The RISC-V Instruction Set Manual: Volume II

Privileged Architecture

Version 20241017: This document is in Ratified state.

Table of Contents

Preamble

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Krste Asanović, Peter Ashenden, Rimas Avižienis, Jacob Bachmeyer, Allen J. Baum, Jonathan Behrens, Paolo Bonzini, Ruslan Bukin, Christopher Celio, Chuanhua Chang, David Chisnall, Anthony Coulter, Palmer Dabbelt, Monte Dalrymple, Paul Donahue, Greg Favor, Dennis Ferguson, Marc Gauthier, Andy Glew, Gary Guo, Mike Frysinger, John Hauser, David Horner, Olof Johansson, David Kruckemyer, Yunsup Lee, Daniel Lustig, Andrew Lutomirski, Martin Maas, Prashanth Mundkur, Jonathan Neuschäfer, Rishiyur Nikhil, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Dmitri Pavlov, Kade Phillips, Josh Scheid, Colin Schmidt, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Ray VanDeWalker, Megan Wachs, Steve Wallach, Andrew Waterman, Claire Wolf, Adam Zabrocki, and Reinoud Zandijk..

Preface

Preface to Version 20241017

This document describes the RISC-V privileged architecture. This release, version 20241017, contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	1.13	Ratified
Smstateen Extension	1.0	Ratified
Smcsrind/Sscsrind Extension	1.0	Ratified
Smepmp	1.0	Ratified
Smcntrpmf	1.0	Ratified
Smrnmi Extension	1.0	Ratified
Smcdeleg	1.0	Ratified
Smdbltrp	1.0	Ratified
Supervisor ISA	1.13	Ratified
Svade Extension	1.0	Ratified
Svnapot Extension	1.0	Ratified
Svpbmt Extension	1.0	Ratified
Svinval Extension	1.0	Ratified
Svadu Extension	1.0	Ratified
Sstc	1.0	Ratified
Sscofpmf	1.0	Ratified
Ssdbltrp	1.0	Ratified
Hypervisor ISA	1.0	Ratified
Shlcofideleg	1.0	Ratified
Svvptc	1.0	Ratified

The following changes have been made since version 1.12 of the Machine and Supervisor ISAs, which, while not strictly backwards compatible, are not anticipated to cause software portability problems in practice:

- Redefined `misa`.MXL to be read-only, making MXLEN a constant.
- Added the constraint that SXLEN≥UXLEN.

Additionally, the following compatible changes have been made to the Machine and Supervisor ISAs since version 1.12:

- Defined the `misa`.B field to reflect that the B extension has been implemented.
- Defined the `misa`.V field to reflect that the V extension has been implemented.
- Defined the RV32-only `medelegh` and `hedelegh` CSRs.
- Defined the misaligned atomicity granule PMA, superseding the proposed Zam extension.
- Allocated interrupt 13 for Sscofpmf LCOFI interrupt.
- Defined hardware error and software check exception codes.
- Specified synchronization requirements when changing the PBMTE fields in `menvcfg` and `henvcfg`.
- Exposed count-overflow interrupts to VS-mode via the Shlcofideleg extension.
- Relaxed behavior of some HINTs when MXLEN > XLEN.

Finally, the following clarifications and document improvements have been made since the last document release:

- Transliterated the document from LaTeX into AsciiDoc.

- Included all ratified extensions through March 2024.
- Clarified that "platform- or custom-use" interrupts are actually "platform-use interrupts", where the platform can choose to make some custom.
- Clarified semantics of explicit accesses to CSRs wider than XLEN bits.
- Clarified that MXLEN≥SXLEN.
- Clarified that WFI is not a HINT instruction.
- Clarified that VS-stage page-table accesses set G-stage A/D bits.
- Clarified ordering rules when PBMT=IO is used on main-memory regions.
- Clarified ordering rules for hardware A/D bit updates.
- Clarified that, for a given exception cause, `xtval` might sometimes be set to a nonzero value but sometimes not.
- Clarified exception behavior of unimplemented or inaccessible CSRs.
- Clarified that Svpbmt allows implementations to override additional PMAs.
- Replaced the concept of vacant memory regions with inaccessible memory or I/O regions.
- Clarified that timer and count-overflow interrupts' arrival in interrupt-pending registers is not immediate.
- Clarified that MXR affects only explicit memory accesses.

Preface to Version 20211203

This document describes the RISC-V privileged architecture. This release, version 20211203, contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	1.12	Ratified
Supervisor ISA	1.12	Ratified
Svnapot Extension	1.0	Ratified
Svpbmt Extension	1.0	Ratified
Svinval Extension	1.0	Ratified
Hypervisor ISA	1.0	Ratified

The following changes have been made since version 1.11, which, while not strictly backwards compatible, are not anticipated to cause software portability problems in practice:

- Changed MRET and SRET to clear `mstatus`.MPRV when leaving M-mode.
- Reserved additional `satp` patterns for future use.
- Stated that the `scause` Exception Code field must implement bits 4–0 at minimum.
- Relaxed I/O regions have been specified to follow RVWMO. The previous specification implied that PPO rules other than fences and acquire/release annotations did not apply.
- Constrained the LR/SC reservation set size and shape when using page-based virtual memory.
- PMP changes require an SFENCE.VMA on any hart that implements page-based virtual memory, even if VM is not currently enabled.
- Allowed for speculative updates of page table entry A bits.
- Clarify that if the address-translation algorithm non-speculatively reaches a PTE in which a bit reserved for future standard use is set, a page-fault exception must be raised.

Additionally, the following compatible changes have been made since version 1.11:

- Removed the N extension.
- Defined the mandatory RV32-only CSR `mstatush`, which contains most of the same fields as the upper 32 bits of RV64's `mstatus`.
- Defined the mandatory CSR `mconfigptr`, which if nonzero contains the address of a configuration data structure.
- Defined optional `mseccfg` and `mseccfgh` CSRs, which control the machine's security configuration.
- Defined `menvcfg`, `henvcfg`, and `senvcfg` CSRs (and RV32-only `menvcfgh` and `henvcfgh` CSRs), which control various characteristics of the execution environment.
- Designated part of SYSTEM major opcode for custom use.
- Permitted the unconditional delegation of less-privileged interrupts.
- Added optional big-endian and bi-endian support.
- Made priority of load/store/AMO address-misaligned exceptions implementation-defined relative to load/store/AMO page-fault and access-fault exceptions.
- PMP reset values are now platform-defined.
- An additional 48 optional PMP registers have been defined.
- Slightly relaxed the atomicity requirement for A and D bit updates performed by the implementation.
- Clarify the architectural behavior of address-translation caches
- Added Sv57 and Sv57x4 address translation modes.
- Software breakpoint exceptions are permitted to write either 0 or the `pc` to `xtval`.
- Clarified that bare S-mode need not support the SFENCE.VMA instruction.
- Specified relaxed constraints for implicit reads of non-idempotent regions.
- Added the Svnapot Standard Extension, along with the N bit in Sv39, Sv48, and Sv57 PTEs.
- Added the Svpbmt Standard Extension, along with the PBMT bits in Sv39, Sv48, and Sv57 PTEs.
- Added the Svinval Standard Extension and associated instructions.

Finally, the hypervisor architecture proposal has been extensively revised.

Preface to Version 1.11

This is version 1.11 of the RISC-V privileged architecture. The document contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	**1.11**	**Ratified**
Supervisor ISA	**1.11**	**Ratified**
Hypervisor ISA	*0.3*	*Draft*

Changes from version 1.10 include:

- Moved Machine and Supervisor spec to **Ratified** status.
- Improvements to the description and commentary.
- Added a draft proposal for a hypervisor extension.
- Specified which interrupt sources are reserved for standard use.

- Allocated some synchronous exception causes for custom use.
- Specified the priority ordering of synchronous exceptions.
- Added specification that xRET instructions may, but are not required to, clear LR reservations if A extension present.
- The virtual-memory system no longer permits supervisor mode to execute instructions from user pages, regardless of the SUM setting.
- Clarified that ASIDs are private to a hart, and added commentary about the possibility of a future global-ASID extension.
- SFENCE.VMA semantics have been clarified.
- Made the `mstatus`.MPP field **WARL**, rather than **WLRL**.
- Made the unused `xip` fields **WPRI**, rather than **WIRI**.
- Made the unused `misa` fields **WARL**, rather than **WIRI**.
- Made the unused `pmpaddr` and `pmpcfg` fields **WARL**, rather than **WIRI**.
- Required all harts in a system to employ the same PTE-update scheme as each other.
- Rectified an editing error that misdescribed the mechanism by which `mstatus.xIE` is written upon an exception.
- Described scheme for emulating misaligned AMOs.
- Specified the behavior of the `misa` and `xepc` registers in systems with variable IALIGN.
- Specified the behavior of writing self-contradictory values to the `misa` register.
- Defined the `mcountinhibit` CSR, which stops performance counters from incrementing to reduce energy consumption.
- Specified semantics for PMP regions coarser than four bytes.
- Specified contents of CSRs across XLEN modification.
- Moved PLIC chapter into its own document.

Preface to Version 1.10

This is version 1.10 of the RISC-V privileged architecture proposal. Changes from version 1.9.1 include:

- The previous version of this document was released under a Creative Commons Attribution 4.0 International License by the original authors, and this and future versions of this document will be released under the same license.
- The explicit convention on shadow CSR addresses has been removed to reclaim CSR space. Shadow CSRs can still be added as needed.
- The `mvendorid` register now contains the JEDEC code of the core provider as opposed to a code supplied by the Foundation. This avoids redundancy and offloads work from the Foundation.
- The interrupt-enable stack discipline has been simplified.
- An optional mechanism to change the base ISA used by supervisor and user modes has been added to the `mstatus` CSR, and the field previously called Base in `misa` has been renamed to `MXL` for consistency.
- Clarified expected use of XS to summarize additional extension state status fields in `mstatus`.
- Optional vectored interrupt support has been added to the `mtvec` and `stvec` CSRs.
- The SEIP and UEIP bits in the `mip` CSR have been redefined to support software injection of external interrupts.

- The `mbadaddr` register has been subsumed by a more general `mtval` register that can now capture bad instruction bits on an illegal instruction fault to speed instruction emulation.
- The machine-mode base-and-bounds translation and protection schemes have been removed from the specification as part of moving the virtual memory configuration to `sptbr` (now `satp`). Some of the motivation for the base and bound schemes are now covered by the PMP registers, but space remains available in `mstatus` to add these back at a later date if deemed useful.
- In systems with only M-mode, or with both M-mode and U-mode but without U-mode trap support, the `medeleg` and `mideleg` registers now do not exist, whereas previously they returned zero.
- Virtual-memory page faults now have `mcause` values distinct from physical-memory access faults. Page-fault exceptions can now be delegated to S-mode without delegating exceptions generated by PMA and PMP checks.
- An optional physical-memory protection (PMP) scheme has been proposed.
- The supervisor virtual memory configuration has been moved from the `mstatus` register to the `sptbr` register. Accordingly, the `sptbr` register has been renamed to `satp` (Supervisor Address Translation and Protection) to reflect its broadened role.
- The SFENCE.VM instruction has been removed in favor of the improved SFENCE.VMA instruction.
- The `mstatus` bit MXR has been exposed to S-mode via `sstatus`.
- The polarity of the PUM bit in `sstatus` has been inverted to shorten code sequences involving MXR. The bit has been renamed to SUM.
- Hardware management of page-table entry Accessed and Dirty bits has been made optional; simpler implementations may trap to software to set them.
- The counter-enable scheme has changed, so that S-mode can control availability of counters to U-mode.
- H-mode has been removed, as we are focusing on recursive virtualization support in S-mode. The encoding space has been reserved and may be repurposed at a later date.
- A mechanism to improve virtualization performance by trapping S-mode virtual-memory management operations has been added.
- The Supervisor Binary Interface (SBI) chapter has been removed, so that it can be maintained as a separate specification.

Preface to Version 1.9.1

This is version 1.9.1 of the RISC-V privileged architecture proposal. Changes from version 1.9 include:

- Numerous additions and improvements to the commentary sections.
- Change configuration string proposal to be use a search process that supports various formats including Device Tree String and flattened Device Tree.
- Made `misa` optionally writable to support modifying base and supported ISA extensions. CSR address of `misa` changed.
- Added description of debug mode and debug CSRs.
- Added a hardware performance monitoring scheme. Simplified the handling of existing hardware counters, removing privileged versions of the counters and the corresponding delta registers.
- Fixed description of SPIE in presence of user-level interrupts.

Chapter 1. Introduction

This document describes the RISC-V privileged architecture, which covers all aspects of RISC-V systems beyond the unprivileged ISA, including privileged instructions as well as additional functionality required for running operating systems and attaching external devices.

Commentary on our design decisions is formatted as in this paragraph, and can be skipped if the reader is only interested in the specification itself.

We briefly note that the entire privileged-level design described in this document could be replaced with an entirely different privileged-level design without changing the unprivileged ISA, and possibly without even changing the ABI. In particular, this privileged specification was designed to run existing popular operating systems, and so embodies the conventional level-based protection model. Alternate privileged specifications could embody other more flexible protection-domain models. For simplicity of expression, the text is written as if this was the only possible privileged architecture.

1.1. RISC-V Privileged Software Stack Terminology

This section describes the terminology we use to describe components of the wide range of possible privileged software stacks for RISC-V.

Figure 1 shows some of the possible software stacks that can be supported by the RISC-V architecture. The left-hand side shows a simple system that supports only a single application running on an application execution environment (AEE). The application is coded to run with a particular application binary interface (ABI). The ABI includes the supported user-level ISA plus a set of ABI calls to interact with the AEE. The ABI hides details of the AEE from the application to allow greater flexibility in implementing the AEE. The same ABI could be implemented natively on multiple different host OSs, or could be supported by a user-mode emulation environment running on a machine with a different native ISA.

Our graphical convention represents abstract interfaces using black boxes with white text, to separate them from concrete instances of components implementing the interfaces.

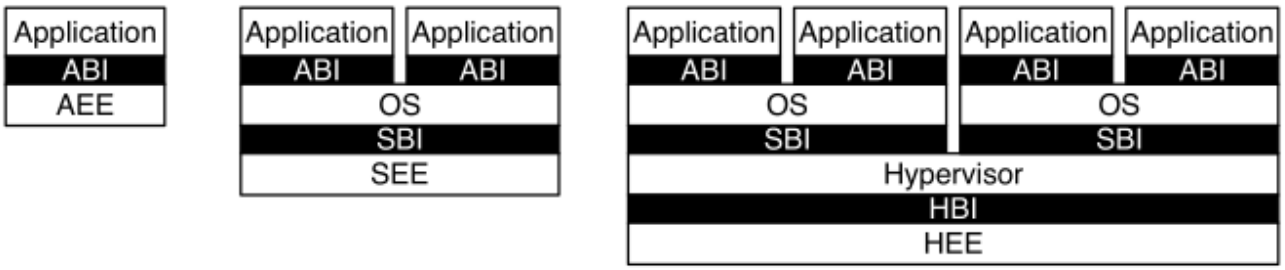


Figure 1. Different implementation stacks supporting various forms of privileged execution.

The middle configuration shows a conventional operating system (OS) that can support multiprogrammed execution of multiple applications. Each application communicates over an ABI with the OS, which provides the AEE. Just as applications interface with an AEE via an ABI, RISC-V operating systems interface with a supervisor execution environment (SEE) via a supervisor binary interface (SBI). An SBI comprises the user-level and supervisor-level ISA together with a set of SBI function calls. Using a single SBI across all SEE implementations allows a single OS binary image to run on any SEE. The SEE can be a simple boot loader and BIOS-style IO system in a low-end hardware platform, or a hypervisor-provided virtual machine in a high-end server, or a thin translation layer over a host operating system in an

architecture simulation environment.

Most supervisor-level ISA definitions do not separate the SBI from the execution environment and/or the hardware platform, complicating virtualization and bring-up of new hardware platforms.

The rightmost configuration shows a virtual machine monitor configuration where multiple multiprogrammed OSs are supported by a single hypervisor. Each OS communicates via an SBI with the hypervisor, which provides the SEE. The hypervisor communicates with the hypervisor execution environment (HEE) using a hypervisor binary interface (HBI), to isolate the hypervisor from details of the hardware platform.

The ABI, SBI, and HBI are still a work-in-progress, but we are now prioritizing support for Type-2 hypervisors where the SBI is provided recursively by an S-mode OS.

Hardware implementations of the RISC-V ISA will generally require additional features beyond the privileged ISA to support the various execution environments (AEE, SEE, or HEE).

1.2. Privilege Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Three RISC-V privilege levels are currently defined as shown in Table 1.

Table 1. RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	`00`	User/Application	U
1	`01`	Supervisor	S
2	`10`	*Reserved*	
3	`11`	Machine	M

Privilege levels are used to provide protection between different components of the software stack, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment.

In the description, we try to separate the privilege level for which code is written, from the privilege mode in which it runs, although the two are often tied. For example, a supervisor-level operating system can run in supervisor-mode on a system with three privilege modes, but can also run in user-mode under a classic virtual machine monitor on systems with two or more privilege modes. In both cases, the same supervisor-level operating system binary code can be used, coded to a supervisor-level SBI and hence expecting to be able to use supervisor-level privileged instructions and CSRs. When running a guest OS in user mode, all supervisor-level actions will be trapped and emulated by the SEE running in the higher-privilege level.

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively.

Each privilege level has a core set of privileged ISA extensions with optional extensions and variants. For example, machine-mode supports an optional standard extension for memory protection. Also, supervisor mode can be extended to support Type-2 hypervisor execution as described in Chapter 19.

Implementations might provide anywhere from 1 to 3 privilege modes trading off reduced isolation for lower implementation cost, as shown in Table 2.

Table 2. Supported combination of privilege modes.

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine. The simplest RISC-V implementations may provide only M-mode, though this will provide no protection against incorrect or malicious application code.

The lock feature of the optional PMP facility can provide some limited protection even with only M-mode implemented.

Many RISC-V implementations will also support at least user mode (U-mode) to protect the rest of the system from application code. Supervisor mode (S-mode) can be added to provide isolation between a supervisor-level operating system and the SEE.

A hart normally runs application code in U-mode until some trap (e.g., a supervisor call or a timer interrupt) forces a switch to a trap handler, which usually runs in a more privileged mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. Traps that increase privilege level are termed *vertical* traps, while traps that remain at the same privilege level are termed *horizontal* traps. The RISC-V privileged architecture provides flexible routing of traps to different privilege layers.

Horizontal traps can be implemented as vertical traps that return control to a horizontal trap handler in the less-privileged mode.

1.3. Debug Mode

Implementations may also include a debug mode to support off-chip debugging and/or manufacturing test. Debug mode (D-mode) can be considered an additional privilege mode, with even more access than M-mode. The separate debug specification proposal describes operation of a RISC-V hart in debug mode. Debug mode reserves a few CSR addresses that are only accessible in D-mode, and may also reserve some portions of the physical address space on a platform.

Chapter 2. Control and Status Registers (CSRs)

The SYSTEM major opcode is used to encode all privileged instructions in the RISC-V ISA. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), which are defined in the Zicsr extension, and all other privileged instructions. The privileged architecture requires the Zicsr extension; which other privileged instructions are required depends on the privileged-architecture feature set.

In addition to the unprivileged state described in Volume I of this manual, an implementation may contain additional CSRs, accessible by some subset of the privilege levels using the CSR instructions described in Volume I. In this chapter, we map out the CSR address space. The following chapters describe the function of each of the CSRs according to privilege level, as well as the other privileged instructions which are generally closely associated with a particular privilege level. Note that although CSRs and instructions are associated with one privilege level, they are also accessible at all higher privilege levels.

Standard CSRs do not have side effects on reads but may have side effects on writes.

2.1. CSR Address Mapping Conventions

The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (csr[11:8]) are used to encode the read and write accessibility of the CSRs according to privilege level as shown in Table 3. The top two bits (csr[11:10]) indicate whether the register is read/write (`00`,`01`, or `10`) or read-only (`11`). The next two bits (csr[9:8]) encode the lowest privilege level that can access the CSR.

The CSR address convention uses the upper bits of the CSR address to encode default access privileges. This simplifies error checking in the hardware and provides a larger CSR space, but does constrain the mapping of CSRs into the address space.

Implementations might allow a more-privileged level to trap otherwise permitted CSR accesses by a less-privileged level to allow these accesses to be intercepted. This change should be transparent to the less-privileged software.

Instructions that access a non-existent CSR are reserved. Attempts to access a CSR without appropriate privilege level raise illegal-instruction exceptions or, as described in Section 19.6.1, virtual-instruction exceptions. Attempts to write a read-only register raise illegal-instruction exceptions. A read/write register might also contain some bits that are read-only, in which case writes to the read-only bits are ignored.

Table 3 also indicates the convention to allocate CSR addresses between standard and custom uses. The CSR addresses designated for custom uses will not be redefined by future standard extensions.

Machine-mode standard read-write CSRs `0x7A0`-`0x7BF` are reserved for use by the debug system. Of these CSRs, `0x7A0`-`0x7AF` are accessible to machine mode, whereas `0x7B0`-`0x7BF` are only visible to debug mode. Implementations should raise illegal-instruction exceptions on machine-mode access to the latter set of registers.

Effective virtualization requires that as many instructions run natively as possible inside a virtualized environment, while any privileged accesses trap to the virtual machine monitor. (Goldberg, 1974) CSRs that are read-only at some lower privilege level are shadowed into separate CSR addresses if they are made read-write at a higher privilege level. This avoids trapping permitted lower-privilege accesses while still causing traps on illegal accesses. Currently, the counters are the only shadowed CSRs.

2.2. CSR Listing

Table 4-Table 8 list the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are standard unprivileged CSRs. The other registers are used by privileged code, as described in the following chapters. Note that not all registers are required on all implementations.

Table 3. Allocation of RISC-V CSR address ranges.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000-0x0FF	Standard read/write
01	00	XXXX	0x400-0x4FF	Standard read/write
10	00	XXXX	0x800-0x8FF	Custom read/write
11	00	0XXX	0xC00-0xC7F	Standard read-only
11	00	10XX	0xC80-0xCBF	Standard read-only
11	00	11XX	0xCC0-0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	0XXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	0XXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	0XXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	0XXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	0XXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAFF	Custom read/write
11	10	0XXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	0XXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write

01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	0XXX	0xB00-0xB7F	Standard read/write
10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	0XXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFBF	Standard read-only
11	11	11XX	0xFC0-0xFFF	Custom read-only

Table 4. Currently allocated RISC-V unprivileged CSR addresses.

Number	Privilege	Name	Description
Unprivileged Floating-Point CSRs			
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm +fflags).
Unprivileged Zicfiss extension CSR			
0x011	URW	ssp	Shadow Stack Pointer.
Unprivileged Counter/Timers			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32 only.
0xC81	URO	timeh	Upper 32 bits of time, RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32 only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32 only.

Table 5. Currently allocated RISC-V supervisor-level CSR addresses.

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Configuration			
0x10A	SRW	senvcfg	Supervisor environment configuration register.
Supervisor Counter Setup			
0x120	SRW	scountinhibit	Supervisor counter-inhibit register.
Supervisor Trap Handling			
0x140	SRW	sscratch	Supervisor scratch register.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor trap value.
0x144	SRW	sip	Supervisor interrupt pending.
0xDA0	SRO	scountovf	Supervisor count overflow.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.
Supervisor State Enable Registers			
0x10C	SRW	sstateen0	Supervisor State Enable 0 Register.
0x10D	SRW	sstateen1	Supervisor State Enable 1 Register.
0x10E	SRW	sstateen2	Supervisor State Enable 2 Register.
0x10F	SRW	sstateen3	Supervisor State Enable 3 Register.

Table 6. Currently allocated RISC-V hypervisor and VS CSR addresses.

Number	Privilege	Name	Description
		Hypervisor Trap Setup	
0x600	HRW	hstatus	Hypervisor status register.
0x602	HRW	hedeleg	Hypervisor exception delegation register.
0x603	HRW	hideleg	Hypervisor interrupt delegation register.
0x604	HRW	hie	Hypervisor interrupt-enable register.
0x606	HRW	hcounteren	Hypervisor counter enable.
0x607	HRW	hgeie	Hypervisor guest external interrupt-enable register.
0x612	HRW	hedelegh	Upper 32 bits of hedeleg, RV32 only.
		Hypervisor Trap Handling	
0x643	HRW	htval	Hypervisor trap value.
0x644	HRW	hip	Hypervisor interrupt pending.
0x645	HRW	hvip	Hypervisor virtual interrupt pending.
0x64A	HRW	htinst	Hypervisor trap instruction (transformed).
0xE12	HRO	hgeip	Hypervisor guest external interrupt pending.
		Hypervisor Configuration	
0x60A	HRW	henvcfg	Hypervisor environment configuration register.
0x61A	HRM	henvcfgh	Upper 32 bits of henvcfg, RV32 only.
		Hypervisor Protection and Translation	
0x680	HRW	hgatp	Hypervisor guest address translation and protection.
		Debug/Trace Registers	
0x6A8	HRW	hcontext	Hypervisor-mode context register.
		Hypervisor Counter/Timer Virtualization Registers	
0x605	HRW	htimedelta	Delta for VS/VU-mode timer.
0x615	HRW	htimedeltah	Upper 32 bits of htimedelta, RV32 only.
		Hypervisor State Enable Registers	
0x60C	HRW	hstateen0	Hypervisor State Enable 0 Register.
0x60D	HRW	hstateen1	Hypervisor State Enable 1 Register.
0x60E	HRW	hstateen2	Hypervisor State Enable 2 Register.
0x60F	HRW	hstateen3	Hypervisor State Enable 3 Register.
0x61C	HRW	hstateen0h	Upper 32 bits of Hypervisor State Enable 0 Register, RV32 only.
0x61D	HRW	hstateen1h	Upper 32 bits of Hypervisor State Enable 1 Register, RV32 only.
0x61E	HRW	hstateen2h	Upper 32 bits of Hypervisor State Enable 2 Register, RV32 only.
0x61F	HRW	hstateen3h	Upper 32 bits of Hypervisor State Enable 3 Register, RV32 only.
		Virtual Supervisor Registers	
0x200	HRW	vsstatus	Virtual supervisor status register.
0x204	HRW	vsie	Virtual supervisor interrupt-enable register.
0x205	HRW	vstvec	Virtual supervisor trap handler base address.
0x240	HRW	vsscratch	Virtual supervisor scratch register.
0x241	HRW	vsepc	Virtual supervisor exception program counter.
0x242	HRW	vscause	Virtual supervisor trap cause.
0x243	HRW	vstval	Virtual supervisor trap value.
0x244	HRW	vsip	Virtual supervisor interrupt pending.
0x280	HRW	vsatp	Virtual supervisor address translation and protection.

Table 7. Currently allocated RISC-V machine-level CSR addresses.

Number	Privilege	Name	Description
		Machine Information Registers	
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
0xF15	MRO	mconfigptr	Pointer to configuration data structure.
		Machine Trap Setup	
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
0x312	MRW	medelegh	Upper 32 bits of medeleg, RV32 only.
		Machine Trap Handling	
0x340	MRW	mscratch	Machine scratch register.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine trap value.
0x344	MRW	mip	Machine interrupt pending.
0x34A	MRW	mtinst	Machine trap instruction (transformed).
0x34B	MRW	mtval2	Machine second trap value.
		Machine Configuration	
0x30A	MRW	menvcfg	Machine environment configuration register.
0x31A	MRW	menvcfgh	Upper 32 bits of menvcfg, RV32 only.
0x747	MRW	mseccfg	Machine security configuration register.
0x757	MRW	mseccfgh	Upper 32 bits of mseccfg, RV32 only.
		Machine Memory Protection	
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
		...	
0x3AE	MRW	pmpcfg14	Physical memory protection configuration.
0x3AF	MRW	pmpcfg15	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		...	
0x3EF	MRW	pmpaddr63	Physical memory protection address register.
		Machine State Enable Registers	
0x30C	MRW	mstateen0	Machine State Enable 0 Register.
0x30D	MRW	mstateen1	Machine State Enable 1 Register.
0x30E	MRW	mstateen2	Machine State Enable 2 Register.
0x30F	MRW	mstateen3	Machine State Enable 3 Register.
0x31C	MRW	mstateen0h	Upper 32 bits of Machine State Enable 0 Register, RV32 only.
0x31D	MRW	mstateen1h	Upper 32 bits of Machine State Enable 1 Register, RV32 only.
0x31E	MRW	mstateen2h	Upper 32 bits of Machine State Enable 2 Register, RV32 only.
0x31F	MRW	mstateen3h	Upper 32 bits of Machine State Enable 3 Register, RV32 only.

Table 8. Currently allocated RISC-V machine-level CSR addresses.

