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        W VX_VWMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVX_VVMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVX_VVMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.925.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

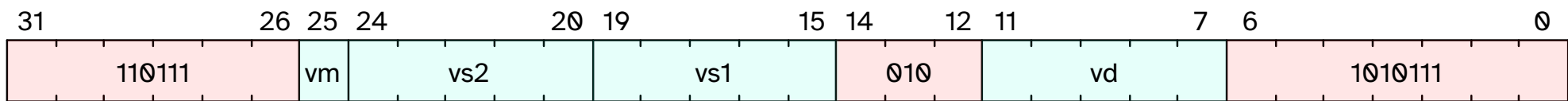
C.926. vwsb.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.926.1. Encoding



C.926.2. Description

No description available.

C.926.3. Access

M	S	U
Always	Always	Always

C.926.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.926.5. IDL Operation

C.926.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WV_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WV_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WV_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WV_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i]))
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.926.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

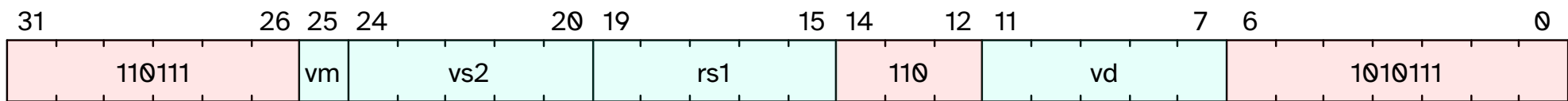
C.927. vwsb.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.927.1. Encoding



C.927.2. Description

No description available.

C.927.3. Access

M	S	U
Always	Always	Always

C.927.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.927.5. IDL Operation

C.927.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen)
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        WX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        WX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        WX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val))
      }
    }
  }
}
```

```
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.927.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

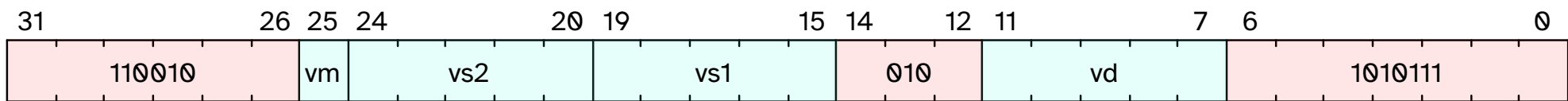
C.928. vwsubu.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.928.1. Encoding



C.928.2. Description

No description available.

C.928.3. Access

M	S	U
Always	Always	Always

C.928.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.928.5. IDL Operation

C.928.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen)) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WVV_VADD  => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WVV_VSUB  => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WVV_VADDU  => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WVV_VSUBU  => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i])),
```

```

        WVV_VWMUL    => to_bits(SEW_widen, signed(vs2_val[i]) * signed(vs1_val[i])),
        WVV_VWMULU   => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(vs1_val[i])),
        WVV_VWMULSU  => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(vs1_val[i]))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.928.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

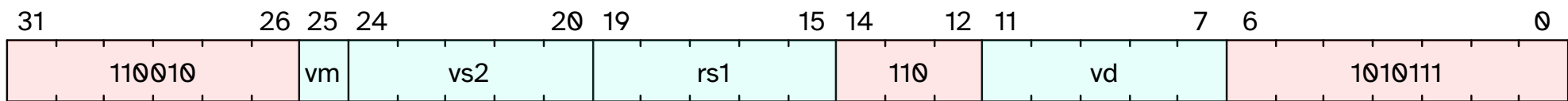
C.929. vwsubu.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.929.1. Encoding



C.929.2. Description

No description available.

C.929.3. Access

M	S	U
Always	Always	Always

C.929.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.929.5. IDL Operation

C.929.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        W VX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        W VX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        W VX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        W VX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val)),
        W VX_VVMUL => to_bits(SEW_widen, signed(vs2_val[i]) * signed(rs1_val)),
```

```

        WVX_VVMULU => to_bits(SEW_widen, unsigned(vs2_val[i]) * unsigned(rs1_val)),
        WVX_VVMULSU => to_bits(SEW_widen, signed(vs2_val[i]) * unsigned(rs1_val))
    }
}
};

write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.929.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

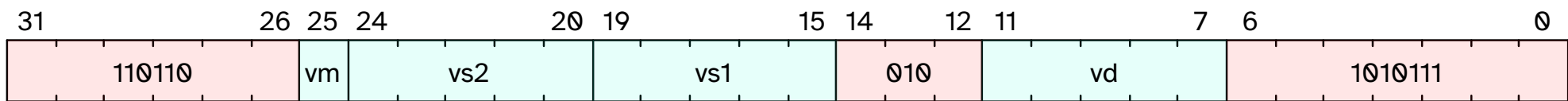
C.930. vwsbu.wv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.930.1. Encoding



C.930.2. Description

No description available.

C.930.3. Access

M	S	U
Always	Always	Always

C.930.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.930.5. IDL Operation

C.930.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen) |
    not(valid_reg_overlap(vs1, vd, LMUL_pow, LMUL_pow_widen))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  let result  : vector('n, dec, bits('o)) = undefined;
  let mask    : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WV_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(vs1_val[i])),
        WV_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(vs1_val[i])),
        WV_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(vs1_val[i])),
        WV_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(vs1_val[i]))
      }
    }
  }
}
```

```
    }  
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.930.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

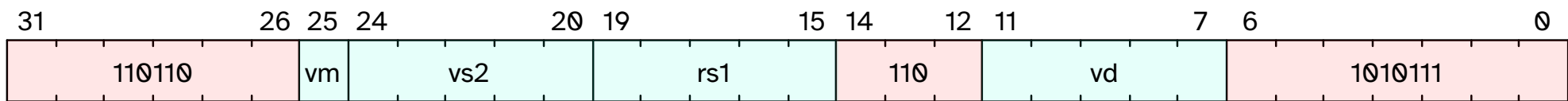
C.931. vwsbu.wx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.931.1. Encoding



C.931.2. Description

No description available.

C.931.3. Access

M	S	U
Always	Always	Always

C.931.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.931.5. IDL Operation

C.931.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmul_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_widen      = SEW * 2;
  let LMUL_pow_widen = LMUL_pow + 1;

  if illegal_variable_width(vd, vm, SEW_widen, LMUL_pow_widen)
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_widen;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd);
  let rs1_val : bits('m)                  = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_widen, LMUL_pow_widen, vs2);
  result      : vector('n, dec, bits('o)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW_widen, LMUL_pow_widen, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        WX_VADD => to_bits(SEW_widen, signed(vs2_val[i]) + signed(rs1_val)),
        WX_VSUB => to_bits(SEW_widen, signed(vs2_val[i]) - signed(rs1_val)),
        WX_VADDU => to_bits(SEW_widen, unsigned(vs2_val[i]) + unsigned(rs1_val)),
        WX_VSUBU => to_bits(SEW_widen, unsigned(vs2_val[i]) - unsigned(rs1_val))
      }
    }
  }
}
```

```
};  
  
write_vreg(num_elem, SEW_widen, LMUL_pow_widen, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.931.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

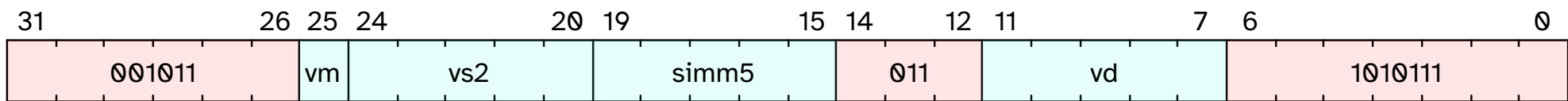
C.932. vxor.vi

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.932.1. Encoding



C.932.2. Description

No description available.

C.932.3. Access

M	S	U
Always	Always	Always

C.932.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> simm5 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.932.5. IDL Operation

C.932.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let imm_val : bits('m)                 = sign_extend(simm);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VI_VADD  => vs2_val[i] + imm_val,
        VI_VRSUB => imm_val - vs2_val[i],
        VI_VAND  => vs2_val[i] & imm_val,
        VI_VOR   => vs2_val[i] | imm_val,
        VI_VXOR  => vs2_val[i] ^ imm_val,
        VI_VSADDU => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, imm_val) ),
        VI_VSADD  => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, imm_val) ),
        VI_VSLL  => {
          let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
          vs2_val[i] << shift_amount
        }
      }
    }
  }
}
```

```

    },
    VI_VSRL    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        vs2_val[i] >> shift_amount
    },
    VI_VSRA    => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VI_VSSRL   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VI_VSSRA   => {
        let shift_amount = get_shift_amount(zero_extend('m, simm), SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.932.7. Exceptions

This instruction does not generate synchronous exceptions.

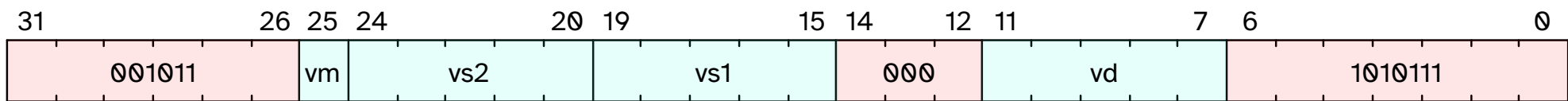
C.933. vxor.vv

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.933.1. Encoding



C.933.2. Description

No description available.

C.933.3. Access

M	S	U
Always	Always	Always

C.933.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.933.5. IDL Operation

C.933.6. Sail Operation

```
{
  let SEW_pow = get_sew_pow();
  let SEW     = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let VLEN_pow = get_vlen_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val : vector('n, dec, bool) = read_vmask(num_elem, vm, 0b000000);
  let vs1_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs1);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VV_VADD      => vs2_val[i] + vs1_val[i],
        VV_VSUB      => vs2_val[i] - vs1_val[i],
        VV_VAND      => vs2_val[i] & vs1_val[i],
        VV_VOR       => vs2_val[i] | vs1_val[i],
        VV_VXOR      => vs2_val[i] ^ vs1_val[i],
        VV_VSADDU    => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, vs1_val[i])),
        VV_VSADD     => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, vs1_val[i])),
        VV_VSSUBU    => {
```

```

        if unsigned(vs2_val[i]) < unsigned(vs1_val[i]) then zeros()
        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1,
vs1_val[i]))
    },
    VV_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, vs1_val[i])),
    VV_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(vs1_val[i]));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VV_VSLL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] << shift_amount
    },
    VV_VSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        vs2_val[i] >> shift_amount
    },
    VV_VSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VV_VSSRL => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VV_VSSRA => {
        let shift_amount = get_shift_amount(vs1_val[i], SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VV_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(vs1_val[i]))),
    VV_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(vs1_val[i]))),
    VV_VRGATHER => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        let idx = unsigned(vs1_val[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    },
    VV_VRGATHEREI16 => {
        if (vs1 == vd | vs2 == vd) then { handle_illegal(); return RETIRE_FAIL };
        /* vrgatherei16.vv uses SEW/LMUL for the data in vs2 but EEW=16 and EMUL = (16/SEW)*LMUL
for the indices in vs1 */
        let vs1_new : vector('n, dec, bits(16)) = read_vreg(num_elem, 16, 4 + LMUL_pow - SEW_pow,
vs1);

        let idx = unsigned(vs1_new[i]);
        let VLMAX = int_power(2, LMUL_pow + VLEN_pow - SEW_pow);
        assert(VLMAX <= 'n);
        if idx < VLMAX then vs2_val[idx] else zeros()
    }
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.933.7. Exceptions

This instruction does not generate synchronous exceptions.

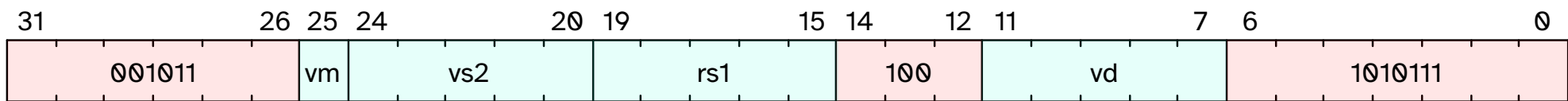
C.934. vxor.vx

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.934.1. Encoding



C.934.2. Description

No description available.

C.934.3. Access

M	S	U
Always	Always	Always

C.934.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> vd = $encoding[11:7];
```

C.934.5. IDL Operation

C.934.6. Sail Operation

```
{
  let SEW      = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);

  if illegal_normal(vd, vm) then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;

  let vm_val  : vector('n, dec, bool)    = read_vmask(num_elem, vm, 0b000000);
  let rs1_val : bits('m)                 = get_scalar(rs1, SEW);
  let vs2_val : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vs2);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VX_VADD    => vs2_val[i] + rs1_val,
        VX_VSUB    => vs2_val[i] - rs1_val,
        VX_VRSUB   => rs1_val - vs2_val[i],
        VX_VAND    => vs2_val[i] & rs1_val,
        VX_VOR     => vs2_val[i] | rs1_val,
        VX_VXOR    => vs2_val[i] ^ rs1_val,
        VX_VSADDU  => unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) + zero_extend('m + 1, rs1_val) ),
        VX_VSADD   => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) + sign_extend('m + 1, rs1_val) ),
        VX_VSSUBU  => {
          if unsigned(vs2_val[i]) < unsigned(rs1_val) then zeros()
```

```

        else unsigned_saturation('m, zero_extend('m + 1, vs2_val[i]) - zero_extend('m + 1, rs1_val) )
    },
    VX_VSSUB => signed_saturation('m, sign_extend('m + 1, vs2_val[i]) - sign_extend('m + 1, rs1_val) ),
    VX_VSMUL => {
        let result_mul = to_bits('m * 2, signed(vs2_val[i]) * signed(rs1_val));
        let rounding_incr = get_fixed_rounding_incr(result_mul, 'm - 1);
        let result_wide = (result_mul >> ('m - 1)) + zero_extend('m * 2, rounding_incr);
        signed_saturation('m, result_wide['m..0])
    },
    VX_VSLL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] << shift_amount
    },
    VX_VSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        vs2_val[i] >> shift_amount
    },
    VX_VSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW)
    },
    VX_VSSRL => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        (vs2_val[i] >> shift_amount) + zero_extend('m, rounding_incr)
    },
    VX_VSSRA => {
        let shift_amount = get_shift_amount(rs1_val, SEW);
        let rounding_incr = get_fixed_rounding_incr(vs2_val[i], shift_amount);
        let v_double : bits('m * 2) = sign_extend(vs2_val[i]);
        slice(v_double >> shift_amount, 0, SEW) + zero_extend('m, rounding_incr)
    },
    VX_VMINU => to_bits(SEW, min(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMIN => to_bits(SEW, min(signed(vs2_val[i]), signed(rs1_val))),
    VX_VMAXU => to_bits(SEW, max(unsigned(vs2_val[i]), unsigned(rs1_val))),
    VX_VMAX => to_bits(SEW, max(signed(vs2_val[i]), signed(rs1_val)))
}
};

write_vreg(num_elem, SEW, LMUL_pow, vd, result);
vstart = zeros();
RETIRE_SUCCESS
}

```

C.934.7. Exceptions

This instruction does not generate synchronous exceptions.

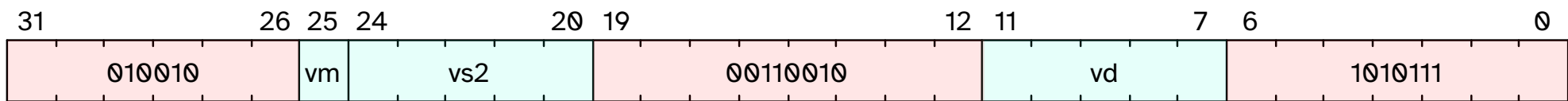
C.935. vzext.vf2

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.935.1. Encoding



C.935.2. Description

No description available.

C.935.3. Access

M	S	U
Always	Always	Always

C.935.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.935.5. IDL Operation

C.935.6. Sail Operation

```
{
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_half = SEW / 2;
  let LMUL_pow_half = LMUL_pow - 1;

  if illegal_variable_width(vd, vm, SEW_half, LMUL_pow_half) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_half, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_half;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_half, LMUL_pow_half, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW > SEW_half);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VEXT2_ZVF2 => zero_extend(vs2_val[i]),
        VEXT2_SVF2 => sign_extend(vs2_val[i])
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.935.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

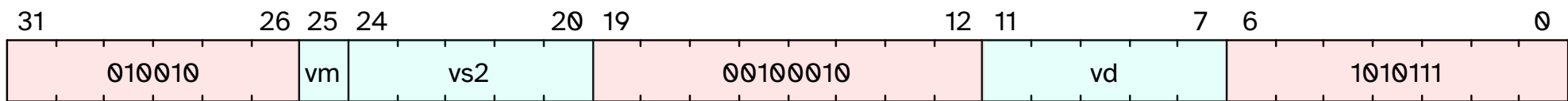
C.936. vzext.vf4

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.936.1. Encoding



C.936.2. Description

No description available.

C.936.3. Access

M	S	U
Always	Always	Always

C.936.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.936.5. IDL Operation

C.936.6. Sail Operation

```
{
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_quart = SEW / 4;
  let LMUL_pow_quart = LMUL_pow - 2;

  if illegal_variable_width(vd, vm, SEW_quart, LMUL_pow_quart) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_quart, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_quart;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_quart, LMUL_pow_quart, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)     = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW > SEW_quart);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VEXT4_ZVF4 => zero_extend(vs2_val[i]),
        VEXT4_SVF4 => sign_extend(vs2_val[i])
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.936.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

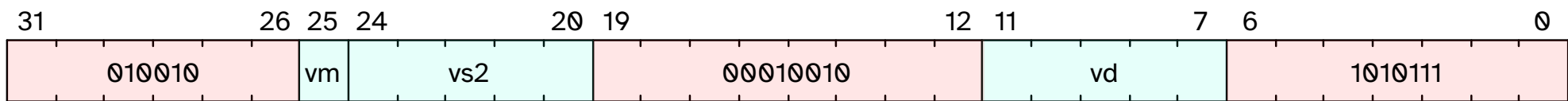
C.937. vzext.vf8

No synopsis available

This instruction is defined by:

- V, version >= V@1.0.0

C.937.1. Encoding



C.937.2. Description

No description available.

C.937.3. Access

M	S	U
Always	Always	Always

C.937.4. Decode Variables

```
Bits<1> vm = $encoding[25];
Bits<5> vs2 = $encoding[24:20];
Bits<5> vd = $encoding[11:7];
```

C.937.5. IDL Operation

C.937.6. Sail Operation

```
{
  let SEW = get_sew();
  let LMUL_pow = get_lmUL_pow();
  let num_elem = get_num_elem(LMUL_pow, SEW);
  let SEW_eighth = SEW / 8;
  let LMUL_pow_eighth = LMUL_pow - 3;

  if illegal_variable_width(vd, vm, SEW_eighth, LMUL_pow_eighth) |
    not(valid_reg_overlap(vs2, vd, LMUL_pow_eighth, LMUL_pow))
  then { handle_illegal(); return RETIRE_FAIL };

  let 'n = num_elem;
  let 'm = SEW;
  let 'o = SEW_eighth;

  let vm_val  : vector('n, dec, bool)      = read_vmask(num_elem, vm, 0b000000);
  let vd_val  : vector('n, dec, bits('m)) = read_vreg(num_elem, SEW, LMUL_pow, vd);
  let vs2_val : vector('n, dec, bits('o)) = read_vreg(num_elem, SEW_eighth, LMUL_pow_eighth, vs2);
  result      : vector('n, dec, bits('m)) = undefined;
  mask        : vector('n, dec, bool)      = undefined;

  (result, mask) = init_masked_result(num_elem, SEW, LMUL_pow, vd_val, vm_val);

  assert(SEW > SEW_eighth);
  foreach (i from 0 to (num_elem - 1)) {
    if mask[i] then {
      result[i] = match funct6 {
        VEXT8_ZVF8 => zero_extend(vs2_val[i]),
        VEXT8_SVF8 => sign_extend(vs2_val[i])
      }
    }
  }
};
```

```
write_vreg(num_elem, SEW, LMUL_pow, vd, result);  
vstart = zeros();  
RETIRE_SUCCESS  
}
```

C.937.7. Exceptions

This instruction does not generate synchronous exceptions.

DRAFT

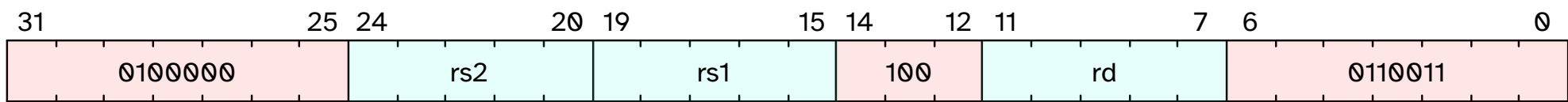
C.938. xnor

Exclusive NOR

This instruction is defined by:

- anyOf:
 - Zbb, version >= Zbb@1.0.0
 - Zbkb, version >= Zbkb@1.0.0

C.938.1. Encoding



C.938.2. Description

This instruction performs the bit-wise exclusive-NOR operation on rs1 and rs2.

C.938.3. Access

M	S	U
Always	Always	Always

C.938.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.938.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = ~(X[rs1] ^ X[rs2]);
```

C.938.6. 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.938.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

DRAFT

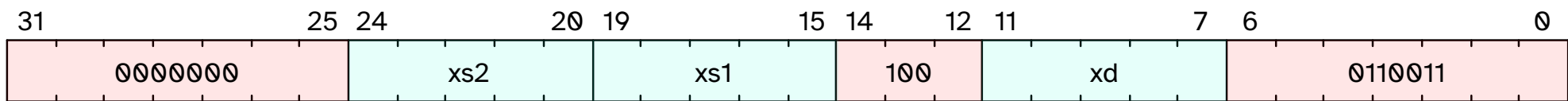
C.939. xor

Exclusive Or

This instruction is defined by:

- I, version >= I@2.1.0

C.939.1. Encoding



C.939.2. Description

Exclusive or xs1 with xs2, and store the result in xd

C.939.3. Access

M	S	U
Always	Always	Always

C.939.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.939.5. IDL Operation

```
X[xd] = X[xs1] ^ X[xs2];
```

C.939.6. 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.939.7. Exceptions

This instruction does not generate synchronous exceptions.

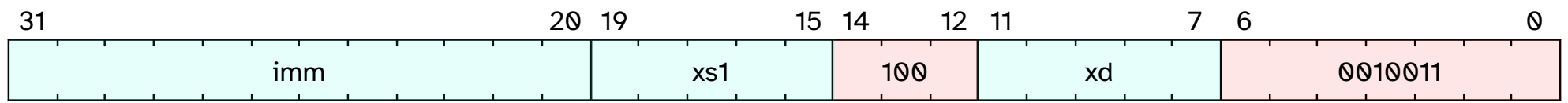
C.940. xori

Exclusive Or immediate

This instruction is defined by:

- I, version >= I@2.1.0

C.940.1. Encoding



C.940.2. Description

Exclusive or an immediate to the value in xs1, and store the result in xd

C.940.3. Access

M	S	U
Always	Always	Always

C.940.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

C.940.5. IDL Operation

```
X[xd] = X[xs1] ^ $signed(imm);
```

C.940.6. 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.940.7. Exceptions

This instruction does not generate synchronous exceptions.

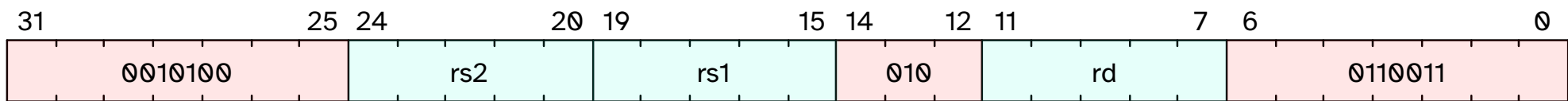
C.941. xperm4

Crossbar permutation (nibbles)

This instruction is defined by:

- Zbkx, version >= Zbkx@1.0.0

C.941.1. Encoding



C.941.2. Description

The xperm4 instruction operates on nibbles. The rs1 register contains a vector of XLEN/4 4-bit elements. The rs2 register contains a vector of XLEN/4 4-bit indexes. The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index into rs2 is out of bounds.

C.941.3. Access

M	S	U
Always	Always	Always

C.941.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.941.5. IDL Operation

```
XReg input1 = X[rs1];
XReg input2 = X[rs2];
XReg output = 0;
for (U32 i = 0; i < (xlen() - 4); i = i + 4) {
    XReg index = input2[i + 3:i];
    if (4 * index < xlen()) {
        output[i + 3:i] = input1[4 * index + 3:4 * index];
    }
}
X[rd] = output;
```

C.941.6. Sail Operation

```
val xperm4_lookup : (bits(4), xlenbits) -> bits(4)
function xperm4_lookup (idx, lut) = {
    (lut >> (idx @ 0b00))[3..0]
}
function clause execute ( XPERM_4 (rs2,rs1,rd)) = {
    result : xlenbits = EXTZ(0b0);
    foreach(i from 0 to xlen by 4) {
        result[i+3..i] = xperm4_lookup(X(rs2)[i+3..i], X(rs1));
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

C.941.7. Exceptions

This instruction does not generate synchronous exceptions.

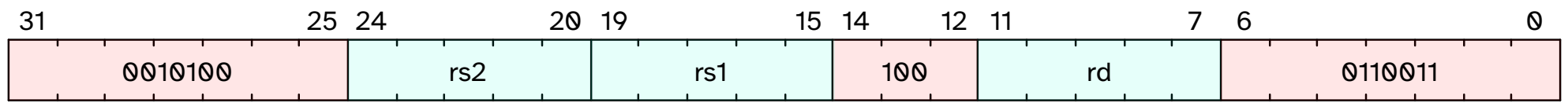
C.942. xperm8

Crossbar permutation (bytes)

This instruction is defined by:

- Zbkx, version >= Zbkx@1.0.0

C.942.1. Encoding



C.942.2. Description

The xperm8 instruction operates on bytes. The rs1 register contains a vector of XLEN/8 8-bit elements. The rs2 register contains a vector of XLEN/8 8-bit indexes. The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index into rs2 is out of bounds.

C.942.3. Access

M	S	U
Always	Always	Always

C.942.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.942.5. IDL Operation

```
XReg input1 = X[rs1];
XReg input2 = X[rs2];
XReg output = 0;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
    XReg index = input2[i + 7:i];
    if (8 * index < xlen()) {
        output[i + 7:i] = input1[8 * index + 7:8 * index];
    }
}
X[rd] = output;
```

C.942.6. Sail Operation

```
val xperm8_lookup : (bits(8), xlenbits) -> bits(8)
function xperm8_lookup (idx, lut) = {
    (lut >> (idx @ 0b00))[7..0]
}
function clause execute ( XPERM_8 (rs2,rs1,rd)) = {
    result : xlenbits = EXTZ(0b0);
    foreach(i from 0 to xlen by 8) {
        result[i+7..i] = xperm8_lookup(X(rs2)[i+7..i], X(rs1));
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

C.942.7. Exceptions

This instruction does not generate synchronous exceptions.

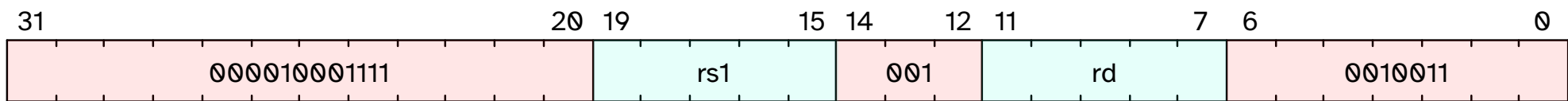
C.943. zip

Bit interleave

This instruction is defined by:

- Zbkb, version >= Zbkb@1.0.0

C.943.1. Encoding



C.943.2. Description

This instruction scatters all of the odd and even bits of a source word into the high and low halves of a destination word. It is the inverse of the unzip instruction. This instruction is available only on RV32.

C.943.3. Access

M	S	U
Always	Always	Always

C.943.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

C.943.5. IDL Operation

```
XReg input = X[rs1];
XReg output = 0;
for (U32 i = 0; i < (xlen() / 2 - 1); i = i + 1) {
    output[2 * i] = input[i];
    output[2 * i + 1] = input[i + xlen() / 2];
}
X[rd] = output;
```

C.943.6. Sail Operation

```
foreach (i from 0 to xlen/2-1) {
    X(rd)[2*i] = X(rs1)[i];
    X(rd)[2*i+1] = X(rs1)[i+xlen/2];
}
```

C.943.7. Exceptions

This instruction does not generate synchronous exceptions.

Appendix D: CSR Details

DRAFT

D.1. cycle

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR [mcycle](#).

Privilege mode access is controlled with [mcounteren.CY](#), [scounteren.CY](#), and [hcounteren.CY](#) as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

D.1.1. Attributes

CSR Address	0xc00
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

D.1.2. Format

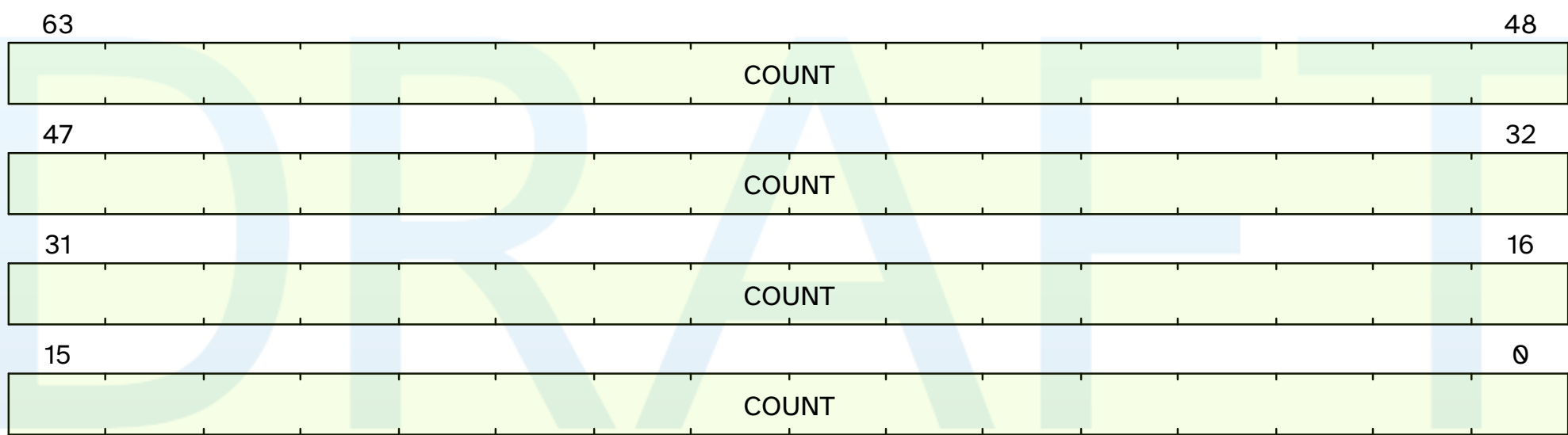


Figure 1. cycle format

D.1.3. Field Summary

Nam e	Location	Type	Reset Value
cycle.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.1.4. Fields

[cycle.COUNT](#) Field

Location:
63:0

Description:
Alias of [mcycle.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.1.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```

if (mode() == PrivilegeMode::S) {
  if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.CY & scounteren.CY) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.CY == 1'b0 && mcounteren.CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.CY & scounteren.CY == 1'b0) && (mcounteren.CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle();

```

DRAFT

D.2. cycleh

High-half cycle counter for RDCYCLE Instruction

i

cycleh

 is only defined in RV32.

Alias for M-mode CSR [mcycleh](#).

Privilege mode access is controlled with [mcounteren.CY](#), [scounteren.CY](#), and [hcounteren.CY](#) as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

D.2.1. Attributes

CSR Address	0xc80
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

D.2.2. Format

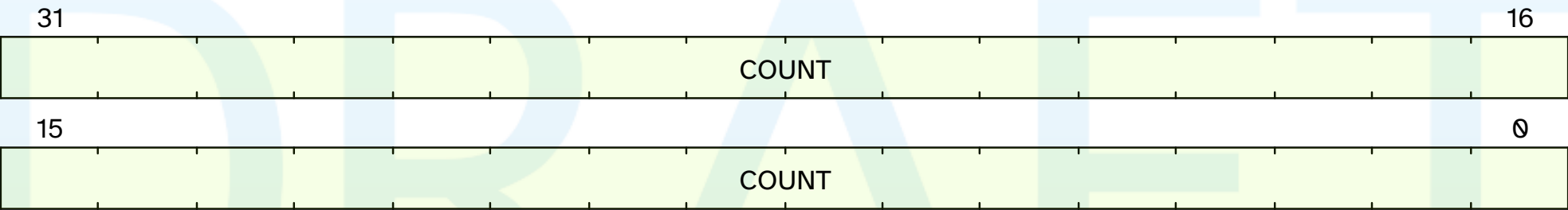


Figure 2. cycleh format

D.2.3. Field Summary

Name	Location	Type	Reset Value
cycleh .COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.2.4. Fields

cycleh.COUNT Field

Location:

31:0

Description:

Alias of [mcycleh.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.2.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.CY == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.CY & scounteren.CY) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.CY == 1'b0 && mcounteren.CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.CY & scounteren.CY) == 1'b0) && (mcounteren.CY == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
}
return read_mcycle()[63:32];

```

DRAFT

D.3. fcsr

Floating-point control and status register (frm + fflags)

The floating-point control and status register, [fcsr](#), is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in [Floating-Point Control and Status Register](#).

Floating-point control and status register

Unresolved directive in 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 16. Accrued exception flag encoding.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact



As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

D.3.1. Attributes

CSR Address	0x3
Defining extension	F, version >= F@2.2.0
Length	32-bit
Privilege Mode	U

D.3.2. Format

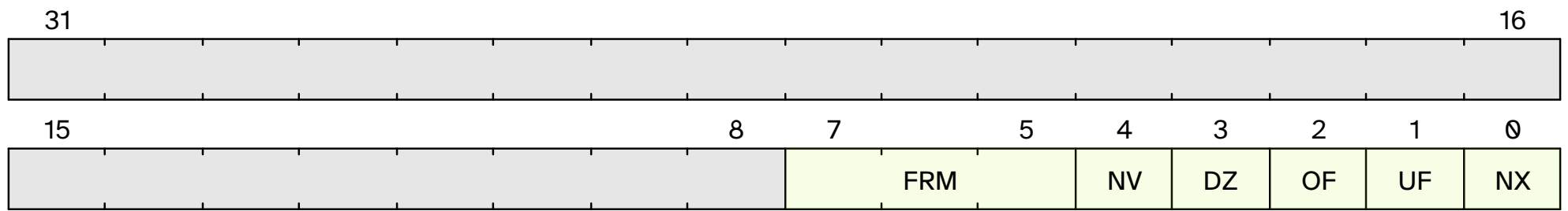


Figure 3. fcsr format

D.3.3. Field Summary

Na me	Location	Type	Reset Value
fcsr. FRM	7:5	RW-H	UNDEFINED_LEGAL
fcsr. NV	4	RW-H	UNDEFINED_LEGAL
fcsr. DZ	3	RW-H	UNDEFINED_LEGAL
fcsr. OF	2	RW-H	UNDEFINED_LEGAL
fcsr. UF	1	RW-H	UNDEFINED_LEGAL
fcsr. NX	0	RW-H	UNDEFINED_LEGAL

D.3.4. Fields

fcsr.FRM Field

Location:

7:5

Description:

Rounding modes are encoded as follows:

.Rounding mode encoding.

[%autowidth,float="center",align="center",cols="",<,"options="header"]

!===

!Rounding Mode |Mnemonic |Meaning

!000 !RNE !Round to Nearest, ties to Even

!001 !RTZ !Round towards Zero

!010 !RDN !Round Down (towards -\infty)

!011 !RUP !Round Up (towards +\infty)

!100 !RMM !Round to Nearest, ties to Max Magnitude

!101 !Reserved for future use.

!110 !Reserved for future use.

!111 !DYN !In instruction's rm field, selects dynamic rounding mode; In Rounding Mode register, reserved.

!===

A value of 111 in the instruction's rm field selects the dynamic rounding mode held in [frm](#). The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is reserved, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111). Some instructions, including widening conversions, have the rm field but are nevertheless mathematically unaffected by the rounding mode; software should set their rm field to RNE (000) but implementations must treat the rm field as usual (in particular, with regard to decoding legal vs. reserved encodings).

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.NV Field

Location:

4

Description:

Invalid Operation

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation is invalid and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.DZ Field

Location:

3

Description:

Divide by zero

Cumulative error flag for floating point operations.

Set by hardware when a floating point divide attempts to divide by zero and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.OF Field

Location:

2

Description:

Overflow

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation overflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.UF Field

Location:

1

Description:

Underflow

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation underflows and stays set until explicitly

cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.NX Field

Location:

0

Description:

Inexact

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation is inexact and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

DRAFT

D.4. fflags

Floating-Point Accrued Exceptions

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software.

The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

D.4.1. Attributes

CSR Address	0x1
Defining extension	F, version >= F@2.2.0
Length	32-bit
Privilege Mode	U

D.4.2. Format

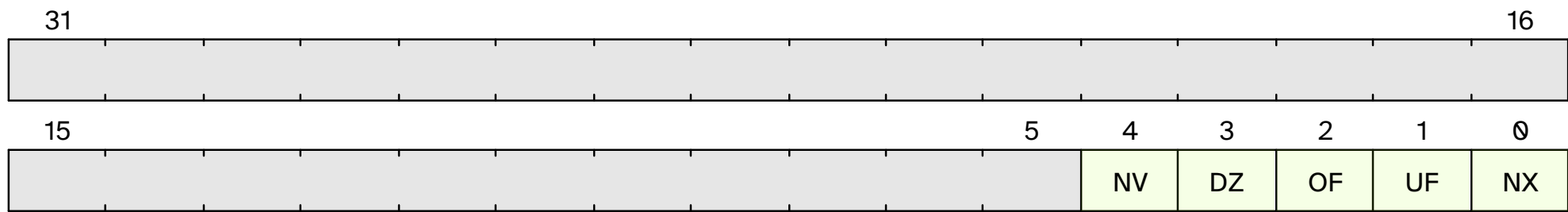


Figure 4. fflags format

D.4.3. Field Summary

Na me	Location	Type	Reset Value
fflag s.NV	4	RW-H	UNDEFINED_LEGAL
fflag s.DZ	3	RW-H	UNDEFINED_LEGAL
fflag s.OF	2	RW-H	UNDEFINED_LEGAL
fflag s.UF	1	RW-H	UNDEFINED_LEGAL
fflag s.NX	0	RW-H	UNDEFINED_LEGAL

D.4.4. Fields

[fflags.NV](#) Field

Location:

4

Description:

Set by hardware when a floating point operation is invalid and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.DZ Field

Location:

3

Description:

Set by hardware when a floating point divide attempts to divide by zero and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.OF Field

Location:

2

Description:

Set by hardware when a floating point operation overflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.UF Field

Location:

1

Description:

Set by hardware when a floating point operation underflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.NX Field

Location:

0

Description:

Set by hardware when a floating point operation is inexact and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

D.4.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
NV = CSR[fcsr].NV = csr_value.NV;
return csr_value.NV;

DZ = CSR[fcsr].DZ = csr_value.DZ;
return csr_value.DZ;

OF = CSR[fcsr].OF = csr_value.OF;
return csr_value.OF;

UF = CSR[fcsr].UF = csr_value.UF;
return csr_value.UF;

NX = CSR[fcsr].NX = csr_value.NX;
return csr_value.NX;
```

D.4.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return (fcsr.NV `<< 4) | (fcsr.DZ `<< 3) | (fcsr.OF `<< 2) | (fcsr.UF `<< 1) | fcsr.NX;
```

DRAFT

D.5. frm

Floating-Point Dynamic Rounding Mode

Rounding modes are encoded as follows:

Table 17. 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 <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, reserved.

The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111).

Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

D.5.1. Attributes

CSR Address	0x2
Defining extension	F, version $\geq$ F@2.2.0
Length	32-bit
Privilege Mode	U

D.5.2. Format

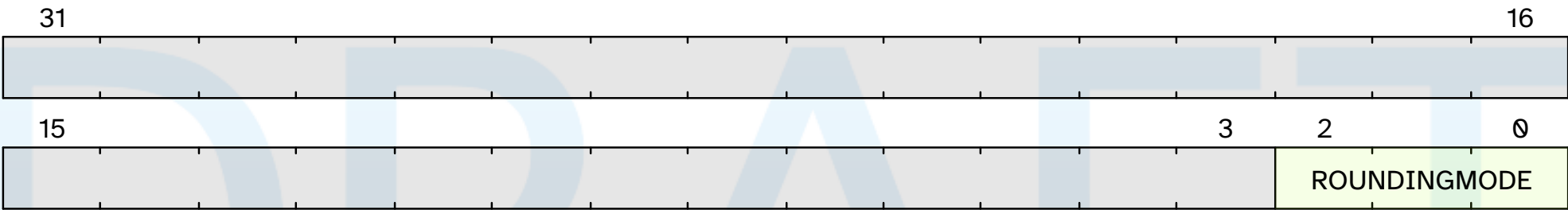


Figure 5. *frm* format

D.5.3. Field Summary

Name	Location	Type	Reset Value
<code>frm.ROUNDINGMODE</code>	2:0	RW-H	UNDEFINED_LEGAL

D.5.4. Fields

`frm.ROUNDINGMODE` Field

Location:

2:0

Description:

Rounding mode data.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

D.5.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through `csrww`), the following determines the written value:

```
ROUNDINGMODE = CSR[fcsr].FRM = csr_value.ROUNDINGMODE;
return csr_value.ROUNDINGMODE;
```

D.5.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return fcsr.FRM;
```

DRAFT

D.6. hcounteren

Hypervisor Counter Enable

Together with [scounteren](#), delegates control of the hardware performance-monitoring counters to VS/VU-mode

See [cycle](#) for a table describing how exceptions occur.

D.6.1. Attributes

CSR Address	0x606
Defining extension	H, version >= H@1.0.0
Length	32-bit
Privilege Mode	S

D.6.2. Format

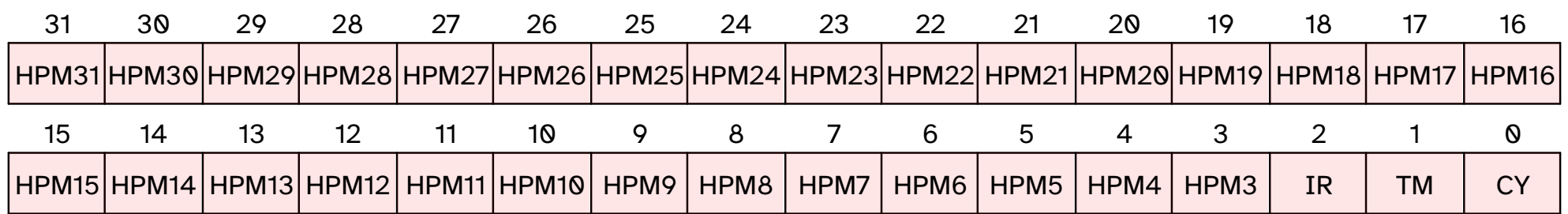


Figure 6. hcounteren format

D.6.3. Field Summary

Name	Location	Type	Reset Value
hcounteren.CY	0	RW	UNDEFINED_LEGAL
		RO	
hcounteren.TM	1	RW	UNDEFINED_LEGAL
		RO	
hcounteren.IR	2	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM3	3	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM4	4	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM5	5	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM6	6	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM7	7	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM8	8	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM9	9	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM10	10	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM11	11	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM12	12	RW	UNDEFINED_LEGAL
		RO	

Name	Location	Type	Reset Value
hcounteren.HPM13	13	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM14	14	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM15	15	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM16	16	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM17	17	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM18	18	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM19	19	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM20	20	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM21	21	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM22	22	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM23	23	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM24	24	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM25	25	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM26	26	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM27	27	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM28	28	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM29	29	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM30	30	RW	UNDEFINED_LEGAL
		RO	
hcounteren.HPM31	31	RW	UNDEFINED_LEGAL
		RO	

D.6.4. Fields

hcounteren.CY Field

Location:

0

Description:

When all of *scounteren.CY*, *mcouteren.CY*, and *hcounteren.CY* are set, the *cycle* CSR (an alias of *mcycle*) is accessible to VU-mode.

When *mcounteren.CY* and *hcounteren.CY* are set,
the *cycle* CSR (an alias of *mcycle*) is accessible to VS-mode.

When *hcounteren.CY* is clear and *mcounteren.CY* is set, then any access to *cycle* in
VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",cols="1,1,1,4,4"]
!===
.2+h! hcounteren.CY .2+h! mcounteren.CY .2+h! scounteren.CY 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.TM Field

Location:

1

Description:

When all of *scounteren.TM*, *mcounteren.TM*, and *hcounteren.TM* are set,
the *time* CSR (an alias of *mtime* memory-mapped CSR) is accessible to VU-mode.

When *mcounteren.TM* and *hcounteren.TM* are set,
the *time* CSR (an alias of *mtime*) is accessible to VS-mode.

When *hcounteren.TM* is clear and *mcounteren.TM* is set, then any access to *time* in
VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.TM .2+h! mcounteren.TM .2+h! scounteren.TM 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.IR Field

Location:

2

Description:

When all of *scounteren.IR*, *mcounteren.IR*, and *hcounteren.IR* are set, the *instret* CSR (an alias of *minstret*) is accessible to VU-mode.

When *mcounteren.IR* and *hcounteren.IR* are set, the *instret* CSR (an alias of *minstret*) is accessible to VS-mode.

When *hcounteren.IR* is clear and *mcounteren.IR* is set, then any access to *instret* in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.IR .2+h! mcounteren.IR .2+h! scounteren.IR 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM3 Field

Location:

3

Description:

When all of *scounteren.HPM3*, *mcounteren.HPM3*, and *hcounteren.HPM3* are set, the *hpmcounter3* CSR (an alias of *mhpmcounter3*) is accessible to VU-mode.

When *mcounteren.HPM3* and *hcounteren.HPM3* are set, the *hpmcounter3* CSR (an alias of *mhpmcounter3*) is accessible to VS-mode.

When *hcounteren.HPM3* is clear and *mcounteren.HPM3* is set, then any access to *hpmcounter3* in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM3 .2+h! mcounteren.HPM3 .2+h! scounteren.HPM3 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM4 Field

Location:

4

Description:

When all of [scounteren.HPM4](#), [mcounteren.HPM4](#), and [hcounteren.HPM4](#) are set, the [hpmcounter4](#) CSR (an alias of [mhpmcounter4](#)) is accessible to VU-mode.

When [mcounteren.HPM4](#) and [hcounteren.HPM4](#) are set, the [hpmcounter4](#) CSR (an alias of [mhpmcounter4](#)) is accessible to VS-mode.

When [hcounteren.HPM4](#) is clear and [mcounteren.HPM4](#) is set, then any access to [hpmcounter4](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM4 .2+h! mcounteren.HPM4 .2+h! scounteren.HPM4 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

[hcounteren.HPM5](#) Field

Location:

5

Description:

When all of [scounteren.HPM5](#), [mcounteren.HPM5](#), and [hcounteren.HPM5](#) are set, the [hpmcounter5](#) CSR (an alias of [mhpmcounter5](#)) is accessible to VU-mode.

When [mcounteren.HPM5](#) and [hcounteren.HPM5](#) are set, the [hpmcounter5](#) CSR (an alias of [mhpmcounter5](#)) is accessible to VS-mode.

When [hcounteren.HPM5](#) is clear and [mcounteren.HPM5](#) is set, then any access to [hpmcounter5](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM5 .2+h! mcounteren.HPM5 .2+h! scounteren.HPM5 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM6 Field

Location:

6

Description:

When all of [scounteren.HPM6](#), [mcounteren.HPM6](#), and [hcounteren.HPM6](#) are set, the [hpmcounter6](#) CSR (an alias of [mhpmcounter6](#)) is accessible to VU-mode.

When [mcounteren.HPM6](#) and [hcounteren.HPM6](#) are set, the [hpmcounter6](#) CSR (an alias of [mhpmcounter6](#)) is accessible to VS-mode.

When [hcounteren.HPM6](#) is clear and [mcounteren.HPM6](#) is set, then any access to [hpmcounter6](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM6 .2+h! mcounteren.HPM6 .2+h! scounteren.HPM6 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM7 Field

Location:

7

Description:

When all of [scounteren.HPM7](#), [mcounteren.HPM7](#), and [hcounteren.HPM7](#) are set, the [hpmcounter7](#) CSR (an alias of [mhpmcounter7](#)) is accessible to VU-mode.

When [mcounteren.HPM7](#) and [hcounteren.HPM7](#) are set, the [hpmcounter7](#) CSR (an alias of [mhpmcounter7](#)) is accessible to VS-mode.

When [hcounteren.HPM7](#) is clear and [mcounteren.HPM7](#) is set, then any access to [hpmcounter7](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM7 .2+h! mcounteren.HPM7 .2+h! scounteren.HPM7 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

hcounteren.HPM8 Field

Location:
8

Description:
When all of [scounteren.HPM8](#), [mcounteren.HPM8](#), and [hcounteren.HPM8](#) are set, the [hpmcounter8](#) CSR (an alias of [mhpmcounter8](#)) is accessible to VU-mode.

When [mcounteren.HPM8](#) and [hcounteren.HPM8](#) are set, the [hpmcounter8](#) CSR (an alias of [mhpmcounter8](#)) is accessible to VS-mode.

When [hcounteren.HPM8](#) is clear and [mcounteren.HPM8](#) is set, then any access to [hpmcounter8](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM8 .2+h! mcounteren.HPM8 .2+h! scounteren.HPM8 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

hcounteren.HPM9 Field

Location:
9

Description:
When all of [scounteren.HPM9](#), [mcounteren.HPM9](#), and [hcounteren.HPM9](#) are set, the [hpmcounter9](#) CSR (an alias of [mhpmcounter9](#)) is accessible to VU-mode.

When [mcounteren.HPM9](#) and [hcounteren.HPM9](#) are set, the [hpmcounter9](#) CSR (an alias of [mhpmcounter9](#)) is accessible to VS-mode.

When [hcounteren.HPM9](#) is clear and [mcounteren.HPM9](#) is set, then any access to [hpmcounter9](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM9 .2+h! mcounteren.HPM9 .2+h! scounteren.HPM9 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

<i>RW</i>
<i>RO</i>
Reset value: <i>UNDEFINED_LEGAL</i>

hcounteren.HPM10 Field

Location: <i>10</i>
Description: <i>When all of scounteren.HPM10, mcounteren.HPM10, and hcounteren.HPM10 are set, the hpmcounter10 CSR (an alias of mhpmcounter10) is accessible to VU-mode.</i> <i>When mcounteren.HPM10 and hcounteren.HPM10 are set, the hpmcounter10 CSR (an alias of mhpmcounter10) is accessible to VS-mode.</i> <i>When hcounteren.HPM10 is clear and mcounteren.HPM10 is set, then any access to hpmcounter10 in VU-mode or VS-mode causes a VirtualInstruction exception.</i> <i>Summary:</i> <i>[separator="!",%autowidth]</i> <i>!===</i> <i>.2+h! hcounteren.HPM10 .2+h! mcounteren.HPM10 .2+h! scounteren.HPM10 2+^.>! cycle access behavior</i> <i>.>h! VS-mode .>h! VU-mode</i> <i>! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction</i> <i>! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction</i> <i>! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction</i> <i>! 1 ! 1 ! 0 ! allowed ! VirtualInstruction</i> <i>! 1 ! 1 ! 1 ! allowed ! allowed</i> <i>!===</i>
Type: <i>RW</i> <i>RO</i> Reset value: <i>UNDEFINED_LEGAL</i>

hcounteren.HPM11 Field

Location: <i>11</i>
Description: <i>When all of scounteren.HPM11, mcounteren.HPM11, and hcounteren.HPM11 are set, the hpmcounter11 CSR (an alias of mhpmcounter11) is accessible to VU-mode.</i> <i>When mcounteren.HPM11 and hcounteren.HPM11 are set, the hpmcounter11 CSR (an alias of mhpmcounter11) is accessible to VS-mode.</i> <i>When hcounteren.HPM11 is clear and mcounteren.HPM11 is set, then any access to hpmcounter11 in VU-mode or VS-mode causes a VirtualInstruction exception.</i> <i>Summary:</i> <i>[separator="!",%autowidth]</i> <i>!===</i> <i>.2+h! hcounteren.HPM11 .2+h! mcounteren.HPM11 .2+h! scounteren.HPM11 2+^.>! cycle access behavior</i> <i>.>h! VS-mode .>h! VU-mode</i> <i>! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction</i> <i>! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction</i> <i>! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction</i> <i>! 1 ! 1 ! 0 ! allowed ! VirtualInstruction</i>

! 1 ! 1 ! 1 ! allowed ! allowed !===
Type: RW RO
Reset value: UNDEFINED_LEGAL

hcounteren.HPM12 Field

Location: 12
Description: When all of scounteren.HPM12, mcounteren.HPM12, and hcounteren.HPM12 are set, the hpmcounter12 CSR (an alias of mhpmcounter12) is accessible to VU-mode. When mcounteren.HPM12 and hcounteren.HPM12 are set, the hpmcounter12 CSR (an alias of mhpmcounter12) is accessible to VS-mode. When hcounteren.HPM12 is clear and mcounteren.HPM12 is set, then any access to hpmcounter12 in VU-mode or VS-mode causes a VirtualInstruction exception. Summary: [separator="!",%autowidth] !=== .2+h! hcounteren.HPM12 .2+h! mcounteren.HPM12 .2+h! scounteren.HPM12 2+^.>! cycle access behavior .>h! VS-mode .>h! VU-mode ! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction ! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction ! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction ! 1 ! 1 ! 0 ! allowed ! VirtualInstruction ! 1 ! 1 ! 1 ! allowed ! allowed !===
Type: RW RO
Reset value: UNDEFINED_LEGAL

hcounteren.HPM13 Field

Location: 13
Description: When all of scounteren.HPM13, mcounteren.HPM13, and hcounteren.HPM13 are set, the hpmcounter13 CSR (an alias of mhpmcounter13) is accessible to VU-mode. When mcounteren.HPM13 and hcounteren.HPM13 are set, the hpmcounter13 CSR (an alias of mhpmcounter13) is accessible to VS-mode. When hcounteren.HPM13 is clear and mcounteren.HPM13 is set, then any access to hpmcounter13 in VU-mode or VS-mode causes a VirtualInstruction exception. Summary: [separator="!",%autowidth] !=== .2+h! hcounteren.HPM13 .2+h! mcounteren.HPM13 .2+h! scounteren.HPM13 2+^.>! cycle access behavior .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction ! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction ! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction ! 1 ! 1 ! 0 ! allowed ! VirtualInstruction ! 1 ! 1 ! 1 ! allowed ! allowed !===
Type: RW RO Reset value: UNDEFINED_LEGAL

hcounteren.HPM14 Field

Location: 14
Description: When all of scounteren.HPM14, mcounteren.HPM14, and hcounteren.HPM14 are set, the hpmcounter14 CSR (an alias of mhpmcounter14) is accessible to VU-mode. When mcounteren.HPM14 and hcounteren.HPM14 are set, the hpmcounter14 CSR (an alias of mhpmcounter14) is accessible to VS-mode. When hcounteren.HPM14 is clear and mcounteren.HPM14 is set, then any access to hpmcounter14 in VU-mode or VS-mode causes a <i>VirtualInstruction</i> exception. Summary: [separator="!",%autowidth] !=== .2+h! hcounteren.HPM14 .2+h! mcounteren.HPM14 .2+h! scounteren.HPM14 2+^.>! cycle access behavior .>h! VS-mode .>h! VU-mode ! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction ! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction ! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction ! 1 ! 1 ! 0 ! allowed ! VirtualInstruction ! 1 ! 1 ! 1 ! allowed ! allowed !===
Type: RW RO Reset value: UNDEFINED_LEGAL

hcounteren.HPM15 Field

Location: 15
Description: When all of scounteren.HPM15, mcounteren.HPM15, and hcounteren.HPM15 are set, the hpmcounter15 CSR (an alias of mhpmcounter15) is accessible to VU-mode. When mcounteren.HPM15 and hcounteren.HPM15 are set, the hpmcounter15 CSR (an alias of mhpmcounter15) is accessible to VS-mode. When hcounteren.HPM15 is clear and mcounteren.HPM15 is set, then any access to hpmcounter15 in VU-mode or VS-mode causes a <i>VirtualInstruction</i> exception. Summary: [separator="!",%autowidth]

```
!===
.2+h! hcounteren.HPM15 .2+h! mcounteren.HPM15 .2+h! scounteren.HPM15 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM16 Field

Location:

16

Description:

When all of [scounteren.HPM16](#), [mcounteren.HPM16](#), and [hcounteren.HPM16](#) are set, the [hpmcounter16](#) CSR (an alias of [mhpmcounter16](#)) is accessible to VU-mode.

When [mcounteren.HPM16](#) and [hcounteren.HPM16](#) are set, the [hpmcounter16](#) CSR (an alias of [mhpmcounter16](#)) is accessible to VS-mode.

When [hcounteren.HPM16](#) is clear and [mcounteren.HPM16](#) is set, then any access to [hpmcounter16](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

[separator="!",%autowidth]

```
!===
.2+h! hcounteren.HPM16 .2+h! mcounteren.HPM16 .2+h! scounteren.HPM16 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM17 Field

Location:

17

Description:

When all of [scounteren.HPM17](#), [mcounteren.HPM17](#), and [hcounteren.HPM17](#) are set, the [hpmcounter17](#) CSR (an alias of [mhpmcounter17](#)) is accessible to VU-mode.

When [mcounteren.HPM17](#) and [hcounteren.HPM17](#) are set, the [hpmcounter17](#) CSR (an alias of [mhpmcounter17](#)) is accessible to VS-mode.

When [hcounteren.HPM17](#) is clear and [mcounteren.HPM17](#) is set, then any access to [hpmcounter17](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

[separator="!",%autowidth]

!===

.2+h! hcounteren.HPM17 .2+h! mcounteren.HPM17 .2+h! scounteren.HPM17 2+^.>! cycle access behavior

.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction

! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction

! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction

! 1 ! 1 ! 0 ! allowed ! VirtualInstruction

! 1 ! 1 ! 1 ! allowed ! allowed

!===

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM18 Field

Location:

18

Description:

When all of scounteren.HPM18, mcounteren.HPM18, and hcounteren.HPM18 are set, the hpmcounter18 CSR (an alias of mhpmpcounter18) is accessible to VU-mode.

When mcounteren.HPM18 and hcounteren.HPM18 are set, the hpmcounter18 CSR (an alias of mhpmpcounter18) is accessible to VS-mode.

When hcounteren.HPM18 is clear and mcounteren.HPM18 is set, then any access to hpmcounter18 in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

[separator="!",%autowidth]

!===

.2+h! hcounteren.HPM18 .2+h! mcounteren.HPM18 .2+h! scounteren.HPM18 2+^.>! cycle access behavior

.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction

! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction

! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction

! 1 ! 1 ! 0 ! allowed ! VirtualInstruction

! 1 ! 1 ! 1 ! allowed ! allowed

!===

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM19 Field

Location:

19

Description:

When all of scounteren.HPM19, mcounteren.HPM19, and hcounteren.HPM19 are set, the hpmcounter19 CSR (an alias of mhpmpcounter19) is accessible to VU-mode.

When mcounteren.HPM19 and hcounteren.HPM19 are set,

the [hpmcounter19](#) CSR (an alias of [mhpmcounter19](#)) is accessible to VS-mode.

When [hcounteren.HPM19](#) is clear and [mcounteren.HPM19](#) is set, then any access to [hpmcounter19](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM19 .2+h! mcounteren.HPM19 .2+h! scounteren.HPM19 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0! 0! -! IllegalInstruction! IllegalInstruction
! 0! 1! -! VirtualInstruction! VirtualInstruction
! 1! 0! -! IllegalInstruction! IllegalInstruction
! 1! 1! 0! allowed! VirtualInstruction
! 1! 1! 1! allowed! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

[hcounteren.HPM20](#) Field

Location:

20

Description:

When all of [scounteren.HPM20](#), [mcounteren.HPM20](#), and [hcounteren.HPM20](#) are set, the [hpmcounter20](#) CSR (an alias of [mhpmcounter20](#)) is accessible to VU-mode.

When [mcounteren.HPM20](#) and [hcounteren.HPM20](#) are set, the [hpmcounter20](#) CSR (an alias of [mhpmcounter20](#)) is accessible to VS-mode.

When [hcounteren.HPM20](#) is clear and [mcounteren.HPM20](#) is set, then any access to [hpmcounter20](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM20 .2+h! mcounteren.HPM20 .2+h! scounteren.HPM20 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0! 0! -! IllegalInstruction! IllegalInstruction
! 0! 1! -! VirtualInstruction! VirtualInstruction
! 1! 0! -! IllegalInstruction! IllegalInstruction
! 1! 1! 0! allowed! VirtualInstruction
! 1! 1! 1! allowed! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

[hcounteren.HPM21](#) Field

Location:

21

Description:

When all of [scounteren.HPM21](#), [mcounteren.HPM21](#), and [hcounteren.HPM21](#) are set, the [hpmcounter21](#) CSR (an alias of [mhpmcounter21](#)) is accessible to VU-mode.

When [mcounteren.HPM21](#) and [hcounteren.HPM21](#) are set, the [hpmcounter21](#) CSR (an alias of [mhpmcounter21](#)) is accessible to VS-mode.

When [hcounteren.HPM21](#) is clear and [mcounteren.HPM21](#) is set, then any access to [hpmcounter21](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM21 .2+h! mcounteren.HPM21 .2+h! scounteren.HPM21 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

[hcounteren.HPM22](#) Field

Location:

22

Description:

When all of [scounteren.HPM22](#), [mcounteren.HPM22](#), and [hcounteren.HPM22](#) are set, the [hpmcounter22](#) CSR (an alias of [mhpmcounter22](#)) is accessible to VU-mode.

When [mcounteren.HPM22](#) and [hcounteren.HPM22](#) are set, the [hpmcounter22](#) CSR (an alias of [mhpmcounter22](#)) is accessible to VS-mode.

When [hcounteren.HPM22](#) is clear and [mcounteren.HPM22](#) is set, then any access to [hpmcounter22](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM22 .2+h! mcounteren.HPM22 .2+h! scounteren.HPM22 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

[hcounteren.HPM23](#) Field

Location:

23

Description:

When all of [scounteren.HPM23](#), [mcounteren.HPM23](#), and [hcounteren.HPM23](#) are set, the [hpmcounter23](#) CSR (an alias of [mhpmcounter23](#)) is accessible to VU-mode.

When [mcounteren.HPM23](#) and [hcounteren.HPM23](#) are set, the [hpmcounter23](#) CSR (an alias of [mhpmcounter23](#)) is accessible to VS-mode.

When [hcounteren.HPM23](#) is clear and [mcounteren.HPM23](#) is set, then any access to [hpmcounter23](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM23 .2+h! mcounteren.HPM23 .2+h! scounteren.HPM23 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

[hcounteren.HPM24](#) Field

Location:

24

Description:

When all of [scounteren.HPM24](#), [mcounteren.HPM24](#), and [hcounteren.HPM24](#) are set, the [hpmcounter24](#) CSR (an alias of [mhpmcounter24](#)) is accessible to VU-mode.

When [mcounteren.HPM24](#) and [hcounteren.HPM24](#) are set, the [hpmcounter24](#) CSR (an alias of [mhpmcounter24](#)) is accessible to VS-mode.

When [hcounteren.HPM24](#) is clear and [mcounteren.HPM24](#) is set, then any access to [hpmcounter24](#) in VU-mode or VS-mode causes a *VirtualInstruction* exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM24 .2+h! mcounteren.HPM24 .2+h! scounteren.HPM24 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM25 Field

Location:

25

Description:

When all of [scounteren.HPM25](#), [mcounteren.HPM25](#), and [hcounteren.HPM25](#) are set, the [hpmcounter25](#) CSR (an alias of [mhpmcounter25](#)) is accessible to VU-mode.

When [mcounteren.HPM25](#) and [hcounteren.HPM25](#) are set, the [hpmcounter25](#) CSR (an alias of [mhpmcounter25](#)) is accessible to VS-mode.

When [hcounteren.HPM25](#) is clear and [mcounteren.HPM25](#) is set, then any access to [hpmcounter25](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM25 .2+h! mcounteren.HPM25 .2+h! scounteren.HPM25 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

hcounteren.HPM26 Field

Location:

26

Description:

When all of [scounteren.HPM26](#), [mcounteren.HPM26](#), and [hcounteren.HPM26](#) are set, the [hpmcounter26](#) CSR (an alias of [mhpmcounter26](#)) is accessible to VU-mode.

When [mcounteren.HPM26](#) and [hcounteren.HPM26](#) are set, the [hpmcounter26](#) CSR (an alias of [mhpmcounter26](#)) is accessible to VS-mode.

When [hcounteren.HPM26](#) is clear and [mcounteren.HPM26](#) is set, then any access to [hpmcounter26](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

```
[separator="!",%autowidth]
!===
.2+h! hcounteren.HPM26 .2+h! mcounteren.HPM26 .2+h! scounteren.HPM26 2+^.>! cycle access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===
```

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

hcounteren.HPM27 Field

Location:
27

Description:

When all of [scounteren.HPM27](#), [mcounteren.HPM27](#), and [hcounteren.HPM27](#) are set, the [hpmcounter27](#) CSR (an alias of [mhpmcounter27](#)) is accessible to VU-mode.

When [mcounteren.HPM27](#) and [hcounteren.HPM27](#) are set, the [hpmcounter27](#) CSR (an alias of [mhpmcounter27](#)) is accessible to VS-mode.

When [hcounteren.HPM27](#) is clear and [mcounteren.HPM27](#) is set, then any access to [hpmcounter27](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

[separator="!",%autowidth]
!===
.2+h! [hcounteren.HPM27](#) .2+h! [mcounteren.HPM27](#) .2+h! [scounteren.HPM27](#) 2+^.>! [cycle](#) access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

hcounteren.HPM28 Field

Location:
28

Description:

When all of [scounteren.HPM28](#), [mcounteren.HPM28](#), and [hcounteren.HPM28](#) are set, the [hpmcounter28](#) CSR (an alias of [mhpmcounter28](#)) is accessible to VU-mode.

When [mcounteren.HPM28](#) and [hcounteren.HPM28](#) are set, the [hpmcounter28](#) CSR (an alias of [mhpmcounter28](#)) is accessible to VS-mode.

When [hcounteren.HPM28](#) is clear and [mcounteren.HPM28](#) is set, then any access to [hpmcounter28](#) in VU-mode or VS-mode causes a VirtualInstruction exception.

Summary:

[separator="!",%autowidth]
!===
.2+h! [hcounteren.HPM28](#) .2+h! [mcounteren.HPM28](#) .2+h! [scounteren.HPM28](#) 2+^.>! [cycle](#) access behavior
.>h! VS-mode .>h! VU-mode

! 0 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 0 ! 1 ! - ! VirtualInstruction ! VirtualInstruction
! 1 ! 0 ! - ! IllegalInstruction ! IllegalInstruction
! 1 ! 1 ! 0 ! allowed ! VirtualInstruction
! 1 ! 1 ! 1 ! allowed ! allowed
!===

Type:

<i>RW</i>
<i>RO</i>
Reset value: <i>UNDEFINED_LEGAL</i>

hcounteren.HPM29 Field

Location: 29
Description: When all of <i>scounteren.HPM29</i> , <i>mcounteren.HPM29</i> , and <i>hcounteren.HPM29</i> are set, the <i>hpmcounter29</i> CSR (an alias of <i>mhpmcounter29</i>) is accessible to VU-mode. When <i>mcounteren.HPM29</i> and <i>hcounteren.HPM29</i> are set, the <i>hpmcounter29</i> CSR (an alias of <i>mhpmcounter29</i>) is accessible to VS-mode. When <i>hcounteren.HPM29</i> is clear and <i>mcounteren.HPM29</i> is set, then any access to <i>hpmcounter29</i> in VU-mode or VS-mode causes a <i>VirtualInstruction</i> exception. Summary: [separator="!",%autowidth] !=== .2+h! <i>hcounteren.HPM29</i> .2+h! <i>mcounteren.HPM29</i> .2+h! <i>scounteren.HPM29</i> 2+^.>! <i>cycle</i> access behavior >h! VS-mode >h! VU-mode ! 0 ! 0 ! - ! <i>IllegalInstruction</i> ! <i>IllegalInstruction</i> ! 0 ! 1 ! - ! <i>VirtualInstruction</i> ! <i>VirtualInstruction</i> ! 1 ! 0 ! - ! <i>IllegalInstruction</i> ! <i>IllegalInstruction</i> ! 1 ! 1 ! 0 ! <i>allowed</i> ! <i>VirtualInstruction</i> ! 1 ! 1 ! 1 ! <i>allowed</i> ! <i>allowed</i> !=== Type: <i>RW</i> <i>RO</i> Reset value: <i>UNDEFINED_LEGAL</i>

hcounteren.HPM30 Field

Location: 30
Description: When all of <i>scounteren.HPM30</i> , <i>mcounteren.HPM30</i> , and <i>hcounteren.HPM30</i> are set, the <i>hpmcounter30</i> CSR (an alias of <i>mhpmcounter30</i>) is accessible to VU-mode. When <i>mcounteren.HPM30</i> and <i>hcounteren.HPM30</i> are set, the <i>hpmcounter30</i> CSR (an alias of <i>mhpmcounter30</i>) is accessible to VS-mode. When <i>hcounteren.HPM30</i> is clear and <i>mcounteren.HPM30</i> is set, then any access to <i>hpmcounter30</i> in VU-mode or VS-mode causes a <i>VirtualInstruction</i> exception. Summary: [separator="!",%autowidth] !=== .2+h! <i>hcounteren.HPM30</i> .2+h! <i>mcounteren.HPM30</i> .2+h! <i>scounteren.HPM30</i> 2+^.>! <i>cycle</i> access behavior >h! VS-mode >h! VU-mode ! 0 ! 0 ! - ! <i>IllegalInstruction</i> ! <i>IllegalInstruction</i> ! 0 ! 1 ! - ! <i>VirtualInstruction</i> ! <i>VirtualInstruction</i> ! 1 ! 0 ! - ! <i>IllegalInstruction</i> ! <i>IllegalInstruction</i> ! 1 ! 1 ! 0 ! <i>allowed</i> ! <i>VirtualInstruction</i>

! 1! 1! 1! allowed ! allowed !===
Type: RW RO
Reset value: UNDEFINED_LEGAL

hcounteren.HPM31 Field

Location: 31
Description: When all of scounteren.HPM31, mcounteren.HPM31, and hcounteren.HPM31 are set, the hpmcounter31 CSR (an alias of mhpmcounter31) is accessible to VU-mode. When mcounteren.HPM31 and hcounteren.HPM31 are set, the hpmcounter31 CSR (an alias of mhpmcounter31) is accessible to VS-mode. When hcounteren.HPM31 is clear and mcounteren.HPM31 is set, then any access to hpmcounter31 in VU-mode or VS-mode causes a VirtualInstruction exception. Summary: [separator="!",%autowidth] !=== .2+h! hcounteren.HPM31 .2+h! mcounteren.HPM31 .2+h! scounteren.HPM31 2+^.>! cycle access behavior .>h! VS-mode .>h! VU-mode ! 0! 0! - ! IllegalInstruction ! IllegalInstruction ! 0! 1! - ! VirtualInstruction ! VirtualInstruction ! 1! 0! - ! IllegalInstruction ! IllegalInstruction ! 1! 1! 0! allowed ! VirtualInstruction ! 1! 1! 1! allowed ! allowed !===
Type: RW RO
Reset value: UNDEFINED_LEGAL

D.7. hedeleg

Hypervisor Exception Delegation

Controls exception delegation from HS-mode to VS-mode.

By default, all traps at any privilege level are handled in M-mode, though M-mode usually uses the [medeleg](#) and [mideleg](#) CSRs to delegate some traps to HS-mode. The [hedeleg](#) and `hideleg` CSRs allow these traps to be further delegated to a VS-mode guest; their layout is the same as [medeleg](#) and [mideleg](#).

A synchronous trap that has been delegated to HS-mode (using [medeleg](#)) is further delegated to VS-mode if `V=1` before the trap and the corresponding [hedeleg](#) bit is set. Each bit of [hedeleg](#) shall be either writable or read-only zero. Many bits of [hedeleg](#) are required specifically to be writable or zero. Bit 0, corresponding to instruction address misaligned exceptions, must be writable if `IALIGN=32`.



Requiring that certain bits of [hedeleg](#) be writable reduces some of the burden on a hypervisor to handle variations of implementation.

When `XLEN=32`, [hedelegh](#) is a 32-bit read/write register that aliases bits 63:32 of [hedeleg](#). Register [hedelegh](#) does not exist when `XLEN=64`.

D.7.1. Attributes

CSR Address	0x602
Defining extension	H, version >= H@1.0.0
Length	64-bit
Privilege Mode	S

D.7.2. Format

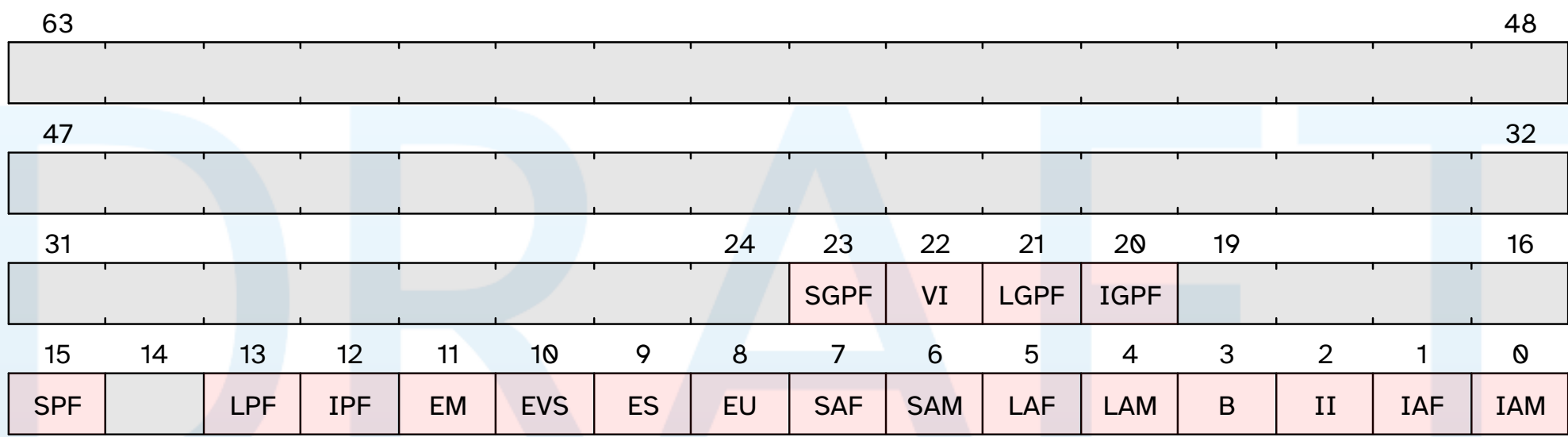


Figure 7. hedeleg format

D.7.3. Field Summary

Name	Location	Type	Reset Value
hedeleg.IAM	0	RW	UNDEFINED_LEGAL
hedeleg.IAF	1	RW	UNDEFINED_LEGAL
hedeleg.II	2	RW	UNDEFINED_LEGAL
hedeleg.B	3	RW	UNDEFINED_LEGAL
hedeleg.LAM	4	RW	UNDEFINED_LEGAL
hedeleg.LAF	5	RW	UNDEFINED_LEGAL
hedeleg.SAM	6	RW	UNDEFINED_LEGAL
hedeleg.SAF	7	RW	UNDEFINED_LEGAL
hedeleg.EU	8	RW	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
hedeleg.ES	9	RO	0
hedeleg.EVS	10	RO	0
hedeleg.EM	11	RO	0
hedeleg.IPF	12	RW	UNDEFINED_LEGAL
hedeleg.LPF	13	RW	UNDEFINED_LEGAL
hedeleg.SPF	15	RW	UNDEFINED_LEGAL
hedeleg.IGPF	20	RO	0
hedeleg.LGPF	21	RO	0
hedeleg.VI	22	RO	0
hedeleg.SGPF	23	RO	0

D.7.4. Fields

[hedeleg.IAM](#) Field

Location:

0

Description:

Instruction Address Misaligned

Controls delegation of Instruction Address Misaligned exceptions to VS-mode.

See [medeleg.IAM](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

[hedeleg.IAF](#) Field

Location:

1

Description:

Instruction Access Fault

Controls delegation of Instruction Access Fault exceptions to VS-mode.

See [medeleg.IAF](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.II Field

Location:

2

Description:

Illegal Instruction

Controls delegation of Illegal Instruction exceptions to VS-mode.

See [medeleg.II](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.B Field

Location:

3

Description:

Breakpoint

Controls delegation of Breakpoint exceptions to VS-mode.

See [medeleg.B](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.LAM Field

Location:

4

Description:

Load Address Misaligned

Controls delegation of Load Address Misaligned exceptions to VS-mode.

See [medeleg.LAM](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.LAF Field

Location:

5

Description:

Load Access Fault

Controls delegation of Load Access Fault exceptions to VS-mode.

See [medeleg.LAF](#) for details.

Type:

RW

Reset value:
UNDEFINED_LEGAL

hedeleg.SAM Field

Location:
6

Description:
Store/AMO Address Misaligned

Controls delegation of Store/AMO Address Misaligned exceptions to VS-mode.

See [medeleg.SAM](#) for details.

Type:
RW

Reset value:
UNDEFINED_LEGAL

hedeleg.SAF Field

Location:
7

Description:
Store/AMO Access Fault

Controls delegation of Store/AMO Access Fault exceptions to VS-mode.

See [medeleg.SAF](#) for details.

Type:
RW

Reset value:
UNDEFINED_LEGAL

hedeleg.EU Field

Location:
8

Description:
Environment Call from VU-mode

Controls delegation of Environment Call from VU-mode exceptions to VS-mode.

See [medeleg.EU](#) for details.

Type:
RW

Reset value:
UNDEFINED_LEGAL

hedeleg.ES Field

Location:
9

Description:
Environment Call from HS-mode

Environment Call from HS-mode exceptions cannot be delegated to VS-mode,

so this field is read-only 0.

See [medeleg.ES](#) for details.

Type:

RO

Reset value:

0

hedeleg.EVS Field

Location:

10

Description:

Environment Call from VS-mode

Environment Call from VS-mode exceptions cannot be delegated to VS-mode, so this field is read-only 0.

See [medeleg.EVS](#) for details.

Type:

RO

Reset value:

0

hedeleg.EM Field

Location:

11

Description:

Environment Call from M-mode

Environment Call from M-mode exceptions cannot be delegated to VS-mode, so this field is read-only 0.

See [medeleg.EM](#) for details.

Type:

RO

Reset value:

0

hedeleg.IPF Field

Location:

12

Description:

Instruction Page Fault

Controls delegation of Instruction Page Fault exceptions to VS-mode.

See [medeleg.IPF](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.LPF Field

Location:

13

Description:

Load Page Fault

Controls delegation of Load Page Fault exceptions to VS-mode.

See [medeleg.LPF](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.SPF Field

Location:

15

Description:

Store/AMO Page Fault

Controls delegation of Store/AMO Page Fault exceptions to VS-mode.

See [medeleg.SPF](#) for details.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hedeleg.IGPF Field

Location:

20

Description:

Instruction Guest Page Fault

Instruction Guest Page Fault exceptions cannot be delegated to VS-mode, so this field is read-only 0.

See [medeleg.IGPF](#) for details.

Type:

RO

Reset value:

0

hedeleg.LGPF Field

Location:

21

Description:

Load Guest Page Fault

Load Guest Page Fault exceptions cannot be delegated to VS-mode, so this field is read-only 0.

See [medeleg.LGPF](#) for details.

Type:

RO

Reset value:

0

hedeleg.VI Field

Location:

22

Description:

Virtual Instruction

Virtual Instruction exceptions cannot be delegated to VS-mode, so this field is read-only 0.

See [medeleg.VI](#) for details.

Type:

RO

Reset value:

0

hedeleg.SGPF Field

Location:

23

Description:

Store/AMO Guest Page Fault

Store/AMO Guest Page Fault exceptions cannot be delegated to VS-mode, so this field is read-only 0.

See [medeleg.SGPF](#) for details.

Type:

RO

Reset value:

0

D.8. hedeleg

Hypervisor Exception Delegation High

 *hedeleg* is only defined in RV32.

Controls exception delegation from HS-mode to VS-mode.

Alias of upper bits of [hedeleg](#)[63:32].

D.8.1. Attributes

CSR Address	0x612
Defining extension	H, version >= H@1.0.0
Length	32-bit
Privilege Mode	S

D.8.2. Format



Figure 8. hedeleg format

D.8.3. Field Summary

Name	Location	Type	Reset Value

D.8.4. Fields

This CSR has no fields. However, it must still exist (not cause an Illegal Instruction trap) and always return zero on a read.

D.9. henvcfg

Hypervisor Environment Configuration

The henvcfg CSR is a 64-bit read/write register that controls certain characteristics of the execution environment when virtualization mode V=1.

If bit [henvcfg.FIOM](#) (Fence of I/O implies Memory) is set to one in henvcfg, [fence](#) instructions executed when V=1 are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses.

[Table 18](#) details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO when FIOM=1 and V=1.

Similarly, when [henvcfg.FIOM](#)=1 and V=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

Table 18. Modified interpretation of FENCE predecessor and successor sets when FIOM=1 and virtualization mode V=1.

Instruction bit	Meaning when set
PI	Predecessor device input and memory reads (PR implied)
PO	Predecessor device output and memory writes (PW implied)
SI	Successor device input and memory reads (SR implied)
SO	Successor device output and memory writes (SW implied)

The PBMTE bit controls whether the Svpbmt extension is available for use in VS-stage address translation. When PBMTE=1, Svpbmt is available for VS-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented for VS-stage address translation. If Svpbmt is not implemented, PBMTE is read-only zero.

If the Svadu extension is implemented, the ADUE bit controls whether hardware updating of PTE A/D bits is enabled for VS-stage address translation. When ADUE=1, hardware updating of PTE A/D bits is enabled during VS-stage address translation, and the implementation behaves as though the Svade extension were not implemented for VS-mode address translation. When ADUE=0, the implementation behaves as though Svade were implemented for VS-stage address translation. If Svadu is not implemented, ADUE is read-only zero.

The definition of the STCE field is furnished by the Sstc extension.

The definition of the CBZE field is furnished by the Zicboz extension.

The definitions of the CBCFE and CBIE fields are furnished by the Zicbom extension.

The definition of the PMM field will be furnished by the forthcoming Ssnpm extension. Its allocation within [henvcfg](#) may change prior to the ratification of that extension.

The Zicfilp extension adds the LPE field in [henvcfg](#). When the LPE field is set to 1, the Zicfilp extension is enabled in VS-mode. When the LPE field is 0, the Zicfilp extension is not enabled in VS-mode and the following rules apply to VS-mode:

- The hart does not update the ELP state; it remains as NO_LP_EXPECTED.
- The LPAD instruction operates as a no-op.

The Zicfiss extension adds the SSE field in [henvcfg](#). If the SSE field is set to 1, the Zicfiss extension is activated in VS-mode. When the SSE field is 0, the Zicfiss extension remains inactive in VS-mode, and the following rules apply when V=1:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS-stage page tables becomes reserved.
- The [senvcfg](#) field will read as zero and is read-only.
- When [menvcfg](#) is one, `SSAMOSWAP.W/D` raises a virtual instruction exception.

The Ssdbltrp extension adds the double-trap-enable (DTE) field in [henvcfg](#). When [henvcfg](#) is zero, the implementation behaves as though Ssdbltrp is not implemented for VS-mode and the [vsstatus](#) bit is read-only zero.

When XLEN=32, [henvcfgh](#) is a 32-bit read/write register that aliases bits 63:32 of [henvcfg](#). Register [henvcfgh](#) does not exist when XLEN=64.

D.9.1. Attributes

CSR Address	0x60a
Defining extension	• allOf: • Sm, version >=1.12 • H, version >= H@1.0.0
Length	64-bit
Privilege Mode	S

D.9.2. Format

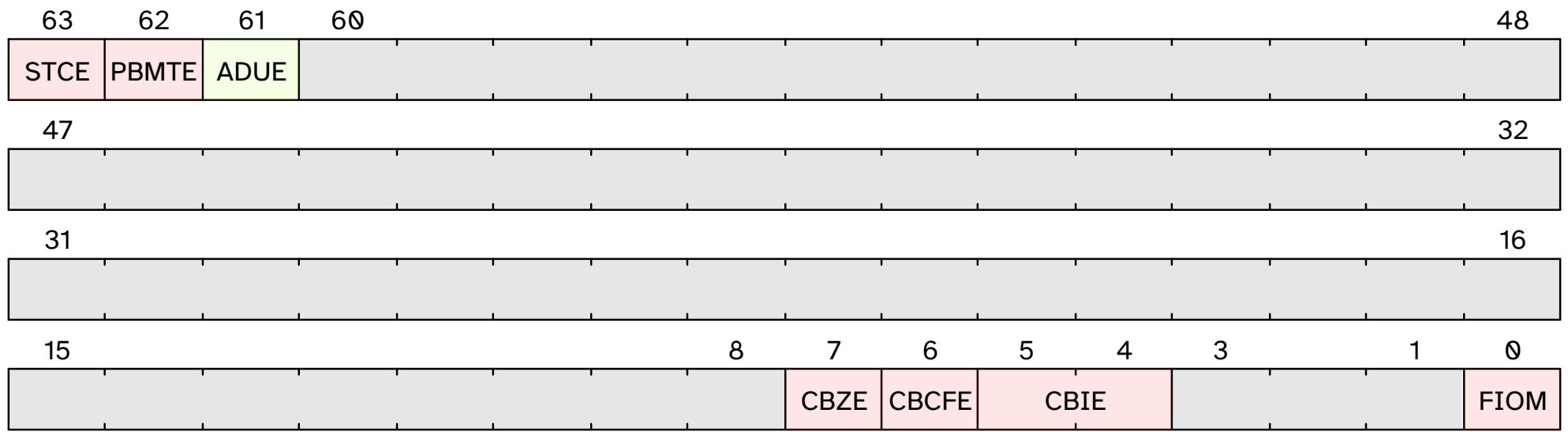


Figure 9. henvcfg format

D.9.3. Field Summary

Name	Location	Type	Reset Value
henvcfg.STCE	63	RO	UNDEFINED_LEGAL
henvcfg.PBMTE	62	RO	UNDEFINED_LEGAL
henvcfg.CBZE	7	RW	UNDEFINED_LEGAL
henvcfg.CBCFE	6	RW	UNDEFINED_LEGAL
henvcfg.CBIE	5:4	RW-R	UNDEFINED_LEGAL
henvcfg.FIO M	0	RW	UNDEFINED_LEGAL

D.9.4. Fields

[henvcfg.STCE](#) Field

Location:

63

Description:

STimecmp Enable

When set, stimecmp is operational in VS-mode if [menvcfg.STCE](#) is also set.

When [menvcfg.STCE](#) is zero:

- [henvcfg.STCE](#) reads-as-zero
- vstimecmp access raises an IllegalInstruction exception.
- hip.VSTIP reverts to its defined behavior as if Sstc is not implemented.
- VS-mode timer interrupts will not be generated

When [menvcfg.STCE](#) is one and [henvcfg.STCE](#) is zero:

- Accessing stimecmp in VS-mode or VU-mode (really vstimecmp) raises a VirtualInterrupt exception
- hip.VSTIP reverts to its defined behavior as if Sstc is not implemented.
- VS-mode timer interrupts will not be generated

Type:

RO

RW

Reset value:

UNDEFINED_LEGAL

henvcfg.PBMTE Field

Location:

62

Description:

Page Based Memory Type Enable

The PBMTE bit controls whether the [Svpbmt](#) extension is available for use in VS-stage address translation.

When PBMTE=1, Svpbmt is available for VS-stage address translation.

When PBMTE=0, the implementation behaves as though [Svpbmt](#) were not implemented for VS-stage address translation.

If [Svpbmt](#) is not implemented, PBMTE is read-only zero.

[henvcfg.PBMTE](#) is read-as-zero if [menvcfg.PBMTE](#) is zero.

If the setting of the PBMTE bit in [menvcfg](#) is changed, an [hfence.gvma](#) instruction with *rs1*=x0 and *rs2*=x0 suffices to synchronize with respect to the altered interpretation of G-stage and VS-stage PTEs' PBMT fields.

By contrast, if the PBMTE bit in [henvcfg](#) is changed, executing an [hfence.vvma](#) with *rs1*=x0 and *rs2*=x0 suffices to synchronize with respect to the altered interpretation of VS-stage PTEs' PBMT fields for the currently active VMID.

[NOTE] + — + No mechanism is provided to atomically change [vsatp](#) and [hgatp](#) together. Hence, to prevent speculative execution causing one guest's VS-stage translations to be cached under another guest's VMID, world-switch code should zero [vsatp](#), then swap [hgatp](#), then finally write the new [vsatp](#) value. Similarly, if [henvcfg.PBMTE](#) need be world-switched, it should be switched after zeroing [vsatp](#) but before writing the new [vsatp](#) value, obviating the need to execute an [hfence.vvma](#) instruction. + — +

Type:

RO

Reset value:

UNDEFINED_LEGAL

henvcfg.CBZE Field

Location:

7

Description:

Cache Block Zero instruction Enable

Enables the execution of the cache block zero instruction, CB0.ZERO,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 0: The instruction raises an illegal instruction or virtual instruction exception
- 1: The instruction is executed

Type:

RW

Reset value:

UNDEFINED_LEGAL

henvcfg.CBCFE Field

Location:

6

Description:

Cache Block Clean and Flush instruction Enable

Enables the execution of the cache block clean instruction, CB0.CLEAN, and the cache block flush instruction, CB0.FLUSH,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 0: The instruction raises an illegal instruction or virtual instruction exception
- 1: The instruction is executed

Type:

RW

Reset value:

UNDEFINED_LEGAL

henvcfg.CBIE Field

Location:

5:4

Description:

Cache Block Invalidate instruction Enable

Enables the execution of the cache block invalidate instruction, CB0.INVALID,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 00: The instruction raises an illegal instruction or virtual instruction exception
- 01: The instruction is executed and performs a flush operation
- 10: Reserved
- 11: The instruction is executed and performs an invalidate operation

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

henvcfg.FIOM Field

Location:

0

Description:

Fence of I/O implies Memory

When *henvcfg.FIOM* is set,
FENCE instructions ordering I/O regions also implicitly order memory regions when executed in any mode less privileged than M-mode.

[separator="!",%autowidth,float="center",align="center",cols="^,<",options="header"]
!===
!Instruction bit !Meaning when set
!PI +

PO
!Predecessor device input and memory reads (PR implied) +
Predecessor device output and memory writes (PW implied)
!SI +
SO
!Successor device input and memory reads (SR implied) +
Successor device output and memory writes (SW implied)
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

D.9.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through `csrrw`), the following determines the written value:

```
STCE = csr_value.STCE
PBMTE = csr_value.PBMTE
CBZE = csr_value.CBZE
CBCFE = csr_value.CBCFE
CBIE = if (csr_value.CBIE == 0 || csr_value.CBIE == 1 || csr_value.CBIE == 3) {
    return csr_value.CBIE;
} else {
    return CSR[menvcfg].CBIE;
}

FIOM = csr_value.FIOM
```

D.9.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
Bits<64> value = $bits(henvcfg);
if (implemented?(ExtensionName::Sstc) && menvcfg.STCE == 0) {
    value = value & ~(1 << 63);
}
if (implemented?(ExtensionName::Svpbmt) && menvcfg.PBMTE == 0) {
    value = value & ~(1 << 62);
}
if (implemented?(ExtensionName::Svadu) && menvcfg.ADUE == 0) {
    value = value & ~(1 << 61);
}
return value;
```

most-significant 32 bits of Hypervisor Environment Configuration

henvcfgh is only defined in RV32.

The `henvcfgh` CSR is a 32-bit read/write register for the most-significant 32 bits of `henvcfg`.

CSR Address	0x61a
Defining extension	- allOf: - Sm, version >=1.12 - H, version >= H@1.0.0
Length	32-bit
Privilege Mode	S

Diagram illustrating the structure of the 32-bit register:

- Bits 31 to 16 are shown in the top row.
- Fields defined in the top row:
 - STCE (bits 31-30)
 - PBMTE (bits 29-28)
 - ADUE (bits 27-26)
- Bits 15 to 0 are shown in the bottom row.

Figure 10. *henvcfgh* format

Name	Location	Type	Reset Value
henvcfgh.STCE	31	RO	UNDEFINED_LEGAL
		RW	
henvcfgh.PBMTE	30	RO	UNDEFINED_LEGAL

henvcfgh.STCE Field

31

STimecmp Enable

When set, stimecmp is operational in VS-mode if *menvcfg.STCE* is also set.

When *menvcfg.STCE* is zero:

- *henvcfg.STCE reads-as-zero*
- *vstimecmp access raises an IllegalInstruction exception.*
- *hip.VSTIP reverts to its defined behavior as if Sstc is not implemented.*
- *VS-mode timer interrupts will not be generated*

When `menvcfg.STCE` is one and `henvcfg.STCE` is zero:

- Accessing stimecmp in VS-mode or VU-mode (really vstimecmp) raises a *VirtualInterrupt* exception
- hip.VSTIP reverts to its defined behavior as if Sstc is not implemented.
- VS-mode timer interrupts will not be generated

RO

RW

Reset value:

UNDEFINED_LEGAL

henvcfg.PBMTE Field

Location:

30

Description:

Page Based Memory Type Enable

The PBMTE bit controls whether the [Svpbmt](#) extension is available for use in VS-stage address translation.

When PBMTE=1, Svpbmt is available for VS-stage address translation.

When PBMTE=0, the implementation behaves as though [Svpbmt](#) were not implemented for VS-stage address translation.

If [Svpbmt](#) is not implemented, PBMTE is read-only zero.

[henvcfg.PBMTE](#) is read-as-zero if [menvcfg.PBMTE](#) is zero.

If the setting of the PBMTE bit in [menvcfg](#) is changed, an [hfence.gvma](#) instruction with *rs1*=x0 and *rs2*=x0 suffices to synchronize with respect to the altered interpretation of G-stage and VS-stage PTEs' PBMT fields.

By contrast, if the PBMTE bit in [henvcfg](#) is changed, executing an [hfence.vvma](#) with *rs1*=x0 and *rs2*=x0 suffices to synchronize with respect to the altered interpretation of VS-stage PTEs' PBMT fields for the currently active VMID.

[NOTE] + — + No mechanism is provided to atomically change [vsatp](#) and [hgap](#) together. Hence, to prevent speculative execution causing one guest's VS-stage translations to be cached under another guest's VMID, world-switch code should zero [vsatp](#), then swap [hgap](#), then finally write the new [vsatp](#) value.

Similarly, if [henvcfg.PBMTE](#) need be world-switched, it should be switched after zeroing [vsatp](#) but before writing the new [vsatp](#) value, obviating the need to execute an [hfence.vvma](#) instruction. + — +

Type:

RO

Reset value:

UNDEFINED_LEGAL

D.10.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return henvcfg.sw_read()[63:32];
```

D.11. hcatp

Hypervisor guest address translation and protection

The hcatp register is an HSXLEN-bit read/write register which controls G-stage address translation and protection, the second stage of two-stage translation for guest virtual addresses. Similar to CSR [satp](#), this register holds the physical page number (PPN) of the guest-physical root page table; a virtual machine identifier (VMID), which facilitates address-translation fences on a per-virtual-machine basis; and the MODE field, which selects the address-translation scheme for guest physical addresses. When [mstatus.TVM](#)=1, attempts to read or write [hcatp](#) while executing in HS-mode will raise an `IllegalInstruction` exception.

[Table 19](#) shows the encodings of the MODE field when HSXLEN=32 and HSXLEN=64. When MODE=Bare, guest physical addresses are equal to supervisor physical addresses, and there is no further memory protection for a guest virtual machine beyond the physical memory protection scheme described in [\[pmp\]](#). In this case, the remaining fields in [hcatp](#) must be set to zeros.

Table 19. Encoding of [hcatp](#) MODE field.

HSXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32x4	Page-based 34-bit virtual addressing (2-bit extension of Sv32).
HSXLEN=64		
Value	Name	Description
0	Bare + — + Sv39x4	No translation or protection.
1-7	Sv48x4	Reserved
8	Sv57x4 + —	Page-based 41-bit virtual addressing (2-bit extension of Sv39).
9		Page-based 50-bit virtual addressing (2-bit extension of Sv48).
10		Page-based 59-bit virtual addressing (2-bit extension of Sv57).
11-15		Reserved

Implementations are not required to support all defined MODE settings when HSXLEN=64.

A write to [hcatp](#) with an unsupported MODE value is not ignored as it is for [satp](#). Instead, the fields of [hcatp](#) are **WARL** in the normal way, when so indicated.

As explained in [\[guest-addr-translation\]](#), for the paged virtual-memory schemes (Sv32x4, Sv39x4, Sv48x4, and Sv57x4), the root page table is 16 KiB and must be aligned to a 16-KiB boundary. In these modes, the lowest two bits of the physical page number (PPN) in [hcatp](#) always read as zeros. An implementation that supports only the defined paged virtual-memory schemes and/or Bare may make PPN[1:0] read-only zero.

The number of VMID bits is UNSPECIFIED and may be zero. The number of implemented VMID bits, termed *VMIDLEN*, may be determined by writing one to every bit position in the VMID field, then reading back the value in [hcatp](#) to see which bit positions in the VMID field hold a one. The least-significant bits of VMID are implemented first: that is, if VMIDLEN > 0, VMID[VMIDLEN-1:0] is writable. The maximal value of VMIDLEN, termed VMIDMAX, is 7 for Sv32x4 or 14 for Sv39x4, Sv48x4, and Sv57x4.

The [hcatp](#) register is considered *active* for the purposes of the address-translation algorithm *unless* the effective privilege mode is U and [hstatus.HU](#)=0.



This definition simplifies the implementation of speculative execution of `hlv`, `hlvx`, and `hsv` instructions.

Note that writing [hcatp](#) does not imply any ordering constraints between page-table updates and subsequent G-stage address translations. If the new virtual machine’s guest physical page tables have been modified, or if a VMID is reused, it may be necessary to execute an `HFENCE.GVMA` instruction (see [\[hfence.vma\]](#)) before or after writing [hcatp](#).

D.11.1. Attributes

CSR Address	0x680
Defining extension	• H, version >= H@1.0.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.11.2. Format

This CSR format changes dynamically with XLEN.

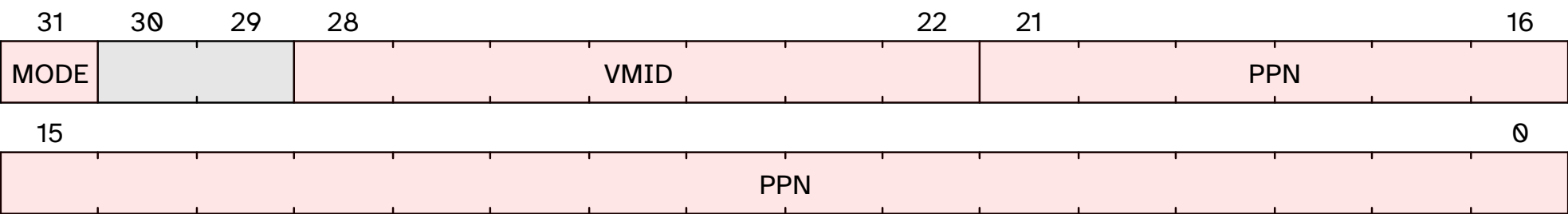
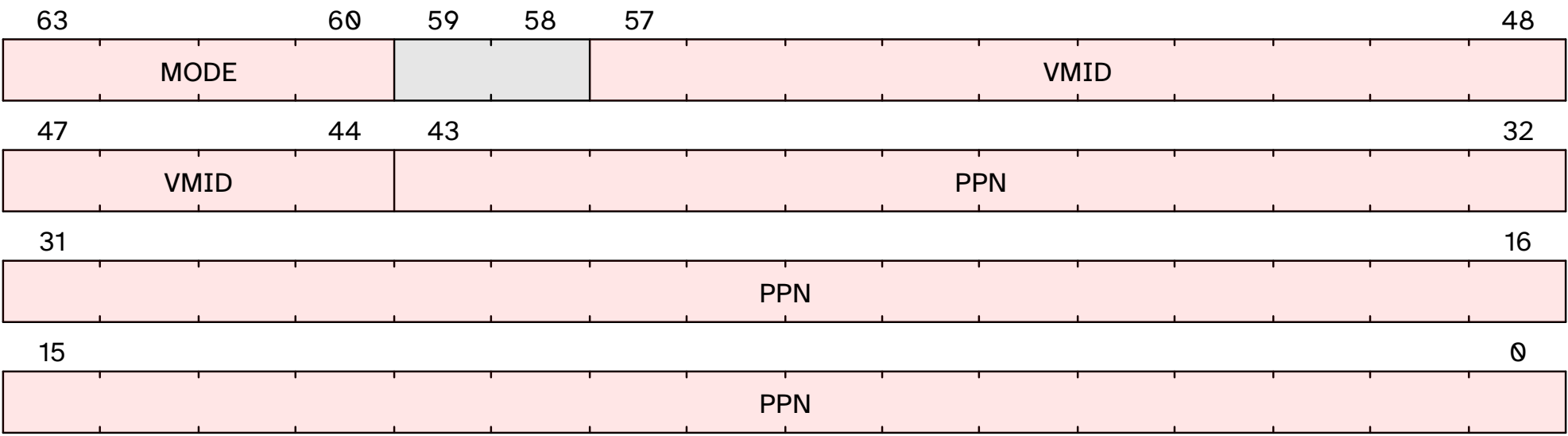


Figure 11. [hcatp](#) Format when CSR[mstatus].SXL == 0



D.11.3. Field Summary

Nam e	Location	Type	Reset Value
hgatp .MODE	31	RW	UNDEFINED_LEGAL
	63:60		
hgatp .VMID	28:22	RO	UNDEFINED_LEGAL
	57:44	RW	
hgatp .PPN	21:0	RO	UNDEFINED_LEGAL
	43:0	RW	

D.11.4. Fields

hgatp.MODE Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63:60 when CSR[mstatus].SXL == 1

Description:

When MODE=Bare, guest physical addresses are equal to supervisor physical addresses, and there is no further memory protection for a guest virtual machine beyond the physical memory protection scheme.
In this case, the remaining fields in hgatp must be set to zeros.

When HSXLEN=32, the only other valid setting for MODE is Sv32x4, which is a modification of the usual Sv32 paged virtual-memory scheme, extended to support 34-bit guest physical addresses.
When HSXLEN=64, modes Sv39x4, Sv48x4, and Sv57x4 are defined as modifications of the Sv39, Sv48, and Sv57 paged virtual-memory schemes.

A write to hgatp with an unsupported MODE value is not ignored as it is for satp.
Instead, the fields of hgatp are WARL in the normal way, when so indicated.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hgatp.VMID Field

Location:

- 28:22 when CSR[mstatus].SXL == 0
- 57:44 when CSR[mstatus].SXL == 1

Description:

Virtual machine ID.

The number of VMID bits is determined by parameter VMID_WIDTH.

VMID_WIDTH may be determined by software by writing one to every bit position in the *VMID* field, then reading back the value in *hgatp* to see which bit positions in the *VMID* field hold a one.

The least-significant bits of *VMID* are implemented first:
that is, if *VMID_WIDTH* > 0, *VMID*[*VMID_WIDTH*-1:0] is writable.

The maximal value of *VMID_WIDTH*, termed *VMIDMAX*, is 7 for Sv32x4 or 14 for Sv39x4, Sv48x4, and Sv57x4.

Type:

RO

RW

Reset value:

UNDEFINED_LEGAL

hgatp.PPN Field

Location:

- 21:0 when *CSR*[*mstatus*].*SXL* == 0
- 43:0 when *CSR*[*mstatus*].*SXL* == 1

Description:

The physical page number (PPN) of the guest-physical root page table.

Type:

RO

RW

Reset value:

UNDEFINED_LEGAL

D.11.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
MODE = if (csr_value.MODE == $bits(HgatpMode::Bare)) {
  if (!GSTAGE_MODE_BARE) {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}

if (csr_value.MODE == $bits(HgatpMode::Sv32x4)) {
  if (!SV32X4_TRANSLATION) {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}

if (csr_value.MODE == $bits(HgatpMode::Sv39x4)) {
  if (!SV39X4_TRANSLATION) {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}

if (csr_value.MODE == $bits(HgatpMode::Sv48x4)) {
  if (!SV48X4_TRANSLATION) {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}

if (csr_value.MODE == $bits(HgatpMode::Sv57x4)) {
  if (!SV57X4_TRANSLATION) {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}

# mode is supported
```

```

return csr_value.MODE;

VMID = if (csr_value.MODE == $bits(HgatpMode::Bare)) {
  # when MODE == Bare, PPN and VMID must be 0
  if (csr_value.VMID == 0) {
    return 0;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}
return csr_value.VMID[VMID_WIDTH-1:0];

PPN = if (csr_value.MODE == $bits(HgatpMode::Bare)) {
  # when MODE == Bare, PPN and VMID must be 0
  if (csr_value.PPN == 0) {
    return 0;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}
return csr_value.PPN;

```

D.11.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```

if ((hgatp.MODE == $bits(HgatpMode::Sv32x4)) || (hgatp.MODE == $bits(HgatpMode::Sv39x4)) || (hgatp.MODE ==
$bits(HgatpMode::Sv48x4)) || (hgatp.MODE == $bits(HgatpMode::Sv57x4))) {
  return $bits(hgatp) & ~64'h3;
} else {
  return $bits(hgatp);
}

```

D.12. 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.12.1. Attributes

CSR Address	0xc0a
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.12.2. Format

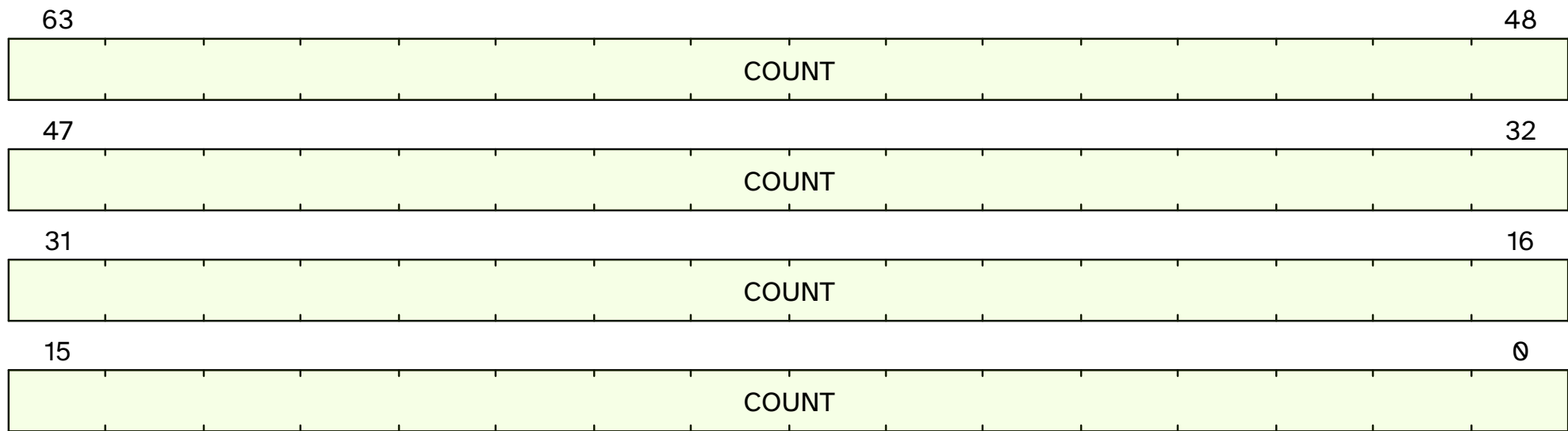


Figure 13. hpmcounter10 format

D.12.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter10.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.12.4. Fields

[hpmcounter10.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter10.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.12.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM10 & scounteren.HPM10) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM10 == 1'b0 && mcounteren.HPM10 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM10 & scounteren.HPM10) == 1'b0 && (mcounteren.HPM10 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
return read_hpm_counter(10);
```

D.13. hpmcounter10h

User-mode Hardware Performance Counter 7, high half

 *hpmcounter10h* is only defined in RV32.

Alias for M-mode CSR [mhpmcounter10h](#).

Privilege mode access is controlled with [mcounteren.HPM10](#), [scounteren.HPM10](#), and [hcounteren.HPM10](#) as follows:

mcounteren.HPM10	scounteren.HPM10	hcounteren.HPM10	hpmcounter10h 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.13.1. Attributes

CSR Address	0xc8a
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.13.2. Format

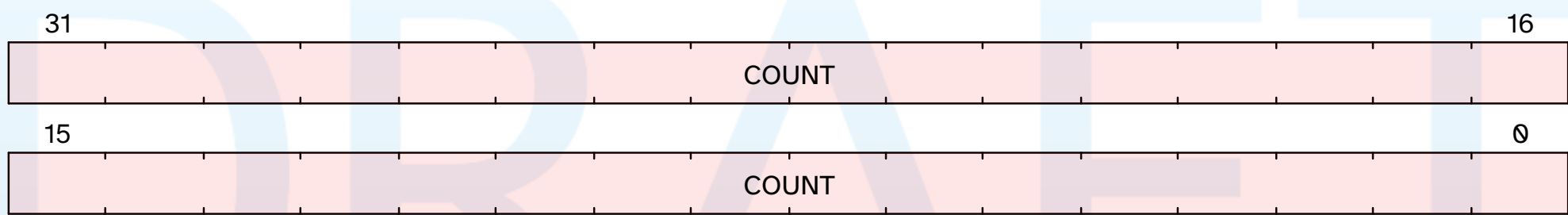


Figure 14. hpmcounter10h format

D.13.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter10h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.13.4. Fields

[hpmcounter10h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter10h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.13.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM10 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM10 & scounteren.HPM10) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM10 == 1'b0 && mcounteren.HPM10 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM10 & scounteren.HPM10) == 1'b0 && (mcounteren.HPM10 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM10 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(10)[63:32];

```

DRAFT

D.14. 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.14.1. Attributes

CSR Address	0xc0b
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.14.2. Format

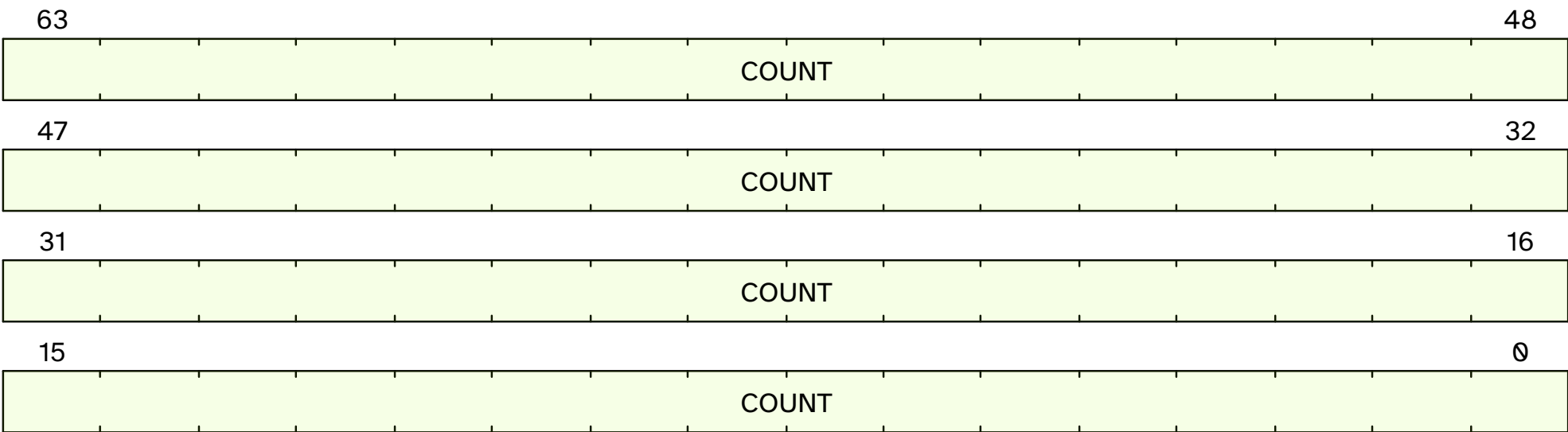


Figure 15. hpmcounter11 format

D.14.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter11.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.14.4. Fields

[hpmcounter11.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter11.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.14.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM11 & scounteren.HPM11) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM11 == 1'b0 && mcounteren.HPM11 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM11 & scounteren.HPM11) == 1'b0 && (mcounteren.HPM11 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(11);
```

D.15. hpmcounter11h

User-mode Hardware Performance Counter 8, high half

i *hpmcounter11h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter11h*.

Privilege mode access is controlled with *mcounteren.HPM11*, *scounteren.HPM11*, and *hcounteren.HPM11* as follows:

<i>mcounteren.HPM11</i>	<i>scounteren.HPM11</i>	<i>hcounteren.HPM11</i>	<i>hpmcounter11h</i> 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.15.1. Attributes

CSR Address	0xc8b
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.15.2. Format

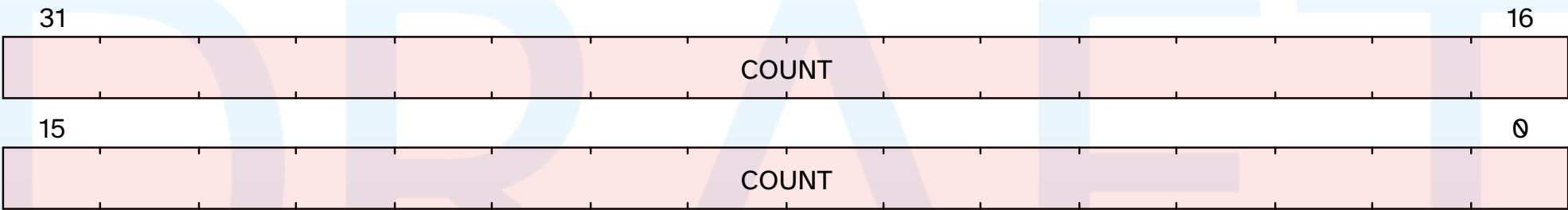


Figure 16. *hpmcounter11h* format

D.15.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter11h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.15.4. Fields

hpmcounter11h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter11h.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.15.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM11 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM11 & scounteren.HPM11) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM11 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM11 == 1'b0 && mcounteren.HPM11 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM11 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM11 & scounteren.HPM11) == 1'b0 && (mcounteren.HPM11 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM11 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(11)[63:32];

```

DRAFT

D.16. 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.16.1. Attributes

CSR Address	0xc0c
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.16.2. Format

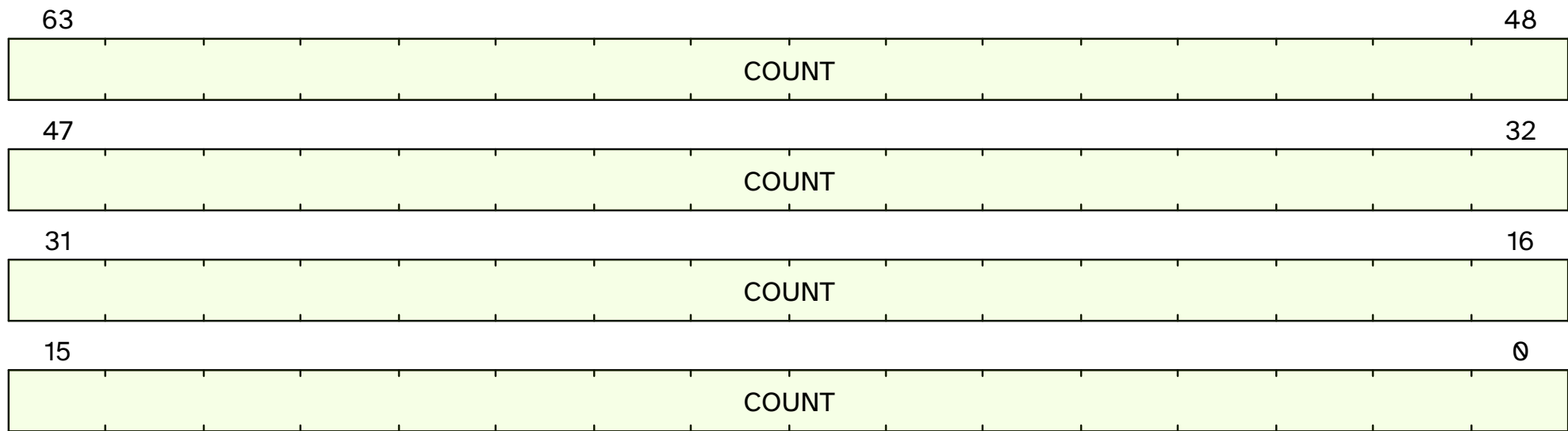


Figure 17. hpmcounter12 format

D.16.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter12.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.16.4. Fields

[hpmcounter12.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter12.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.16.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM12 & scounteren.HPM12) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM12 == 1'b0 && mcounteren.HPM12 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM12 & scounteren.HPM12) == 1'b0 && (mcounteren.HPM12 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(12);
```

D.17. hpmcounter12h

User-mode Hardware Performance Counter 9, high half

i *hpmcounter12h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter12h*.

Privilege mode access is controlled with *mcouteren.HPM12*, *scouteren.HPM12*, and *hcounteren.HPM12* as follows:

<i>mcouteren.HPM12</i>	<i>scouteren.HPM12</i>	<i>hcounteren.HPM12</i>	<i>hpmcounter12h</i> 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.17.1. Attributes

CSR Address	0xc8c
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.17.2. Format

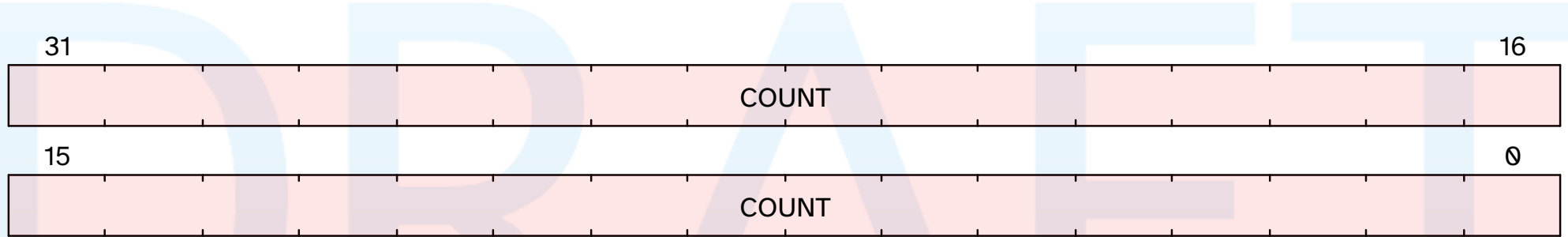


Figure 18. *hpmcounter12h* format

D.17.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter12h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.17.4. Fields

hpmcounter12h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter12h.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 (mcouteren.HPM12 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM12 & scounteren.HPM12) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM12 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM12 == 1'b0 && mcounteren.HPM12 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM12 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM12 & scounteren.HPM12) == 1'b0 && (mcounteren.HPM12 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM12 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(12)[63:32];

```

DRAFT

D.18. 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.18.1. Attributes

CSR Address	0xc0d
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.18.2. Format

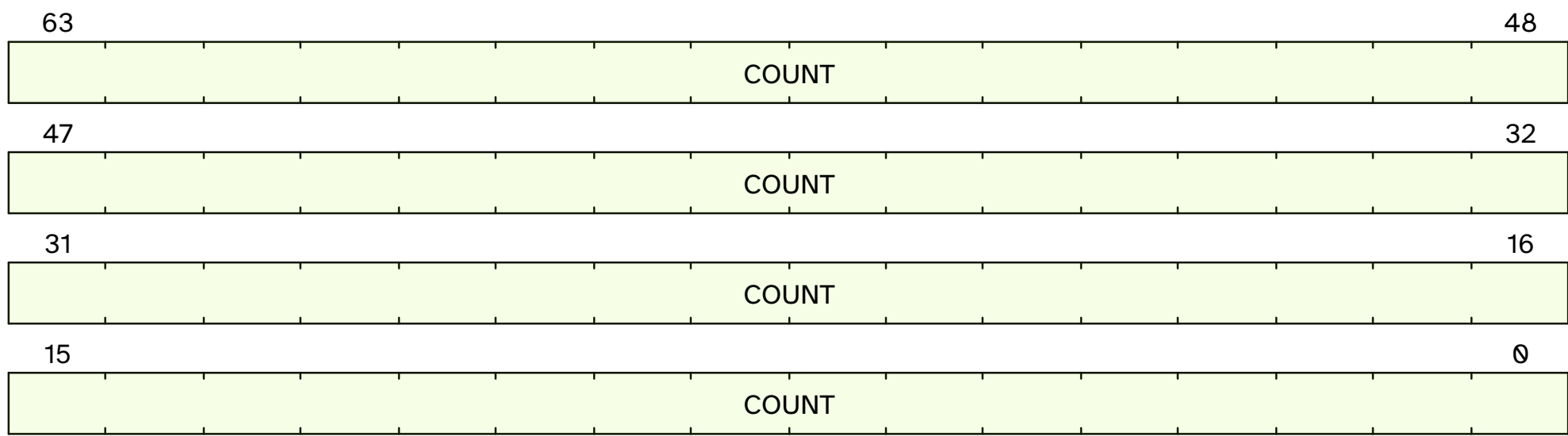


Figure 19. hpmcounter13 format

D.18.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter13.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.18.4. Fields

[hpmcounter13.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter13.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.18.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM13 & scounteren.HPM13) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM13 == 1'b0 && mcounteren.HPM13 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM13 & scounteren.HPM13) == 1'b0 && (mcounteren.HPM13 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(13);
```

D.19. hpmcounter13h

User-mode Hardware Performance Counter 10, high half



hpmcounter13h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter13h](#).

Privilege mode access is controlled with [mcounteren.HPM13](#), [scounteren.HPM13](#), and [hcounteren.HPM13](#) as follows:

mcounteren.HPM13	scounteren.HPM13	hcounteren.HPM13	hpmcounter13h 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.19.1. Attributes

CSR Address	0xc8d
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.19.2. Format

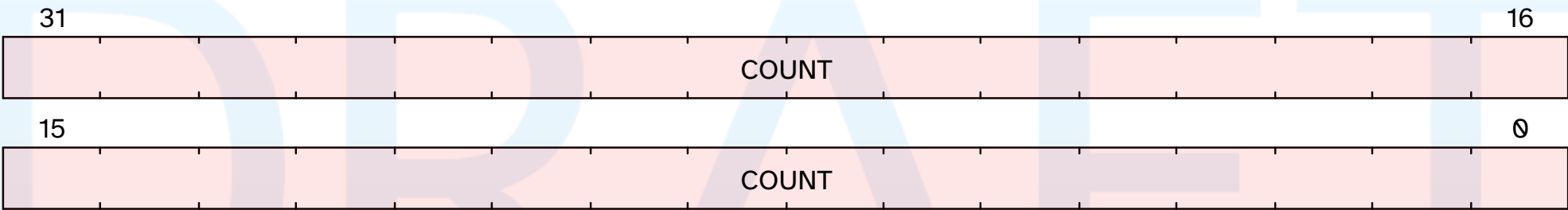


Figure 20. *hpmcounter13h* format

D.19.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter13h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.19.4. Fields

[hpmcounter13h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter13h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.19.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM13 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM13 & scounteren.HPM13) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM13 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM13 == 1'b0 && mcounteren.HPM13 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM13 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM13 & scounteren.HPM13) == 1'b0 && (mcounteren.HPM13 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM13 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(13)[63:32];

```

DRAFT

D.20. 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.20.1. Attributes

CSR Address	0xc0e
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.20.2. Format

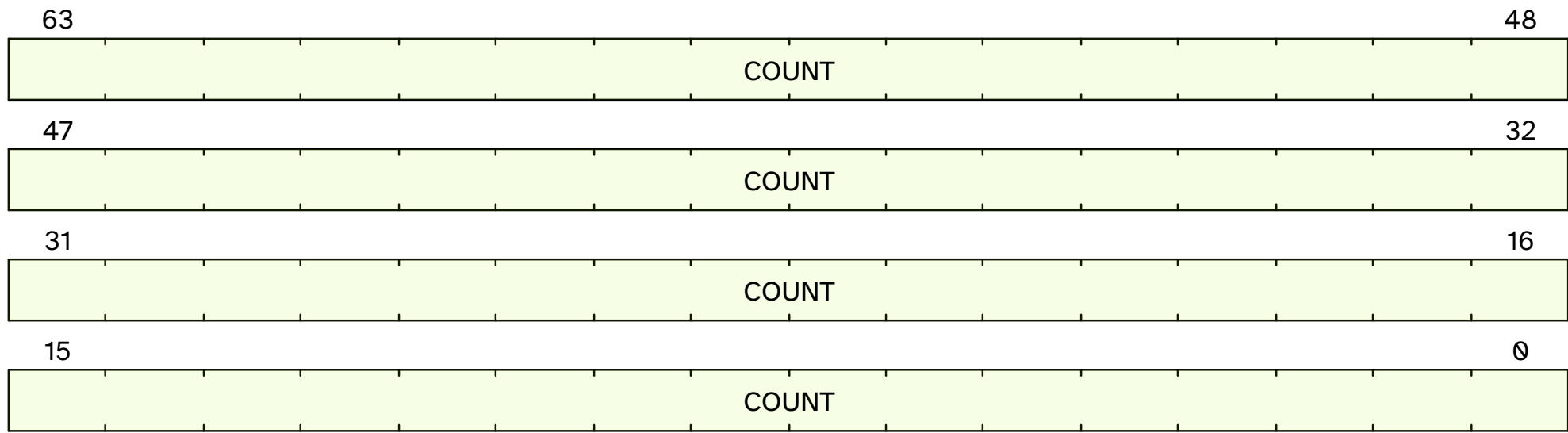


Figure 21. hpmcounter14 format

D.20.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter14.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.20.4. Fields

[hpmcounter14.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter14.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.20.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM14 & scounteren.HPM14) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM14 == 1'b0 && mcounteren.HPM14 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM14 & scounteren.HPM14) == 1'b0 && (mcounteren.HPM14 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(14);
```

D.21. hpmcounter14h

User-mode Hardware Performance Counter 11, high half



hpmcounter14h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter14h](#).

Privilege mode access is controlled with [mcounteren.HPM14](#), [scounteren.HPM14](#), and [hcounteren.HPM14](#) as follows:

mcounteren.HPM14	scounteren.HPM14	hcounteren.HPM14	hpmcounter14h 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.21.1. Attributes

CSR Address	0xc8e
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.21.2. Format

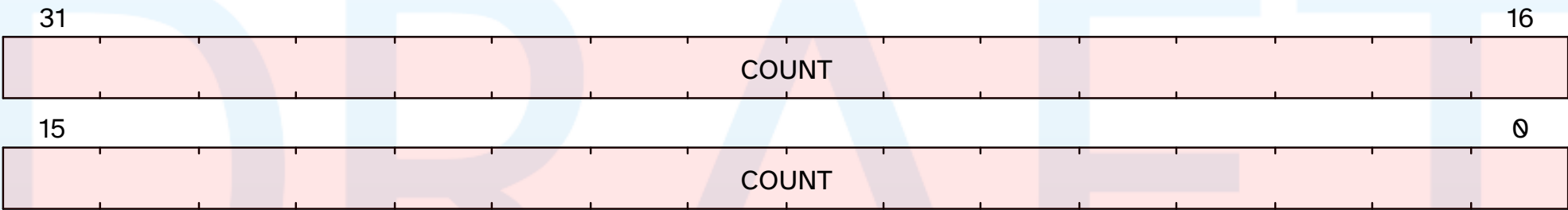


Figure 22. *hpmcounter14h* format

D.21.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter14h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.21.4. Fields

[hpmcounter14h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter14h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.21.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM14 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM14 & scounteren.HPM14) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM14 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM14 == 1'b0 && mcounteren.HPM14 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM14 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM14 & scounteren.HPM14) == 1'b0 && (mcounteren.HPM14 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM14 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(14)[63:32];

```

DRAFT

D.22. 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.22.1. Attributes

CSR Address	0xc0f
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.22.2. Format

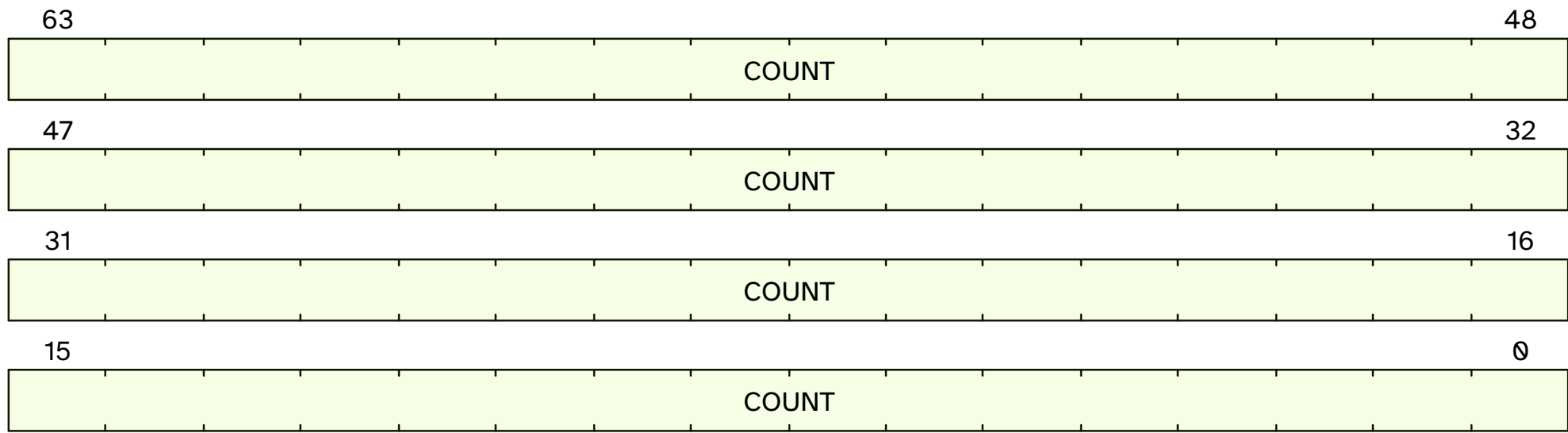


Figure 23. hpmcounter15 format

D.22.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter15.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.22.4. Fields

[hpmcounter15.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter15.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.22.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM15 & scounteren.HPM15) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM15 == 1'b0 && mcounteren.HPM15 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM15 & scounteren.HPM15) == 1'b0 && (mcounteren.HPM15 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(15);
```

D.23. hpmcounter15h

User-mode Hardware Performance Counter 12, high half

i *hpmcounter15h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter15h*.

Privilege mode access is controlled with *mcounteren.HPM15*, *scounteren.HPM15*, and *hcounteren.HPM15* as follows:

<i>mcounteren.HPM15</i>	<i>scounteren.HPM15</i>	<i>hcounteren.HPM15</i>	<i>hpmcounter15h</i> 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.23.1. Attributes

CSR Address	0xc8f
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.23.2. Format

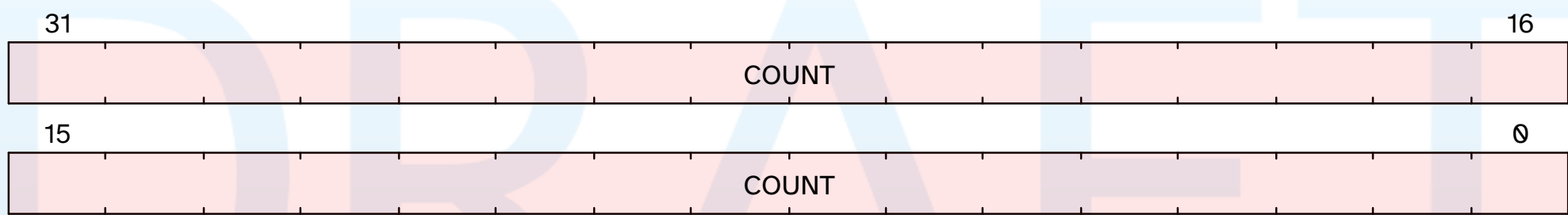


Figure 24. *hpmcounter15h* format

D.23.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter15h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.23.4. Fields

hpmcounter15h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter15h.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.23.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM15 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM15 & scounteren.HPM15) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM15 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM15 == 1'b0 && mcounteren.HPM15 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM15 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM15 & scounteren.HPM15) == 1'b0 && (mcounteren.HPM15 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM15 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(15)[63:32];

```

DRAFT

D.24. 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.24.1. Attributes

CSR Address	0xc10
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.24.2. Format

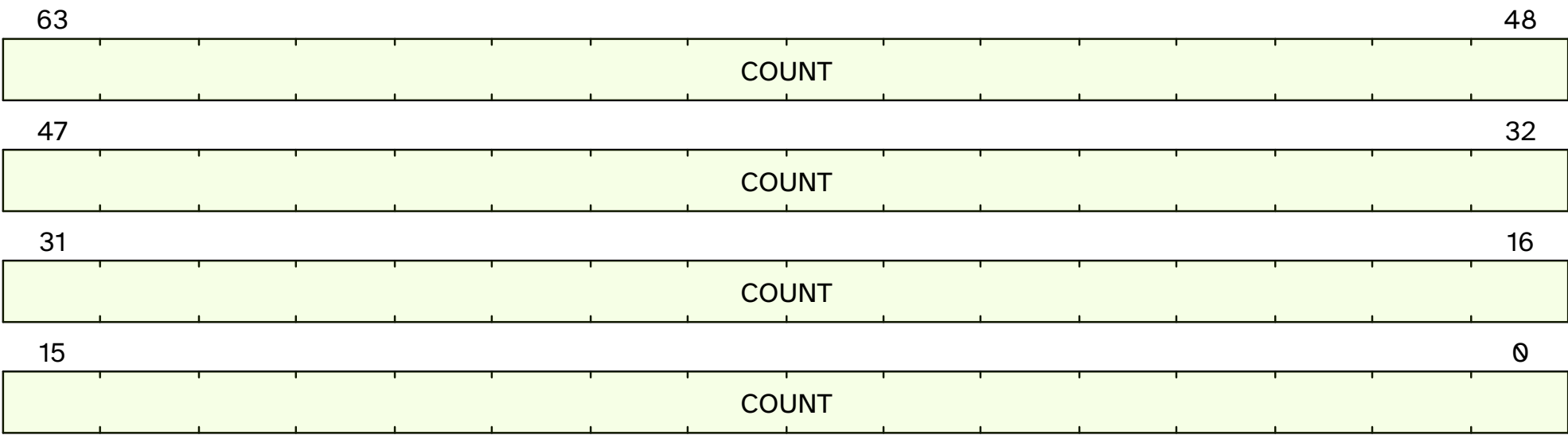


Figure 25. hpmcounter16 format

D.24.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter16.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.24.4. Fields

[hpmcounter16.COUNT](#) Field

Location:

63:0

Description:

Alias of [mhpmcounter16.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.24.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM16 & scounteren.HPM16) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM16 == 1'b0 && mcounteren.HPM16 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM16 & scounteren.HPM16) == 1'b0 && (mcounteren.HPM16 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(16);
```

D.25. hpmcounter16h

User-mode Hardware Performance Counter 13, high half

i *hpmcounter16h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter16h*.

Privilege mode access is controlled with *mcounteren.HPM16*, *scounteren.HPM16*, and *hcounteren.HPM16* as follows:

<i>mcounteren.HPM16</i>	<i>scounteren.HPM16</i>	<i>hcounteren.HPM16</i>	<i>hpmcounter16h</i> 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.25.1. Attributes

CSR Address	0xc90
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.25.2. Format

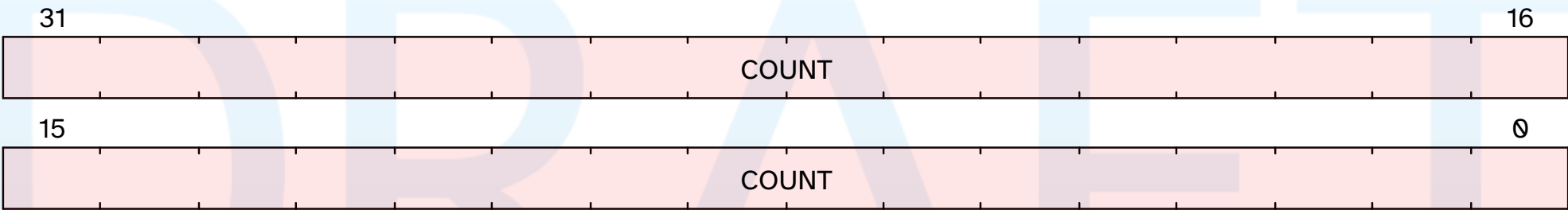


Figure 26. hpmcounter16h format

D.25.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter16h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.25.4. Fields

hpmcounter16h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter16h.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.25.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM16 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM16 & scounteren.HPM16) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM16 == 1'b0 && mcounteren.HPM16 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM16 & scounteren.HPM16) == 1'b0 && (mcounteren.HPM16 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
}
return read_hpm_counter(16)[63:32];

```

DRAFT

D.26. 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.26.1. Attributes

CSR Address	0xc11
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.26.2. Format

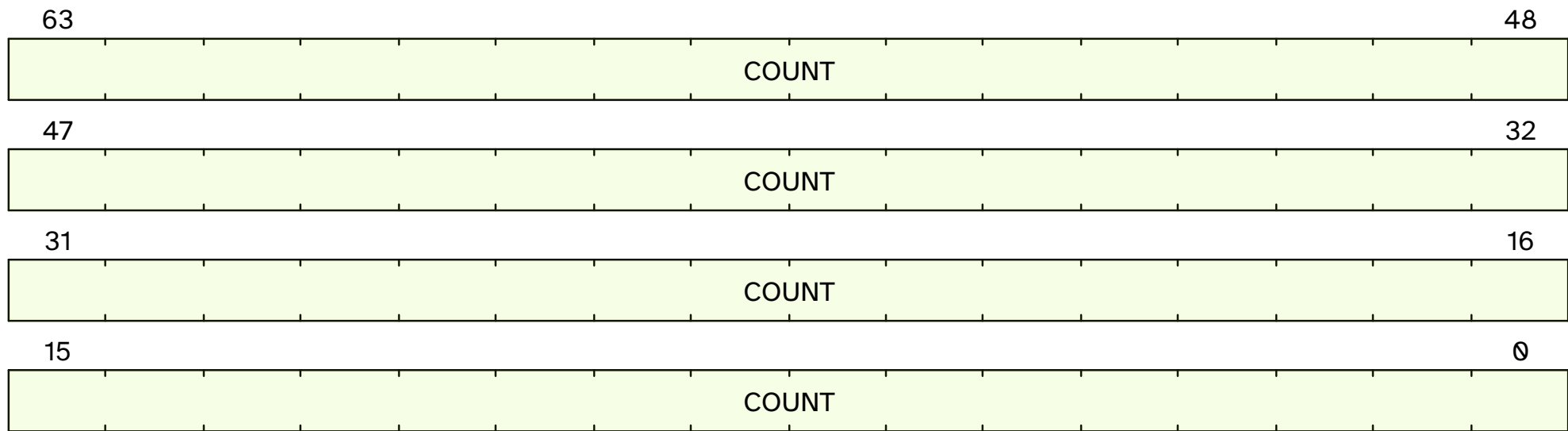


Figure 27. hpmcounter17 format

D.26.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter17.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.26.4. Fields

[hpmcounter17.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter17.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.26.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM17 & scounteren.HPM17) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM17 == 1'b0 && mcounteren.HPM17 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM17 & scounteren.HPM17) == 1'b0 && (mcounteren.HPM17 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(17);
```

D.27. hpmcounter17h

User-mode Hardware Performance Counter 14, high half



hpmcounter17h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter17h](#).

Privilege mode access is controlled with [mcounteren.HPM17](#), [scounteren.HPM17](#), and [hcounteren.HPM17](#) as follows:

mcounteren.HPM17	scounteren.HPM17	hcounteren.HPM17	hpmcounter17h 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.27.1. Attributes

CSR Address	0xc91
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.27.2. Format

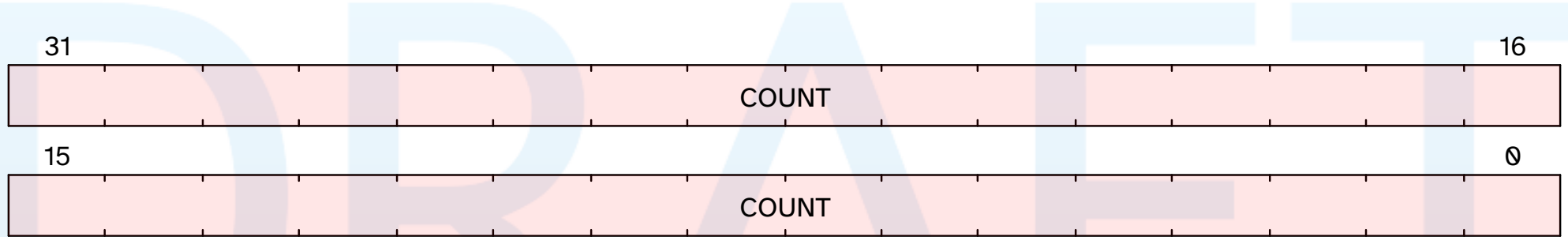


Figure 28. *hpmcounter17h* format

D.27.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter17h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.27.4. Fields

[hpmcounter17h.COUNT](#) Field

Location:
31:0

Description:
Alias of [mhpmcounter17h.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.27.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM17 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM17 & scounteren.HPM17) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM17 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM17 == 1'b0 && mcounteren.HPM17 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM17 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM17 & scounteren.HPM17) == 1'b0 && (mcounteren.HPM17 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM17 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(17)[63:32];

```

DRAFT

D.28. 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.28.1. Attributes

CSR Address	0xc12
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.28.2. Format

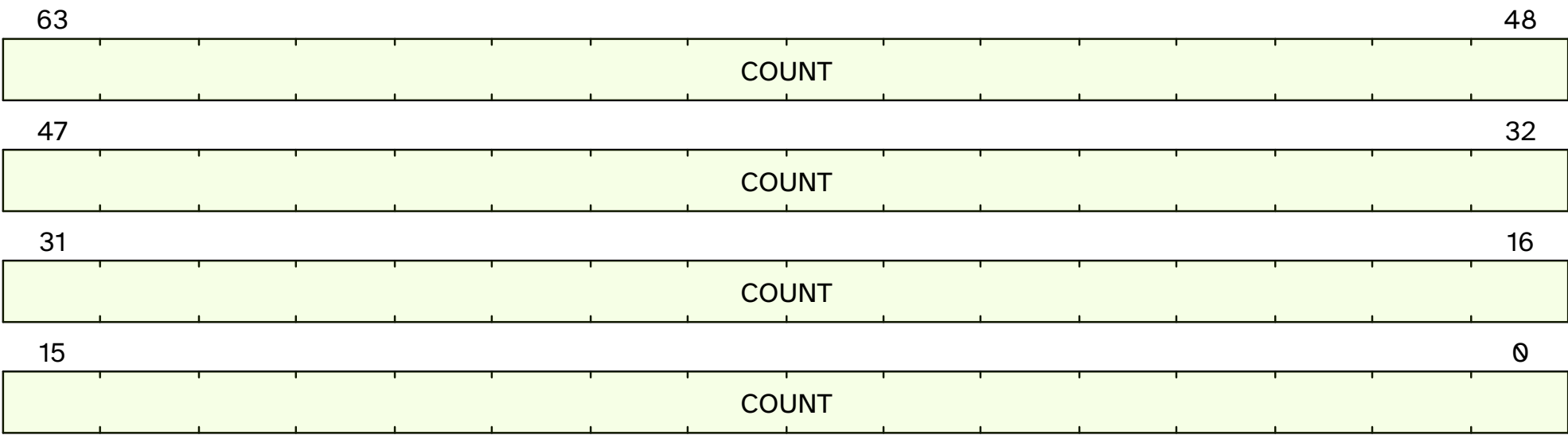


Figure 29. hpmcounter18 format

D.28.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter18.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.28.4. Fields

hpmcounter18.COUNT Field

Location:
63:0

Description:
Alias of *mhpmcounter18.COUNT*.