Number	Privilege	Name	Description
Machine Non-Maskable Interrupt Handling			
0x740	MRW	mnscratch	Resumable NMI scratch register.
0x741	MRW	mnepc	Resumable NMI program counter.
0x742	MRW	mncause	Resumable NMI cause.
0x744	MRW	mnstatus	Resumable NMI status.
Machine Counter/Timers			
0xB00	MRW	mcycle	Machine cycle counter.
0xB02	MRW	minstret	Machine instructions-retired counter.
0xB03	MRW	mhpmcounter3	Machine performance-monitoring counter.
0xB04	MRW	mhpmcounter4	Machine performance-monitoring counter.
⋮		⋮	
0xB1F	MRW	mhpmcounter31	Machine performance-monitoring counter.
0xB80	MRW	mcycleh	Upper 32 bits of mcycle, RV32 only.
0xB82	MRW	minstreth	Upper 32 bits of minstret, RV32 only.
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3, RV32 only.
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4, RV32 only.
⋮		⋮ mhpmcounter31h	
0xB9F	MRW		Upper 32 bits of mhpmcounter31, RV32 only.
Machine Counter Setup			
0x320	MRW	mcountinhibit	Machine counter-inhibit register.
0x323	MRW	mhpmevent3	Machine performance-monitoring event selector.
0x324	MRW	mhpmevent4	Machine performance-monitoring event selector.
⋮		⋮	
0x33F	MRW	mhpmevent31	Machine performance-monitoring event selector.
0x723	MRW	mhpmevent3h	Upper 32 bits of mhpmevent3, RV32 only.
0x724	MRW	mhpmevent4h	Upper 32 bits of mhpmevent4, RV32 only.
⋮		⋮	
0x73F	MRW	mhpmevent31h	Upper 32 bits of mhpmevent31, RV32 only.
Debug/Trace Registers (shared with Debug Mode)			
0x7A0	MRW	tselect	Debug/Trace trigger register select.
0x7A1	MRW	tdata1	First Debug/Trace trigger data register.
0x7A2	MRW	tdata2	Second Debug/Trace trigger data register.
0x7A3	MRW	tdata3	Third Debug/Trace trigger data register.
0x7A8	MRW	mcontext	Machine-mode context register.
Debug Mode Registers			
0x7B0	DRW	dcsr	Debug control and status register.
0x7B1	DRW	dpc	Debug program counter.
0x7B2	DRW	dscratch0	Debug scratch register 0.
0x7B3	DRW	dscratch1	Debug scratch register 1.

2.3. CSR Field Specifications

The following definitions and abbreviations are used in specifying the behavior of fields within the CSRs.

2.3.1. Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Some whole read/write fields are reserved for future use. Software should ignore the values read from these fields, and should preserve the values held in these fields when writing values to other fields of the same register. For forward compatibility, implementations that do not furnish these fields must make them read-only zero. These fields are labeled **WPRI** in the register descriptions.

To simplify the software model, any backward-compatible future definition of previously reserved fields within a CSR must cope with the possibility that a non-atomic read/modify/write sequence is used to update other fields in the CSR. Alternatively, the original CSR definition must specify that subfields can only be updated atomically, which may require a two-instruction clear bit/set bit sequence in general that can be problematic if intermediate values are not legal.

2.3.2. Write/Read Only Legal Values (WLRL)

Some read/write CSR fields specify behavior for only a subset of possible bit encodings, with other bit encodings reserved. Software should not write anything other than legal values to such a field, and should not assume a read will return a legal value unless the last write was of a legal value, or the register has not been written since another operation (e.g., reset) set the register to a legal value. These fields are labeled **WLRL** in the register descriptions.

Hardware implementations need only implement enough state bits to differentiate between the supported values, but must always return the complete specified bit-encoding of any supported value when read.

Implementations are permitted but not required to raise an illegal-instruction exception if an instruction attempts to write a non-supported value to a **WLRL** field. Implementations can return arbitrary bit patterns on the read of a **WLRL** field when the last write was of an illegal value, but the value returned should deterministically depend on the illegal written value and the value of the field prior to the write.

2.3.3. Write Any Values, Reads Legal Values (WARL)

Some read/write CSR fields are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read. Assuming that writing the CSR has no other side effects, the range of supported values can be determined by attempting to write a desired setting then reading to see if the value was retained. These fields are labeled **WARL** in the register descriptions.

Implementations will not raise an exception on writes of unsupported values to a **WARL** field. Implementations can return any legal value on the read of a **WARL** field when the last write was of an illegal value, but the legal value returned should deterministically depend on the illegal written value and the architectural state of the hart.

2.4. CSR Field Modulation

If a write to one CSR changes the set of legal values allowed for a field of a second CSR, then unless specified otherwise, the second CSR's field immediately gets an `UNSPECIFIED` value from among its new legal values. This is true even if the field's value before the write remains legal after the write; the value of the field may be changed in consequence of the write to the controlling CSR.

As a special case of this rule, the value written to one CSR may control whether a field of a second CSR is writable (with multiple legal values) or is read-only. When a write to the controlling CSR causes the second CSR's field to change from previously read-only to now writable, that field immediately gets an `UNSPECIFIED` *but legal value, unless specified otherwise.*

Some CSR fields are, when writable, defined as aliases of other CSR fields. Let x be such a CSR field, and let y be the CSR field it aliases when writable. If a write to a controlling CSR causes field x to change from previously read-only to now writable, the new value of x is not

> `UNSPECIFIED` *but instead immediately reflects the existing value of its alias y, as required.*

A change to the value of a CSR for this reason is not a write to the affected CSR and thus does not trigger any side effects specified for that CSR.

2.5. Implicit Reads of CSRs

Implementations sometimes perform *implicit* reads of CSRs. (For example, all S-mode instruction fetches implicitly read the `satp` CSR.) Unless otherwise specified, the value returned by an implicit read of a CSR is the same value that would have been returned by an explicit read of the CSR, using a CSR-access instruction in a sufficient privilege mode.

2.6. CSR Width Modulation

If the width of a CSR is changed (for example, by changing SXLEN or UXLEN, as described in Section 3.1.6.3), the values of the *writable* fields and bits of the new-width CSR are, unless specified otherwise, determined from the previous-width CSR as though by this algorithm:

1. The value of the previous-width CSR is copied to a temporary register of the same width.
2. For the read-only bits of the previous-width CSR, the bits at the same positions in the temporary register are set to zeros.
3. The width of the temporary register is changed to the new width. If the new width *W* is narrower than the previous width, the least-significant *W* bits of the temporary register are retained and the more-significant bits are discarded. If the new width is wider than the previous width, the temporary register is zero-extended to the wider width.
4. Each writable field of the new-width CSR takes the value of the bits at the same positions in the temporary register.

Changing the width of a CSR is not a read or write of the CSR and thus does not trigger any side effects.

2.7. Explicit Accesses to CSRs Wider than XLEN

If a standard CSR is wider than XLEN bits, then an explicit read of the CSR returns the register's least-significant XLEN bits, and an explicit write to the CSR modifies only the register's least-significant XLEN bits, leaving the upper bits unchanged.

Some standard CSRs, such as the counter CSRs of extension Zicntr, are always 64 bits, even when XLEN=32 (RV32). For each such 64-bit CSR (for example, counter `time`), a corresponding 32-bit *high-half CSR* is usually defined with the same name but with the letter 'h' appended at the end (`timeh`). The high-half CSR aliases bits 63:32 of its namesake 64-bit CSR, thus providing a way for RV32 software to read and modify the otherwise-unreachable 32 bits.

Standard high-half CSRs are accessible only when the base RISC-V instruction set is RV32 (XLEN=32). For RV64 (when XLEN=64), the addresses of all standard high-half CSRs are reserved, so an attempt to access a high-half CSR typically raises an illegal-instruction exception.

Chapter 3. Machine-Level ISA, Version 1.13

This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V hart. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset. M-mode can also be used to implement features that are too difficult or expensive to implement in hardware directly. The RISC-V machine-level ISA contains a common core that is extended depending on which other privilege levels are supported and other details of the hardware implementation.

3.1. Machine-Level CSRs

In addition to the machine-level CSRs described in this section, M-mode code can access all CSRs at lower privilege levels.

3.1.1. Machine ISA (`misa`) Register

The `misa` CSR is a **WARL** read-write register reporting the ISA supported by the hart. This register must be readable in any implementation, but a value of zero can be returned to indicate the `misa` register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.

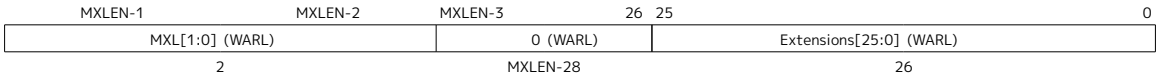


Figure 2. Machine ISA register (misa)

The MXL (Machine XLEN) field encodes the native base integer ISA width as shown in Table 9. The MXL field is read-only. If `misa` is nonzero, the MXL field indicates the effective XLEN in M-mode, a constant termed *MXLEN*. XLEN is never greater than MXLEN, but XLEN might be smaller than MXLEN in less-privileged modes.

Table 9. Encoding of MXL field in `misa`

MXL	XLEN
1	32
2	64
3	128

The `misa` CSR is MXLEN bits wide.

The base width can be quickly ascertained using branches on the sign of the returned `misa` *value, and possibly a shift left by one and a second branch on the sign. These checks can be written in assembly code without knowing the register width (MXLEN) of the hart. The base width is given by* $MXLEN=2^{MXL+4}$.

The base width can also be found if `misa` *is zero, by placing the immediate 4 in a register, then shifting the register left by 31 bits at a time. If zero after one shift, then the hart is RV32. If zero after two shifts, then the hart is RV64, else RV128.*

The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet (bit 0 encodes presence of extension "A" , bit 1 encodes presence of extension "B", through to bit 25 which encodes "Z"). The "I" bit will be set for RV32I, RV64I, and RV128I base ISAs, and the "E" bit will be set for RV32E and RV64E. The Extensions field is a **WARL** field that can contain writable bits where the implementation allows the supported ISA to be modified. At reset, the Extensions field shall contain the

maximal set of supported extensions, and "I" shall be selected over "E" if both are available.

When a standard extension is disabled by clearing its bit in `misa`, the instructions and CSRs defined or modified by the extension revert to their defined or reserved behaviors as if the extension is not implemented.

For a given RISC-V execution environment, an instruction, extension, or other feature of the RISC-V ISA is ordinarily judged to be implemented or not by the observable execution behavior in that environment. For example, the F extension is said to be implemented for an execution environment if and only if the instructions that the RISC-V Unprivileged ISA defines for F execute as specified.

With this definition of implemented, disabling an extension by clearing its bit in `misa` *results in the extension being considered not implemented in M-mode. For example, setting* `misa`*.F=0 results in the F extension being not implemented for M-mode, because the F extension's instructions will not act as the Unprivileged ISA requires but may instead raise an illegal-instruction exception.*

Defining the term implemented based strictly on the observable behavior might conflict with other common understandings of the same word. In particular, although common usage may allow for the combination "implemented but disabled," in this document it is considered a contradiction of terms, because disabled implies execution will not behave as required for the feature to be considered implemented. In the same vein, "implemented and enabled" is redundant here; "implemented" suffices.

Table 10. Encoding of Extensions field in `misa`*. All bits that are reserved for future use must return zero when read.*

Bit	Character	Description
0	A	Atomic extension
1	B	B extension
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E/64E base ISA
5	F	Single-precision floating-point extension
6	G	*Reserved*
7	H	Hypervisor extension
8	I	RV32I/64I/128I base ISA
9	J	*Reserved*
10	K	*Reserved*
11	L	*Reserved*
12	M	Integer Multiply/Divide extension
13	N	*Tentatively reserved for User-Level Interrupts extension*
14	O	*Reserved*
15	P	*Tentatively reserved for Packed-SIMD extension*
16	Q	Quad-precision floating-point extension
17	R	*Reserved*
18	S	Supervisor mode implemented
19	T	*Reserved*
20	U	User mode implemented
21	V	Vector extension
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

The design of the RV128I base ISA is not yet complete, and while much of the remainder of this specification is expected to apply to RV128, this version of the document focuses only on RV32 and RV64.

The "U" and "S" bits will be set if there is support for user and supervisor modes respectively.

The "X" bit will be set if there are any non-standard extensions.

When "B" bit is 1, the implementation supports the instructions provided by the Zba, Zbb, and Zbs extensions. When "B" bit is 0, it indicates that the implementation may not support one or more of the Zba, Zbb, or Zbs extensions.

The `misa` CSR exposes a rudimentary catalog of CPU features to machine-mode code. More extensive information can be obtained in machine mode by probing other machine registers, and examining other ROM storage in the system as part of the boot process.

We require that lower privilege levels execute environment calls instead of reading CPU registers to determine features available at each privilege level. This enables virtualization layers to alter the ISA observed at any level, and supports a much richer command interface without burdening hardware designs.

The "E" bit is read-only. Unless `misa` is all read-only zero, the "E" bit always reads as the complement of the "I" bit. If an execution environment supports both RV32E and RV32I, software can select RV32E by clearing the "I" bit.

If an ISA feature *x* depends on an ISA feature *y*, then attempting to enable feature *x* but disable feature *y* results in both features being disabled. For example, setting "F"=0 and "D"=1 results in both "F" and "D" being cleared.

An implementation may impose additional constraints on the collective setting of two or more `misa` fields, in which case they function collectively as a single **WARL** field. An attempt to write an unsupported combination causes those bits to be set to some supported combination.

Writing `misa` may increase IALIGN, e.g., by disabling the "C" extension. If an instruction that would write `misa` increases IALIGN, and the subsequent instruction's address is not IALIGN-bit aligned, the write to `misa` is suppressed, leaving `misa` unchanged.

When software enables an extension that was previously disabled, then all state uniquely associated with that extension is UNSPECIFIED, unless otherwise specified by that extension.

Although one of the bits 25—0 in `misa` being set to 1 implies that the corresponding feature is implemented, the inverse is not necessarily true: one of these bits being clear does not necessarily imply that the corresponding feature is not implemented. This follows from the fact that, when a feature is not implemented, the corresponding opcodes and CSRs become reserved, not necessarily illegal.

3.1.2. Machine Vendor ID (`mvendorid`) Register

The `mvendorid` CSR is a 32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented or that this is a non-commercial implementation.

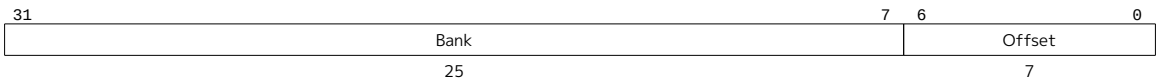


Figure 3. Vendor ID register (`mvendorid`)

JEDEC manufacturer IDs are ordinarily encoded as a sequence of one-byte continuation codes `0x7f`, terminated by a one-byte ID not equal to `0x7f`, with an odd parity bit in the most-significant bit of each

byte. `mvendorid` encodes the number of one-byte continuation codes in the Bank field, and encodes the final byte in the Offset field, discarding the parity bit. For example, the JEDEC manufacturer ID `0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x8a` (twelve continuation codes followed by `0x8a`) would be encoded in the `mvendorid` CSR as `0x60a`.

In JEDEC's parlance, the bank number is one greater than the number of continuation codes; hence, the `mvendorid` Bank field encodes a value that is one less than the JEDEC bank number.

Previously the vendor ID was to be a number allocated by RISC-V International, but this duplicates the work of JEDEC in maintaining a manufacturer ID standard. At time of writing, registering a manufacturer ID with JEDEC has a one-time cost of $500.

3.1.3. Machine Architecture ID (`marchid`) Register

The `marchid` CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of `mvendorid` and `marchid` should uniquely identify the type of hart microarchitecture that is implemented.

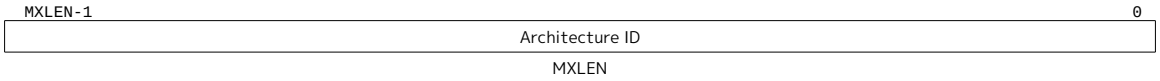


Figure 4. Machine Architecture ID (`marchid`) register

Open-source project architecture IDs are allocated globally by RISC-V International, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining MXLEN-1 bits.

The intent is for the architecture ID to represent the microarchitecture associated with the repo around which development occurs rather than a particular organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original architecture ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source architecture IDs are administered by RISC-V International and should only be allocated to released, functioning open-source projects. Commercial architecture IDs can be managed independently by any registered vendor but are required to have IDs disjoint from the open-source architecture IDs (MSB set) to prevent collisions if a vendor wishes to use both closed-source and open-source microarchitectures.

The convention adopted within the following Implementation field can be used to segregate branches of the same architecture design, including by organization. The `misa` register also helps distinguish different variants of a design.

3.1.4. Machine Implementation ID (`mimpid`) Register

The `mimpid` CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

Figure 5. Machine Implementation ID (`mimpid`) register

The format of this field is left to the provider of the architecture source code, but will often be printed by standard tools as a hexadecimal string without any leading or trailing zeros, so the Implementation value can be left-justified (i.e., filled in from most-significant nibble down) with subfields aligned on nibble boundaries to ease human readability.

3.1.5. Hart ID (`mhartid`) Register

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.

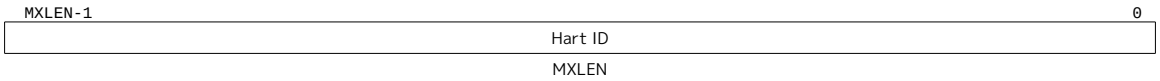


Figure 6. Hart ID (`mhartid`) register

In certain cases, we must ensure exactly one hart runs some code (e.g., at reset), and so require one hart to have a known hart ID of zero.

For efficiency, system implementers should aim to reduce the magnitude of the largest hart ID used in a system.

3.1.6. Machine Status (`mstatus` and `mstatush`) Registers

The `mstatus` register is an MXLEN-bit read/write register formatted as shown in Figure 7 for RV32 and Figure 8 for RV64. The `mstatus` register keeps track of and controls the hart's current operating state. A restricted view of `mstatus` appears as the `sstatus` register in the S-level ISA.

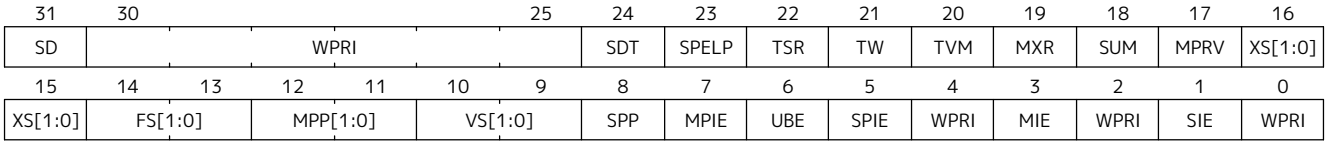


Figure 7. Machine-mode status (`mstatus`) register for RV32

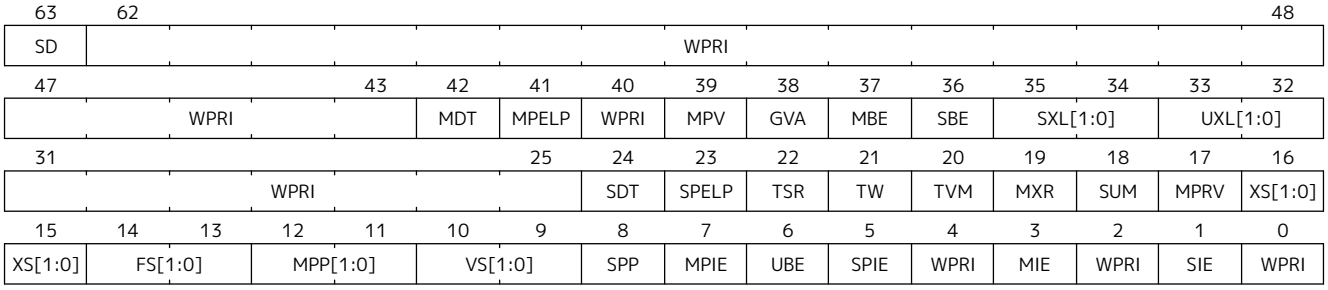


Figure 8. Machine-mode status (`mstatus`) register for RV64

For RV32 only, `mstatush` is a 32-bit read/write register formatted as shown in Figure 9. Bits 30:4 of `mstatush` generally contain the same fields found in bits 62:36 of `mstatus` for RV64. Fields SD, SXL, and UXL do not exist in `mstatush`.

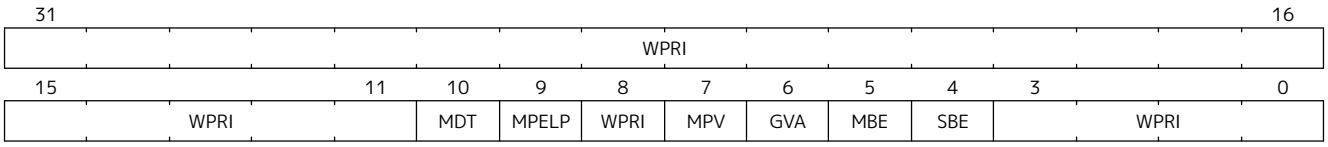


Figure 9. Additional machine-mode status (`mstatush`) register for RV32.

3.1.6.1. Privilege and Global Interrupt-Enable Stack in `mstatus` register

Global interrupt-enable bits, MIE and SIE, are provided for M-mode and S-mode respectively. These bits are primarily used to guarantee atomicity with respect to interrupt handlers in the current privilege mode.

The global xIE bits are located in the low-order bits of `mstatus`, allowing them to be atomically set or cleared with a single CSR instruction.

When a hart is executing in privilege mode x, interrupts are globally enabled when xIE=1 and globally disabled when xIE=0. Interrupts for lower-privilege modes, $w<x$, are always globally disabled regardless of the setting of any global wIE bit for the lower-privilege mode. Interrupts for higher-privilege modes, $y>x$, are always globally enabled regardless of the setting of the global yIE bit for the higher-privilege mode. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected higher-privilege-mode interrupts before ceding control to a lower-privilege mode.

A higher-privilege mode y could disable all of its interrupts before ceding control to a lower-privilege mode but this would be unusual as it would leave only a synchronous trap, non-maskable interrupt, or reset as means to regain control of the hart.

To support nested traps, each privilege mode x that can respond to interrupts has a two-level stack of interrupt-enable bits and privilege modes. xPIE holds the value of the interrupt-enable bit active prior to the trap, and xPP holds the previous privilege mode. The xPP fields can only hold privilege modes up to x, so MPP is two bits wide and SPP is one bit wide. When a trap is taken from privilege mode y into privilege mode x, xPIE is set to the value of xIE; xIE is set to 0; and xPP is set to y.

For lower privilege modes, any trap (synchronous or asynchronous) is usually taken at a higher privilege mode with interrupts disabled upon entry. The higher-level trap handler will either service the trap and return using the stacked information, or, if not returning immediately to the interrupted context, will save the privilege stack before re-enabling interrupts, so only one entry per stack is required.

An MRET or SRET instruction is used to return from a trap in M-mode or S-mode respectively. When executing an xRET instruction, supposing xPP holds the value y, xIE is set to xPIE; the privilege mode is changed to y; xPIE is set to 1; and xPP is set to the least-privileged supported mode (U if U-mode is implemented, else M). If $y \neq$ M, xRET also sets MPRV=0.

Setting xPP to the least-privileged supported mode on an xRET helps identify software bugs in the management of the two-level privilege-mode stack.

Trap handlers must be designed to neither enable interrupts nor cause exceptions during the phase of handling where the trap handler preserves the critical state information required to handle and resume from the trap. An exception or interrupt in this critical phase of trap handling may lead to a trap that can overwrite such critical state. This could result in the loss of data needed to recover from the initial trap. Further, if an exception occurs in the code path needed to handle traps, then such a situation may lead to an infinite loop of traps. To prevent this, trap handlers must be meticulously designed to identify and safely manage exceptions within their operational flow.

xPP fields are **WARL** fields that can hold only privilege mode x and any implemented privilege mode lower

than *x*. If privilege mode *x* is not implemented, then *x*PP must be read-only 0.

M-mode software can determine whether a privilege mode is implemented by writing that mode to MPP then reading it back.

If the machine provides only U and M modes, then only a single hardware storage bit is required to represent either 00 or 11 in MPP.

3.1.6.2. Double Trap Control in `mstatus` Register

A double trap typically arises during a sensitive phase in trap handling operations — when an exception or interrupt occurs while the trap handler (the component responsible for managing these events) is in a non-reentrant state. This non-reentrancy usually occurs in the early phase of trap handling, wherein the trap handler has not yet preserved the necessary state to handle and resume from the trap. The occurrence of a trap during this phase can lead to an overwrite of critical state information, resulting in the loss of data needed to recover from the initial trap. The trap that caused this critical error condition is henceforth called the *unexpected trap*. Trap handlers are designed to neither enable interrupts nor cause exceptions during this phase of handling. However, managing Hardware-Error exceptions, which may occur unpredictably, presents significant challenges in trap handler implementation due to the potential risk of a double trap.

The M-mode-disable-trap (`MDT`) bit is a WARL field introduced by the Smdbltrp extension. Upon reset, the `MDT` field is set to 1. When the `MDT` bit is set to 1 by an explicit CSR write, the `MIE` (Machine Interrupt Enable) bit is cleared to 0. For RV64, this clearing occurs regardless of the value written, if any, to the `MIE` bit by the same write. The `MIE` bit can only be set to 1 by an explicit CSR write if the `MDT` bit is already 0 or, for RV64, is being set to 0 by the same write (For RV32, the `MDT` bit is in `mstatush` and the `MIE` bit in `mstatus` register).

When a trap is to be taken into M-mode, if the `MDT` bit is currently 0, it is then set to 1, and the trap is delivered as expected. However, if `MDT` is already set to 1, then this is an *unexpected trap*. When the Smrnmi extension is implemented, a trap caused by an RNMI is not considered an *unexpected trap* irrespective of the state of the `MDT` bit. A trap caused by an RNMI does not set the `MDT` bit. However, a trap that occurs when executing in M-mode with `mnstatus.NMIE` set to 0 is an *unexpected trap*.

In the event of a *unexpected trap*, the handling is as follows:

- When the Smrnmi extension is implemented and `mnstatus.NMIE` is 1, the hart traps to the RNMI handler. To deliver this trap, the `mnepc` and `mncause` registers are written with the values that the *unexpected trap* would have written to the `mepc` and `mcause` registers respectively. The privilege mode information fields in the `mnstatus` register are written to indicate M-mode and its `NMIE` field is set to 0.

The consequence of this specification is that on occurrence of double trap the RNMI handler is not provided with information that a trap reports in the `mtval` and the `mtval2` registers. This information, if needed, can be obtained by the RNMI handler by decoding the instruction at the address in `mnepc` and examining its source register contents.

- When the Smrnmi extension is not implemented, or if the Smrnmi extension is implemented and `mnstatus.NMIE` is 0, the hart enters a critical-error state without updating any architectural state, including the `pc`. This state involves ceasing execution, disabling all interrupts (including NMIs), and asserting a `critical-error` signal to the platform.

The actions performed by the platform when a hart asserts a `critical-error` signal are platform-specific. The range of possible actions include restarting the affected hart or restarting the entire platform, among others.

The `MRET` and `SRET` instructions, when executed in M-mode, set the `MDT` bit to 0. If the new privilege mode is

U, VS, or VU, then `sstatus.SDT` is also set to 0. Additionally, if it is VU, then `vsstatus.SDT` is also set to 0.

The `MNRET` instruction, provided by the Smrnmi extension, sets the `MDT` bit to 0 if the new privilege mode is not M. If it is U, VS, or VU, then `sstatus.SDT` is also set to 0. Additionally, if it is VU, then `vsstatus.SDT` is also set to 0.

3.1.6.3. Base ISA Control in `mstatus` Register

For RV64 harts, the SXL and UXL fields are **WARL** fields that control the value of XLEN for S-mode and U-mode, respectively. The encoding of these fields is the same as the MXL field of `misa`, shown in Table 9. The effective XLEN in S-mode and U-mode are termed *SXLEN* and *UXLEN*, respectively.

When MXLEN=32, the SXL and UXL fields do not exist, and SXLEN=32 and UXLEN=32.

When MXLEN=64, if S-mode is not supported, then SXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of SXLEN. In particular, an implementation may make SXL be a read-only field whose value always ensures that SXLEN=MXLEN.

When MXLEN=64, if U-mode is not supported, then UXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=MXLEN or UXLEN=SXLEN.

If S-mode is implemented, the set of legal values that the UXL field may assume excludes those that would cause UXLEN to be greater than SXLEN.

Whenever XLEN in any mode is set to a value less than the widest supported XLEN, all operations must ignore source operand register bits above the configured XLEN, and must sign-extend results to fill the entire widest supported XLEN in the destination register. Similarly, `pc` bits above XLEN are ignored, and when the `pc` is written, it is sign-extended to fill the widest supported XLEN.

We require that operations always fill the entire underlying hardware registers with defined values to avoid implementation-defined behavior.

To reduce hardware complexity, the architecture imposes no checks that lower-privilege modes have XLEN settings less than or equal to the next-higher privilege mode. In practice, such settings would almost always be a software bug, but machine operation is well-defined even in this case.

Some HINT instructions are encoded as integer computational instructions that overwrite their destination register with its current value, e.g., `c.addi x8, 0`. When such a HINT is executed with XLEN < MXLEN and bits MXLEN..XLEN of the destination register not all equal to bit XLEN-1, it is implementation-defined whether bits MXLEN..XLEN of the destination register are unchanged or are overwritten with copies of bit XLEN-1.

This definition allows implementations to elide register writeback for some HINTs, while allowing them to execute other HINTs in the same manner as other integer computational instructions. The implementation choice is observable only by privilege modes with an XLEN setting greater than the current XLEN; it is invisible to the current privilege mode.

3.1.6.4. Memory Privilege in `mstatus` Register

The MPRV (Modify PRiVilege) bit modifies the *effective privilege mode*, i.e., the privilege level at which loads and stores execute. When MPRV=0, loads and stores behave as normal, using the translation and protection mechanisms of the current privilege mode. When MPRV=1, load and store memory addresses

are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected by the setting of MPRV. MPRV is read-only 0 if U-mode is not supported.

An MRET or SRET instruction that changes the privilege mode to a mode less privileged than M also sets MPRV=0.

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in Figure 60) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect. MXR is read-only 0 if S-mode is not supported.

The MPRV and MXR mechanisms were conceived to improve the efficiency of M-mode routines that emulate missing hardware features, e.g., misaligned loads and stores. MPRV obviates the need to perform address translation in software. MXR allows instruction words to be loaded from pages marked execute-only.

The current privilege mode and the privilege mode specified by MPP might have different XLEN settings. When MPRV=1, load and store memory addresses are treated as though the current XLEN were set to MPP's XLEN, following the rules in Section 3.1.6.3.

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Figure 60) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it *is* in effect when MPRV=1 and MPP=S. SUM is read-only 0 if S-mode is not supported or if `satp`.MODE is read-only 0.

The MXR and SUM mechanisms only affect the interpretation of permissions encoded in page-table entries. In particular, they have no impact on whether access-fault exceptions are raised due to PMAs or PMP.

3.1.6.5. Endianness Control in `mstatus` and `mstatush` Registers

The MBE, SBE, and UBE bits in `mstatus` and `mstatush` are **WARL** fields that control the endianness of memory accesses other than instruction fetches. Instruction fetches are always little-endian.

MBE controls whether non-instruction-fetch memory accesses made from M-mode (assuming `mstatus`.MPRV=0) are little-endian (MBE=0) or big-endian (MBE=1).

If S-mode is not supported, SBE is read-only 0. Otherwise, SBE controls whether explicit load and store memory accesses made from S-mode are little-endian (SBE=0) or big-endian (SBE=1).

If U-mode is not supported, UBE is read-only 0. Otherwise, UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1).

For *implicit* accesses to supervisor-level memory management data structures, such as page tables, endianness is always controlled by SBE. Since changing SBE alters the implementation's interpretation of these data structures, if any such data structures remain in use across a change to SBE, M-mode software must follow such a change to SBE by executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0`.

Only in contrived scenarios will a given memory-management data structure be interpreted as both little-endian and big-endian. In practice, SBE will only be changed at runtime on world switches, in which case neither the old nor new memory-management data structure will be reinterpreted in a different endianness. In this case, no additional SFENCE.VMA is necessary,

beyond what would ordinarily be required for a world switch.

If S-mode is supported, an implementation may make SBE be a read-only copy of MBE. If U-mode is supported, an implementation may make UBE be a read-only copy of either MBE or SBE.

An implementation supports only little-endian memory accesses if fields MBE, SBE, and UBE are all read-only 0. An implementation supports only big-endian memory accesses (aside from instruction fetches) if MBE is read-only 1 and SBE and UBE are each read-only 1 when S-mode and U-mode are supported.

Volume I defines a hart's address space as a circular sequence of 2^{XLEN} bytes at consecutive addresses. The correspondence between addresses and byte locations is fixed and not affected by any endianness mode. Rather, the applicable endianness mode determines the order of mapping between memory bytes and a multibyte quantity (halfword, word, etc.).

Standard RISC-V ABIs are expected to be purely little-endian-only or big-endian-only, with no accommodation for mixing endianness. Nevertheless, endianness control has been defined so as to permit, for instance, an OS of one endianness to execute user-mode programs of the opposite endianness. Consideration has been given also to the possibility of non-standard usages whereby software flips the endianness of memory accesses as needed.

RISC-V instructions are uniformly little-endian to decouple instruction encoding from the current endianness settings, for the benefit of both hardware and software. Otherwise, for instance, a RISC-V assembler or disassembler would always need to know the intended active endianness, despite that the endianness mode might change dynamically during execution. In contrast, by giving instructions a fixed endianness, it is sometimes possible for carefully written software to be endianness-agnostic even in binary form, much like position-independent code.

The choice to have instructions be only little-endian does have consequences, however, for RISC-V software that encodes or decodes machine instructions. In big-endian mode, such software must account for the fact that explicit loads and stores have endianness opposite that of instructions, for example by swapping byte order after loads and before stores.

3.1.6.6. Virtualization Support in `mstatus` Register

The TVM (Trap Virtual Memory) bit is a **WARL** field that supports intercepting supervisor virtual-memory management operations. When TVM=1, attempts to read or write the `satp` CSR or execute an SFENCE.VMA or SINVAL.VMA instruction while executing in S-mode will raise an illegal-instruction exception. When TVM=0, these operations are permitted in S-mode. TVM is read-only 0 when S-mode is not supported.

The TVM mechanism improves virtualization efficiency by permitting guest operating systems to execute in S-mode, rather than classically virtualizing them in U-mode. This approach obviates the need to trap accesses to most S-mode CSRs.

Trapping `satp` accesses and the SFENCE.VMA and SINVAL.VMA instructions provides the hooks necessary to lazily populate shadow page tables.

The TW (Timeout Wait) bit is a **WARL** field that supports intercepting the WFI instruction (see Section 3.3.3). When TW=0, the WFI instruction may execute in lower privilege modes when not prevented for

some other reason. When TW=1, then if WFI is executed in any less-privileged mode, and it does not complete within an implementation-specific, bounded time limit, the WFI instruction causes an illegal-instruction exception. An implementation may have WFI always raise an illegal-instruction exception in less-privileged modes when TW=1, even if there are pending globally-disabled interrupts when the instruction is executed. TW is read-only 0 when there are no modes less privileged than M.

Trapping the WFI instruction can trigger a world switch to another guest OS, rather than wastefully idling in the current guest.

When S-mode is implemented, then executing WFI in U-mode causes an illegal-instruction exception, unless it completes within an implementation-specific, bounded time limit. A future revision of this specification might add a feature that allows S-mode to selectively permit WFI in U-mode. Such a feature would only be active when TW=0.

The TSR (Trap SRET) bit is a **WARL** field that supports intercepting the supervisor exception return instruction, SRET. When TSR=1, attempts to execute SRET while executing in S-mode will raise an illegal-instruction exception. When TSR=0, this operation is permitted in S-mode. TSR is read-only 0 when S-mode is not supported.

Trapping SRET is necessary to emulate the hypervisor extension (see Chapter 19) on implementations that do not provide it.

3.1.6.7. Extension Context Status in `mstatus` Register

Supporting substantial extensions is one of the primary goals of RISC-V, and hence we define a standard interface to allow unchanged privileged-mode code, particularly a supervisor-level OS, to support arbitrary user-mode state extensions.

To date, the V extension is the only standard extension that defines additional state beyond the floating-point CSR and data registers.

The FS[1:0] and VS[1:0] **WARL** fields and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit state, including the floating-point registers `f0`–`f31` and the CSRs `fcsr`, `frm`, and `fflags`. The VS field encodes the status of the vector extension state, including the vector registers `v0`–`v31` and the CSRs `vcsr`, `vxrm`, `vxsat`, `vstart`, `vl`, `vtype`, and `vlenb`. The XS field encodes the status of additional user-mode extensions and associated state. These fields can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

The design anticipates that most context switches will not need to save/restore state in either or both of the floating-point unit or other extensions, so provides a fast check via the SD bit.

The FS, VS, and XS fields use the same status encoding as shown in Table 11, with the four possible status values being Off, Initial, Clean, and Dirty.

Table 11. Encoding of FS[1:0], VS[1:0], and XS[1:0] status fields

Status	FS and VS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

If the F extension is implemented, the FS field shall not be read-only zero.

If neither the F extension nor S-mode is implemented, then FS is read-only zero. If S-mode is implemented but the F extension is not, FS may optionally be read-only zero.

Implementations with S-mode but without the F extension are permitted, but not required, to make the FS field be read-only zero. Some such implementations will choose not to have the FS field be read-only zero, so as to enable emulation of the F extension for both S-mode and U-mode via invisible traps into M-mode.

If the v registers are implemented, the VS field shall not be read-only zero.

If neither the v registers nor S-mode is implemented, then VS is read-only zero. If S-mode is implemented but the v registers are not, VS may optionally be read-only zero.

In harts without additional user extensions requiring new state, the XS field is read-only zero. Every additional extension with state provides a CSR field that encodes the equivalent of the XS states. The XS field represents a summary of all extensions' status as shown in Table 11.

The XS field effectively reports the maximum status value across all user-extension status fields, though individual extensions can use a different encoding than XS.

The SD bit is a read-only bit that summarizes whether either the FS, VS, or XS fields signal the presence of some dirty state that will require saving extended user context to memory. If FS, XS, and VS are all read-only zero, then SD is also always zero.

When an extension's status is set to Off, any instruction that attempts to read or write the corresponding state will cause an illegal-instruction exception. When the status is Initial, the corresponding state should have an initial constant value. When the status is Clean, the corresponding state is potentially different from the initial value, but matches the last value stored on a context swap. When the status is Dirty, the corresponding state has potentially been modified since the last context save.

During a context save, the responsible privileged code need only write out the corresponding state if its status is Dirty, and can then reset the extension's status to Clean. During a context restore, the context need only be loaded from memory if the status is Clean (it should never be Dirty at restore). If the status is Initial, the context must be set to an initial constant value on context restore to avoid a security hole, but this can be done without accessing memory. For example, the floating-point registers can all be initialized to the immediate value 0.

The FS and XS fields are read by the privileged code before saving the context. The FS field is set directly by privileged code when resuming a user context, while the XS field is set indirectly by writing to the status register of the individual extensions. The status fields will also be updated during execution of instructions, regardless of privilege mode.

Extensions to the user-mode ISA often include additional user-mode state, and this state can be considerably larger than the base integer registers. The extensions might only be used for some applications, or might only be needed for short phases within a single application. To improve performance, the user-mode extension can define additional instructions to allow user-mode software to return the unit to an initial state or even to turn off the unit.

For example, a coprocessor might require to be configured before use and can be "unconfigured" after use. The unconfigured state would be represented as the Initial state for context save. If the same application remains running between the unconfigure and the next configure (which would set status to Dirty), there is no need to actually reinitialize the state at the unconfigure instruction, as all state is local to the user process, i.e., the Initial state may only cause the coprocessor state to be initialized to a constant value at

context restore, not at every unconfigure.

Executing a user-mode instruction to disable a unit and place it into the Off state will cause an illegal-instruction exception to be raised if any subsequent instruction tries to use the unit before it is turned back on. A user-mode instruction to turn a unit on must also ensure the unit's state is properly initialized, as the unit might have been used by another context meantime.

Changing the setting of FS has no effect on the contents of the floating-point register state. In particular, setting FS=Off does not destroy the state, nor does setting FS=Initial clear the contents. Similarly, the setting of VS has no effect on the contents of the vector register state. Other extensions, however, might not preserve state when set to Off.

Implementations may choose to track the dirtiness of the floating-point register state imprecisely by reporting the state to be dirty even when it has not been modified. On some implementations, some instructions that do not mutate the floating-point state may cause the state to transition from Initial or Clean to Dirty. On other implementations, dirtiness might not be tracked at all, in which case the valid FS states are Off and Dirty, and an attempt to set FS to Initial or Clean causes it to be set to Dirty.

This definition of FS does not disallow setting FS to Dirty as a result of errant speculation. Some platforms may choose to disallow speculatively writing FS to close a potential side channel.

If an instruction explicitly or implicitly writes a floating-point register or the `fcsr` but does not alter its contents, and FS=Initial or FS=Clean, it is implementation-defined whether FS transitions to Dirty.

Implementations may choose to track the dirtiness of the vector register state in an analogous imprecise fashion, including possibly setting VS to Dirty when software attempts to set VS=Initial or VS=Clean. When VS=Initial or VS=Clean, it is implementation-defined whether an instruction that writes a vector register or vector CSR but does not alter its contents causes VS to transition to Dirty.

Table 12 shows all the possible state transitions for the FS, VS, or XS status bits. Note that the standard floating-point and vector extensions do not support user-mode unconfigure or disable/enable instructions.

Table 12. FS, VS, and XS state transitions.

Current State Action	Off	Initial	Clean	Dirty
At context save in privileged code				
Save state? Next state	No Off	No Initial	No Clean	Yes Clean
At context restore in privileged code				
Restore state? Next state	No Off	Yes, to initial Initial	Yes, from memory Clean	N/A N/A
Execute instruction to read state				
Action? Next state	Exception Off	Execute Initial	Execute Clean	Execute Dirty
Execute instruction that possibly modifies state, including configuration				
Action? Next state	Exception Off	Execute Dirty	Execute Dirty	Execute Dirty
Execute instruction to unconfigure unit				
Action? Next state	Exception Off	Execute Initial	Execute Initial	Execute Initial
Execute instruction to disable unit				
Action? Next state	Execute Off	Execute Off	Execute Off	Execute Off
Execute instruction to enable unit				
Action? Next state	Execute Initial	Execute Initial	Execute Initial	Execute Initial

Standard privileged instructions to initialize, save, and restore extension state are provided to insulate privileged code from details of the added extension state by treating the state as an opaque object.

Many coprocessor extensions are only used in limited contexts that allows software to safely unconfigure or even disable units when done. This reduces the context-switch overhead of large stateful coprocessors.

We separate out floating-point state from other extension state, as when a floating-point unit is present the floating-point registers are part of the standard calling convention, and so user-mode software cannot know when it is safe to disable the floating-point unit.

The XS field provides a summary of all added extension state, but additional microarchitectural bits might be maintained in the extension to further reduce context save and restore overhead.

The SD bit is read-only and is set when either the FS, VS, or XS bits encode a Dirty state (i.e., SD=FS==11) OR (XS==11) OR (VS==11). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and `pc`.

The floating-point unit state is always initialized, saved, and restored using standard instructions (F, D, and/or Q), and privileged code must be aware of FLEN to determine the appropriate space to reserve for each `f` register.

Machine and Supervisor modes share a single copy of the FS, VS, and XS bits. Supervisor-level software normally uses the FS, VS, and XS bits directly to record the status with respect to the supervisor-level saved context. Machine-level software must be more conservative in saving and restoring the extension state in their corresponding version of the context.

In any reasonable use case, the number of context switches between user and supervisor level should far outweigh the number of context switches to other privilege levels. Note that coprocessors should not require their context to be saved and restored to service asynchronous interrupts, unless the interrupt results in a user-level context swap.

3.1.6.8. Previous Expected Landing Pad (ELP) State in `mstatus` Register

The Zicfilp extension adds the `SPELP` and `MPELP` fields that hold the previous `ELP`, and are updated as specified in Section 20.1.2. The *x*`PELP` fields are encoded as follows:

- 0 - `NO_LP_EXPECTED` - no landing pad instruction expected.
- 1 - `LP_EXPECTED` - a landing pad instruction is expected.

3.1.7. Machine Trap-Vector Base-Address (`mtvec`) Register

The `mtvec` register is an MXLEN-bit **WARL** read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

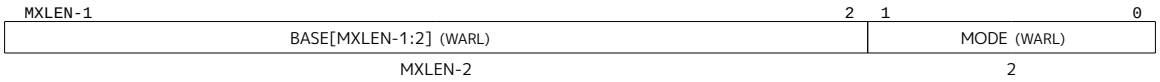


Figure 10. Encoding of mtvec MODE field.

The `mtvec` register must always be implemented, but can contain a read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

We allow for considerable flexibility in implementation of the trap vector base address. On the one hand, we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems.

Table 13. Encoding of mtvec MODE field.

Value	Name	Description
0	Direct	All traps set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	---	*Reserved*

The encoding of the MODE field is shown in Table 13. When MODE=Direct, all traps into machine mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number. For example, a machine-mode timer interrupt (see Table 14) causes the `pc` to be set to BASE+`0x1c`.

An implementation may have different alignment constraints for different modes. In particular,

MODE=Vectored may have stricter alignment constraints than MODE=Direct.

Allowing coarser alignments in Vectored mode enables vectoring to be implemented without a hardware adder circuit.

Reset and NMI vector locations are given in a platform specification.

3.1.8. Machine Trap Delegation (`medeleg` and `mideleg`) Registers

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 (Section 3.3.2). To increase performance, implementations can provide individual read/write bits within `medeleg` and `mideleg` to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (`medeleg`) is a 64-bit read/write register. The machine interrupt delegation (`mideleg`) register is an MXLEN-bit read/write register.

In harts with S-mode, the `medeleg` and `mideleg` registers must exist, and setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. In harts without S-mode, the `medeleg` and `mideleg` registers should not exist.

In versions 1.9.1 and earlier , these registers existed but were 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.

When a trap is delegated to S-mode, the `scause` register is written with the trap cause; the `sepc` register is written with the virtual address of the instruction that took the trap; the `stval` register is written with an exception-specific datum; the SPP field of `mstatus` is written with the active privilege mode at the time of the trap; the SPIE field of `mstatus` is written with the value of the SIE field at the time of the trap; and the SIE field of `mstatus` is cleared. The `mcause`, `mepc`, and `mtval` registers and the MPP and MPIE fields of `mstatus` are not written.

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 `medeleg` or `mideleg` to see which bit positions hold a one.

An implementation shall not have any bits of `medeleg` be read-only one, i.e., any synchronous trap that can be delegated must support not being delegated. Similarly, an implementation shall not fix as read-only one any bits of `mideleg` corresponding to machine-level interrupts (but may do so for lower-level interrupts).

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.

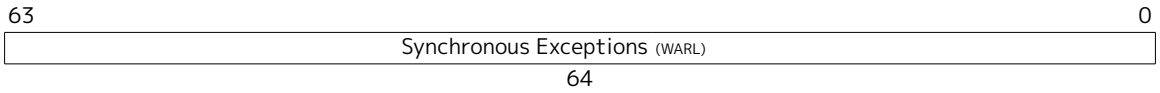


Figure 11. Machine Exception Delegation (`medeleg`) register.

`medeleg` has a bit position allocated for every synchronous exception shown in Table 14, with the index of the bit position equal to the value returned in the `mcause` register (i.e., setting bit 8 allows user-mode environment calls to be delegated to a lower-privilege trap handler).

When XLEN=32, `medelegh` is a 32-bit read/write register that aliases bits 63:32 of `medeleg`. The `medelegh` register does not exist when XLEN=64.

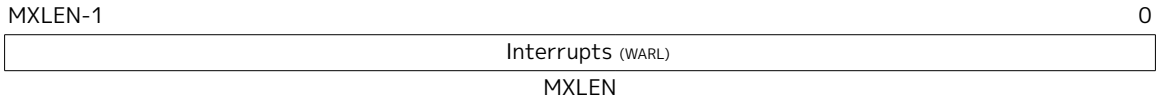


Figure 12. Machine Interrupt Delegation (`mideleg`) Register.

`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.

3.1.9. Machine Interrupt (`mip` and `mie`) Registers

The `mip` register is an MXLEN-bit read/write register containing information on pending interrupts, while `mie` is the corresponding MXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number *i* (as reported in CSR `mcause`, Section 3.1.15) corresponds with bit *i* in both `mip` and `mie`. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.

Interrupts designated for platform use may be designated for custom use at the platform's discretion.

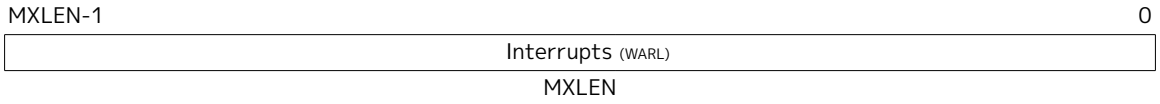


Figure 13. Machine Interrupt-Pending (`mip`) register.

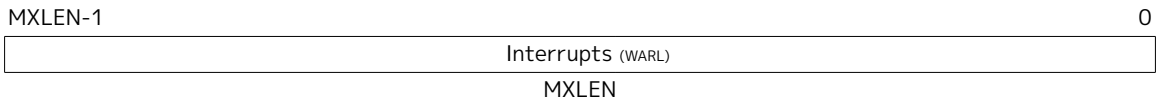


Figure 14. Machine Interrupt-Enable (`mie`) register

An interrupt *i* will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true: (a) either the current privilege mode is M and the MIE bit in the `mstatus` register is set, or the current privilege mode has less privilege than M-mode; (b) bit *i* is set in both `mip` and `mie`; and (c) if register `mideleg` exists, bit *i* is not set in `mideleg`.

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in `mip`, and must also be evaluated immediately following the execution of an *x*RET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including `mip`, `mie`, `mstatus`, and `mideleg`).

Interrupts to M-mode take priority over any interrupts to lower privilege modes.

Each individual bit in register `mip` may be writable or may be read-only. When bit *i* in `mip` is writable, a pending interrupt *i* can be cleared by writing 0 to this bit. If interrupt *i* can become pending but bit *i* in `mip` is read-only, the implementation must provide some other mechanism for clearing the pending interrupt.

A bit in `mie` must be writable if the corresponding interrupt can ever become pending. Bits of `mie` that are not writable must be read-only zero.

The standard portions (bits 15:0) of the `mip` and `mie` registers are formatted as shown in Figure 15 and Figure 16 respectively.

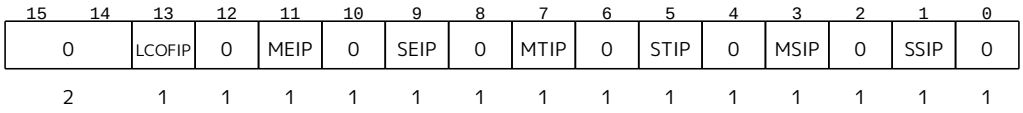


Figure 15. Standard portion (bits 15:0) of `mip`.

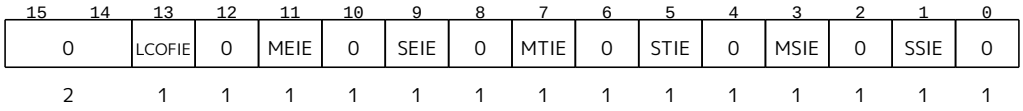


Figure 16. Standard portion (bits 15:0) of `mie`.

The machine-level interrupt registers handle a few root interrupt sources which are assigned a fixed service priority for simplicity, while separate external interrupt controllers can implement a more complex prioritization scheme over a much larger set of interrupts that are then muxed into the machine-level interrupt sources.

The non-maskable interrupt is not made visible via the `mip` register as its presence is implicitly known when executing the NMI trap handler.

Bits `mip`.MEIP and `mie`.MEIE are the interrupt-pending and interrupt-enable bits for machine-level external interrupts. MEIP is read-only in `mip`, and is set and cleared by a platform-specific interrupt controller.

Bits `mip`.MTIP and `mie`.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is read-only in the `mip` register, and is cleared by writing to the memory-mapped machine-mode timer compare register.

Bits `mip`.MSIP and `mie`.MSIE are the interrupt-pending and interrupt-enable bits for machine-level software interrupts. MSIP is read-only in `mip`, and is written by accesses to memory-mapped control registers, which are used by remote harts to provide machine-level interprocessor interrupts. A hart can write its own MSIP bit using the same memory-mapped control register. If a system has only one hart, or if a platform standard supports the delivery of machine-level interprocessor interrupts through external interrupts (MEI) instead, then `mip`.MSIP and `mie`.MSIE may both be read-only zeros.

If supervisor mode is not implemented, bits SEIP, STIP, and SSIP of `mip` and SEIE, STIE, and SSIE of `mie` are read-only zeros.

If supervisor mode is implemented, bits `mip`.SEIP and `mie`.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. SEIP is writable in `mip`, and may be written by M-mode software to indicate to S-mode that an external interrupt is pending. Additionally, the platform-level interrupt controller may generate supervisor-level external interrupts. Supervisor-level external interrupts are made pending based on the logical-OR of the software-writable SEIP bit and the signal from the external interrupt controller. When `mip` is read with a CSR instruction, the value of the SEIP bit returned in

the `rd` destination register is the logical-OR of the software-writable bit and the interrupt signal from the interrupt controller, but the signal from the interrupt controller is not used to calculate the value written to SEIP. Only the software-writable SEIP bit participates in the read-modify-write sequence of a CSRRS or CSRRC instruction.

For example, if we name the software-writable SEIP bit `B` *and the signal from the external interrupt controller* `E`*, then if* `csrrs t0, mip, t1` *is executed,* `t0[9]` *is written with* `B || E`*, then* `B` *is written with* `B || t1[9]`*. If* `csrrw t0, mip, t1` *is executed, then* `t0[9]` *is written with* `B || E`*, and* `B` *is simply written with* `t1[9]`*. In neither case does* `B` *depend upon* `E`*.*

The SEIP field behavior is designed to allow a higher privilege layer to mimic external interrupts cleanly, without losing any real external interrupts. The behavior of the CSR instructions is slightly modified from regular CSR accesses as a result.

If supervisor mode is implemented, bits `mip`.STIP and `mie`.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. STIP is writable in `mip`, and may be written by M-mode software to deliver timer interrupts to S-mode.

If supervisor mode is implemented, bits `mip`.SSIP and `mie`.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. SSIP is writable in `mip` and may also be set to 1 by a platform-specific interrupt controller.

If the Sscofpmf extension is implemented, bits `mip`.LCOFIP and `mie`.LCOFIE are the interrupt-pending and interrupt-enable bits for local counter-overflow interrupts. LCOFIP is read-write in `mip` and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the `mhpmeventn`.OF bits being set. If the Sscofpmf extension is not implemented, `mip`.LCOFIP and `mie`.LCOFIE are read-only zeros.

Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

The machine-level interrupt fixed-priority ordering rules were developed with the following rationale.

Interrupts for higher privilege modes must be serviced before interrupts for lower privilege modes to support preemption.

The platform-specific machine-level interrupt sources in bits 16 and above have platform-specific priority, but are typically chosen to have the highest service priority to support very fast local vectored interrupts.

External interrupts are handled before internal (timer/software) interrupts as external interrupts are usually generated by devices that might require low interrupt service times.

Software interrupts are handled before internal timer interrupts, because internal timer interrupts are usually intended for time slicing, where time precision is less important, whereas software interrupts are used for inter-processor messaging. Software interrupts can be avoided when high-precision timing is required, or high-precision timer interrupts can be routed via a different interrupt path. Software interrupts are located in the lowest four bits of `mip` *as these are often written by software, and this position allows the use of a single CSR instruction with a five-bit immediate.*

Restricted views of the `mip` and `mie` registers appear as the `sip` and `sie` registers for supervisor level. If an interrupt is delegated to S-mode by setting a bit in the `mideleg` register, it becomes visible in the `sip` register and is maskable using the `sie` register. Otherwise, the corresponding bits in `sip` and `sie` are read-only zero.

3.1.10. Hardware Performance Monitor

M-mode includes a basic hardware performance-monitoring facility. The `mcycle` CSR counts the number of clock cycles executed by the processor core on which the hart is running. The `minstret` CSR counts the number of instructions the hart has retired. The `mcycle` and `minstret` registers have 64-bit precision on all RV32 and RV64 harts.

The counter registers have an arbitrary value after the hart is reset, and can be written with a given value. Any CSR write takes effect after the writing instruction has otherwise completed. 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.

The hardware performance monitor includes 29 additional 64-bit event counters, `mhpmcounter3`-`mhpmcounter31`. The event selector CSRs, `mhpmevent3`-`mhpmevent31`, are 64-bit **WARL** registers that control which event causes the corresponding counter to increment. The meaning of these events is defined by the platform, but event 0 is defined to mean "no event." All counters should be implemented, but a legal implementation is to make both the counter and its corresponding event selector be read-only 0.

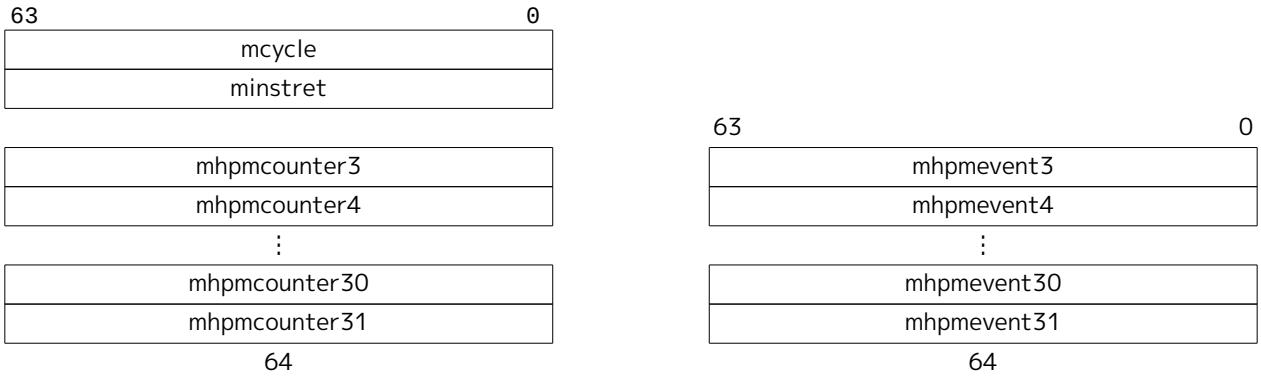


Figure 17. Hardware performance monitor counters.