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.28.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM18 & scounteren.HPM18) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM18 == 1'b0 && mcounteren.HPM18 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM18 & scounteren.HPM18) == 1'b0 && (mcounteren.HPM18 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(18);
```

D.29. hpmcounter18h

User-mode Hardware Performance Counter 15, high half

i *hpmcounter18h* is only defined in RV32.

Alias for M-mode CSR [mhpmcounter18h](#).

Privilege mode access is controlled with [mcounteren.HPM18](#), [scounteren.HPM18](#), and [hcounteren.HPM18](#) as follows:

mcounteren.HPM18	scounteren.HPM18	hcounteren.HPM18	hpmcounter18h 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.29.1. Attributes

CSR Address	0xc92
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.29.2. Format

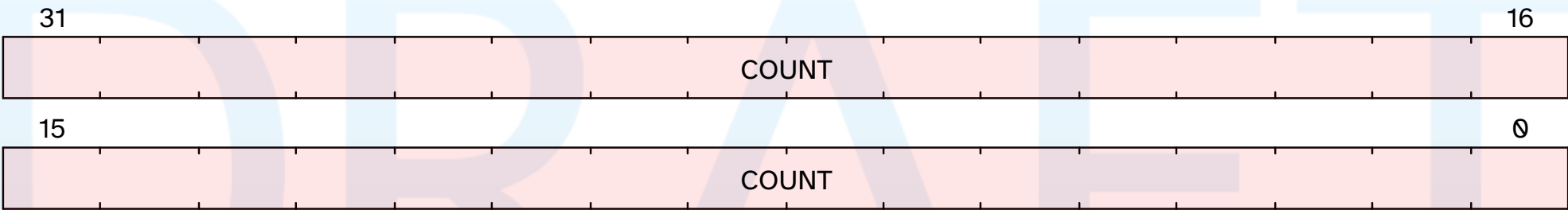


Figure 30. hpmcounter18h format

D.29.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter18h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.29.4. Fields

[hpmcounter18h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter18h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.29.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM18 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM18 & scounteren.HPM18) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM18 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM18 == 1'b0 && mcounteren.HPM18 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM18 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM18 & scounteren.HPM18) == 1'b0 && (mcounteren.HPM18 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM18 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(18)[63:32];

```

DRAFT

D.30. 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.30.1. Attributes

CSR Address	0xc13
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.30.2. Format

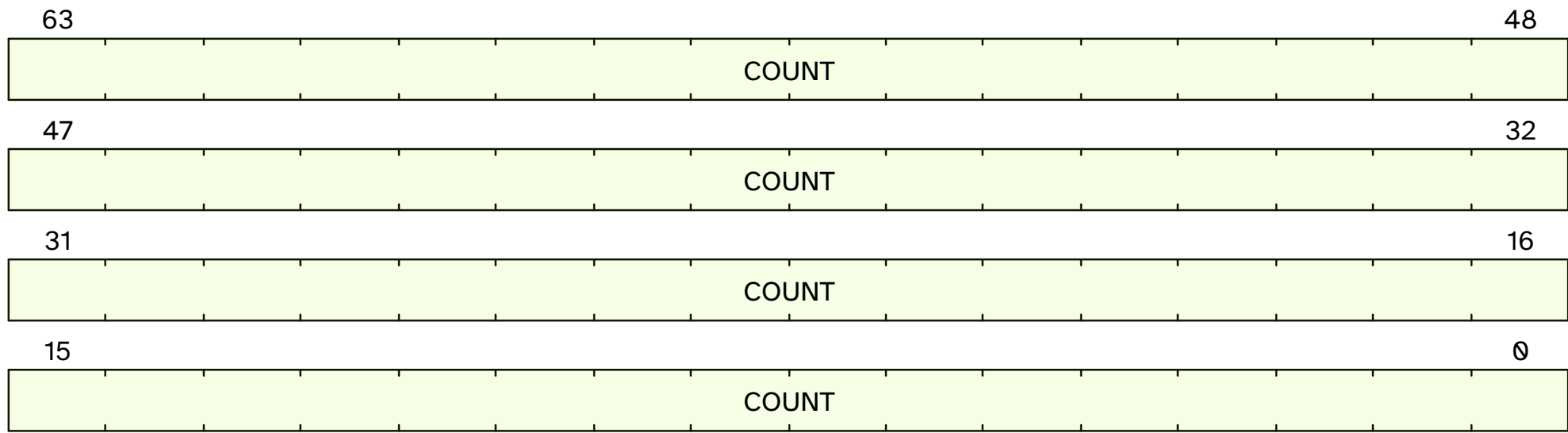


Figure 31. hpmcounter19 format

D.30.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter19.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.30.4. Fields

[hpmcounter19.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter19.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.30.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM19 & scounteren.HPM19) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM19 == 1'b0 && mcounteren.HPM19 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM19 & scounteren.HPM19) == 1'b0 && (mcounteren.HPM19 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(19);
```

D.31. hpmcounter19h

User-mode Hardware Performance Counter 16, high half

i

hpmcounter19h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter19h](#).

Privilege mode access is controlled with [mcounteren.HPM19](#), [scounteren.HPM19](#), and [hcounteren.HPM19](#) as follows:

mcounteren.HPM19	scounteren.HPM19	hcounteren.HPM19	hpmcounter19h 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.31.1. Attributes

CSR Address	0xc93
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.31.2. Format

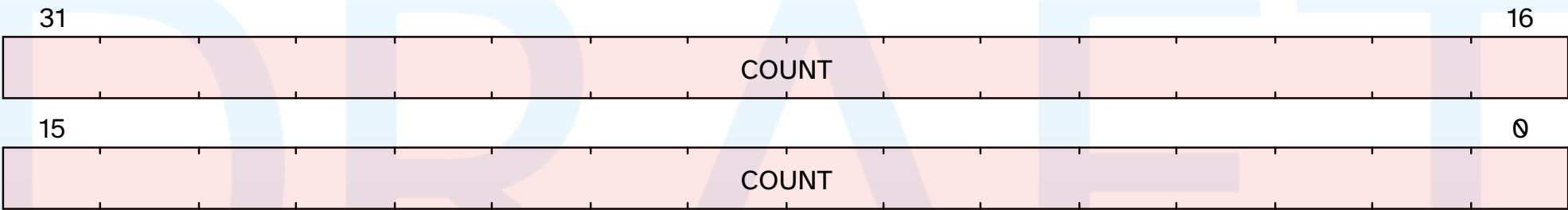


Figure 32. hpmcounter19h format

D.31.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter19h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.31.4. Fields

[hpmcounter19h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter19h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.31.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM19 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM19 & scounteren.HPM19) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM19 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM19 == 1'b0 && mcounteren.HPM19 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM19 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM19 & scounteren.HPM19) == 1'b0 && (mcounteren.HPM19 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM19 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(19)[63:32];

```

DRAFT

D.32. 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.32.1. Attributes

CSR Address	0xc14
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.32.2. Format

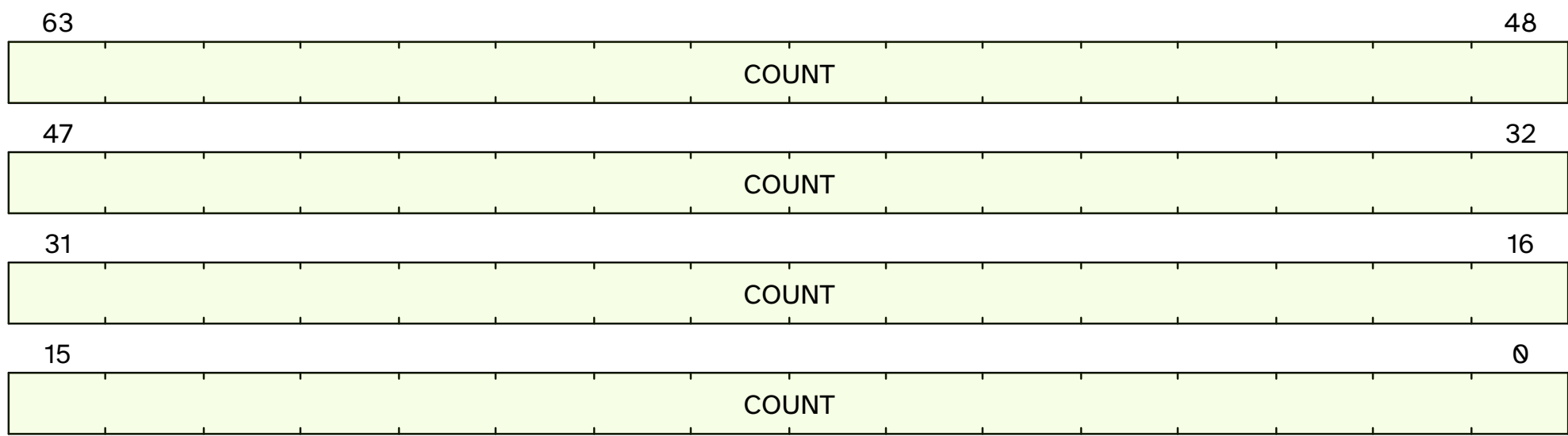


Figure 33. hpmcounter20 format

D.32.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter20.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.32.4. Fields

[hpmcounter20.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter20.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.32.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM20 & scounteren.HPM20) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM20 == 1'b0 && mcounteren.HPM20 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM20 & scounteren.HPM20) == 1'b0 && (mcounteren.HPM20 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(20);
```

D.33. hpmcounter20h

User-mode Hardware Performance Counter 17, high half



hpmcounter20h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter20h](#).

Privilege mode access is controlled with [mcounteren.HPM20](#), [scounteren.HPM20](#), and [hcounteren.HPM20](#) as follows:

mcounteren.HPM20	scounteren.HPM20	hcounteren.HPM20	hpmcounter20h 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.33.1. Attributes

CSR Address	0xc94
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.33.2. Format

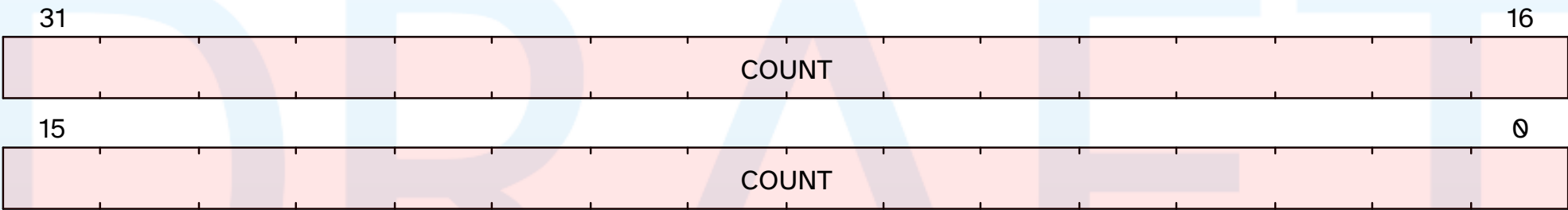


Figure 34. hpmcounter20h format

D.33.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter20h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.33.4. Fields

[hpmcounter20h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter20h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.33.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM20 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM20 & scounteren.HPM20) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM20 == 1'b0 && mcounteren.HPM20 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM20 & scounteren.HPM20) == 1'b0 && (mcounteren.HPM20 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
}
return read_hpm_counter(20)[63:32];

```

DRAFT

D.34. 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.34.1. Attributes

CSR Address	0xc15
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.34.2. Format

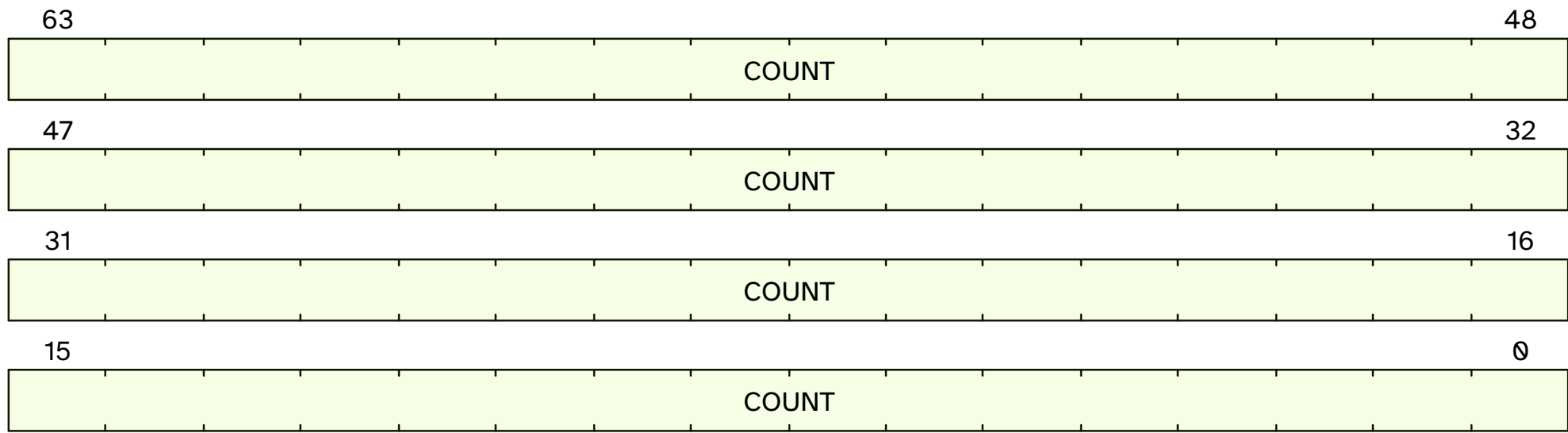


Figure 35. hpmcounter21 format

D.34.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter21.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.34.4. Fields

[hpmcounter21.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter21.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.34.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM21 & scounteren.HPM21) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM21 == 1'b0 && mcounteren.HPM21 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM21 & scounteren.HPM21) == 1'b0 && (mcounteren.HPM21 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(21);
```

D.35. hpmcounter21h

User-mode Hardware Performance Counter 18, high half

i *hpmcounter21h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter21h*.

Privilege mode access is controlled with *mcouteren.HPM21*, *scounteren.HPM21*, and *hcounteren.HPM21* as follows:

<i>mcouteren.HPM21</i>	<i>scounteren.HPM21</i>	<i>hcounteren.HPM21</i>	<i>hpmcounter21h</i> behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

D.35.1. Attributes

CSR Address	0xc95
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.35.2. Format

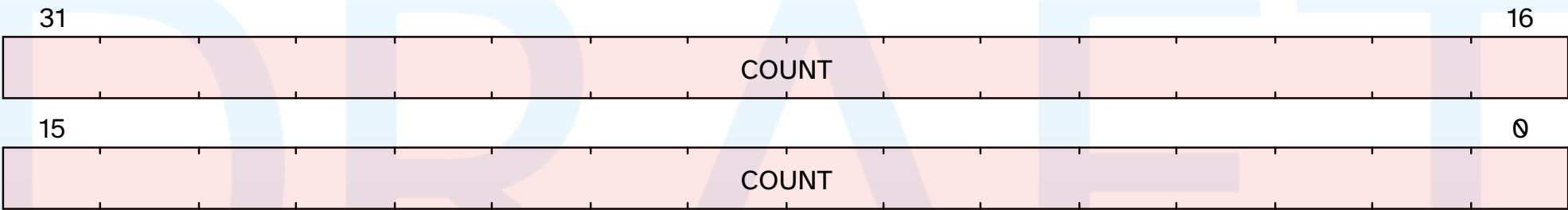


Figure 36. *hpmcounter21h* format

D.35.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter21h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.35.4. Fields

hpmcounter21h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter21h.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.35.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcouteren.HPM21 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM21 & scounteren.HPM21) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM21 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM21 == 1'b0 && mcounteren.HPM21 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM21 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM21 & scounteren.HPM21) == 1'b0 && (mcounteren.HPM21 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM21 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(21)[63:32];

```

DRAFT

D.36. 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.36.1. Attributes

CSR Address	0xc16
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.36.2. Format

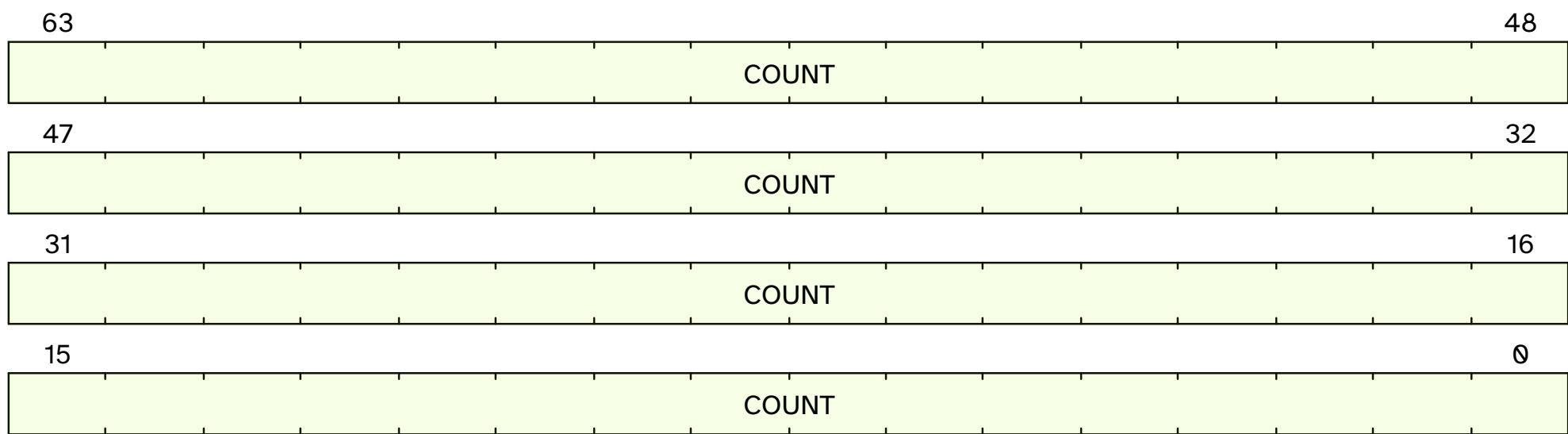


Figure 37. hpmcounter22 format

D.36.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter22.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.36.4. Fields

hpmcounter22.COUNT Field

Location:
63:0

Description:
Alias of *mhpmcounter22.COUNT*.

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.36.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM22 & scounteren.HPM22) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM22 == 1'b0 && mcounteren.HPM22 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM22 & scounteren.HPM22) == 1'b0 && (mcounteren.HPM22 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(22);
```

D.37. hpmcounter22h

User-mode Hardware Performance Counter 19, high half

i

hpmcounter22h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter22h](#).

Privilege mode access is controlled with [mcounteren.HPM22](#), [scounteren.HPM22](#), and [hcounteren.HPM22](#) as follows:

mcounteren.HPM22	scounteren.HPM22	hcounteren.HPM22	hpmcounter22h 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.37.1. Attributes

CSR Address	0xc96
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.37.2. Format

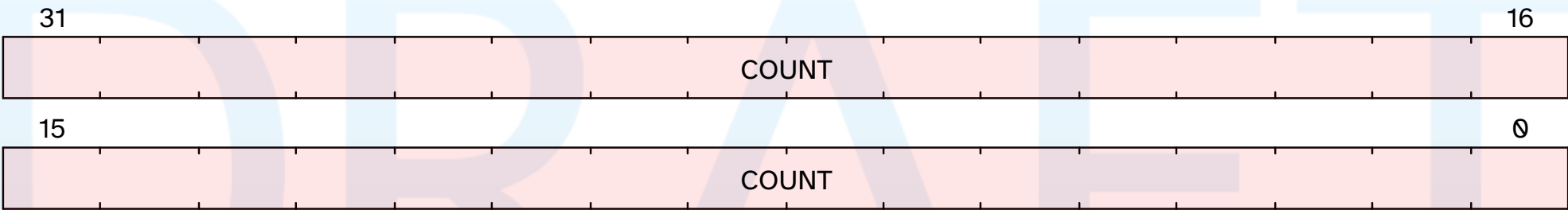


Figure 38. *hpmcounter22h* format

D.37.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter22h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.37.4. Fields

[hpmcounter22h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter22h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.37.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM22 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM22 & scounteren.HPM22) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM22 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM22 == 1'b0 && mcounteren.HPM22 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM22 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM22 & scounteren.HPM22) == 1'b0 && (mcounteren.HPM22 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM22 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(22)[63:32];

```

DRAFT

D.38. 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.38.1. Attributes

CSR Address	0xc17
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.38.2. Format

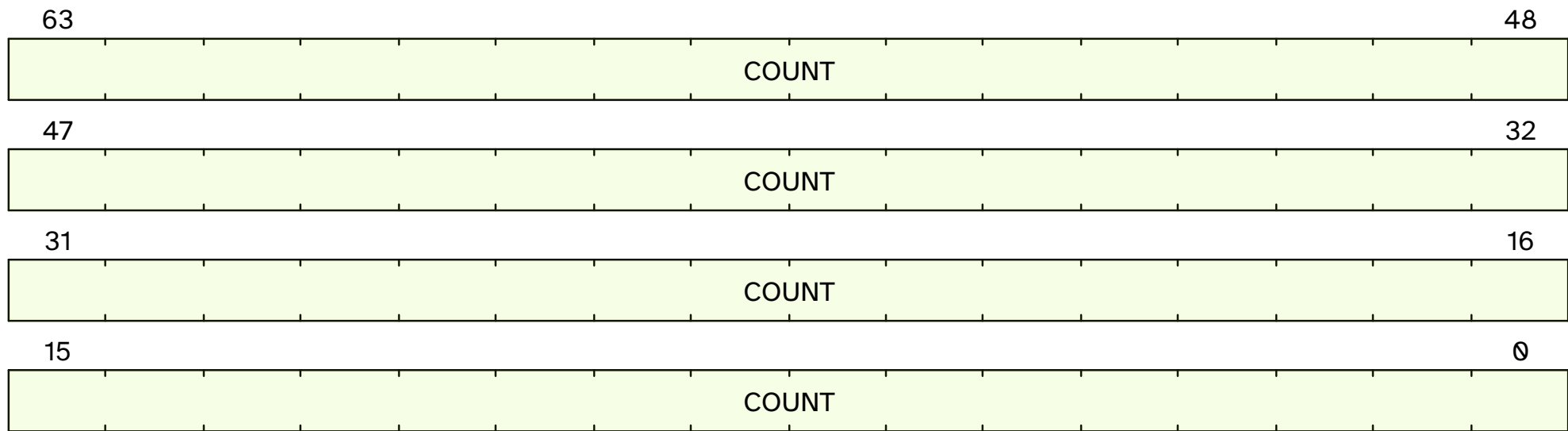


Figure 39. hpmcounter23 format

D.38.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter23.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.38.4. Fields

[hpmcounter23.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter23.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.38.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM23 & scounteren.HPM23) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM23 == 1'b0 && mcounteren.HPM23 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM23 & scounteren.HPM23) == 1'b0 && (mcounteren.HPM23 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(23);
```

D.39. hpmcounter23h

User-mode Hardware Performance Counter 20, high half

 *hpmcounter23h* is only defined in RV32.

Alias for M-mode CSR [mhpmcounter23h](#).

Privilege mode access is controlled with [mcounteren.HPM23](#), [scounteren.HPM23](#), and [hcounteren.HPM23](#) as follows:

mcounteren.HPM23	scounteren.HPM23	hcounteren.HPM23	hpmcounter23h 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.39.1. Attributes

CSR Address	0xc97
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.39.2. Format

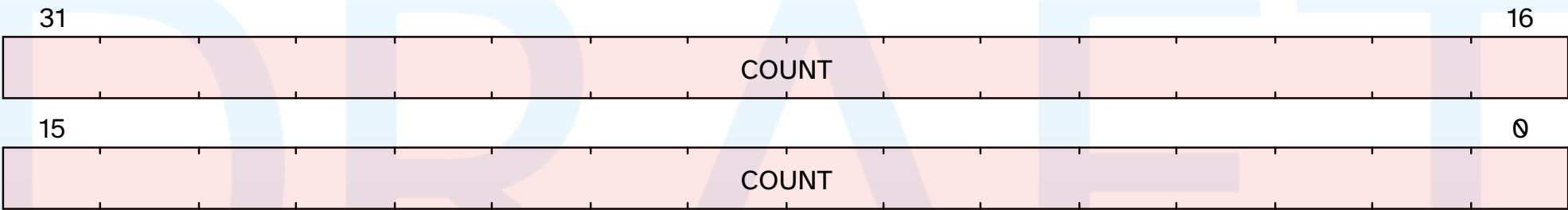


Figure 40. hpmcounter23h format

D.39.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter23h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.39.4. Fields

[hpmcounter23h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter23h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.39.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM23 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM23 & scounteren.HPM23) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM23 == 1'b0 && mcounteren.HPM23 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM23 & scounteren.HPM23) == 1'b0 && (mcounteren.HPM23 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
}
return read_hpm_counter(23)[63:32];

```

DRAFT

D.40. 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.40.1. Attributes

CSR Address	0xc18
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.40.2. Format

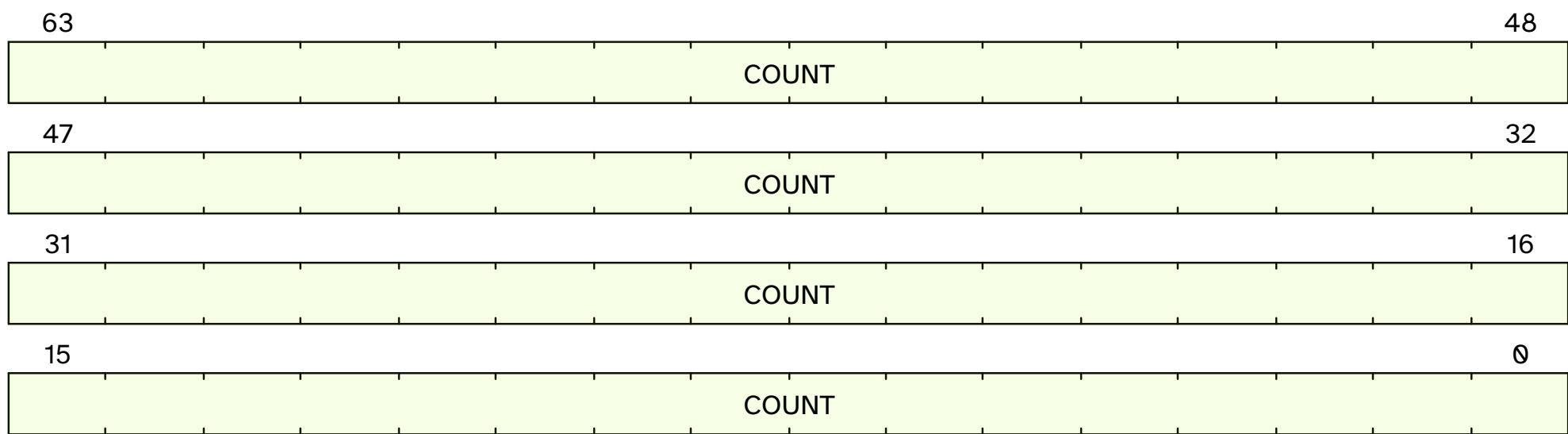


Figure 41. hpmcounter24 format

D.40.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter24.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.40.4. Fields

[hpmcounter24.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter24.COUNT](#).

Type:
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.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM24 & scounteren.HPM24) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM24 == 1'b0 && mcounteren.HPM24 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM24 & scounteren.HPM24) == 1'b0 && (mcounteren.HPM24 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(24);
```

D.41. hpmcounter24h

User-mode Hardware Performance Counter 21, high half

 *hpmcounter24h* is only defined in RV32.

Alias for M-mode CSR [mhpmcounter24h](#).

Privilege mode access is controlled with [mcounteren.HPM24](#), [scounteren.HPM24](#), and [hcounteren.HPM24](#) as follows:

mcounteren.HPM24	scounteren.HPM24	hcounteren.HPM24	hpmcounter24h 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.41.1. Attributes

CSR Address	0xc98
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.41.2. Format

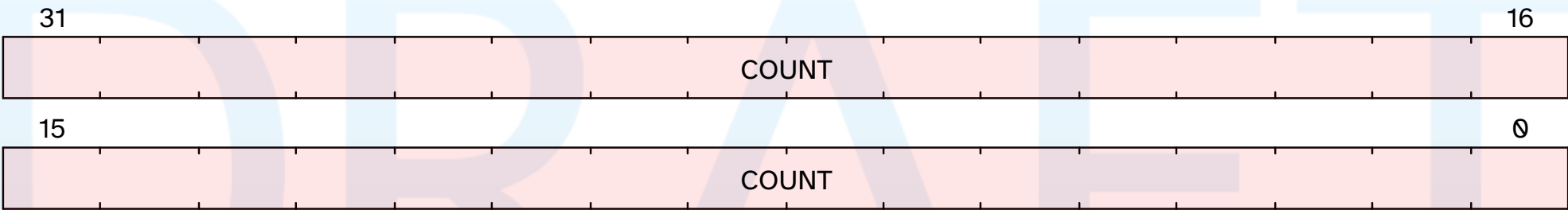


Figure 42. hpmcounter24h format

D.41.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter24h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.41.4. Fields

[hpmcounter24h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter24h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.41.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM24 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM24 & scounteren.HPM24) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM24 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM24 == 1'b0 && mcounteren.HPM24 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM24 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM24 & scounteren.HPM24) == 1'b0 && (mcounteren.HPM24 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM24 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(24)[63:32];

```

DRAFT

D.42. 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.42.1. Attributes

CSR Address	0xc19
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.42.2. Format

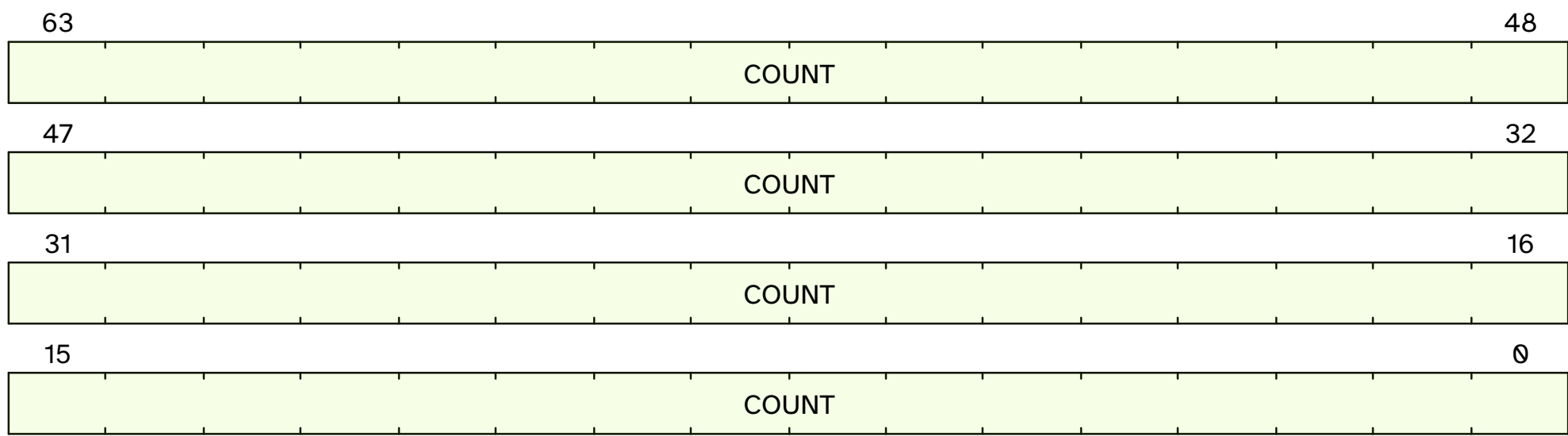


Figure 43. hpmcounter25 format

D.42.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter25.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.42.4. Fields

[hpmcounter25.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter25.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.42.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM25 & scounteren.HPM25) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM25 == 1'b0 && mcounteren.HPM25 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM25 & scounteren.HPM25) == 1'b0 && (mcounteren.HPM25 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(25);
```

D.43. hpmcounter25h

User-mode Hardware Performance Counter 22, high half



hpmcounter25h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter25h](#).

Privilege mode access is controlled with [mcounteren.HPM25](#), [scounteren.HPM25](#), and [hcounteren.HPM25](#) as follows:

mcounteren.HPM25	scounteren.HPM25	hcounteren.HPM25	hpmcounter25h 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.43.1. Attributes

CSR Address	0xc99
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.43.2. Format

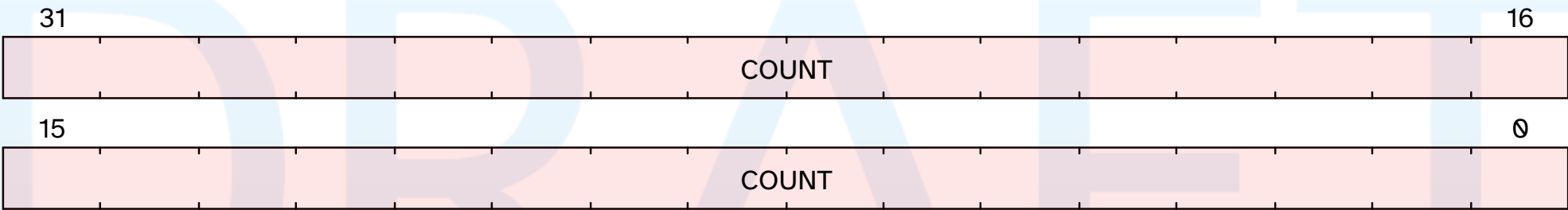


Figure 44. *hpmcounter25h* format

D.43.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter25h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.43.4. Fields

[hpmcounter25h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter25h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.43.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM25 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM25 & scounteren.HPM25) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM25 == 1'b0 && mcounteren.HPM25 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM25 & scounteren.HPM25) == 1'b0 && (mcounteren.HPM25 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
}
return read_hpm_counter(25)[63:32];

```

DRAFT

D.44. 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.44.1. Attributes

CSR Address	0xc1a
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.44.2. Format

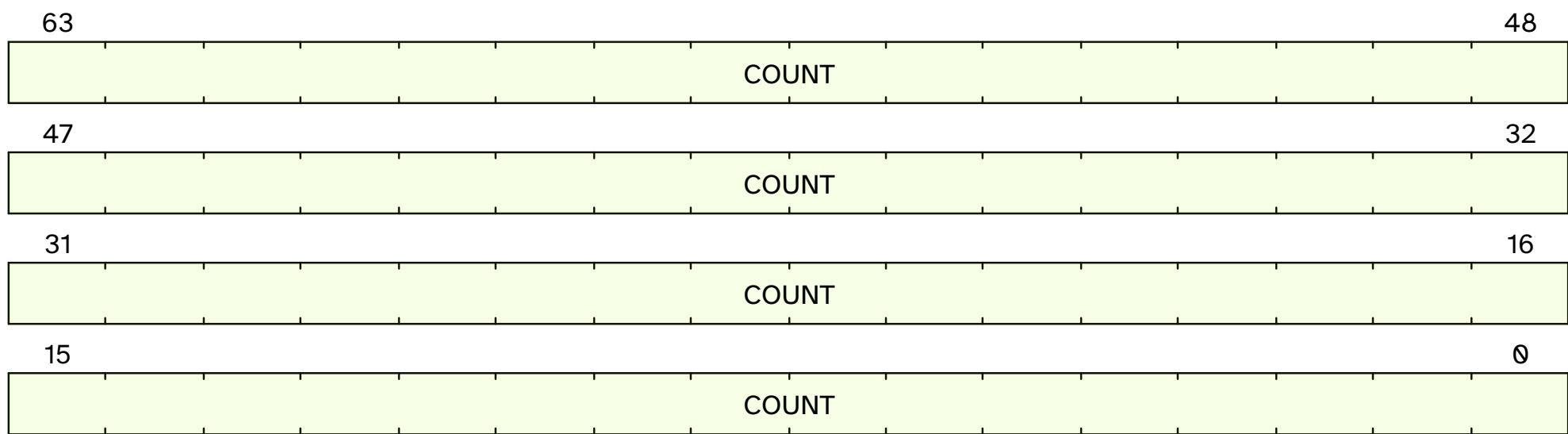


Figure 45. hpmcounter26 format

D.44.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter26.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.44.4. Fields

[hpmcounter26.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter26.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.44.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM26 & scounteren.HPM26) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM26 == 1'b0 && mcounteren.HPM26 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM26 & scounteren.HPM26) == 1'b0 && (mcounteren.HPM26 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(26);
```

D.45. hpmcounter26h

User-mode Hardware Performance Counter 23, high half



hpmcounter26h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter26h](#).

Privilege mode access is controlled with [mcounteren.HPM26](#), [scounteren.HPM26](#), and [hcounteren.HPM26](#) as follows:

mcounteren.HPM26	scounteren.HPM26	hcounteren.HPM26	hpmcounter26h 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.45.1. Attributes

CSR Address	0xc9a
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.45.2. Format

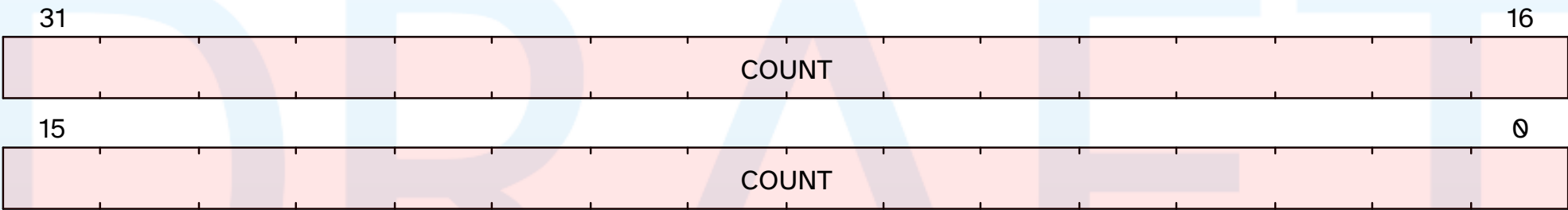


Figure 46. *hpmcounter26h* format

D.45.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter26h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.45.4. Fields

[hpmcounter26h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter26h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.45.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM26 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM26 & scounteren.HPM26) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM26 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM26 == 1'b0 && mcounteren.HPM26 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM26 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM26 & scounteren.HPM26) == 1'b0 && (mcounteren.HPM26 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM26 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(26)[63:32];

```

DRAFT

D.46. 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.46.1. Attributes

CSR Address	0xc1b
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.46.2. Format

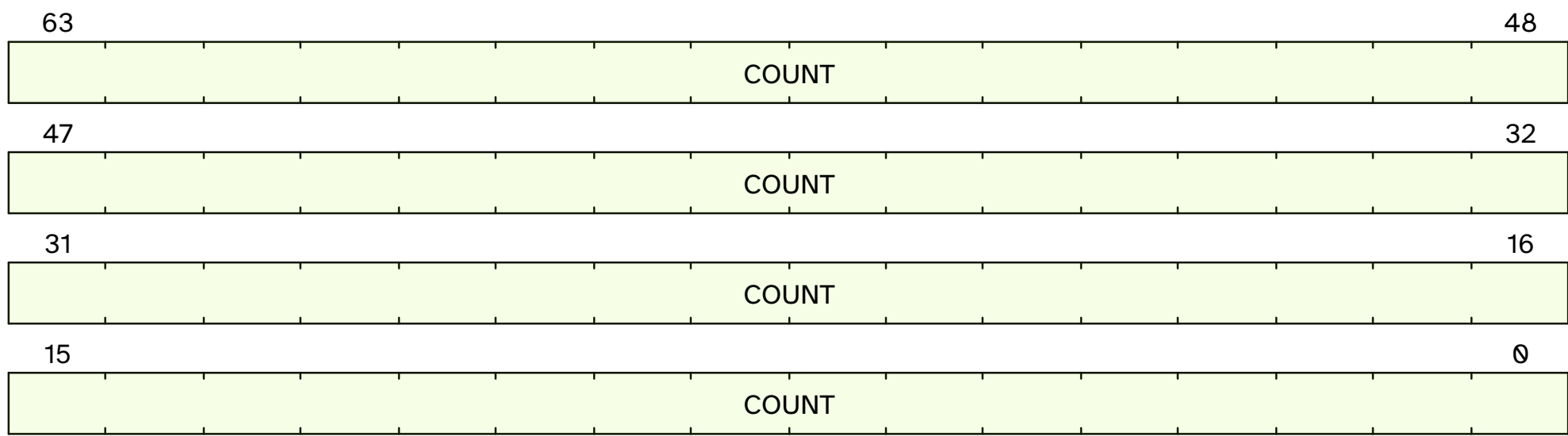


Figure 47. hpmcounter27 format

D.46.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter27.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.46.4. Fields

[hpmcounter27.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter27.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.46.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM27 & scounteren.HPM27) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM27 == 1'b0 && mcounteren.HPM27 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM27 & scounteren.HPM27) == 1'b0 && (mcounteren.HPM27 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(27);
```

D.47. hpmcounter27h

User-mode Hardware Performance Counter 24, high half



hpmcounter27h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter27h](#).

Privilege mode access is controlled with [mcounteren.HPM27](#), [scounteren.HPM27](#), and [hcounteren.HPM27](#) as follows:

mcounteren.HPM27	scounteren.HPM27	hcounteren.HPM27	hpmcounter27h 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.47.1. Attributes

CSR Address	0xc9b
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.47.2. Format

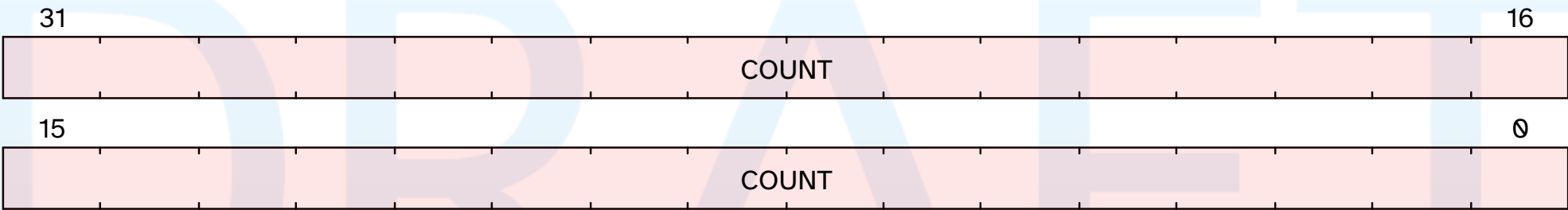


Figure 48. *hpmcounter27h* format

D.47.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter27h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.47.4. Fields

[hpmcounter27h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter27h.COUNT](#).

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 (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM27 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM27 & scounteren.HPM27) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM27 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM27 == 1'b0 && mcounteren.HPM27 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM27 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM27 & scounteren.HPM27) == 1'b0 && (mcounteren.HPM27 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM27 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(27)[63:32];

```

DRAFT

D.48. 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.48.1. Attributes

CSR Address	0xc1c
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.48.2. Format

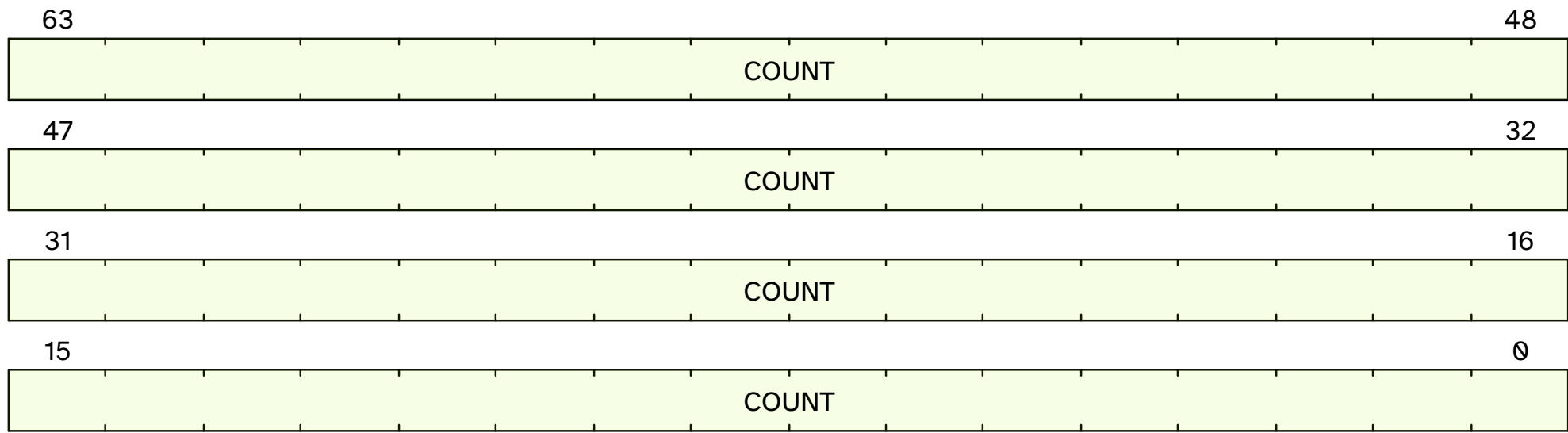


Figure 49. hpmcounter28 format

D.48.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter28.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.48.4. Fields

[hpmcounter28.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter28.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.48.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM28 & scounteren.HPM28) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM28 == 1'b0 && mcounteren.HPM28 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM28 & scounteren.HPM28) == 1'b0 && (mcounteren.HPM28 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(28);
```

D.49. hpmcounter28h

User-mode Hardware Performance Counter 25, high half

i *hpmcounter28h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter28h*.

Privilege mode access is controlled with *mcounteren.HPM28*, *scounteren.HPM28*, and *hcounteren.HPM28* as follows:

<i>mcounteren.HPM28</i>	<i>scounteren.HPM28</i>	<i>hcounteren.HPM28</i>	<i>hpmcounter28h</i> 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.49.1. Attributes

CSR Address	0xc9c
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.49.2. Format

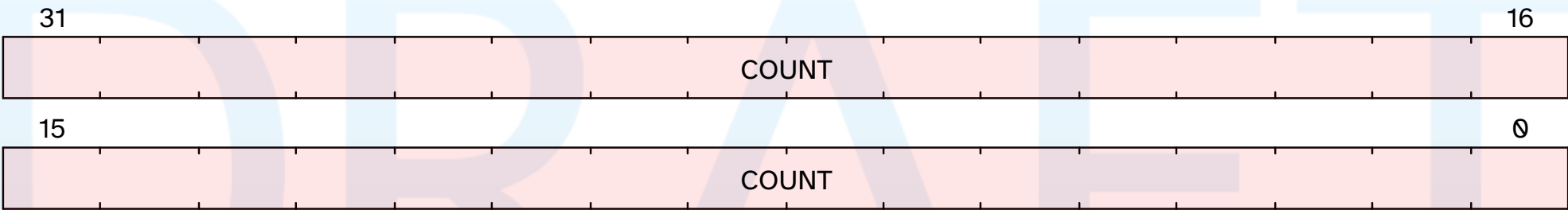


Figure 50. *hpmcounter28h* format

D.49.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter28h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.49.4. Fields

hpmcounter28h.COUNT Field

Location:
31:0

Description:
*Alias of *mhpmcounter28h.COUNT*.*

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.49.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM28 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM28 & scounteren.HPM28) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM28 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM28 == 1'b0 && mcounteren.HPM28 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM28 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM28 & scounteren.HPM28) == 1'b0 && (mcounteren.HPM28 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM28 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(28)[63:32];

```

DRAFT

D.50. 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.50.1. Attributes

CSR Address	0xc1d
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.50.2. Format

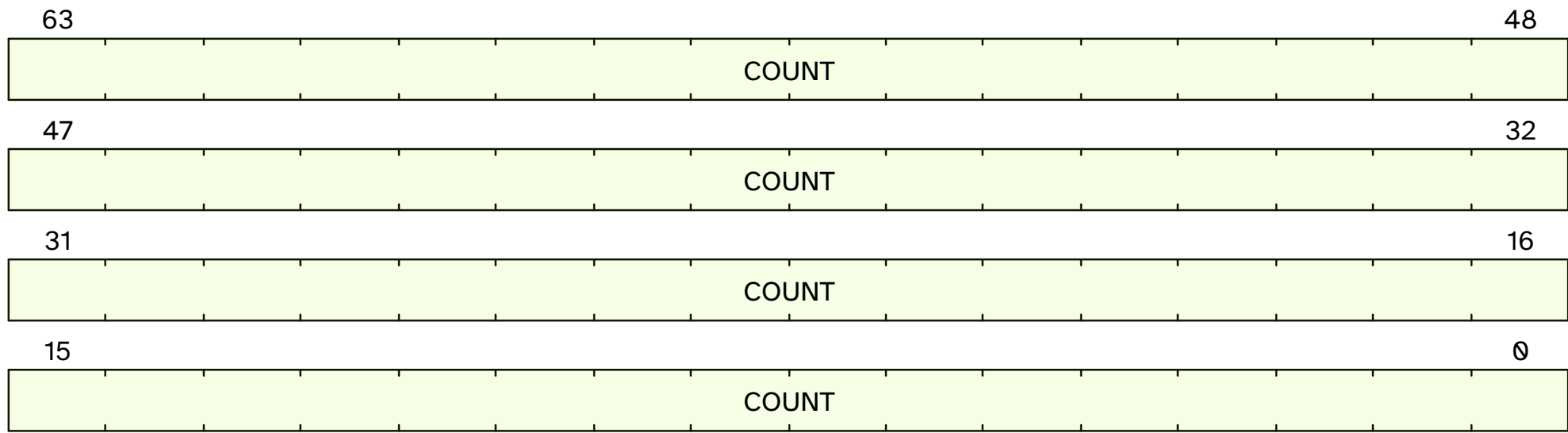


Figure 51. hpmcounter29 format

D.50.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter29.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.50.4. Fields

[hpmcounter29.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter29.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.50.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM29 & scounteren.HPM29) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM29 == 1'b0 && mcounteren.HPM29 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM29 & scounteren.HPM29) == 1'b0 && (mcounteren.HPM29 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(29);
```

D.51. hpmcounter29h

User-mode Hardware Performance Counter 26, high half

i *hpmcounter29h* is only defined in RV32.

Alias for M-mode CSR [mhpmcounter29h](#).

Privilege mode access is controlled with [mcounteren.HPM29](#), [scounteren.HPM29](#), and [hcounteren.HPM29](#) as follows:

mcounteren.HPM29	scounteren.HPM29	hcounteren.HPM29	hpmcounter29h 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.51.1. Attributes

CSR Address	0xc9d
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.51.2. Format

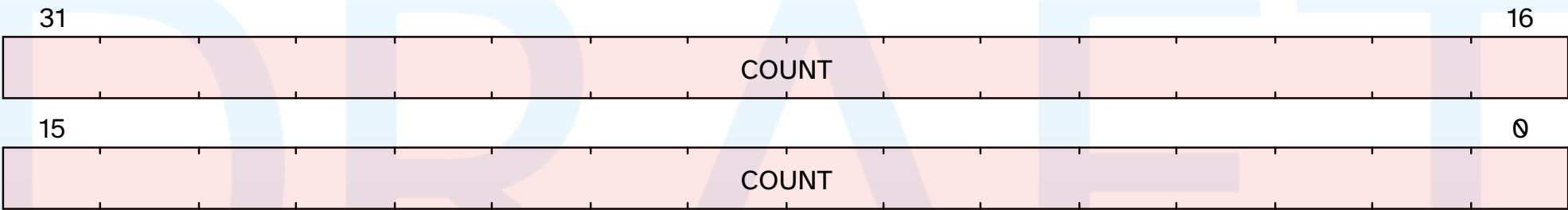


Figure 52. hpmcounter29h format

D.51.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter29h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.51.4. Fields

[hpmcounter29h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter29h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.51.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM29 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM29 & scounteren.HPM29) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM29 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM29 == 1'b0 && mcounteren.HPM29 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM29 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM29 & scounteren.HPM29) == 1'b0 && (mcounteren.HPM29 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM29 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(29)[63:32];

```

DRAFT

D.52. 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.52.1. Attributes

CSR Address	0xc03
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.52.2. Format

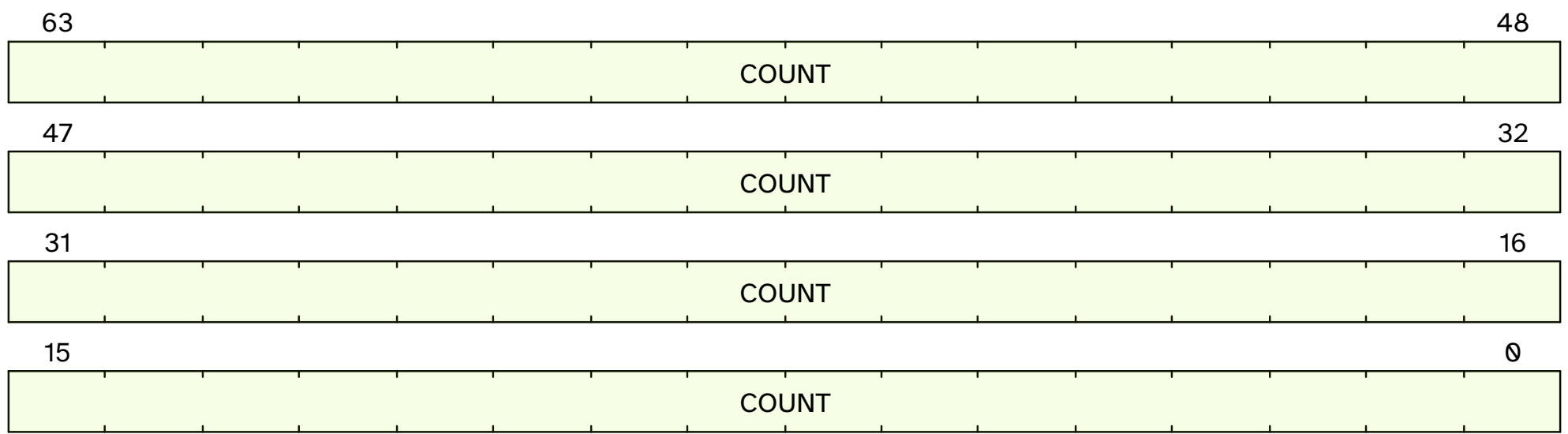


Figure 53. hpmcounter3 format

D.52.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter3.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.52.4. Fields

hpmcounter3.COUNT Field

Location:
63:0

Description:
Alias of *mhpmcounter3.COUNT*.

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.52.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM3 & scounteren.HPM3) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM3 == 1'b0 && mcounteren.HPM3 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM3 & scounteren.HPM3) == 1'b0 && (mcounteren.HPM3 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(3);
```

D.53. 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.53.1. Attributes

CSR Address	0xc1e
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.53.2. Format

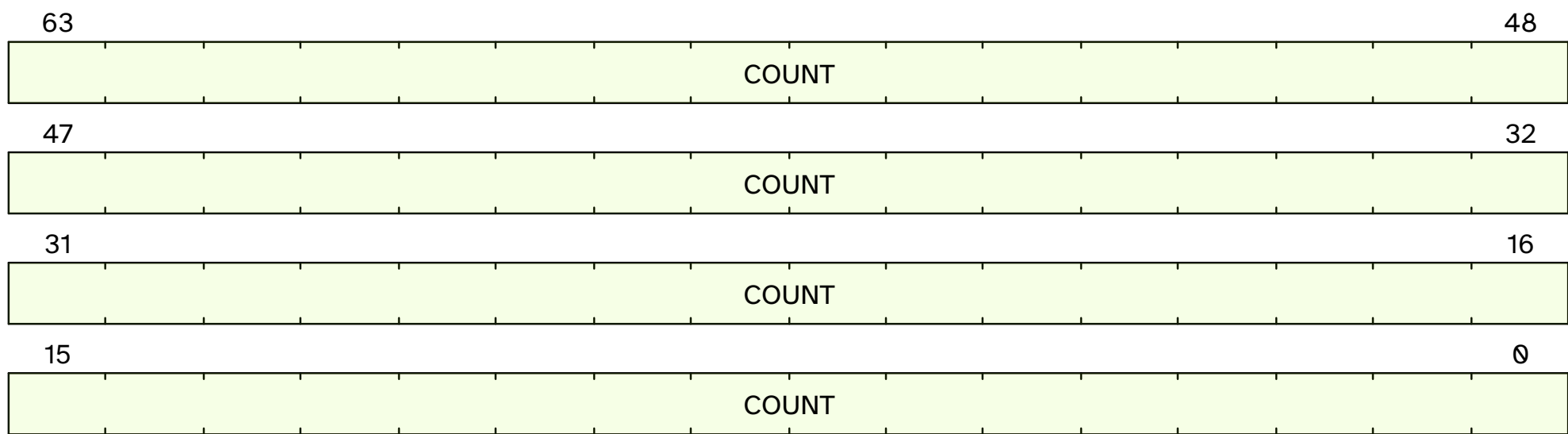


Figure 54. hpmcounter30 format

D.53.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter30.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.53.4. Fields

[hpmcounter30.COUNT](#) Field

Location:

63:0

Description:

Alias of [mhpmcounter30.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.53.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM30 & scounteren.HPM30) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM30 == 1'b0 && mcounteren.HPM30 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM30 & scounteren.HPM30) == 1'b0 && (mcounteren.HPM30 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(30);
```

D.54. hpmcounter30h

User-mode Hardware Performance Counter 27, high half



hpmcounter30h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter30h](#).