The `mhpmcounters` are **WARL** registers that support up to 64 bits of precision on RV32 and RV64.

When XLEN=32, reads of the `mcycle`, `minstret`, `mhpmcounter`*n*, and `mhpmevent`*n* CSRs return bits 31-0 of the corresponding register, and writes change only bits 31-0; reads of the `mcycleh`, `minstreth`, `mhpmcounter`*n*`h`, and `mhpmevent`*n*`h` CSRs return bits 63-32 of the corresponding register, and writes change only bits 63-32. The `mhpmevent`*n*`h` CSRs are provided only if the Sscofpmf extension is implemented.

3.1.11. Machine Counter-Enable (`mcounteren`) Register

The counter-enable `mcounteren` register is a 32-bit register that controls the availability of the hardware performance-monitoring counters to the next-lower privileged mode.

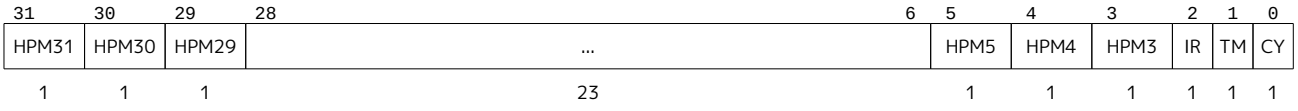


Figure 18. Counter-enable (`mcounteren`) register.

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 HPM*n* bit in the `mcounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcountern` register while executing in S-mode or U-mode will cause an illegal-instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode (S-mode if implemented, otherwise U-mode).

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

The `cycle`, `instret`, and `hpmcountern` CSRs are read-only shadows of `mcycle`, `minstret`, and `mhpmcounter n`, respectively. The `time` CSR is a read-only shadow of the memory-mapped `mtime` register. Analogously, when XLEN=32, the `cycleh`, `instreth` and `hpmcounternh` CSRs are read-only shadows of `mcycleh`, `minstreth` and `mhpmcounternh`, respectively. When XLEN=32, 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`.

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.

In harts with U-mode, the `mcounteren` must be implemented, but all fields are **WARL** and may be read-only zero, indicating reads to the corresponding counter will cause an illegal-instruction exception when executing in a less-privileged mode. In harts without U-mode, the `mcounteren` register should not exist.

3.1.12. Machine Counter-Inhibit (`mcountinhibit`) Register

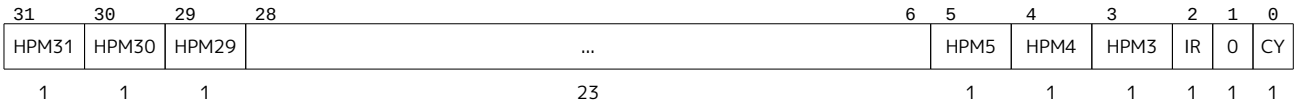


Figure 19. Counter-inhibit `mcountinhibit` register

The counter-inhibit register `mcountinhibit` is a 32-bit **WARL** register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; their accessibility is not affected by the setting of this register.

When the CY, IR, or HPM*n* bit in the `mcountinhibit` register is clear, the `mcycle`, `minstret`, or `mhpmcountern` register increments as usual. When the CY, IR, or HPM_n_ bit is set, the corresponding counter does not increment.

The `mcycle` CSR may be shared between harts on the same core, in which case the `mcountinhibit.CY` field is also shared between those harts, and so writes to `mcountinhibit.CY` will be visible to those harts.

If the `mcountinhibit` register is not implemented, the implementation behaves as though the register were set to zero.

When the `mcycle` and `minstret` counters are not needed, it is desirable to conditionally inhibit them to reduce energy consumption. Providing a single CSR to inhibit all counters also allows the counters to be atomically sampled.

Because the `mtime` counter can be shared between multiple cores, it cannot be inhibited with the `mcountinhibit` mechanism.

3.1.13. Machine Scratch (`mscratch`) Register

The `mscratch` register is an MXLEN-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

Figure 20. Machine-mode scratch register.

The MIPS ISA allocated two user registers (`k0`/`k1`) for use by the operating system. Although the MIPS scheme provides a fast and simple implementation, it also reduces available user registers, and does not scale to further privilege levels, or nested traps. It can also require both registers are cleared before returning to user level to avoid a potential security hole and to provide deterministic debugging behavior.

The RISC-V user ISA was designed to support many possible privileged system environments and so we did not want to infect the user-level ISA with any OS-dependent features. The RISC-V CSR swap instructions can quickly save/restore values to the `mscratch` register. Unlike the MIPS design, the OS can rely on holding a value in the `mscratch` register while the user context is running.

3.1.14. Machine Exception Program Counter (`mepc`) Register

`mepc` is an MXLEN-bit read/write register formatted as shown in Figure 21. The low bit of `mepc` (`mepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`mepc[1:0]`) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR `misa`, for example), then, whenever IALIGN=32, bit `mepc[1]` is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MRET instruction. Though masked, `mepc[1]` remains writable when IALIGN=32.

`mepc` is a **WARL** register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing `mepc`, implementations may convert an invalid address into some other invalid address that `mepc` is capable of holding.

When address translation is not in effect, virtual addresses and physical addresses are equal. Hence, the set of addresses `mepc` must be able to represent includes the set of physical addresses that can be used as a valid `pc` or effective address.

When a trap is taken into M-mode, `mepc` is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, `mepc` is never written by the implementation, though it may be explicitly written by software.

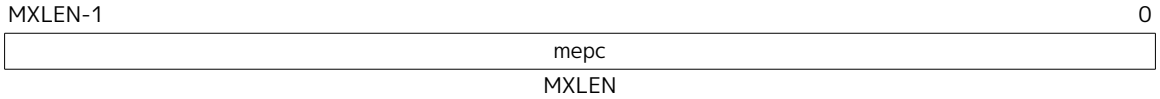


Figure 21. Machine exception program counter register.

3.1.15. Machine Cause (`mcause`) Register

The `mcause` register is an MXLEN-bit read-write register formatted as shown in Figure 22. When a trap is taken into M-mode, `mcause` is written with a code indicating the event that caused the trap. Otherwise, `mcause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `mcause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Table 14 lists the possible machine-level

exception codes. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

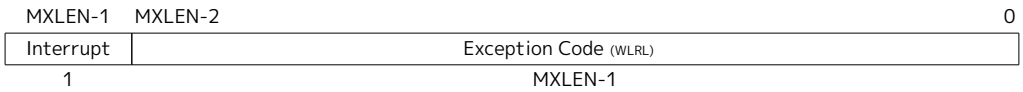


Figure 22. Machine Cause (`mcause`) register.

Note that load and load-reserved instructions generate load exceptions, whereas store, store-conditional, and AMO instructions generate store/AMO exceptions.

> *Interrupts can be separated from other traps with a single branch on the sign of the `mcause` register value. A shift left can remove the interrupt bit and scale the exception codes to index into a trap vector table.*

> *We do not distinguish privileged instruction exceptions from illegal-instruction exceptions. This simplifies the architecture and also hides details of which higher-privilege instructions are supported by an implementation. The privilege level servicing the trap can implement a policy on whether these need to be distinguished, and if so, whether a given opcode should be treated as illegal or privileged.*

If an instruction may raise multiple synchronous exceptions, the decreasing priority order of Table 15 indicates which exception is taken and reported in `mcause`. The priority of any custom synchronous exceptions is implementation-defined.

Table 14. Machine cause (`mcause`) register values after trap.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12	*Reserved*
1	13	Counter-overflow interrupt
1	14-15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16	Double trap
0	17	*Reserved*
0	18	Software check
0	19	Hardware error
0	20-23	*Reserved*
0	24-31	*Designated for custom use*
0	32-47	*Reserved*
0	48-63	*Designated for custom use*
0	≥64	*Reserved*

Table 15. Synchronous exception priority in decreasing priority order.

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
	12, 1	During instruction address translation: First encountered page fault or access fault
	1	With physical address for instruction: Instruction access fault
	2 0 8,9,11 3 3	Illegal instruction Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	4,6	Optionally: Load/store/AMO address misaligned
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault
	5,7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4,6	If not higher priority: Load/store/AMO address misaligned

When a virtual address is translated into a physical address, the address translation algorithm determines what specific exception may be raised.

Load/store/AMO address-misaligned exceptions may have either higher or lower priority than load/store/AMO page-fault and access-fault exceptions.

The relative priority of load/store/AMO address-misaligned and page-fault exceptions is implementation-defined to flexibly cater to two design points. Implementations that never support misaligned accesses can unconditionally raise the misaligned-address exception without performing address translation or protection checks. Implementations that support misaligned accesses only to some physical addresses must translate and check the address before determining whether the misaligned access may proceed, in which case raising the page-fault exception or access is more appropriate.

Instruction address breakpoints have the same cause value as, but different priority than, data address breakpoints (a.k.a. watchpoints) and environment break exceptions (which are raised by the EBREAK instruction).

Instruction address misaligned exceptions are raised by control-flow instructions with misaligned targets, rather than by the act of fetching an instruction. Therefore, these exceptions have lower priority than other instruction address exceptions.

A Software Check exception is a synchronous exception that is triggered when there are violations of checks and assertions defined by ISA extensions that aim to safeguard the integrity of software assets, including e.g. control-flow and memory-access constraints. When this exception is raised, the `xtval` register is set either to 0 or to an informative value defined by the extension that stipulated the exception be raised. The priority of this exception, relative to other synchronous exceptions, depends on the cause of this exception and is defined by the

extension that stipulated the exception be raised.

A Hardware Error exception is a synchronous exception triggered when corrupted or uncorrectable data is accessed explicitly or implicitly by an instruction. In this context, "data" encompasses all types of information used within a RISC-V hart. Upon a hardware error exception, the `xepc` *register is set to the address of the instruction that attempted to access corrupted data, while the* `xtval` *register is set either to 0 or to the virtual address of an instruction fetch, load, or store that attempted to access corrupted data. The priority of Hardware Error exception is implementation-defined, but any given occurrence is generally expected to be recognized at the point in the overall priority order at which the hardware error is discovered.*

3.1.16. Machine Trap Value (`mtval`) Register

The `mtval` register is an MXLEN-bit read-write register formatted as shown in Figure 23. When a trap is taken into M-mode, `mtval` is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, `mtval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `mtval` informatively, which may unconditionally set it to zero, and which may exhibit either behavior, depending on the underlying event that caused the exception. If the hardware platform specifies that no exceptions set `mtval` to a nonzero value, then `mtval` is read-only zero.

If `mtval` is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, then `mtval` will contain the faulting virtual address.

When page-based virtual memory is enabled, `mtval` is written with the faulting virtual address, even for physical-memory access-fault exceptions. This design reduces datapath cost for most implementations, particularly those with hardware page-table walkers.

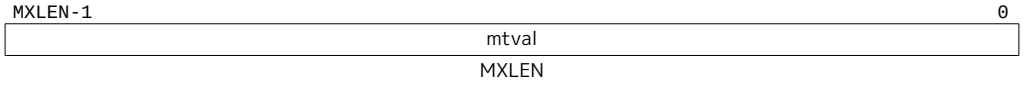


Figure 23. Machine Trap Value (`mtval`) register.

If `mtval` is written with a nonzero value when a misaligned load or store causes an access-fault or page-fault exception, then `mtval` will contain the virtual address of the portion of the access that caused the fault.

If `mtval` is written with a nonzero value when an instruction access-fault or page-fault exception occurs on a hart with variable-length instructions, then `mtval` will contain the virtual address of the portion of the instruction that caused the fault, while `mepc` will point to the beginning of the instruction.

The `mtval` register can optionally also be used to return the faulting instruction bits on an illegal-instruction exception (`mepc` points to the faulting instruction in memory). If `mtval` is written with a nonzero value when an illegal-instruction exception occurs, then `mtval` will contain the shortest of:

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first MXLEN bits of the faulting instruction

The value loaded into `mtval` on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

Capturing the faulting instruction in `mtval` *reduces the overhead of instruction emulation, potentially avoiding several partial instruction loads if the instruction is misaligned, and likely*

> *data cache misses or slow uncached accesses when loads are used to fetch the instruction into a data register. There is also a problem of atomicity if another agent is manipulating the instruction memory, as might occur in a dynamic translation system.*
>
> *A requirement is that the entire instruction (or at least the first MXLEN bits) are fetched into* `mtval` *before taking the trap. This should not constrain implementations, which would typically fetch the entire instruction before attempting to decode the instruction, and avoids complicating software handlers.*
>
> *A value of zero in* `mtval` *signifies either that the feature is not supported, or an illegal zero instruction was fetched. A load from the instruction memory pointed to by* `mepc` *can be used to distinguish these two cases (or alternatively, the system configuration information can be interrogated to install the appropriate trap handling before runtime).*

On a trap caused by a software check exception, the `mtval` register holds the cause for the exception. The following encodings are defined:

- 0 - No information provided.
- 2 - Landing Pad Fault. Defined by the Zicfilp extension (Section 20.1).
- 3 - Shadow Stack Fault. Defined by the Zicfiss extension (Section 20.2).

For other traps, `mtval` is set to zero, but a future standard may redefine `mtval`'s setting for other traps.

If `mtval` is not read-only zero, it is a **WARL** register that must be able to hold all valid virtual addresses and the value zero. It need not be capable of holding all possible invalid addresses. Prior to writing `mtval`, implementations may convert an invalid address into some other invalid address that `mtval` is capable of holding. If the feature to return the faulting instruction bits is implemented, `mtval` must also be able to hold all values less than 2^N, where N is the smaller of MXLEN and ILEN.

3.1.17. Machine Configuration Pointer (`mconfigptr`) Register

The `mconfigptr` register is an MXLEN-bit read-only CSR, formatted as shown in Figure 24, that holds the physical address of a configuration data structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration.

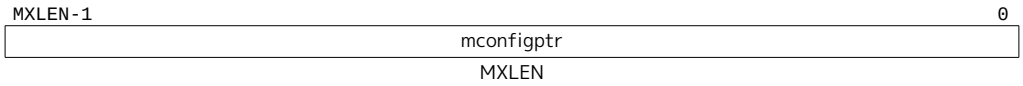


Figure 24. Machine Configuration Pointer (`mconfigptr`) register.

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

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

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

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

3.1.18. Machine Environment Configuration (`menvcfg`) Register

The `menvcfg` CSR is a 64-bit read/write register, formatted as shown in Figure 25, that controls certain characteristics of the execution environment for modes less privileged than M.

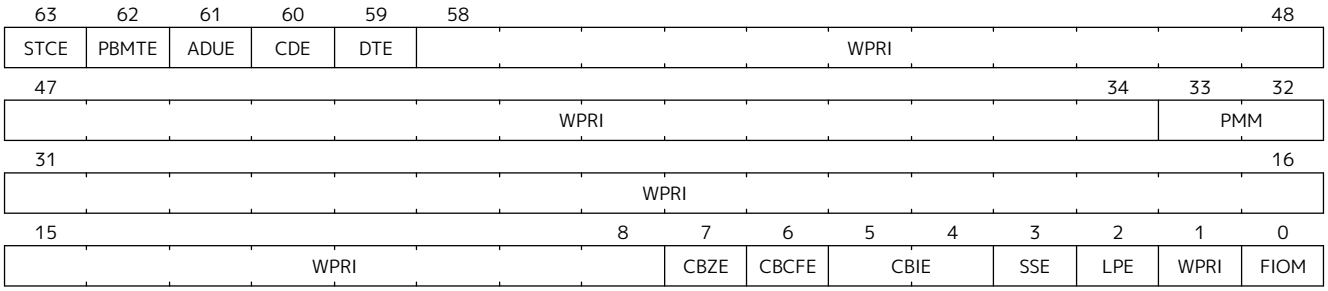


Figure 25. Machine environment configuration (`menvcfg`) register.

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 16 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 16. Modified interpretation of FENCE predecessor and successor sets for modes less privileged than M when FIOM=1.

Instruction bit	Meaning when set
PI PO	Predecessor device input and memory reads (PR implied) Predecessor device output and memory writes (PW implied)
SI SO	Successor device input and memory reads (SR implied) Successor device output and memory writes (SW implied)

Bit FIOM is needed in `menvcfg` so M-mode can emulate the hypervisor extension of Chapter 19, 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 Section 19.5.3 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 Chapter 9. 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 is furnished by the Smnpm 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.
- `SSAMOSWAP.W/D` raises an illegal-instruction exception.

When `menvcfg.SSE` is 0, the `henvcfg.SSE` and `senvcfg.SSE` fields are read-only zero.

The Ssdbltrp extension adds the double-trap-enable (`DTE`) field in `menvcfg`. When `menvcfg.DTE` is zero, the implementation behaves as though Ssdbltrp is not implemented. When Ssdbltrp is not implemented `sstatus.SDT`, `vsstatus.SDT`, and `henvcfg.DTE` bits are read-only zero.

When XLEN=32, `menvcfgh` is a 32-bit read/write register that aliases bits 63:32 of `menvcfg`. The `menvcfgh` register does not exist when XLEN=64.

If U-mode is not supported, then registers `menvcfg` and `menvcfgh` do not exist.

3.1.19. Machine Security Configuration (`mseccfg`) Register

`mseccfg` is an optional 64-bit read/write register, formatted as shown in Figure 26, that controls security features.

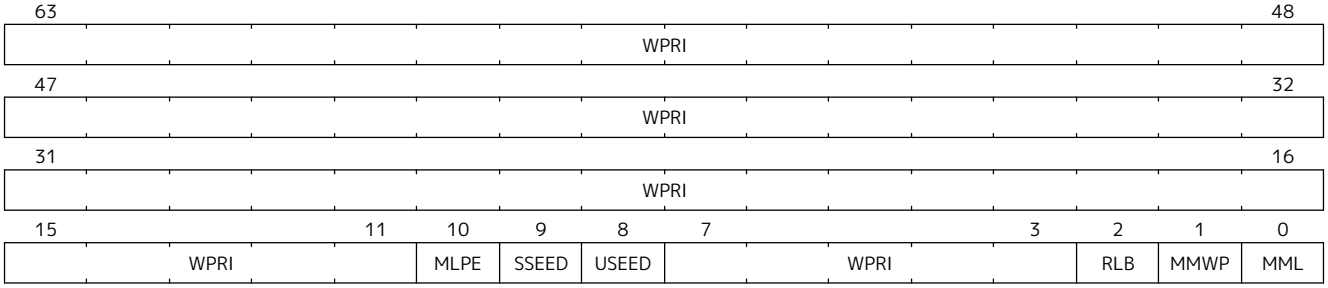


Figure 26. Machine security configuration (`mseccfg`) register.

The definitions of the SSEED and USEED fields are furnished by the entropy-source extension, Zkr.

The definitions of the RLB, MMWP, and MML fields are furnished by the PMP-enhancement extension, Smepmp.

The definition of the PMM field is furnished by the Smmpm extension.

The Zicfilp extension adds the `MLPE` field in `mseccfg`. When `MLPE` field is 1, Zicfilp extension is enabled in M-mode. When the `MLPE` field is 0, the Zicfilp extension is not enabled in M-mode and the following rules apply to M-mode.

- The hart does not update the `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

When XLEN=32 only, `mseccfgh` is a 32-bit read/write register that aliases bits 63:32 of `mseccfg`. Register `mseccfgh` does not exist when XLEN=64.

3.2. Machine-Level Memory-Mapped Registers

3.2.1. Machine Timer (`mtime` and `mtimecmp`) Registers

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode read-write register, `mtime`. `mtime` must increment at constant frequency, and the platform must provide a mechanism for determining the period of an `mtime` tick. The `mtime` register will wrap around if the count overflows.

The `mtime` register has a 64-bit precision on all RV32 and RV64 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (`mtimecmp`). A machine timer interrupt becomes pending whenever `mtime` contains a value greater than or equal to `mtimecmp`, treating the values as unsigned integers. The interrupt remains posted until `mtimecmp` becomes greater than `mtime` (typically as a result of writing `mtimecmp`). The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

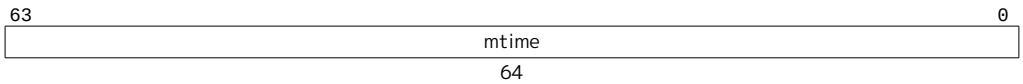


Figure 27. Machine time register (memory-mapped control register).

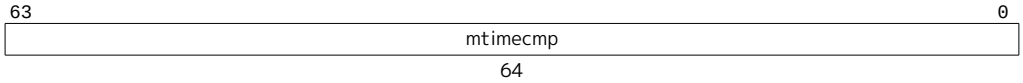


Figure 28. Machine time compare register (memory-mapped control register).

The timer facility is defined to use wall-clock time rather than a cycle counter to support modern processors that run with a highly variable clock frequency to save energy through dynamic voltage and frequency scaling.

Accurate real-time clocks (RTCs) are relatively expensive to provide (requiring a crystal or MEMS oscillator) and have to run even when the rest of system is powered down, and so there is usually only one in a system located in a different frequency/voltage domain from the processors. Hence, the RTC must be shared by all the harts in a system and accesses to the RTC will potentially incur the penalty of a voltage-level-shifter and clock-domain crossing. It is thus more natural to expose `mtime` *as a memory-mapped register than as a CSR.*

Lower privilege levels do not have their own `timecmp` *registers. Instead, machine-mode software can implement any number of virtual timers on a hart by multiplexing the next timer interrupt into the* `mtimecmp` *register.*

Simple fixed-frequency systems can use a single clock for both cycle counting and wall-clock time.

If the result of the comparison between `mtime` and `mtimecmp` changes, it is guaranteed to be reflected in MTIP eventually, but not necessarily immediately.

A spurious timer interrupt might occur if an interrupt handler increments `mtimecmp` *then immediately returns, because MTIP might not yet have fallen in the interim. All software should be written to assume this event is possible, but most software should assume this event is extremely unlikely. It is almost always more performant to incur an occasional spurious timer interrupt than to poll MTIP until it falls.*

In RV32, memory-mapped writes to `mtimecmp` modify only one 32-bit part of the register. The following code sequence sets a 64-bit `mtimecmp` value without spuriously generating a timer interrupt due to the intermediate value of the comparand:

For RV64, naturally aligned 64-bit memory accesses to the `mtime` and `mtimecmp` registers are additionally supported and are atomic.

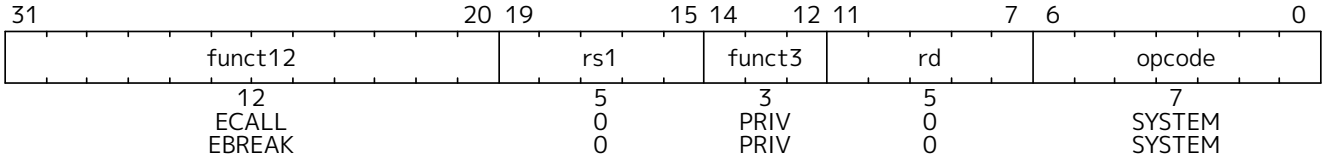
```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1)     # No smaller than old value.
sw a1, 4(t1)     # No smaller than new value.
sw a0, 0(t1)     # New value.
```

Sample code for setting the 64-bit time comparand in RV32 assuming a little-endian memory system and that the registers live in a strongly ordered I/O region. Storing -1 to the low-order bits of `mtimecmp` *prevents* `mtimecmp` *from temporarily becoming smaller than the lesser of the old and new values.*

The `time` CSR is a read-only shadow of the memory-mapped `mtime` register. When XLEN=32, 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`. When `mtime` changes, it is guaranteed to be reflected in `time` and `timeh` eventually, but not necessarily immediately.

3.3. Machine-Mode Privileged Instructions

3.3.1. Environment Call and Breakpoint



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

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

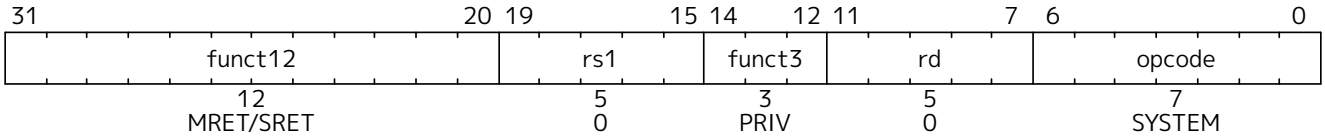
The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. Unless overridden by an external debug environment, EBREAK raises a breakpoint exception and performs no other operation.

As described in the "C" Standard Extension for Compressed Instructions in Volume I of this manual, the C.EBREAK instruction performs the same operation as the EBREAK instruction.

ECALL and EBREAK cause the receiving privilege mode's `epc` register to be set to the address of the ECALL or EBREAK instruction itself, *not* the address of the following instruction. As ECALL and EBREAK cause synchronous exceptions, they are not considered to retire, and should not increment the `minstret` CSR.

3.3.2. Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.



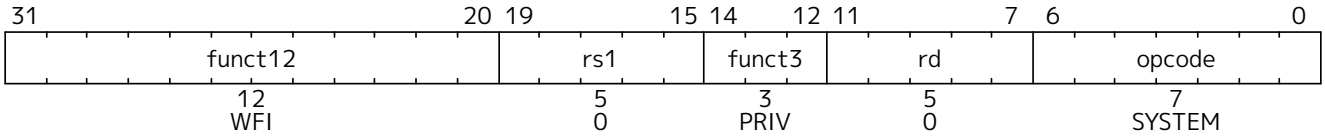
To return after handling a trap, there are separate trap return instructions per privilege level, MRET and SRET. MRET is always provided. SRET must be provided if supervisor mode is supported, and should raise an illegal-instruction exception otherwise. SRET should also raise an illegal-instruction exception when TSR=1 in `mstatus`, as described in Section 3.1.6.6. An *x*RET instruction can be executed in privilege mode *x* or higher, where executing a lower-privilege *x*RET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. In addition to manipulating the privilege stack as described in Section 3.1.6.1, *x*RET sets the `pc` to the value stored in the `xepc` register.

If the A extension is supported, the *x*RET instruction is allowed to clear any outstanding LR address reservation but is not required to. Trap handlers should explicitly clear the reservation if required (e.g., by using a dummy SC) before executing the *x*RET.

If xRET instructions always cleared LR reservations, it would be impossible to single-step through LR/SC sequences using a debugger.

3.3.3. Wait for Interrupt

The Wait for Interrupt instruction (WFI) informs the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart. WFI is available in all privileged modes, and optionally available to U-mode. This instruction may raise an illegal-instruction exception when TW=1 in `mstatus`, as described in Section 3.1.6.6.



If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt trap will be taken on the following instruction, i.e., execution resumes in the trap handler and `mepc` = `pc` + 4.

The following instruction takes the interrupt trap so that a simple return from the trap handler will execute code after the WFI instruction.

Implementations are permitted to resume execution for any reason, even if an enabled interrupt has not become pending. Hence, a legal implementation is to simply implement the WFI instruction as a NOP.

If the implementation does not stall the hart on execution of the instruction, then the interrupt will be taken on some instruction in the idle loop containing the WFI, and on a simple return from the handler, the idle loop will resume execution.

The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in `mstatus` (MIE and SIE) and the delegation register `mideleg` (i.e., the hart must resume if a locally enabled interrupt becomes pending, even if it has been delegated to a less-privileged mode), but should honor the individual interrupt enables (e.g, MTIE) (i.e., implementations should avoid resuming the hart if the interrupt is pending but not individually enabled). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level.

If the event that causes the hart to resume execution does not cause an interrupt to be taken, execution will resume at `pc` + 4, and software must determine what action to take, including looping back to repeat the WFI if there was no actionable event.

By allowing wakeup when interrupts are disabled, an alternate entry point to an interrupt handler can be called that does not require saving the current context, as the current context can be saved or discarded before the WFI is executed.

As implementations are free to implement WFI as a NOP, software must explicitly check for any relevant pending but disabled interrupts in the code following an WFI, and should loop back to the WFI if no suitable interrupt was detected. The `mip` or `sip` registers can be interrogated to determine the presence of any interrupt in machine or supervisor mode respectively.

The operation of WFI is unaffected by the delegation register settings.

WFI is defined so that an implementation can trap into a higher privilege mode, either immediately on encountering the WFI or after some interval to initiate a machine-mode transition to a lower power state, for example.

The same "wait-for-event" template might be used for possible future extensions that wait on memory locations changing, or message arrival.

3.3.4. Custom SYSTEM Instructions

The subspace of the SYSTEM major opcode shown in Figure 29 is designated for custom use. It is recommended that these instructions use bits 29:28 to designate the minimum required privilege mode, as do other SYSTEM instructions.

31 26	25 15	14 12	11 7	6 0	
funct6	custom	funct3	custom	opcode	Recommended Purpose
6	11	3	5	7	
100011	custom	0	custom	SYSTEM	Unprivileged or User-Level
110011	custom	0	custom	SYSTEM	Unprivileged or User-Level
100111	custom	0	custom	SYSTEM	Supervisor-Level
110111	custom	0	custom	SYSTEM	Supervisor-Level
101011	custom	0	custom	SYSTEM	Hypervisor-Level
111011	custom	0	custom	SYSTEM	Hypervisor-Level
101111	custom	0	custom	SYSTEM	Machine-Level
111111	custom	0	custom	SYSTEM	Machine-Level

Figure 29. SYSTEM instruction encodings designated for custom use.

3.4. Reset

Upon reset, a hart's privilege mode is set to M. The `mstatus` fields MIE and MPRV are reset to 0. If little-endian memory accesses are supported, the `mstatus`/`mstatush` field MBE is reset to 0. The `misa` register is reset to enable the maximal set of supported extensions, as described in Section 3.1.1. For implementations with the "A" standard extension, there is no valid load reservation. The `pc` is set to an implementation-defined reset vector. The `mcause` register is set to a value indicating the cause of the reset. Writable PMP registers' A and L fields are set to 0, unless the platform mandates a different reset value for some PMP registers' A and L fields. If the hypervisor extension is implemented, the `hgatp`.MODE and `vsatp`.MODE fields are reset to 0. If the Smrnmi extension is implemented, the `mnstatus`.NMIE field is reset to 0. No **WARL** field contains an illegal value. If the Zicfilp extension is implemented, the `mseccfg`.MLPE field is reset to 0. All other hart state is UNSPECIFIED.

The `mcause` values after reset have implementation-specific interpretation, but the value 0 should be returned on implementations that do not distinguish different reset conditions. Implementations that distinguish different reset conditions should only use 0 to indicate the most complete reset.

Some designs may have multiple causes of reset (e.g., power-on reset, external hard reset, brownout detected, watchdog timer elapse, sleep-mode wakeup), which machine-mode software and debuggers may wish to distinguish.

`mcause` *reset values may alias* `mcause` *values following synchronous exceptions. There should be no ambiguity in this overlap, since on reset the* `pc` *is typically set to a different value than on other traps.*

3.5. Non-Maskable Interrupts

Non-maskable interrupts (NMIs) are only used for hardware error conditions, and cause an immediate jump to an implementation-defined NMI vector running in M-mode regardless of the state of a hart's interrupt enable bits. The `mepc` register is written with the virtual address of the instruction that was interrupted, and `mcause` is set to a value indicating the source of the NMI. The NMI can thus overwrite state

in an active machine-mode interrupt handler.

The values written to `mcause` on an NMI are implementation-defined. The high Interrupt bit of `mcause` should be set to indicate that this was an interrupt. An Exception Code of 0 is reserved to mean "unknown cause" and implementations that do not distinguish sources of NMIs via the `mcause` register should return 0 in the Exception Code.

Unlike resets, NMIs do not reset processor state, enabling diagnosis, reporting, and possible containment of the hardware error.

3.6. Physical Memory Attributes

The physical memory map for a complete system includes various address ranges, some corresponding to memory regions and some to memory-mapped control registers, portions of which might not be accessible. Some memory regions might not support reads, writes, or execution; some might not support subword or subblock accesses; some might not support atomic operations; and some might not support cache coherence or might have different memory models. Similarly, memory-mapped control registers vary in their supported access widths, support for atomic operations, and whether read and write accesses have associated side effects. In RISC-V systems, these properties and capabilities of each region of the machine's physical address space are termed *physical memory attributes* (PMAs). This section describes RISC-V PMA terminology and how RISC-V systems implement and check PMAs.

PMAs are inherent properties of the underlying hardware and rarely change during system operation. Unlike physical memory protection values described in Section 3.7, PMAs do not vary by execution context. The PMAs of some memory regions are fixed at chip design time—for example, for an on-chip ROM. Others are fixed at board design time, depending, for example, on which other chips are connected to off-chip buses. Off-chip buses might also support devices that could be changed on every power cycle (cold pluggable) or dynamically while the system is running (hot pluggable). Some devices might be configurable at run time to support different uses that imply different PMAs—for example, an on-chip scratchpad RAM might be cached privately by one core in one end-application, or accessed as a shared non-cached memory in another end-application.

Most systems will require that at least some PMAs are dynamically checked in hardware later in the execution pipeline after the physical address is known, as some operations will not be supported at all physical memory addresses, and some operations require knowing the current setting of a configurable PMA attribute. While many other architectures specify some PMAs in the virtual memory page tables and use the TLB to inform the pipeline of these properties, this approach injects platform-specific information into a virtualized layer and can cause system errors unless attributes are correctly initialized in each page-table entry for each physical memory region. In addition, the available page sizes might not be optimal for specifying attributes in the physical memory space, leading to address-space fragmentation and inefficient use of expensive TLB entries.

For RISC-V, we separate out specification and checking of PMAs into a separate hardware structure, the *PMA checker*. In many cases, the attributes are known at system design time for each physical address region, and can be hardwired into the PMA checker. Where the attributes are run-time configurable, platform-specific memory-mapped control registers can be provided to specify these attributes at a granularity appropriate to each region on the platform (e.g., for an on-chip SRAM that can be flexibly divided between cacheable and uncacheable uses). PMAs are checked for any access to physical memory, including accesses that have undergone virtual to physical memory translation. To aid in system debugging, we strongly recommend that, where possible, RISC-V processors precisely trap physical memory accesses that fail PMA checks. Precisely trapped PMA violations manifest as instruction, load, or store access-fault exceptions, distinct from virtual-memory page-fault exceptions. Precise PMA traps might not always be possible, for example, when probing a legacy bus architecture that uses access failures as part of the discovery mechanism. In this case, error responses from peripheral devices will be reported as

imprecise bus-error interrupts.

PMAs must also be readable by software to correctly access certain devices or to correctly configure other hardware components that access memory, such as DMA engines. As PMAs are tightly tied to a given physical platform's organization, many details are inherently platform-specific, as is the means by which software can learn the PMA values for a platform. Some devices, particularly legacy buses, do not support discovery of PMAs and so will give error responses or time out if an unsupported access is attempted. Typically, platform-specific machine-mode code will extract PMAs and ultimately present this information to higher-level less-privileged software using some standard representation.

Where platforms support dynamic reconfiguration of PMAs, an interface will be provided to set the attributes by passing requests to a machine-mode driver that can correctly reconfigure the platform. For example, switching cacheability attributes on some memory regions might involve platform-specific operations, such as cache flushes, that are available only to machine-mode.

3.6.1. Main Memory versus I/O Regions

The most important characterization of a given memory address range is whether it holds regular main memory or I/O devices. Regular main memory is required to have a number of properties, specified below, whereas I/O devices can have a much broader range of attributes. Memory regions that do not fit into regular main memory, for example, device scratchpad RAMs, are categorized as I/O regions.

What previous versions of this specification termed vacant regions are no longer a distinct category; they are now described as I/O regions that are not accessible (i.e. lacking read, write, and execute permissions). Main memory regions that are not accessible are also allowed.

3.6.2. Supported Access Type PMAs

Access types specify which access widths, from 8-bit byte to long multi-word burst, are supported, and also whether misaligned accesses are supported for each access width.

Although software running on a RISC-V hart cannot directly generate bursts to memory, software might have to program DMA engines to access I/O devices and might therefore need to know which access sizes are supported.

Main memory regions always support read and write of all access widths required by the attached devices, and can specify whether instruction fetch is supported.

Some platforms might mandate that all of main memory support instruction fetch. Other platforms might prohibit instruction fetch from some main memory regions.

In some cases, the design of a processor or device accessing main memory might support other widths, but must be able to function with the types supported by the main memory.

I/O regions can specify which combinations of read, write, or execute accesses to which data widths are supported.

For systems with page-based virtual memory, I/O and memory regions can specify which combinations of hardware page-table reads and hardware page-table writes are supported.

Unix-like operating systems generally require that all of cacheable main memory supports page-table walks.

3.6.3. Atomicity PMAs

Atomicity PMAs describes which atomic instructions are supported in this address region. Support for atomic instructions is divided into two categories: *LR/SC* and *AMOs*.

Some platforms might mandate that all of cacheable main memory support all atomic operations required by the attached processors.

3.6.3.1. AMO PMA

Within AMOs, there are four levels of support: *AMONone*, *AMOSwap*, *AMOLogical*, and *AMOArithmetic*. AMONone indicates that no AMO operations are supported. AMOSwap indicates that only `amoswap` instructions are supported in this address range. AMOLogical indicates that swap instructions plus all the logical AMOs (`amoand`, `amoor`, `amoxor`) are supported. AMOArithmetic indicates that all RISC-V AMOs are supported. For each level of support, naturally aligned AMOs of a given width are supported if the underlying memory region supports reads and writes of that width. Main memory and I/O regions may only support a subset or none of the processor-supported atomic operations.

Table 17. Classes of AMOs supported by I/O regions.

AMO Class	Supported Operations
AMONone	*None*
AMOSwap	`amoswap`
AMOLogical	above + `amoand`, `amoor`, `amoxor`
AMOArithmetic	above + `amoadd`, `amomin`, `amomax`, `amominu`, `amomaxu`

We recommend providing at least AMOLogical support for I/O regions where possible.

3.6.3.2. Reservability PMA

For *LR/SC*, there are three levels of support indicating combinations of the reservability and eventuality properties: *RsrvNone*, *RsrvNonEventual*, and *RsrvEventual*. RsrvNone indicates that no LR/SC operations are supported (the location is non-reservable). RsrvNonEventual indicates that the operations are supported (the location is reservable), but without the eventual success guarantee described in the unprivileged ISA specification. RsrvEventual indicates that the operations are supported and provide the eventual success guarantee.

We recommend providing RsrvEventual support for main memory regions where possible. Most I/O regions will not support LR/SC accesses, as these are most conveniently built on top of a cache-coherence scheme, but some may support RsrvNonEventual or RsrvEventual.

When LR/SC is used for memory locations marked RsrvNonEventual, software should provide alternative fall-back mechanisms used when lack of progress is detected.

3.6.4. Misaligned Atomicity Granule PMA

The misaligned atomicity granule PMA provides constrained support for misaligned AMOs. This PMA, if present, specifies the size of a *misaligned atomicity granule*, a naturally aligned power-of-two number of bytes. Specific supported values for this PMA are represented by MAG*NN*, e.g., MAG16 indicates the misaligned atomicity granule is at least 16 bytes.

The misaligned atomicity granule PMA applies only to AMOs, loads and stores defined in the base ISAs,

and loads and stores of no more than XLEN bits defined in the F, D, and Q extensions. For an instruction in that set, if all accessed bytes lie within the same misaligned atomicity granule, the instruction will not raise an exception for reasons of address alignment, and the instruction will give rise to only one memory operation for the purposes of RVWMO—i.e., it will execute atomically.

If a misaligned AMO accesses a region that does not specify a misaligned atomicity granule PMA, or if not all accessed bytes lie within the same misaligned atomicity granule, then an exception is raised. For regular loads and stores that access such a region or for which not all accessed bytes lie within the same atomicity granule, then either an exception is raised, or the access proceeds but is not guaranteed to be atomic. Implementations may raise access-fault exceptions instead of address-misaligned exceptions for some misaligned accesses, indicating the instruction should not be emulated by a trap handler.

LR/SC instructions are unaffected by this PMA and so always raise an exception when misaligned. Vector memory accesses are also unaffected, so might execute non-atomically even when contained within a misaligned atomicity granule. Implicit accesses are similarly unaffected by this PMA.

3.6.5. Memory-Ordering PMAs

Regions of the address space are classified as either *main memory* or *I/O* for the purposes of ordering by the FENCE instruction and atomic-instruction ordering bits.

Accesses by one hart to main memory regions are observable not only by other harts but also by other devices with the capability to initiate requests in the main memory system (e.g., DMA engines). Coherent main memory regions always have either the RVWMO or RVTSO memory model. Incoherent main memory regions have an implementation-defined memory model.

Accesses by one hart to an I/O region are observable not only by other harts and bus mastering devices but also by the targeted I/O devices, and I/O regions may be accessed with either *relaxed* or *strong* ordering. Accesses to an I/O region with relaxed ordering are generally observed by other harts and bus mastering devices in a manner similar to the ordering of accesses to an RVWMO memory region, as discussed in Section A.4.2 in Volume I of this specification. By contrast, accesses to an I/O region with strong ordering are generally observed by other harts and bus mastering devices in program order.

Each strongly ordered I/O region specifies a numbered ordering channel, which is a mechanism by which ordering guarantees can be provided between different I/O regions. Channel 0 is used to indicate point-to-point strong ordering only, where only accesses by the hart to the single associated I/O region are strongly ordered.

Channel 1 is used to provide global strong ordering across all I/O regions. Any accesses by a hart to any I/O region associated with channel 1 can only be observed to have occurred in program order by all other harts and I/O devices, including relative to accesses made by that hart to relaxed I/O regions or strongly ordered I/O regions with different channel numbers. In other words, any access to a region in channel 1 is equivalent to executing a `fence io,io` instruction before and after the instruction.

Other larger channel numbers provide program ordering to accesses by that hart across any regions with the same channel number.

Systems might support dynamic configuration of ordering properties on each memory region.

Strong ordering can be used to improve compatibility with legacy device driver code, or to enable increased performance compared to insertion of explicit ordering instructions when the implementation is known to not reorder accesses.

Local strong ordering (channel 0) is the default form of strong ordering as it is often

straightforward to provide if there is only a single in-order communication path between the hart and the I/O device.

Generally, different strongly ordered I/O regions can share the same ordering channel without additional ordering hardware if they share the same interconnect path and the path does not reorder requests.

3.6.6. Coherence and Cacheability PMAs

Coherence is a property defined for a single physical address, and indicates that writes to that address by one agent will eventually be made visible to other coherent agents in the system. Coherence is not to be confused with the memory consistency model of a system, which defines what values a memory read can return given the previous history of reads and writes to the entire memory system. In RISC-V platforms, the use of hardware-incoherent regions is discouraged due to software complexity, performance, and energy impacts.

The cacheability of a memory region should not affect the software view of the region except for differences reflected in other PMAs, such as main memory versus I/O classification, memory ordering, supported accesses and atomic operations, and coherence. For this reason, we treat cacheability as a platform-level setting managed by machine-mode software only.

Where a platform supports configurable cacheability settings for a memory region, a platform-specific machine-mode routine will change the settings and flush caches if necessary, so the system is only incoherent during the transition between cacheability settings. This transitory state should not be visible to lower privilege levels.

Coherence is straightforward to provide for a shared memory region that is not cached by any agent. The PMA for such a region would simply indicate it should not be cached in a private or shared cache.

Coherence is also straightforward for read-only regions, which can be safely cached by multiple agents without requiring a cache-coherence scheme. The PMA for this region would indicate that it can be cached, but that writes are not supported.

Some read-write regions might only be accessed by a single agent, in which case they can be cached privately by that agent without requiring a coherence scheme. The PMA for such regions would indicate they can be cached. The data can also be cached in a shared cache, as other agents should not access the region.

If an agent can cache a read-write region that is accessible by other agents, whether caching or non-caching, a cache-coherence scheme is required to avoid use of stale values. In regions lacking hardware cache coherence (hardware-incoherent regions), cache coherence can be implemented entirely in software, but software coherence schemes are notoriously difficult to implement correctly and often have severe performance impacts due to the need for conservative software-directed cache-flushing. Hardware cache-coherence schemes require more complex hardware and can impact performance due to the cache-coherence probes, but are otherwise invisible to software.

For each hardware cache-coherent region, the PMA would indicate that the region is coherent and which hardware coherence controller to use if the system has multiple coherence controllers. For some systems, the coherence controller might be an outer-level shared cache, which might itself access further outer-level cache-coherence controllers hierarchically.

Most memory regions within a platform will be coherent to software, because they will be fixed

as either uncached, read-only, hardware cache-coherent, or only accessed by one agent.

If a PMA indicates non-cacheability, then accesses to that region must be satisfied by the memory itself, not by any caches.

For implementations with a cacheability-control mechanism, the situation may arise that a program uncacheably accesses a memory location that is currently cache-resident. In this situation, the cached copy must be ignored. This constraint is necessary to prevent more-privileged modes' speculative cache refills from affecting the behavior of less-privileged modes' uncacheable accesses.

3.6.7. Idempotency PMAs

Idempotency PMAs describe whether reads and writes to an address region are idempotent. Main memory regions are assumed to be idempotent. For I/O regions, idempotency on reads and writes can be specified separately (e.g., reads are idempotent but writes are not). If accesses are non-idempotent, i.e., there is potentially a side effect on any read or write access, then speculative or redundant accesses must be avoided.

For the purposes of defining the idempotency PMAs, changes in observed memory ordering created by redundant accesses are not considered a side effect.

While hardware should always be designed to avoid speculative or redundant accesses to memory regions marked as non-idempotent, it is also necessary to ensure software or compiler optimizations do not generate spurious accesses to non-idempotent memory regions.

Non-idempotent regions might not support misaligned accesses. Misaligned accesses to such regions should raise access-fault exceptions rather than address-misaligned exceptions, indicating that software should not emulate the misaligned access using multiple smaller accesses, which could cause unexpected side effects.

For non-idempotent regions, implicit reads and writes must not be performed early or speculatively, with the following exceptions. When a non-speculative implicit read is performed, an implementation is permitted to additionally read any of the bytes within a naturally aligned power-of-2 region containing the address of the non-speculative implicit read. Furthermore, when a non-speculative instruction fetch is performed, an implementation is permitted to additionally read any of the bytes within the *next* naturally aligned power-of-2 region of the same size (with the address of the region taken modulo 2^{XLEN}. The results of these additional reads may be used to satisfy subsequent early or speculative implicit reads. The size of these naturally aligned power-of-2 regions is implementation-defined, but, for systems with page-based virtual memory, must not exceed the smallest supported page size.

3.7. Physical Memory Protection

To support secure processing and contain faults, it is desirable to limit the physical addresses accessible by software running on a hart. An optional physical memory protection (PMP) unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. The PMP values are checked in parallel with the PMA checks described in Section 3.6.

The granularity of PMP access control settings are platform-specific, but the standard PMP encoding supports regions as small as four bytes. Certain regions' privileges can be hardwired—for example, some regions might only ever be visible in machine mode but in no lower-privilege layers.

Platforms vary widely in demands for physical memory protection, and some platforms may

provide other PMP structures in addition to or instead of the scheme described in this section.

PMP checks are applied to all accesses whose effective privilege mode is S or U, including instruction fetches and data accesses in S and U mode, and data accesses in M-mode when the MPRV bit in `mstatus` is set and the MPP field in `mstatus` contains S or U. PMP checks are also applied to page-table accesses for virtual-address translation, for which the effective privilege mode is S. Optionally, PMP checks may additionally apply to M-mode accesses, in which case the PMP registers themselves are locked, so that even M-mode software cannot change them until the hart is reset. In effect, PMP can *grant* permissions to S and U modes, which by default have none, and can *revoke* permissions from M-mode, which by default has full permissions.

PMP violations are always trapped precisely at the processor.

3.7.1. Physical Memory Protection CSRs

PMP entries are described by an 8-bit configuration register and one MXLEN-bit address register. Some PMP settings additionally use the address register associated with the preceding PMP entry. Up to 64 PMP entries are supported. Implementations may implement zero, 16, or 64 PMP entries; the lowest-numbered PMP entries must be implemented first. All PMP CSR fields are **WARL** and may be read-only zero. PMP CSRs are only accessible to M-mode.

The PMP configuration registers are densely packed into CSRs to minimize context-switch time. For RV32, sixteen CSRs, `pmpcfg0`–`pmpcfg15`, hold the configurations `pmp0cfg`–`pmp63cfg` for the 64 PMP entries, as shown in Figure 30. For RV64, eight even-numbered CSRs, `pmpcfg0`, `pmpcfg2`, ..., `pmpcfg14`, hold the configurations for the 64 PMP entries, as shown in Figure 31. For RV64, the odd-numbered configuration registers, `pmpcfg1`, `pmpcfg3`, ..., `pmpcfg15`, are illegal.

RV64 harts use `pmpcfg2`, rather than `pmpcfg1`, to hold configurations for PMP entries 8-15. This design reduces the cost of supporting multiple MXLEN values, since the configurations for PMP entries 8-11 appear in `pmpcfg2`[31:0] for both RV32 and RV64.

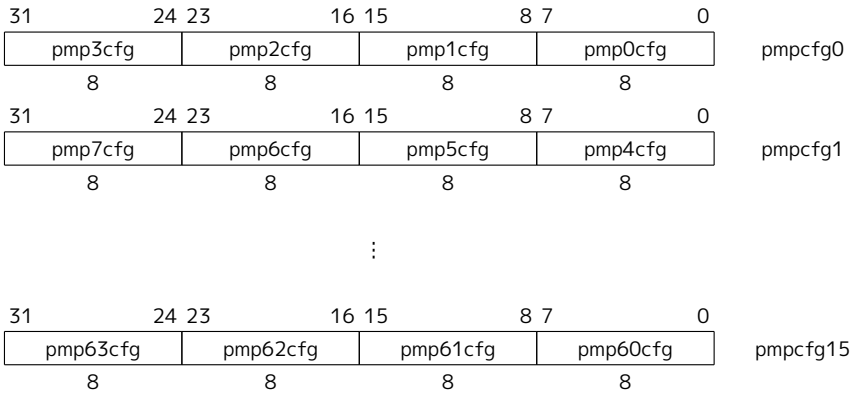


Figure 30. RV32 PMP configuration CSR layout.

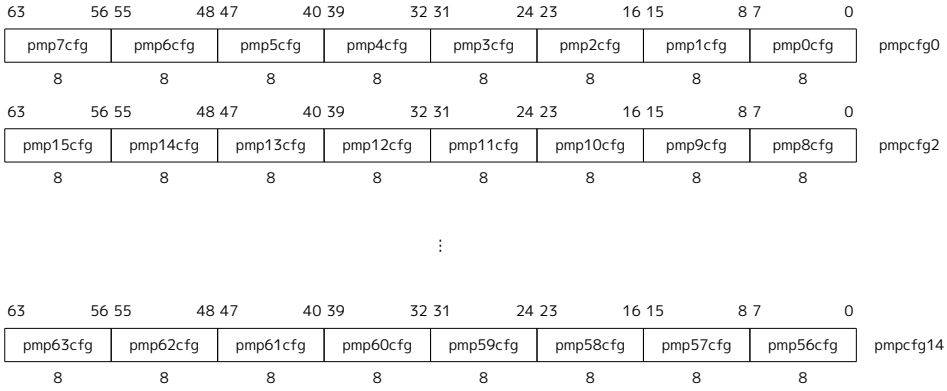


Figure 31. RV64 PMP configuration CSR layout.

The PMP address registers are CSRs named `pmpaddr0`-`pmpaddr63`. Each PMP address register encodes bits 33-2 of a 34-bit physical address for RV32, as shown in Figure 32. For RV64, each PMP address register encodes bits 55-2 of a 56-bit physical address, as shown in Figure 33. Not all physical address bits may be implemented, and so the `pmpaddr` registers are **WARL**.

The Sv32 page-based virtual-memory scheme described in Section 11.3 supports 34-bit physical addresses for RV32, so the PMP scheme must support addresses wider than XLEN for RV32. The Sv39 and Sv48 page-based virtual-memory schemes described in Section 11.4 and Section 11.5 support a 56-bit physical address space, so the RV64 PMP address registers impose the same limit.

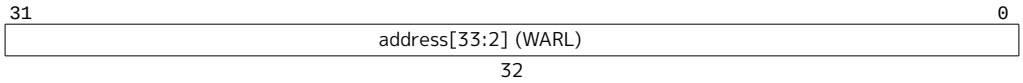


Figure 32. PMP address register format, RV32.

Figure 33. PMP address register format, RV64.

Figure 34 shows the layout of a PMP configuration register. The R, W, and X bits, when set, indicate that the PMP entry permits read, write, and instruction execution, respectively. When one of these bits is clear, the corresponding access type is denied. The R, W, and X fields form a collective **WARL** field for which the combinations with R=0 and W=1 are reserved. The remaining two fields, A and L, are described in the following sections.

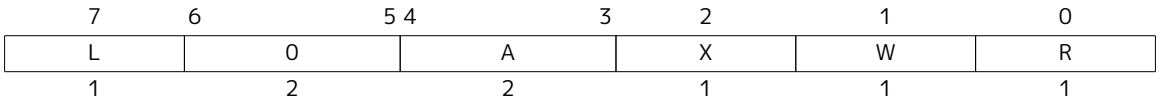


Figure 34. PMP configuration register format.

Attempting to fetch an instruction from a PMP region that does not have execute permissions raises an instruction access-fault exception. Attempting to execute a load or load-reserved instruction which accesses a physical address within a PMP region without read permissions raises a load access-fault exception. Attempting to execute a store, store-conditional, or AMO instruction which accesses a physical address within a PMP region without write permissions raises a store access-fault exception.

3.7.1.1. Address Matching

The A field in a PMP entry's configuration register encodes the address-matching mode of the associated PMP address register. The encoding of this field is shown in Table 18. When A=0, this PMP entry is

disabled and matches no addresses. Two other address-matching modes are supported: naturally aligned power-of-2 regions (NAPOT), including the special case of naturally aligned four-byte regions (NA4); and the top boundary of an arbitrary range (TOR). These modes support four-byte granularity.

Table 18. Encoding of A field in PMP configuration registers.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

NAPOT ranges make use of the low-order bits of the associated address register to encode the size of the range, as shown in Table 19.

Table 19. `NAPOT` *range encoding in PMP address and configuration registers.*

`pmpaddr`	`pmpcfg.A`	Match type and size
`yyyy…yyyy`	NA4	4-byte NAPOT range
`yyyy…yyy0`	NAPOT	8-byte NAPOT range
`yyyy…yy01`	NAPOT	16-byte NAPOT range
`yyyy…y011`	NAPOT	32-byte NAPOT range
…	…	…
`yy01…1111`	NAPOT	2^{XLEN}-byte NAPOT range
`y011…1111`	NAPOT	2^{XLEN+1}-byte NAPOT range
`0111…1111`	NAPOT	2^{XLEN+2}-byte NAPOT range
`1111…1111`	NAPOT	2^{XLEN+3}-byte NAPOT range

If TOR is selected, the associated address register forms the top of the address range, and the preceding PMP address register forms the bottom of the address range. If PMP entry *i*'s A field is set to TOR, the entry matches any address *y* such that $\texttt{pmpaddr}_{i-1} \le y < \texttt{pmpaddr}_i$ (irrespective of the value of $\texttt{pmpcfg}_{i-1}$). If PMP entry 0's A field is set to TOR, zero is used for the lower bound, and so it matches any address $y < \texttt{pmpaddr}_0$.

If $\texttt{pmpaddr}_{i-1} \ge \texttt{pmpaddr}_i$ *and* $\texttt{pmpcfg}_i\texttt{.A}$*=TOR, then PMP entry i matches no addresses.*

Although the PMP mechanism supports regions as small as four bytes, platforms may specify coarser PMP regions. In general, the PMP grain is 2^{G+2} bytes and must be the same across all PMP regions. When $G \ge 1$, the NA4 mode is not selectable. When $G \ge 2$ and $pmpcfg_i$.A[1] is set, i.e. the mode is NAPOT, then bits $pmpaddr_i$[G-2:0] read as all ones. When $G \ge 1$ and $pmpcfg_i$.A[1] is clear, i.e. the mode is OFF or TOR, then bits $pmpaddr_i$[G-1:0] read as all zeros. Bits $pmpaddr_i$[G-1:0] do not affect the TOR address-matching logic. Although changing $pmpcfg_i$.A[1] affects the value read from $pmpaddr_i$, it does not affect the underlying value stored in that register—in particular, $pmpaddr_i$[G-1] retains its original value when $pmpcfg_i$.A is changed from NAPOT to TOR/OFF then back to NAPOT.

Software may determine the PMP granularity by writing zero to `pmp0cfg`*, then writing all ones to* `pmpaddr0`*, then reading back* `pmpaddr0`*. If G is the index of the least-significant bit set, the PMP granularity is* 2^{G+2} *bytes.*

If the current XLEN is greater than MXLEN, the PMP address registers are zero-extended from MXLEN to XLEN bits for the purposes of address matching.

3.7.1.2. Locking and Privilege Mode

The L bit indicates that the PMP entry is locked, i.e., writes to the configuration register and associated address registers are ignored. Locked PMP entries remain locked until the hart is reset. If PMP entry *i* is locked, writes to `pmpicfg` and `pmpaddri` are ignored. Additionally, if PMP entry *i* is locked and `pmpicfg.A` is set

to TOR, writes to `pmpaddr`*i*-1 are ignored.

Setting the L bit locks the PMP entry even when the A field is set to OFF.

In addition to locking the PMP entry, the L bit indicates whether the R/W/X permissions are enforced on M-mode accesses. When the L bit is set, these permissions are enforced for all privilege modes. When the L bit is clear, any M-mode access matching the PMP entry will succeed; the R/W/X permissions apply only to S and U modes.

3.7.1.3. Priority and Matching Logic

PMP entries are statically prioritized. The lowest-numbered PMP entry that matches any byte of an access determines whether that access succeeds or fails. The matching PMP entry must match all bytes of an access, or the access fails, irrespective of the L, R, W, and X bits. For example, if a PMP entry is configured to match the four-byte range `0xC`–`0xF`, then an 8-byte access to the range `0x8`–`0xF` will fail, assuming that PMP entry is the highest-priority entry that matches those addresses.