Privilege mode access is controlled with [mcounteren.HPM30](#), [scounteren.HPM30](#), and [hcounteren.HPM30](#) as follows:

mcounteren.HPM30	scounteren.HPM30	hcounteren.HPM30	hpmcounter30h 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.54.1. Attributes

CSR Address	0xc9e
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.54.2. Format

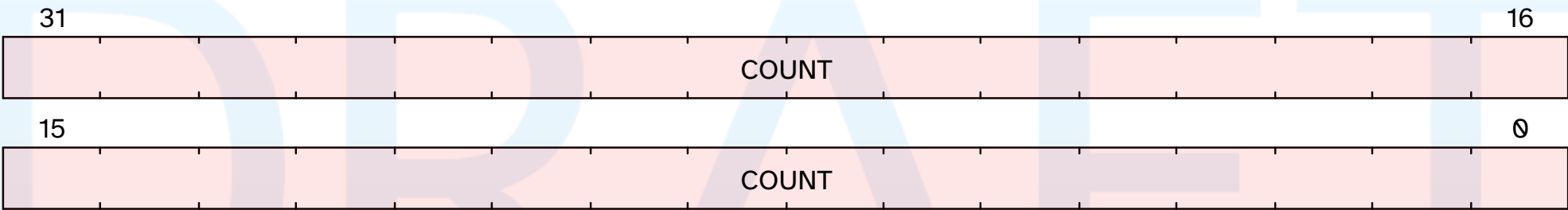


Figure 55. *hpmcounter30h* format

D.54.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter30h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.54.4. Fields

[hpmcounter30h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter30h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.54.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM30 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM30 & scounteren.HPM30) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM30 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM30 == 1'b0 && mcounteren.HPM30 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM30 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM30 & scounteren.HPM30) == 1'b0 && (mcounteren.HPM30 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM30 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(30)[63:32];

```

DRAFT

D.55. 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.55.1. Attributes

CSR Address	0xc1f
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.55.2. Format

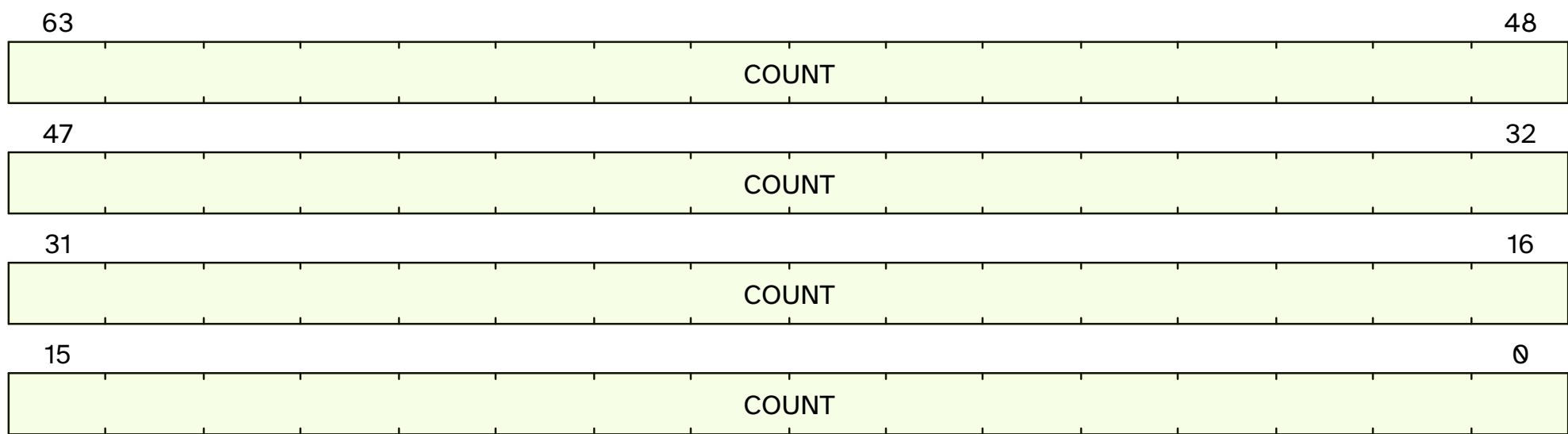


Figure 56. hpmcounter31 format

D.55.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter31.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.55.4. Fields

[hpmcounter31.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter31.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.55.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM31 & scounteren.HPM31) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM31 == 1'b0 && mcounteren.HPM31 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM31 & scounteren.HPM31) == 1'b0 && (mcounteren.HPM31 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(31);
```

D.56. hpmcounter31h

User-mode Hardware Performance Counter 28, high half



hpmcounter31h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter31h](#).

Privilege mode access is controlled with [mcounteren.HPM31](#), [scounteren.HPM31](#), and [hcounteren.HPM31](#) as follows:

mcounteren.HPM31	scounteren.HPM31	hcounteren.HPM31	hpmcounter31h 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.56.1. Attributes

CSR Address	0xc9f
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.56.2. Format

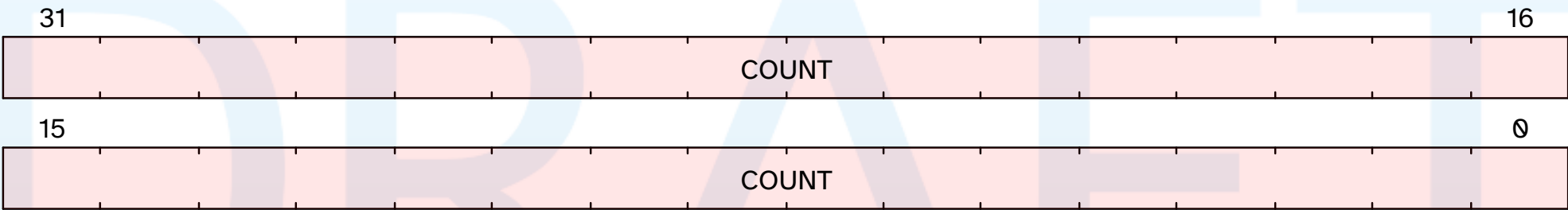


Figure 57. *hpmcounter31h* format

D.56.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter31h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.56.4. Fields

[hpmcounter31h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter31h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.56.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM31 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM31 & scounteren.HPM31) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM31 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM31 == 1'b0 && mcounteren.HPM31 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM31 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM31 & scounteren.HPM31) == 1'b0 && (mcounteren.HPM31 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM31 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(31)[63:32];

```

DRAFT

D.57. hpmcounter3h

User-mode Hardware Performance Counter 0, high half

i

hpmcounter3h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter3h](#).

Privilege mode access is controlled with [mcounteren.HPM3](#), [scounteren.HPM3](#), and [hcounteren.HPM3](#) as follows:

mcounteren.HPM3	scounteren.HPM3	hcounteren.HPM3	hpmcounter3h 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.57.1. Attributes

CSR Address	0xc83
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.57.2. Format

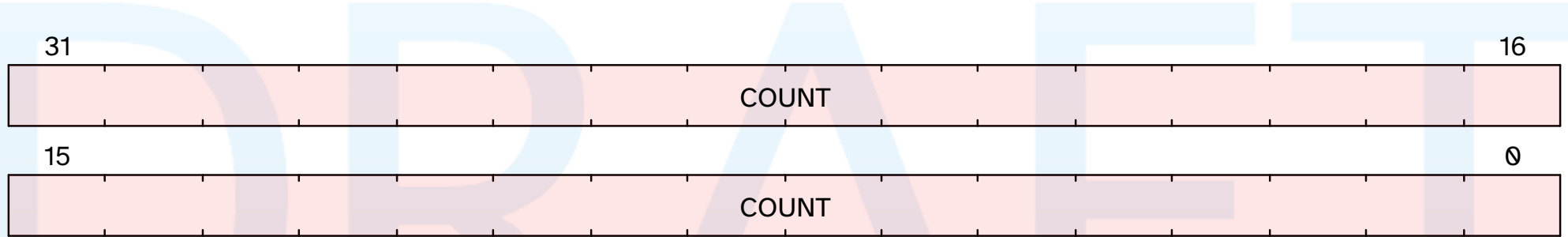


Figure 58. hpmcounter3h format

D.57.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter3h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.57.4. Fields

[hpmcounter3h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter3h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.57.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM3 == 1'b0) {  
        // ...  
    }  
}
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM3 & scounteren.HPM3) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM3 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM3 == 1'b0 && mcounteren.HPM3 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM3 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM3 & scounteren.HPM3) == 1'b0 && (mcounteren.HPM3 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM3 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(3)[63:32];

```

DRAFT

D.58. 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.58.1. Attributes

CSR Address	0xc04
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.58.2. Format

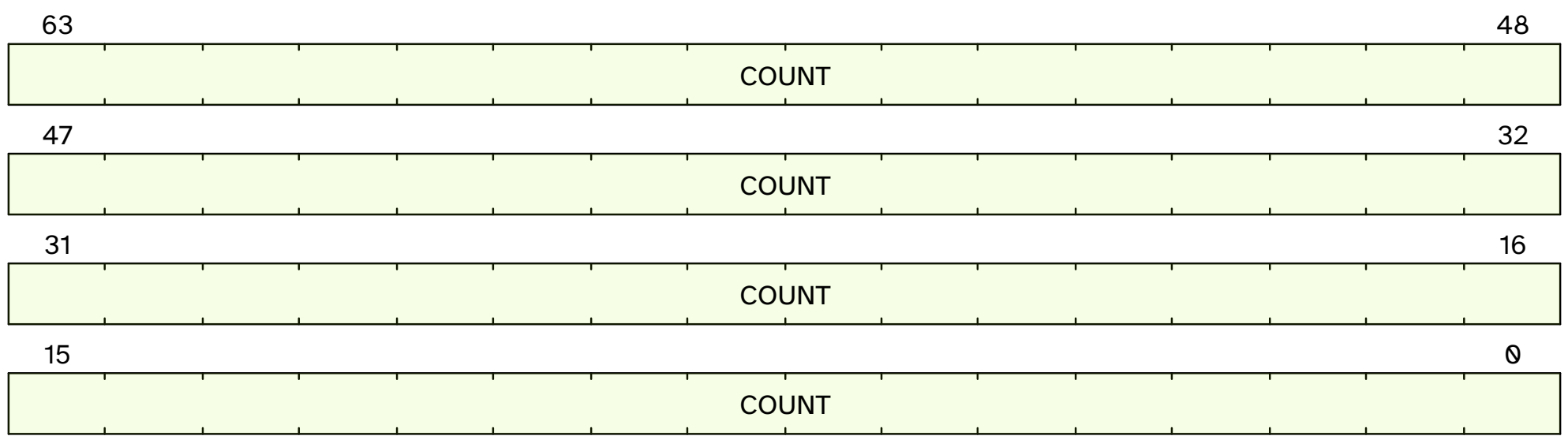


Figure 59. hpmcounter4 format

D.58.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter4.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.58.4. Fields

[hpmcounter4.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter4.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.58.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM4 & scounteren.HPM4) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM4 == 1'b0 && mcounteren.HPM4 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM4 & scounteren.HPM4) == 1'b0 && (mcounteren.HPM4 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(4);
```

D.59. hpmcounter4h

User-mode Hardware Performance Counter 1, high half



hpmcounter4h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter4h](#).

Privilege mode access is controlled with [mcounteren.HPM4](#), [scounteren.HPM4](#), and [hcounteren.HPM4](#) as follows:

mcounteren.HPM4	scounteren.HPM4	hcounteren.HPM4	hpmcounter4h 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.59.1. Attributes

CSR Address	0xc84
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.59.2. Format

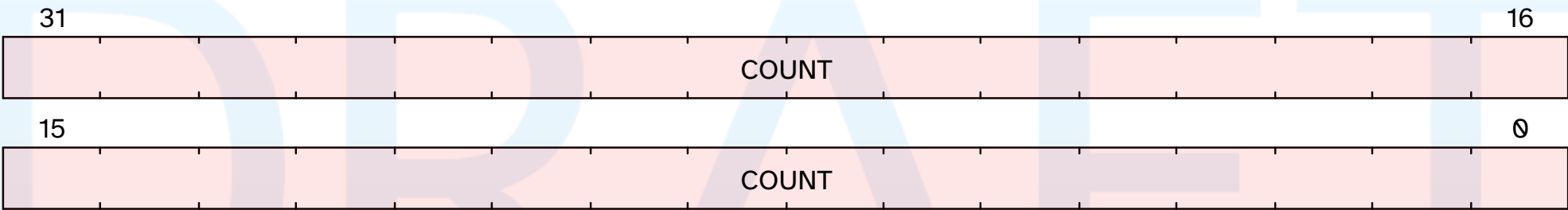


Figure 60. *hpmcounter4h* format

D.59.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter4h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.59.4. Fields

[hpmcounter4h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter4h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.59.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM4 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM4 & scounteren.HPM4) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM4 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM4 == 1'b0 && mcounteren.HPM4 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM4 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM4 & scounteren.HPM4) == 1'b0 && (mcounteren.HPM4 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM4 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(4)[63:32];

```

DRAFT

D.60. 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.60.1. Attributes

CSR Address	0xc05
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.60.2. Format

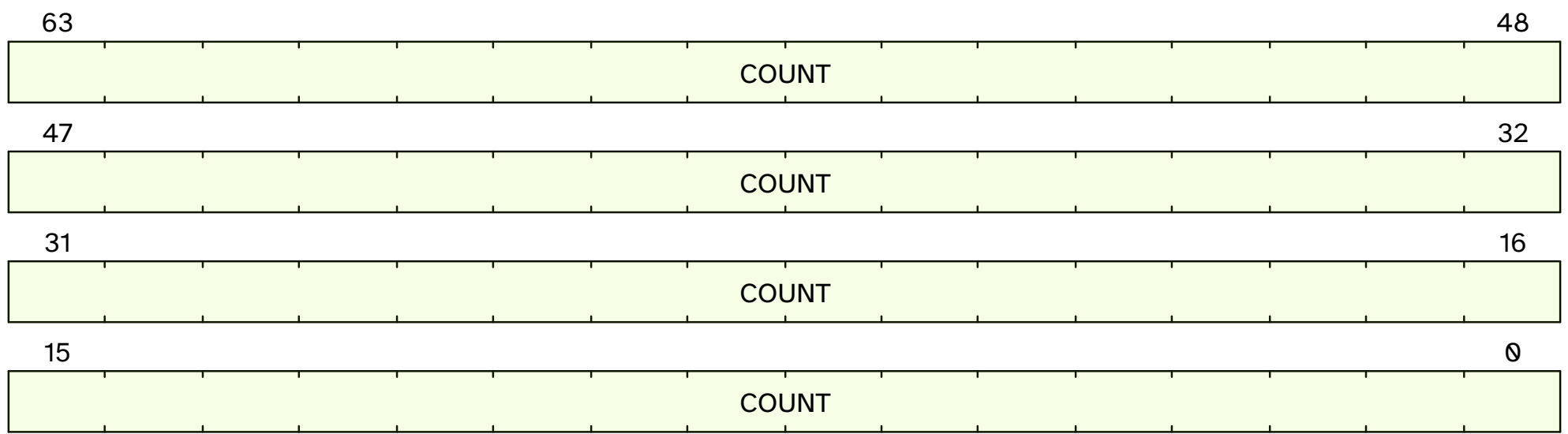


Figure 61. hpmcounter5 format

D.60.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter5.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.60.4. Fields

hpmcounter5.COUNT Field

Location:

63:0

Description:

Alias of *mhpmcounter5.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.60.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM5 & scounteren.HPM5) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM5 == 1'b0 && mcounteren.HPM5 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM5 & scounteren.HPM5) == 1'b0 && (mcounteren.HPM5 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
return read_hpm_counter(5);
```

D.61. hpmcounter5h

User-mode Hardware Performance Counter 2, high half

i *hpmcounter5h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter5h*.

Privilege mode access is controlled with *mcounteren.HPM5*, *scounteren.HPM5*, and *hcounteren.HPM5* as follows:

<i>mcounteren.HPM5</i>	<i>scounteren.HPM5</i>	<i>hcounteren.HPM5</i>	<i>hpmcounter5h</i> 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.61.1. Attributes

CSR Address	0xc85
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.61.2. Format

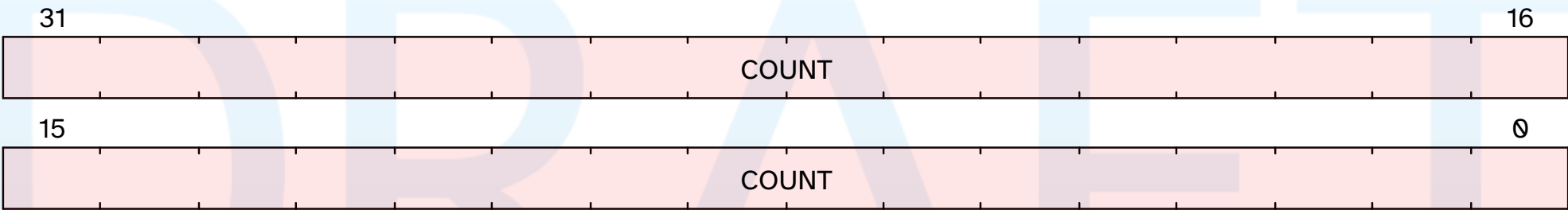


Figure 62. *hpmcounter5h* format

D.61.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter5h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.61.4. Fields

hpmcounter5h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter5h.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.61.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM5 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM5 & scounteren.HPM5) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM5 == 1'b0 && mcounteren.HPM5 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM5 & scounteren.HPM5) == 1'b0 && (mcounteren.HPM5 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM5 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(5)[63:32];

```

DRAFT

D.62. 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.62.1. Attributes

CSR Address	0xc06
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.62.2. Format

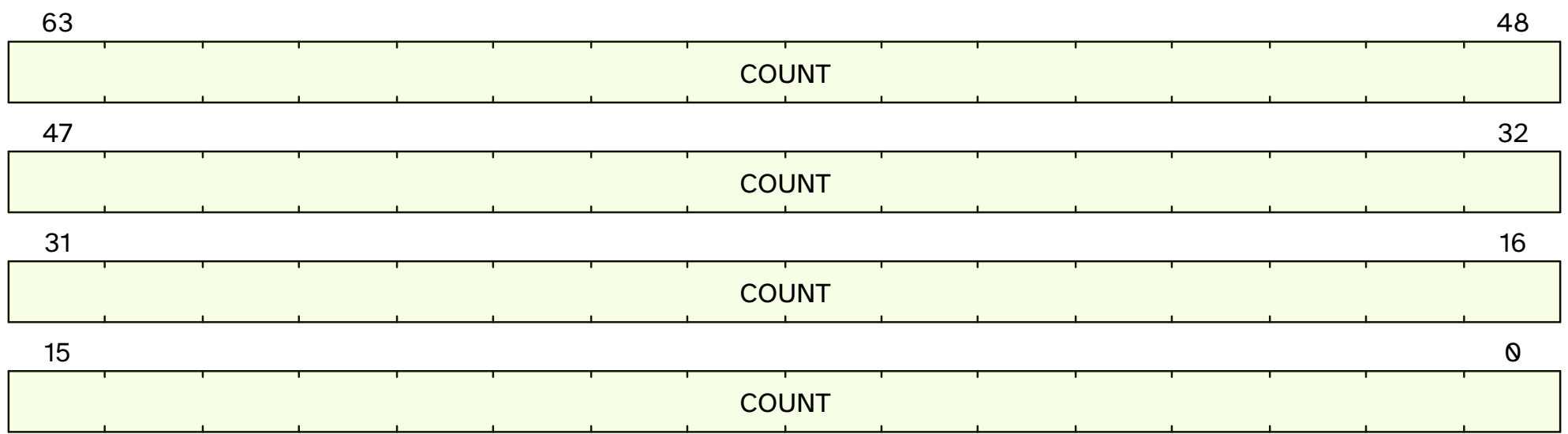


Figure 63. hpmcounter6 format

D.62.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter6.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.62.4. Fields

hpmcounter6.COUNT Field

Location:

63:0

Description:

Alias of *mhpmcounter6.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.62.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM6 & scounteren.HPM6) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM6 == 1'b0 && mcounteren.HPM6 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM6 & scounteren.HPM6) == 1'b0 && (mcounteren.HPM6 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(6);
```

D.63. hpmcounter6h

User-mode Hardware Performance Counter 3, high half

i *hpmcounter6h* is only defined in RV32.

Alias for M-mode CSR [mhpmcounter6h](#).

Privilege mode access is controlled with [mcounteren.HPM6](#), [scounteren.HPM6](#), and [hcounteren.HPM6](#) as follows:

mcounteren.HPM6	scounteren.HPM6	hcounteren.HPM6	hpmcounter6h 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.63.1. Attributes

CSR Address	0xc86
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.63.2. Format

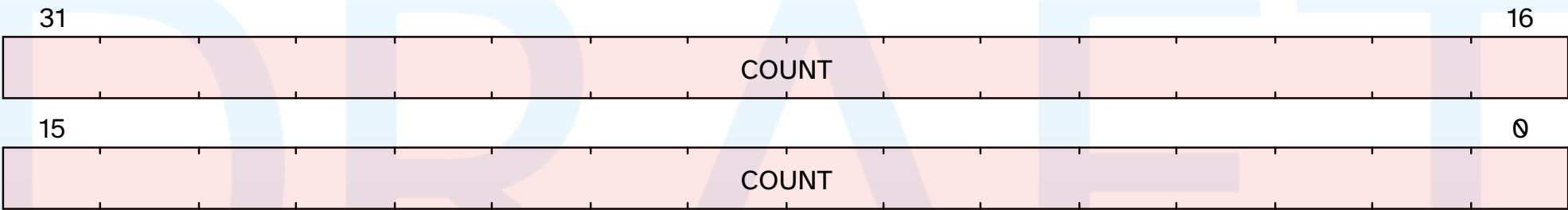


Figure 64. hpmcounter6h format

D.63.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter6h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.63.4. Fields

[hpmcounter6h.COUNT](#) Field

Location:
31:0

Description:
Alias of [mhpmcounter6h.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.63.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcounteren.HPM6 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM6 & scounteren.HPM6) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM6 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM6 == 1'b0 && mcounteren.HPM6 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM6 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM6 & scounteren.HPM6) == 1'b0 && (mcounteren.HPM6 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM6 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(6)[63:32];

```

DRAFT

D.64. 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.64.1. Attributes

CSR Address	0xc07
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.64.2. Format

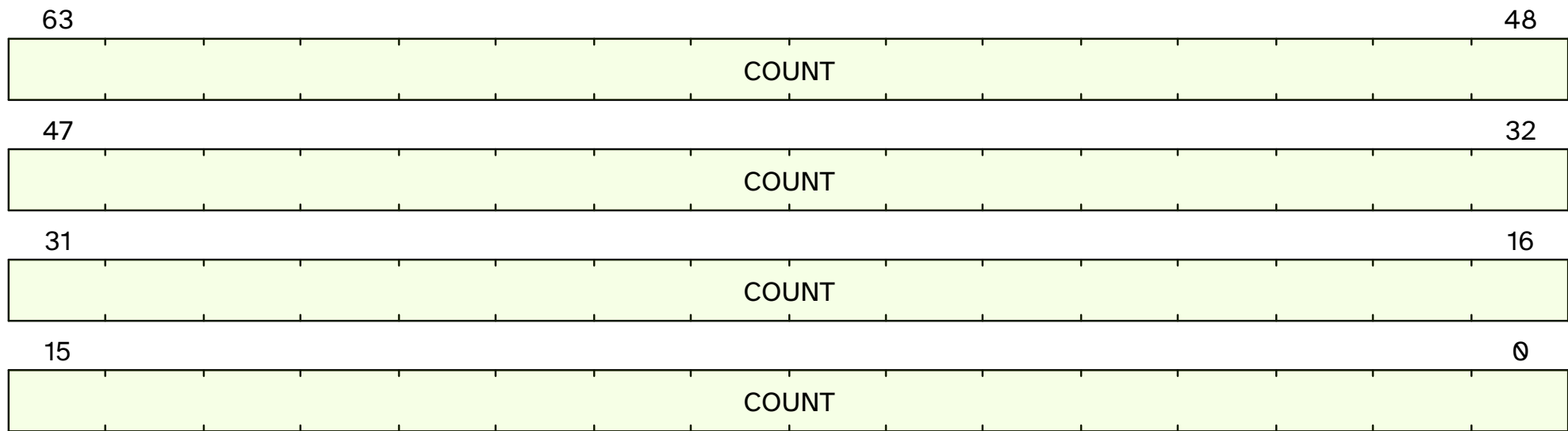


Figure 65. hpmcounter7 format

D.64.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter7.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.64.4. Fields

hpmcounter7.COUNT Field

Location:

63:0

Description:

Alias of *mhpmcounter7.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.64.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM7 & scounteren.HPM7) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM7 == 1'b0 && mcounteren.HPM7 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM7 & scounteren.HPM7) == 1'b0 && (mcounteren.HPM7 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(7);
```

D.65. hpmcounter7h

User-mode Hardware Performance Counter 4, high half



hpmcounter7h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter7h](#).

Privilege mode access is controlled with [mcounteren.HPM7](#), [scounteren.HPM7](#), and [hcounteren.HPM7](#) as follows:

mcounteren.HPM7	scounteren.HPM7	hcounteren.HPM7	hpmcounter7h 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.65.1. Attributes

CSR Address	0xc87
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.65.2. Format

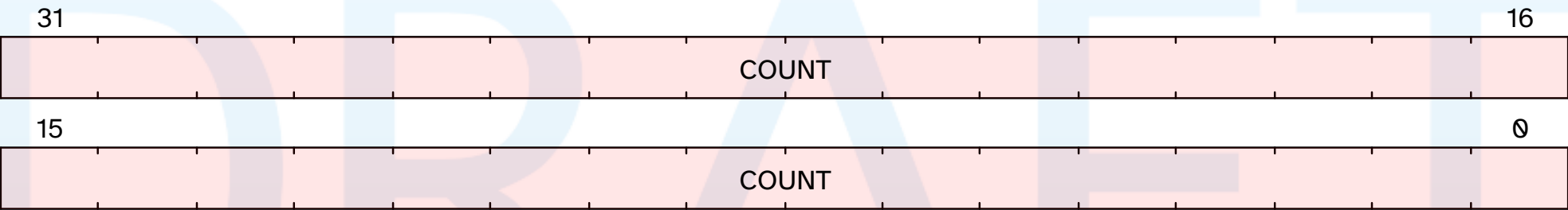


Figure 66. *hpmcounter7h* format

D.65.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter7h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.65.4. Fields

[hpmcounter7h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter7h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.65.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM7 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM7 & scounteren.HPM7) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM7 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM7 == 1'b0 && mcounteren.HPM7 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM7 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM7 & scounteren.HPM7) == 1'b0 && (mcounteren.HPM7 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM7 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(7)[63:32];

```

DRAFT

D.66. 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.66.1. Attributes

CSR Address	0xc08
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.66.2. Format

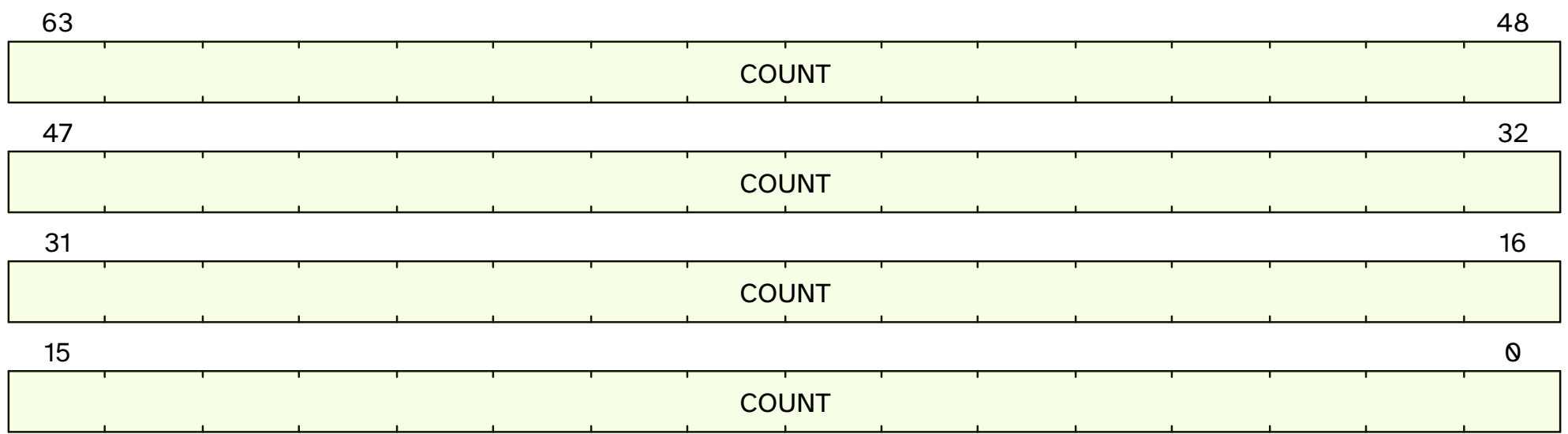


Figure 67. hpmcounter8 format

D.66.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter8.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.66.4. Fields

hpmcounter8.COUNT Field

Location:
63:0

Description:
Alias of *mhpmcounter8.COUNT*.

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.66.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM8 & scounteren.HPM8) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM8 == 1'b0 && mcounteren.HPM8 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM8 & scounteren.HPM8) == 1'b0 && (mcounteren.HPM8 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(8);
```

D.67. hpmcounter8h

User-mode Hardware Performance Counter 5, high half

i

hpmcounter8h is only defined in RV32.

Alias for M-mode CSR [mhpmcounter8h](#).

Privilege mode access is controlled with [mcounteren.HPM8](#), [scounteren.HPM8](#), and [hcounteren.HPM8](#) as follows:

mcounteren.HPM8	scounteren.HPM8	hcounteren.HPM8	hpmcounter8h 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.67.1. Attributes

CSR Address	0xc88
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.67.2. Format

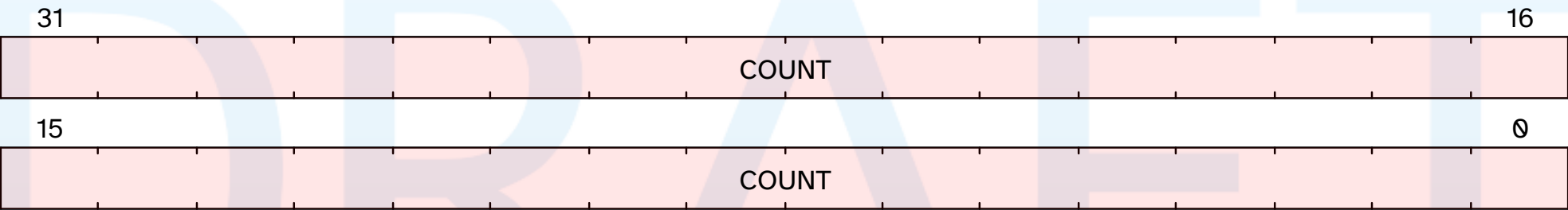


Figure 68. hpmcounter8h format

D.67.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter8h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.67.4. Fields

[hpmcounter8h.COUNT](#) Field

Location:

31:0

Description:

Alias of [mhpmcounter8h.COUNT](#).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.67.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM8 == 1'b0) {
```

```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM8 & scounteren.HPM8) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM8 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM8 == 1'b0 && mcounteren.HPM8 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM8 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM8 & scounteren.HPM8) == 1'b0 && (mcounteren.HPM8 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM8 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(8)[63:32];

```

DRAFT

D.68. 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.68.1. Attributes

CSR Address	0xc09
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

D.68.2. Format

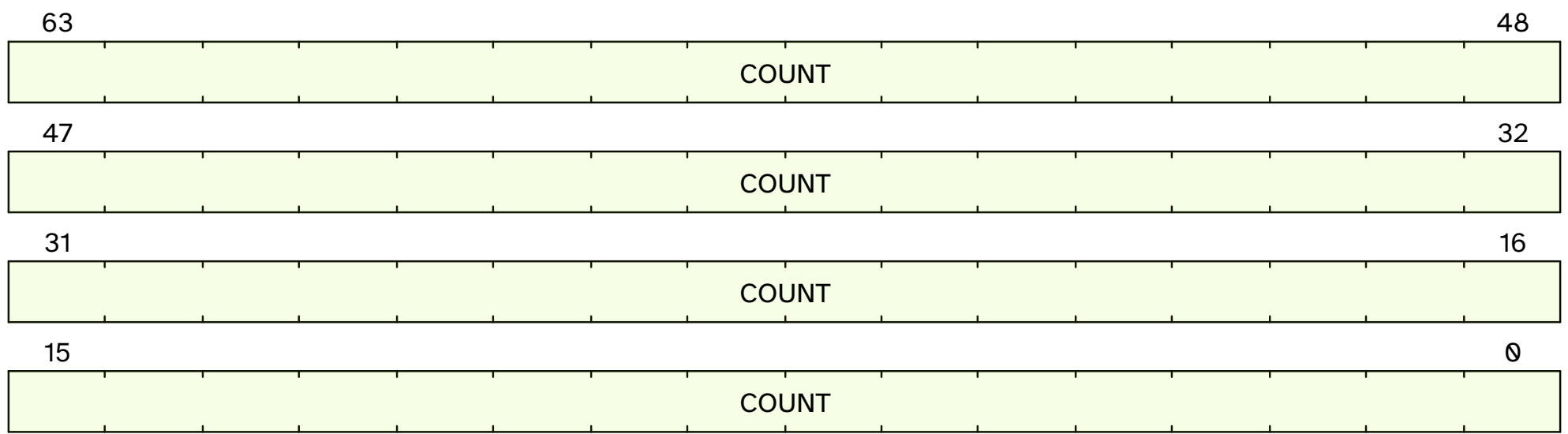


Figure 69. hpmcounter9 format

D.68.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter9.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.68.4. Fields

[hpmcounter9.COUNT](#) Field

Location:
63:0

Description:
Alias of [mhpmcounter9.COUNT](#).

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

D.68.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM9 & scounteren.HPM9) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM9 == 1'b0 && mcounteren.HPM9 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM9 & scounteren.HPM9) == 1'b0 && (mcounteren.HPM9 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
return read_hpm_counter(9);
```

D.69. hpmcounter9h

User-mode Hardware Performance Counter 6, high half

i *hpmcounter9h* is only defined in RV32.

Alias for M-mode CSR *mhpmcounter9h*.

Privilege mode access is controlled with *mcounteren.HPM9*, *scounteren.HPM9*, and *hcounteren.HPM9* as follows:

<i>mcounteren.HPM9</i>	<i>scounteren.HPM9</i>	<i>hcounteren.HPM9</i>	<i>hpmcounter9h</i> 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.69.1. Attributes

CSR Address	0xc89
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	U

D.69.2. Format

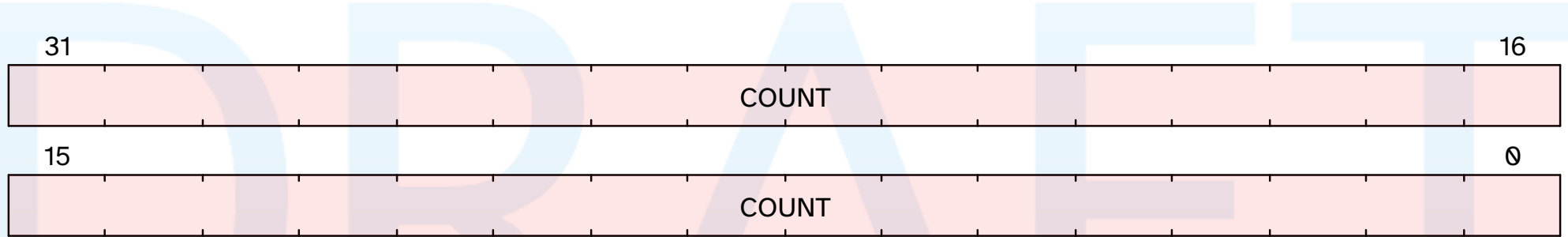


Figure 70. *hpmcounter9h* format

D.69.3. Field Summary

Name	Location	Type	Reset Value
<i>hpmcounter9h.COUNT</i>	31:0	RO-H	UNDEFINED_LEGAL

D.69.4. Fields

hpmcounter9h.COUNT Field

Location:

31:0

Description:

Alias of *mhpmcounter9h.COUNT*.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.69.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
    if (mcounteren.HPM9 == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.HPM9 & scounteren.HPM9) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.HPM9 == 1'b0 && mcounteren.HPM9 == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.HPM9 & scounteren.HPM9) == 1'b0 && (mcounteren.HPM9 == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.HPM9 == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_hpm_counter(9)[63:32];

```

DRAFT

D.70. hstatus

Hypervisor Status

The hstatus register tracks and controls a VS-mode guest.

Unlike fields in [sstatus](#), which are all aliases of fields [mstatus](#), bits in [hstatus](#) are independent bits and do not have aliases.

D.70.1. Attributes

CSR Address	0x600
Defining extension	H, version >= H@1.0.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.70.2. Format

This CSR format changes dynamically with XLEN.

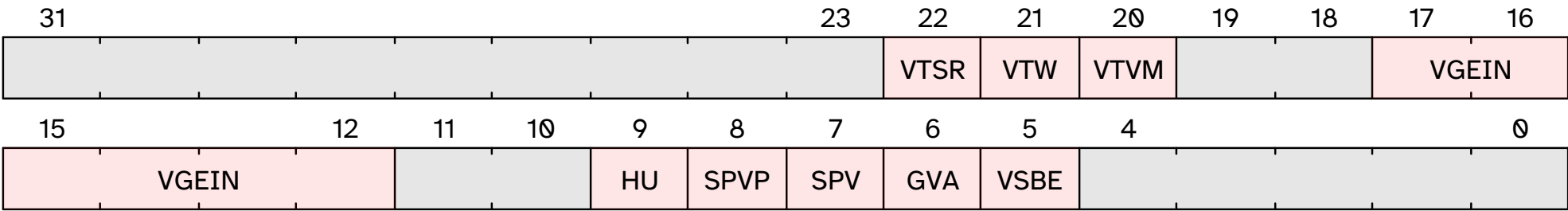


Figure 71. hstatus Format when CSR[mstatus].SXL == 0

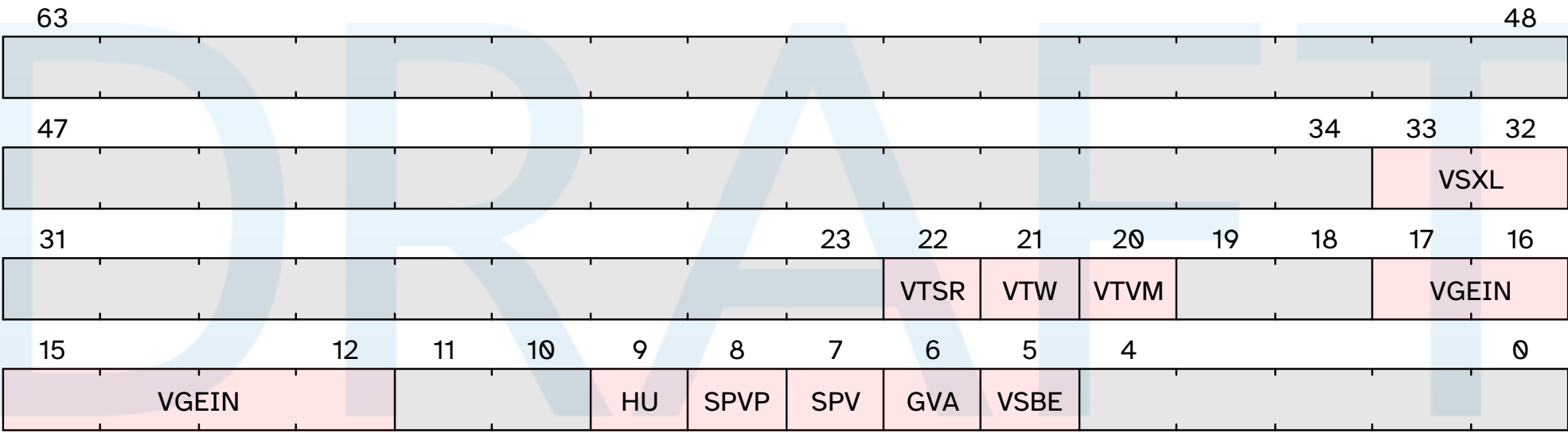


Figure 72. hstatus Format when CSR[mstatus].SXL == 1

D.70.3. Field Summary

Name	Location	Type	Reset Value
hstatus.VSXL	33:32	RO RW	UNDEFINED_LEGAL
hstatus.VTSR	22	RW	UNDEFINED_LEGAL
hstatus.VTW	21	RW	UNDEFINED_LEGAL
hstatus.VTVM	20	RW	UNDEFINED_LEGAL
hstatus.VGEIN	17:12	RW RW-R	UNDEFINED_LEGAL
hstatus.HU	9	RW	UNDEFINED_LEGAL
hstatus.SPVP	8	RW	UNDEFINED_LEGAL
hstatus.SPV	7	RW	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
hstatus.GVA	6	RW	UNDEFINED_LEGAL
hstatus.VSBE	5	RW RO	UNDEFINED_LEGAL

D.70.4. Fields

[hstatus.VSXL](#) Field



VSXL is only defined in RV64 (CSR[mstatus].SXL == 1)

Location:

33:32

Description:

When return VSXLEN == 3264;

Determines the effective XLEN in VS-mode. Valid values are:

*[separator="!"]
!===
! Value ! VSXLEN*

*! 0 ! 32
! 1 ! 64
!===*

When return VSXLEN == 32;

Because the implementation only supports a single VSXLEN == 32, this field is read-only-0.

When return VSXLEN == 64;

Because the implementation only supports a single VSXLEN == 64, this field is read-only-1.

Type:

RO

RW

Reset value:

UNDEFINED_LEGAL

[hstatus.VTSR](#) Field

Location:

22

Description:

When [hstatus.VTSR](#) is set, executing the [sret](#) instruction in VS-mode raises a Virtual Instruction exception.

When [hstatus.VTSR](#) is clear, an [sret](#) instruction in VS-mode returns control to the mode stored in [vsstatus.SPP](#).

Type:

RW

Reset value:

UNDEFINED_LEGAL

hstatus.VTW Field

Location:

21

Description:

When *hstatus.VTW* is set, a *wfi* instruction executed in VS-mode raises a Virtual Instruction exception after waiting an implementation-defined amount of time (which can be 0).

When both *hstatus.VTW* and *mstatus.TW* are clear, a *wfi* instruction executes in VS-mode without a timeout period.

The *wfi* instruction is also affected by *mstatus.TW*, as shown below:

[separator="!",%autowidth,%footer]
!===
.2+! *mstatus.TW* .2+! *hstatus.VTW* 4+^.>! *wfi* behavior
h! HS-mode h! U-mode h! VS-mode h! VU-mode

! 0 ! 0 ! Wait ! Trap (I) ! Wait ! Trap (V)
! 0 ! 1 ! Wait ! Trap (I) ! Trap (V) ! Trap (V)
! 1 ! - ! Trap (I) ! Trap (I) ! Trap (I) ! Trap (I)

6+! Trap (I) - Trap with Illegal Instruction code +
Trap (V) - Trap with Virtual Instruction code
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

hstatus.VTVM Field

Location:

20

Description:

When set, a 'Virtual Instruction' trap occurs when executing an *sfence.vma*, *sinval.vma*, or an explicit CSR access of the *satp* (really *vsatp*) register when in VS-mode.

When clear, the instructions execute as normal in VS-mode.

Notably, *hstatus.VTVM* does **not** cause *hfence.vma*, *sfence.w.inval*, or *sfence.inval.ir* to trap.

mstatus.TVM does not affect the VS-mode instructions controlled by *hstatus*.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hstatus.VGEIN Field

Location:

17:12

Description:

Selects the guest external interrupt source for VS-level external interrupts.

When *hstatus.VGEIN* == 0, no external interrupt source is selected.

When *hstatus.VGEIN* != 0, it selects which bit of *hgeip* is currently active in VS-mode.

Type:

RW

RW-R

Reset value:

UNDEFINED_LEGAL

hstatus.HU Field

Location:

9

Description:

When set, the hypervisor load/store instructions (hlv, hlvx, and hsv) can be executed in U-mode.

When clear, the hypervisor load/store instructions cause an Illegal Instruction trap.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hstatus.SPVP Field

Location:

8

Description:

Written by hardware:

- When taking a trap into HS-mode from VS-mode or VU-mode, [hstatus.SPVP](#) is written with the nominal privilege mode

Notably, unlike its analog [mstatus.SPP](#), [hstatus.SPVP](#) is **not** cleared when returning from a trap.

Can also be written by software without immediate side-effect.

Affects execution by:

- Controls the effective privilege level applied to the hypervisor load/store instructions, hlv, hlvx, and hsv.

Type:

RW

Reset value:

UNDEFINED_LEGAL

hstatus.SPV Field

Location:

7

Description:

Written by hardware:

- On a trap into HS-mode, hardware writes 1 when the prior mode was VS-mode or VU-mode, and 0 otherwise.

Can also be written by software without immediate side-effect.

Affects execution by:

- When an [sret](#) instruction is executed in HS-mode or M-mode, control returns to VS-mode or VU-mode (as selected by [mstatus.SPP](#)) when [hstatus.SPV](#) is 1 and to HS-mode or U-mode otherwise.

Type:

RW

Reset value:
UNDEFINED_LEGAL

hstatus.GVA Field

Location:
6

Description:
Written by hardware whenever a trap is taken into HS-mode:

- *Writes 1 when a trap causes a guest virtual address to be written into [stval](#) (Breakpoint, * Address Misaligned, * Access Fault, * Page Fault, or * Guest-Page Fault).*
- *Writes 0 otherwise*

Does not affect execution.

Type:
RW

Reset value:
UNDEFINED_LEGAL

hstatus.VSBE Field

Location:
5

Description:
*Controls the endianness of data VS-mode (0 = little, 1 = big).
Instructions are always little endian, regardless of the data setting.*

When return VS_MODE_ENDIANNES == "little";

Since the CPU does not support big endian in VS-mode, this is hardwired to 0.

When return VS_MODE_ENDIANNES == "big";

Since the CPU does not support little endian in VS-mode, this is hardwired to 1.

Type:
RW
RO

Reset value:
UNDEFINED_LEGAL

D.70.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrww](#)), the following determines the written value:

```
VSXL = if (CSR[mstatus].SXL < csr_value.VSXL) {
    # cannot set VSXLEN less than SXLEN
    return UNDEFINED_LEGAL_DETERMINISTIC;
} else if ((csr_value.VSXL & 0b10) != 0) {
    # MSB of VSXL represents XLEN > 64, which isn't legal
    return UNDEFINED_LEGAL_DETERMINISTIC;
} else {
    return csr_value.VSXL;
}

VTSR = csr_value.VTSR
VTW = csr_value.VTW
```

```
VTVM = csr_value.VTVM
VGEIN = if (csr_value.VGEIN <= NUM_EXTERNAL_GUEST_INTERRUPTS) {
    return csr_value.VGEIN;
} else {
    return ILLEGAL_WLRL;
}

HU = csr_value.HU
SPVP = csr_value.SPVP
SPV = csr_value.SPV
GVA = csr_value.GVA
VSBE = csr_value.VSBE
```

DRAFT

D.71. htimedelta

Hypervisor time delta

The [htimedelta](#) CSR is a 64-bit read/write register that contains the delta between the value of the [time](#) CSR and the value returned in VS-mode or VU-mode. That is, reading the [time](#) CSR in VS or VU mode returns the sum of the contents of [htimedelta](#) and the actual value of [time](#).



Because overflow is ignored when summing [htimedelta](#) and [time](#), large values of [htimedelta](#) may be used to represent negative time offsets.

D.71.1. Attributes

CSR Address	0x605
Defining extension	H, version >= H@1.0.0
Length	64-bit
Privilege Mode	S

D.71.2. Format

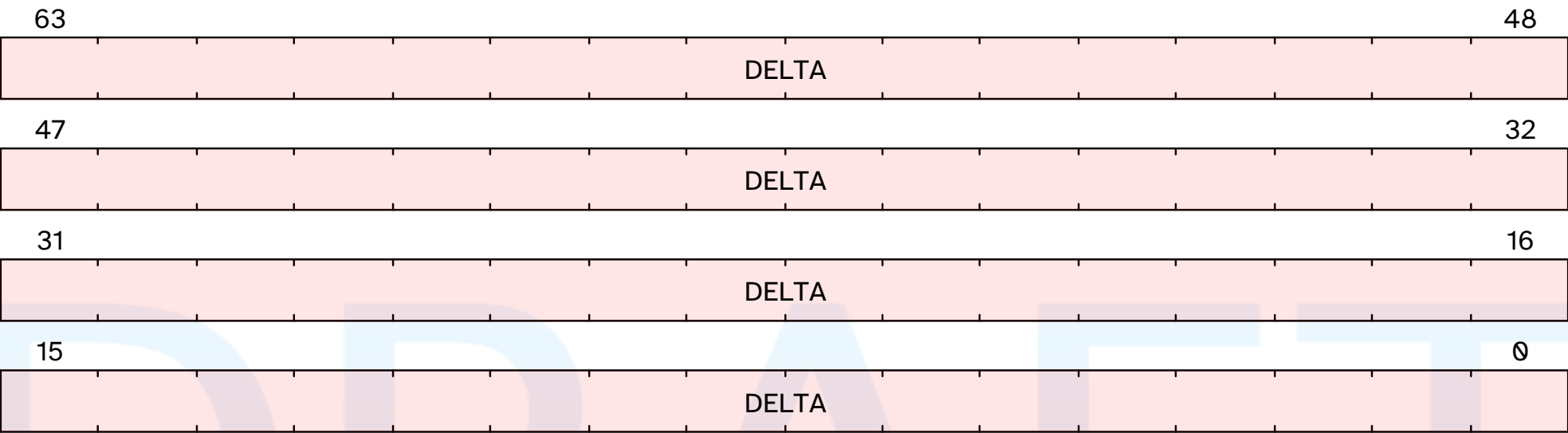


Figure 73. `htimedelta` format

D.71.3. Field Summary

Name	Location	Type	Reset Value
htimedelta.DELTA	63:0	RW	UNDEFINED_LEGAL

D.71.4. Fields

[htimedelta.DELTA](#) Field

Location:
63:0

Description:
Signed delta

Type:
RW

Reset value:
UNDEFINED_LEGAL

D.72. htimedeltah

Hypervisor time delta, upper half

Upper half of the [htimedelta](#) CSR.

D.72.1. Attributes

CSR Address	0x615
Defining extension	H, version >= H@1.0.0
Length	32-bit
Privilege Mode	S

D.72.2. Format

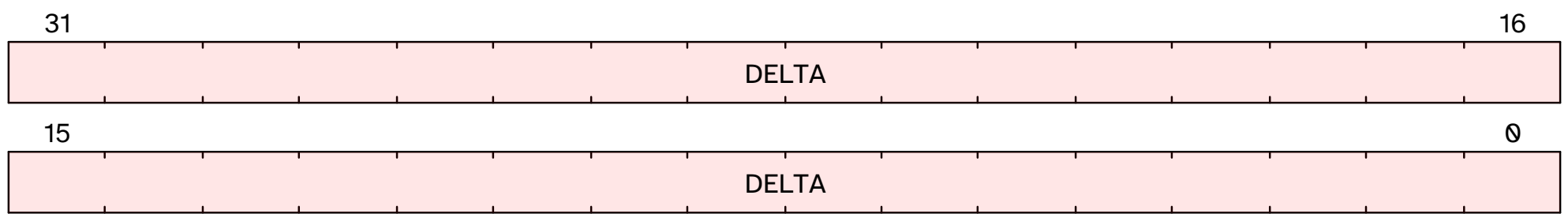


Figure 74. htimedeltah format

D.72.3. Field Summary

Name	Location	Type	Reset Value
htimedeltah.DELTA	31:0	RW	UNDEFINED_LEGAL

D.72.4. Fields

[htimedeltah.DELTA](#) Field

Location:

31:0

Description:

Signed delta

Type:

RW

Reset value:

UNDEFINED_LEGAL

D.72.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
DELTA = CSR[htimedelta].DELTA = {csr_value.DELTA, CSR[htimedelta].DELTA[31:0]};
return csr_value.DELTA;
```

D.72.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return htimedelta.DELTA[31:0];
```

D.73. htinst

Hypervisor Trap Instruction Register

When a trap is taken into HS-mode, mtinst is written with a value that, if nonzero, provides information about the instruction that trapped, to assist software in handling the trap. The values that may be written to mtinst on a trap are documented in TODO.

htinst is a WARL register that need only be able to hold the values that the implementation may automatically write to it on a trap.

D.73.1. Attributes

CSR Address	0x64a
Defining extension	H, version >= H@1.0.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.73.2. Format

This CSR format changes dynamically with XLEN.

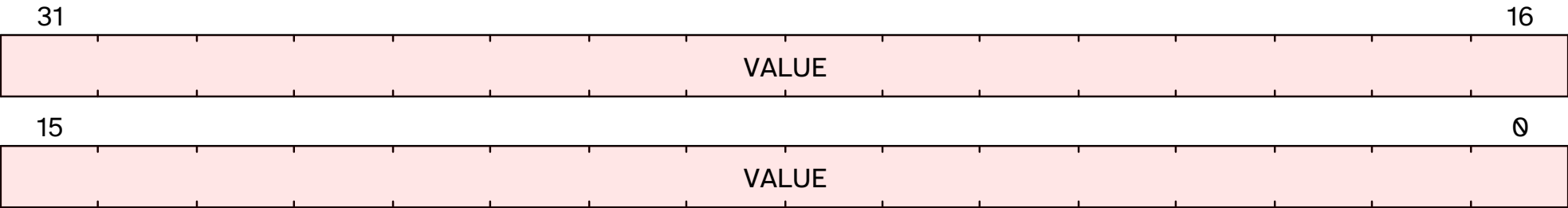


Figure 75. htinst Format when CSR[mstatus].SXL == 0

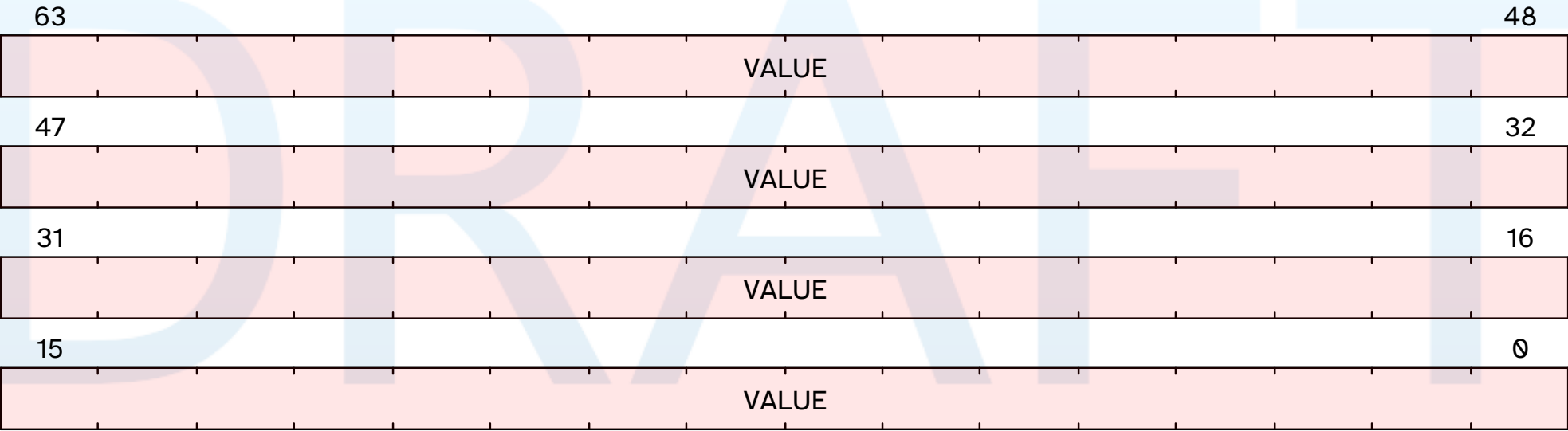


Figure 76. htinst Format when CSR[mstatus].SXL == 1

D.73.3. Field Summary

Nam e	Location	Type	Reset Value
htinst .VALUE	31:0	RW-H	UNDEFINED_LEGAL
	63:0	RO	

D.73.4. Fields

htinst.VALUE Field

Location:

- 31:0 when CSR[mstatus].SXL == 0
- 63:0 when CSR[mstatus].SXL == 1

Description:

Exception-specific information for a trap into HS-mode.

Type:

RW-H

RO

Reset value:

UNDEFINED_LEGAL

DRAFT

D.74. htval

Hypervisor Trap Value Register

When a trap is taken into HS-mode, htval is written with additional exception-specific information, alongside stval, to assist software in handling the trap.

When a guest-page-fault trap is taken into HS-mode, htval is written with either zero or the guest physical address that faulted, shifted right by 2 bits. For other traps, htval is set to zero, but a future standard or extension may redefine htval’s setting for other traps.

A guest-page fault may arise due to an implicit memory access during first-stage (VS-stage) address translation, in which case a guest physical address written to htval is that of the implicit memory access that faulted—for example, the address of a VS-level page table entry that could not be read. (The guest physical address corresponding to the original virtual address is unknown when VS-stage translation fails to complete.) Additional information is provided in CSR htinst to disambiguate such situations.

Otherwise, for misaligned loads and stores that cause guest-page faults, a nonzero guest physical address in htval corresponds to the faulting portion of the access as indicated by the virtual address in stval. For instruction guest-page faults on systems with variable-length instructions, a nonzero htval corresponds to the faulting portion of the instruction as indicated by the virtual address in stval.

htval is a WARL register that must be able to hold zero and may be capable of holding only an arbitrary subset of other 2-bit-shifted guest physical addresses, if any.

D.74.1. Attributes

CSR Address	0x643
Defining extension	H, version >= H@1.0.0
Length	64-bit
Privilege Mode	M

D.74.2. Format

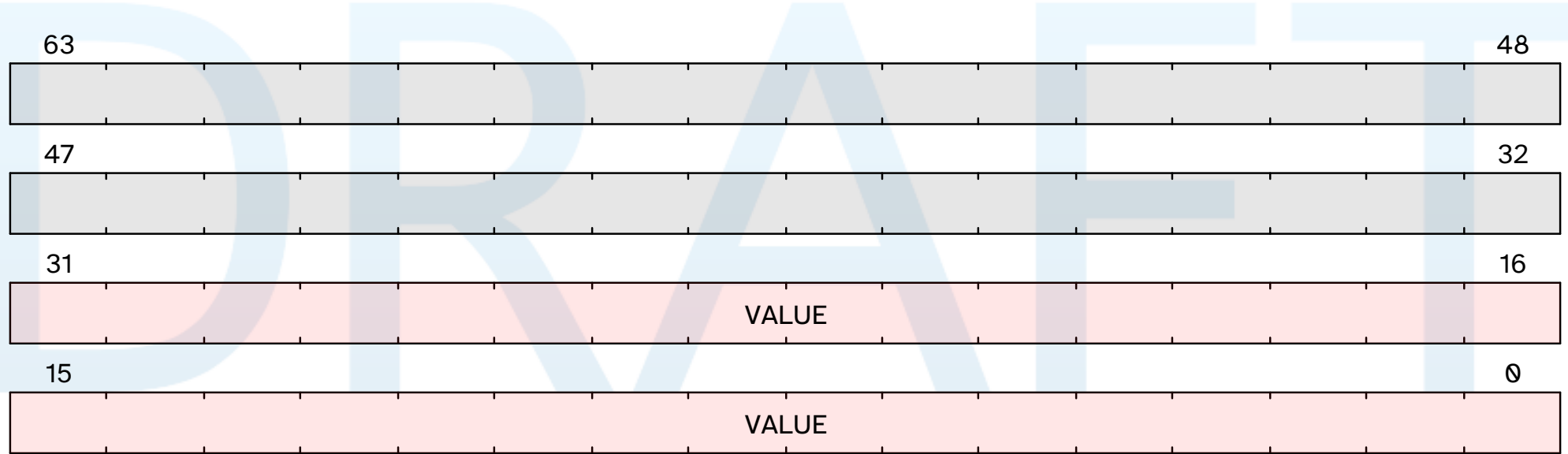


Figure 77. htval format

D.74.3. Field Summary

Nam e	Location	Type	Reset Value
htval. VALU E	63:0	RW-H RO	UNDEFINED_LEGAL

D.74.4. Fields

htval.VALUE Field

Location:
63:0

Description:
Exception-specific information for a trap into M-mode.

Type:
RW-H
RO

Reset value:

UNDEFINED_LEGAL

DRAFT

D.75. 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.75.1. Attributes

CSR Address	0xc02
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

D.75.2. Format

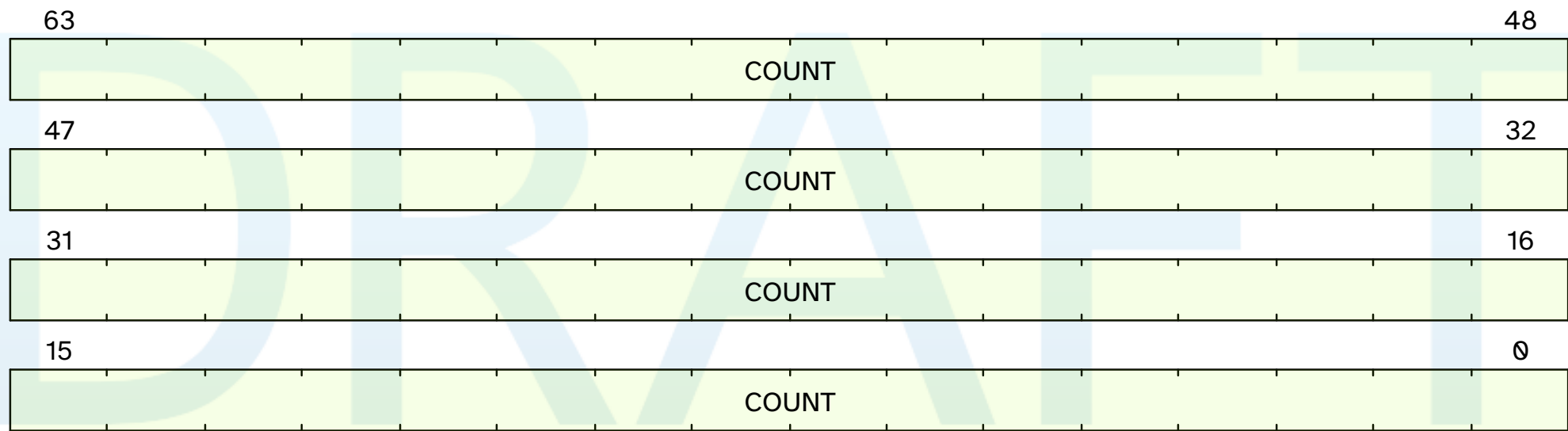


Figure 78. instret format

D.75.3. Field Summary

Name	Location	Type	Reset Value
instret .COUNT	63:0	RO-H	0

D.75.4. Fields

[instret](#).COUNT Field

Location:
63:0

Description:
Alias of [minstret](#).COUNT.

Type:
RO-H

Reset value:
0

D.75.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.IR & scounteren.IR) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.IR == 1'b0 && mcounteren.IR == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.IR & scounteren.IR == 1'b0) && (mcounteren.IR == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return minstret.COUNT;
```

DRAFT

D.76. instreth

Instructions retired counter, high bits

instreth is only defined in RV32.

Alias for high bits of M-mode CSR `minstret`[63:32].

Privilege mode access is controlled with `mcouteren.IR`, `scouteren.IR`, and `hcouteren.IR` as follows:

mcouteren.IR	scouteren.IR	hcouteren.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.76.1. Attributes

CSR Address	0xc82
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

D.76.2. Format

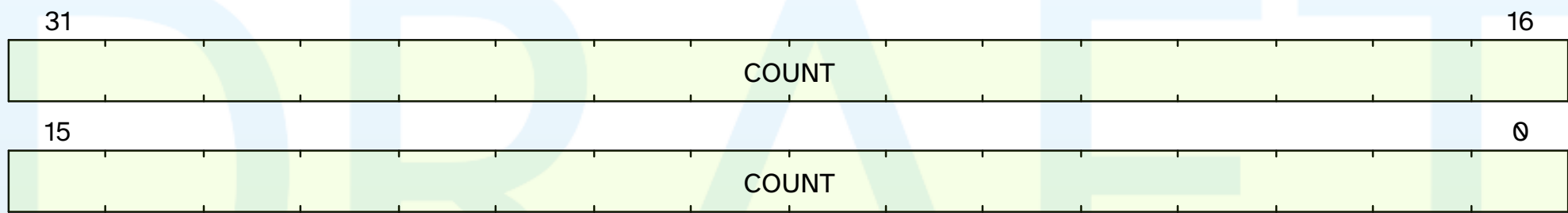


Figure 79. instreth format

D.76.3. Field Summary

Name	Location	Type	Reset Value
instret h.COUNT NT	31:0	RO-H	UNDEFINED_LEGAL

D.76.4. Fields

instreth.COUNT Field

Location:

31:0

Description:

Alias of `minstret.COUNT`[63:32].

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.76.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {  
    if (mcouteren.IR == 1'b0) {
```



```

    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.IR & scounteren.IR) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.IR == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.IR == 1'b0 && mcounteren.IR == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.IR == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.IR & scounteren.IR) == 1'b0) && (mcounteren.IR == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.IR == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
}
return read_mcycle();

```

DRAFT

D.77. 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 `mhpmcountern`, 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.77.1. Attributes

CSR Address	0x306
Defining extension	U, version $\geq$ U@1.0.0
Length	32-bit
Privilege Mode	M

D.77.2. Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
HPM31	HPM30	HPM29	HPM28	HPM27	HPM26	HPM25	HPM24	HPM23	HPM22	HPM21	HPM20	HPM19	HPM18	HPM17	HPM16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HPM15	HPM14	HPM13	HPM12	HPM11	HPM10	HPM9	HPM8	HPM7	HPM6	HPM5	HPM4	HPM3	IR	TM	CY

Figure 80. mcounteren format

D.77.3. Field Summary

Name	Location	Type	Reset Value
<code>mcount eren.CY</code>	0	RW RO	UNDEFINED_LEGAL
<code>mcount eren.TM</code>	1	RO	0
<code>mcount eren.IR</code>	2	RW RO	UNDEFINED_LEGAL
<code>mcount eren.HP M3</code>	3	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
mcount eren.HP M4	4	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M5	5	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M6	6	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M7	7	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M8	8	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M9	9	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M10	10	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M11	11	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M12	12	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M13	13	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M14	14	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M15	15	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M16	16	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M17	17	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M18	18	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M19	19	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M20	20	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M21	21	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M22	22	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M23	23	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M24	24	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M25	25	RW	UNDEFINED_LEGAL
		RO	

Name	Location	Type	Reset Value
mcount eren.HP M26	26	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M27	27	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M28	28	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M29	29	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M30	30	RW	UNDEFINED_LEGAL
		RO	
mcount eren.HP M31	31	RW	UNDEFINED_LEGAL
		RO	

D.77.4. 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

DRAFT

D.78. mcycle

Machine Cycle Counter

Counts the number of clock cycles executed by the processor core on which the hart is running. The counter has 64-bit precision on all RV32 and RV64 harts.

The [mcycle](#) CSR may be shared between harts on the same core, in which case writes to [mcycle](#) will be visible to those harts. The platform should provide a mechanism to indicate which harts share an [mcycle](#) CSR.

D.78.1. Attributes

CSR Address	0xb00
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	M

D.78.2. Format

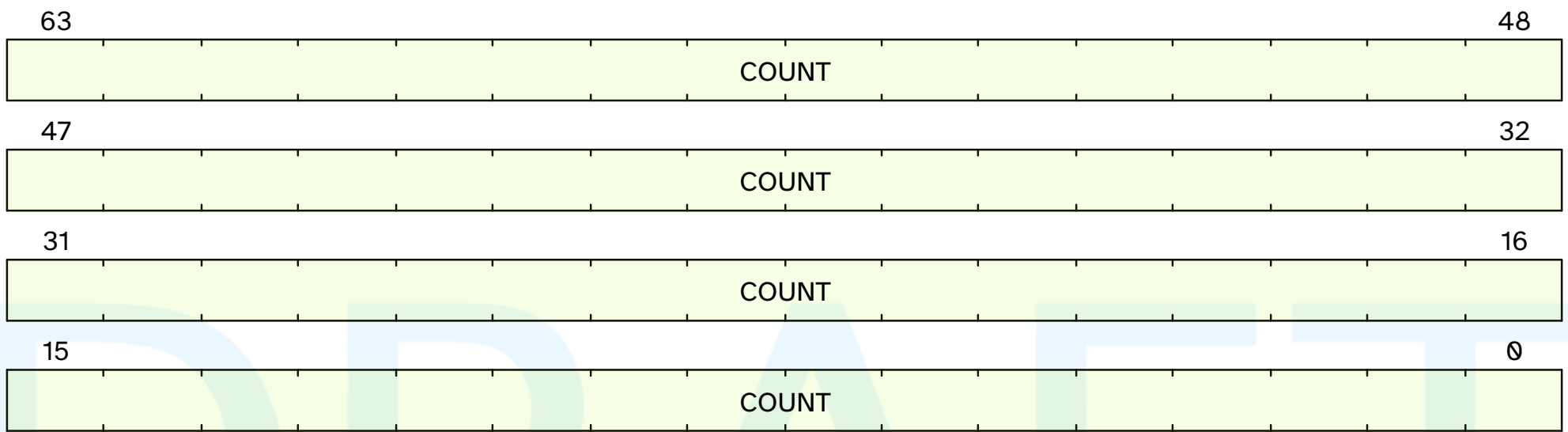


Figure 81. mcycle format

D.78.3. Field Summary

Name	Location	Type	Reset Value
mcycle.COUNT	63:0	RW-RH	UNDEFINED_LEGAL

D.78.4. Fields

[mcycle.COUNT](#) Field

Location:

63:0

Description:

Cycle counter.