If a PMP entry matches all bytes of an access, then the L, R, W, and X bits determine whether the access succeeds or fails. If the L bit is clear and the privilege mode of the access is M, the access succeeds. Otherwise, if the L bit is set or the privilege mode of the access is S or U, then the access succeeds only if the R, W, or X bit corresponding to the access type is set.

If no PMP entry matches an M-mode access, the access succeeds. If no PMP entry matches an S-mode or U-mode access, but at least one PMP entry is implemented, the access fails.

If at least one PMP entry is implemented, but all PMP entries' A fields are set to OFF, then all S-mode and U-mode memory accesses will fail.

Failed accesses generate an instruction, load, or store access-fault exception. Note that a single instruction may generate multiple accesses, which may not be mutually atomic. An access-fault exception is generated if at least one access generated by an instruction fails, though other accesses generated by that instruction may succeed with visible side effects. Notably, instructions that reference virtual memory are decomposed into multiple accesses.

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before an access-fault exception occurs. In particular, a portion of a misaligned store that passes the PMP check may become visible, even if another portion fails the PMP check. The same behavior may manifest for stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

3.7.2. Physical Memory Protection and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in Chapter 11. When paging is enabled, instructions that access virtual memory may result in multiple physical-memory accesses, including implicit references to the page tables. The PMP checks apply to all of these accesses. The effective privilege mode for implicit page-table accesses is S.

Implementations with virtual memory are permitted to perform address translations speculatively and earlier than required by an explicit memory access, and are permitted to cache them in address translation cache structures—including possibly caching the identity mappings from effective address to physical address used in Bare translation modes and M-mode. The PMP settings for the resulting physical address may be checked (and possibly cached) at any point between the address translation and the explicit memory access. Hence, when the PMP settings are modified, M-mode software must synchronize the PMP settings with the virtual memory system and any PMP or address-translation caches. This is accomplished

by executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0`, after the PMP CSRs are written. See Section 19.5.3 for additional synchronization requirements when the hypervisor extension is implemented.

If page-based virtual memory is not implemented, memory accesses check the PMP settings synchronously, so no SFENCE.VMA is needed.

Chapter 4. "Smstateen/Ssstateen" Extensions, Version 1.0

The implementation of optional RISC-V extensions has the potential to open covert channels between separate user threads, or between separate guest OSes running under a hypervisor. The problem occurs when an extension adds processor state — usually explicit registers, but possibly other forms of state — that the main OS or hypervisor is unaware of (and hence won't context-switch) but that can be modified/written by one user thread or guest OS and perceived/examined/read by another.

For example, the Advanced Interrupt Architecture (AIA) for RISC-V adds to a hart as many as ten supervisor-level CSRs (`siselect`, `sireg`, `stopi`, `sseteipnum`, `sclreipnum`, `sseteienum`, `sclreienum`, `sclaimei`, `sieh`, and `siph`) and provides also the option for hardware to be backward-compatible with older, pre-AIA software. Because an older hypervisor that is oblivious to the AIA will not know to swap any of the AIA's new CSRs on context switches, the registers may then be used as a covert channel between multiple guest OSes that run atop this hypervisor. Although traditional practices might consider such a communication channel harmless, the intense focus on security today argues that a means be offered to plug such channels.

The `f` registers of the RISC-V floating-point extensions and the `v` registers of the vector extension would similarly be potential covert channels between user threads, except for the existence of the FS and VS fields in the `sstatus` register. Even if an OS is unaware of, say, the vector extension and its `v` registers, access to those registers is blocked when the VS field is initialized to zero, either at machine level or by the OS itself initializing `sstatus`.

Obviously, one way to prevent the use of new user-level CSRs as covert channels would be to add to `mstatus` or `sstatus` an "XS" field for each relevant extension, paralleling the V extension's VS field. However, this is not considered a general solution to the problem due to the number of potential future extensions that may add small amounts of state. Even with a 64-bit `sstatus` (necessitating adding `sstatush` for RV32), it is not certain there are enough remaining bits in `sstatus` to accommodate all future user-level extensions. In any event, there is no need to strain `sstatus` (and add `sstatush`) for this purpose. The "enable" flags that are needed to plug covert channels are not generally expected to require swapping on context switches of user threads, making them a less-than-compelling candidate for inclusion in `sstatus`. Hence, a new place is provided for them instead.

4.1. State Enable Extensions

The Smstateen and Ssstateen extensions collectively specify machine-mode and supervisor-mode features. The Smstateen extension specification comprises the mstateen*, sstateen*, and hstateen* CSRs and their functionality. The Ssstateen extension specification comprises only the sstateen* and hstateen* CSRs and their functionality.

For RV64 harts, this extension adds four new 64-bit CSRs at machine level: `mstateen0` (Machine State Enable 0), `mstateen1`, `mstateen2`, and `mstateen3`.

If supervisor mode is implemented, another four CSRs are defined at supervisor level: `sstateen0`, `sstateen1`, `sstateen2`, and `sstateen3`.

And if the hypervisor extension is implemented, another set of CSRs is added: `hstateen0`, `hstateen1`, `hstateen2`, and `hstateen3`.

For RV32, the registers listed above are 32-bit, and for the machine-level and hypervisor CSRs there is a corresponding set of high-half CSRs for the upper 32 bits of each register: `mstateen0h`, `mstateen1h`, `mstateen2h`, `mstateen3h`, `hstateen0h`, `hstateen1h`, `hstateen2h`, and `hstateen3h`.

For the supervisor-level `sstateen` registers, high-half CSRs are not added at this time because it is expected

the upper 32 bits of these registers will always be zeros, as explained later below.

Each bit of a `stateen` CSR controls less-privileged access to an extension's state, for an extension that was not deemed "worthy" of a full XS field in `sstatus` like the FS and VS fields for the F and V extensions. The number of registers provided at each level is four because it is believed that 4 * 64 = 256 bits for machine and hypervisor levels, and 4 * 32 = 128 bits for supervisor level, will be adequate for many years to come, perhaps for as long as the RISC-V ISA is in use. The exact number four is an attempted compromise between providing too few bits on the one hand and going overboard with CSRs that will never be used on the other. A possible future doubling of the number of `stateen` CSRs is covered later.

The `stateen` registers at each level control access to state at all less-privileged levels, but not at its own level. This is analogous to how the existing `counteren` CSRs control access to performance counter registers. Just as with the `counteren` CSRs, when a `stateen` CSR prevents access to state by less-privileged levels, an attempt in one of those privilege modes to execute an instruction that would read or write the protected state raises an illegal instruction exception, or, if executing in VS or VU mode and the circumstances for a virtual instruction exception apply, raises a virtual instruction exception instead of an illegal instruction exception.

When this extension is not implemented, all state added by an extension is accessible as defined by that extension.

When a `stateen` CSR prevents access to state for a privilege mode, attempting to execute in that privilege mode an instruction that *implicitly* updates the state without reading it may or may not raise an illegal instruction or virtual instruction exception. Such cases must be disambiguated by being explicitly specified one way or the other.

In some cases, the bits of the `stateen` CSRs will have a dual purpose as enables for the ISA extensions that introduce the controlled state.

Each bit of a supervisor-level `sstateen` CSR controls user-level access (from U-mode or VU-mode) to an extension's state. The intention is to allocate the bits of `sstateen` CSRs starting at the least-significant end, bit 0, through to bit 31, and then on to the next-higher-numbered `sstateen` CSR.

For every bit with a defined purpose in an `sstateen` CSR, the same bit is defined in the matching `mstateen` CSR to control access below machine level to the same state. The upper 32 bits of an `mstateen` CSR (or for RV32, the corresponding high-half CSR) control access to state that is inherently inaccessible to user level, so no corresponding enable bits in the supervisor-level `sstateen` CSR are applicable. The intention is to allocate bits for this purpose starting at the most-significant end, bit 63, through to bit 32, and then on to the next-higher `mstateen` CSR. If the rate that bits are being allocated from the least-significant end for `sstateen` CSRs is sufficiently low, allocation from the most-significant end of `mstateen` CSRs may be allowed to encroach on the lower 32 bits before jumping to the next-higher `mstateen` CSR. In that case, the bit positions of "encroaching" bits will remain forever read-only zeros in the matching `sstateen` CSRs.

With the hypervisor extension, the `hstateen` CSRs have identical encodings to the `mstateen` CSRs, except controlling accesses for a virtual machine (from VS and VU modes).

Each standard-defined bit of a `stateen` CSR is WARL and may be read-only zero or one, subject to the following conditions.

Bits in any `stateen` CSR that are defined to control state that a hart doesn't implement are read-only zeros for that hart. Likewise, all reserved bits not yet given a defined meaning are also read-only zeros. For every bit in an `mstateen` CSR that is zero (whether read-only zero or set to zero), the same bit appears as read-only zero in the matching `hstateen` and `sstateen` CSRs. For every bit in an `hstateen` CSR that is zero (whether read-only zero or set to zero), the same bit appears as read-only zero in `sstateen` when accessed in VS-mode.

A bit in a supervisor-level `sstateen` CSR cannot be read-only one unless the same bit is read-only one in the matching `mstateen` CSR and, if it exists, in the matching `hstateen` CSR. A bit in an `hstateen` CSR cannot be read-only one unless the same bit is read-only one in the matching `mstateen` CSR.

On reset, all writable `mstateen` bits are initialized by the hardware to zeros. If machine-level software changes these values, it is responsible for initializing the corresponding writable bits of the `hstateen` and `sstateen` CSRs to zeros too. Software at each privilege level should set its respective `stateen` CSRs to indicate the state it is prepared to allow less-privileged software to access. For OSes and hypervisors, this usually means the state that the OS or hypervisor is prepared to swap on a context switch, or to manage in some other way.

For each `mstateen` CSR, bit 63 is defined to control access to the matching `sstateen` and `hstateen` CSRs. That is, bit 63 of `mstateen0` controls access to `sstateen0` and `hstateen0`; bit 63 of `mstateen1` controls access to `sstateen1` and `hstateen1`; etc. Likewise, bit 63 of each `hstateen` correspondingly controls access to the matching `sstateen` CSR.

A hypervisor may need this control over accesses to the `sstateen` CSRs if it ever must emulate for a virtual machine an extension that is supposed to be affected by a bit in an `sstateen` CSR. Even if such emulation is uncommon, it should not be excluded.

Machine-level software needs identical control to be able to emulate the hypervisor extension. That is, machine level needs control over accesses to the supervisor-level `sstateen` CSRs in order to emulate the `hstateen` CSRs, which have such control.

Bit 63 of each `mstateen` CSR may be read-only zero only if the hypervisor extension is not implemented and the matching supervisor-level `sstateen` CSR is all read-only zeros. In that case, machine-level software should emulate attempts to access the affected `sstateen` CSR from S-mode, ignoring writes and returning zero for reads. Bit 63 of each `hstateen` CSR is always writable (not read-only).

4.2. State Enable 0 Registers

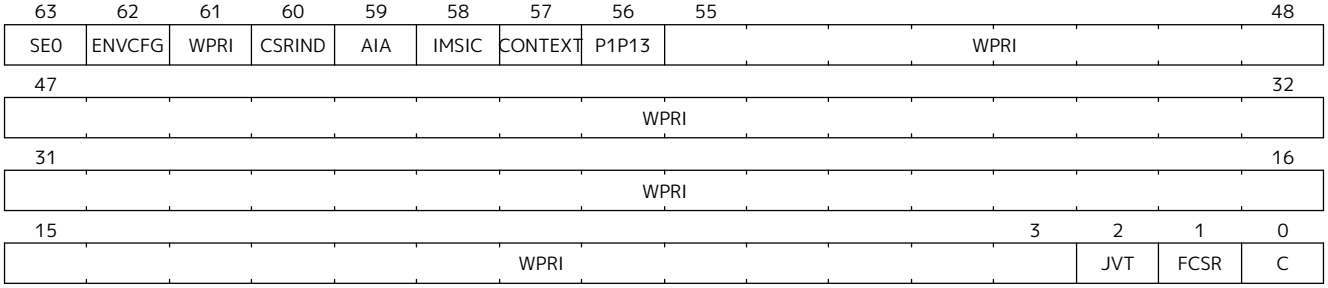


Figure 35. Machine State Enable 0 Register (`mstateen0`)

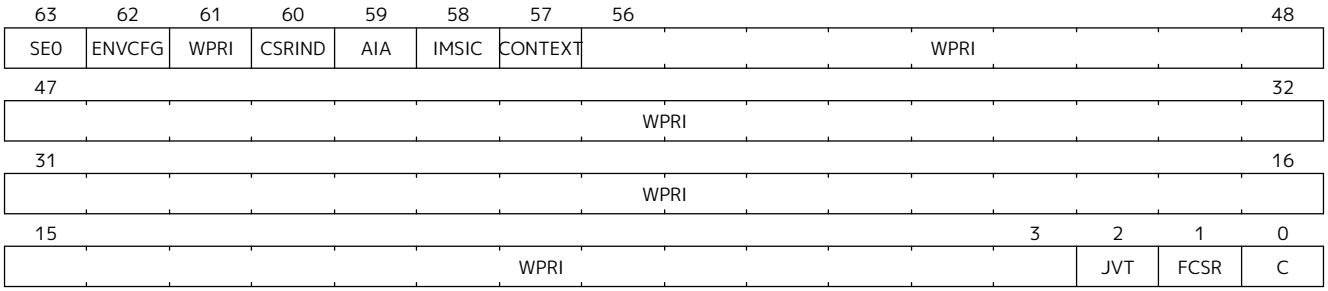


Figure 36. Hypervisor State Enable 0 Register (`hstateen0`)

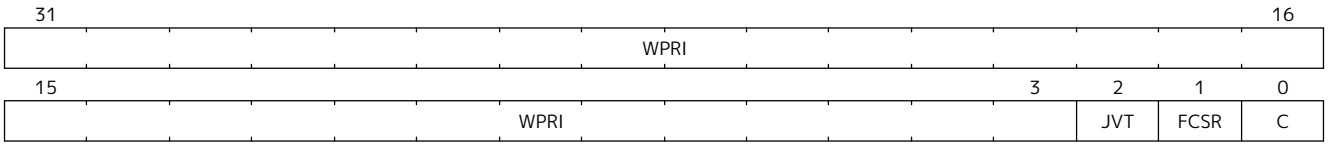


Figure 37. Supervisor State Enable 0 Register (`sstateen0`)

The C bit controls access to any and all custom state. This bit is not custom state itself. The C bit of these registers is not custom state itself; it is a standard field of a standard CSR, either `mstateen0`, `hstateen0`, or `sstateen0`.

The requirements that non-standard extensions must meet to be conforming are not relaxed due solely to changes in the value of this bit. In particular, if software sets this bit but does not execute any custom instructions or access any custom state, the software must continue to execute as specified by all relevant RISC-V standards, or the hardware is not standard-conforming.

The FCSR bit controls access to `fcsr` for the case when floating-point instructions operate on `x` registers instead of `f` registers as specified by the Zfinx and related extensions (Zdinx, etc.). Whenever `misa.F` = 1, FCSR bit of `mstateen0` is read-only zero (and hence read-only zero in `hstateen0` and `sstateen0` too). For convenience, when the `stateen` CSRs are implemented and `misa.F` = 0, then if the FCSR bit of a controlling `stateen0` CSR is zero, all floating-point instructions cause an illegal instruction trap (or virtual instruction trap, if relevant), as though they all access `fcsr`, regardless of whether they really do.

The JVT bit controls access to the `jvt` CSR provided by the Zcmt extension.

The SEO bit in `mstateen0` controls access to the `hstateen0`, `hstateen0h`, and the `sstateen0` CSRs. The SEO bit in `hstateen0` controls access to the `sstateen0` CSR.

The ENVCFG bit in `mstateen0` controls access to the `henvcfg`, `henvcfgh`, and the `senvcfg` CSRs. The ENVCFG bit in `hstateen0` controls access to the `senvcfg` CSRs.

The CSRIND bit in `mstateen0` controls access to the `siselect`, `sireg*`, `vsiselect`, and the `vsireg*` CSRs provided by the Sscsrind extensions. The CSRIND bit in `hstateen0` controls access to the `siselect` and the `sireg*`, (really `vsiselect` and `vsireg*`) CSRs provided by the Sscsrind extensions.

The IMSIC bit in `mstateen0` controls access to the IMSIC state, including CSRs `stopei` and `vstopei`, provided by the Ssaia extension. The IMSIC bit in `hstateen0` controls access to the guest IMSIC state, including CSRs `stopei` (really `vstopei`), provided by the Ssaia extension.

Setting the IMSIC bit in `hstateen0` to zero prevents a virtual machine from accessing the hart's IMSIC the same as setting `hstatus`.VGEIN = 0.

The AIA bit in `mstateen0` controls access to all state introduced by the Ssaia extension and not controlled by either the CSRIND or the IMSIC bits. The AIA bit in `hstateen0` controls access to all state introduced by the Ssaia extension and not controlled by either the CSRIND or the IMSIC bits of `hstateen0`.

The CONTEXT bit in `mstateen0` controls access to the `scontext` and `hcontext` CSRs provided by the Sdtrig extension. The CONTEXT bit in `hstateen0` controls access to the `scontext` CSR provided by the Sdtrig extension.

The P1P13 bit in `mstateen0` controls access to the `hedelegh` introduced by Privileged Specification Version 1.13.

4.3. Usage

After the writable bits of the machine-level `mstateen` CSRs are initialized to zeros on reset, machine-level software can set bits in these registers to enable less-privileged access to the controlled state. This may be either because machine-level software knows how to swap the state or, more likely, because machine-level software isn't swapping supervisor-level environments. (Recall that the main reason the `mstateen` CSRs must exist is so machine level can emulate the hypervisor extension. When machine level isn't emulating the hypervisor extension, it is likely there will be no need to keep any implemented `mstateen` bits zero.)

If machine level sets any writable `mstateen` bits to nonzero, it must initialize the matching `hstateen` CSRs, if they exist, by writing zeros to them. And if any `mstateen` bits that are set to one have matching bits in the `sstateen` CSRs, machine-level software must also initialize those `sstateen` CSRs by writing zeros to them. Ordinarily, machine-level software will want to set bit 63 of all `mstateen` CSRs, necessitating that it write zero to all `hstateen` CSRs.

Software should ensure that all writable bits of `sstateen` CSRs are initialized to zeros when an OS at supervisor level is first entered. The OS can then set bits in these registers to enable user-level access to the controlled state, presumably because it knows how to context-swap the state.

For the `sstateen` CSRs whose access by a guest OS is permitted by bit 63 of the corresponding `hstateen` CSRs, a hypervisor must include the `sstateen` CSRs in the context it swaps for a guest OS. When it starts a new guest OS, it must ensure the writable bits of those `sstateen` CSRs are initialized to zeros, and it must emulate accesses to any other `sstateen` CSRs.

If software at any privilege level does not support multiple contexts for less-privilege levels, then it may choose to maximize less-privileged access to all state by writing a value of all ones to the `stateen` CSRs at its level (the `mstateen` CSRs for machine level, the `sstateen` CSRs for an OS, and the `hstateen` CSRs for a hypervisor), without knowing all the state to which it is granting access. This is justified because there is no risk of a covert channel between execution contexts at the less-privileged level when only one context exists at that level. This situation is expected to be common for machine level, and it might also arise, for example, for a type-1 hypervisor that hosts only a single guest virtual machine.

If a need is anticipated, the set of `stateen` *CSRs could in the future be doubled by adding these:*

- `0x38C mstateen4`, `0x39C mstateen4h`
- `0x38D mstateen5`, `0x39D mstateen5h`
- `0x38E mstateen6`, `0x39E mstateen6h`
- `0x38F mstateen7`, `0x39F mstateen7h`
- `0x18C sstateen4`
- `0x18D sstateen5`
- `0x18E sstateen6`
- `0x18F sstateen7`
- `0x68C hstateen4`, `0x69C hstateen4h`
- `0x68D hstateen5`, `0x69D hstateen5h`
- `0x68E hstateen6`, `0x69E hstateen6h`
- `0x68F hstateen7`, `0x69F hstateen7h`

These additional CSRs are not a definite part of the original proposal because it is unclear whether they will ever be needed, and it is believed the rate of consumption of bits in the first group, registers numbered 0-3, will be slow enough that any looming shortage will be

> *perceptible many years in advance. At the moment, it is not known even how many years it may take to exhaust just* `mstateen0`, `sstateen0`, *and* `hstateen0`.

Chapter 5. "Smcsrind/Sscsrind" Indirect CSR Access, Version 1.0

5.1. Introduction

Smcsrind/Sscsrind is an ISA extension that extends the indirect CSR access mechanism originally defined as part of the Smaia/Ssaia extensions, in order to make it available for use by other extensions without creating an unnecessary dependence on Smaia/Ssaia.

This extension confers two benefits:

1. It provides a means to access an array of registers via CSRs without requiring allocation of large chunks of the limited CSR address space.
2. It enables software to access each of an array of registers by index, without requiring a switch statement with a case for each register.

CSRs are accessed indirectly via this extension using select values, in contrast to being accessed directly using standard CSR numbers. A CSR accessible via one method may or may not be accessible via the other method. Select values are a separate address space from CSR numbers, and from tselect values in the Sdtrig extension. If a CSR is both directly and indirectly accessible, the CSR's select value is unrelated to its CSR number.

Further, Machine-level and Supervisor-level select values are separate address spaces from each other; however, Machine-level and Supervisor-level CSRs with the same select value may be defined by an extension as partial or full aliases with respect to each other. This typically would be done for CSRs that can be delegated from Machine-level to Supervisor-level.

The machine-level extension **Smcsrind** encompasses all added CSRs and all behavior modifications for a hart, over all privilege levels. For a supervisor-level environment, extension **Sscsrind** is essentially the same as Smcsrind except excluding the machine-level CSRs and behavior not directly visible to supervisor level.

5.2. Machine-level CSRs

Number	Privilege	Width	Name	Description
0x350	MRW	XLEN	`miselect`	Machine indirect register select
0x351	MRW	XLEN	`mireg`	Machine indirect register alias
0x352	MRW	XLEN	`mireg2`	Machine indirect register alias 2
0x353	MRW	XLEN	`mireg3`	Machine indirect register alias 3
0x355	MRW	XLEN	`mireg4`	Machine indirect register alias 4
0x356	MRW	XLEN	`mireg5`	Machine indirect register alias 5
0x357	MRW	XLEN	`mireg6`	Machine indirect register alias 6

The `mireg*` CSR numbers are not consecutive because miph is CSR number 0x354.

The CSRs listed in the table above provide a window for accessing register state indirectly. The value of `miselect` determines which register is accessed upon read or write of each of the machine indirect alias CSRs (`mireg*`). `miselect` value ranges are allocated to dependent extensions, which specify the register state accessible via each `miregi` register, for each `miselect` value. `miselect` is a WARL register.

The `miselect` register implements at least enough bits to support all implemented `miselect` values (corresponding to the implemented extensions that utilize `miselect`/`mireg*` to indirectly access register state). The `miselect` register may be read-only zero if there are no extensions implemented that utilize it.

Values of `miselect` with the most-significant bit set (bit XLEN - 1 = 1) are designated only for custom use, presumably for accessing custom registers through the alias CSRs. Values of `miselect` with the most-significant bit clear are designated only for standard use and are reserved until allocated to a standard architecture extension. If XLEN is changed, the most-significant bit of `miselect` moves to the new position, retaining its value from before.

An implementation is not required to support any custom values for `miselect`.

The behavior upon accessing `mireg*` from M-mode, while `miselect` holds a value that is not implemented, is UNSPECIFIED.

It is expected that implementations will typically raise an illegal instruction exception for such accesses, so that, for example, they can be identified as software bugs. Platform specs, profile specs, and/or the Privileged ISA spec may place more restrictions on behavior for such accesses.

Attempts to access `mireg*` while `miselect` holds a number in an allocated and implemented range results in a specific behavior that, for each combination of `miselect` and `miregi`, is defined by the extension to which the `miselect` value is allocated.

Ordinarily, each `miregi` *will access register state, access read-only 0 state, or raise an illegal instruction exception.*

For RV32, if an extension defines an indirectly accessed register as 64 bits wide, it is recommended that the lower 32 bits of the register are accessed through one of `mireg`, `mireg2`, *or* `mireg3`, *while the upper 32 bits are accessed through* `mireg4`, `mireg5`, *or* `mireg6`, *respectively.*

Six `*ireg*` *registers are defined in order to ensure that the needs of extensions in development are covered, with some room for growth. For example, for an* `siselect` *value associated with counter X,* `sireg`/`sireg2` *could be used to access* `mhpmcounterX`/`mhpmeventX`, *while* `sireg4`/`sireg5` *could access* `mhpmcounterXh`/`mhpmeventXh`. *Six* `*ireg*` *registers allows for accessing up to 3 CSR arrays per index (*`*iselect`*) with RV32-only CSRs, or up to 6 CSR arrays per index value without RV32-only CSRs.*

5.3. Supervisor-level CSRs

Number	Privilege	Width	Name	Description
0x150	SRW	XLEN	`siselect`	Supervisor indirect register select
0x151	SRW	XLEN	`sireg`	Supervisor indirect register alias
0x152	SRW	XLEN	`sireg2`	Supervisor indirect register alias 2
0x153	SRW	XLEN	`sireg3`	Supervisor indirect register alias 3
0x155	SRW	XLEN	`sireg4`	Supervisor indirect register alias 4
0x156	SRW	XLEN	`sireg5`	Supervisor indirect register alias 5
0x157	SRW	XLEN	`sireg6`	Supervisor indirect register alias 6

The CSRs in the table above are required if S-mode is implemented.

The `siselect` register will support the value range 0..0xFFF at a minimum. A future extension may define a

value range outside of this minimum range. Only if such an extension is implemented will `siselect` be required to support larger values.

Requiring a range of 0–0xFFF for `siselect`*, even though most or all of the space may be reserved or inaccessible, permits M-mode to emulate indirectly accessed registers in this implemented range, including registers that may be standardized in the future.*

Values of `siselect` with the most-significant bit set (bit XLEN - 1 = 1) are designated only for custom use, presumably for accessing custom registers through the alias CSRs. Values of `siselect` with the most-significant bit clear are designated only for standard use and are reserved until allocated to a standard architecture extension. If XLEN is changed, the most-significant bit of `siselect` moves to the new position, retaining its value from before.

The behavior upon accessing `sireg*` from M-mode or S-mode, while `siselect` holds a value that is not implemented at supervisor level, is UNSPECIFIED.

It is recommended that implementations raise an illegal instruction exception for such accesses, to facilitate possible emulation (by M-mode) of these accesses.

An extension is considered not to be implemented at supervisor level if machine level has disabled the extension for S-mode, such as by the settings of certain fields in CSR `menvcfg`*, for example.*

Otherwise, attempts to access `sireg*` from M-mode or S-mode while `siselect` holds a number in a standard-defined and implemented range result in specific behavior that, for each combination of `siselect` and `siregi`, is defined by the extension to which the `siselect` value is allocated.

Ordinarily, each `siregi` *will access register state, access read-only 0 state, or, unless executing in a virtual machine (covered in the next section), raise an illegal instruction exception.*

Note that the widths of `siselect` and `sireg*` are always the current XLEN rather than SXLEN. Hence, for example, if MXLEN = 64 and SXLEN = 32, then these registers are 64 bits when the current privilege mode is M (running RV64 code) but 32 bits when the privilege mode is S (RV32 code).

5.4. Virtual Supervisor-level CSRs

Number	Privilege	Width	Name	Description
0x250	HRW	XLEN	`vsiselect`	Virtual supervisor indirect register select
0x251	HRW	XLEN	`vsireg`	Virtual supervisor indirect register alias
0x252	HRW	XLEN	`vsireg2`	Virtual supervisor indirect register alias 2
0x253	HRW	XLEN	`vsireg3`	Virtual supervisor indirect register alias 3
0x255	HRW	XLEN	`vsireg4`	Virtual supervisor indirect register alias 4
0x256	HRW	XLEN	`vsireg5`	Virtual supervisor indirect register alias 5
0x257	HRW	XLEN	`vsireg6`	Virtual supervisor indirect register alias 6

The CSRs in the table above are required if the hypervisor extension is implemented. These VS CSRs all match supervisor CSRs, and substitute for those supervisor CSRs when executing in a virtual machine (in VS-mode or VU-mode).

The `vsiselect` register will support the value range 0..0xFFF at a minimum. A future extension may define a value range outside of this minimum range. Only if such an extension is implemented will `vsiselect` be required to support larger values.

Requiring a range of 0–0xFFF for `vsiselect`*, even though most or all of the space may be reserved or inaccessible, permits a hypervisor to emulate indirectly accessed registers in this implemented range, including registers that may be standardized in the future.*

More generally it is recommended that `vsiselect` *and* `siselect` *be implemented with the same number of bits. This also avoids creation of a virtualization hole due to observable differences between* `vsiselect` *and* `siselect` *widths.*

Values of `vsiselect` with the most-significant bit set (bit XLEN - 1 = 1) are designated only for custom use, presumably for accessing custom registers through the alias CSRs. Values of `vsiselect` with the most-significant bit clear are designated only for standard use and are reserved until allocated to a standard architecture extension. If XLEN is changed, the most-significant bit of `vsiselect` moves to the new position, retaining its value from before.