<%- if ext?(:Zicntr) -%>
Aliased as [cycle](#).
<%- end -%>

Increments every cycle unless:

- [mcountinhibit.CY](#) <%- if ext?(:Smcdeleg) -%>or its alias `scountinhibit.CY`<%- end -%> is set
<%- if ext?(:Smcntrpmf) -%>
- `mcyclecfg.MINH` is set and the current privilege level is M
<%- if ext?(:S) -%>
- `mcyclecfg.SINH` <%- if ext?(:Sccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is (H)S
<%- end -%>
<%- if ext?(:U) -%>
- `mcyclecfg.UINH` <%- if ext?(:Sccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is U
<%- end -%>

<%- if ext?(:H) -%>

- mcyclecfg.VSINH <%- if ext?(:Sccfg) -%>or its alias instretcfg.SINH<%- end -%> is set and the current privilege level is VS
 - mcyclecfg.VUINH <%- if ext?(:Sccfg) -%>or its alias instretcfg.SINH<%- end -%> is set and the current privilege level is VU
- <%- end -%>
- <%- end -%>

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

D.78.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
    return sw_write_mcycle({read_mcycle()[63:31], csr_value.COUNT[31:0]});
} else {
    return sw_write_mcycle(csr_value.COUNT);
}
```

D.78.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return read_mcycle();
```

D.79. mcycleh

High-half machine Cycle Counter

i

mcycleh

 is only defined in RV32.

High-half alias of [mcycle](#).

D.79.1. Attributes

CSR Address	0xb80
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	M

D.79.2. Format

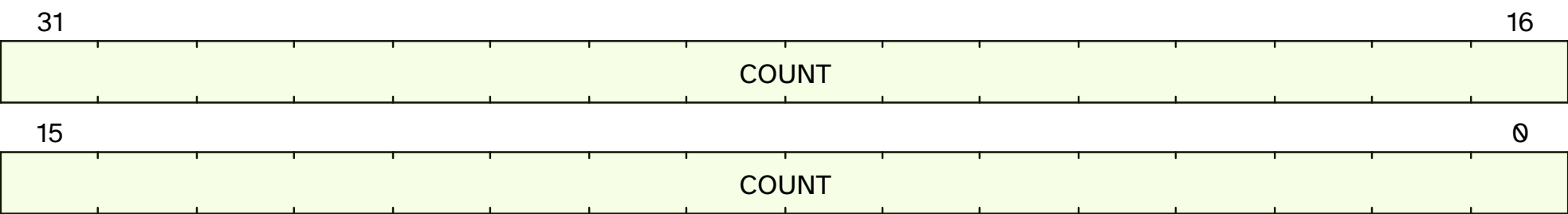


Figure 82. mcycleh format

D.79.3. Field Summary

Name	Location	Type	Reset Value
mcycleh.COUNT	31:0	RW-RH	UNDEFINED_LEGAL

D.79.4. Fields

[mcycleh.COUNT](#) Field

Location:

31:0

Description:

Alias of upple half of [mcycle.COUNT](#).

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

D.79.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
    return sw_write_mcycle({csr_value.COUNT[31:0], read_mcycle()[31:0]});
} else {
    return sw_write_mcycle(csr_value.COUNT);
}
```

D.79.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return read_mcycle()[63:32];
```

DRAFT

D.80. 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.80.1. Attributes

CSR Address	0x302
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	M

D.80.2. Format

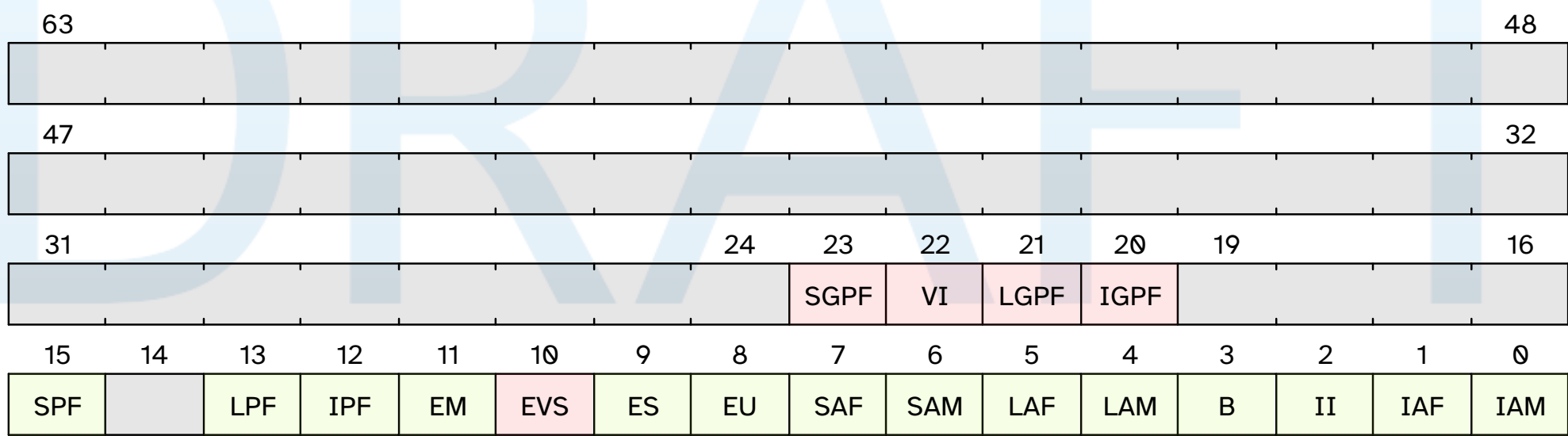


Figure 83. medeleg format

D.80.3. Field Summary

Name	Location	Type	Reset Value
medel eg.IA M	0	RW	UNDEFINED_LEGAL
medel eg.IAF	1	RW	UNDEFINED_LEGAL
medel eg.II	2	RW	UNDEFINED_LEGAL
medel eg.B	3	RW	UNDEFINED_LEGAL
medel eg.LA M	4	RW	UNDEFINED_LEGAL
medel eg.LA F	5	RW	UNDEFINED_LEGAL
medel eg.SA M	6	RW	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
medeleg.SAF	7	RW	UNDEFINED_LEGAL
medeleg.EU	8	RW	UNDEFINED_LEGAL
medeleg.ES	9	RW	UNDEFINED_LEGAL
medeleg.EVS	10	RW	UNDEFINED_LEGAL
medeleg.EM	11	RO	0
medeleg.IPF	12	RW	UNDEFINED_LEGAL
medeleg.LPF	13	RW	UNDEFINED_LEGAL
medeleg.SPF	15	RW	UNDEFINED_LEGAL
medeleg.IGPF	20	RW	UNDEFINED_LEGAL
medeleg.LGPF	21	RW	UNDEFINED_LEGAL
medeleg.VI	22	RW	UNDEFINED_LEGAL
medeleg.SGPF	23	RW	UNDEFINED_LEGAL

D.80.4. 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

DRAFT

D.81. menvcfg

Machine Environment Configuration

Contains fields that control certain characteristics of the execution environment for modes less privileged than M-mode.

The `menvcfg` CSR controls certain characteristics of the execution environment for modes less privileged than M.

If bit FIOM (Fence of I/O implies Memory) is set to one in `menvcfg`, FENCE instructions executed in modes less privileged than M are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. Table 20 details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO for modes less privileged than M when FIOM=1.

Similarly, for modes less privileged than M when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

If S-mode is not supported, or if `satp.MODE` is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 20. Modified interpretation of FENCE predecessor and successor sets for modes less privileged than M when FIOM=1.

Instruction bit Meaning when set	PI PO
Predecessor device input and memory reads (PR implied)	SI
Predecessor device output and memory writes (PW implied)	SO



Bit FIOM is needed in `menvcfg` so M-mode can emulate the *H* (hypervisor) extension, which has an equivalent FIOM bit in the hypervisor CSR `henvcfg`.

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode and G-stage address translation (i.e., for page tables pointed to by `satp` or `hgatp`). When PBMTE=1, Svpbmt is available for S-mode and G-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read-only zero. Furthermore, for implementations with the hypervisor extension, `henvcfg.PBMTE` is read-only zero if `menvcfg.PBMTE` is zero.

After changing `menvcfg.PBMTE`, executing an SFENCE.VMA instruction with *rs1*=x0 and *rs2*=x0 suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' PBMT fields. See [hyp-mm-fences] for additional synchronization requirements when the hypervisor extension is implemented.

If the Svadu extension is implemented, the ADUE bit controls whether hardware updating of PTE A/D bits is enabled for S-mode and G-stage address translations. When ADUE=1, hardware updating of PTE A/D bits is enabled during S-mode address translation, and the implementation behaves as though the Svade extension were not implemented for S-mode address translation. When the hypervisor extension is implemented, if ADUE=1, hardware updating of PTE A/D bits is enabled during G-stage address translation, and the implementation behaves as though the Svade extension were not implemented for G-stage address translation. When ADUE=0, the implementation behaves as though Svade were implemented for S-mode and G-stage address translation. If Svadu is not implemented, ADUE is read-only zero. Furthermore, for implementations with the hypervisor extension, `henvcfg.ADUE` is read-only zero if `menvcfg.ADUE` is zero.



The Svade extension requires page-fault exceptions be raised when PTE A/D bits need be set, hence Svade is implemented when ADUE=0.

If the Smcdeleg extension is implemented, the CDE (Counter Delegation Enable) bit controls whether Zicntr and Zihpm counters can be delegated to S-mode. When CDE=1, the Smcdeleg extension is enabled, see [smcdeleg]. When CDE=0, the Smcdeleg and Ssccfg extensions appear to be not implemented. If Smcdeleg is not implemented, CDE is read-only zero.

The definition of the STCE field is furnished by the Sstc extension.

The definition of the CBZE field is furnished by the Zicboz extension.

The definitions of the CBCFE and CBIE fields are furnished by the Zicbom extension.

The definition of the PMM field will be furnished by the forthcoming Smnpm extension. Its allocation within `menvcfg` may change prior to the ratification of that extension.

The Zicfilp extension adds the LPE field in `menvcfg`. When the LPE field is set to 1 and S-mode is implemented, the Zicfilp extension is enabled in S-mode. If LPE field is set to 1 and S-mode is not implemented, the Zicfilp extension is enabled in U-mode. When the LPE field is 0, the Zicfilp extension is not enabled in S-mode, and the following rules apply to S-mode. If the LPE field is 0 and S-mode is not implemented, then the same rules apply to U-mode.

- The hart does not update the ELP state; it remains as NO_LP_EXPECTED.
- The LPAD instruction operates as a no-op.

The Zicfiss extension adds the SSE field to `menvcfg`. When the SSE field is set to 1 the Zicfiss extension is activated in S-mode. When SSE field is 0, the following rules apply to privilege modes that are less than M:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS/S-stage page tables becomes reserved.
- The `henvcfg` and `senvcfg` fields will read as zero and are read-only.
- SSAM0SWAP.W/D raises an illegal-instruction exception.

The Ssdbltrp extension adds the double-trap-enable (DTE) field in `menvcfg`. When `menvcfg` is zero, the implementation behaves as though Ssdbltrp is not

implemented. When Ssdbltrp is not implemented [sstatus](#), [vsstatus](#), and [henvcfg](#) bits are read-only zero.

When XLEN=32, [menvcfg](#) is a 32-bit read/write register that aliases bits 63:32 of [menvcfg](#). The [menvcfg](#) register does not exist when XLEN=64.

If U-mode is not supported, then registers [menvcfg](#) and [menvcfgh](#) do not exist.

D.81.1. Attributes

CSR Address	0x30a
Defining extension	- allOf: - Sm, version >=1.12 - U, version >= U@1.0.0
Length	64-bit
Privilege Mode	M

D.81.2. Format

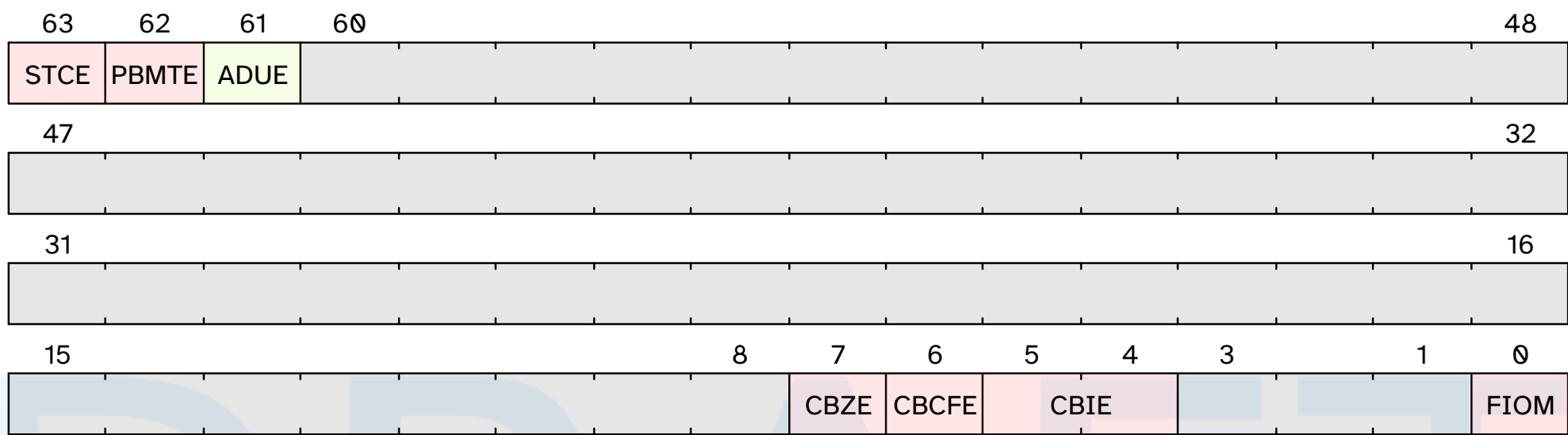


Figure 84. menvcfg format

D.81.3. Field Summary

Name	Location	Type	Reset Value
menvcfg.STCE	63	RW	UNDEFINED_LEGAL
menvcfg.PBMTE	62	RW	UNDEFINED_LEGAL
menvcfg.CBZE	7	RW	UNDEFINED_LEGAL
menvcfg.CBCFE	6	RW	UNDEFINED_LEGAL
menvcfg.CBIE	5:4	RW-R	UNDEFINED_LEGAL
menvcfg.FIOM	0	RW	UNDEFINED_LEGAL

D.81.4. Fields

[menvcfg.STCE](#) Field

Location:

63

Description:

STimecmp Enable

When set, stimecmp is operational.

When clear, stimecmp access in a mode other than M-mode raises an Illegal Instruction trap.

S-mode timer interrupts will not be generated when clear, and [mip](#) and [sip](#) revert to their prior behavior without [Sstc](#).

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.PBMTE Field

Location:

62

Description:

Page Based Memory Type Enable

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode<% if ext?(:H) %>and G-stage<% end %> address translation (i.e., for page tables pointed to by satp<% if ext?(:H) %> or hgatp<% end %>). When PBMTE=1, Svpbmt is available for S-mode <% if ext?(:H) %> and G-stage <% end %> address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read-only zero.

<% if ext?(:H) %>
Furthermore, menvcfg.PBMTE is read-only zero if
menvcfg.PBMTE is zero.
<% end %>

After changing [menvcfg.PBMTE](#), executing an [sfence.vma](#) instruction with rs1=x0 and rs2=x0 suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' PBMT fields.

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.CBZE Field

Location:

7

Description:

Cache Block Zero instruction Enable

Enables the execution of the cache block zero instruction, CB0.ZERO,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 0: The instruction raises an illegal instruction or virtual instruction exception
- 1: The instruction is executed

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.CBCFE Field

Location:

6

Description:

Cache Block Clean and Flush instruction Enable

Enables the execution of the cache block clean instruction, CB0.CLEAN, and the cache block flush instruction, CB0.FLUSH,

<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 0: The instruction raises an illegal instruction or virtual instruction exception
- 1: The instruction is executed

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.CBIE Field

Location:

5:4

Description:

Cache Block Invalidate instruction Enable

Enables the execution of the cache block invalidate instruction, CB0.INVALID,

<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 00: The instruction raises an illegal instruction or virtual instruction exception
- 01: The instruction is executed and performs a flush operation
- 10: Reserved
- 11: The instruction is executed and performs an invalidate operation

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

menvcfg.FIOM Field

Location:

0

Description:

Fence of I/O implies Memory

When [menvcfg.FIOM](#) is set,
FENCE instructions ordering I/O regions also implicitly order memory regions when executed in any mode less privileged than M-mode.

[separator="!",%autowidth,float="center",align="center",cols="^,<",options="header"]
!===
!Instruction bit !Meaning when set
!PI +
PO
!Predecessor device input and memory reads (PR implied) +
Predecessor device output and memory writes (PW implied)
!SI +
SO
!Successor device input and memory reads (SR implied) +
Successor device output and memory writes (SW implied)
!===

RVA22 Profile Release | © RISC-V International

Similarly, for modes less privileged than *M* when *FIOM*=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

Type:

RW

Reset value:

UNDEFINED_LEGAL

D.81.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
STCE = csr_value.STCE
PBMTE = csr_value.PBMTE
CBZE = csr_value.CBZE
CBCFE = csr_value.CBCFE
CBIE = if ((csr_value.CBIE == 0) ||
           (csr_value.CBIE == 1) ||
           ((!FORCE_UPGRADE_CB0_INVALID_TO_FLUSH) && (csr_value.CBIE == 3))) {
    return csr_value.CBIE;
} else {
    return CSR[menvcfg].CBIE;
}

FIOM = csr_value.FIOM
```



D.82. menvcfgh

Machine Environment Configuration



menvcfgh is only defined in RV32.

Contains bits to enable/disable extensions

D.82.1. Attributes

CSR Address	0x31a
Defining extension	- allOf: - Sm, version >=1.12 - U, version >= U@1.0.0
Length	32-bit
Privilege Mode	M

D.82.2. Format

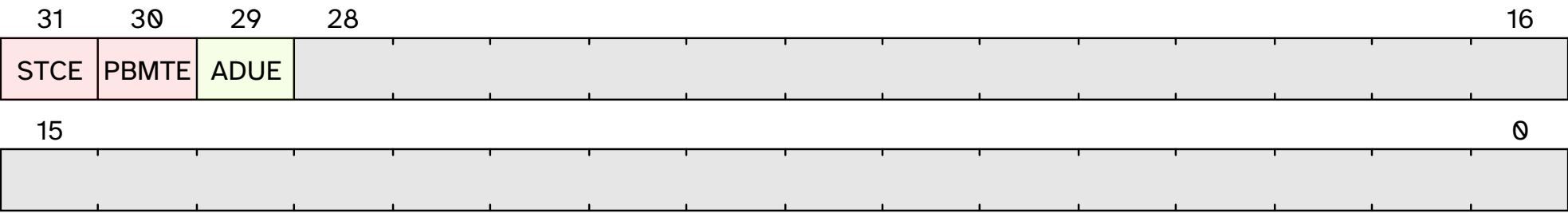


Figure 85. menvcfgh format

D.82.3. Field Summary

Name	Location	Type	Reset Value
menvcfgh.STCE	31	RW	UNDEFINED_LEGAL
menvcfgh.PBMTE	30	RW	UNDEFINED_LEGAL

D.82.4. Fields

[menvcfgh.STCE](#) Field

Location:
31

Description:
STimecmp **E**nable

Alias of [menvcfg.STCE](#)

Type:
RW

Reset value:
UNDEFINED_LEGAL

[menvcfgh.PBMTE](#) Field

Location:
30

Description:
Page Based Memory Type **E**nable

Alias of [menvcfg.PBMTE](#)

Type:

RW

Reset value:

UNDEFINED_LEGAL

DRAFT

D.83. mhpmevent10h

Machine Hardware Performance Counter 10 Control, High half



mhpmevent10h is only defined in RV32.

Alias of [mhpmevent10](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.83.1. Attributes

CSR Address	0x72a
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.83.2. Format

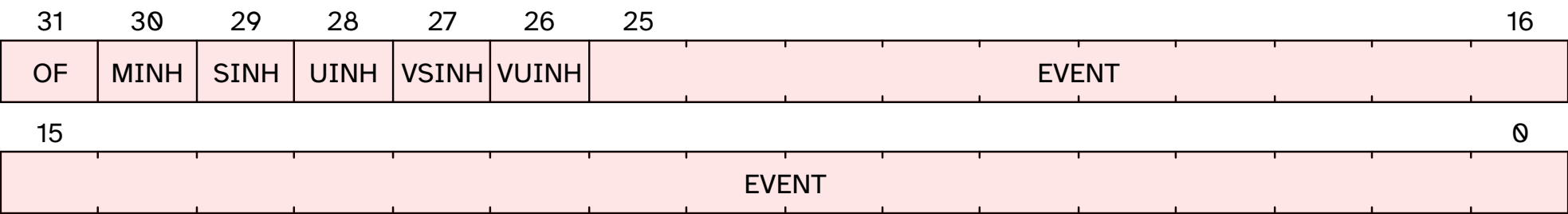


Figure 86. *mhpmevent10h* format

D.83.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent10h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent10h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent10h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent10h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent10h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent10h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent10h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.83.4. Fields

[mhpmevent10h.OF](#) Field

Location:

31

Description:

Alias of [mhpmevent10.OF](#).

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent10h.MINH Field

Location:
30

Description:
Alias of mhpmevent10.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent10h.SINH Field

Location:
29

Description:
Alias of mhpmevent10.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent10h.UINH Field

Location:
28

Description:
Alias of mhpmevent10.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent10h.VSINH Field

Location:
27

Description:
Alias of mhpmevent10.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent10h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent10.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent10h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter10.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.84. mhpmevent11h

Machine Hardware Performance Counter 11 Control, High half



mhpmevent11h is only defined in RV32.

Alias of [mhpmevent11](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.84.1. Attributes

CSR Address	0x72b
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.84.2. Format

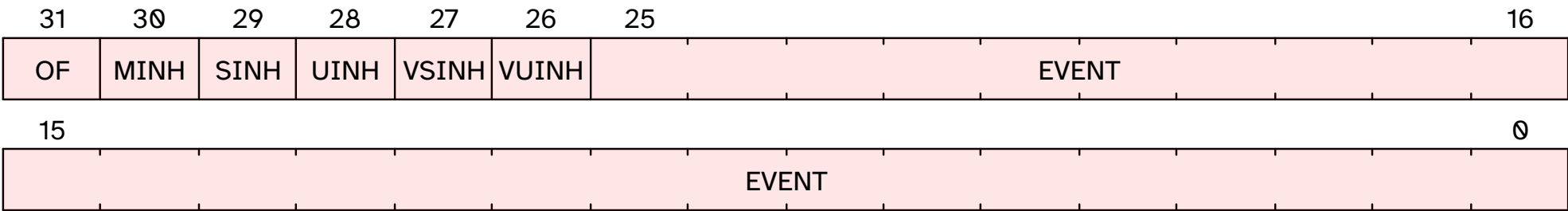


Figure 87. mhpmevent11h format

D.84.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent11h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent11h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent11h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent11h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent11h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent11h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent11h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.84.4. Fields

[mhpmevent11h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent11.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent11h.MINH Field

Location:
30

Description:
Alias of mhpmevent11.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent11h.SINH Field

Location:
29

Description:
Alias of mhpmevent11.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent11h.UINH Field

Location:
28

Description:
Alias of mhpmevent11.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent11h.VSINH Field

Location:
27

Description:
Alias of mhpmevent11.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent11h.VUINH Field

Location:
26

Description:
Alias of mhpmevent11.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent11h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter11](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.85. mhpmevent12h

Machine Hardware Performance Counter 12 Control, High half



mhpmevent12h is only defined in RV32.

Alias of [mhpmevent12](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.85.1. Attributes

CSR Address	0x72c
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.85.2. Format

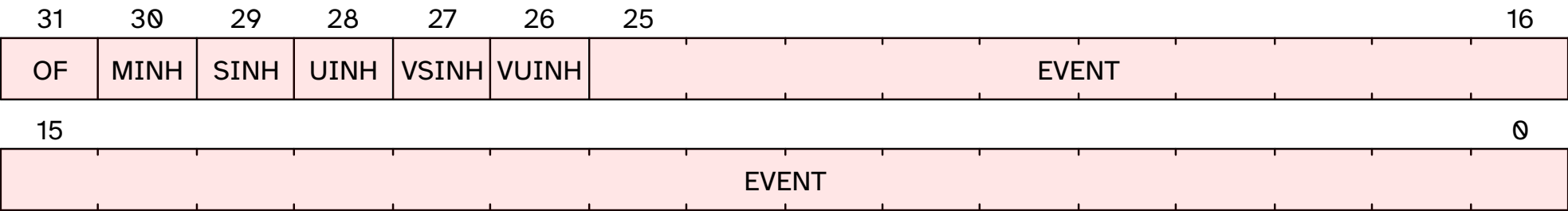


Figure 88. mhpmevent12h format

D.85.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent12h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent12h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent12h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent12h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent12h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent12h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent12h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.85.4. Fields

[mhpmevent12h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent12.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent12h.MINH Field

Location:
30

Description:
Alias of mhpmevent12.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent12h.SINH Field

Location:
29

Description:
Alias of mhpmevent12.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent12h.UINH Field

Location:
28

Description:
Alias of mhpmevent12.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent12h.VSINH Field

Location:
27

Description:
Alias of mhpmevent12.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent12h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent12.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent12h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter12.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.86. mhpmevent13h

Machine Hardware Performance Counter 13 Control, High half



mhpmevent13h is only defined in RV32.

Alias of [mhpmevent13](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.86.1. Attributes

CSR Address	0x72d
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.86.2. Format

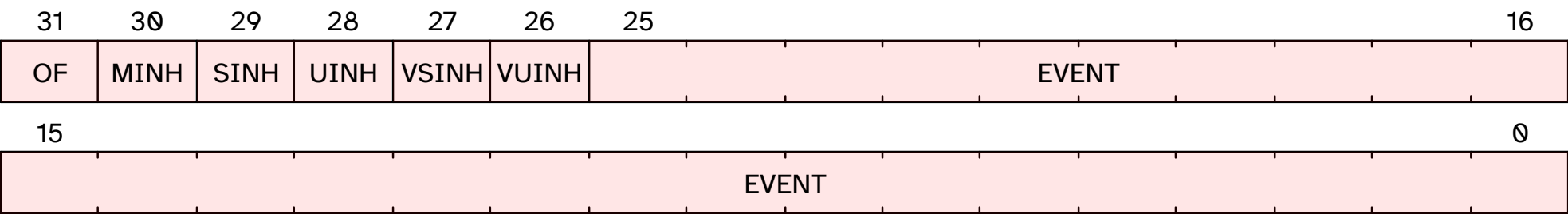


Figure 89. mhpmevent13h format

D.86.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent13h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent13h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent13h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent13h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent13h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent13h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent13h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.86.4. Fields

[mhpmevent13h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent13.OF*.

Type:

RW-H

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent13h.MINH Field

Location: 30
Description: <i>Alias of mhpmevent13.MINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent13h.SINH Field

Location: 29
Description: <i>Alias of mhpmevent13.SINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent13h.UINH Field

Location: 28
Description: <i>Alias of mhpmevent13.UINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent13h.VSINH Field

Location: 27
Description: <i>Alias of mhpmevent13.VSINH.</i>
Type: RW RO

Reset value:
UNDEFINED_LEGAL

mhpmevent13h.VUINH Field

Location:
26

Description:
Alias of mhpmevent13.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent13h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter13](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.87. mhpmevent14h

Machine Hardware Performance Counter 14 Control, High half



mhpmevent14h is only defined in RV32.

Alias of [mhpmevent14](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.87.1. Attributes

CSR Address	0x72e
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.87.2. Format

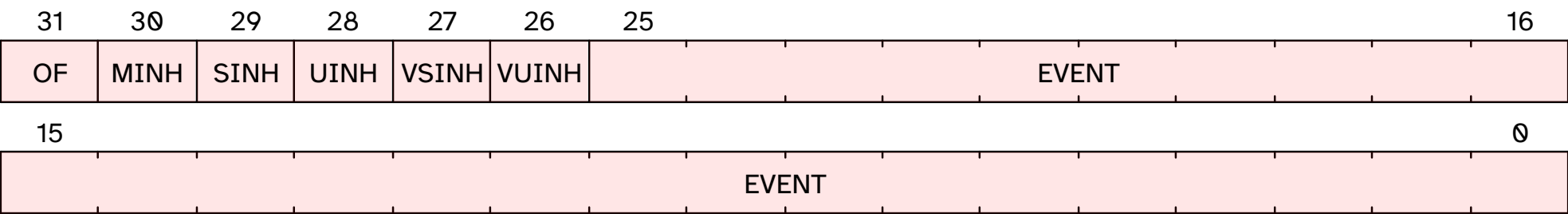


Figure 90. mhpmevent14h format

D.87.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent14h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent14h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent14h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent14h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent14h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent14h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent14h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.87.4. Fields

[mhpmevent14h.OF](#) Field

Location:

31

Description:

Alias of mhpmevent14.OF.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent14h.MINH Field

Location:
30

Description:
Alias of mhpmevent14.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent14h.SINH Field

Location:
29

Description:
Alias of mhpmevent14.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent14h.UINH Field

Location:
28

Description:
Alias of mhpmevent14.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent14h.VSINH Field

Location:
27

Description:
Alias of mhpmevent14.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent14h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent14.VUINH.</i>
Type: <i>RW</i> <i>RO</i>
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent14h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter14.</i>
Type: <i>RW</i> <i>RO</i>
Reset value: <i>UNDEFINED_LEGAL</i>

D.88. mhpmevent15h

Machine Hardware Performance Counter 15 Control, High half



mhpmevent15h is only defined in RV32.

Alias of [mhpmevent15](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.88.1. Attributes

CSR Address	0x72f
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.88.2. Format

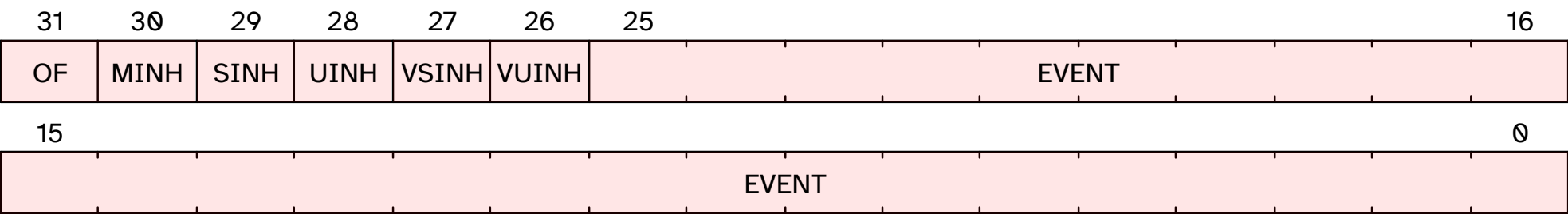


Figure 91. mhpmevent15h format

D.88.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent15h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent15h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent15h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent15h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent15h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent15h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent15h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.88.4. Fields

[mhpmevent15h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent15.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent15h.MINH Field

Location:
30

Description:
Alias of mhpmevent15.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent15h.SINH Field

Location:
29

Description:
Alias of mhpmevent15.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent15h.UINH Field

Location:
28

Description:
Alias of mhpmevent15.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent15h.VSINH Field

Location:
27

Description:
Alias of mhpmevent15.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent15h.VUINH Field

Location:
26

Description:
Alias of mhpmevent15.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent15h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter15](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.89. mhpmevent16h

Machine Hardware Performance Counter 16 Control, High half



mhpmevent16h is only defined in RV32.

Alias of [mhpmevent16](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.89.1. Attributes

CSR Address	0x730
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.89.2. Format

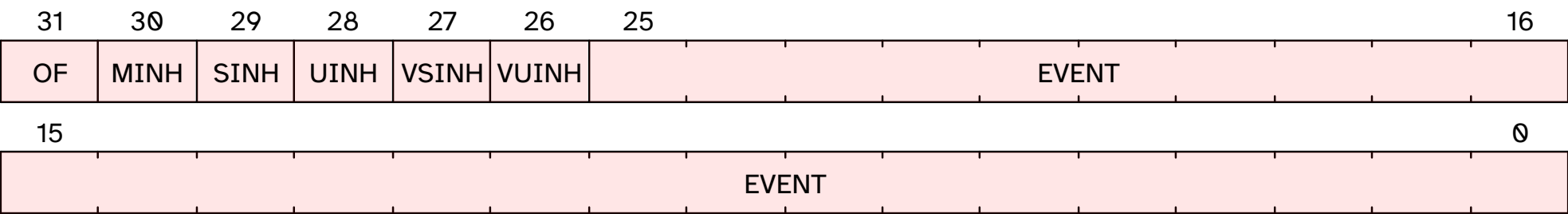


Figure 92. mhpmevent16h format

D.89.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent16h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent16h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent16h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent16h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent16h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent16h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent16h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.89.4. Fields

[mhpmevent16h.OF](#) Field

Location:

31

Description:

Alias of mhpmevent16.OF.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent16h.MINH Field

Location:
30

Description:
Alias of mhpmevent16.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent16h.SINH Field

Location:
29

Description:
Alias of mhpmevent16.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent16h.UINH Field

Location:
28

Description:
Alias of mhpmevent16.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent16h.VSINH Field

Location:
27

Description:
Alias of mhpmevent16.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent16h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent16.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent16h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter16.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.90. mhpmevent17h

Machine Hardware Performance Counter 17 Control, High half



mhpmevent17h is only defined in RV32.

Alias of [mhpmevent17](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.90.1. Attributes

CSR Address	0x731
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.90.2. Format

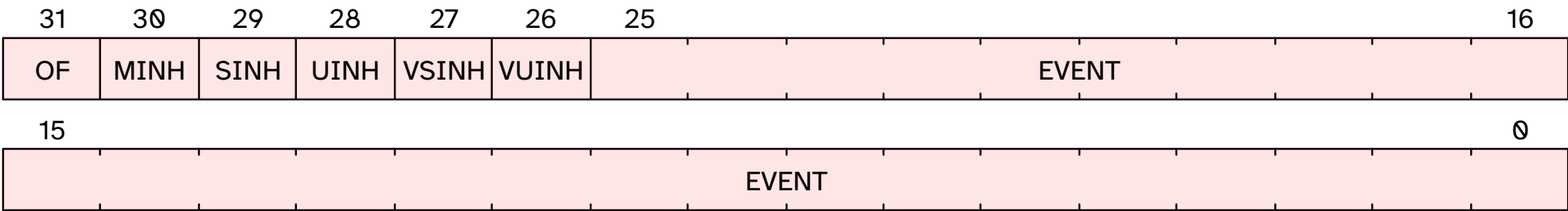


Figure 93. mhpmevent17h format

D.90.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent17h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent17h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent17h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent17h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent17h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent17h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent17h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.90.4. Fields

[mhpmevent17h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent17.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent17h.MINH Field

Location:
30

Description:
Alias of mhpmevent17.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent17h.SINH Field

Location:
29

Description:
Alias of mhpmevent17.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent17h.UINH Field

Location:
28

Description:
Alias of mhpmevent17.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent17h.VSINH Field

Location:
27

Description:
Alias of mhpmevent17.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent17h.VUINH Field

Location:
26

Description:
Alias of mhpmevent17.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent17h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter17](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.91. mhpmevent18h

Machine Hardware Performance Counter 18 Control, High half



mhpmevent18h is only defined in RV32.

Alias of [mhpmevent18](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.91.1. Attributes

CSR Address	0x732
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.91.2. Format

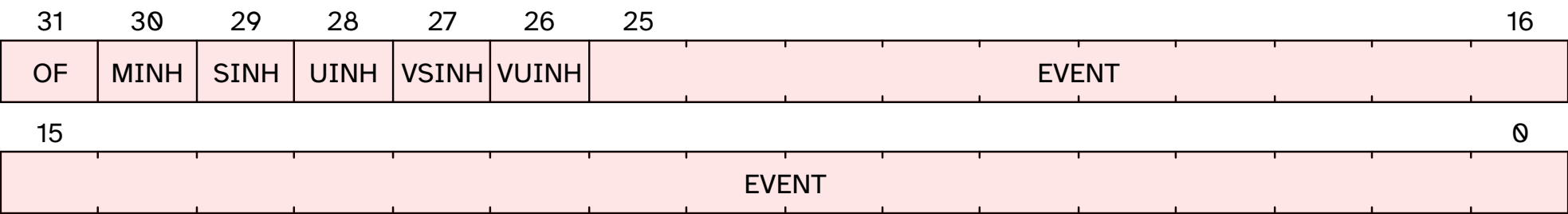


Figure 94. mhpmevent18h format

D.91.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent18h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent18h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent18h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent18h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent18h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent18h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent18h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.91.4. Fields

[mhpmevent18h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent18.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent18h.MINH Field

Location:
30

Description:
Alias of mhpmevent18.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent18h.SINH Field

Location:
29

Description:
Alias of mhpmevent18.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent18h.UINH Field

Location:
28

Description:
Alias of mhpmevent18.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent18h.VSINH Field

Location:
27

Description:
Alias of mhpmevent18.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent18h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent18.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent18h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter18.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.92. mhpmevent19h

Machine Hardware Performance Counter 19 Control, High half



mhpmevent19h is only defined in RV32.

Alias of [mhpmevent19](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.92.1. Attributes

CSR Address	0x733
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.92.2. Format

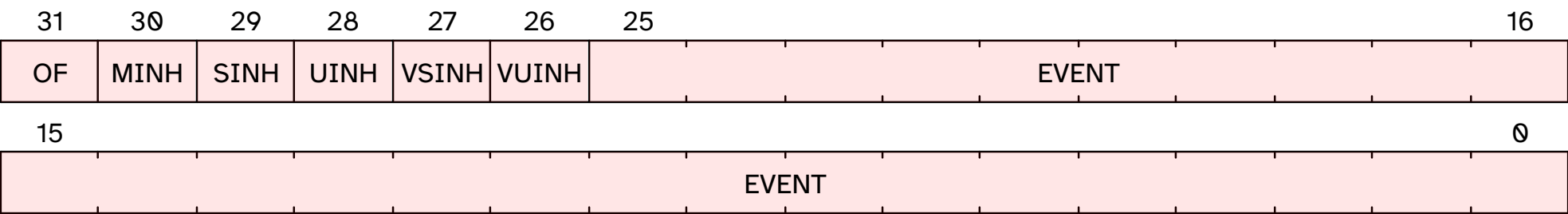


Figure 95. mhpmevent19h format

D.92.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent19h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent19h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent19h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent19h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent19h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent19h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent19h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.92.4. Fields

[mhpmevent19h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent19.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent19h.MINH Field

Location:
30

Description:
Alias of mhpmevent19.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent19h.SINH Field

Location:
29

Description:
Alias of mhpmevent19.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent19h.UINH Field

Location:
28

Description:
Alias of mhpmevent19.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent19h.VSINH Field

Location:
27

Description:
Alias of mhpmevent19.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent19h.VUINH Field

Location:
26

Description:
Alias of mhpmevent19.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent19h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter19](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.93. mhpmevent20h

Machine Hardware Performance Counter 20 Control, High half



mhpmevent20h is only defined in RV32.

Alias of mhpmevent20[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of mhpmevent#{hpm_num}.

D.93.1. Attributes

CSR Address	0x734
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.93.2. Format

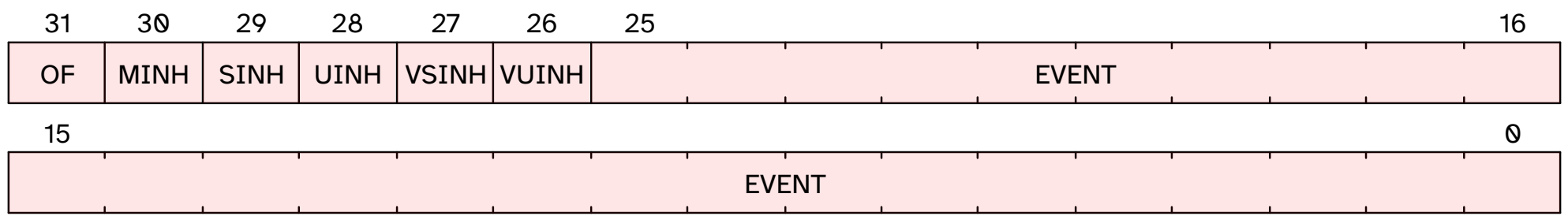


Figure 96. mhpmevent20h format

D.93.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent20h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent20h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent20h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent20h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent20h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent20h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent20h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.93.4. Fields

mhpmevent20h.OF Field

Location:

31

Description:

Alias of mhpmevent20.OF.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent20h.MINH Field

Location:
30

Description:
Alias of mhpmevent20.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent20h.SINH Field

Location:
29

Description:
Alias of mhpmevent20.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent20h.UINH Field

Location:
28

Description:
Alias of mhpmevent20.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent20h.VSINH Field

Location:
27

Description:
Alias of mhpmevent20.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent20h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent20.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent20h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter20.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.94. mhpmevent21h

Machine Hardware Performance Counter 21 Control, High half



mhpmevent21h is only defined in RV32.

Alias of [mhpmevent21](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.94.1. Attributes

CSR Address	0x735
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.94.2. Format

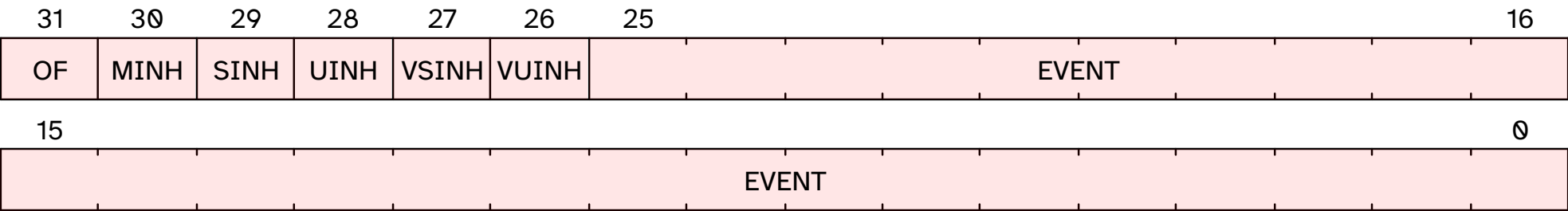


Figure 97. mhpmevent21h format

D.94.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent21h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent21h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent21h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent21h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent21h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent21h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent21h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.94.4. Fields

[mhpmevent21h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent21.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent21h.MINH Field

Location:
30

Description:
Alias of mhpmevent21.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent21h.SINH Field

Location:
29

Description:
Alias of mhpmevent21.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent21h.UINH Field

Location:
28

Description:
Alias of mhpmevent21.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent21h.VSINH Field

Location:
27

Description:
Alias of mhpmevent21.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent21h.VUINH Field

Location:
26

Description:
Alias of mhpmevent21.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent21h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter21](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.95. mhpmevent22h

Machine Hardware Performance Counter 22 Control, High half



mhpmevent22h is only defined in RV32.

Alias of [mhpmevent22](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.95.1. Attributes

CSR Address	0x736
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.95.2. Format

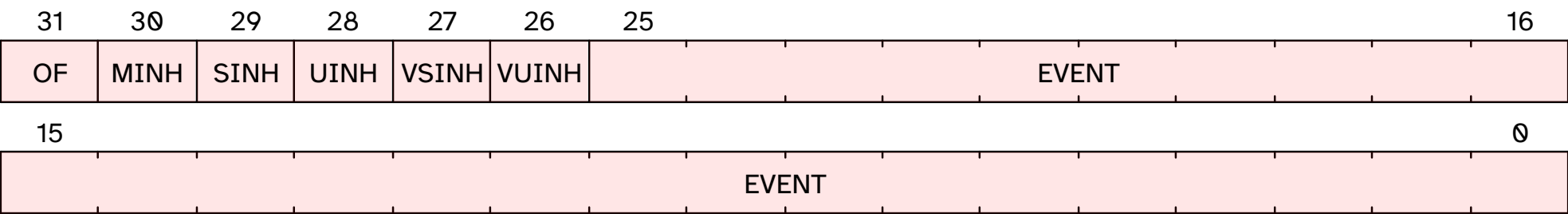


Figure 98. *mhpmevent22h* format

D.95.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent22h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent22h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent22h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent22h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent22h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent22h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent22h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.95.4. Fields

[mhpmevent22h.OF](#) Field

Location:

31

Description:

Alias of [mhpmevent22.OF](#).

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent22h.MINH Field

Location:
30

Description:
Alias of mhpmevent22.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent22h.SINH Field

Location:
29

Description:
Alias of mhpmevent22.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent22h.UINH Field

Location:
28

Description:
Alias of mhpmevent22.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent22h.VSINH Field

Location:
27

Description:
Alias of mhpmevent22.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent22h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent22.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent22h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter22.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.96. mhpmevent23h

Machine Hardware Performance Counter 23 Control, High half



mhpmevent23h is only defined in RV32.

Alias of [mhpmevent23](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.96.1. Attributes

CSR Address	0x737
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.96.2. Format

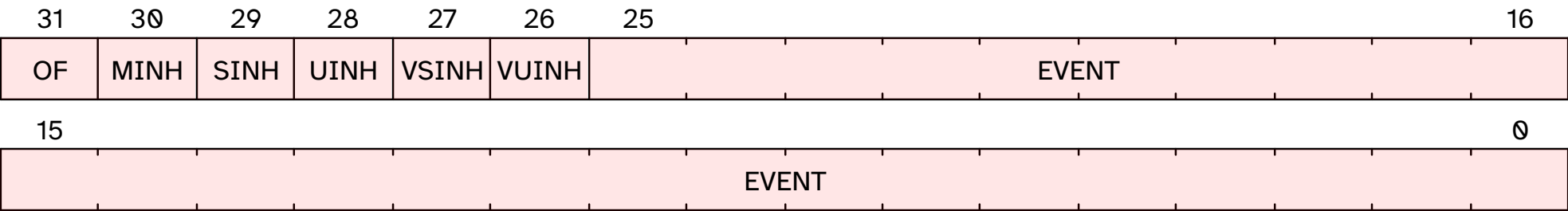


Figure 99. mhpmevent23h format

D.96.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent23h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent23h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent23h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent23h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent23h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent23h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent23h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.96.4. Fields

[mhpmevent23h.OF](#) Field

Location:

31

Description:

Alias of [mhpmevent23.OF](#).

Type:

RW-H

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent23h.MINH Field

Location: 30
Description: <i>Alias of mhpmevent23.MINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent23h.SINH Field

Location: 29
Description: <i>Alias of mhpmevent23.SINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent23h.UINH Field

Location: 28
Description: <i>Alias of mhpmevent23.UINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent23h.VSINH Field

Location: 27
Description: <i>Alias of mhpmevent23.VSINH.</i>
Type: RW RO

Reset value:
UNDEFINED_LEGAL

mhpmevent23h.VUINH Field

Location:
26

Description:
Alias of mhpmevent23.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent23h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter23](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.97. mhpmevent24h

Machine Hardware Performance Counter 24 Control, High half



mhpmevent24h is only defined in RV32.

Alias of [mhpmevent24](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.97.1. Attributes

CSR Address	0x738
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.97.2. Format

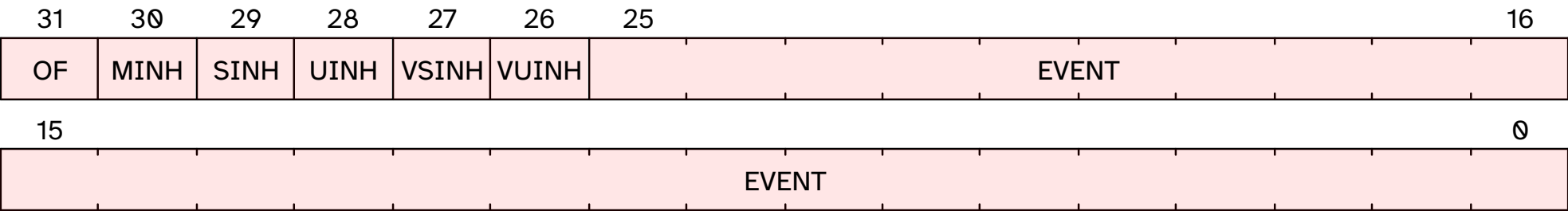


Figure 100. *mhpmevent24h* format

D.97.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent24h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent24h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent24h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent24h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent24h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent24h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent24h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.97.4. Fields

[mhpmevent24h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent24.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent24h.MINH Field

Location:
30

Description:
Alias of mhpmevent24.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent24h.SINH Field

Location:
29

Description:
Alias of mhpmevent24.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent24h.UINH Field

Location:
28

Description:
Alias of mhpmevent24.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent24h.VSINH Field

Location:
27

Description:
Alias of mhpmevent24.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent24h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent24.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent24h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter24.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.98. mhpmevent25h

Machine Hardware Performance Counter 25 Control, High half



mhpmevent25h is only defined in RV32.

Alias of [mhpmevent25](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.98.1. Attributes

CSR Address	0x739
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.98.2. Format

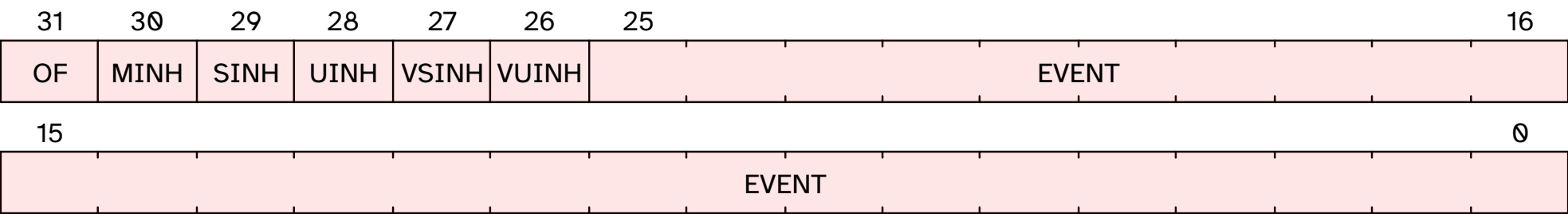


Figure 101. *mhpmevent25h* format

D.98.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent25h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent25h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent25h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent25h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent25h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent25h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent25h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.98.4. Fields

[mhpmevent25h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent25.OF*.

Type:

RW-H

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent25h.MINH Field

Location: 30
Description: <i>Alias of mhpmevent25.MINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent25h.SINH Field

Location: 29
Description: <i>Alias of mhpmevent25.SINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent25h.UINH Field

Location: 28
Description: <i>Alias of mhpmevent25.UINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent25h.VSINH Field

Location: 27
Description: <i>Alias of mhpmevent25.VSINH.</i>
Type: RW RO

Reset value:
UNDEFINED_LEGAL

mhpmevent25h.VUINH Field

Location:
26

Description:
Alias of mhpmevent25.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent25h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter25](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.99. mhpmevent26h

Machine Hardware Performance Counter 26 Control, High half



mhpmevent26h is only defined in RV32.

Alias of [mhpmevent26](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.99.1. Attributes

CSR Address	0x73a
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.99.2. Format

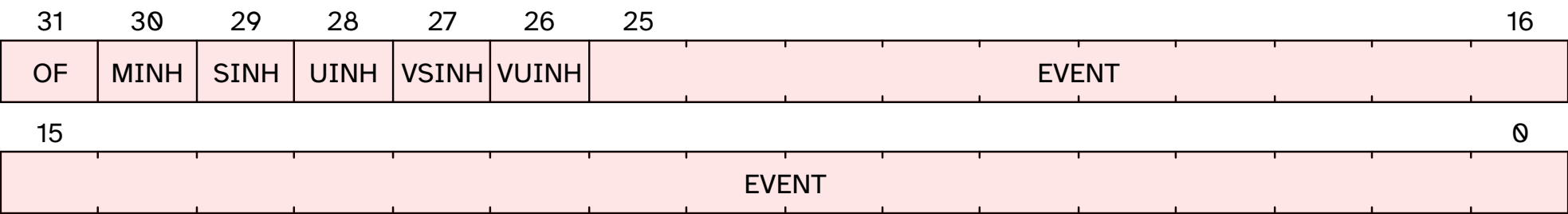


Figure 102. *mhpmevent26h* format

D.99.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent26h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent26h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent26h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent26h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent26h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent26h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent26h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.99.4. Fields

[mhpmevent26h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent26.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent26h.MINH Field

Location:
30

Description:
Alias of mhpmevent26.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent26h.SINH Field

Location:
29

Description:
Alias of mhpmevent26.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent26h.UINH Field

Location:
28

Description:
Alias of mhpmevent26.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent26h.VSINH Field

Location:
27

Description:
Alias of mhpmevent26.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent26h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent26.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent26h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter26.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.100. mhpmevent27h

Machine Hardware Performance Counter 27 Control, High half



mhpmevent27h is only defined in RV32.

Alias of mhpmevent27[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of mhpmevent#{hpm_num}.

D.100.1. Attributes

CSR Address	0x73b
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.100.2. Format

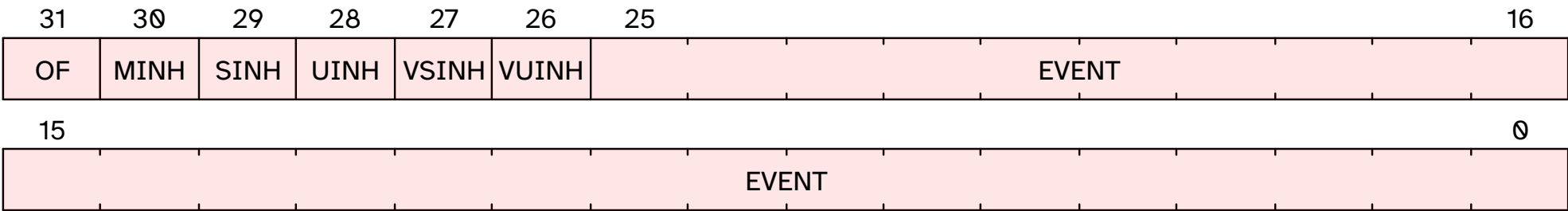


Figure 103. mhpmevent27h format

D.100.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent27h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent27h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent27h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent27h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent27h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent27h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent27h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.100.4. Fields

mhpmevent27h.OF Field

Location:

31

Description:

Alias of mhpmevent27.OF.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent27h.MINH Field

Location:
30

Description:
Alias of mhpmevent27.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent27h.SINH Field

Location:
29

Description:
Alias of mhpmevent27.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent27h.UINH Field

Location:
28

Description:
Alias of mhpmevent27.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent27h.VSINH Field

Location:
27

Description:
Alias of mhpmevent27.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent27h.VUINH Field

Location:
26

Description:
Alias of mhpmevent27.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent27h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter27](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.101. mhpmevent28h

Machine Hardware Performance Counter 28 Control, High half



mhpmevent28h is only defined in RV32.

Alias of [mhpmevent28](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.101.1. Attributes

CSR Address	0x73c
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.101.2. Format

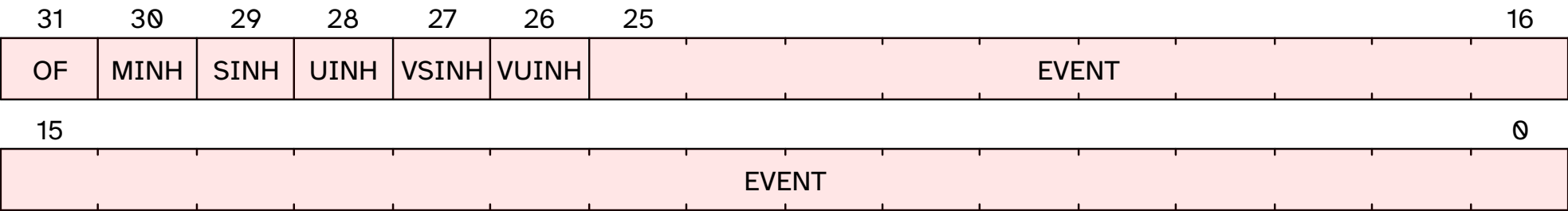


Figure 104. mhpmevent28h format

D.101.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent28h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent28h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent28h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent28h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent28h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent28h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent28h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.101.4. Fields

[mhpmevent28h.OF](#) Field

Location:

31

Description:

Alias of [mhpmevent28.OF](#).

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent28h.MINH Field

Location:
30

Description:
Alias of mhpmevent28.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent28h.SINH Field

Location:
29

Description:
Alias of mhpmevent28.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent28h.UINH Field

Location:
28

Description:
Alias of mhpmevent28.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent28h.VSINH Field

Location:
27

Description:
Alias of mhpmevent28.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent28h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent28.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent28h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter28.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.102. mhpmevent29h

Machine Hardware Performance Counter 29 Control, High half



mhpmevent29h is only defined in RV32.

Alias of [mhpmevent29](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.102.1. Attributes

CSR Address	0x73d
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.102.2. Format

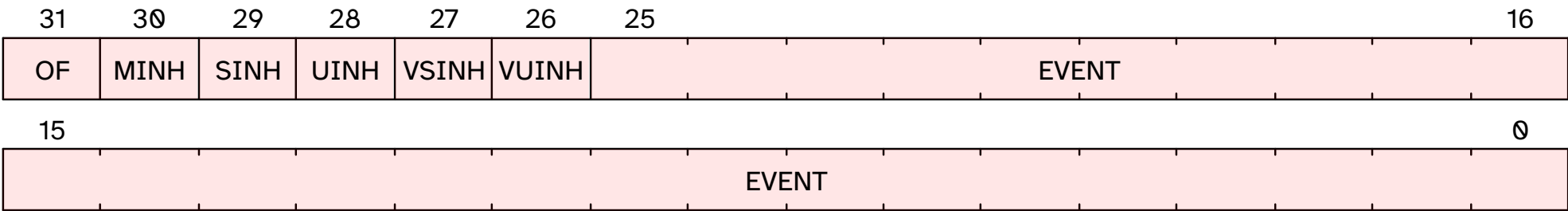


Figure 105. mhpmevent29h format

D.102.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent29h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent29h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent29h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent29h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent29h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent29h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent29h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.102.4. Fields

[mhpmevent29h.OF](#) Field

Location:

31

Description:

Alias of [mhpmevent29.OF](#).

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent29h.MINH Field

Location:
30

Description:
Alias of mhpmevent29.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent29h.SINH Field

Location:
29

Description:
Alias of mhpmevent29.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent29h.UINH Field

Location:
28

Description:
Alias of mhpmevent29.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent29h.VSINH Field

Location:
27

Description:
Alias of mhpmevent29.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent29h.VUINH Field

Location:
26

Description:
Alias of mhpmevent29.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent29h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter29](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.103. mhpmevent30h

Machine Hardware Performance Counter 30 Control, High half



mhpmevent30h is only defined in RV32.

Alias of mhpmevent30[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of mhpmevent#{hpm_num}.

D.103.1. Attributes

CSR Address	0x73e
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.103.2. Format

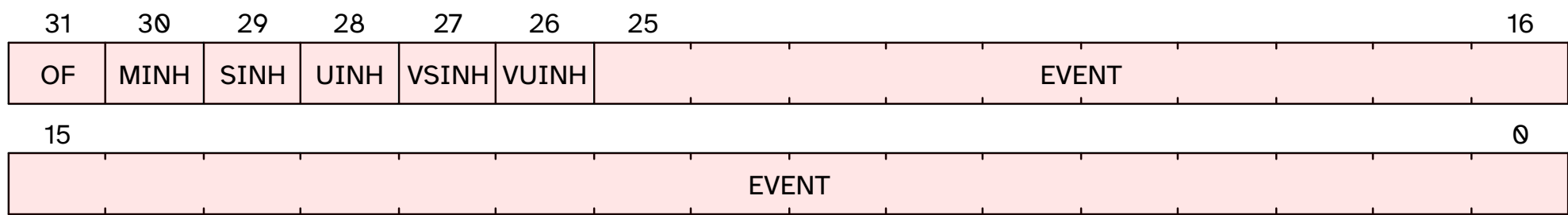


Figure 106. mhpmevent30h format

D.103.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent30h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent30h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent30h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent30h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent30h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent30h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent30h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.103.4. Fields

mhpmevent30h.OF Field

Location:

31

Description:

Alias of mhpmevent30.OF.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent30h.MINH Field

Location:
30

Description:
Alias of mhpmevent30.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent30h.SINH Field

Location:
29

Description:
Alias of mhpmevent30.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent30h.UINH Field

Location:
28

Description:
Alias of mhpmevent30.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent30h.VSINH Field

Location:
27

Description:
Alias of mhpmevent30.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent30h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent30.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent30h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter30.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.104. mhpmevent31h

Machine Hardware Performance Counter 31 Control, High half



mhpmevent31h is only defined in RV32.

Alias of mhpmevent31[63:32].

Introduced with the Sscofpmf extension. Prior to that, there was no way to access the upper 32-bits of mhpmevent#{hpm_num}.

D.104.1. Attributes

CSR Address	0x73f
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.104.2. Format

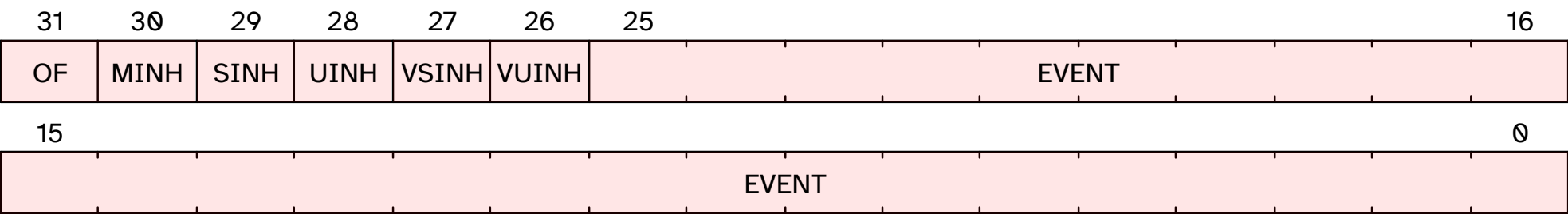


Figure 107. mhpmevent31h format

D.104.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent31h.OF	31	RW-H RO	UNDEFINED_LEGAL
mhpmevent31h.MINH	30	RW RO	UNDEFINED_LEGAL
mhpmevent31h.SINH	29	RW RO	UNDEFINED_LEGAL
mhpmevent31h.UINH	28	RW RO	UNDEFINED_LEGAL
mhpmevent31h.VSINH	27	RW RO	UNDEFINED_LEGAL
mhpmevent31h.VUINH	26	RW RO	UNDEFINED_LEGAL
mhpmevent31h.EVENT	25:0	RW RO	UNDEFINED_LEGAL

D.104.4. Fields

mhpmevent31h.OF Field

Location:

31

Description:

Alias of mhpmevent31.OF.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent31h.MINH Field

Location:
30

Description:
Alias of mhpmevent31.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent31h.SINH Field

Location:
29

Description:
Alias of mhpmevent31.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent31h.UINH Field

Location:
28

Description:
Alias of mhpmevent31.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent31h.VSINH Field

Location:
27

Description:
Alias of mhpmevent31.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent31h.VUINH Field

Location:
26

Description:
Alias of mhpmevent31.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent31h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter31](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.105. mhpmevent3h

Machine Hardware Performance Counter 3 Control, High half



mhpmevent3h is only defined in RV32.

Alias of [mhpmevent3](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.105.1. Attributes

CSR Address	0x723
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.105.2. Format

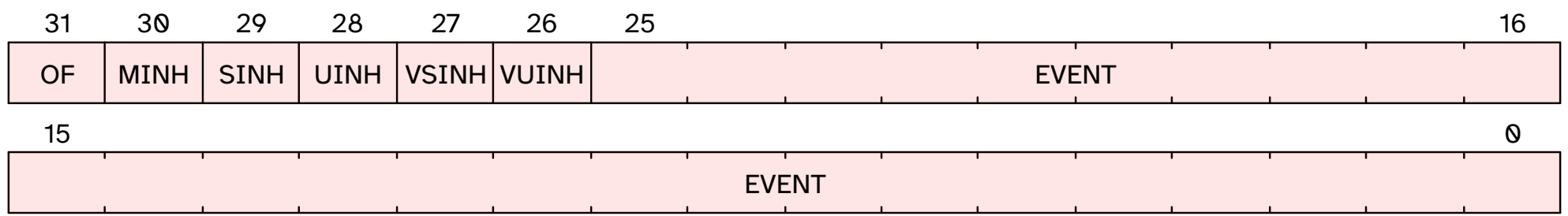


Figure 108. mhpmevent3h format

D.105.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent3h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent3h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent3h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent3h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent3h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent3h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent3h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.105.4. Fields

[mhpmevent3h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent3.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent3h.MINH Field

Location:
30

Description:
Alias of mhpmevent3.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent3h.SINH Field

Location:
29

Description:
Alias of mhpmevent3.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent3h.UINH Field

Location:
28

Description:
Alias of mhpmevent3.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent3h.VSINH Field

Location:
27

Description:
Alias of mhpmevent3.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent3h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent3.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent3h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter3.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.106. mhpmevent4h

Machine Hardware Performance Counter 4 Control, High half



mhpmevent4h is only defined in RV32.

Alias of [mhpmevent4](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.106.1. Attributes

CSR Address	0x724
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.106.2. Format

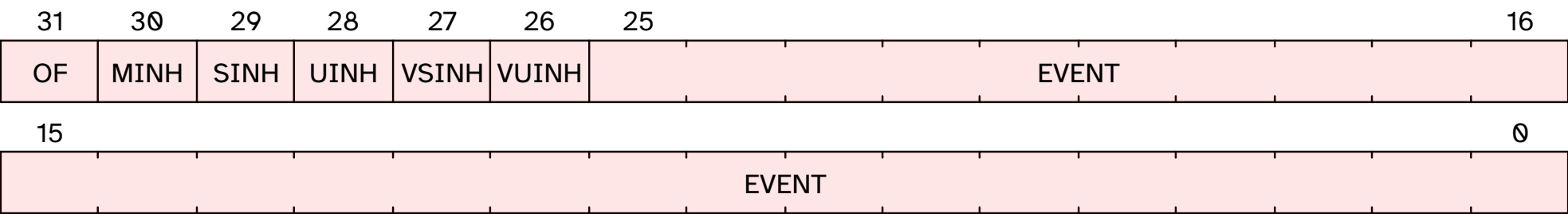


Figure 109. mhpmevent4h format

D.106.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent4h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent4h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent4h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent4h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent4h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent4h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent4h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.106.4. Fields

[mhpmevent4h.OF](#) Field

Location:

31

Description:

Alias of [mhpmevent4.OF](#).

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent4h.MINH Field

Location:
30

Description:
Alias of mhpmevent4.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent4h.SINH Field

Location:
29

Description:
Alias of mhpmevent4.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent4h.UINH Field

Location:
28

Description:
Alias of mhpmevent4.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent4h.VSINH Field

Location:
27

Description:
Alias of mhpmevent4.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent4h.VUINH Field

Location:
26

Description:
Alias of mhpmevent4.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent4h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter4](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.107. mhpmevent5h

Machine Hardware Performance Counter 5 Control, High half



mhpmevent5h is only defined in RV32.

Alias of [mhpmevent5](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.107.1. Attributes

CSR Address	0x725
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.107.2. Format

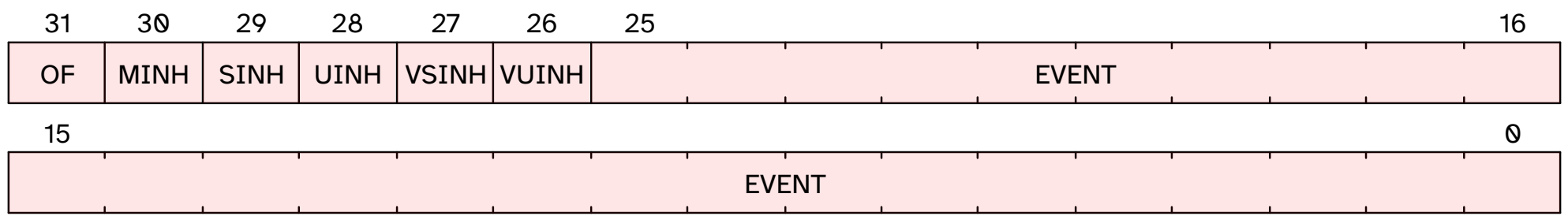


Figure 110. mhpmevent5h format

D.107.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent5h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent5h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent5h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent5h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent5h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent5h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent5h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.107.4. Fields

[mhpmevent5h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent5.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent5h.MINH Field

Location:
30

Description:
Alias of mhpmevent5.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent5h.SINH Field

Location:
29

Description:
Alias of mhpmevent5.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent5h.UINH Field

Location:
28

Description:
Alias of mhpmevent5.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent5h.VSINH Field

Location:
27

Description:
Alias of mhpmevent5.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent5h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent5.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent5h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter5.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.108. mhpmevent6h

Machine Hardware Performance Counter 6 Control, High half



mhpmevent6h is only defined in RV32.

Alias of [mhpmevent6](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.108.1. Attributes

CSR Address	0x726
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.108.2. Format

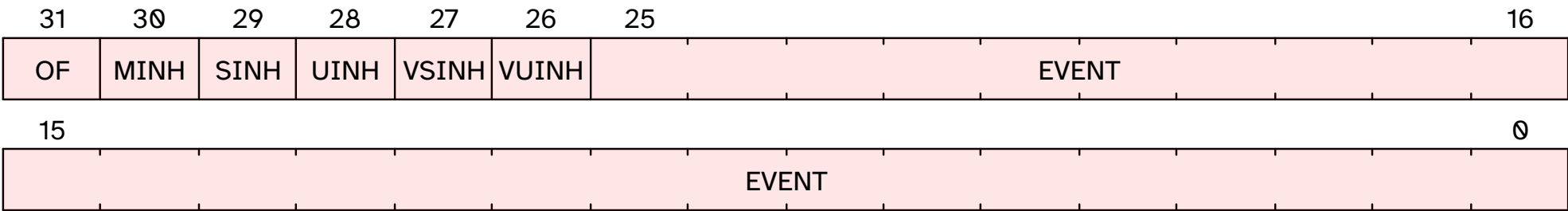


Figure 111. mhpmevent6h format

D.108.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent6h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent6h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent6h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent6h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent6h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent6h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent6h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.108.4. Fields

[mhpmevent6h.OF](#) Field

Location:

31

Description:

Alias of `mhpmevent6.OF`.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent6h.MINH Field

Location:
30

Description:
Alias of mhpmevent6.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent6h.SINH Field

Location:
29

Description:
Alias of mhpmevent6.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent6h.UINH Field

Location:
28

Description:
Alias of mhpmevent6.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent6h.VSINH Field

Location:
27

Description:
Alias of mhpmevent6.VSINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent6h.VUINH Field

Location:
26

Description:
Alias of mhpmevent6.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent6h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter6](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.109. mhpmevent7h

Machine Hardware Performance Counter 7 Control, High half



mhpmevent7h is only defined in RV32.

Alias of [mhpmevent7](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of mhpmevent#{hpm_num}.

D.109.1. Attributes

CSR Address	0x727
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.109.2. Format

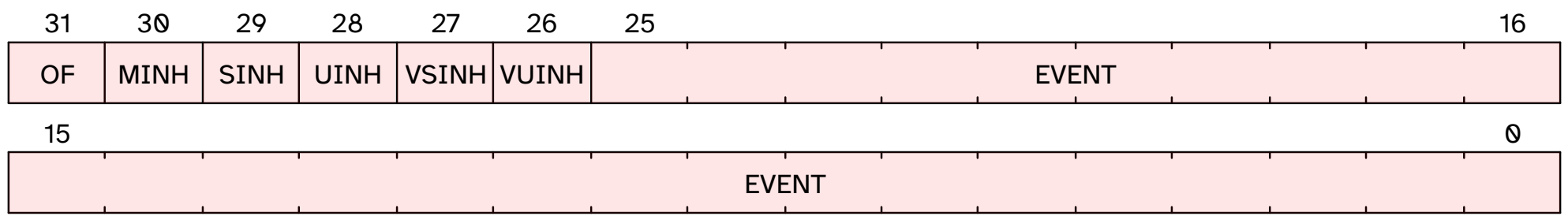


Figure 112. mhpmevent7h format

D.109.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent7h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent7h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent7h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent7h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent7h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent7h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent7h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.109.4. Fields

[mhpmevent7h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent7.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent7h.MINH Field

Location:
30

Description:
Alias of mhpmevent7.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent7h.SINH Field

Location:
29

Description:
Alias of mhpmevent7.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent7h.UINH Field

Location:
28

Description:
Alias of mhpmevent7.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent7h.VSINH Field

Location:
27

Description:
Alias of mhpmevent7.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent7h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent7.VUINH.</i>
Type: <i>RW</i> <i>RO</i>
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent7h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter7.</i>
Type: <i>RW</i> <i>RO</i>
Reset value: <i>UNDEFINED_LEGAL</i>

D.110. mhpmevent8h

Machine Hardware Performance Counter 8 Control, High half



mhpmevent8h is only defined in RV32.

Alias of [mhpmevent8](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.110.1. Attributes

CSR Address	0x728
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.110.2. Format

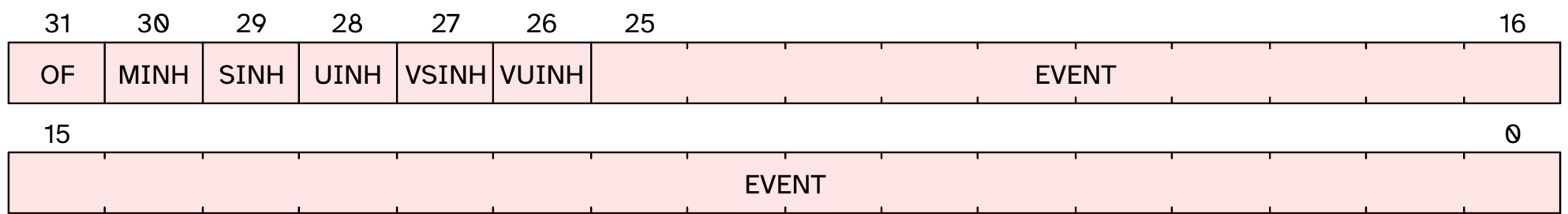


Figure 113. *mhpmevent8h* format

D.110.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent8h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent8h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent8h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent8h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent8h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent8h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent8h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.110.4. Fields

[mhpmevent8h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent8.OF*.

Type:

RW-H

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent8h.MINH Field

Location: 30
Description: <i>Alias of mhpmevent8.MINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent8h.SINH Field

Location: 29
Description: <i>Alias of mhpmevent8.SINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent8h.UINH Field

Location: 28
Description: <i>Alias of mhpmevent8.UINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent8h.VSINH Field

Location: 27
Description: <i>Alias of mhpmevent8.VSINH.</i>
Type: RW RO

Reset value:
UNDEFINED_LEGAL

mhpmevent8h.VUINH Field

Location:
26

Description:
Alias of mhpmevent8.VUINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent8h.EVENT Field

Location:
25:0

Description:
High part of event selector for performance counter [mhpmcounter8](#).

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

D.111. mhpmevent9h

Machine Hardware Performance Counter 9 Control, High half



mhpmevent9h is only defined in RV32.

Alias of [mhpmevent9](#)[63:32].

Introduced with the [Sscofpmf](#) extension. Prior to that, there was no way to access the upper 32-bits of `mhpmevent#{hpm_num}`.

D.111.1. Attributes

CSR Address	0x729
Defining extension	Sscofpmf, version >= Sscofpmf@1.0.0
Length	32-bit
Privilege Mode	M

D.111.2. Format

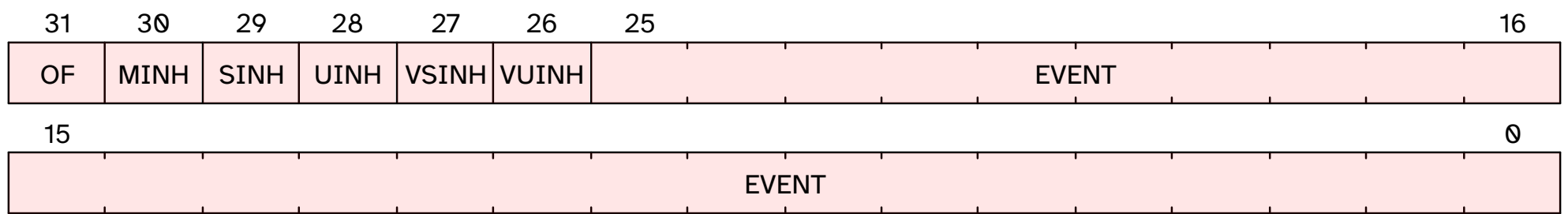


Figure 114. mhpmevent9h format

D.111.3. Field Summary

Name	Location	Type	Reset Value
mhpmevent9h.OF	31	RW-H	UNDEFINED_LEGAL
		RO	
mhpmevent9h.MINH	30	RW	UNDEFINED_LEGAL
		RO	
mhpmevent9h.SINH	29	RW	UNDEFINED_LEGAL
		RO	
mhpmevent9h.UINH	28	RW	UNDEFINED_LEGAL
		RO	
mhpmevent9h.VSINH	27	RW	UNDEFINED_LEGAL
		RO	
mhpmevent9h.VUINH	26	RW	UNDEFINED_LEGAL
		RO	
mhpmevent9h.EVENT	25:0	RW	UNDEFINED_LEGAL
		RO	

D.111.4. Fields

[mhpmevent9h.OF](#) Field

Location:

31

Description:

Alias of *mhpmevent9.OF*.

Type:

RW-H

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent9h.MINH Field

Location:
30

Description:
Alias of mhpmevent9.MINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent9h.SINH Field

Location:
29

Description:
Alias of mhpmevent9.SINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent9h.UINH Field

Location:
28

Description:
Alias of mhpmevent9.UINH.

Type:

RW

RO

Reset value:
UNDEFINED_LEGAL

mhpmevent9h.VSINH Field

Location:
27

Description:
Alias of mhpmevent9.VSINH.

Type:

RW

RO

Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent9h.VUINH Field

Location: 26
Description: <i>Alias of mhpmevent9.VUINH.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

mhpmevent9h.EVENT Field

Location: 25:0
Description: <i>High part of event selector for performance counter mhpmcounter9.</i>
Type: RW RO
Reset value: <i>UNDEFINED_LEGAL</i>

D.112. mideleg

Machine Interrupt Delegation

Controls exception delegation from M-mode to HS/S-mode

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction. To increase performance, implementations can provide individual read/write bits within mideleg to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level.

In harts with S-mode, the mideleg register must exist, and setting a bit mideleg will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler <%- if ext?:(H) -%> (which could further be delegated to VS-mode through hideleg) <%- end -%> . <%- if ext?:(S, ">1.9.1") -%> In harts without S-mode, the mideleg register should not exist.



In versions 1.9.1 and earlier , this register existed but was hardwired to zero in M-mode only, or M/U without N harts. There is no reason to require they return zero in those cases, as the misa register indicates whether they exist.

<%- else -%> In harts without S-mode, the mideleg register is read-only zero. <%- end -%>

An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in mideleg to see which bit positions hold a one.



Version 1.11 and earlier prohibited having any bits of mideleg be read-only one. Platform standards may always add such restrictions.

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M-mode has delegated illegal-instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode. By contrast, traps may be taken horizontally. Using the same example, if M-mode has delegated illegal-instruction exceptions to S-mode, and S-mode software later executes an illegal instruction, the trap is taken in S-mode.

Delegated interrupts result in the interrupt being masked at the delegator privilege level. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting mideleg[5], STIs will not be taken when executing in M-mode. By contrast, if mideleg[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.

mideleg holds trap delegation bits for individual interrupts, with the layout of bits matching those in the mip register (i.e., STIP interrupt delegation control is located in bit 5).

For exceptions that cannot occur in less privileged modes, the corresponding medeleg bits should be read-only zero. In particular, medeleg[11] is read-only zero.

The medeleg[16] is read-only zero as double trap is not delegatable.

D.112.1. Attributes

CSR Address	0x303
Defining extension	- oneOf: - Sm, version ≤ 1.9.1 - allOf: - S, version > 1.9.1 - Sm, version > 1.9.1
Length	64-bit
Privilege Mode	M

D.112.2. Format

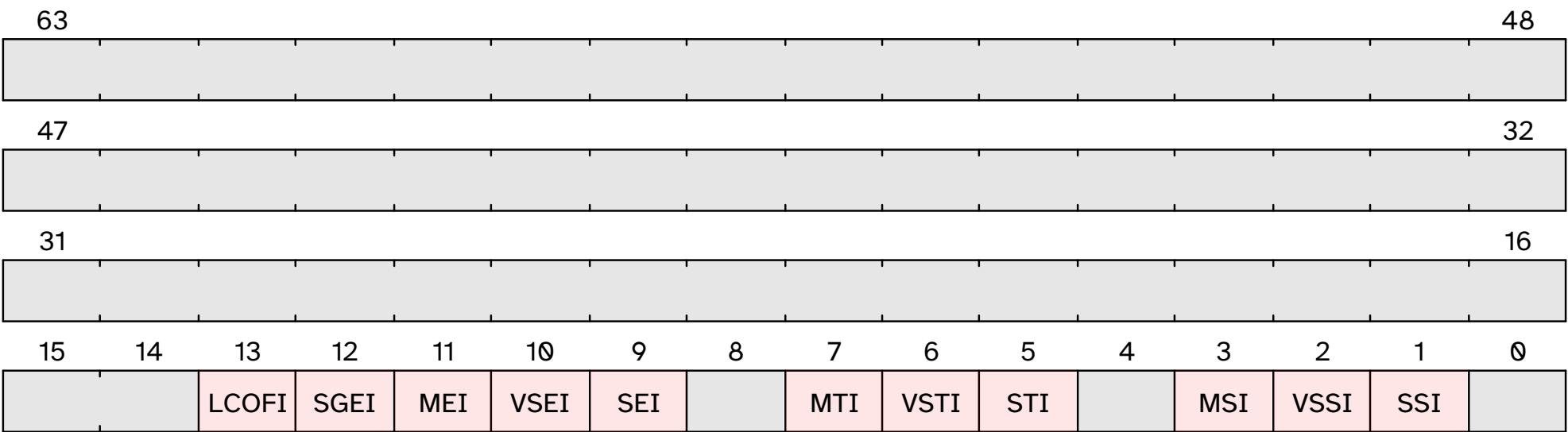


Figure 115. mideleg format

D.112.3. Field Summary

Name	Location	Type	Reset Value
mideleg.SSI	1	RW	0
mideleg.VSSI	2	RO	1
mideleg.MSI	3	RO	0
mideleg.STI	5	RW	0
mideleg.VSTI	6	RO	1
mideleg.MTI	7	RO	0
mideleg.SEI	9	RW	0
mideleg.VSEI	10	RO	1
mideleg.MEI	11	RO	0
mideleg.SGEI	12	RO	1
mideleg.LCOFI	13	RW	0

D.112.4. Fields

[mideleg.SSI](#) Field

Location:

1

Description:

Supervisor Software Interrupt delegation

When 1, Supervisor Software interrupts are delegated to HS/S-mode.

Type:

RW

Reset value:

0

[mideleg.VSSI](#) Field

Location:

2

Description:

Virtual Supervisor Software Interrupt delegation

When 1, Virtual Supervisor Software interrupts are delegated to HS-mode.

Virtual Supervisor Software Interrupts are always delegated to HS-mode, so this field is read-only one.

Type:

RO

Reset value:

1

mideleg.MSI Field

Location:

3

Description:

Machine Software interrupt delegation

Since M-mode interrupts cannot be delegated, this field is read-only zero.

Type:

RO

Reset value:

0

mideleg.STI Field

Location:

5

Description:

Supervisor Timer interrupt delegation

When 1, Supervisor Timer interrupts are delegated to HS/S-mode.

Type:

RW

Reset value:

0

mideleg.VSTI Field

Location:

6

Description:

Virtual Supervisor Timer interrupt delegation

When 1, Virtual Supervisor Timer interrupts are delegated to HS-mode.

Virtual Supervisor Time Interrupts are always delegated to HS-mode, so this field is read-only one.

Type:

RO

Reset value:

1

mideleg.MTI Field

Location:

7

Description:

Machine Timer interrupt delegation

Since M-mode interrupts cannot be delegated, this field is read-only zero.

Type:

RO

Reset value: 0

mideleg.SEI Field

Location: 9
Description: Supervisor External interrupt delegation <i>When 1, Supervisor External interrupts are delegated to HS/S-mode.</i>
Type: RW
Reset value: 0

mideleg.VSEI Field

Location: 10
Description: Virtual Supervisor External interrupt delegation <i>Virtual Supervisor External Interrupts are always delegated to HS-mode, so this field is read-only one.</i>
Type: RO
Reset value: 1

mideleg.MEI Field

Location: 11
Description: Machine External interrupt delegation <i>Since M-mode interrupts cannot be delegated, this field is read-only zero.</i>
Type: RO
Reset value: 0

mideleg.SGEI Field

Location: 12
Description: Supervisor Guest External Interrupt delegation <i>Supervisor Guest External interrupts are always delegated to HS-mode, so this field is read-only one.</i>
Type: RO
Reset value: 1

mideleg.LCOFI Field

Location:

13

Description:

Local Counter Overflow Interrupt delegation

When 1, local counter overflow interrupts are delegated to (H)S-mode.

Type:

RW

Reset value:

0

DRAFT

D.113. minstret

Machine Instructions Retired Counter

Counts the number of instructions retired by this hart from some arbitrary start point in the past.



Instructions that cause synchronous exceptions, including *ecall* and *ebreak*, are not considered to retire and hence do not increment the *minstret* CSR.

D.113.1. Attributes

CSR Address	0xb02
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	M

D.113.2. Format

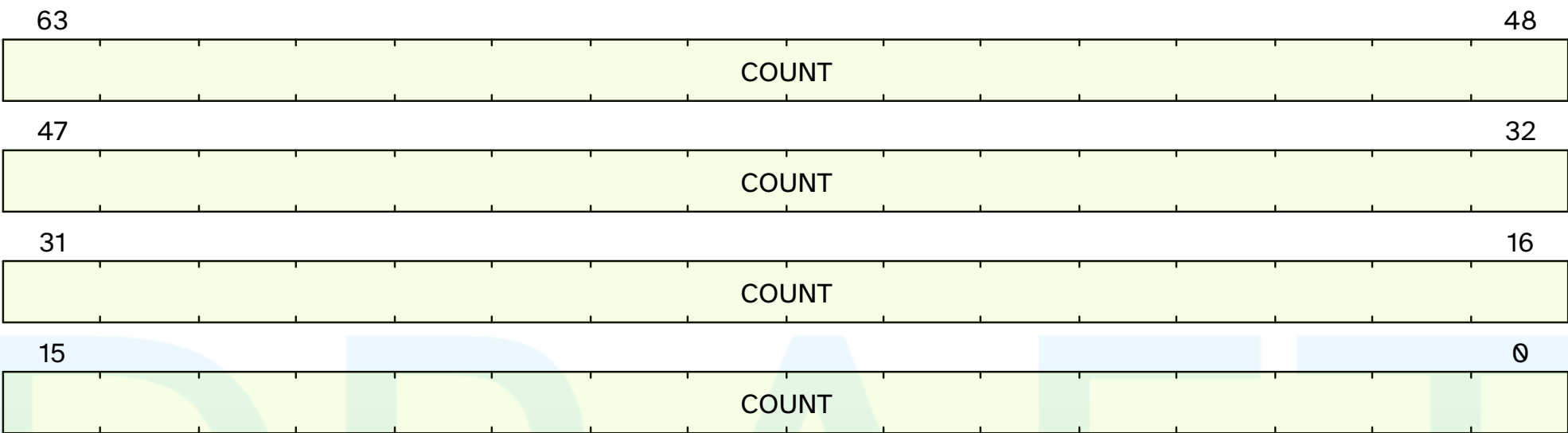


Figure 116. minstret format

D.113.3. Field Summary

Name	Location	Type	Reset Value
<i>minstret</i> .COUNT	63:0	RW-H	UNDEFINED_LEGAL

D.113.4. Fields

minstret.COUNT Field

Location:

63:0

Description:

Instructions retired counter.

<%- if ext?(:Zicntr) -%>
Aliased as *instret*.COUNT.
<%- end -%>

Increments every time an instruction retires unless:

- mcounthinhibit*.IR <%- if ext?(:Smcdeleg) -%>or its alias *scountinhibit*.IR<%- end -%> is set <%- if ext?(:Smcntrpmf) -%>
- minstretcfg*.MINH is set and the current privilege level is M <%- if ext?(:S) -%>
- minstretcfg*.SINH <%- if ext?(:Sccfg) -%>or its alias *instretcfg*.SINH<%- end -%> is set and the current privilege level is (H)S <%- end -%> <%- if ext?(:U) -%>
- minstretcfg*.UINH <%- if ext?(:Sccfg) -%>or its alias *instretcfg*.SINH<%- end -%> is set and the current privilege level is U <%- end -%> <%- if ext?(:H) -%>

- minstretcfg.VSINH <%- if ext?(:Sscfg) -%>or its alias instretcfg.SINH<%- end -%> is set and the current privilege level is VS
- minstretcfg.VUINH <%- if ext?(:Sscfg) -%>or its alias instretcfg.SINH<%- end -%> is set and the current privilege level is VU
<%- end -%>
<%- end -%>

An instruction that causes an exception, notably including MRET/SRET, does not retire and does not cause [minstret.COUNT](#) to increment.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

DRAFT

D.114. minstreth

Machine Instructions Retired Counter

Upper half of 64-bit instructions retired counters.

See [minstret](#) for details.

D.114.1. Attributes

CSR Address	0xb82
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	M

D.114.2. Format

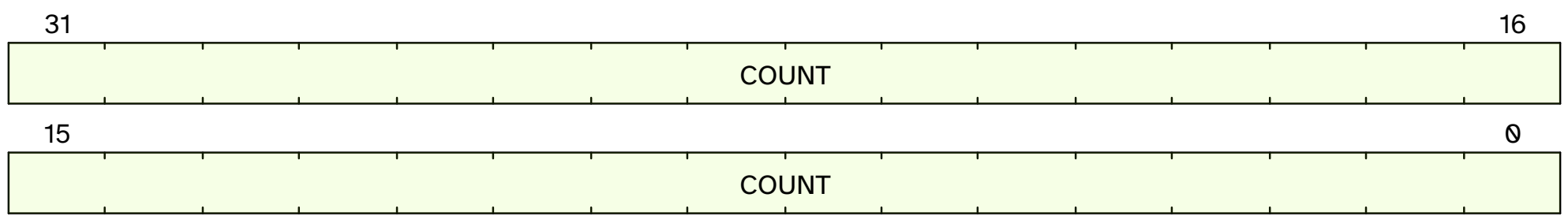


Figure 117. minstreth format

D.114.3. Field Summary

Name	Location	Type	Reset Value
minstreth.COUNT	31:0	RW-H	UNDEFINED_LEGAL

D.114.4. Fields

[minstreth.COUNT](#) Field

Location:

31:0

Description:

Instructions retired counter

Upper half of [minstret](#).

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

D.114.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through [csrrw](#)), the following determines the written value:

```
COUNT = CSR[mcycle].COUNT = {csr_value.COUNT[31:0], CSR[minstret].COUNT[31:0]};
return csr_value.COUNT;
```

D.114.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return minstret.COUNT[63:32];
```

DRAFT

D.115. mtinst

Machine Trap Instruction Register

When a trap is taken into M-mode, mtinst is written with a value that, if nonzero, provides information about the instruction that trapped, to assist software in handling the trap. The values that may be written to mtinst on a trap are documented in TODO.

mtinst is a WARL register that need only be able to hold the values that the implementation may automatically write to it on a trap.

D.115.1. Attributes

CSR Address	0x34a
Defining extension	H, version >= H@1.0.0
Length	64-bit
Privilege Mode	M

D.115.2. Format

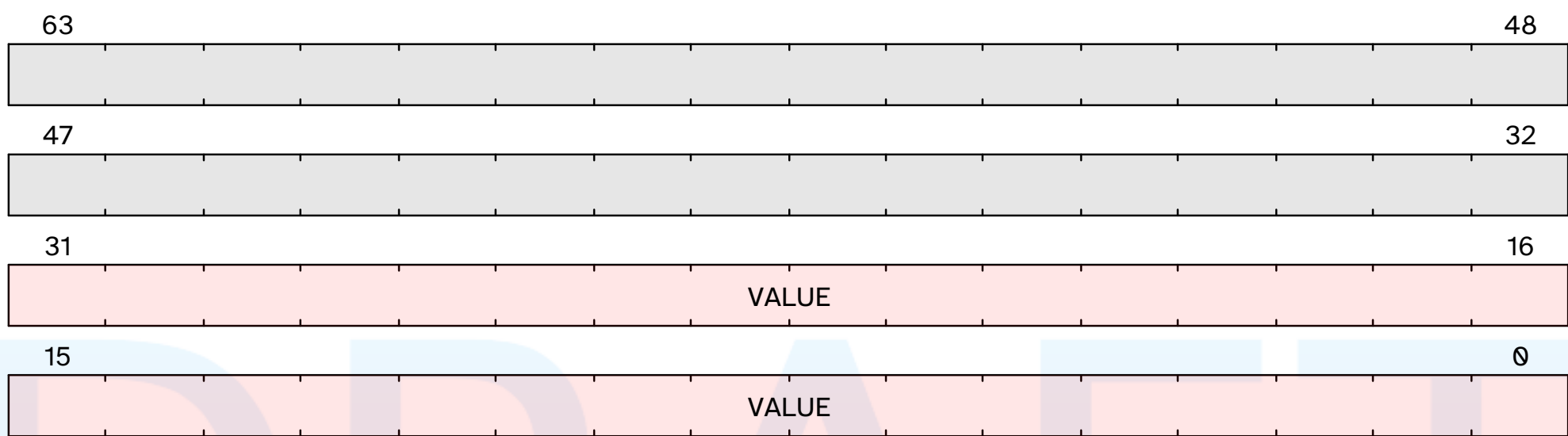


Figure 118. mtinst format

D.115.3. Field Summary

Name	Location	Type	Reset Value
mtinst.VALUE	63:0	RW-H	UNDEFINED_LEGAL
		RO	

D.115.4. Fields

mtinst.VALUE Field

Location:

63:0

Description:

Exception-specific information for a trap into M-mode.

Type:

RW-H

RO

Reset value:

UNDEFINED_LEGAL

D.116. mtval2

Machine Second Trap Value Register

When a trap is taken into M-mode from a virtual mode, mtval2 is written with additional exception-specific information, alongside mtval, to assist software in handling the trap.

When a guest-page-fault trap is taken into M-mode, mtval2 is written with either zero or the guest physical address that faulted, shifted right by 2 bits. For other traps, mtval2 is set to zero, but a future standard or extension may redefine mtval2’s setting for other traps.

If a guest-page fault is due to an implicit memory access during first-stage (VS-stage) address translation, a guest physical address written to mtval2 is that of the implicit memory access that faulted. Additional information is provided in CSR mtinst to disambiguate such situations.

Otherwise, for misaligned loads and stores that cause guest-page faults, a nonzero guest physical address in mtval2 corresponds to the faulting portion of the access as indicated by the virtual address in mtval. For instruction guest-page faults on systems with variable-length instructions, a nonzero mtval2 corresponds to the faulting portion of the instruction as indicated by the virtual address in mtval.

mtval2 is a WARL register that must be able to hold zero and may be capable of holding only an arbitrary subset of other 2-bit-shifted guest physical addresses, if any.

D.116.1. Attributes

CSR Address	0x34b
Defining extension	H, version >= H@1.0.0
Length	64-bit
Privilege Mode	M

D.116.2. Format

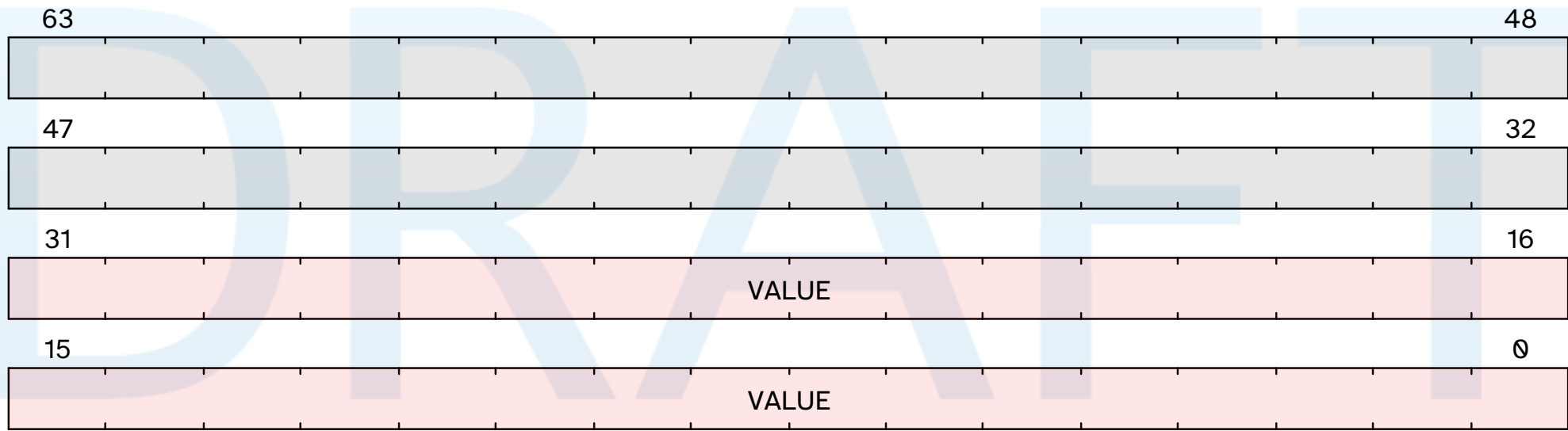


Figure 119. mtval2 format

D.116.3. Field Summary

Name	Location	Type	Reset Value
mtval 2.VAL UE	63:0	RW-H RO	UNDEFINED_LEGAL

D.116.4. Fields

mtval2.VALUE Field

Location:

63:0

Description:

Exception-specific information for a trap into M-mode.

Type:

RW-H

RO

Reset value:

UNDEFINED_LEGAL

D.117. 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.117.1. Attributes

CSR Address	0x180
Defining extension	S, version >= S@1.11.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.117.2. Format

This CSR format changes dynamically with XLEN.

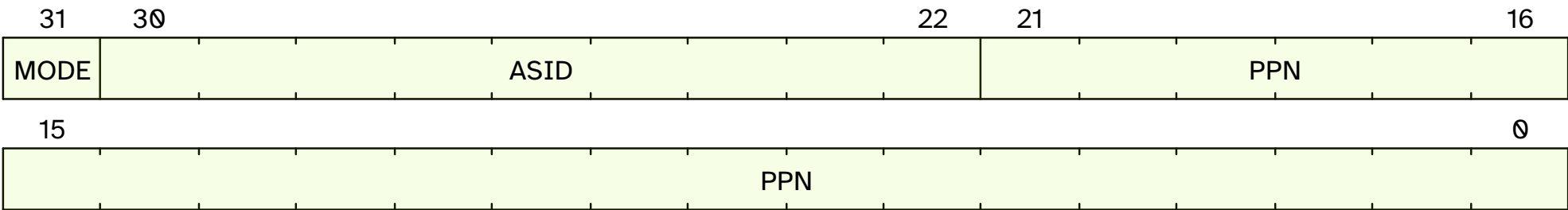


Figure 120. satp Format when CSR[mstatus].SXL == 0

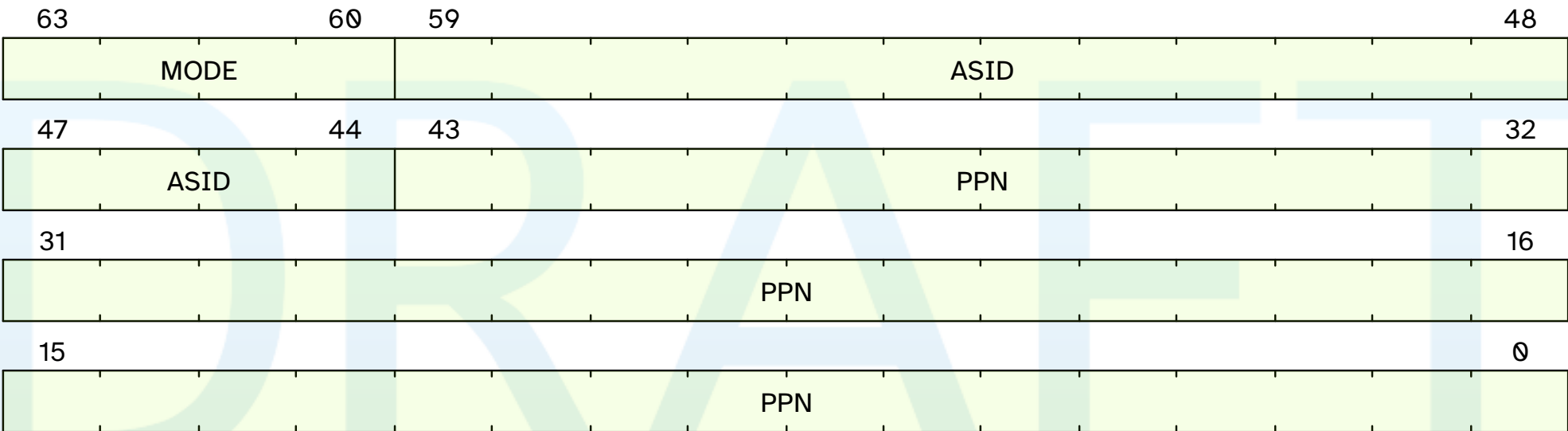


Figure 121. satp Format when CSR[mstatus].SXL == 1

D.117.3. Field Summary

Name	Location	Type	Reset Value
satp.MODE	31 63:60	RW-R	UNDEFINED_LEGAL
satp.ASID	30:22 59:44	RW-R	UNDEFINED_LEGAL
satp.PPN	21:0 43:0	RW-R	UNDEFINED_LEGAL

D.117.4. Fields

satp.MODE Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63:60 when CSR[mstatus].SXL == 1

Description:

Translation Mode

Controls the current translation mode according to the table below.

[separator="!",%autowidth]
!===
! Value ! Name ! Description

! 0 ! Bare a! No translation → virtual address == physical address
<%- if ext?:(Sv39) -%>
! 8 ! Sv39 ! 39-bit virtual address translation
<%- end -%>
<%- if ext?:(Sv48) -%>
! 9 ! Sv48 ! 48-bit virtual address translation
<%- end -%>
<%- if ext?:(Sv57) -%>
! 10 ! Sv57 ! 57-bit virtual address translation
<%- end -%>
!===

Any other value shall be ignored on a write.

Type:
RW-R

Reset value:
UNDEFINED_LEGAL

satp.ASID Field

Location:

- 30:22 when CSR[mstatus].SXL == 0
- 59:44 when CSR[mstatus].SXL == 1

Description:
Address Space ID

Type:
RW-R

Reset value:
UNDEFINED_LEGAL

satp.PPN Field

Location:

- 21:0 when CSR[mstatus].SXL == 0
- 43:0 when CSR[mstatus].SXL == 1

Description:
Physical Page Number

The physical address of the active root page table is PPN << 12.

Can only hold values that correspond to a valid page table base, which will be implementation-dependent.

Type:
RW-R

Reset value:
UNDEFINED_LEGAL

D.117.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through csrrw), the following determines the written value:

MODE = if (SATP_MODE_BARE) {
 if (csr_value.MODE == 0) {
 # In Bare, ASID and PPN must be zero, else the entire write is ignored
 if (csr_value.ASID == 0 && csr_value.PPN == 0) {

```

    if (CSR[satp].MODE != 0) {
        # changes *to* Bare mode take effect immediately without needing sfence.vma
        # thus, an implicit sfence.vma occurs now
        VmaOrderType order_type;
        order_type.global = true;
        order_pgtbl_writes_before_vmafence(order_type);

        invalidate_translations(order_type);

        order_pgtbl_reads_after_vmafence(order_type);
    }
    return csr_value.MODE;
} else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
}
}
}
else if (implemented?(ExtensionName::Sv39) && csr_value.MODE == 8) {
    if (CSR[satp].MODE == 0) {
        # changes *from* Bare mode take effect immediately without needing sfence.vma
        # thus, an implicit sfence.vma occurs now
        VmaOrderType order_type;
        order_type.global = true;
        order_pgtbl_writes_before_vmafence(order_type);

        invalidate_translations(order_type);

        order_pgtbl_reads_after_vmafence(order_type);
    }
    return csr_value.MODE;
}
else if (implemented?(ExtensionName::Sv48) && csr_value.MODE == 9) {
    if (CSR[satp].MODE == 0) {
        # changes *from* Bare mode take effect immediately without needing sfence.vma
        # thus, an implicit sfence.vma occurs now
        VmaOrderType order_type;
        order_type.global = true;
        order_pgtbl_writes_before_vmafence(order_type);

        invalidate_translations(order_type);

        order_pgtbl_reads_after_vmafence(order_type);
    }

    return csr_value.MODE;
}
else if (implemented?(ExtensionName::Sv57) && csr_value.MODE == 10) {
    if (CSR[satp].MODE == 0) {
        # changes *from* Bare mode take effect immediately without needing sfence.vma
        # thus, an implicit sfence.vma occurs now
        VmaOrderType order_type;
        order_type.global = true;
        order_pgtbl_writes_before_vmafence(order_type);

        invalidate_translations(order_type);

        order_pgtbl_reads_after_vmafence(order_type);
    }

    return csr_value.MODE;
}
else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
}

ASID = if (csr_value.MODE == 0) {
    # when MODE == Bare, PPN and ASID must be zero
    if (csr_value.ASID == 0 && csr_value.PPN == 0) {
        return csr_value.ASID;
    } else {
        return UNDEFINED_LEGAL_DETERMINISTIC;
    }
} else {
    XReg shamt = (xlen() == 32 || (CSR[mstatus].SXL == $bits(XRegWidth::XLEN32))) ? 9 : 16;
    XReg all_ones = ((MXLEN'1 << shamt) - 1);

```

```

XReg largest_allowed_asid = (MXLEN'1 << shamt) - 1;

if (csr_value.ASID == all_ones) {
    # the specification states that if all 1's are written to the ASID field, then
    # you must return the largest asid
    return largest_allowed_asid;
} else if (csr_value.ASID > largest_allowed_asid) {
    # ... but is silent on what happens on any other illegal value
    return UNDEFINED_LEGAL_DETERMINISTIC;
} else {
    # unrestricted
    return csr_value.ASID;
}
}

PPN = if (csr_value.MODE == 0) {
    # when MODE == Bare, PPN and ASID must be zero
    if (csr_value.ASID == 0 && csr_value.PPN == 0) {
        return csr_value.PPN;
    } else {
        return UNDEFINED_LEGAL_DETERMINISTIC;
    }
} else {
    # unrestricted
    return csr_value.PPN;
}
}

```

DRAFT

D.118. scause

Supervisor Cause

Reports the cause of the latest exception.

D.118.1. Attributes

CSR Address	0x142
Defining extension	S, version >= S@1.11.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.118.2. Format

This CSR format changes dynamically with XLEN.

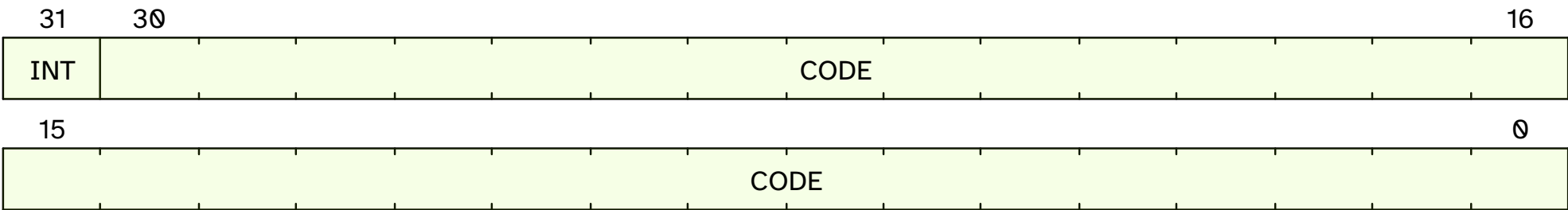


Figure 122. scause Format when CSR[mstatus].SXL == 0

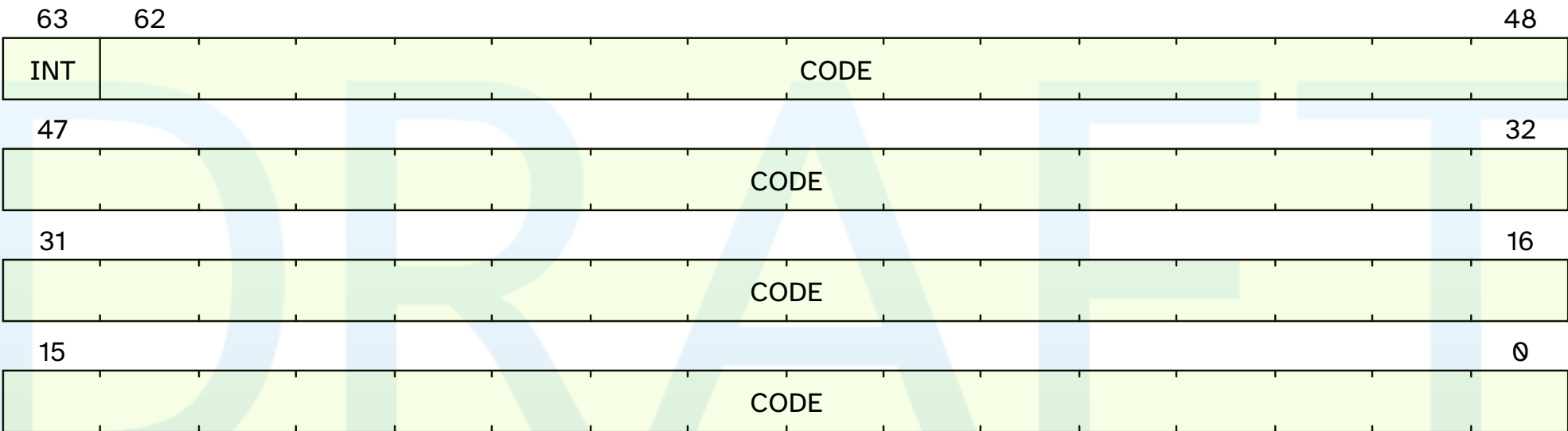


Figure 123. scause Format when CSR[mstatus].SXL == 1

D.118.3. Field Summary

Name	Location	Type	Reset Value
scause.INT	31	RW-RH	UNDEFINED_LEGAL
	63		
scause.CODE	30:0	RW-RH	UNDEFINED_LEGAL
	62:0		

D.118.4. Fields

[scause.INT](#) Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63 when CSR[mstatus].SXL == 1

Description:

Written by hardware when a trap is taken into S-mode.

When set, the last exception was caused by an asynchronous Interrupt.

[scause.INT](#) is writable.

[when,"TRAP_ON_ILLEGAL_WLRL == true"]

If [scause](#) is written with an undefined cause (combination of [scause.INT](#) and [scause.CODE](#)), an Illegal Instruction exception occurs.

[when,"TRAP_ON_ILLEGAL_WLRL == false"]
If *scause* is written with an undefined cause (combination of *scause.INT* and *scause.CODE*), neither *scause.INT* nor *scause.CODE* are modified.

Type:
RW-RH

Reset value:
UNDEFINED_LEGAL

scause.CODE Field

Location:
• 30:0 when CSR[mstatus].SXL == 0
• 62:0 when CSR[mstatus].SXL == 1

Description:
Written by hardware when a trap is taken into S-mode.

Holds the interrupt or exception code for the last taken trap.

scause.CODE is writable.

[when,"TRAP_ON_ILLEGAL_WLRL == true"]
If *scause* is written with an undefined cause (combination of *scause.INT* and *scause.CODE*), an Illegal Instruction exception occurs.

[when,"TRAP_ON_ILLEGAL_WLRL == false"]
If *scause* is written with an undefined cause (combination of *scause.INT* and *scause.CODE*), neither *scause.INT* nor *scause.CODE* are modified.

```
Valid interrupt codes are:  
[separator="!"]  
!===  
<%- interrupt_codes.sort_by { |code| code.num }.each do |code| -%>  
! <%= code.num %> ! <%= code.name %>  
<%- end -%>  
!===  
  
Valid exception codes are:  
[separator="!"]  
!===  
<%- exception_codes.sort_by { |code| code.num }.each do |code| -%>  
! <%= code.num %> ! <%= code.name %>  
<%- end -%>  
!===
```

Type:
RW-RH

Reset value:
UNDEFINED_LEGAL

D.118.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (e.g., through *csrrw*), the following determines the written value:

```
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;
}
```

DRAFT

D.119. scounteren

Supervisor Counter Enable

Delegates control of the hardware performance-monitoring counters to U-mode

D.119.1. Attributes

CSR Address	0x106
Defining extension	S, version >= S@1.11.0
Length	32-bit
Privilege Mode	S

D.119.2. Format

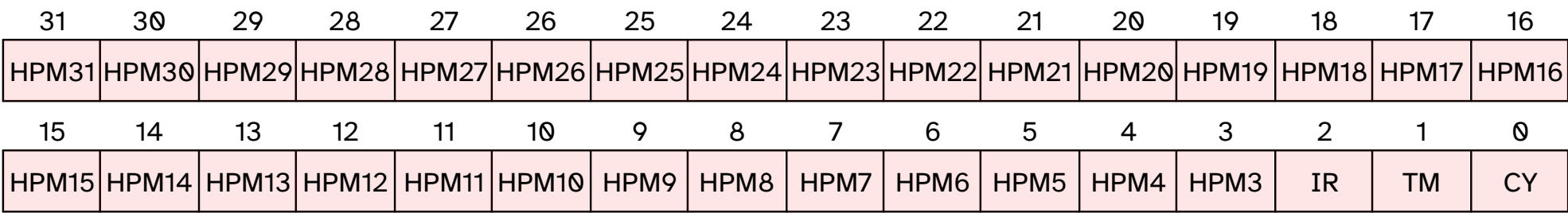


Figure 124. scounteren format

D.119.3. Field Summary

Name	Location	Type	Reset Value
scounteren.CY	0	RW	UNDEFINED_LEGAL
		RO	
scounteren.TM	1	RW	UNDEFINED_LEGAL
		RO	
scounteren.IR	2	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM3	3	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM4	4	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM5	5	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM6	6	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM7	7	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM8	8	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM9	9	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM10	10	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM11	11	RW	UNDEFINED_LEGAL
		RO	
scounteren.HPM12	12	RW	UNDEFINED_LEGAL
		RO	

Name	Location	Type	Reset Value
scouteren.HPM13	13	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM14	14	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM15	15	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM16	16	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM17	17	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM18	18	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM19	19	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM20	20	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM21	21	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM22	22	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM23	23	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM24	24	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM25	25	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM26	26	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM27	27	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM28	28	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM29	29	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM30	30	RW	UNDEFINED_LEGAL
		RO	
scouteren.HPM31	31	RW	UNDEFINED_LEGAL
		RO	

D.119.4. Fields

scouteren.CY Field

Location:

0

Description:

When both *scouteren.CY* and *mcouteren.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

scouteren.TM Field

Location:

1

Description:

When both [scouteren.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

scouteren.IR Field

Location:

2

Description:

When both [scouteren.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

scouteren.HPM3 Field

Location:

3

Description:

When both [scouteren.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

scouteren.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

scouteren.HPM12 Field

Location:

12

Description:

When both [scouteren.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

scouteren.HPM13 Field

Location:

13

Description:

When both [scouteren.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

scouteren.HPM14 Field

Location:

14

Description:

When both [scouteren.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

scouteren.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

scouteren.HPM19 Field

Location:

19

Description:

When both [scouteren.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

scouteren.HPM20 Field

Location:

20

Description:

When both [scouteren.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

scouteren.HPM21 Field

Location:

21

Description:

When both [scouteren.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

scouteren.HPM22 Field

Location:

22

Description:

When both [scouteren.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.120. senvcfg

Supervisor Environment Configuration

Contains fields that control certain characteristics of the U-mode execution environment.

D.120.1. Attributes

CSR Address	0x10a
Defining extension	- allOf: - S, version >=1.12 - U, version >= U@1.0.0
Length	64-bit
Privilege Mode	S

D.120.2. Format

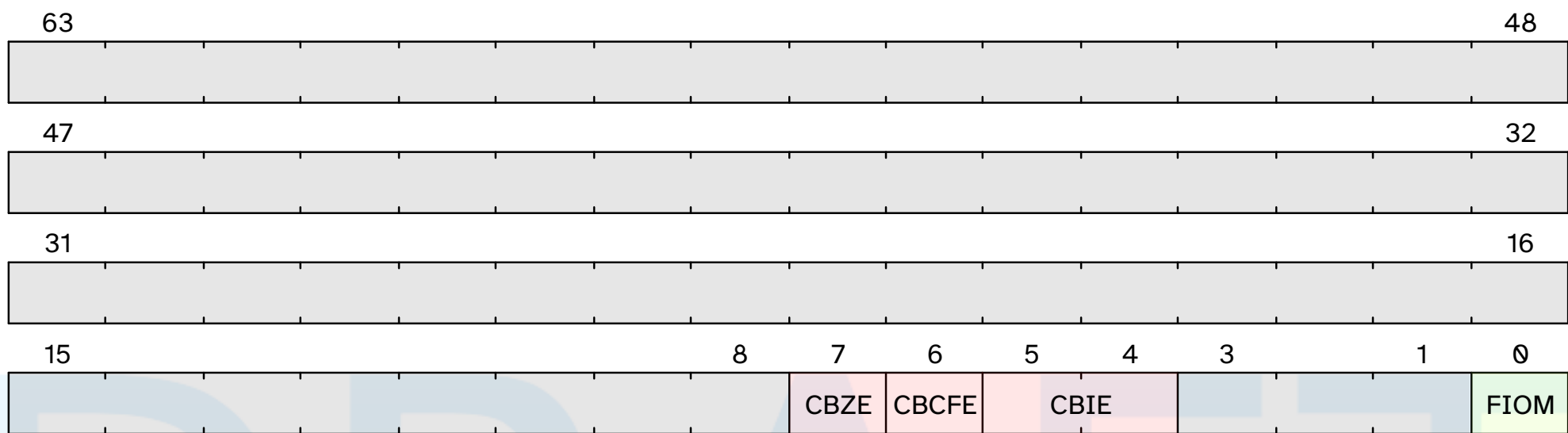


Figure 125. senvcfg format

D.120.3. Field Summary

Name	Location	Type	Reset Value
senvcfg.CBZE	7	RW	UNDEFINED_LEGAL
senvcfg.CBCFE	6	RW	UNDEFINED_LEGAL
senvcfg.CBIE	5:4	RW-R	UNDEFINED_LEGAL
senvcfg.FIOM	0	RW	0

D.120.4. Fields

[senvcfg.CBZE](#) Field

Location:

7

Description:

Cache Block Zero instruction Enable

Bit is read-only 0 when [menvcfg.CBZE](#) is clear.

Enables the execution of the cache block zero instruction, [cbo.zero](#), in U-mode<% if ext?(:H) %> and (in conjunction with [henvcfg.CBZE](#)) VU-mode<% end %>.

- 0: The instruction raises an illegal instruction<% if ext?(:H) %> or virtual instruction exception<% end %>
- 1: The instruction is executed

To summarize access:
[separator="!",%autowidth]
!===
! [menvcfg.CBZE](#) ! [senvcfg.CBZE](#) behavior

! 0 ! read-only 0
! 1
a! writable, independent bit from [menvcfg.CBZE](#)
!===

See [cbo.zero](#) for a summary of the effect.

Type:

RW

Reset value:

UNDEFINED_LEGAL

[senvcfg.CBCFE](#) Field

Location:

6

Description:

Cache Block Clean and Flush instruction Enable

Enables the execution of the cache block clean instruction, [cbo.clean](#), and the cache block flush instruction, [cbo.flush](#),
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- 0: The instruction raises an illegal instruction <% if ext?(:H) %>or virtual instruction exception<% end %>
- 1: The instruction is executed

To summarize access:
[separator="!",%autowidth]
!===
! [menvcfg.CBCFE](#) ! [senvcfg.CBCFE](#) behavior

! 0 ! read-only 0
! 1
a! writable, independent bit from [menvcfg.CBCFE](#)
!===

See [cbo.clean](#) and/or [cbo.flush](#) for a summary of the effect.

Type:

RW

Reset value:

UNDEFINED_LEGAL

[senvcfg.CBIE](#) Field

Location:

5:4

Description:

Cache Block Invalidate instruction Enable

This field has restricted values based on the value of [menvcfg.CBIE](#).
When an invalid value is written, it is ignored and the field remains unchanged.

[separator="!",%autowidth,cols=",>"]
!===
! [menvcfg.CBIE](#) ! Valid values of [senvcfg.CBIE](#)


```
! 00 ! 00
! 01 ! 00, 01
! 11 ! 00, 01, 11
!===
```

Controls execution of the cache block invalidate instruction, [cbo.inval](#), in U-mode
<% if ext?(:H) %>
and VU-mode (together with [henvcfg.CBIE](#))
<% end %>

1. + +
- 00: The instruction raises an illegal instruction or virtual instruction exception
 - 01: The instruction is executed and performs a flush operation
 - 10: Reserved
 - 11: The instruction is executed and performs an invalidate operation

See [cbo.inval](#) for more details.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

senvcfg.FIOM Field

Location:

0

Description:

Fence of I/O implies Memory

When either [senvcfg.FIOM](#) or [menvcfg.FIOM](#) is set, FENCE instructions ordering I/O regions also implicitly order memory regions when executed in U-mode as follows:

```
[separator="!",%autowidth,float="center",align="center",cols="^,<",options="header"]
!===
!Instruction bit !Meaning when set
!PI +
PO
!Predecessor device input and memory reads (PR implied) +
Predecessor device output and memory writes (PW implied)
!SI +
SO
!Successor device input and memory reads (SR implied) +
Successor device output and memory writes (SW implied)
!===
```

Similarly, in U-mode when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its aq and/or rl bit set, then that instruction is ordered as though it accesses both device I/O and memory.

See [fence](#) for more details.

Type:

RW

Reset value:

0

D.120.5. 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 [csrsw](#)), the following determines the written value:

CBZE = csr_value.CBZE

```
CBCFE = csr_value.CBCFE
CBIE = if (csr_value.CBIE == 0 || csr_value.CBIE == 1 || csr_value.CBIE == 3) {
    return csr_value.CBIE;
} else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
}

FIOM = csr_value.FIOM
```

DRAFT

D.121. 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.121.1. Attributes

CSR Address	0x141
Defining extension	S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

D.121.2. Format

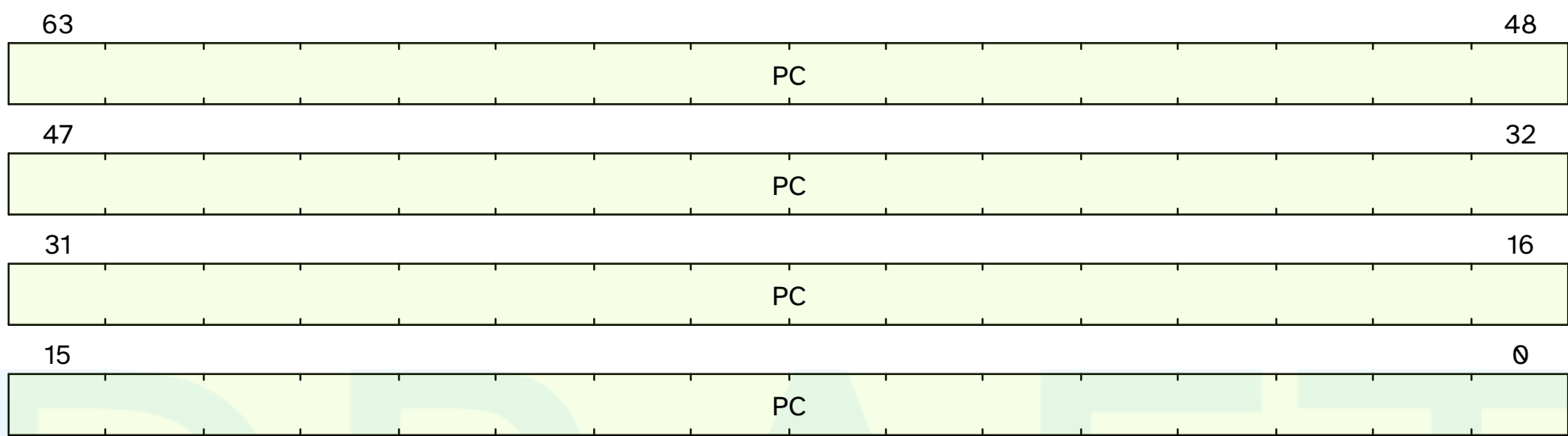


Figure 126. sepc format

D.121.3. Field Summary

Na me	Location	Type	Reset Value
sepc.PC	63:0	RW-RH	UNDEFINED_LEGAL

D.121.4. 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.121.5. 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.121.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (implemented?(ExtensionName::C) && misa.C == 1'b1) {  
  return sepc.PC & ~64'b1;  
} else {  
  return sepc.PC;  
}
```

DRAFT

D.122. 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.122.1. Attributes

CSR Address	0x144
Defining extension	S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

D.122.2. Format

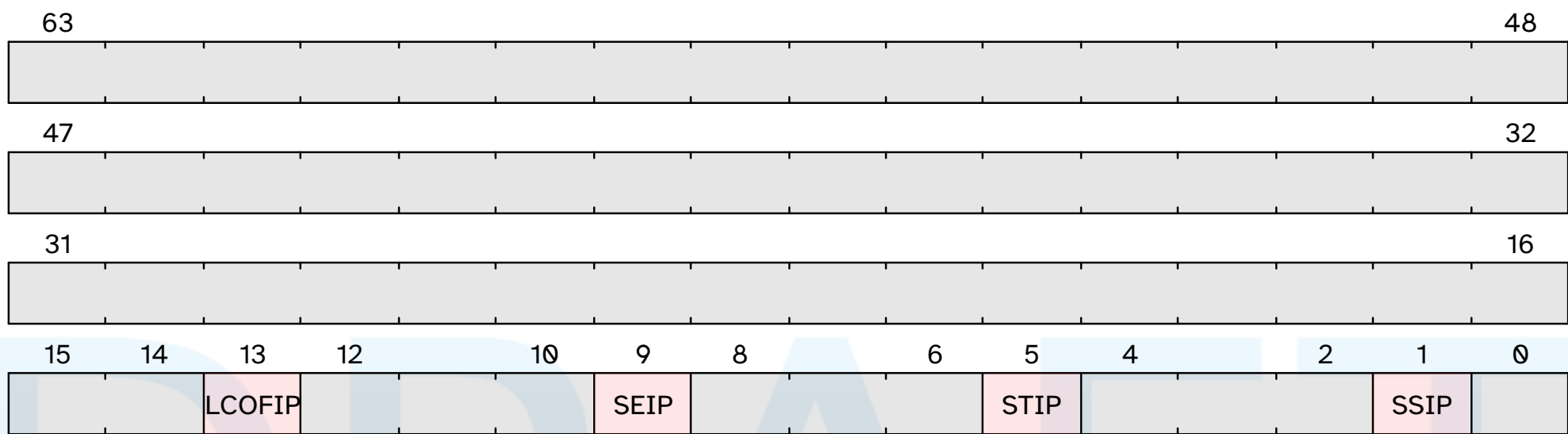


Figure 127. sip format

D.122.3. Field Summary

Nam e	Location	Type	Reset Value
sip.S SIP	1	RW	UNDEFINED_LEGAL
sip.S TIP	5	RO-H	UNDEFINED_LEGAL
sip.S EIP	9	RO-H	UNDEFINED_LEGAL
sip.L COFI P	13	RW-H	UNDEFINED_LEGAL

D.122.4. 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.123. sscratch

Supervisor Scratch Register

Scratch register for software use. Bits are not interpreted by hardware.

D.123.1. Attributes

CSR Address	0x140
Defining extension	S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

D.123.2. Format

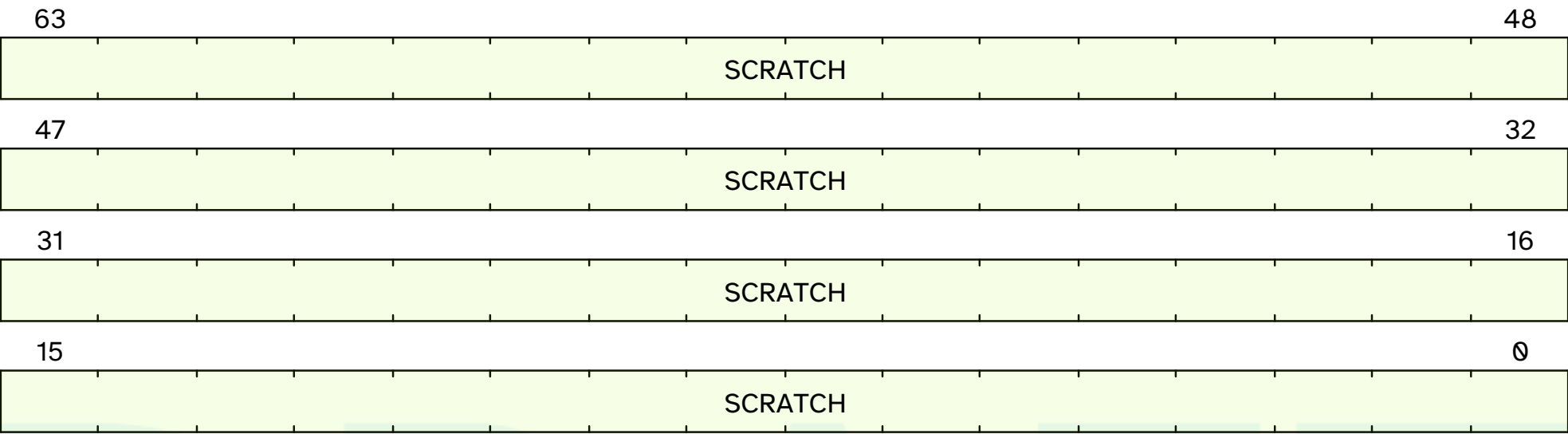


Figure 128. sscratch format

D.123.3. Field Summary

Name	Location	Type	Reset Value
sscratch.SCRATCH	63:0	RW	UNDEFINED_LEGAL

D.123.4. Fields

sscratch.SCRATCH Field

Location:
63:0

Description:
Scratch value

Type:
RW

Reset value:
UNDEFINED_LEGAL

D.124. 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.124.1. Attributes

CSR Address	0x100
Defining extension	S, version >= S@1.11.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.124.2. Format

This CSR format changes dynamically with XLEN.

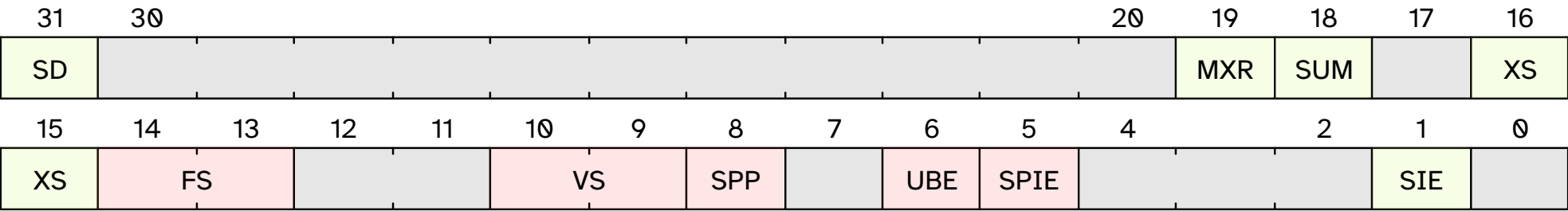


Figure 129. sstatus Format when CSR[mstatus].SXL == 0

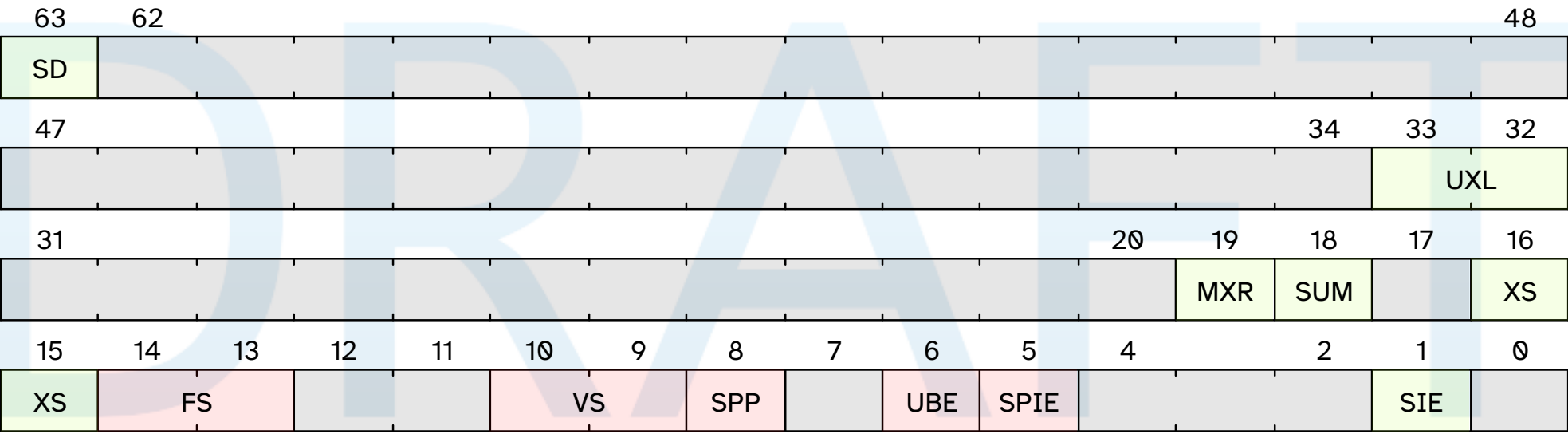


Figure 130. sstatus Format when CSR[mstatus].SXL == 1

D.124.3. Field Summary

Nam e	Location	Type	Reset Value
sstat us.SD	31	RO-H	UNDEFINED_LEGAL
sstat us.UXL	63		
sstat us.UXL	33:32	RO	UNDEFINED_LEGAL
sstat us.MXR	19	RW	UNDEFINED_LEGAL
sstat us.SUM	18	RW	UNDEFINED_LEGAL
sstat us.XS	16:15	RO	UNDEFINED_LEGAL
sstat us.FS	14:13	RW-H	UNDEFINED_LEGAL
sstat us.VS	10:9	RW-H	UNDEFINED_LEGAL
sstat us.SPP	8	RW-H	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
sstatus.UBE	6	RO	UNDEFINED_LEGAL
sstatus.SPIE	5	RW-H	UNDEFINED_LEGAL
sstatus.SIE	1	RW-H	UNDEFINED_LEGAL

D.124.4. Fields

sstatus.SD Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63 when CSR[mstatus].SXL == 1

Description:

State Dirty

Alias of mstatus.SD.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sstatus.UXL Field



UXL is only defined in RV64 (CSR[mstatus].SXL == 1)

Location:

33:32

Description:

U-mode XLEN

Alias of mstatus.UXL.

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.MXR Field

Location:

19

Description:

Make eXecutable Readable

Alias of mstatus.MXR.

Type:

RW

Reset value:

UNDEFINED_LEGAL

sstatus.SUM Field

Location:

18

Description:

permit Supervisor Memory Access

Alias of [mstatus.SUM](#).

Type:

RW

Reset value:

UNDEFINED_LEGAL

sstatus.XS Field

Location:

16:15

Description:

Custom (X) extension context Status.

Alias of [mstatus.XS](#).

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.FS Field

Location:

14:13

Description:

Floating point context status.

Alias of [mstatus.FS](#).

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.VS Field

Location:

10:9

Description:

Vector context status.

Alias of [mstatus.VS](#).

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.SPP Field

Location:
8

Description:
S-mode Previous Privilege

Alias of [mstatus.SPP](#).

Type:
RW-H

Reset value:
UNDEFINED_LEGAL

sstatus.UBE Field

Location:
6

Description:
U-mode Big Endian

Alias of [mstatus.UBE](#).

Type:
RO

Reset value:
UNDEFINED_LEGAL

sstatus.SPIE Field

Location:
5

Description:
S-mode Previous Interrupt Enable

Alias of [mstatus.SPIE](#).

Type:
RW-H

Reset value:
UNDEFINED_LEGAL

sstatus.SIE Field

Location:
1

Description:
S-mode Interrupt Enable

Alias of [mstatus.SIE](#).

Type:
RW-H

Reset value:
UNDEFINED_LEGAL

D.125. stval

Supervisor Trap Value

Holds trap-specific information

D.125.1. Attributes

CSR Address	0x143
Defining extension	S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

D.125.2. Format

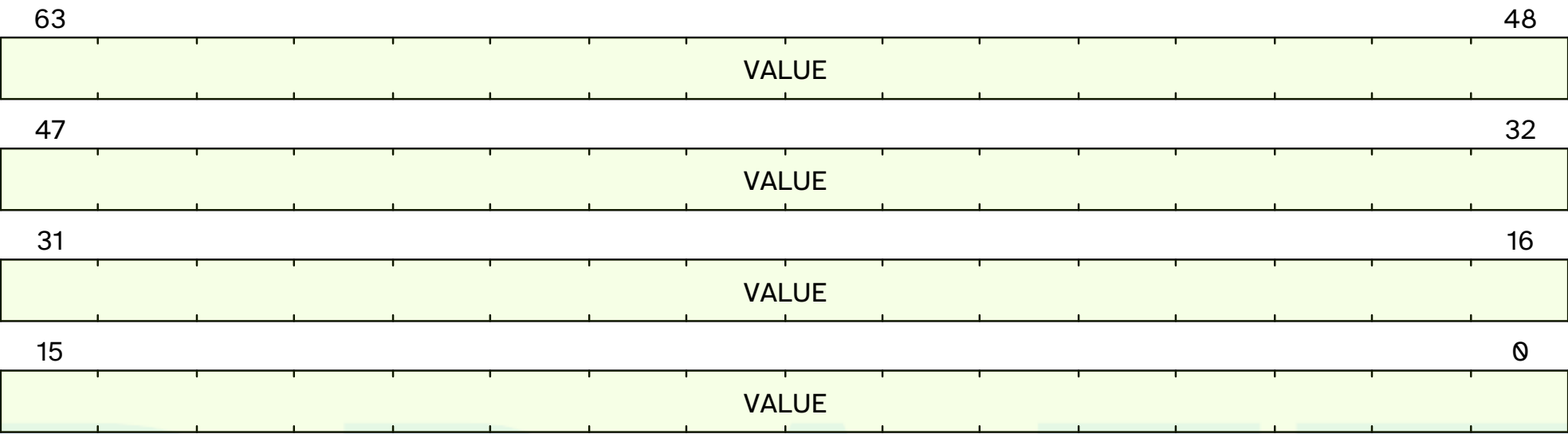


Figure 131. stval format

D.125.3. Field Summary

Nam e	Location	Type	Reset Value
stval. VALU E	63:0	RW-H	0

D.125.4. 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.126. stvec

Supervisor Trap Vector

Controls where traps jump.

D.126.1. Attributes

CSR Address	0x105
Defining extension	S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

D.126.2. Format

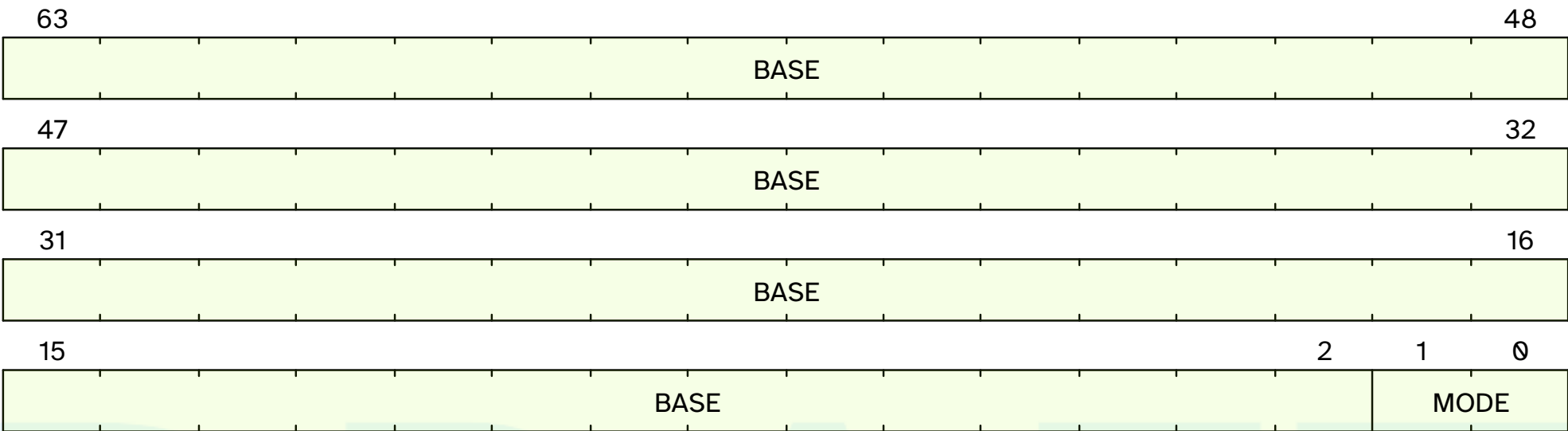


Figure 132. stvec format

D.126.3. Field Summary

Nam e	Location	Type	Reset Value
stvec. BASE	63:2	RW-R	UNDEFINED_LEGAL
stvec. MOD E	1:0	RW-R	0

D.126.4. 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.126.5. 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.127. 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.127.1. Attributes

CSR Address	0xc01
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

D.127.2. Format

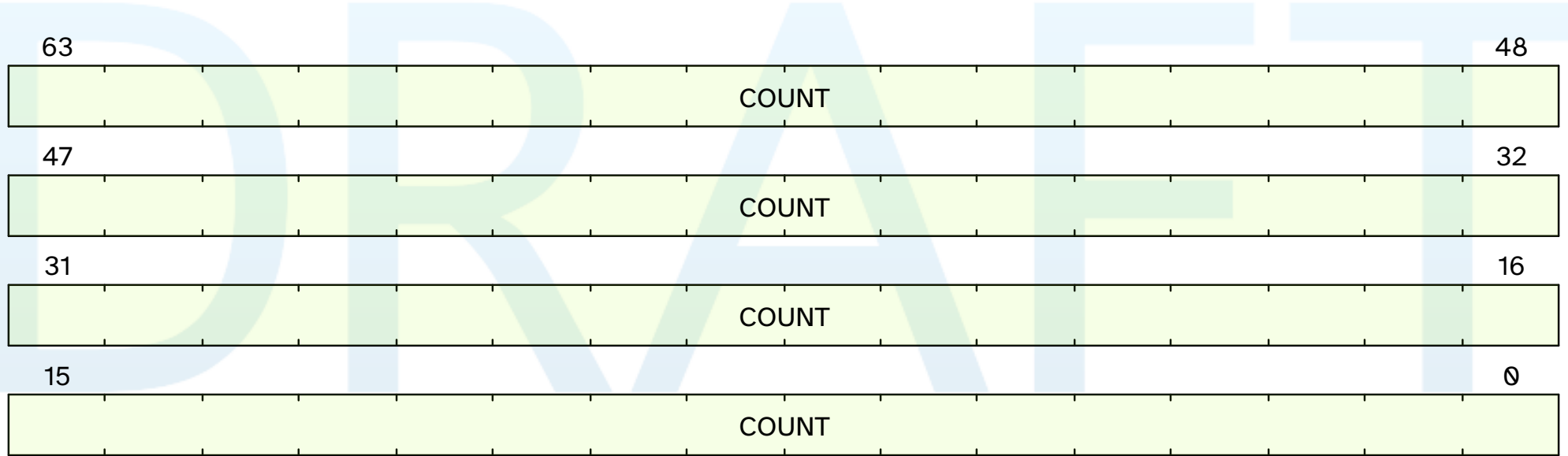


Figure 133. time format

D.127.3. Field Summary

Nam e	Location	Type	Reset Value
time.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.127.4. 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.127.5. Software read**

This CSR may return a value that is different from what is stored in hardware.