For alias CSRs `sireg*` and `vsireg*`, the hypervisor extension's usual rules for when to raise a virtual instruction exception (based on whether an instruction is HS-qualified) are not applicable. The rules given in this section for `sireg` and `vsireg` apply instead, unless overridden by the requirements specified in the section below, which take precedence over this section when extension Smstateen is also implemented.

A virtual instruction exception is raised for attempts from VS-mode or VU-mode to directly access `vsiselect` or `vsireg*`, or attempts from VU-mode to access `siselect` or `sireg*`.

The behavior upon accessing `vsireg*` from M-mode or HS-mode, or accessing `sireg*` (really `vsireg*`) from VS-mode, while `vsiselect` holds a value that is not implemented at HS level, is UNSPECIFIED.

It is recommended that implementations raise an illegal instruction exception for such accesses, to facilitate possible emulation (by M-mode) of these accesses.

Otherwise, while `vsiselect` holds a number in a standard-defined and implemented range, attempts to access `vsireg*` from a sufficiently privileged mode, or to access `sireg*` (really `vsireg*`) from VS-mode, result in specific behavior that, for each combination of `vsiselect` and `vsiregi`, is defined by the extension to which the `vsiselect` value is allocated.

Ordinarily, each `vsiregi` *will access register state, access read-only 0 state, or raise an exception (either an illegal instruction exception or, for select accesses from VS-mode, a virtual instruction exception). When* `vsiselect` *holds a value that is implemented at HS level but not at VS level, attempts to access* `sireg*` *(really* `vsireg*`*) from VS-mode will typically raise a virtual instruction exception. But there may be cases specific to an extension where different behavior is more appropriate.*

Like `siselect` and `sireg*`, the widths of `vsiselect` and `vsireg*` are always the current XLEN rather than VSXLEN. Hence, for example, if HSXLEN = 64 and VSXLEN = 32, then these registers are 64 bits when accessed by a hypervisor in HS-mode (running RV64 code) but 32 bits for a guest OS in VS-mode (RV32 code).

5.5. Access control by the state-enable CSRs

If extension Smstateen is implemented together with Smcsrind, bit 60 of state-enable register `mstateen0` controls access to `siselect`, `sireg*`, `vsiselect`, and `vsireg*`. When `mstateen0`[60]=0, an attempt to access one of these CSRs from a privilege mode less privileged than M-mode results in an illegal instruction exception. As always, the state-enable CSRs do not affect the accessibility of any state when in M-mode, only in less privileged modes. For more explanation, see the documentation for extension Smstateen.

Other extensions may specify that certain mstateen bits control access to registers accessed indirectly

through `siselect` + `sireg*`, and/or `vsiselect` + `vsireg*`. However, regardless of any other mstateen bits, if `mstateen0`[60] = 1, a virtual instruction exception is raised as described in the previous section for all attempts from VS-mode or VU-mode to directly access `vsiselect` or `vsireg*`, and for all attempts from VU-mode to access `siselect` or `sireg*`.

If the hypervisor extension is implemented, the same bit is defined also in hypervisor CSR `hstateen0`, but controls access to only `siselect` and `sireg*` (really `vsiselect` and `vsireg*`), which is the state potentially accessible to a virtual machine executing in VS or VU-mode. When `hstateen0`[60]=0 and `mstateen0`[60]=1, all attempts from VS or VU-mode to access `siselect` or `sireg*` raise a virtual instruction exception, not an illegal instruction exception, regardless of the value of `vsiselect` or any other mstateen bit.

Extension Ssstateen is defined as the supervisor-level view of Smstateen. Therefore, the combination of Sscsrind and Ssstateen incorporates the bit defined above for `hstateen0` but not that for `mstateen0`, since machine-level CSRs are not visible to supervisor level.

CSR address space is reserved for a possible future "Sucsrind" extension that extends indirect CSR access to user mode.

Chapter 6. "Smepmp" Extension for PMP Enhancements for memory access and execution prevention in Machine mode, Version 1.0

6.1. Introduction

Being able to access the memory of a process running at a high privileged execution mode, such as the Supervisor or Machine mode, from a lower privileged mode such as the User mode, introduces an obvious attack vector since it allows for an attacker to perform privilege escalation, and tamper with the code and/or data of that process. A less obvious attack vector exists when the reverse happens, in which case an attacker instead of tampering with code and/or data that belong to a high-privileged process, can tamper with the memory of an unprivileged / less-privileged process and trick the high-privileged process to use or execute it.

To prevent this attack vector, two mechanisms known as Supervisor Memory Access Prevention (SMAP) and Supervisor Memory Execution Prevention (SMEP) were introduced in recent systems. The first one prevents the OS from accessing the memory of an unprivileged process unless a specific code path is followed, and the second one prevents the OS from executing the memory of an unprivileged process at all times. RISC-V already includes support for SMAP, through the `sstatus.SUM` bit, and for SMEP by always denying execution of virtual memory pages marked with the U bit, with Supervisor mode (OS) privileges, as mandated on the Privilege Spec.

Terms:

- **PMP Entry**: *A pair of* `pmpcfg[i]` / `pmpaddr[i]` *registers.*
- **PMP Rule**: *The contents of a pmpcfg register and its associated pmpaddr register(s), that encode a valid protected physical memory region, where* `pmpcfg[i].A != OFF`*, and if* `pmpcfg[i].A == TOR`, `pmpaddr[i-1] < pmpaddr[i]`.
- **Ignored**: *Any permissions set by a matching PMP rule are ignored, and all accesses to the requested address range are allowed.*
- **Enforced**: *Only access types configured in the PMP rule matching the requested address range are allowed; failures will cause an access-fault exception.*
- **Denied**: *Any permissions set by a matching PMP rule are ignored, and no accesses to the requested address range are allowed.; failures will cause an access-fault exception.*
- **Locked**: *A PMP rule/entry where the* `pmpcfg.L` *bit is set.*
- **PMP reset**: *A reset process where all PMP settings of the hart, including locked rules/settings, are re-initialized to a set of safe defaults, before releasing the hart (back) to the firmware / OS / application.*

6.1.1. Threat model

However, there are no such mechanisms available on Machine mode in the current (v1.11) Privileged Spec. It is not possible for a PMP rule to be **enforced** only on non-Machine modes and **denied** on Machine mode, to only allow access to a memory region by less-privileged modes. it is only possible to have a **locked** rule that will be **enforced** on all modes, or a rule that will be **enforced** on non-Machine modes and be **ignored** by Machine mode. So for any physical memory region which is not protected with a Locked rule, Machine mode has unlimited access, including the ability to execute it.

Without being able to protect less-privileged modes from Machine mode, it is not possible to prevent the

mentioned attack vector. This becomes even more important for RISC-V than on other architectures, since implementations are allowed where a hart only has Machine and User modes available, so the whole OS will run on Machine mode instead of the non-existent Supervisor mode. In such implementations the attack surface is greatly increased, and the same kind of attacks performed on Supervisor mode and mitigated through SMAP/SMEP, can be performed on Machine mode without any available mitigations. Even on implementations with Supervisor mode present attacks are still possible against the Firmware and/or the Secure Monitor running on Machine mode.

6.2. Proposal

1. **Machine Security Configuration (mseccfg)** is a new RW Machine mode CSR, used for configuring various security mechanisms present on the hart, and only accessible to Machine mode. It is 64 bits wide, and is at address **0x747 on RV64** and **0x747 (low 32bits), 0x757 (high 32bits) on RV32**. All mseccfg fields defined on this proposal are WARL, and the remaining bits are reserved for future standard use and should always read zero. The reset value of mseccfg is implementation-specific, otherwise if backwards compatibility is a requirement it should reset to zero on hard reset.
2. On `mseccfg` we introduce a field on bit 2 called **Rule Locking Bypass (mseccfg.RLB)** with the following functionality:
   a. When `mseccfg.RLB` is 1 **locked** PMP rules may be removed/modified and **locked** PMP entries may be edited.
   b. When `mseccfg.RLB` is 0 and `pmpcfg.L` is 1 in any rule or entry (including disabled entries), then `mseccfg.RLB` remains 0 and any further modifications to `mseccfg.RLB` are ignored until a **PMP reset**.

Note that this feature is intended to be used as a debug mechanism, or as a temporary workaround during the boot process for simplifying software, and optimizing the allocation of memory and PMP rules. Using this functionality under normal operation, after the boot process is completed, should be avoided since it weakens the protection of M-mode-only rules. Vendors who don't need this functionality may hardwire this field to 0.

3. On `mseccfg` we introduce a field in bit 1 called **Machine-Mode alloWlist Policy (mseccfg.MMWP)**. This is a sticky bit, meaning that once set it cannot be unset until a **PMP reset**. When set it changes the default PMP policy for M-mode when accessing memory regions that don't have a matching PMP rule, to **denied** instead of **ignored**.
4. On `mseccfg` we introduce a field in bit 0 called **Machine Mode Lockdown (mseccfg.MML)**. This is a sticky bit, meaning that once set it cannot be unset until a **PMP reset**. When `mseccfg.MML` is set the system's behavior changes in the following way:
   a. The meaning of `pmpcfg.L` changes: Instead of marking a rule as **locked** and **enforced** in all modes, it now marks a rule as **M-mode-only** when set and **S/U-mode-only** when unset. The formerly reserved encoding of `pmpcfg.RW=01`, and the encoding `pmpcfg.LRWX=1111`, now encode a **Shared-Region**.

   An *M-mode-only* rule is **enforced** on Machine mode and **denied** in Supervisor or User mode. It also remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.

   An *S/U-mode-only* rule is **enforced** on Supervisor and User modes and **denied** on Machine mode.

   A *Shared-Region* rule is **enforced** on all modes, with restrictions depending on the `pmpcfg.L` and `pmpcfg.X` bits:

   - A *Shared-Region* rule where `pmpcfg.L` is not set can be used for sharing data between M-mode

and S/U-mode, so is not executable. M-mode has read/write access to that region, and S/U-mode has read access if `pmpcfg.X` is not set, or read/write access if `pmpcfg.X` is set.

- A *Shared-Region* rule where `pmpcfg.L` is set can be used for sharing code between M-mode and S/U-mode, so is not writeable. Both M-mode and S/U-mode have execute access on the region, and M-mode also has read access if `pmpcfg.X` is set. The rule remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.
- The encoding `pmpcfg.LRWX=1111` can be used for sharing data between M-mode and S/U mode, where both modes only have read-only access to the region. The rule remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.

b. Adding a rule with executable privileges that either is **M-mode-only** or a **locked Shared-Region** is not possible and such `pmpcfg` writes are ignored, leaving `pmpcfg` unchanged. This restriction can be temporarily lifted by setting `mseccfg.RLB` e.g. during the boot process.

c. Executing code with Machine mode privileges is only possible from memory regions with a matching **M-mode-only** rule or a **locked Shared-Region** rule with executable privileges. Executing code from a region without a matching rule or with a matching *S/U-mode-only* rule is **denied**.

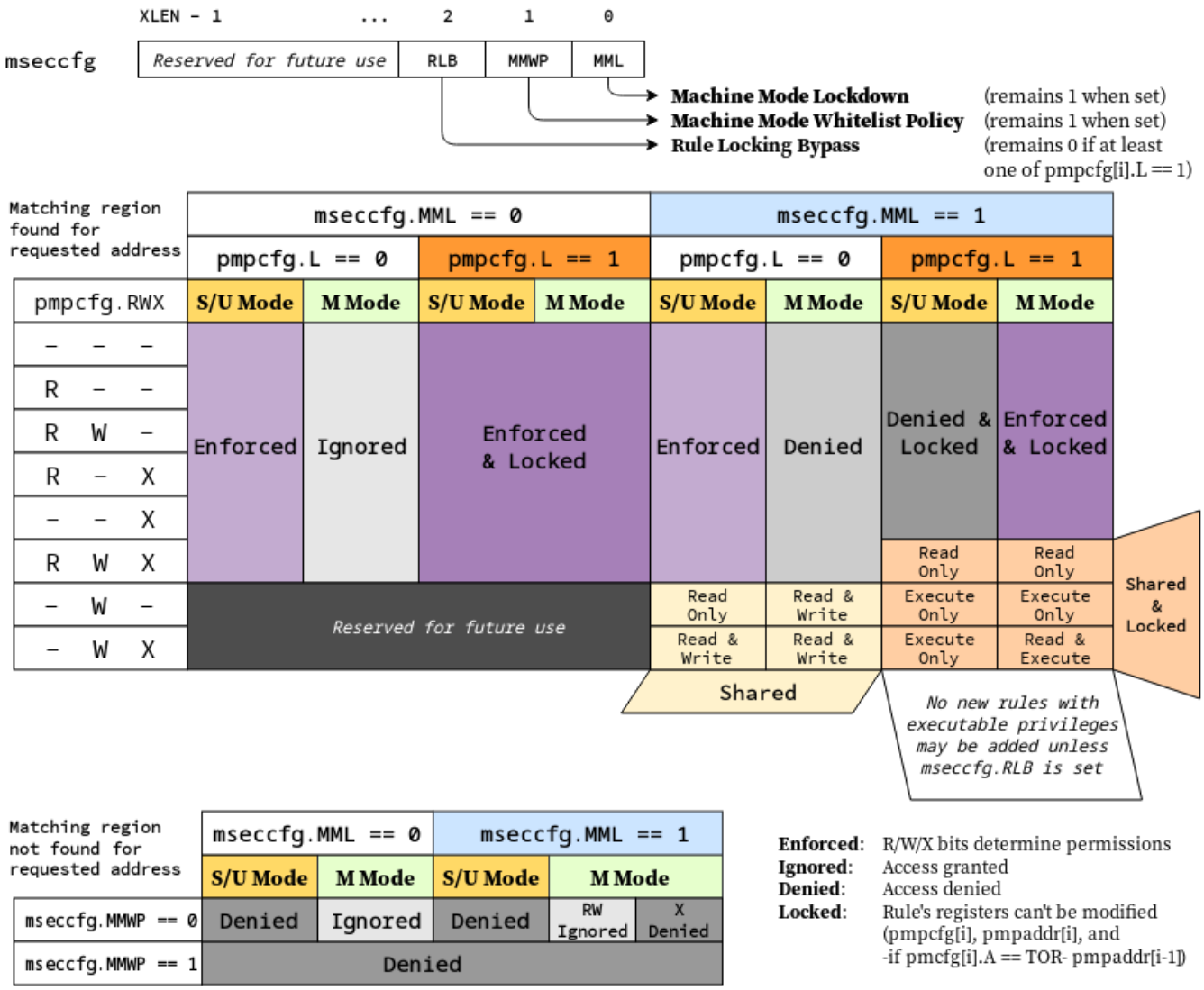
d. If `mseccfg.MML` is not set, the combination of `pmpcfg.RW=01` remains reserved for future standard use.

6.2.1. Truth table when mseccfg.MML is set

Bits on *pmpcfg* register				Result	
L	R	W	X	M Mode	S/U Mode
0	0	0	0	Inaccessible region (Access Exception)	
0	0	0	1	Access Exception	Execute-only region
0	0	1	0	Shared data region: Read/write on M mode, read-only on S/U mode	
0	0	1	1	Shared data region: Read/write for both M and S/U mode	
0	1	0	0	Access Exception	Read-only region
0	1	0	1	Access Exception	Read/Execute region
0	1	1	0	Access Exception	Read/Write region
0	1	1	1	Access Exception	Read/Write/Execute region
1	0	0	0	Locked inaccessible region* (Access Exception)	
1	0	0	1	Locked Execute-only region*	Access Exception
1	0	1	0	Locked Shared code region: Execute only on both M and S/U mode.*	
1	0	1	1	Locked Shared code region: Execute only on S/U mode, read/execute on M mode.*	
1	1	0	0	Locked Read-only region*	Access Exception
1	1	0	1	Locked Read/Execute region*	Access Exception
1	1	1	0	Locked Read/Write region*	Access Exception
1	1	1	1	Locked Shared data region: Read only on both M and S/U mode.*	

: ***Locked** rules cannot be removed or modified until a **PMP reset**, unless `mseccfg.RLB` is set.

6.2.2. Visual representation of the proposal



6.3. Smepmp software discovery

Since all fields defined on `mseccfg` as part of this proposal are locked when set (`MMWP`/`MML`) or locked when cleared (`RLB`), software can't poll them for determining the presence of Smepmp. It is expected that BootROM will set `mseccfg.MMWP` and/or `mseccfg.MML` during early boot, before jumping to the firmware, so that the firmware will be able to determine the presence of Smepmp by reading `mseccfg` and checking the state of `mseccfg.MMWP` and `mseccfg.MML`.

6.4. Rationale

1. Since a CSR for security and / or global PMP behavior settings is not available with the current spec, we needed to define a new one. This new CSR will allow us to add further security configuration options in the future and also allow developers to verify the existence of the new mechanisms defined on this proposal.
2. There are use cases where developers want to enforce PMP rules in M-mode during the boot process, that are also able to modify, merge, and / or remove later on. Since a rule that is enforced in M-mode also needs to be locked (or else badly written or malicious M-mode software can remove it at any time), the only way for developers to approach this is to keep adding PMP rules to the chain and rely on rule priority. This is a waste of PMP rules and since it's only needed during boot, `mseccfg.RLB` is a simple workaround that can be used temporarily and then disabled and locked down.

Also when `mseccfg.MML` is set, according to 4b it's not possible to add a *Shared-Region* rule with executable privileges. So RLB can be set temporarily during the boot process to register such regions. Note that it's still possible to register executable *Shared-Region* rules using initial register settings (that may include `mseccfg.MML` being set and the rule being set on PMP registers) on **PMP reset**, without using RLB.

Be aware that RLB introduces a security vulnerability if left set after the boot process is over and in general it should be used with caution, even when used temporarily. *Having editable PMP rules in M-mode gives a false sense of security since it only takes a few malicious instructions to lift any PMP restrictions this way. It doesn't make sense to have a security control in place and leave it unprotected. Rule Locking Bypass is only meant as a way to optimize the allocation of PMP rules, catch errors durring debugging, and allow the bootrom/firmware to register executable Shared-Region rules. If developers / vendors have no use for such functionality, they should never set* `mseccfg.RLB` *and if possible hard-wire it to 0. In any case* **RLB should be disabled and locked as soon as possible**.

If `mseccfg.RLB` *is not used and left unset, it will be locked as soon as a PMP rule/entry with the* `pmpcfg.L` *bit set is configured.*

Since PMP rules with a higher priority override rules with a lower priority, locked rules must precede non-locked rules.

3. With the current spec M-mode can access any memory region unless restricted by a PMP rule with the `pmpcfg.L` bit set. There are cases where this approach is overly permissive, and although it's possible to restrict M-mode by adding PMP rules during the boot process, this can also be seen as a waste of PMP rules. Having the option to block anything by default, and use PMP as an allowlist for M-mode is considered a safer approach. This functionality may be used during the boot process or upon **PMP reset**, using initial register settings.
4. The current dual meaning of the `pmpcfg.L` bit that marks a rule as Locked and **enforced** on all modes is neither flexible nor clean. With the introduction of *Machine Mode Lock-down* the `pmpcfg.L` bit distinguishes between rules that are **enforced only** in M-mode (*M-mode-only*) or **only** in S/U-modes (*S/U-mode-only*). The rule locking becomes part of the definition of an *M-mode-only* rule, since when a rule is added in M mode, if not locked, can be modified or removed in a few instructions. On the other hand, S/U modes can't modify PMP rules anyway so locking them doesn't make sense.
    a. This separation between *M-mode-only* and *S/U-mode-only* rules also allows us to distinguish which regions are to be used by processes in Machine mode (`pmpcfg.L == 1`) and which by Supervisor or User mode processes (`pmpcfg.L == 0`), in the same way the U bit on the Virtual Memory's PTEs marks which Virtual Memory pages are to be used by User mode applications (U=1) and which by the Supervisor / OS (U=0). With this distinction in place we are able to implement memory access and execution prevention in M-mode for any physical memory region that is not *M-mode-only*.

    An attacker that manages to tamper with a memory region used by S/U mode, even after successfully tricking a process running in M-mode to use or execute that region, will fail to perform a successful attack since that region will be *S/U-mode-only* hence any access when in M-mode will trigger an access exception.

In order to support zero-copy transfers between M-mode and S/U-mode we need to either allow shared memory regions, or introduce a mechanism similar to the `sstatus.SUM` *bit to temporary allow the high-privileged mode (in this case M-mode) to be able to perform loads and stores on the region of a less-privileged process (in this case S/U-mode). In our case after discussion within the group it seemed a better idea to follow the first approach and have this functionality encoded on a per-rule basis to avoid the risk of leaving a temporary, global bypass active when exiting M-mode,*

hence rendering memory access prevention useless.

Although it's possible to use `mstatus.MPRV` *in M-mode to read/write data on an S/U-mode-only region using general purpose registers for copying, this will happen with S/U-mode permissions, honoring any MMU restrictions put in place by S-mode. Of course it's still possible for M-mode to tamper with the page tables and / or add S/U-mode-only rules and bypass the protections put in place by S-mode but if an attacker has managed to compromise M-mode to such extent, no security guarantees are possible in any way.* **Also note that the threat model we present here assumes buggy software in M-mode, not compromised software**. *We considered disabling* `mstatus.MPRV` *but it seemed too much and out of scope.*

Shared-region rules can be used both for zero-copy data transfers and for sharing code segments. The latter may be used for example to allow S/U-mode to execute code by the vendor, that makes use of some vendor-specific ISA extension, without having to go through the firmware with an ecall. This is similar to the vDSO approach followed on Linux, that allows userspace code to execute kernel code without having to perform a system call.

To make sure that shared data regions can't be executed and shared code regions can't be modified, the encoding changes the meaning of the `pmpcfg.X bit`. In case of shared data regions, with the exception of the `pmpcfg.LRWX=1111` encoding, the `pmpcfg.X` bit marks the capability of S/U-mode to write to that region, so it's not possible to encode an executable shared data region. In case of shared code regions, the `pmpcfg.X` bit marks the capability of M-mode to read from that region, and since `pmpcfg.RW=01` is used for encoding the shared region, it's not possible to encode a shared writable code region.

For adding Shared-region rules with executable privileges to share code segments between M-mode and S/U-mode, `mseccfg.RLB` *needs to be implemented, or else such rules can only be added together with* `mseccfg.MML` *being set on* **PMP Reset**. *That's because the reserved encoding* `pmpcfg.RW=01` *being used for Shared-region rules is only defined when* `mseccfg.MML` *is set, and 4b prevents the addition of rules with executable privileges on M-mode after* `mseccfg.MML` *is set unless* `mseccfg.RLB` *is also set.*

Using the `pmpcfg.LRWX=1111` *encoding for a locked shared read-only data region was decided later on, its initial meaning was an M-mode-only read/write/execute region. The reason for that change was that the already defined shared data regions were not locked, so r/w access to M-mode couldn't be restricted. In the same way we have execute-only shared code regions for both modes, it was decided to also be able to allow a least-privileged shared data region for both modes. This approach allows for example to share the .text section of an ELF with a shared code region and the .rodata section with a locked shared data region, without allowing M-mode to modify .rodata. We also decided that having a locked read/write/execute region in M-mode doesn't make much sense and could be dangerous, since M-mode won't be able to add further restrictions there (as in the case of S/U-mode where S-mode can further limit access to an* `pmpcfg.LWRX=0111` *region through the MMU), leaving the possibility of modifying an executable region in M-mode open.*

For encoding Shared-region rules initially we used one of the two reserved bits on pmpcfg (bit 5) but in order to avoid allocating an extra bit, since those bits are a very limited resource, it was decided to use the reserved R=0,W=1 combination.

b. The idea with this restriction is that after the Firmware or the OS running in M-mode is initialized and `mseccfg.MML` is set, no new code regions are expected to be added since nothing else is expected to run in M-mode (everything else will run in S/U mode). Since we want to limit the attack surface

of the system as much as possible, it makes sense to disallow any new code regions which may include malicious code, to be added/executed in M-mode.

c. In case `mseccfg.MMWP` is not set, M-mode can still access and execute any region not covered by a PMP rule. Since we try to prevent M-mode from executing malicious code and since an attacker may manage to place code on some region not covered by PMP (e.g. a directly-addressable flash memory), we need to ensure that M-mode can only execute the code segments initialized during firmware / OS initialization.

d. We are only using the encoding `pmpcfg.RW=01` together with `mseccfg.MML`, if `mseccfg.MML` is not set the encoding remains usable for future use.

Chapter 7. "Smcntrpmf" Cycle and Instret Privilege Mode Filtering, Version 1.0

7.1. Introduction

The cycle and instret counters serve to support user mode self-profiling usages, wherein a user can read the counter(s) twice and compute the delta(s) to evaluate user software performance and behavior. By default, these counters are not filtered by privilege mode, and thus they continue to increment while traps (e.g., page faults or interrupts) to more privileged code are handled. This causes two problems:

- It introduces unpredictable noise to the counter values observed by the user.
- It leaks information about privileged software execution to user mode.

Smcntrpmf remedies these issues by introducing privilege mode filtering for the cycle and instret counters.

7.2. CSRs

7.2.1. Machine Counter Configuration (`mcyclecfg`, `minstretcfg`) Registers

mcyclecfg and minstretcfg are 64-bit registers that configure privilege mode filtering for the cycle and instret counters, respectively.

63	62	61	60	59	58	57:0
0	MINH	SINH	UINH	VSINH	VUINH	*WPRI*

Field	Description
MINH	If set, then counting of events in M-mode is inhibited
SINH	If set, then counting of events in S/HS-mode is inhibited
UINH	If set, then counting of events in U-mode is inhibited
VSINH	If set, then counting of events in VS-mode is inhibited
VUINH	If set, then counting of events in VU-mode is inhibited

When all *x*INH bits are zero, event counting is enabled in all modes.

For each bit in 61:58, if the associated privilege mode is not implemented, the bit is read-only zero.

For RV32, bits 63:32 of mcyclecfg can be accessed via the mcyclecfgh CSR, and bits 63:32 of minstretcfg can be accessed via the minstretcfgh CSR.

The CSR numbers are 0x321 for mcyclecfg, 0x322 for minstretcfg, 0x721 for mcyclecfgh, and 0x722 for minstretcfgh.

The content of these registers may be accessible from Supervisor level if the Smcdeleg/Ssccfg extensions are implemented.

The more natural CSR number for mcyclecfg would be 0x320, but that was allocated to mcountinhibit.

This register format matches that specified for programmable counters by Sscofpmf. The bit

position for the OF bit (bit 63) is read-only 0, since these counters do not generate local counter overflow interrupts on overflow.

7.3. Counter Behavior

The fundamental behavior of cycle and instret is modified in that counting does not occur while executing in an inhibited privilege mode. Further, the following defines how transitions between a non-inhibited privilege mode and an inhibited privilege mode are counted.

The cycle counter will simply count CPU cycles while the CPU is in a non-inhibited privilege mode. Mode transition operations (traps and trap returns) may take multiple clock cycles, and the change of privilege mode may be reported as occurring in any one of those cycles (possibly different for each occurrence of a trap or trap return).

The RISC-V ISA has no requirement that the number of cycles for a trap or trap return be the same for all occurrences. Implementations are free to determine the extent to which this number may be consistent and predictable (or not), and the same is true for the specific cycle in which privilege mode changes.

For the instret counter, most instructions do not affect mode transitions, so for those the behavior is clear: instructions that retire in a non-inhibited mode increment instret, and instructions that retire in an inhibited mode do not. There are two types of instructions that can affect a privilege mode change: instructions that cause synchronous exceptions to a more privileged mode, and xRET instructions that return to a less privileged mode. The former are not considered to retire, and hence do not increment instret. The latter do retire, and should increment instret only if the originating privilege mode is not inhibited.

The instret definition above is intended to ensure that the counter increments in a predictable fashion. For example, consider a scenario where minstretcfg is configured such that all modes other than U-mode are inhibited. A user mode load should increment only once, even if it takes a page fault or other exception. With this definition, the faulting execution of the load will not increment (it does not retire), the handler instructions will not increment (they execute in an inhibited mode), including the xRET (it arguably retires in a non-inhibited mode, but it originates in an inhibited mode). Only once the load is re-executed and retires will it increment instret.

In cases where an instruction is emulated by software running in a privilege mode that is inhibited in minstretcfg, the emulation routine must emulate the instret increment.

Chapter 8. "Smrnmi" Extension for Resumable Non-Maskable Interrupts, Version 1.0

The base machine-level architecture supports only unresumable non-maskable interrupts (UNMIs), where the NMI jumps to a handler in machine mode, overwriting the current `mepc` and `mcause` register values. If the hart had been executing machine-mode code in a trap handler, the previous values in `mepc` and `mcause` would not be recoverable and so execution is not generally resumable.

The Smrnmi extension adds support for resumable non-maskable interrupts (RNMIs) to RISC-V. The extension adds four new CSRs (`mnepc`, `mncause`, `mnstatus`, and `mnscratch`) to hold the interrupted state, and one new instruction, MNRET, to resume from the RNMI handler.