```

if (!TIME_CSR_IMPLEMENTED) {
    unimplemented_csr($encoding);
}
if (mode() == PrivilegeMode::S) {
    if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.TM & scounteren.TM) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.TM == 1'b0 && mcounteren.TM == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.TM & scounteren.TM) == 1'b0 && (mcounteren.IR == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
return read_mtime();

```


D.128. timeh

High-half timer for RDTIME Instruction

i

timeh

 is only defined in RV32.

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR mtimeh.

Privilege mode access is controlled with [mcounteren.TM](#), [scounteren.TM](#), and [hcounteren.TM](#) as follows:

mcounteren.TM	scounteren.TM	scounteren.TM	time behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	0	0	read-only	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	1	0	read-only	read-only	Illegal Instruction	Illegal Instruction
1	0	1	read-only	Illegal Instruction	read-only	Illegal Instruction
1	1	1	read-only	read-only	read-only	read-only

D.128.1. Attributes

CSR Address	0xc81
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

D.128.2. Format

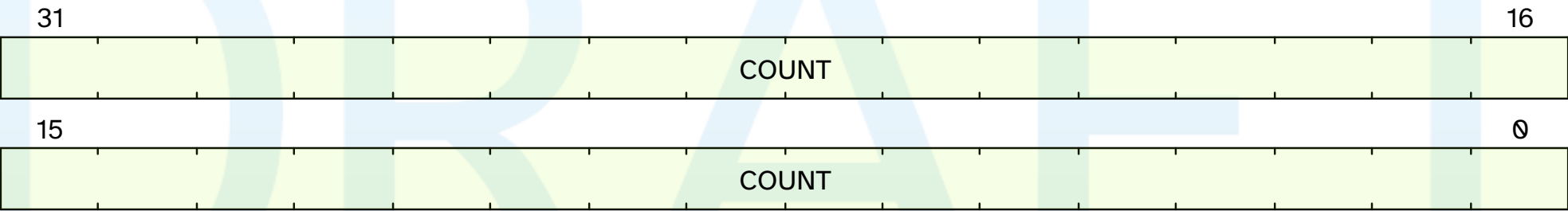


Figure 134. timeh format

D.128.3. Field Summary

Name	Location	Type	Reset Value
timeh.COUNT	31:0	RO-H	UNDEFINED_LEGAL

D.128.4. Fields

timeh.COUNT Field

Location:

31:0

Description:

Reports the most significant 32 bits of the current wall-clock time from the timer device.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.128.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```

if (!TIME_CSR_IMPLEMENTED) {
    unimplemented_csr($encoding);
}
if (mode() == PrivilegeMode::S) {
    if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::U) {
    if (misa.S == 1'b1) {
        if ((mcounteren.TM & scounteren.TM) == 1'b0) {
            raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
        }
    } else if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VS) {
    if (hcounteren.TM == 1'b0 && mcounteren.TM == 1'b1) {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
} else if (mode() == PrivilegeMode::VU) {
    if (hcounteren.TM & scounteren.TM) == 1'b0 && (mcounteren.IR == 1'b1 {
        raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    } else if (mcounteren.TM == 1'b0) {
        raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
}
return read_mtime()[63:32];

```

DRAFT

D.129. vsatp

Virtual Supervisor Address Translation and Protection

The `vsatp` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `satp`. When V=1, `vsatp` substitutes for the usual `satp`, so instructions that normally read or modify `satp` actually access `vsatp` instead. `vsatp` controls VS-stage address translation, the first stage of two-stage translation for guest virtual addresses.

The `vsatp` register is considered active for the purposes of the address-translation algorithm unless the effective privilege mode is U and `hstatus.HU=0`. However, even when `vsatp` is active, VS-stage page-table entries' A bits must not be set as a result of speculative execution, unless the effective privilege mode is VS or VU.



In particular, virtual-machine load/store (hlv, 'hlvx', or 'hsv') instructions that are misspeculatively executed must not cause VS-stage A bits to be set.

When V=0, a write to `vsatp` with an unsupported MODE value is either ignored as it is for `satp`, or the fields of `vsatp` are treated as WARL in the normal way. However, when V=1, a write to `satp` with an unsupported MODE value is ignored and no write to `vsatp` is effected.

D.129.1. Attributes

CSR Address	0x280
Virtual CSR Address	0x180
Defining extension	• H, version >= H@1.0.0
Length	32 when CSR[hstatus].VSXL == 0 64 when CSR[hstatus].VSXL == 1
Privilege Mode	VS

D.129.2. Format

This CSR format changes dynamically with XLEN.

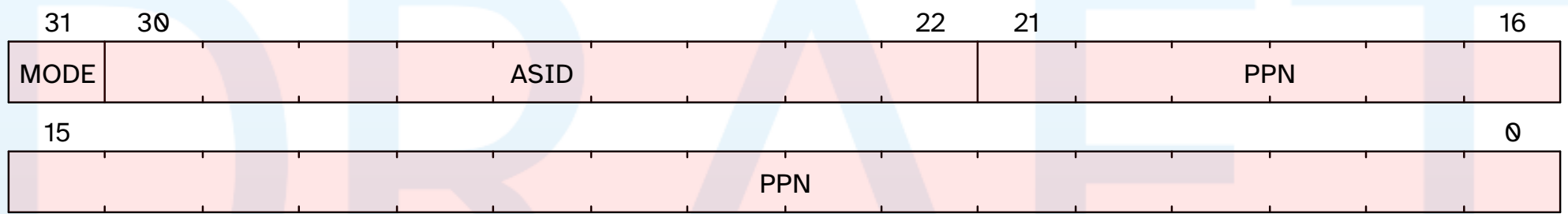


Figure 135. vsatp Format when CSR[hstatus].VSXL == 0

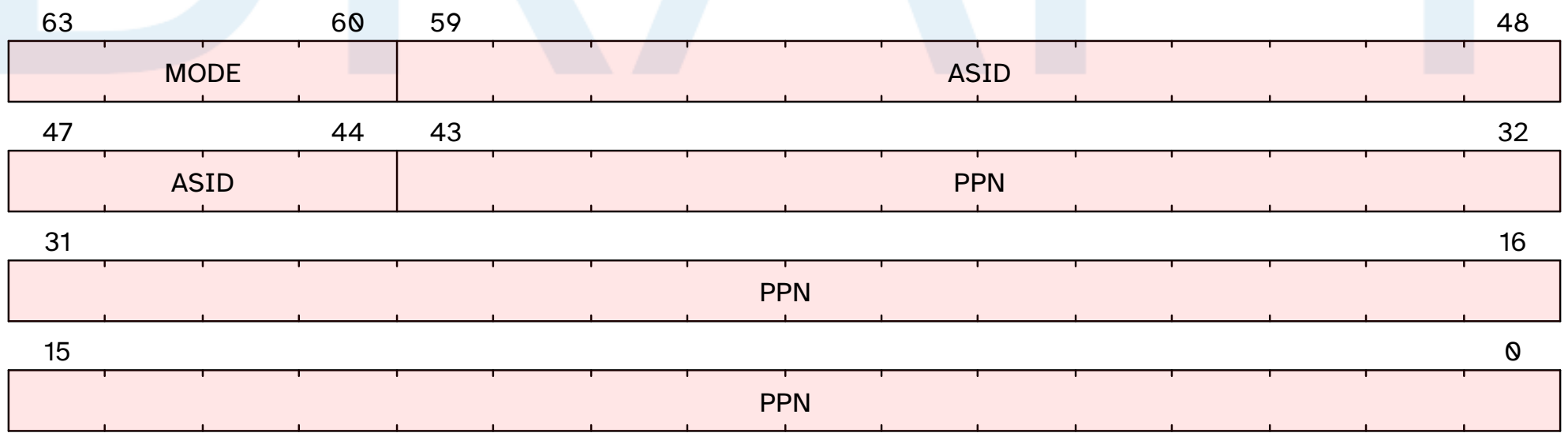


Figure 136. vsatp Format when CSR[hstatus].VSXL == 1

D.129.3. Field Summary

Nam e	Location	Type	Reset Value
<code>vsatp.MO D E</code>	31 63:60	RW-R	UNDEFINED_LEGAL
<code>vsatp.AS I D</code>	30:22 59:44	RW-R	UNDEFINED_LEGAL
<code>vsatp.P P N</code>	21:0 43:0	RW-R	UNDEFINED_LEGAL

D.129.4. Fields

vsatp.MODE Field

Location:

- 31 when CSR[hstatus].VSXL == 0
- 63:60 when CSR[hstatus].VSXL == 1

Description:

Translation Mode

Controls the current translation mode in VS-mode according to the table below.

[separator="!",%autowidth]

!===

! Value ! Name ! Description

! 0 ! Bare a! No translation → virtual address == physical address

<%- if ext?(:Sv39) -%>

! 8 ! Sv39 ! 39-bit virtual address translation

<%- end -%>

<%- if ext?(:Sv48) -%>

! 9 ! Sv48 ! 48-bit virtual address translation

<%- end -%>

<%- if ext?(:Sv57) -%>

! 10 ! Sv57 ! 57-bit virtual address translation

<%- end -%>

!===

Any other value shall be ignored on a write.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

vsatp.ASID Field

Location:

- 30:22 when CSR[hstatus].VSXL == 0
- 59:44 when CSR[hstatus].VSXL == 1

Description:

Address Space ID

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

vsatp.PPN Field

Location:

- 21:0 when CSR[hstatus].VSXL == 0
- 43:0 when CSR[hstatus].VSXL == 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.129.5. 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 (csr_value.MODE == 0) {
  if (virtual_mode?() || IGNORE_INVALID_VSATP_MODE_WRITES_WHEN_V_EQ_ZERO) {
    # In Bare, ASID and PPN must be zero, else the entire write is ignored
    if (csr_value.ASID == 0 && csr_value.PPN == 0) {
      return csr_value.MODE;
    } else {
      return UNDEFINED_LEGAL_DETERMINISTIC;
    }
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}
else if (implemented?(ExtensionName::Sv39) && csr_value.MODE == 8) { return csr_value.MODE; }
else if (implemented?(ExtensionName::Sv48) && csr_value.MODE == 9) { return csr_value.MODE; }
else if (implemented?(ExtensionName::Sv57) && csr_value.MODE == 10) { return csr_value.MODE; }
else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}

ASID = if (csr_value.MODE == 0) {
  if (virtual_mode?() || IGNORE_INVALID_VSATP_MODE_WRITES_WHEN_V_EQ_ZERO) {
    # 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 {
    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) {
  if (virtual_mode?() || IGNORE_INVALID_VSATP_MODE_WRITES_WHEN_V_EQ_ZERO) {
    # 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 {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else {
  # unrestricted
  return csr_value.PPN;
}

```

D.130. vscause

Virtual Supervisor Cause

Reports the cause of the latest exception taken in VS-mode.

D.130.1. Attributes

CSR Address	0x242
Virtual CSR Address	0x142
Defining extension	H, version >= H@1.0.0
Length	32 when CSR[hstatus].VSXL == 0 64 when CSR[hstatus].VSXL == 1
Privilege Mode	VS

D.130.2. Format

This CSR format changes dynamically with XLEN.

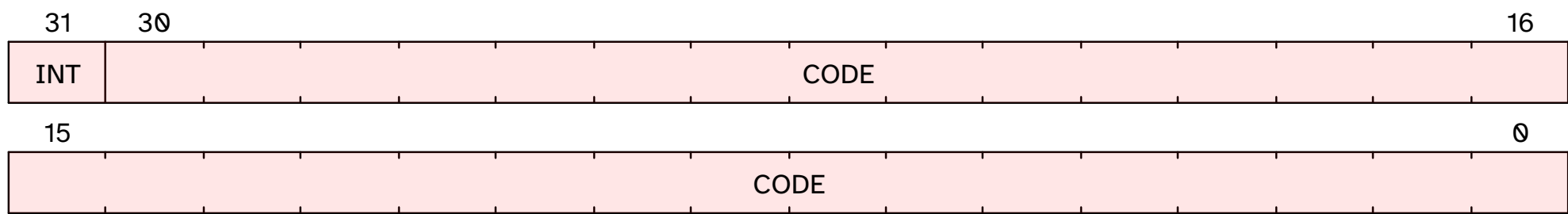


Figure 137. vscause Format when CSR[hstatus].VSXL == 0

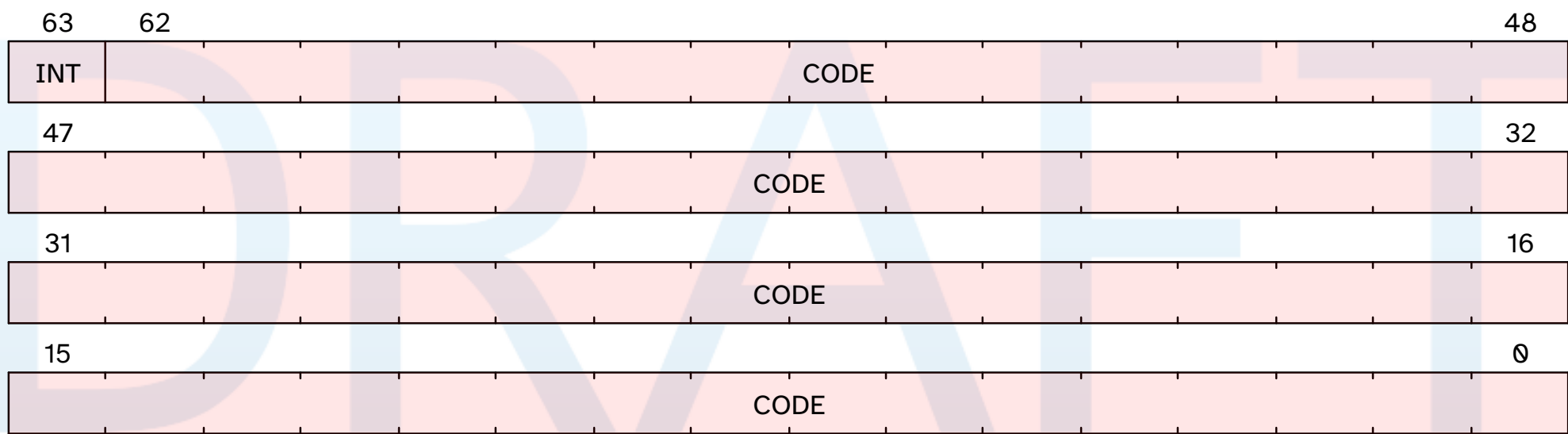


Figure 138. vscause Format when CSR[hstatus].VSXL == 1

D.130.3. Field Summary

Name	Location	Type	Reset Value
vscause.INT	31	RW-RH	UNDEFINED_LEGAL
	63		
vscause.CODE	30:0	RW-RH	UNDEFINED_LEGAL
	62:0		

D.130.4. Fields

vscause.INT Field

Location:

- 31 when CSR[hstatus].VSXL == 0
- 63 when CSR[hstatus].VSXL == 1

Description:

Written by hardware when a trap is taken into VS-mode.

When set, the last exception was caused by an asynchronous Interrupt.

vscause.INT is writable.

[when,"TRAP_ON_ILLEGAL_WLRL == true"]
If **vscause** is written with an undefined cause (combination of **vscause.INT** and **vscause.CODE**), an Illegal Instruction exception occurs.

[when,"TRAP_ON_ILLEGAL_WLRL == false"]
If **vscause** is written with an undefined cause (combination of **vscause.INT** and **vscause.CODE**), neither **vscause.INT** nor **vscause.CODE** are modified.

Type:
RW-RH

Reset value:
UNDEFINED_LEGAL

vscause.CODE Field

Location:

- 30:0 when CSR[hstatus].VSXL == 0
- 62:0 when CSR[hstatus].VSXL == 1

Description:

Written by hardware when a trap is taken into VS-mode.

Holds the interrupt or exception code for the last taken trap.

vscause.CODE is writable.

[when,"TRAP_ON_ILLEGAL_WLRL == true"]
If **vscause** is written with an undefined cause (combination of **vscause.INT** and **vscause.CODE**), an Illegal Instruction exception occurs.

[when,"TRAP_ON_ILLEGAL_WLRL == false"]
If **vscause** is written with an undefined cause (combination of **vscause.INT** and **vscause.CODE**), neither **vscause.INT** nor **vscause.CODE** are modified.

Valid interrupt codes are:

[separator="!"]
!===
<%- interrupt_codes.sort_by { |code| code.num }.each do |code| -%>
! <%= code.num %> ! <%= code.name %>
<%- end -%>
!===

Valid exception codes are:

[separator="!"]
!===
<%- exception_codes.sort_by { |code| code.num }.each do |code| -%>
! <%= code.num %> ! <%= code.name %>
<%- end -%>
!===

Type:
RW-RH

Reset value:
UNDEFINED_LEGAL

D.130.5. 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;
  }
}
```

RVA22 Profile Release | © RISC-V International

```
}
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;
}
```

DRAFT

D.131. vsepc

Virtual Supervisor Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in VS-mode.

Also controls where the hart jumps on an exception return from VS-mode.

D.131.1. Attributes

CSR Address	0x241
Virtual CSR Address	0x141
Defining extension	• H, version >= H@1.0.0
Length	64-bit
Privilege Mode	VS

D.131.2. Format

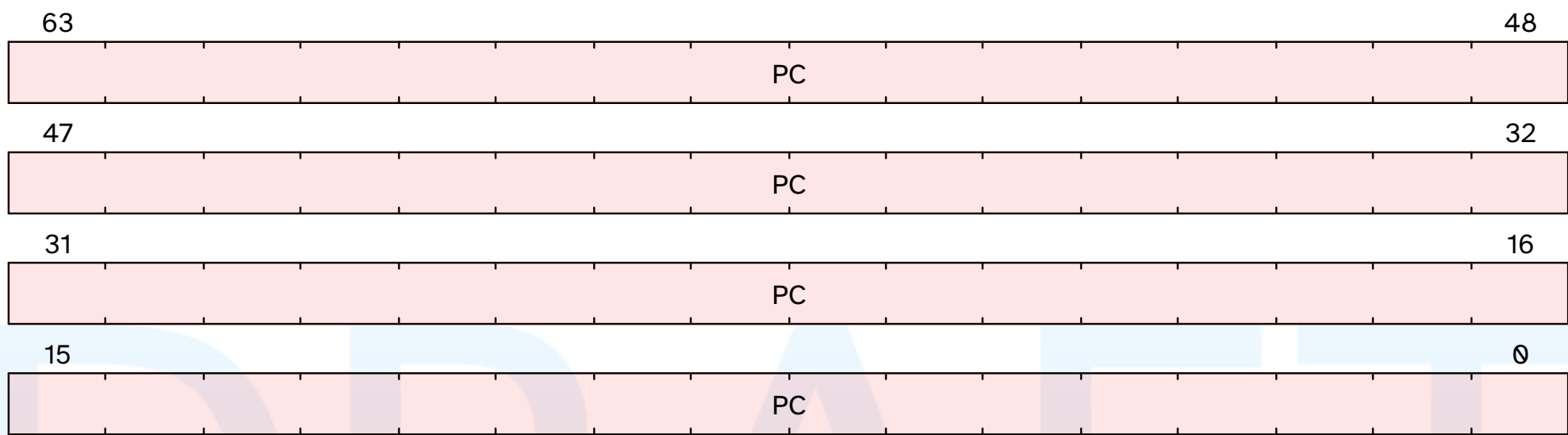


Figure 139. vsepc format

D.131.3. Field Summary

Na me	Location	Type	Reset Value
vsepc.PC	63:0	RW-RH	UNDEFINED_LEGAL

D.131.4. Fields

vsepc.PC Field

Location:

63:0

Description:

When a trap is taken into VS-mode, [vsepc.PC](#) is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, [vsepc.PC](#) is never written by the implementation, though it may be explicitly written by software.

On an exception return from VS-mode (from the SRET instruction), control transfers to the virtual address read out of [vsepc.PC](#).

Because PCs are always <% if ext?(:C) %>halfword<% else %>word<% end %>-aligned, <% if ext?(:C) %>bit 0<% else %>bits 1:0<% end %> of [vsepc.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.131.5. 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.131.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (implemented?(ExtensionName::C) && misa.C == 1'b1) {  
  return sepc.PC & ~64'b1;  
} else {  
  return sepc.PC;  
}
```

DRAFT

D.132. vsstatus

Virtual Supervisor Status

The vsstatus register tracks and controls the hart’s current operating state.

It is VS-mode’s version of sstatus, and substitutes for it when in VS-mode (i.e., in VS-mode CSR address 0x100 is vsstatus, not sstatus).

Unlike the relationship between sstatus and mstatus, none of the bits in vsstatus are aliases of another field.

D.132.1. Attributes

CSR Address	0x200
Virtual CSR Address	0x100
Defining extension	• H, version >= H@1.0.0
Length	32 when CSR[hstatus].VSXL == 0 64 when CSR[hstatus].VSXL == 1
Privilege Mode	VS

D.132.2. Format

This CSR format changes dynamically with XLEN.

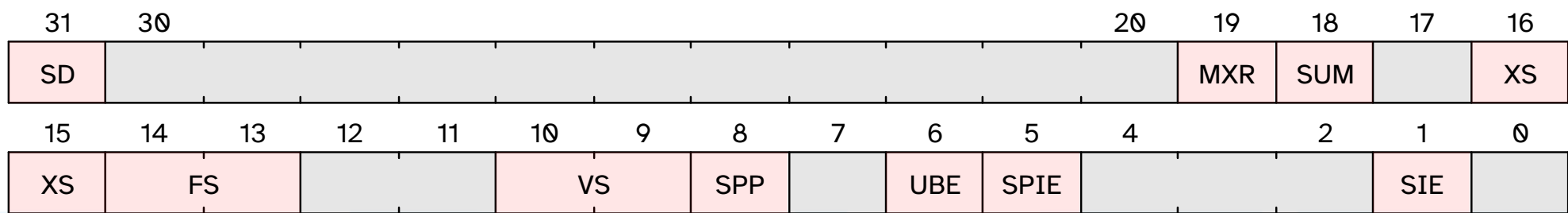


Figure 140. vsstatus Format when CSR[hstatus].VSXL == 0

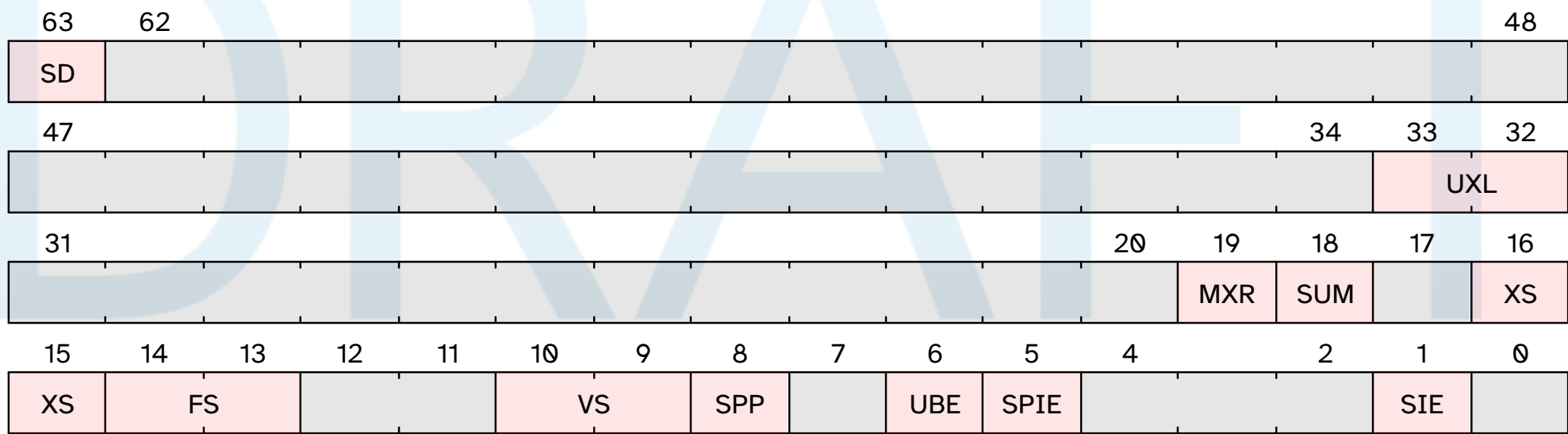


Figure 141. vsstatus Format when CSR[hstatus].VSXL == 1

D.132.3. Field Summary

Name	Location	Type	Reset Value
vsstat us.SD	31 63	RO-H	UNDEFINED_LEGAL
vsstat us.UXL	33:32	RW RO	UNDEFINED_LEGAL
vsstat us.MXR	19	RW	UNDEFINED_LEGAL
vsstat us.SUM	18	RW	UNDEFINED_LEGAL
vsstat us.XS	16:15	RO	0
vsstat us.FS	14:13	RW-H	UNDEFINED_LEGAL
vsstat us.VS	10:9	RW-H	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
vsstatus.SP	8	RW-H	UNDEFINED_LEGAL
vsstatus.UBE	6	RW RO	UNDEFINED_LEGAL
vsstatus.SPIE	5	RW-H	UNDEFINED_LEGAL
vsstatus.SIE	1	RW-H	UNDEFINED_LEGAL

D.132.4. Fields

[vsstatus.SD](#) Field

Location:

- 31 when `CSR[hstatus].VSXL == 0`
- 63 when `CSR[hstatus].VSXL == 1`

Description:

State Dirty

Read-only bit that summarizes whether any of the [vsstatus.FS](#), `<% if ext?(:V) %> vsstatus.VS, <% end %> or vsstatus.XS fields signal the presence of some dirty state (i.e., any of them hold the value 1).`

This bit is not an alias of [mstatus.SD](#) since it only reflects the state visible to VS-mode (e.g., `status.FS` does not affect [vsstatus.SD](#)).

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

[vsstatus.UXL](#) Field



UXL is only defined in RV64 (`CSR[hstatus].VSXL == 1`)

Location:

33:32

Description:

VU-mode XLEN

Sets the effective XLEN for VU-mode (0 = 32-bit, 1 = 64-bit, 2 = 128-bit).

[when, "`VUXLEN == 32`"]
Since the hart only supports `VUXLEN==32`, this is hardwired to 0.

[when, "`VUXLEN == 64`"]
Since the hart only supports `VUXLEN==64`, this is hardwired to 1.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

vsstatus.MXR Field

Location:

19

Description:

Make eXecutable Readable

Makes it possible to read executable pages when loading from effective VU-mode or VS-mode (normally, executable pages are not readable).

- When 1, load in effective VU-mode or VS-mode from pages marked readable **or executable** are allowed as long as the page is marked readable (or executable and `status.MXR` is set) in the G-stage translation.
- When 0, load in effective VU-mode or VS-mode from pages marked executable raise a Page Fault exception (unless `sstatus.MXR` is also set, in which case the above applies).

'vsstatus.MXR' affects all loads that operate under effective VU-mode or VS-mode:

- All loads in VU-mode
- All loads in VS-mode
- Loads in M-mode when `mstatus.MPRV` == 1, `mstatus.MPV` == 1, and `mstatus.MPP[1]` == 0
- Loads generated by one of the `hlv.*` instructions.

Type:

RW

Reset value:

UNDEFINED_LEGAL

vsstatus.SUM Field

Location:

18

Description:

permit Supervisor Memory Access

Allows VS-mode to read user pages.

Applies to the following loads and stores:

- All loads and stores in VS-mode.
- All loads and stores in M-mode when `mstatus.MPRV` == 1, `mstatus.MPP` == 1, and `mstatus.MPV` == 1
- Loads and stores generated by one of the `hlv.`, `hlvx.`, or `hsv.*` instructions.

When `vsstatus.SUM` is 0, the loads and stores from the above categories cause an Illegal Instruction exception if they access a user page during VS-level translation. Otherwise, a load or store from the above categories is permitted to access a user page during VS-level translation.

Type:

RW

Reset value:

UNDEFINED_LEGAL

vsstatus.XS Field

Location:

16:15

Description:

Custom (X) extension context Status

Summarizes the current state of any custom extension state.

Either 0 - Off, 1 - Initial, 2 - Clean, 3 - Dirty.
Since there are no custom extensions, this field is read-only 0.

Type:

RO

Reset value:

0

vsstatus.FS Field

Location:

14:13

Description:

Floating point context status

When 0, floating point instructions (from F and D extensions) in VS-mode or VU-mode are disabled, and cause *ILLEGAL INSTRUCTION* exceptions.
Floating point instructions in all modes, including VS-mode and VU-mode, are similarly disabled when *mstatus.FS* is clear.

When a floating point register, or the *fcsr* register is written in VS-mode or VU-mode, *vsstatus.FS* is written with the value 3.

Values 1 and 2 are valid write values for software, but are not interpreted by hardware other than to possibly enable a previously-disabled floating point unit.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

vsstatus.VS Field

Location:

10:9

Description:

Vector context status

When 0, vector instructions (from the V extension) are disabled, and cause *ILLEGAL INSTRUCTION* exceptions.
When a vector register or vector CSR is written, VS obtains the value 3.
Values 1 and 2 are valid write values for software, but are not interpreted by hardware other than to possibly enable a previously-disabled vector unit.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

vsstatus.SPP Field

Location:

8

Description:

VS-mode Previous Privilege

Written with the prior nominal privilege level (i.e., 0 for VU-mode and 1 for VS-mode) when entering VS-mode from an exception/interrupt.
Can also be written by software without immediate side-effect.

On a return from an exception from VS-mode, the machine will enter the nominal privilege level stored in *vsstatus.SPP*.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

vsstatus.UBE Field

Location:

6

Description:

VU-mode Big Endian

Controls the endianness of VU-mode (0 = little, 1 = big).

*[when,"VU_MODE_ENDIANNESSESS == 'little']
Since the CPU does not support big endian, this is hardwired to 0.*

*[when,"VU_MODE_ENDIANNESSESS == 'big']
Since the CPU does not support big endian, this is hardwired to 1.*

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

vsstatus.SPIE Field

Location:

5

Description:

VS-mode Previous Interrupt Enable

Written by hardware in two cases:

- *Written with prior value of vsstatus.SIE when entering VS-mode from an exception/interrupt.*
- *Written with the value 1 when returning from an exception in VS-mode (via the sret instruction).*

Can also be written by software without immediate side effect.

Other than serving as a record of nested traps as described above, vsstatus.SPIE does not affect execution.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

vsstatus.SIE Field

Location:

1

Description:

VS-mode Interrupt Enable

Written by hardware in two cases:

- *Written with the value 0 when entering VS-mode from an exception/interrupt.*
- *Written with the prior value of vsstatus.SPIE when returning from an exception in VS-mode (via sret).*

Affects execution by:

- *When 0, all VS-mode interrupts are disabled when the current privilege level is VS ((H)S-mode and M-mode interrupts are still enabled).*
- *When 1, VS-mode interrupts that are not otherwise disabled with a field in vsie are enabled.*

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

DRAFT

D.133. vstval

Virtual supervisor Trap Value

Holds trap-specific information

D.133.1. Attributes

CSR Address	0x243
Defining extension	H, version >= H@1.0.0
Length	32 when CSR[hstatus].VSXL == 0 64 when CSR[hstatus].VSXL == 1
Privilege Mode	S

D.133.2. Format

This CSR format changes dynamically with XLEN.

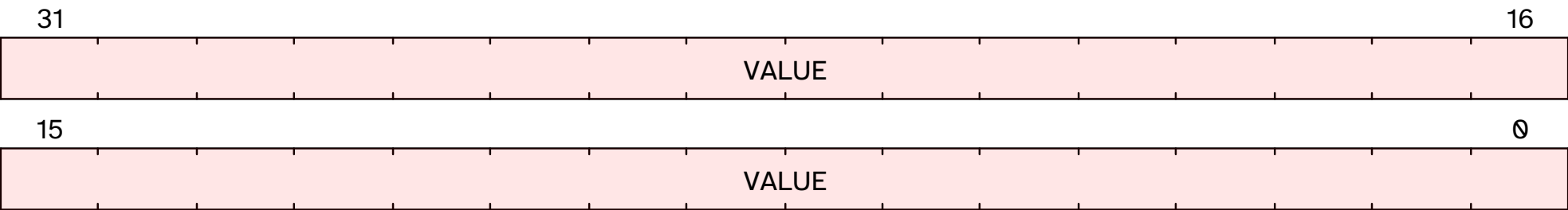


Figure 142. vstval Format when CSR[hstatus].VSXL == 0

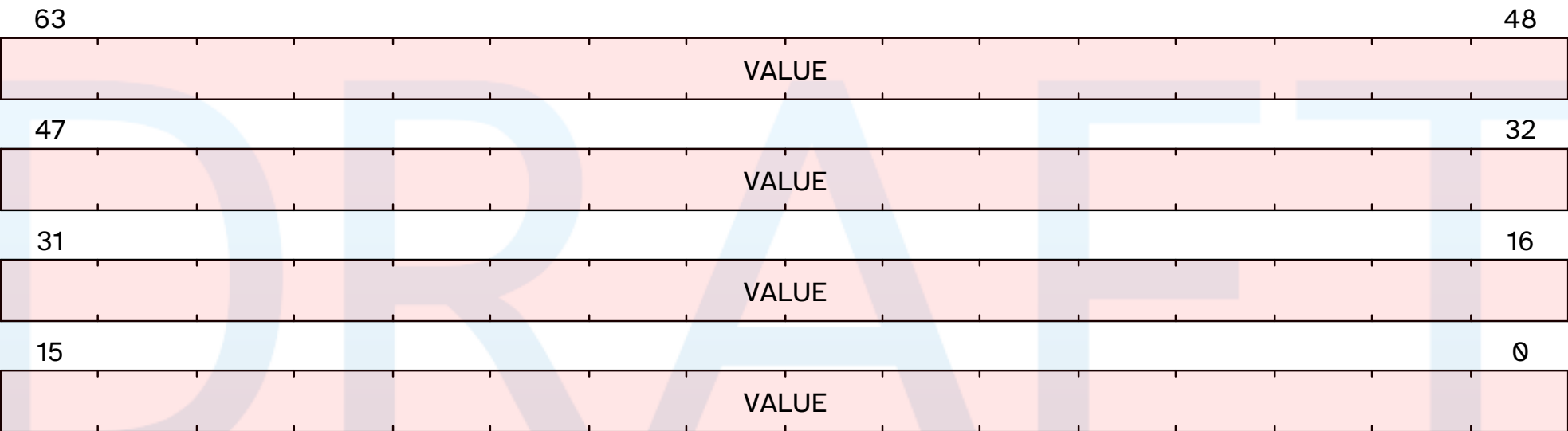


Figure 143. vstval Format when CSR[hstatus].VSXL == 1

D.133.3. Field Summary

Nam e	Location	Type	Reset Value
vstval . VALUE	31:0 63:0	RW-H	0

D.133.4. Fields

[vstval](#).[VALUE](#) Field

Location:

- 31:0 when CSR[mstatus].SXL == 0
- 63:0 when CSR[mstatus].SXL == 1

Description:

Written with trap-specific information when a trap is taken into VS-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

a! [when,"REPORT_VA_IN_VSTVAL_ON_BREAKPOINT == true"]

When caused by an EBREAK instruction, the virtual PC of the breakpoint instruction.

[when,"REPORT_VA_IN_VSTVAL_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

DRAFT

D.134. vstvec

Supervisor Trap Vector

Controls where traps jump.

D.134.1. Attributes

CSR Address	0x205
Defining extension	H, version >= H@1.0.0
Length	64-bit
Privilege Mode	S

D.134.2. Format

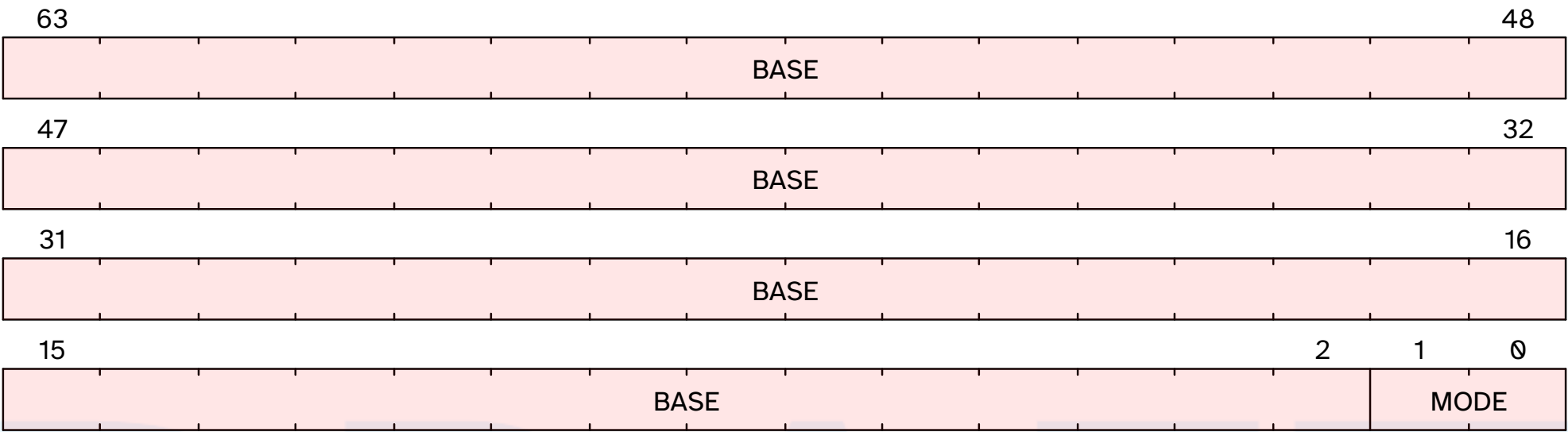


Figure 144. vstvec format

D.134.3. Field Summary

Nam e	Location	Type	Reset Value
vstvec.BAS E	63:2	RW-R	0
vstvec.MO DE	1:0	RW-R	0

D.134.4. Fields

vstvec.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 VS-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:

0

vstvec.MODE Field

Location:

1:0

Description:

Vectoring mode for asynchronous interrupts taken into VS-mode.

0 - Direct, 1 - Vectored

When Direct, all synchronous exceptions and asynchronous interrupts jump to (*vstvec.BASE* << 2).

When Vectored, asynchronous interrupts jump to (*vstvec.BASE* << 2 + *vscause**4) while synchronous exceptions continue to jump to (*vstvec.BASE* << 2).

Type:

RW-R

Reset value:

0

D.134.5. 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 (VSTVEC_MODE_DIRECT && csr_value.MODE == 0) {
    return 0;
} else if (VSTVEC_MODE_VECTORED && csr_value.MODE == 1) {
    return 1;
} else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
}
```

Appendix E: IDL Function Details

E.1. implemented? (generated)

Return true if the implementation supports extension.

Return Type	Boolean
Arguments	ExtensionName extension

E.2. implemented_version? (generated)

Return true if the implementation supports extension meeting 'version_requirement'.

Return Type	Boolean
Arguments	ExtensionName extension, String version_requirement

E.3. implemented_csr? (generated)

Return true if csr_addr is an implemented CSR

Return Type	Boolean
Arguments	Bits<12> csr_addr

E.4. direct_csr_lookup (generated)

Return CSR info for a CSR with direct address csr_addr.

If no CSR exists, <return_value>.valid == false

Return Type	Csr
Arguments	Bits<12> csr_addr

E.5. indirect_csr_lookup (generated)

Return CSR info for a CSR with indirect address csr_addr at window slot window_slot.

If no CSR exists, <return_value>.valid == false

Return Type	Csr
Arguments	Bits<MXLEN> csr_addr, Bits<4> window_slot

E.6. csr_hw_read (generated)

Returns the raw value of csr

Return Type	Bits
Arguments	Csr csr

E.7. csr_sw_read (generated)

Returns the result of CSR[csr].sw_read(); i.e., the software view of the register

Return Type	Bits
Arguments	Csr csr

E.8. csr_sw_write (generated)

Writes value to csr, applying an WARL transformations first.

Uses the sw_write(...) functions of CSR field definitions.

Return Type	void
Arguments	Csr csr, Bits<MXLEN> value

E.9. unpredictable (builtin)

Indicate that the hart has reached a state that is unpredictable because the RISC-V spec allows multiple behaviors. Generally, this will be a fatal condition to any emulation, since it is unclear what to do next.

The single argument *why* is a string describing why the hart entered an unpredictable state.

Return Type	void
Arguments	String why

E.10. read_hpm_counter (builtin)

Returns the value of hpmcounterN.

N must be between 3..31.

hpmcounterN must be implemented.

Return Type	Bits
Arguments	Bits<5> n

E.11. hartid (builtin)

Returns the value for [mhartid](#) as seen by this hart.

Must obey the rules of the priv spec:

The [mhartid](#) CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

Return Type	XReg
Arguments	None

E.12. read_mcycle (builtin)

Return the current value of the cycle counter.

Return Type	Bits
Arguments	None

E.13. read_mtime (builtin)

Return the current value of the real time device.

Return Type	Bits
Arguments	None

E.14. sw_write_mcycle (builtin)

Given a *value* that software is trying to write into mcycle, perform the write and return the value that will actually be written.

Return Type	Bits
Arguments	Bits<64> value

E.15. cache_block_zero (builtin)

Zero the cache block at the given physical address.

The cache block may be zeroed using 1 or more writes.

A cache-block-sized region is zeroed regardless of whether or not the memory is in a cacheable PMA region.

Return Type	void
Arguments	XReg cache_block_physical_address

E.16. eei_ecall_from_m (builtin)

When TRAP_ON_ECALL_FROM_M is false, this function will be called to emulate the EEI handling of ECALL-from-M.

If TRAP_ON_ECALL_FROM_M is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	void
Arguments	None

E.17. eei_ecall_from_s (builtin)

When TRAP_ON_ECALL_FROM_S is false, this function will be called to emulate the EEI handling of ECALL-from-S.

If TRAP_ON_ECALL_FROM_S is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	void
Arguments	None

E.18. eei_ecall_from_u (builtin)

When TRAP_ON_ECALL_FROM_U is false, this function will be called to emulate the EEI handling of ECALL-from-U.

If TRAP_ON_ECALL_FROM_U is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	void

Arguments	None

E.19. eei_ecall_from_vs (builtin)

When TRAP_ON_ECALL_FROM_VS is false, this function will be called to emulate the EEI handling of ECALL-from-VS.

If TRAP_ON_ECALL_FROM_VS is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	void
Arguments	None

E.20. eei_ebreak (builtin)

When TRAP_ON_EBREAK is false, this function will be called to emulate the EEI handling of EBREAK

If TRAP_ON_EBREAK is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	void
Arguments	None

E.21. memory_model_acquire (builtin)

Perform an acquire; that is, ensure that no subsequent operation in program order appears to an external observer to occur after the operation calling this function.

Return Type	void
Arguments	None

E.22. memory_model_release (builtin)

Perform a release; that is, ensure that no prior store in program order can be observed external to this hart after this function returns.

Return Type	void
Arguments	None

E.23. assert (builtin)

Assert that a condition is true. Failure represents an error in the IDL model.

Return Type	void
Arguments	Boolean test, String message

E.24. notify_mode_change (builtin)

Called whenever the privilege mode changes. Downstream tools can use this to hook events.

Return Type	void
Arguments	PrivilegeMode new_mode, PrivilegeMode old_mode

E.25. abort_current_instruction (builtin)

Abort the current instruction, and start refetching from \$pc.

Return Type	void
Arguments	None

E.26. ebreak (builtin)

Raise an Environment Break exception, returning control to the debug environment.

Return Type	void
Arguments	None

E.27. prefetch_instruction (builtin)

Hint to prefetch a block containing virtual_address for an upcoming fetch.

Return Type	void
Arguments	XReg virtual_address

E.28. prefetch_read (builtin)

Hint to prefetch a block containing virtual_address for an upcoming load.

Return Type	void
Arguments	XReg virtual_address

E.29. prefetch_write (builtin)

Hint to prefetch a block containing virtual_address for an upcoming store.

Return Type	void
Arguments	XReg virtual_address

E.30. fence (builtin)

Execute a memory ordering fence.(according to the FENCE instruction).

Return Type	void
Arguments	Boolean pi, Boolean pr, Boolean po, Boolean pw, Boolean si, Boolean sr, Boolean so, Boolean sw

E.31. fence_tso (builtin)

Execute a TSO memory ordering fence.(according to the FENCE instruction).

Return Type	void
Arguments	None

E.32. ifence (builtin)

Execute a memory ordering instruction fence (according to FENCE.I).

Return Type	void
Arguments	None

E.33. pause (builtin)

Pause hart retirement for a implementation-defined period of time, which may be zero.

See [Zihintpause](#) for more.

Return Type	void
Arguments	None

E.34. pow (generated)

Return value to the power exponent.

Return Type	XReg
Arguments	XReg value, XReg exponent

E.35. maybe_cache_translation (generated)

Given a translation result, potentially cache the result for later use. This function models a TLB fill operation. A valid implementation does nothing.

Return Type	void
Arguments	XReg vaddr, MemoryOperation op, TranslationResult result

E.36. cached_translation (generated)

Possibly returns a cached translation result matching vaddr.

CachedTranslationResult contains a Boolean 'valid' field. If valid, 'result' is a usable translation. Otherwise, the cache lookup failed.

Return Type	CachedTranslationResult
Arguments	XReg vaddr, MemoryOperation op

E.37. order_pgtbl_writes_before_vmafence (builtin)

Orders all writes prior to this call in global memory order that affect a page table in the set identified by order_type before any subsequent sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.W.INVALID.

A valid implementation does nothing if address caching is not used.

Return Type	void
Arguments	VmaOrderType order_type

E.38. order_pgtbl_reads_after_vmafence (builtin)

Orders all reads after to this call in global memory order to a page table in the set identified by order_type after any prior sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.INVALID.IR.

A valid implementation does nothing if address caching is not used.

Return Type	void
Arguments	VmaOrderType order_type

E.39. invalidate_translations (generated)

Locally invalidate the cached S-mode/VS-mode/G-stage address translations contained in the set identified by inval_type.

A valid implementation does nothing if address caching is not used.

Return Type	void
Arguments	VmaOrderType inval_type

E.40. read_physical_memory

Read from physical memory.

Return Type	Bits<len>
Arguments	XReg paddr

```
if (len == 8) {
    return read_physical_memory_8(paddr);
} else if (len == 16) {
    return read_physical_memory_16(paddr);
} else if (len == 32) {
    return read_physical_memory_32(paddr);
} else if (len == 64) {
    return read_physical_memory_64(paddr);
} else {
    assert(false, "Invalid len");
}
```

E.41. read_physical_memory_8 (builtin)

Read a byte from physical memory.

Return Type	Bits ⁸
Arguments	XReg paddr

E.42. read_physical_memory_16 (builtin)

Read two bytes from physical memory.

Return Type	Bits ¹⁶

Arguments	XReg paddr

E.43. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	Bits
Arguments	XReg paddr

E.44. read_physical_memory_64 (builtin)

Read eight bytes from physical memory.

Return Type	Bits
Arguments	XReg paddr

E.45. write_physical_memory

Write to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<len> value

```
if (len == 8) {
    write_physical_memory_8(paddr, value);
} else if (len == 16) {
    write_physical_memory_16(paddr, value);
} else if (len == 32) {
    write_physical_memory_32(paddr, value);
} else if (len == 64) {
    write_physical_memory_64(paddr, value);
} else {
    assert(false, "Invalid len");
}
```

E.46. write_physical_memory_8 (builtin)

Write a byte to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<8> value

E.47. write_physical_memory_16 (builtin)

Write two bytes to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<16> value

E.48. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<32> value

E.49. write_physical_memory_64 (builtin)

Write eight bytes to physical memory.

Return Type	void
Arguments	XReg paddr, Bits<64> value

E.50. wfi (builtin)

Wait-for-interrupt: hint that the processor should enter a low power state until the next interrupt.

A valid implementation is a no-op.

The model will advance the PC; this function does not need to.

Return Type	void
Arguments	None

E.51. pma_applies? (builtin)

Checks if *attr* is applied to the entire physical address region between [paddr, paddr + len) based on static PMA attributes.

Return Type	Boolean
Arguments	PmaAttribute attr, Bits<PHYS_ADDR_WIDTH> paddr, U32 len

E.52. atomic_check_then_write_32 (builtin)

Atomically:

- Reads 32-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 32-bits

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, Bits<32> compare_value, Bits<32> write_value

E.53. atomic_check_then_write_64 (builtin)

Atomically:

- Reads 64-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 64-bits

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, Bits<64> compare_value, Bits<64> write_value

E.54. atomically_set_pte_a (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
 - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' bit and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len

E.55. atomically_set_pte_a_d (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
 - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' and 'D' bits and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len

E.56. atomic_read_modify_write_32 (builtin)

Atomically read-modify-write 32-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it’s assumed the RMW is OK to proceed.

Return Type	Bits

Arguments	
	Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<32> op value, AmoOperation

E.57. `atomic_read_modify_write_64` (builtin)

Atomically read-modify-write 64-bits starting at `phys_address` using `value` and `op`.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	Bits
Arguments	Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<64> value, AmoOperation op

E.58. set_external_interrupt

Set an external interrupt targeting target_mode

Return Type	<code>void</code>
Arguments	<code>PrivilegeMode target_mode</code>

```

if (target_mode == PrivilegeMode::M) {
    mip.MEIP = 1'b1;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
    pending_smode_external_interrupt = true;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
    mip.VSEIP = 1'b1;
} else {
    assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();

```

E.59. clear_external_interrupt

Clear an external interrupt targeting target_mode

Return Type	<code>void</code>
Arguments	<code>PrivilegeMode target_mode</code>

```

if (target_mode == PrivilegeMode::M) {
    mip.MEIP = 1'b0;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
    pending_smode_external_interrupt = false;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
    mip.VSEIP = 1'b0;
} else {
    assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();

```

E.60. set_software_interrupt

Set a software interrupt targeting target_mode

Return Type	void
Arguments	PrivilegeMode target_mode

```
if (target_mode == PrivilegeMode::M) {
    mip.MSIP = 1'b1;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
    mip.SSIP = 1'b1;
} else {
    assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

E.61. clear_software_interrupt

Clear a software interrupt targeting target_mode

Return Type	void
Arguments	PrivilegeMode target_mode

```
if (target_mode == PrivilegeMode::M) {
    mip.MSIP = 1'b0;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
    mip.SSIP = 1'b0;
} else {
    assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

E.62. set_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	void
Arguments	PrivilegeMode target_mode

```
if (target_mode == PrivilegeMode::M) {
    mip.MTIP = 1'b1;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
    mip.STIP = 1'b1;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
    pending_vsmode_timer_interrupt = true;
} else {
    assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

E.63. clear_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	void
Arguments	PrivilegeMode target_mode

```
if (target_mode == PrivilegeMode::M) {
    mip.MTIP = 1'b0;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
    mip.STIP = 1'b0;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
    pending_vsmode_timer_interrupt = false;
} else {
    assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

E.64. refresh_pending_interrupts

refreshes the calculation of a pending interrupt

needs to be called after any state update that could change a pending interrupt. This includes: - CSR[mip] - CSR[mie] - CSR[mstatus].MIE - CSR[mstatus].SIE - CSR[vsstatus].SIE - CSR[mideleg] - CSR[sideleg] - CSR[hideleg] - CSR[hvip] - CSR[hgeip] - CSR[hgeie] - mode changes

Return Type	void
Arguments	None

```
Bits<MXLEN> pending_ints = mip.sw_read() & $bits(mie);
if (pending_ints == 0) {
    pending_and_enabled_interrupts = 0;
    return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") && implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_enabled_ints = mode() == PrivilegeMode::M && (mstatus.MIE == 1'b0 ? 0 : ($bits(mie) & (HAS_MIDELEG ? ~$bits(mideleg) : ~MXLEN'0)));
Bits<MXLEN> mmode_pending_and_enabled = pending_ints & mmode_enabled_ints;
if (mmode_pending_and_enabled != 0) {
    pending_and_enabled_interrupts = mmode_pending_and_enabled;
    return ;
}
if (misa.S == 1'b1) {
    Bits<MXLEN> smode_enabled_ints = mode() == PrivilegeMode::M || (mstatus.SIE == 1'b0 ? 0 : $bits(mie) & ($bits(mideleg)));
    Bits<MXLEN> smode_pending_and_enabled = pending_ints & smode_enabled_ints;
    if (smode_pending_and_enabled != 0) {
        pending_and_enabled_interrupts = smode_pending_and_enabled;
        return ;
    }
}
pending_and_enabled_interrupts = 0;
```

E.65. highest_priority_interrupt

Given a bitmask of interrupts in the format of MIE/MIP, return the highest priority interrupt code that is set

Interrupt priority is: MEI, MSI, MTI, SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI

Return Type	InterruptCode
Arguments	Bits<MXLEN> int_mask

```
if (int_mask[$bits(InterruptCode::MachineExternal)] == 1'b1) {
    return InterruptCode::MachineExternal;
} else if (int_mask[$bits(InterruptCode::MachineSoftware)] == 1'b1) {
    return InterruptCode::MachineSoftware;
} else if (int_mask[$bits(InterruptCode::MachineTimer)] == 1'b1) {
    return InterruptCode::MachineTimer;
}
if (misa.S == 1'b1) {
    if (int_mask[$bits(InterruptCode::SupervisorExternal)] == 1'b1) {
```

```
    return InterruptCode::SupervisorExternal;
  } else if (int_mask[$bits(InterruptCode::SupervisorSoftware)] == 1'b1) {
    return InterruptCode::SupervisorSoftware;
  } else if (int_mask[$bits(InterruptCode::SupervisorTimer)] == 1'b1) {
    return InterruptCode::SupervisorTimer;
  }
}
if (implemented?(ExtensionName::Sscofpmf)) {
  if (int_mask[$bits(InterruptCode::LocalCounterOverflow)] == 1'b1) {
    return InterruptCode::LocalCounterOverflow;
  }
}
assert(false, "There is no valid interrupt");
```

E.66. choose_interrupt

Return the highest priority interrupt that is both pending and enabled and the mode it will be taken in

Return Type	InterruptCode, PrivilegeMode
Arguments	None

```
InterruptCode chosen;
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") && implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_pending_and_enabled = pending_and_enabled_interrupts & ~(HAS_MIDELEG ? $bits(mideleg) : MXLEN'0);
if (mmode_pending_and_enabled != 0) {
  assert((mode() != PrivilegeMode::M) || (mstatus.MIE == 1'b1), "M-mode interrupts are not enabled");
  chosen = highest_priority_interrupt(mmode_pending_and_enabled);
} else if (misa.S == 1'b1) {
  Bits<MXLEN> smode_pending_and_enabled = (pending_and_enabled_interrupts & $bits(mideleg));
  if (smode_pending_and_enabled != 0) {
    assert((mode() == PrivilegeMode::U) || (mode() == PrivilegeMode::VU) || (mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::S) && (mstatus.SIE == 1'b1), "S-mode interrupt can't be triggered");
    chosen = highest_priority_interrupt(smode_pending_and_enabled);
  }
}
assert($bits(chosen) != 0, "Didn't pick interrupt?");
PrivilegeMode to_mode;
Bits<MXLEN> chosen_mask = (MXLEN'1 << $bits(chosen));
if (((HAS_MIDELEG ? $bits(mideleg) : MXLEN'0) & chosen_mask) == 0) {
  to_mode = PrivilegeMode::M;
} else {
  if (misa.S == 1'b1) {
    to_mode = PrivilegeMode::S;
  } else {
    to_mode = PrivilegeMode::U;
  }
}
return chosen, to_mode;
```

E.67. take_interrupt

Take (adjust CSRs and set PC to handler) the highest priority interrupt that is both pending and enabled

Return Type	void
Arguments	None

```
PrivilegeMode to_mode;
InterruptCode code;
(code, to_mode = choose_interrupt());
if (to_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
  mstatus.MPP = $bits(mode())[1:0];
  if (misa.H == 1'b1) {
    mstatus.MPV = $bits(mode())[2];
    mtval2.VALUE = 0;
  }
}
```

```
mtinst.VALUE = 0;
}
mcause.CODE = $bits(code);
mcause.INT = 1'b1;
mtval.VALUE = 0;
if (mtvec.MODE == 0) {
    $pc = {mtvec.BASE, 2'b00};
} else if (mtvec.MODE == 1'b1) {
    $pc = {mtvec.BASE, 2'b00} + ($bits(code) * 4);
}
} else if ((misa.S == 1'b1) && (to_mode == PrivilegeMode::S)) {
    sepc.PC = $pc;
    mstatus.SPP = $bits(mode())[0];
    if (misa.H == 1'b1) {
        hstatus.SPV = $bits(mode())[2];
    }
    scause.CODE = $bits(code);
    scause.INT = 1'b1;
    stval.VALUE = 0;
    if (stvec.MODE == 0) {
        $pc = {stvec.BASE, 2'b00};
    } else if (stvec.MODE == 1'b1) {
        $pc = {stvec.BASE, 2'b00} + ($bits(code) * 4);
    }
} else if ((misa.H == 1'b1) && (to_mode == PrivilegeMode::VS)) {
    vsepc.PC = $pc;
    vsstatus.SPP = $bits(mode())[0];
    vscause.CODE = $bits(code);
    vscause.INT = 1'b1;
    vstval.VALUE = 0;
    if (vstvec.MODE == 0) {
        $pc = {vstvec.BASE, 2'b00};
    } else if (vstvec.MODE == 1'b1) {
        $pc = {vstvec.BASE, 2'b00} + ($bits(code) * 4);
    }
}
}
set_mode_no_refresh(to_mode);
```

E.68. fetch_memory_aligned_16

Fetch 16 bits from virtual memory using a known aligned address.

Return Type	Bits ¹⁶
Arguments	XReg virtual_address

```
TranslationResult result;
if (misa.S == 1) {
    result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
    result.paddr = virtual_address;
}
access_check(result.paddr, 16, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<16>(result.paddr);
```

E.69. fetch_memory_aligned_32

Fetch 32 bits from virtual memory using a known aligned address.

Return Type	Bits
Arguments	XReg virtual_address

```
TranslationResult result;
```

```
if (misa.S == 1) {
    result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
    result.paddr = virtual_address;
}
access_check(result.paddr, 32, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<32>(result.paddr);
```

E.70. power_of_2?

Returns true if value is a power of two, false otherwise

Return Type	Boolean
Arguments	Bits<N> value

```
return (value != 0) && value & (value - 1 == 0);
```

E.71. ary_includes?

Returns true if *value* is an element of *ary*, and false otherwise

Return Type	Boolean
Arguments	Bits<ELEMENT_SIZE> ary[ARY_SIZE], Bits<ELEMENT_SIZE> value

```
for (U32 i = 0; i < ARY_SIZE; i++) {
    if (ary[i] == value) {
        return true;
    }
}
return false;
```

E.72. has_virt_mem?

Returns true if some virtual memory translation (Sv*) is supported in the config.

Return Type	Boolean
Arguments	None

```
return implemented?(ExtensionName::Sv32) || implemented?(ExtensionName::Sv39) || implemented?(ExtensionName::Sv48) ||
implemented?(ExtensionName::Sv57);
```

E.73. highest_set_bit

Returns the position of the highest (nearest MSB) bit that is '1', or -1 if value is zero.

Return Type	XReg
Arguments	XReg value

```
for (U32 i = xlen() - 1; i >= 0; i--) {
    if (value[i] == 1) {
        return i;
    }
}
```

```
}
return - 'sd1;
```

E.74. lowest_set_bit

Returns the position of the lowest (nearest LSB) bit that is '1', or XLEN if value is zero.

Return Type	XReg
Arguments	XReg value

```
for (U32 i = 0; i < xlen(); i++) {
    if (value[i] == 1) {
        return i;
    }
}
return xlen();
```

E.75. bit_length

Returns the minimum number of bits needed to represent value.

Only works on unsigned values.

The value 0 returns 1.

Return Type	XReg
Arguments	XReg value

```
for (XReg i = 63; i > 0; i--) {
    if (value[i] == 1) {
        return i;
    }
}
return 1;
```

E.76. count_leading_zeros

Returns the number of leading 0 bits before the most-significant 1 bit of value, or N if value is zero.

Return Type	Bits<bit_length(N)>
Arguments	Bits<N> value

```
for (U32 i = 0; i < N; i++) {
    if (value[N - 1 - i] == 1) {
        return i;
    }
}
return N;
```

E.77. sext

Sign extend value starting at first_extended_bit.

Bits [XLEN-1: first_extended_bit`] of the return value should get the value of bit (first_extended bit - 1).

Return Type	XReg
Arguments	XReg value, XReg first_extended_bit

```
if (first_extended_bit == MXLEN) {
    return value;
} else {
    Bits<1> sign = value[first_extended_bit - 1];
    for (U32 i = MXLEN - 1; i >= first_extended_bit; i--) {
        value[i] = sign;
    }
    return value;
}
```

E.78. is_naturally_aligned

Checks if value is naturally aligned to N bits.

Return Type	Boolean
Arguments	XReg value

```
return true if (N == 8);
XReg Mask = (N / 8) - 1;
return (value & ~Mask) == value;
```

E.79. in_naturally_aligned_region?

Checks if a length-bit access starting at address lies entirely within an N-bit naturally-aligned region.

Return Type	Boolean
Arguments	XReg address, U32 length

```
XReg Mask = (N / 8) - 1;
return (address & ~Mask) == ((address + length - 1) & ~Mask);
```

E.80. contains?

Given a *region* defined by region_start, region_size, determine if a *target* defined by target_start, target_size is completely contained with the region.

Return Type	Boolean
Arguments	XReg region_start, U32 region_size, XReg target_start, U32 target_size

```
return target_start >= region_start && (target_start + target_size) <= (region_start + region_size);
```

E.81. set_fp_flag

Add flag to the sticky flags bits in CSR[fcsr]

Return Type	void

Arguments	FpFlag flag

```
if (flag == FpFlag::NX) {
    fcsr.NX = 1;
} else if (flag == FpFlag::UF) {
    fcsr.UF = 1;
} else if (flag == FpFlag::OF) {
    fcsr.OF = 1;
} else if (flag == FpFlag::DZ) {
    fcsr.DZ = 1;
} else if (flag == FpFlag::NV) {
    fcsr.NV = 1;
}
```

E.82. rm_to_mode

Convert rm to a RoundingMode.

encoding is the full encoding of the instruction rm comes from.

Will raise an IllegalInstruction exception if rm is a reserved encoding.

Return Type	RoundingMode
Arguments	Bits<3> rm, Bits<32> encoding

```
if (rm == $bits(RoundingMode::RNE)) {
    return RoundingMode::RNE;
} else if (rm == $bits(RoundingMode::RTZ)) {
    return RoundingMode::RTZ;
} else if (rm == $bits(RoundingMode::RDN)) {
    return RoundingMode::RDN;
} else if (rm == $bits(RoundingMode::RUP)) {
    return RoundingMode::RUP;
} else if (rm == $bits(RoundingMode::RMM)) {
    return RoundingMode::RMM;
} else if (rm == $bits(RoundingMode::DYN)) {
    return $enum(RoundingMode, fcsr.FRM);
} else {
    raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

E.83. mark_f_state_dirty

Potentially updates mstatus.FS to the Dirty (3) state, depending on configuration settings.

Return Type	void
Arguments	None

```
if (HW_MSTATUS_FS_DIRTY_UPDATE == "precise") {
    mstatus.FS = 3;
} else if (HW_MSTATUS_FS_DIRTY_UPDATE == "imprecise") {
    unpredictable("The hart may or may not update mstatus.FS now");
}
```

E.84. nan_box

Produces a properly NaN-boxed floating-point value from a floating-point value of smaller size by adding all 1’s to the upper bits.

Return Type	Bits<T0_SIZE>
Arguments	Bits<FROM_SIZE> from_value

```
assert(FROM_SIZE < T0_SIZE, "Bad template arguments; FROM_SIZE must be less than T0_SIZE");
return {{T0_SIZE - FROM_SIZE{1'b1}}, from_value};
```

E.85. check_f_ok

Checks if instructions from the F extension can be executed, and, if not, raise an exception.

Return Type	void
Arguments	Bits<INSTR_ENC_SIZE> encoding

```
if (MUTABLE_MISA_F && misa.F == 0) {
    raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
if (mstatus.FS == 0) {
    raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

E.86. is_sp_neg_inf?

Return true if sp_value is negative infinity.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return sp_value == SP_NEG_INF;
```

E.87. is_sp_pos_inf?

Return true if sp_value is positive infinity.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return sp_value == SP_POS_INF;
```

E.88. is_sp_neg_norm?

Returns true if sp_value is a negative normal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 1) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

E.89. is_sp_pos_norm?

Returns true if sp_value is a positive normal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 0) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

E.90. is_sp_neg_subnorm?

Returns true if sp_value is a negative subnormal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 1) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

E.91. is_sp_pos_subnorm?

Returns true if sp_value is a positive subnormal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 0) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

E.92. is_sp_neg_zero?

Returns true if sp_value is negative zero.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return sp_value == SP_NEG_ZERO;
```

E.93. is_sp_pos_zero?

Returns true if sp_value is positive zero.

Return Type	Boolean

Arguments	Bits<32> sp_value
return sp_value == SP_POS_ZERO;	

E.94. is_sp_nan?

Returns true if sp_value is a NaN (quiet or signaling)

Return Type	Boolean
Arguments	Bits<32> sp_value
return (sp_value[30:23] == 0b11111111) && (sp_value[22:0] != 0);	

E.95. is_sp_signaling_nan?

Returns true if sp_value is a signaling NaN

Return Type	Boolean
Arguments	Bits<32> sp_value
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 0) && (sp_value[21:0] != 0);	

E.96. is_sp_quiet_nan?

Returns true if sp_value is a quiet NaN

Return Type	Boolean
Arguments	Bits<32> sp_value
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 1);	

E.97. softfloat_shiftRightJam32

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned. The value of dist can be arbitrarily large. In particular, if dist is greater than 32, the result will be either 0 or 1, depending on whether a is zero or nonzero.

Return Type	Bits
Arguments	Bits<32> a, Bits<32> dist
return (dist < 31) ? a >> dist (a << (-dist & 31 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);	

E.98. softfloat_shiftRightJam64

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned.

The value of 'dist' can be arbitrarily large. In particular, if dist is greater than 64, the result will be either 0 or 1, depending on whether a is zero or nonzero.

Return Type	Bits
Arguments	Bits<64> a, Bits<32> dist

```
return (dist < 63) ? a >> dist | (a << (-dist & 63 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

E.99. softfloat_roundToI32

Round to unsigned 32-bit integer, using rounding_mode

Return Type	Bits
Arguments	Bits<1> sign, Bits<64> sig, RoundingMode roundingMode

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
    roundIncrement = 0;
    if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
        roundIncrement = 0xFFF;
    }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
    set_fp_flag(FpFlag::NV);
    return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
    sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 0) != (sign == 1) {
    set_fp_flag(FpFlag::NV);
    return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
if (roundBits != 0) {
    set_fp_flag(FpFlag::NX);
}
return z;
```

E.100. packToF32UI

Pack components into a 32-bit value

Return Type	Bits
Arguments	Bits<1> sign, Bits<8> exp, Bits<23> sig

```
return {sign, exp, sig};
```

E.101. packToF16UI

Pack components into a 16-bit value

Return Type	Bits
Arguments	Bits<1> sign, Bits<5> exp, Bits<10> sig

```
return {sign, exp, sig};
```

E.102. softfloat_normSubnormalF16Sig

normalize subnormal half-precision value

Return Type	Bits<5>, Bits@
Arguments	Bits<16> hp_value

```
Bits<8> shift_dist = count_leading_zeros<16>(hp_value);
return 1 - shift_dist, hp_value << shift_dist;
```

E.103. softfloat_roundPackToF32

Round FP value according to mmode and then pack it in IEEE format.

Return Type	Bits
Arguments	Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
    roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
    if ($signed(exp) < 0) {
        Boolean isTiny = ($signed(exp) < -8's1) || (sig + roundIncrement < 0x80000000);
        sig = softfloat_shiftRightJam32(sig, -exp);
        exp = 0;
        roundBits = sig & 0x7F;
        if (isTiny && (roundBits != 0)) {
            set_fp_flag(FpFlag::UF);
        }
    } else if (0xFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
        set_fp_flag(FpFlag::OF);
        set_fp_flag(FpFlag::NX);
        return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0 ? 1 : 0;    } } sig = (sig + roundIncrement);
if (sig == 0) {
    exp = 0;
}
return packToF32UI(sign, exp, sig);
```

E.104. softfloat_normRoundPackToF32

Normalize, round, and pack into a 32-bit floating point value

Return Type	Bits
Arguments	Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
    return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
    return softfloat_roundPackToF32(sign, exp, sig << shiftDist, mode);
}
```

E.105. signF32UI

Extract sign-bit of a 32-bit floating point number

Return Type	Bits①
Arguments	Bits<32> a

```
return a[31];
```

E.106. expF32UI

Extract exponent of a 32-bit floating point number

Return Type	Bits⑧
Arguments	Bits<32> a

```
return a[30:23];
```

E.107. fracF32UI

Extract significand of a 32-bit floating point number

Return Type	Bits
Arguments	Bits<32> a

```
return a[22:0];
```

E.108. returnNonSignalingNaN

Returns a non-signalling NaN version of the floating-point number Does not modify the input

Return Type	U32
Arguments	U32 a

```
U32 a_copy = a;
a_copy[22] = 1'b1;
return a_copy;
```


E.109. returnMag

Returns magnitude of the given number Does not modify the input

Return Type	U32
Arguments	U32 a

```
U32 a_copy = a;
a_copy[31] = 1'b0;
return a_copy;
```

E.110. returnLargerMag

Returns the larger number between a and b by magnitude If either number is signaling NaN then that is made quiet

Return Type	U32
Arguments	U32 a, U32 b

```
U32 mag_a = returnMag(a);
U32 mag_b = returnMag(b);
U32 nonsig_a = returnNonSignalingNaN(a);
U32 nonsig_b = returnNonSignalingNaN(b);
if (mag_a < mag_b) {
    return nonsig_b;
}
if (mag_b < mag_a) {
    return nonsig_a;
}
return (nonsig_a < nonsig_b) ? nonsig_a : nonsig_b;
```

E.111. softfloat_propagateNaNF32UI

Interpreting 'a' and 'b' as the bit patterns of two 32-bit floating- | point values, at least one of which is a NaN, returns the bit pattern of | the combined NaN result. If either 'a' or 'b' has the pattern of a | signaling NaN, the invalid exception is raised.

Return Type	U32
Arguments	U32 a, U32 b

```
Boolean isSigNaN_a = is_sp_signaling_nan?(a);
Boolean isSigNaN_b = is_sp_signaling_nan?(b);
if (isSigNaN_a || isSigNaN_b) {
    set_fp_flag(FpFlag::NV);
}
return SP_CANONICAL_NAN;
```

E.112. softfloat_addMagsF32

Returns sum of the magnitudes of 2 floating point numbers

Return Type	U32
Arguments	U32 a, U32 b, RoundingMode mode

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
    if (expA == 8'd0) {
        z = a + b;
        return z;
    }
    if (expA == 8'hFF) {
        if ((sigA != 8'd0) || (sigB != 8'd0)) {
            return softfloat_propagateNaNF32UI(a, b);
        }
        return a;
    }
    signZ = signF32UI(a);
    expZ = expA;
    sigZ = 32'h01000000 + sigA + sigB;
    if (sigZ & 0x1) == 0) && (expZ < 8'hFE {
        sigZ = sigZ >> 1;
        return (32'h0 + (signZ << 31) + (expZ << 23) + sigZ);
    }
    sigZ = sigZ << 6;
} else {
    signZ = signF32UI(a);
    U32 sigA_32 = 32'h0 + (sigA << 6);
    U32 sigB_32 = 32'h0 + (sigA << 6);
    if (expDiff < 0) {
        if (expB == 8'hFF) {
            if (sigB != 0) {
                return softfloat_propagateNaNF32UI(a, b);
            }
            return packToF32UI(signZ, 8'hFF, 23'h0);
        }
        expZ = expB;
        sigA_32 = (expA == 0) ? 2 * sigA_32 : (sigA_32 + 0x200000000);
        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 + 0x200000000);
        sigB_32 = softfloat_shiftRightJam32(sigB_32, (32'h0 + expDiff));
    }
    U32 sigZ = 0x20000000 + sigA + sigB;
    if (sigZ < 0x40000000) {
        expZ = expZ - 1;
        sigZ = sigZ << 1;
    }
}
return softfloat_roundPackToF32(signZ, expZ, sigZ[22:0], mode);
```

E.113. softfloat_subMagsF32

Returns difference of the magnitudes of 2 floating point numbers

Return Type	U32
Arguments	U32 a, U32 b, RoundingMode mode


```

Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
U32 sigDiff;
U32 sigX;
U32 sigY;
U32 sigA_32;
U32 sigB_32;
Bits<8> shiftDist;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
    if (expA == 8'hFF) {
        if ((sigA != 8'd0) || (sigB != 8'd0)) {
            return softfloat_propagateNaNF32UI(a, b);
        }
        return a;
    }
    sigDiff = sigA - sigB;
    if (sigDiff == 0) {
        return packToF32UI(((mode == RoundingMode::RDN) ? 1 : 0), 0, 0);
    }
    if (expA != 0) {
        expA = expA - 1;
    }
    signZ = signF32UI(a);
    if (sigDiff < 0) {
        signZ = ~signZ;
        sigDiff = -32'sh1 * sigDiff;
    }
    shiftDist = count_leading_zeros<32>(sigDiff) - 8;
    expZ = expA - shiftDist;
    if (expZ < 0) {
        shiftDist = expA;
        expZ = 0;
    }
    return packToF32UI(signZ, expZ, sigDiff << shiftDist);
} else {
    signZ = signF32UI(a);
    sigA_32 = 32'h0 + (sigA << 7);
    sigB_32 = 32'h0 + (sigB << 7);
    if (expDiff < 0) {
        signZ = ~signZ;
        if (expB == 0xFF) {
            if (sigB_32 != 0) {
                return softfloat_propagateNaNF32UI(a, b);
            }
            return packToF32UI(signZ, expB, 0);
        }
        expZ = expB - 1;
        sigX = sigB_32 | 0x40000000;
        sigY = sigA_32 + ((expA != 0) ? 0x40000000 : sigA_32);
        expDiff = -expDiff;
    } else {
        if (expA == 0xFF) {
            if (sigA_32 != 0) {
                return softfloat_propagateNaNF32UI(a, b);
            }
            return a;
        }
        expZ = expA - 1;
        sigX = sigA_32 | 0x40000000;
        sigY = sigB_32 + ((expB != 0) ? 0x40000000 : sigB_32);
    }
    return softfloat_normRoundPackToF32(signZ, expZ, sigX - softfloat_shiftRightJam32(sigY, expDiff), mode);
}

```

E.114. f32_add

Returns sum of 2 floating point numbers

Return Type	U32
Arguments	U32 a, U32 b, RoundingMode mode

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
    return softfloat_subMagsF32(a, b, mode);
} else {
    return softfloat_addMagsF32(a, b, mode);
}
```

E.115. f32_sub

Returns difference of 2 floating point numbers

Return Type	U32
Arguments	U32 a, U32 b, RoundingMode mode

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
    return softfloat_addMagsF32(a, b, mode);
} else {
    return softfloat_subMagsF32(a, b, mode);
}
```

E.116. i32_to_f32

Converts 32-bit signed integer to 32-bit floating point

Return Type	U32
Arguments	U32 a, RoundingMode mode

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFFF) == 0) {
    return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32(sign, 0x9C, magnitude_of_A, mode);
```

E.117. ui32_to_f32

Converts 32-bit unsigned integer to 32-bit floating point

Return Type	U32
Arguments	U32 a, RoundingMode mode

```
if (a == 0) {
    return a;
}
if (a[31] == 1) {
    return softfloat_roundPackToF32(0, 0x9D, a >> 1 | (a & 1), mode);
}
```

```
    } else {
        return softfloat_normRoundPackToF32(0, 0x9C, a, mode);
    }
```

E.118. mode

Returns the current active privilege mode.

Return Type	PrivilegeMode
Arguments	None

```
if (!implemented?(ExtensionName::S) && !implemented?(ExtensionName::U) && !implemented?(ExtensionName::H)) {
    return PrivilegeMode::M;
} else {
    return current_mode;
}
```

E.119. set_mode_no_refresh

Set the current privilege mode to new_mode, but don't refresh interrupts

Return Type	void
Arguments	PrivilegeMode new_mode

```
if (new_mode != current_mode) {
    notify_mode_change(new_mode, current_mode);
    current_mode = new_mode;
}
```

E.120. set_mode

Set the current privilege mode to new_mode

Return Type	void
Arguments	PrivilegeMode new_mode

```
if (new_mode != current_mode) {
    notify_mode_change(new_mode, current_mode);
    current_mode = new_mode;
    refresh_pending_interrupts();
}
```

E.121. compatible_mode?

Returns true if target_mode is more privileged than actual_mode.

Return Type	Boolean
Arguments	PrivilegeMode target_mode, PrivilegeMode actual_mode

```
if (target_mode == PrivilegeMode::M) {
    return actual_mode == PrivilegeMode::M;
} else if (target_mode == PrivilegeMode::S) {
```

```
return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S);
} else if (target_mode == PrivilegeMode::U) {
    return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::U);
} else if (target_mode == PrivilegeMode::VS) {
    return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS);
} else if (target_mode == PrivilegeMode::VU) {
    return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS) ||
(actual_mode == PrivilegeMode::VU);
}
```

E.122. exception_handling_mode

Returns the target privilege mode that will handle synchronous exception exception_code

Return Type	PrivilegeMode
Arguments	ExceptionCode exception_code

```
if (mode() == PrivilegeMode::M) {
    return PrivilegeMode::M;
} else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::HS) || (mode() == PrivilegeMode::U) {
    if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
        return PrivilegeMode::HS;
    } else {
        return PrivilegeMode::M;
    }
} else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU, "Unexpected
mode");
    if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
        if (($bits(hedeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
            return PrivilegeMode::VS;
        } else {
            return PrivilegeMode::HS;
        }
    } else {
        return PrivilegeMode::M;
    }
}
```

E.123. creg2reg

Maps a C register index (e.g., rs1' in the specification) to an X register index. From the specification:

Table 21. 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	Bits ⁵
Arguments	Bits<3> creg_idx

```
return {2'b01, creg_idx};
```

E.124. unimplemented_csr

Either raises an IllegalInstruction exception or enters unpredictable state, depending on the setting of the TRAP_ON_UNIMPLEMENTED_CSR parameter.

Return Type	void
Arguments	Bits<INSTR_ENC_SIZE> encoding

```
if (TRAP_ON_UNIMPLEMENTED_CSR) {
    raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else {
    unpredictable("Accessing an unimplmented CSR");
}
```

E.125. mtval_readonly?

Returns whether or not CSR[mtval] is read-only based on implementation options

Return Type	Boolean
Arguments	None

```
return !(REPORT_VA_IN_MTVAL_ON_BREAKPOINT || REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_MTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_MTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
```

E.126. stval_readonly?

Returns whether or not CSR[stval] is read-only based on implementation options

Return Type	Boolean
Arguments	None

```
if (implemented?(ExtensionName::S)) {
    return !(REPORT_VA_IN_STVAL_ON_BREAKPOINT || REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_STVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_STVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
    return true;
}
```

E.127. vstval_readonly?

Returns whether or not CSR[vstval] is read-only based on implementation options

Return Type	Boolean
Arguments	None

```
if (implemented?(ExtensionName::H)) {
    return !(REPORT_VA_IN_VSTVAL_ON_BREAKPOINT || REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_VSTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
```

```
REPORT_CAUSE_IN_VSTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
    return true;
}
```

E.128. mtval_for

Given an exception code and a **legal** non-zero value for mtval, returns the value to be written in mtval considering implementation options

Return Type	XReg
Arguments	ExceptionCode exception_code, XReg tval

```
if (exception_code == ExceptionCode::Breakpoint) {
    return REPORT_VA_IN_MTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
    return REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
    return REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
    return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
    return REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
    return REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
    return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
    return REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
    return REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
    return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
    return REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
    return tval;
} else {
    return 0;
}
```

E.129. stval_for

Given an exception code and a **legal** non-zero value for stval, returns the value to be written in stval considering implementation options

Return Type	XReg
Arguments	ExceptionCode exception_code, XReg tval

```
if (exception_code == ExceptionCode::Breakpoint) {
    return REPORT_VA_IN_STVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
    return REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
    return REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
    return REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
    return REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
    return REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
    return REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
```



```
    return REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
    return REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
    return REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
    return REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
    return tval;
} else {
    return 0;
}
```

E.130. vstval_for

Given an exception code and a **legal** non-zero value for vstval, returns the value to be written in vstval considering implementation options

Return Type	XReg
Arguments	ExceptionCode exception_code, XReg tval

```
if (exception_code == ExceptionCode::Breakpoint) {
    return REPORT_VA_IN_VSTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
    return REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
    return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
    return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
    return REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
    return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
    return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
    return REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
    return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
    return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
    return REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
    return tval;
} else {
    return 0;
}
```

E.131. raise_guest_page_fault

Raise a guest page fault exception.

Return Type	void
Arguments	MemoryOperation op, XReg gpa, XReg gva, XReg tinst_value, PrivilegeMode from_mode

```
ExceptionCode code;
Boolean write_gpa_in_tval;
if (op == MemoryOperation::Read) {
    code = ExceptionCode::LoadGuestPageFault;
    write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_LOAD_GUEST_PAGE_FAULT;
} else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    code = ExceptionCode::StoreAmoGuestPageFault;
```

```
write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_STORE_AMO_GUEST_PAGE_FAULT;
} else {
    assert(op == MemoryOperation::Fetch, "unexpected memory operation");
    code = ExceptionCode::InstructionGuestPageFault;
    write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_INSTRUCTION_GUEST_PAGE_FAULT;
}
PrivilegeMode handling_mode = exception_handling_mode(code);
if (handling_mode == PrivilegeMode::S) {
    htval.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
    htinst.VALUE = tinst_value;
    sepc.PC = $pc;
    if (!stval_readonly?()) {
        stval.VALUE = stval_for(code, gva);
    }
    $pc = {stvec.BASE, 2'b00};
    scause.INT = 1'b0;
    scause.CODE = $bits(code);
    hstatus.GVA = 1;
    hstatus.SPV = 1;
    hstatus.SPVP = $bits(from_mode)[0];
    mstatus.SPP = $bits(from_mode)[0];
} else {
    assert(handling_mode == PrivilegeMode::M, "unexpected privilege mode");
    mtval2.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
    mtinst.VALUE = tinst_value;
    mstatus.MPP = $bits(from_mode)[1:0];
    if (MXLEN == 64) {
        mstatus.MPV = 1;
    } else {
        mstatush.MPV = 1;
    }
}
set_mode(handling_mode);
abort_current_instruction();
```

E.132. raise

Raise synchronous exception number `exception_code`.

The exception may be imprecise, and will cause execution to enter an unpredictable state, if `PRECISE_SYNCHRONOUS_EXCEPTIONS` is false.

Otherwise, the exception will be precise.

Return Type	void
Arguments	ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval

```
if (!PRECISE_SYNCHRONOUS_EXCEPTIONS) {
    unpredictable("Imprecise synchronous exception");
} else {
    raise_precise(exception_code, from_mode, tval);
}
```

E.133. raise_precise

Raise synchronous exception number `exception_code`.

Return Type	void
Arguments	ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval

```
PrivilegeMode handling_mode = exception_handling_mode(exception_code);
if (handling_mode == PrivilegeMode::M) {
    mepc.PC = $pc;
    if (!mtval_readonly?()) {
```



```

    mtval.VALUE = mtval_for(exception_code, tval);
}
$pc = {mtvec.BASE, 2'b00};
mcause.INT = 1'b0;
mcause.CODE = $bits(exception_code);
if (misa.H == 1) {
    mtval2.VALUE = 0;
    mtinst.VALUE = 0;
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
        if (MXLEN == 32) {
            mstatush.MPV = 1;
        } else {
            mstatus.MPV = 1;
        }
    } else {
        if (MXLEN == 32) {
            mstatush.MPV = 0;
        } else {
            mstatus.MPV = 0;
        }
    }
}
mstatus.MPP = $bits(from_mode);
} else if (misa.S == 1 && (handling_mode == PrivilegeMode::S)) {
    sepc.PC = $pc;
    if (!stval_readonly?()) {
        stval.VALUE = stval_for(exception_code, tval);
    }
    $pc = {stvec.BASE, 2'b00};
    scause.INT = 1'b0;
    scause.CODE = $bits(exception_code);
    mstatus.SPP = $bits(from_mode)[0];
    if (misa.H == 1) {
        htval.VALUE = 0;
        htinst.VALUE = 0;
        hstatus.SPV = $bits(from_mode)[2];
        if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
            hstatus.SPV = 1;
            if (exception_code == ExceptionCode::Breakpoint) && (REPORT_VA_IN_STVAL_ON_BREAKPOINT || exception_code ==
ExceptionCode::LoadAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || exception_code ==
ExceptionCode::StoreAmoAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || exception_code ==
ExceptionCode::InstructionAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || exception_code ==
ExceptionCode::LoadAccessFault) && (REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || exception_code ==
ExceptionCode::StoreAmoAccessFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || exception_code ==
ExceptionCode::InstructionAccessFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || exception_code ==
ExceptionCode::LoadPageFault) && (REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || exception_code ==
ExceptionCode::StoreAmoPageFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || exception_code ==
ExceptionCode::InstructionPageFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT) {
                hstatus.GVA = 1;
            } else {
                hstatus.GVA = 0;
            }
            hstatus.SPVP = $bits(from_mode)[0];
        } else {
            hstatus.SPV = 0;
            hstatus.GVA = 0;
        }
    }
}
} else if (misa.H == 1 && (handling_mode == PrivilegeMode::VS)) {
    vsepc.PC = $pc;
    if (!vstval_readonly?()) {
        vstval.VALUE = vstval_for(exception_code, tval);
    }
    $pc = {vstvec.BASE, 2'b00};
    vscause.INT = 1'b0;
    vscause.CODE = $bits(exception_code);
    vsstatus.SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();

```

E.134. ialign

Returns IALIGN, the smallest instruction encoding size, in bits.

Return Type	Bits@
Arguments	None

```
if (implemented?(ExtensionName::C) && (misa.C == 0x1)) {
    return 16;
} else {
    return 32;
}
```

E.135. jump

Jump to virtual address target_addr.

If target address is misaligned, raise a MisalignedAddress exception.

Return Type	void
Arguments	XReg target_addr

```
if ((ialign() == 16) && target_addr & 0x1) != 0 {
    raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
} else if ((ialign() == 32) && (target_addr & 0x3) != 0) {
    raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
}
$pc = target_addr;
```

E.136. jump_halfword

Jump to virtual halfword address target_hw_addr.

If target address is misaligned, raise a MisalignedAddress exception.

Return Type	void
Arguments	XReg target_hw_addr

```
assert((target_hw_addr & 0x1) == 0x0, "Expected halfword-aligned address in jump_halfword");
if (ialign() != 16) {
    if ((target_hw_addr & 0x3) != 0) {
        raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_hw_addr);
    }
}
$pc = target_hw_addr;
```

E.137. valid_interrupt_code?

Returns true if code is a legal interrupt number.

Return Type	Boolean
Arguments	XReg code

```
if (code > 1 `<< $enum_element_size(InterruptCode - 1)) {
```

```
    return false;
}
if (ary_includes?<$enum_size(InterruptCode), $enum_element_size(InterruptCode)>($enum_to_a(InterruptCode), code)) {
    return true;
} else {
    return false;
}
```

E.138. valid_exception_code?

Returns true if *code* is a legal exception number.

Return Type	Boolean
Arguments	XReg code

```
if (code > 1 `<< $enum_element_size(ExceptionCode - 1)) {
    return false;
}
if (ary_includes?<$enum_size(InterruptCode), $enum_element_size(InterruptCode)>($enum_to_a(InterruptCode), code)) {
    return true;
} else {
    return false;
}
```

E.139. xlen

Returns the effective XLEN for the current privilege mode.

Return Type	Bits®
Arguments	None

```
if (MXLEN == 32) {
    return 32;
} else {
    if (mode() == PrivilegeMode::M) {
        if (misa.MXL == $bits(XRegWidth::XLEN32)) {
            return 32;
        } else if (misa.MXL == $bits(XRegWidth::XLEN64)) {
            return 64;
        }
    } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
        if (mstatus.SXL == $bits(XRegWidth::XLEN32)) {
            return 32;
        } else if (mstatus.SXL == $bits(XRegWidth::XLEN64)) {
            return 64;
        }
    } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
        if (mstatus.UXL == $bits(XRegWidth::XLEN32)) {
            return 32;
        } else if (mstatus.UXL == $bits(XRegWidth::XLEN64)) {
            return 64;
        }
    } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
        if (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
            return 32;
        } else if (hstatus.VSXL == $bits(XRegWidth::XLEN64)) {
            return 64;
        }
    } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU) {
        if (vsstatus.UXL == $bits(XRegWidth::XLEN32)) {
            return 32;
        } else if (vsstatus.UXL == $bits(XRegWidth::XLEN64)) {
            return 64;
        }
    }
}
```

```
}  
}
```

E.140. virtual_mode?

Returns True if the current mode is virtual (VS or VU).

Return Type	Boolean
Arguments	None

```
return (mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU);
```

E.141. mask_eaddr

Mask upper N bits of an effective address if pointer masking is enabled

Return Type	XReg
Arguments	XReg eaddr

```
return eaddr;
```

E.142. pmp_match_64

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	PmpMatchResult, PmpCfg
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size

```
Bits<12> pmpcfg0_addr = 0x3a0;  
Bits<12> pmpaddr0_addr = 0x3b0;  
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {  
    Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 8) * 2;  
    Bits<6> shamt = (i % 8) * 8;  
    Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);  
    PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];  
    Bits<12> pmpaddr_idx = pmpaddr0_addr + i;  
    Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);  
    Bits<64> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);  
    Bits<PHYS_ADDR_WIDTH> range_hi = 0;  
    Bits<PHYS_ADDR_WIDTH> range_lo = 0;  
    if (cfg.A == $bits(PmpCfg_A::TOR)) {  
        if (i == 0) {  
            range_lo = 0;  
        } else {  
            Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);  
            range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];  
        }  
        range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];  
    } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {  
        Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];  
        Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);  
        range_lo = (pmpaddr_value & ~mask) << 2;  
        Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;  
        range_hi = pmpaddr_value & ~mask + len << 2;    } else if (cfg.A == $bits(PmpCfg_A::NA4) {  
            range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];  
            range_hi = range_lo + 4;  
        }  
    }
```

```
    }
    if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
        return PmpMatchResult::FullMatch, cfg;
    } else if (! {
        return PmpMatchResult::PartialMatch, -;
    }
}
return PmpMatchResult::NoMatch, -;
```

E.143. pmp_match_32

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	PmpMatchResult, PmpCfg
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
    Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 4);
    Bits<6> shamt = (i % 4) * 8;
    Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
    PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
    Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
    Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
    Bits<32> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
    Bits<PHYS_ADDR_WIDTH> range_hi = 0;
    Bits<PHYS_ADDR_WIDTH> range_lo = 0;
    if (cfg.A == $bits(PmpCfg_A::TOR)) {
        if (i == 0) {
            range_lo = 0;
        } else {
            Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
            range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];
        }
        range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
    } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
        Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
        Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);
        range_lo = (pmpaddr_value & ~mask) << 2;
        Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;
        range_hi = pmpaddr_value & ~mask + len << 2;    } else if (cfg.A == $bits(PmpCfg_A::NA4) {
        range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
        range_hi = range_lo + 4;
    }
    if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
        return PmpMatchResult::FullMatch, cfg;
    } else if (! {
        return PmpMatchResult::PartialMatch, -;
    }
}
return PmpMatchResult::NoMatch, -;
```

E.144. pmp_match

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	PmpMatchResult, PmpCfg
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size

```
if (MXLEN == 64) {
    return pmp_match_64(paddr, access_size);
} else {
    return pmp_match_32(paddr, access_size);
}
```

E.145. mpv

Returns the current value of CSR[mstatus].MPV (when MXLEN == 64) of CSR[mstatush].MPV (when MXLEN == 32)

Return Type	Bits①
Arguments	None

```
if (implemented?(ExtensionName::H)) {
    return (MXLEN == 32) ? mstatush.MPV : mstatus.MPV;
} else {
    assert(false, "TODO");
}
```

E.146. effective_ldst_mode

Returns the effective privilege mode for normal explicit loads and stores, taking into account the current actual privilege mode and modifications from mstatus.MPRV.

Return Type	PrivilegeMode
Arguments	None

```
if (mode() == PrivilegeMode::M) {
    if (misa.U == 1 && mstatus.MPRV == 1) {
        if (mstatus.MPP == 0b00) {
            if (misa.H == 1 && mpv() == 0b1) {
                return PrivilegeMode::VU;
            } else {
                return PrivilegeMode::U;
            }
        }
        if (misa.S == 1 && mstatus.MPP == 0b01) {
            if (misa.H == 1 && mpv() == 0b1) {
                return PrivilegeMode::VS;
            } else {
                return PrivilegeMode::S;
            }
        }
    }
}
return mode();
```

E.147. pmp_check

Given a physical address and operation type, return whether or not the access is allowed by PMP.

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, MemoryOperation type

```
PrivilegeMode mode = effective_ldst_mode();
PmpMatchResult match_result;
PmpCfg cfg;
(match_result, cfg = pmp_match(paddr, access_size));
if (match_result == PmpMatchResult::FullMatch) {
    if (mode == PrivilegeMode::M && (cfg.L == 0)) {
```



```
    return true;
  }
  if (type == MemoryOperation::Write && (cfg.W == 0)) {
    return false;
  } else if (type == MemoryOperation::Read && (cfg.R == 0)) {
    return false;
  } else if (type == MemoryOperation::Fetch && (cfg.X == 0)) {
    return false;
  }
} else if (match_result == PmpMatchResult::NoMatch) {
  if (mode == PrivilegeMode::M) {
    return true;
  } else {
    return false;
  }
} else {
  assert(match_result == PmpMatchResult::PartialMatch, "PMP matching logic error");
  return false;
}
return true;
```

E.148. access_check

Checks if the physical address paddr is able to access memory, and raises the appropriate exception if not.

Return Type	void
Arguments	Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, XReg vaddr, MemoryOperation type, ExceptionCode fault_type, PrivilegeMode from_mode

```
if (paddr > 1 << PHYS_ADDR_WIDTH) - access_size {
  raise(fault_type, from_mode, vaddr);
}
if (implemented?(ExtensionName::Smpmp)) {
  if (!pmp_check(paddr[PHYS_ADDR_WIDTH - 1:0], access_size, type)) {
    raise(fault_type, from_mode, vaddr);
  }
}
```

E.149. base32?

return True iff current effective XLEN == 32

Return Type	Boolean
Arguments	None

```
if (MXLEN == 32) {
  return true;
} else {
  XRegWidth xlen32 = XRegWidth::XLEN32;
  if (mode() == PrivilegeMode::M) {
    return misa.MXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    return mstatus.SXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    return mstatus.UXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    return hstatus.VSXL == $bits(xlen32);
  } else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU, "Unexpected mode");
    return vsstatus.UXL == $bits(xlen32);
  }
}
```

E.150. base64?

return True iff current effective XLEN == 64

Return Type	Boolean
Arguments	None

```
return xlen() == 64;
```

E.151. current_translation_mode

Returns the current first-stage translation mode for an explicit load or store from mode given the machine state (e.g., value of `satp` or `vsatp` csr).

Returns SatpMode::Reserved if the setting found in `satp` or `vsatp` is invalid.

Return Type	SatpMode
Arguments	PrivilegeMode mode

```
PrivilegeMode effective_mode = effective_ldst_mode();
if (effective_mode == PrivilegeMode::M) {
    return SatpMode::Bare;
}
if (misa.H == 1'b1) {
    if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU) {
        Bits<4> mode_val = vsatp.MODE;
        if (mode_val == $bits(SatpMode::Sv32)) {
            if (MXLEN == 64) {
                if ((effective_mode == PrivilegeMode::VS) && (hstatus.VSXL != $bits(XRegWidth::XLEN32))) {
                    return SatpMode::Reserved;
                }
                if ((effective_mode == PrivilegeMode::VU) && (vsstatus.UXL != $bits(XRegWidth::XLEN32))) {
                    return SatpMode::Reserved;
                }
            }
            if (!SV32_VSMODE_TRANSLATION) {
                return SatpMode::Reserved;
            }
            return SatpMode::Sv32;
        } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
            if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
                return SatpMode::Reserved;
            }
            if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
                return SatpMode::Reserved;
            }
        }
        if (!SV39_VSMODE_TRANSLATION) {
            return SatpMode::Reserved;
        }
        return SatpMode::Sv39;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
        if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
    }
    if (!SV48_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
    }
    return SatpMode::Sv48;
} else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
    if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
    }
}
```



```

    }
    if (!SV57_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
    }
    return SatpMode::Sv57;
} else {
    return SatpMode::Reserved;
}
}
} else if (misa.S == 1'b1) {
    assert(effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U, "unexpected priv mode");
    Bits<4> mode_val = satp.MODE;
    if (mode_val == $bits(SatpMode::Sv32)) {
        if (MXLEN == 64) {
            if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN32)) {
                return SatpMode::Reserved;
            }
            if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN32)) {
                return SatpMode::Reserved;
            }
        }
        if (!implemented?(ExtensionName::Sv32)) {
            return SatpMode::Reserved;
        }
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
        if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (!implemented?(ExtensionName::Sv39)) {
            return SatpMode::Reserved;
        }
        return SatpMode::Sv39;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
        if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (!implemented?(ExtensionName::Sv48)) {
            return SatpMode::Reserved;
        }
        return SatpMode::Sv48;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
        if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
            return SatpMode::Reserved;
        }
        if (!implemented?(ExtensionName::Sv57)) {
            return SatpMode::Reserved;
        }
        return SatpMode::Sv57;
    } else {
        return SatpMode::Reserved;
    }
}
}
```

E.152. current_gstage_translation_mode

Returns the current second-stage translation mode for a load or store from VS-mode or VU-mode.

Return Type	HgatpMode
Arguments	None

```
return $enum(HgatpMode, hgatp.MODE);
```

E.153. translate_gstage

Translates a guest physical address to a physical address.

Return Type	TranslationResult
Arguments	XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding

```
TranslationResult result;
if (effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U) {
    result.paddr = gpaddr;
    return result;
}
Boolean mxr = mstatus.MXR == 1;
if (GSTAGE_MODE_BARE && hcatp.MODE == $bits(HcatpMode::Bare)) {
    result.paddr = gpaddr;
    return result;
} else if (SV32X4_TRANSLATION && hcatp.MODE == $bits(HcatpMode::Sv32x4)) {
    return gstage_page_walk<32, 34, 32, 2>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV39X4_TRANSLATION && hcatp.MODE == $bits(HcatpMode::Sv39x4)) {
    return gstage_page_walk<39, 56, 64, 3>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV48X4_TRANSLATION && hcatp.MODE == $bits(HcatpMode::Sv48x4)) {
    return gstage_page_walk<48, 56, 64, 4>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV57X4_TRANSLATION && hcatp.MODE == $bits(HcatpMode::Sv57x4)) {
    return gstage_page_walk<57, 56, 64, 5>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else {
    if (op == MemoryOperation::Read) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
    } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
    } else {
        assert(op == MemoryOperation::Fetch, "unexpected memory op");
        raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
    }
}
```

E.154. tinst_value_for_guest_page_fault

Returns the value of htinst/mtinst for a Guest Page Fault

Return Type	XReg
Arguments	MemoryOperation op, Bits<INSTR_ENC_SIZE> encoding, Boolean for_final_vs_pte

```
if (for_final_vs_pte) {
    if (op == MemoryOperation::Fetch) {
        if (TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always zero") {
            return 0;
        } else {
            assert(TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always pseudoinstruction", "Instruction guest page faults can only report zero/pseudo instruction in tval");
            return 0x00002000;
        }
    } else if (op == MemoryOperation::Read) {
        if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always zero") {
            return 0;
        } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always pseudoinstruction") {
            if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
                return 0x00002000;
            } else {
                return 0x00003000;
            }
        } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always transformed standard instruction") {
```

```
    return tinst_transform(encoding, 0);
  } else {
    unpredictable("Custom value written into htinst/mtinst");
  }
} else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
  if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always pseudoinstruction") {
    if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
      return 0x00002020;
    } else {
      return 0x00003020;
    }
  } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always transformed standard instruction") {
    return tinst_transform(encoding, 0);
  } else {
    unpredictable("Custom value written into htinst/mtinst");
  }
}
} else {
  if (REPORT_GPA_IN_TVAL_ON_INTERMEDIATE_GUEST_PAGE_FAULT) {
    if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
      return 0x00002000;
    } else if ((VSXLEN == 64) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN64)) {
      return 0x00003000;
    }
  }
}
}
```

E.155. tinst_transform

Returns the standard transformation of an encoding for htinst/mtinst

Return Type	Bits<INSTR_ENC_SIZE>
Arguments	Bits<INSTR_ENC_SIZE> encoding, Bits<5> addr_offset

```
if (encoding[1:0] == 0b11) {
  if (encoding[6:2] == 5'b00001) {
    return {{12{1'b0}}, addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b01000) {
    return {{7{1'b0}}, encoding[24:20], addr_offset, encoding[14:12], {5{1'b0}}, encoding[6:0]};
  } else if (encoding[6:2] == 5'b01011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b00011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else {
    assert(false, "Bad transform");
  }
} else {
  assert(false, "TODO: compressed instruction");
}
```

E.156. transformed_standard_instruction_for_tinst

Transforms an instruction encoding for htinst.

Return Type	Bits<INSTR_ENC_SIZE>
Arguments	Bits<INSTR_ENC_SIZE> original

```
assert(false, "TODO");
return 0;
```

E.157. tinst_value

Returns the value of htinst/mtinst for the given exception code.

Return Type	XReg
Arguments	ExceptionCode code, Bits<INSTR_ENC_SIZE> encoding

```
if (code == ExceptionCode::InstructionAddressMisaligned) {
    if (TINST_VALUE_ON_INSTRUCTION_ADDRESS_MISALIGNED == "always zero") {
        return 0;
    } else {
        unpredictable("An unpredictable value is written into tinst in response to an InstructionAddressMisaligned
exception");
    }
} else if (code == ExceptionCode::InstructionAccessFault) {
    return 0;
} else if (code == ExceptionCode::IllegalInstruction) {
    return 0;
} else if (code == ExceptionCode::Breakpoint) {
    if (TINST_VALUE_ON_BREAKPOINT == "always zero") {
        return 0;
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a Breakpoint exception");
    }
} else if (code == ExceptionCode::VirtualInstruction) {
    if (TINST_VALUE_ON_VIRTUAL_INSTRUCTION == "always zero") {
        return 0;
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a VirtualInstruction exception");
    }
} else if (code == ExceptionCode::LoadAddressMisaligned) {
    if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always zero") {
        return 0;
    } else if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always transformed standard instruction") {
        return transformed_standard_instruction_for_tinst(encoding);
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a LoadAddressMisaligned exception");
    }
} else if (code == ExceptionCode::LoadAccessFault) {
    if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always zero") {
        return 0;
    } else if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always transformed standard instruction") {
        return transformed_standard_instruction_for_tinst(encoding);
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a LoadAccessFault exception");
    }
} else if (code == ExceptionCode::StoreAmoAddressMisaligned) {
    if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always zero") {
        return 0;
    } else if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always transformed standard instruction") {
        return transformed_standard_instruction_for_tinst(encoding);
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAddressMisaligned exception");
    }
} else if (code == ExceptionCode::StoreAmoAccessFault) {
    if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always zero") {
        return 0;
    } else if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always transformed standard instruction") {
        return transformed_standard_instruction_for_tinst(encoding);
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAccessFault exception");
    }
} else if (code == ExceptionCode::Ucall) {
    if (TINST_VALUE_ON_UCALL == "always zero") {
        return 0;
    } else {
        unpredictable("An unpredictable value is written into tinst in response to a UCall exception");
    }
} else if (code == ExceptionCode::Scall) {
    if (TINST_VALUE_ON_SCALL == "always zero") {
```

```
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a SCall exception");
  }
} else if (code == ExceptionCode::Mcall) {
  if (TINST_VALUE_ON_MCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a MCall exception");
  }
} else if (code == ExceptionCode::VScall) {
  if (TINST_VALUE_ON_VSCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VSCall exception");
  }
} else if (code == ExceptionCode::InstructionPageFault) {
  return 0;
} else if (code == ExceptionCode::LoadPageFault) {
  if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadPageFault exception");
  }
} else if (code == ExceptionCode::StoreAmoPageFault) {
  if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoPageFault exception");
  }
} else {
  assert(false, "Unhandled exception type");
}
```

E.158. gstage_page_walk

Translate guest physical address to physical address through a page walk.

May raise a Guest Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated physical address.

Return Type	TranslationResult
Arguments	XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Boolean for_final_vs_pte, Bits<INSTR_ENC_SIZE> encoding

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadGuestPageFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionGuestPageFault : ExceptionCode::StoreAmoGuestPageFault);
Boolean mxr = for_final_vs_pte && (mstatus.MXR == 1);
Boolean pbmte = menvcfg.PBMTE == 1;
Boolean adue = menvcfg.ADUE == 1;
Bits<32> tinst = tinst_value_for_guest_page_fault(op, encoding, for_final_vs_pte);
U32 max_gpa_width = LEVELS * VPN_SIZE + 2 + 12;
if (gpaddr >> max_gpa_width != 0) {
  raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
}
```

```

}
ppn = hgap.PPN;
for (U32 i = (LEVELS - 1); i >= 0; i--) {
    U32 this_vpn_size = (i == (LEVELS - 1)) ? VPN_SIZE + 2 : VPN_SIZE;
    U32 vpn = (gpaddr >> (12 + VPN_SIZE * i)) & 1 << this_vpn_size - 1;    Bits<PA_SIZE> pte_paddr = (ppn << 12) + (vpn
* (PTESIZE / 8);
    if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_paddr, PTESIZE)) {
        raise(access_fault_code, PrivilegeMode::U, vaddr);
    }
    access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
    Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_paddr);
    PteFlags pte_flags = pte[9:0];
    if ((VA_SIZE != 32) && (pte[60:54] != 0)) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (!implemented?(ExtensionName::Svnapot)) {
        if ((PTESIZE >= 64) && pte[63] != 0) {
            raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
        }
    }
    if ((PTESIZE >= 64) && !pbmte && (pte[62:61] != 0)) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((PTESIZE >= 64) && pbmte && (pte[62:61] == 3)) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.V == 0) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.R == 0 && pte_flags.W == 1) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.R == 1 || pte_flags.X == 1) {
        if (pte_flags.U == 0) {
            raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
        }
        if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
            raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
        } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
            raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
        } else if ((op == MemoryOperation::Read) || (op == MemoryOperation::ReadModifyWrite)) {
            if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
                raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
            }
        }
    }
    if ((i > 0) && (pte[(i - 1) * VPN_SIZE:10] != 0)) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op ==
MemoryOperation::ReadModifyWrite)) {
        if (adue) {
            if (!pma_applies?(PmaAttribute::RsrvEventual, pte_paddr, PTESIZE)) {
                raise(access_fault_code, PrivilegeMode::U, vaddr);
            }
            if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_paddr, PTESIZE)) {
                raise(access_fault_code, PrivilegeMode::U, vaddr);
            }
            access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
            Boolean success;
            Bits<PTESIZE> updated_pte;
            if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
                updated_pte = pte | 0b11000000;
            } else {
                updated_pte = pte | 0b01000000;
            }
            if (PTESIZE == 32) {
                success = atomic_check_then_write_32(pte_paddr, pte, updated_pte);
            } else if (PTESIZE == 64) {
                success = atomic_check_then_write_64(pte_paddr, pte, updated_pte);
            } else {
                assert(false, "Unexpected PTESIZE");
            }
            if (!success) {
                i = i + 1;
            } else {

```



```
        result.paddr = pte_paddr;
        if (PTESIZE >= 64) {
            result.pbmt = $enum(Pbmt, pte[62:61]);
        }
        result.pte_flags = pte_flags;
        return result;
    }
} else {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
}
}
} else {
    if (i == 0) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
}
}
```

E.159. stage1_page_walk

Translate virtual address to physical address through a page walk.

May raise a Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated guest physical address.

Return Type	TranslationResult
Arguments	Bits<MXLEN> vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadPageFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionPageFault : ExceptionCode::StoreAmoPageFault);
Boolean sse = false;
Boolean adue;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    adue = henvcfg.ADUE == 1;
} else {
    adue = menvcfg.ADUE == 1;
}
Boolean pbmte;
if (VA_SIZE == 32) {
    pbmte = false;
} else {
    if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
        pbmte = henvcfg.PBMTE == 1;
    } else {
        pbmte = menvcfg.PBMTE == 1;
    }
}
```

```

Boolean mxr;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    mxr = (mstatus.MXR == 1) || (vsstatus.MXR == 1);
} else {
    mxr = mstatus.MXR == 1;
}
Boolean sum;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS)) {
    sum = vsstatus.SUM == 1;
} else {
    sum = mstatus.SUM == 1;
}
ppn = vsatp.PPN;
if ((VA_SIZE < xlen()) && (vaddr[xlen() - 1:VA_SIZE] != {xlen() - VA_SIZE{vaddr[VA_SIZE - 1]}})) {
    raise(page_fault_code, effective_mode, vaddr);
}
for (U32 i = (LEVELS - 1); i >= 0; i--) {
    U32 vpn = (vaddr >> (12 + VPN_SIZE * i)) & 1 << VPN_SIZE - 1;    Bits<PA_SIZE> pte_gpaddr = (ppn << 12) + (vpn *
(PTESIZE / 8);
    TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Read, effective_mode, encoding);
    if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_phys.paddr, PTESIZE)) {
        raise(access_fault_code, effective_mode, vaddr);
    }
    access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
    Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_phys.paddr);
    PteFlags pte_flags = pte[9:0];
    Boolean ss_page = (pte_flags.R == 0) && (pte_flags.W == 1) && (pte_flags.X == 0);
    if ((VA_SIZE != 32) && (pte[60:54] != 0)) {
        raise(page_fault_code, effective_mode, vaddr);
    }
    if (pte_flags.V == 0) {
        raise(page_fault_code, effective_mode, vaddr);
    }
    if (!sse) {
        if ((pte_flags.R == 0) && (pte_flags.W == 1)) {
            raise(page_fault_code, effective_mode, vaddr);
        }
    }
    if (pbmte) {
        if (pte[62:61] == 3) {
            raise(page_fault_code, effective_mode, vaddr);
        }
    } else {
        if ((PTESIZE >= 64) && (pte[62:61] != 0)) {
            raise(page_fault_code, effective_mode, vaddr);
        }
    }
    if (!implemented?(ExtensionName::Svnapot)) {
        if ((PTESIZE >= 64) && (pte[63] != 0)) {
            raise(page_fault_code, effective_mode, vaddr);
        }
    }
    if (pte_flags.R == 1 || pte_flags.X == 1) {
        if (op == MemoryOperation::Read || op == MemoryOperation::ReadModifyWrite) {
            if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
                raise(page_fault_code, effective_mode, vaddr);
            }
            if (effective_mode == PrivilegeMode::U && pte_flags.U == 0) {
                raise(page_fault_code, effective_mode, vaddr);
            } else if (misa.H == 1 && effective_mode == PrivilegeMode::VU && pte_flags.U == 0) {
                raise(page_fault_code, effective_mode, vaddr);
            } else if (effective_mode == PrivilegeMode::S && pte_flags.U == 1 && !sum) {
                raise(page_fault_code, effective_mode, vaddr);
            } else if (effective_mode == PrivilegeMode::VS && pte_flags.U == 1 && !sum) {
                raise(page_fault_code, effective_mode, vaddr);
            }
        }
        if (op == MemoryOperation::Write || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0))) {
            raise(page_fault_code, effective_mode, vaddr);
        } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
            raise(page_fault_code, effective_mode, vaddr);
        } else if ((op == MemoryOperation::Fetch) && ss_page) {
            raise(page_fault_code, effective_mode, vaddr);
        }
        raise(page_fault_code, effective_mode, vaddr) if;
    }
}

```



```

    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op ==
MemoryOperation::ReadModifyWrite)) {
        if (adue) {
            TranslationResult pte_phys = translate\_gstage(pte_gpaddr, vaddr, MemoryOperation::Write, effective_mode,
encoding);
            if (!pma\_applies?(PmaAttribute::RsrvEventual, pte_phys.paddr, PTESIZE)) {
                raise(access_fault_code, effective_mode, vaddr);
            }
            if (!pma\_applies?(PmaAttribute::HardwarePageTableWrite, pte_phys.paddr, PTESIZE)) {
                raise(access_fault_code, effective_mode, vaddr);
            }
            access\_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
            Boolean success;
            Bits<PTESIZE> updated_pte;
            if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
                updated_pte = pte | 0b11000000;
            } else {
                updated_pte = pte | 0b01000000;
            }
            if (PTESIZE == 32) {
                success = atomic\_check\_then\_write\_32(pte_phys.paddr, pte, updated_pte);
            } else if (PTESIZE == 64) {
                success = atomic\_check\_then\_write\_64(pte_phys.paddr, pte, updated_pte);
            } else {
                assert(false, "Unexpected PTESIZE");
            }
            if (!success) {
                i = i + 1;
            } else {
                TranslationResult pte_phys = translate\_gstage(({pte[PA_SIZE - 3:(i * VPN_SIZE) + 10] << 2}, vaddr[11:0]}),
vaddr, op, effective_mode, encoding);
                result.paddr = pte_phys.paddr;
                result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
                result.pte_flags = pte_flags;
                return result;
            }
        } else {
            raise(page_fault_code, effective_mode, vaddr);
        }
    }
    TranslationResult pte_phys = translate\_gstage(({pte[PA_SIZE - 3:(i * VPN_SIZE) + 10] << 2}, vaddr[11:0]}), vaddr, op,
effective_mode, encoding);
    result.paddr = pte_phys.paddr;
    if (PTESIZE >= 64) {
        result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
    }
    result.pte_flags = pte_flags;
    return result;
} else {
    if (i == 0) {
        raise(page_fault_code, effective_mode, vaddr);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
        raise(page_fault_code, effective_mode, vaddr);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
        raise(page_fault_code, effective_mode, vaddr);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
        raise(page_fault_code, effective_mode, vaddr);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
}
}

```

E.160. translate

Translate a virtual address for operation type `op` that appears to execute at `effective_mode`.

The translation will depend on the effective privilege mode.

May raise a Page Fault or Access Fault.

The final physical address is **not** access checked (for PMP, PMA, etc., violations). (though intermediate page table reads will be)

Return Type	TranslationResult
Arguments	XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding

```
Boolean cached_translation_valid;
CachedTranslationResult cached_translation_result;
cached_translation_result = cached_translation(vaddr, op);
if (cached_translation_result.valid) {
    return cached_translation_result.result;
}
TranslationResult result;
if (effective_mode == PrivilegeMode::M) {
    result.paddr = vaddr;
    return result;
}
SatpMode translation_mode = current_translation_mode(effective_mode);
if (translation_mode == SatpMode::Reserved) {
    if (op == MemoryOperation::Read) {
        raise(ExceptionCode::LoadPageFault, effective_mode, vaddr);
    } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
        raise(ExceptionCode::StoreAmoPageFault, effective_mode, vaddr);
    } else {
        assert(op == MemoryOperation::Fetch, "Unexpected memory operation");
        raise(ExceptionCode::InstructionPageFault, effective_mode, vaddr);
    }
}
if (translation_mode == SatpMode::Sv32) {
    result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv39) {
    result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv48) {
    result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv57) {
    result = stage1_page_walk<57, 56, 64, 5>(vaddr, op, effective_mode, encoding);
} else {
    assert(false, "Unexpected SatpMode");
}
maybe_cache_translation(vaddr, op, result);
return result;
```

E.161. canonical_vaddr?

Returns whether or not *vaddr* is a valid (*i.e.*, canonical) virtual address.

If pointer masking (*S**pm*) is enabled, then *vaddr* will be masked before checking the canonical address.

Return Type	Boolean
Arguments	XReg vaddr

```
if (misa.S == 1'b0) {
    return true;
}
SatpMode satp_mode;
if (virtual_mode?()) {
    satp_mode = $enum(SatpMode, vsatp.MODE);
} else {
    satp_mode = $enum(SatpMode, satp.MODE);
}
XReg eaddr = mask_eaddr(vaddr);
if (satp_mode == SatpMode::Bare) {
    return true;
} else if (satp_mode == SatpMode::Sv32) {
    return true;
} else if (satp_mode == SatpMode::Sv39) {
```

```
return eaddr[63:39] == {25{eaddr[38]}};
} else if (satp_mode == SatpMode::Sv48) {
    return eaddr[63:48] == {16{eaddr[47]}};
} else if (satp_mode == SatpMode::Sv57) {
    return eaddr[63:57] == {6{eaddr[56]}};
}
```

E.162. canonical_gpaddr?

Returns whether or not gpaddr is a valid (i.e., canonical) guest physical address.

Return Type	Boolean
Arguments	XReg gpaddr

```
SatpMode satp_mode = $enum(SatpMode, satp.MODE);
if (satp_mode == SatpMode::Bare) {
    return true;
} else if (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.163. misaligned_is_atomic?

Returns true if an access starting at physical_address that is N bits long is atomic.

This function takes into account any Atomicity Granule PMAs, so **it should not be used for load-reserved/store-conditional**, since those PMAs do not apply to those accesses.

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> physical_address

```
return false if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE == 0);
if (pma_applies?(PmaAttribute::MAG16, physical_address, N) && in_naturally_aligned_region?<128>(physical_address, N)) {
    return true;
} else if (pma_applies?(PmaAttribute::MAG8, physical_address, N) && in_naturally_aligned_region?<4>(physical_address, N)) {
    return true;
} else if (pma_applies?(PmaAttribute::MAG4, physical_address, N) && in_naturally_aligned_region?<32>(physical_address, N)) {
    return true;
} else if (pma_applies?(PmaAttribute::MAG2, physical_address, N) && in_naturally_aligned_region?<16>(physical_address, N)) {
    return true;
} else {
    return false;
}
```

E.164. read_memory_aligned

Read from virtual memory using a known aligned address.

Return Type	Bits<LEN>

Arguments	XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding

```
TranslationResult result;
if (misa.S == 1) {
    result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
    result.paddr = virtual_address;
}
access_check(result.paddr, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
return read_physical_memory<LEN>(result.paddr);
```

E.165. read_memory

Read from virtual memory.

Return Type	Bits<LEN>
Arguments	XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
    return read_memory_aligned<LEN>(virtual_address, encoding);
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
    assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with
large atomicity granule");
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
    if (misaligned_is_atomic?<LEN>(physical_address)) {
        access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
        return read_physical_memory<LEN>(physical_address);
    }
}
if (!MISALIGNED_LDST) {
    if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
        physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
        access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
    }
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
} else {
    if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
        Bits<LEN> result = 0;
        for (U32 i = 0; i <= (LEN / 8); i++) {
            result = result | (read_memory_aligned<8>(virtual_address + i, encoding) << (8 * i));
        }
        return result;
    } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
        unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior
when any part of the misaligned access causes an exception");
    }
}
```

E.166. read_memory_xlen

Read XLEN bits from memory

Return Type	Bits<MXLEN>

Arguments	XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding

```
if (xlen() == 32) {
    return read_memory<32>(virtual_address, encoding);
} else {
    return read_memory<64>(virtual_address, encoding);
}
```

E.167. write_memory_xlen

Read XLEN bits from memory

Return Type	void
Arguments	XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding

```
if (xlen() == 32) {
    return write_memory<32>(virtual_address, value, encoding);
} else {
    return write_memory<64>(virtual_address, value, encoding);
}
```

E.168. read_memory_xlen_aligned

Read from virtual memory XLEN (which may be runtime-determined) bits using a known aligned address.

Return Type	Bits<MXLEN>
Arguments	XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding

```
TranslationResult result;
if (misa.S == 1) {
    result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
    result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault, effective_ldst_mode());
if (xlen() == 32) {
    return read_physical_memory<32>(result.paddr);
} else {
    return read_physical_memory<64>(result.paddr);
}
```

E.169. invalidate_reservation_set

Invalidates any currently held reservation set.



This function may be called by the platform, independent of any actions occurring in the local hart, for any or no reason.

*The platform **must** call this function if an external hart or device accesses part of this reservation set while reservation_set_valid could be true.*

Return Type	void
Arguments	None

```
reservation_set_valid = false;
```

E.170. register_reservation_set

Register a reservation for a physical address range that subsumes [physical_address, physical_address + N).

Return Type	void
Arguments	Bits<MXLEN> physical_address, Bits<MXLEN> length

```
reservation_set_valid = true;
reservation_set_address = physical_address;
if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 64-byte region") {
    reservation_set_address = physical_address & ~MXLEN'h3f;
    reservation_set_size = 64;
} else if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 128-byte region") {
    reservation_set_address = physical_address & ~MXLEN'h7f;
    reservation_set_size = 128;
} else if (LRSC_RESERVATION_STRATEGY == "reserve exactly enough to cover the access") {
    reservation_set_address = physical_address;
    reservation_set_size = length;
} else if (LRSC_RESERVATION_STRATEGY == "custom") {
    unpredictable("Implementations may set reservation sets of any size, as long as they cover the reserved accessed");
} else {
    assert(false, "Unexpected LRSC_RESERVATION_STRATEGY");
}
```

E.171. load_reserved

Register a reservation for virtual_address at least N bits long and read the value from memory.

If aq is set, then also perform a memory model acquire.

If rl is set, then also perform a memory model release (software is discouraged from doing so).

This function assumes alignment checks have already occurred.

Return Type	Bits<N>
Arguments	Bits<MXLEN> virtual_address, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
}
if (aq == 1) {
    memory_model_acquire();
}
if (rl == 1) {
    memory_model_release();
}
register_reservation_set(physical_address, N);
if (misa.S == 1 && LRSC_FAIL_ON_VA_SYNONYM) {
    reservation_virtual_address = virtual_address;
}
return read_memory_aligned<N>(physical_address, encoding);
```

E.172. store_conditional

Atomically check the reservation set to ensure:

- it is valid
- it covers the region addressed by this store
- the address setting the reservation set matches virtual address

If the preceding are met, perform the store and return 0. Otherwise, return 1.

Return Type	Boolean
Arguments	Bits<MXLEN> virtual_address, Bits<MXLEN> value, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
    raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
}
access_check(physical_address, N, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (aq == 1) {
    memory_model_acquire();
}
if (rl == 1) {
    memory_model_release();
}
if (reservation_set_valid == false) {
    return false;
}
if (!contains?(reservation_set_address, reservation_set_size, physical_address, N)) {
    invalidate_reservation_set();
    return false;
}
if (LRSC_FAIL_ON_NON_EXACT_LRSC) {
    if (reservation_physical_address != physical_address || reservation_size != N) {
        invalidate_reservation_set();
        return false;
    }
}
if (LRSC_FAIL_ON_VA_SYNONYM) {
    if (reservation_virtual_address != virtual_address || reservation_size != N) {
        invalidate_reservation_set();
        return false;
    }
}
write_physical_memory<N>(physical_address, value);
return true;
```

E.173. amo

Atomically read-modify-write the location at virtual_address.

The value written to virtual_address will depend on op.

If aq is 1, then the amo also acts as a memory model acquire. If rl is 1, then the amo also acts as a memory model release.

Return Type	Bits<N>
Arguments	XReg virtual_address, Bits<N> value, AmoOperation op, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<N>(virtual_address);
if (!aligned && MISALIGNED_LDST_EXCEPTION_PRIORITY == "high") {
    raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
}
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::ReadModifyWrite,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::AmoNone, physical_address, N)) {
    raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
} else if (op == AmoOperation::Add || op == AmoOperation::Max || op == AmoOperation::Maxu || op == AmoOperation::Min ||
op == AmoOperation::Minu && !pma_applies?(PmaAttribute::AmoArithmetic, physical_address, N)) {
    raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
} else if (op == AmoOperation::And || op == AmoOperation::Or || op == AmoOperation::Xor && !
```

```
pma_applies?(PmaAttribute::AmoLogical, physical_address, N)) {
    raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
} else {
    assert(pma_applies?(PmaAttribute::AmoSwap, physical_address, N) && op == AmoOperation::Swap, "Bad AMO operation");
}
if (!aligned && !misaligned_is_atomic<N>(physical_address)) {
    raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
}
if (N == 32) {
    return atomic_read_modify_write_32(physical_address, value, op);
} else {
    return atomic_read_modify_write_64(physical_address, value, op);
}
```

E.174. write_memory_aligned

Write to virtual memory using a known aligned address.

Return Type	void
Arguments	XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
write_physical_memory<LEN>(physical_address, value);
```

E.175. write_memory

Write to virtual memory

Return Type	void
Arguments	XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
    write_memory_aligned<LEN>(virtual_address, value, encoding);
    return ;
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
    assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with
large atomicity granule");
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
    if (misaligned_is_atomic<LEN>(physical_address)) {
        access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
        write_physical_memory<LEN>(physical_address, value);
        return ;
    }
}
if (!MISALIGNED_LDST) {
    if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
        physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
        access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
    }
    raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
} else {
```



```
if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    for (U32 i = 0; i <= (LEN / 8); i++) {
        write_memory_aligned<8>(virtual_address + i, (value >> (8 * i))[7:0], encoding);
    }
} else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior
when any part of the misaligned access causes an exception");
}
}
```

E.176. write_memory_xlen_aligned

Write to virtual memory XLEN bits (which may be runtime determined) using a known aligned address.

Return Type	void
Arguments	XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
access_check(physical_address, xlen(), virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
    write_physical_memory<32>(physical_address, value);
} else {
    write_physical_memory<64>(physical_address, value);
}
```

E.177. mstatus_sd_has_known_reset

Returns true if the mstatus.SD bit has a defined reset value, as determined by various parameters.

Return Type	Boolean
Arguments	None

```
Boolean fs_has_single_value = !implemented?(ExtensionName::F || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1));
Boolean vs_has_single_value = !implemented?(ExtensionName::V || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1));
return fs_has_single_value && vs_has_single_value;
```

E.178. mstatus_sd_reset_value

Returns the reset value of mstatus.SD when known

Return Type	Bits①
Arguments	None

```
assert(mstatus_sd_has_known_reset(), "mstatus_sd_reset_value is only defined when mstatus_sd_has_known_reset() ==
true");
Bits<2> fs_value, vs_value;
if ((!implemented?(ExtensionName::F)) || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1)) {
    fs_value = (!implemented?(ExtensionName::F)) ? 0 : MSTATUS_FS_LEGAL_VALUES[0];
}
if ((!implemented?(ExtensionName::V)) || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1)) {
    fs_value = (!implemented?(ExtensionName::V)) ? 0 : MSTATUS_VS_LEGAL_VALUES[0];
}
return fs_value == 3) || (vs_value == 3 ? 1 : 0;
```