8.1. RNMI Interrupt Signals

The `rnmi` interrupt signals are inputs to the hart. These interrupts have higher priority than any other interrupt or exception on the hart and cannot be disabled by software. Specifically, they are not disabled by clearing the `mstatus`.MIE register.

8.2. RNMI Handler Addresses

The RNMI interrupt trap handler address is implementation-defined.

RNMI also has an associated exception trap handler address, which is implementation defined.

For example, some implementations might use the address specified in `mtvec` as the RNMI exception trap handler.

8.3. RNMI CSRs

This extension adds additional M-mode CSRs to enable a resumable non-maskable interrupt (RNMI).

Figure 38. Resumable NMI scratch register `mnscratch`

The `mnscratch` CSR holds an MXLEN-bit read-write register which enables the NMI trap handler to save and restore the context that was interrupted.

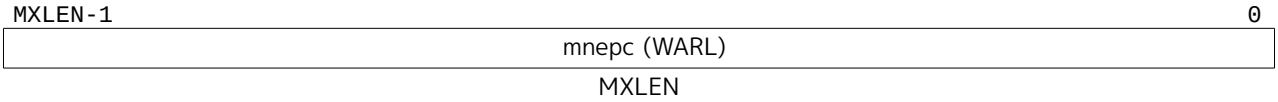


Figure 39. Resumable NMI program counter `mnepc`.

The `mnepc` CSR is an MXLEN-bit read-write register which on entry to the NMI trap handler holds the PC of the instruction that took the interrupt.

The low bit of `mnepc` (`mnepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`mnepc[1:0]`) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR `misa`, for example), then, whenever IALIGN=32, bit `mnepc[1]` is masked on reads so that it appears to be 0. This masking occurs also

for the implicit read by the MRET instruction. Though masked, `mnepc[1]` remains writable when IALIGN=32.

`mnepc` is a **WARL** register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing `mnepc`, implementations may convert an invalid address into some other invalid address that `mnepc` is capable of holding.

Figure 40. Resumable NMI cause `mncause`.

The `mncause` CSR holds the reason for the NMI. If the reason is an interrupt, bit MXLEN-1 is set to 1, and the NMI cause is encoded in the least-significant bits. If the reason is an interrupt and NMI causes are not supported, bit MXLEN-1 is set to 1, and zero is written to the least-significant bits. If the reason is an exception within M-mode that results in a double trap as specified in the Smdbltrp extension, bit MXLEN-1 is set to 0 and the least-significant bits are set to the cause code corresponding to the exception that precipitated the double trap.

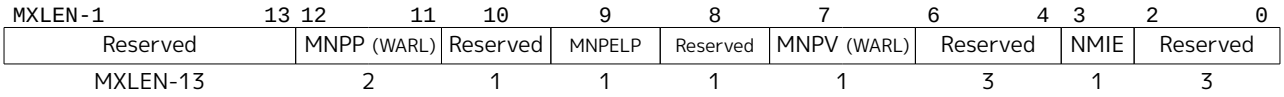


Figure 41. Resumable NMI status register `mnstatus`.

The `mnstatus` CSR holds a two-bit field, MNPP, which on entry to the RNMI trap handler holds the privilege mode of the interrupted context, encoded in the same manner as `mstatus`.MPP. It also holds a one-bit field, MNPV, which on entry to the RNMI trap handler holds the virtualization mode of the interrupted context, encoded in the same manner as `mstatus`.MPV.

If the Zicfilp extension is implemented, `mnstatus` also holds the MNPELP field, which on entry to the RNMI trap handler holds the previous `ELP` state. When an RNMI trap is taken, MNPELP is set to `ELP` and `ELP` is set to 0.

`mnstatus` also holds the NMIE bit. When NMIE=1, nonmaskable interrupts are enabled. When NMIE=0, *all* interrupts are disabled.

When NMIE=0, the hart behaves as though `mstatus`.MPRV were clear, regardless of the current setting of `mstatus`.MPRV.

Upon reset, NMIE contains the value 0.

RNMIs are masked out of reset to give software the opportunity to initialize data structures and devices for subsequent RNMI handling.

Software can set NMIE to 1, but attempts to clear NMIE have no effect.

Normally, only reset sequences will explicitly set the NMIE bit.

That the NMIE bit is settable does not suffice to support the nesting of RNMIs. To support this feature in a direct manner would have required allowing software to clear the NMIE bit—a design choice that would have contravened the concept of non-maskability.

Software that wishes to minimize the latency until the next RNMI is taken can follow the top-half/bottom-half model, where the RNMI handler itself only enqueues a task to a task queue

then returns. The bulk of the interrupt servicing is performed later, with RNMIs enabled.

For the purposes of the WFI instruction, NMIE is a global interrupt enable, meaning that the setting of NMIE does not affect the operation of the WFI instruction.

The other bits in `mnstatus` are *reserved*; software should write zeros and hardware implementations should return zeros.

8.4. MNRET Instruction

MNRET is an M-mode-only instruction that uses the values in `mnepc` and `mnstatus` to return to the program counter, privilege mode, and virtualization mode of the interrupted context. This instruction also sets `mnstatus`.NMIE. If MNRET changes the privilege mode to a mode less privileged than M, it also sets `mstatus`.MPRV to 0. If the Zicfilp extension is implemented, then if the new privileged mode is *y*, MNRET sets `ELP` to the logical AND of *y*LPE (see Section 20.1.1) and `mnstatus`.MNPELP.

8.5. RNMI Operation

When an RNMI interrupt is detected, the interrupted PC is written to the `mnepc` CSR, the type of RNMI to the `mncause` CSR, and the privilege mode of the interrupted context to the `mnstatus` CSR. The `mnstatus`.NMIE bit is cleared, masking all interrupts.

The hart then enters machine-mode and jumps to the RNMI trap handler address.

The RNMI handler can resume original execution using the new MNRET instruction, which restores the PC from `mnepc`, the privilege mode from `mnstatus`, and also sets `mnstatus`.NMIE, which re-enables interrupts.

If the hart encounters an exception while executing in M-mode with the `mnstatus`.NMIE bit clear, the actions taken are the same as if the exception had occurred while `mnstatus`.NMIE were set, except that the program counter is set to the RNMI exception trap handler address.

The Smrnmi extension does not change the behavior of the MRET and SRET instructions. In particular, MRET and SRET are unaffected by the `mnstatus`*.NMIE bit, and their execution does not alter the* `mnstatus`*.NMIE bit.*

Chapter 9. "Smcdeleg" Counter Delegation Extension, Version 1.0

In modern “Rich OS” environments, hardware performance monitoring resources are managed by the kernel, kernel driver, and/or hypervisor. Counters may be configured with differing scopes, in some cases counting events system-wide, while in others counting events on behalf of a single virtual machine or application. In such environments, the latency of counter writes has a direct impact on overall profiling overhead as a result of frequent counter writes during:

1. Sample collection, to clear overflow indication, and reload overflowed counter(s)
2. Context switch, between processes, threads, containers, or virtual machines

This extension provides a means for M-mode to allow writing select counters and event selectors from S/HS-mode. The purpose is to avert transitions to and from M-mode that add latency to these performance critical supervisor/hypervisor code sections. This extension also defines one new CSR, scountinhibit.

For a Machine-level environment, extension **Smcdeleg** (‘Sm’ for Privileged architecture and Machine-level extension, ‘cdeleg’ for Counter Delegation) encompasses all added CSRs and all behavior modifications for a hart, over all privilege levels. For a Supervisor-level environment, extension **Ssccfg** (‘Ss’ for Privileged architecture and Supervisor-level extension, ‘ccfg’ for Counter Configuration) provides access to delegated counters, and to new supervisor-level state.

9.1. Counter Delegation

The `mcounteren` register allows M-mode to provide the next-lower privilege mode with read access to select counters. When the Smcdeleg/Ssccfg extension is enabled (`menvcfg`.CDE=1), it further allows M-mode to delegate select counters to S-mode.

The `siselect` (and `vsiselect`) index range 0x40-0x5F is reserved for delegated counter access. When a counter *i* is delegated (`mcounteren`[*i*]=1 and `menvcfg`.CDE=1), the register state associated with counter *i* can be read or written via `sireg*`, while `siselect` holds 0x40+*i*. The counter state accessible via alias CSRs is shown in the table below.

Table 20. Indirect HPM State Mappings

`siselect` value	`sireg`	`sireg4`	`sireg2`	`sireg5`
0x40	`cycle`[1]	`cycleh`[1]	`cyclecfg`[14]	`cyclecfgh`[14]
0x41	*See below*			
0x42	`instret`[1]	`instreth`[1]	`instretcfg`[14]	`instretcfgh`[14]
0x43	`hpmcounter3`[2]	`hpmcounter3h`[2]	`hpmevent3`[2]	`hpmevent3h`[23]
…	…	…	…	…
0x5F	`hpmcounter31`[2]	`hpmcounter31h`[2]	`hpmevent31`[2]	`hpmevent31h`[23]

[1] Depends on Zicntr support
[2] Depends on Zihpm support
[3] Depends on Sscofpmf support
[4] Depends on Smcntrpmf support

`hpmeventi` represents a subset of the state accessed by the `mhpmeventi` register. Likewise, `cyclecfg` and `instretcfg` represent a subset of the state accessed by the `mcyclecfg` and `minstretcfg` registers, respectively. See below for subset details.

If extension Smstateen is implemented, refer to extension Smcsrind/Sscsrind (Chapter 5) for how setting bit 60 of CSR mstateen0 to zero prevents access to registers siselect, sireg*, vsiselect, and vsireg* from privileged modes less privileged than M-mode, and likewise how setting bit 60 of hstateen0 to zero prevents access to siselect and sireg* (really vsiselect and vsireg*) from VS-mode.

The remaining rules of this section apply only when access to a CSR is not blocked by mstateen0[60] = 0 or hstateen0[60] = 0.

While the privilege mode is M or S and siselect holds a value in the range 0x40-0x5F, illegal instruction exceptions are raised for the following cases:

- attempts to access any sireg* when menvcfg.CDE = 0;
- attempts to access sireg3 or sireg6;
- attempts to access sireg4 or sireg5 when XLEN = 64;
- attempts to access sireg* when siselect = 0x41, or when the counter selected by siselect is not delegated to S-mode (the corresponding bit in mcounteren = 0).

The memory-mapped mtime register is not a performance monitoring counter to be managed by supervisor software, hence the special treatment of siselect value 0x41 described above.

For each siselect and sireg* combination defined in Table 20, the table further indicates the extensions upon which the underlying counter state depends. If any extension upon which the underlying state depends is not implemented, an attempt from M or S mode to access the given state through sireg* raises an illegal instruction exception.

If the hypervisor (H) extension is also implemented, then as specified by extension Smcsrind/Sscsrind, a virtual instruction exception is raised for attempts from VS-mode or VU-mode to directly access vsiselect or vsireg*, or attempts from VU-mode to access siselect or sireg*. Furthermore, while vsiselect holds a value in the range 0x40-0x5F:

- An attempt to access any vsireg* from M or S mode raises an illegal instruction exception.
- An attempt from VS-mode to access any sireg* (really vsireg*) raises an illegal instruction exception if menvcfg.CDE = 0, or a virtual instruction exception if menvcfg.CDE = 1.

If Sscofpmf is implemented, sireg2 and sireg5 provide access only to a subset of the event selector registers. Specifically, event selector bit 62 (MINH) is read-only 0 when accessed through sireg*. Similarly, if Smcntrpmf is implemented, sireg2 and sireg5 provide access only to a subset of the counter configuration registers. Counter configuration register bit 62 (MINH) is read-only 0 when accessed through sireg*.

9.2. Supervisor Counter Inhibit (scountinhibit) Register

Smcdeleg/Ssccfg defines a new scountinhibit register, a masked alias of mcountinhibit. For counters delegated to S-mode, the associated mcountinhibit bits can be accessed via scountinhibit. For counters not delegated to S-mode, the associated bits in scountinhibit are read-only zero.

When menvcfg.CDE=0, attempts to access scountinhibit raise an illegal instruction exception. When the Supervisor Counter Delegation extension is enabled, attempts to access scountinhibit from VS-mode or VU-mode raise a virtual instruction exception.

9.3. Virtualizing scountovf

For implementations that support Smcdeleg/Ssccfg, Sscofpmf, and the H extension, when menvcfg.CDE=1, attempts to read scountovf from VS-mode or VU-mode raise a virtual instruction exception.

9.4. Virtualizing Local Counter Overflow Interrupts

For implementations that support Smcdeleg, Sscofpmf, and Smaia, the local counter overflow interrupt (LCOFI) bit (bit 13) in each of CSRs mvip and mvien is implemented and writable.

For implementations that support Smcdeleg/Ssccfg, Sscofpmf, Smaia/Ssaia, and the H extension, the LCOFI bit (bit 13) in each of hvip and hvien is implemented and writable.

The hvip register is defined by the hypervisor (H) extension, while the mvien and hvien registers are defined by the Smaia/Ssaia extension.

By virtue of implementing hvip.LCOFI, it is implicit that the LCOFI bit (bit 13) in each of vsie and vsip is also implemented.

Requiring support for the LCOFI bits listed above ensures that virtual LCOFIs can be delivered to an OS running in S-mode, and to a guest OS running in VS-mode. It is optional whether the LCOFI bit (bit 13) in each of mideleg and hideleg, which allows all LCOFIs to be delegated to S-mode and VS-mode, respectively, is implemented and writable.

Chapter 10. "Smdbltrp" Double Trap Extension, Version 1.0

The Smdbltrp extension addresses a double trap (See Section 3.1.6.2) in M-mode. When the Smrnmi extension (Chapter 8) is implemented, it enables invocation of the RNMI handler on a double trap in M-mode to handle the critical error. If the Smrnmi extension is not implemented or if a double trap occurs during the RNMI handler's execution, this extension helps transition the hart to a critical error state and enables signaling the critical error to the platform.

To improve error diagnosis and resolution, this extension supports debugging harts in a critical error state. The extension introduces a mechanism to enter Debug Mode instead of asserting a critical-error signal to the platform when the hart is in a critical error state. See (*The RISC-V Debug Specification*, n.d.) for details.

See Section 3.1.6.2 for the operational details.

Chapter 11. Supervisor-Level ISA, Version 1.13

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.

11.1. Supervisor CSRs

A number of CSRs are provided for the supervisor.

The supervisor should only view CSR state that should be visible to a supervisor-level operating system. In particular, there is no information about the existence (or non-existence) of higher privilege levels (machine level or other) visible in the CSRs accessible by the supervisor.

Many supervisor CSRs are a subset of the equivalent machine-mode CSR, and the machine-mode chapter should be read first to help understand the supervisor-level CSR descriptions.

11.1.1. Supervisor Status (`sstatus`) Register

The `sstatus` register is an SXLEN-bit read/write register formatted as shown in Figure 42 when SXLEN=32 and Figure 43 when SXLEN=64. The `sstatus` register keeps track of the processor's current operating state.

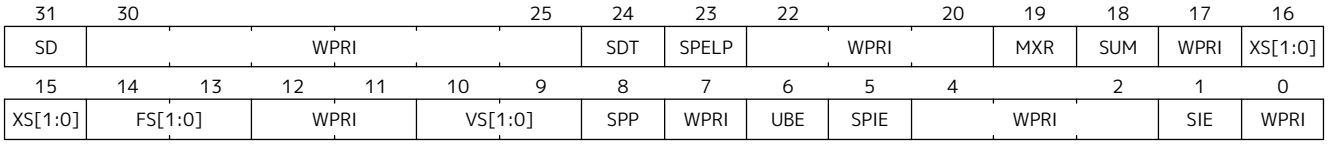


Figure 42. Supervisor-mode status (`sstatus`) register when SXLEN=32.

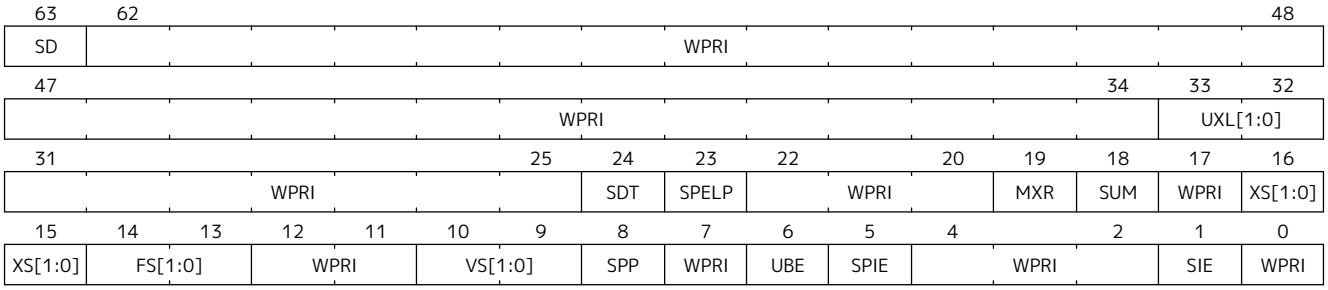


Figure 43. Supervisor-mode status (`sstatus`) register when SXLEN=64.

The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction (see Section 3.3.2) is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the

`sie` CSR.

The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1.

The `sstatus` register is a subset of the `mstatus` register.

In a straightforward implementation, reading or writing any field in `sstatus` *is equivalent to reading or writing the homonymous field in* `mstatus`.

11.1.1.1. Base ISA Control in `sstatus` Register

The UXL field controls the value of XLEN for U-mode, termed *UXLEN*, which may differ from the value of XLEN for S-mode, termed *SXLEN*. The encoding of UXL is the same as that of the MXL field of `misa`, shown in Table 9.

When SXLEN=32, the UXL field does not exist, and UXLEN=32. When SXLEN=64, it is a **WARL** field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=SXLEN.

If UXLEN≠SXLEN, instructions executed in the narrower mode must ignore source register operand bits above the configured XLEN, and must sign-extend results to fill the widest supported XLEN in the destination register.

If UXLEN < SXLEN, user-mode instruction-fetch addresses and load and store effective addresses are taken modulo 2^{UXLEN}. For example, when UXLEN=32 and SXLEN=64, user-mode memory accesses reference the lowest 4 GiB of the address space.

Some HINT instructions are encoded as integer computational instructions that overwrite their destination register with its current value, e.g., `c.addi x8, 0`. When such a HINT is executed with XLEN < SXLEN and bits SXLEN..XLEN of the destination register not all equal to bit XLEN-1, it is implementation-defined whether bits SXLEN..XLEN of the destination register are unchanged or are overwritten with copies of bit XLEN-1.

This definition allows implementations to elide register writeback for some HINTs, while allowing them to execute other HINTs in the same manner as other integer computational instructions. The implementation choice is observable only by S-mode with SXLEN > UXLEN; it is invisible to U-mode.

11.1.1.2. Memory Privilege in `sstatus` Register

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in Figure 60) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect.

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Figure 60) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect, nor when executing in U-mode. Note that S-mode can never execute instructions from user pages, regardless of the state of SUM.

SUM is read-only 0 if `satp`.MODE is read-only 0.

The SUM mechanism prevents supervisor software from inadvertently accessing user memory. Operating systems can execute the majority of code with SUM clear; the few code segments that should access user memory can temporarily set SUM.

The SUM mechanism does not avail S-mode software of permission to execute instructions in user code pages. Legitimate uses cases for execution from user memory in supervisor context are rare in general and nonexistent in POSIX environments. However, bugs in supervisors that lead to arbitrary code execution are much easier to exploit if the supervisor exploit code can be stored in a user buffer at a virtual address chosen by an attacker.

Some non-POSIX single address space operating systems do allow certain privileged software to partially execute in supervisor mode, while most programs run in user mode, all in a shared address space. This use case can be realized by mapping the physical code pages at multiple virtual addresses with different permissions, possibly with the assistance of the instruction page-fault handler to direct supervisor software to use the alternate mapping.

11.1.1.3. Endianness Control in `sstatus` Register

The UBE bit is a **WARL** field that controls the endianness of explicit memory accesses made from U-mode, which may differ from the endianness of memory accesses in S-mode. An implementation may make UBE be a read-only field that always specifies the same endianness as for S-mode.

UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1).

UBE has no effect on instruction fetches, which are *implicit* memory accesses that are always little-endian.

For *implicit* accesses to supervisor-level memory management data structures, such as page tables, S-mode endianness always applies and UBE is ignored.

Standard RISC-V ABIs are expected to be purely little-endian-only or big-endian-only, with no accommodation for mixing endianness. Nevertheless, endianness control has been defined so as to permit an OS of one endianness to execute user-mode programs of the opposite endianness.

11.1.1.4. Previous Expected Landing Pad (ELP) State in `sstatus` Register

Access to the `SPELP` field, added by Zicfilp, accesses the homonymous fields of `mstatus` when `V=0`, and the homonymous fields of `vsstatus` when `V=1`.

11.1.1.5. Double Trap Control in `sstatus` Register

The S-mode-disable-trap (`SDT`) bit is a WARL field introduced by the Ssdbltrp extension to address double trap (See Section 3.1.6.2) at privilege modes lower than M.

When the `SDT` bit is set to 1 by an explicit CSR write, the `SIE` (Supervisor Interrupt Enable) bit is cleared to 0. This clearing occurs regardless of the value written, if any, to the `SIE` bit by the same write. The `SIE` bit can only be set to 1 by an explicit CSR write if the `SDT` bit is being set to 0 by the same write or is already 0.

When a trap is to be taken into S-mode, if the `SDT` bit is currently 0, it is then set to 1, and the trap is delivered as expected. However, if `SDT` is already set to 1, then this is an *unexpected trap*. In the event of an *unexpected trap*, a double-trap exception trap is delivered into M-mode. To deliver this trap, the hart writes registers, except `mcause` and `mtval2`, with the same information that the *unexpected trap* would have written if it was taken into M-mode. The `mtval2` register is then set to what would be otherwise written into the

`mcause` register by the *unexpected trap*. The `mcause` register is set to 16, the double-trap exception code.

An `SRET` instruction sets the `SDT` bit to 0.

After a trap handler has saved the state, such as `scause`, `sepc`, and `stval`, needed for resuming from the trap and is reentrant, it should clear the `SDT` bit.

Resetting the `SDT` by an `SRET` enables the trap handler to detect a double trap that may occur during the tail phase, where it restores critical state to return from a trap.

The consequence of this specification is that if a critical error condition was caused by a guest page-fault, then the GPA will not be available in `mtval2` when the double trap is delivered to M-mode. This condition arises if the HS-mode invokes a hypervisor virtual-machine load or store instruction when `SDT` is 1 and the instruction raises a guest page-fault. The use of such an instruction in this phase of trap handling is not common. However, not recording the GPA is considered benign because, if required, it can still be obtained — albeit with added effort — through the process of walking the page tables.

For a double trap that originates in VS-mode, M-mode should redirect the exception to HS-mode by copying the values of M-mode CSRs updated by the trap to HS-mode CSRs and should use an `MRET` to resume execution at the address in `stvec`.

Supervisor Software Events (SSE), an extension to the SBI, provide a mechanism for supervisor software to register and service system events emanating from an SBI implementation, such as firmware or a hypervisor. In the event of a double trap, HS-mode and M-mode can utilize the SSE mechanism to invoke a critical-error handler in VS-mode or S/HS-mode, respectively. Additionally, the implementation of an SSE protocol can be considered as an optional measure to aid in the recovery from such critical errors.

11.1.2. Supervisor Trap Vector Base Address (`stvec`) Register

The `stvec` register is an SXLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

Figure 44. Supervisor trap vector base address (`stvec`) register.

The BASE field in `stvec` is a field that can hold any valid virtual or physical address, subject to the following alignment constraints: the address must be 4-byte aligned, and MODE settings other than Direct might impose additional alignment constraints on the value in the BASE field.

Table 21. Encoding of `stvec` MODE field.

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2		*Reserved*

The encoding of the MODE field is shown in Table 21. When MODE=Direct, all traps into supervisor mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into supervisor mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number. For example, a

supervisor-mode timer interrupt (see Table 22) causes the `pc` to be set to BASE+`0x14`. Setting MODE=Vectored may impose a stricter alignment constraint on BASE.

11.1.3. Supervisor Interrupt (`sip` and `sie`) Registers

The `sip` register is an SXLEN-bit read/write register containing information on pending interrupts, while `sie` is the corresponding SXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number *i* (as reported in CSR `scause`, Section 11.1.8) corresponds with bit *i* in both `sip` and `sie`. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.

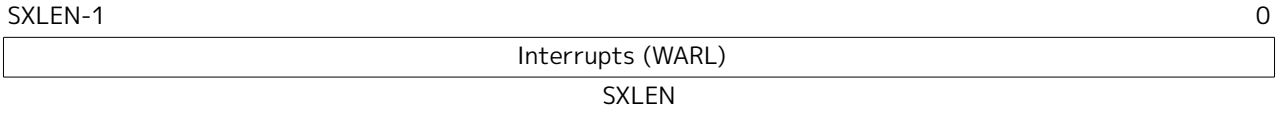


Figure 45. Supervisor interrupt-pending register (`sip`).

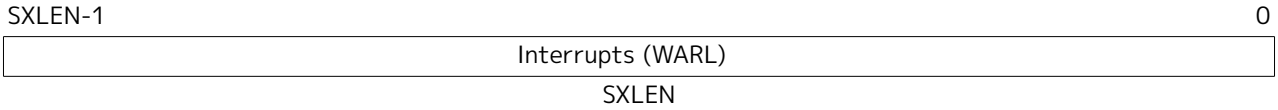


Figure 46. Supervisor interrupt-enable register (`sie`).

An interrupt *i* will trap to S-mode if both of the following are true: (a) either the current privilege mode is S and the SIE bit in the `sstatus` register is set, or the current privilege mode has less privilege than S-mode; and (b) bit *i* is set in both `sip` and `sie`.

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in `sip`, and must also be evaluated immediately following the execution of an SRET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including `sip`, `sie` and `sstatus`).

Interrupts to S-mode take priority over any interrupts to lower privilege modes.

Each individual bit in register `sip` may be writable or may be read-only. When bit *i* in `sip` is writable, a pending interrupt *i* can be cleared by writing 0 to this bit. If interrupt *i* can become pending but bit *i* in `sip` is read-only, the implementation must provide some other mechanism for clearing the pending interrupt (which may involve a call to the execution environment).

A bit in `sie` must be writable if the corresponding interrupt can ever become pending. Bits of `sie` that are not writable are read-only zero.

The standard portions (bits 15:0) of registers `sip` and `sie` are formatted as shown in Figures Figure 47 and Figure 48 respectively.

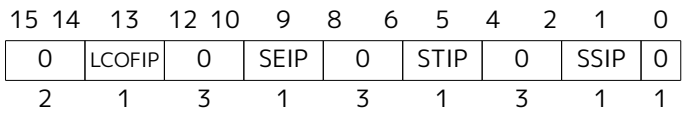


Figure 47. Standard portion (bits 15:0) of `sip`.

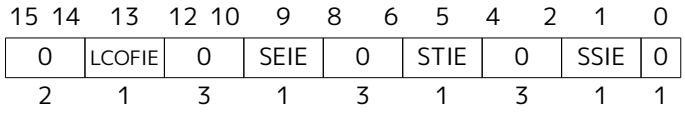


Figure 48. Standard portion (bits 15:0) of `sie`.

Bits `sip`.SEIP and `sie`.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external

interrupts. If implemented, SEIP is read-only in `sip`, and is set and cleared by the execution environment, typically through a platform-specific interrupt controller.

Bits `sip`.STIP and `sie`.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If implemented, STIP is read-only in `sip`, and is set and cleared by the execution environment.

Bits `sip`.SSIP and `sie`.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. If implemented, SSIP is writable in `sip` and may also be set to 1 by a platform-specific interrupt controller.

If the Sscofpmf extension is implemented, bits `sip`.LCOFIP and `sie`.LCOFIE are the interrupt-pending and interrupt-enable bits for local counter-overflow interrupts. LCOFIP is read-write in `sip` and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the `mhpmeventn`.OF bits being set. If the Sscofpmf extension is not implemented, `sip`.LCOFIP and `sie`.LCOFIE are read-only zeros.

Interprocessor interrupts are sent to other harts by implementation-specific means, which will ultimately cause the SSIP bit to be set in the recipient hart's `sip` register.

Each standard interrupt type (SEI, STI, SSI, or LCOFI) may not be implemented, in which case the corresponding interrupt-pending and interrupt-enable bits are read-only zeros. All bits in `sip` and `sie` are **WARL** fields. The implemented interrupts may be found by writing one to every bit location in `sie`, then reading back to see which bit positions hold a one.

The `sip` and `sie` registers are subsets of the `mip` and `mie` registers. Reading any implemented field, or writing any writable field, of `sip`/`sie` effects a read or write of the homonymous field of `mip`/`mie`.

Bits 3, 7, and 11 of `sip` and `sie` correspond to the machine-mode software, timer, and external interrupts, respectively. Since most platforms will choose not to make these interrupts delegatable from M-mode to S-mode, they are shown as 0 in Figure 47 and Figure 48.

Multiple simultaneous interrupts destined for supervisor mode are handled in the following decreasing priority order: SEI, SSI, STI, LCOFI.

11.1.4. Supervisor Timers and Performance Counters

Supervisor software uses the same hardware performance monitoring facility as user-mode software, including the `time`, `cycle`, and `instret` CSRs. The implementation should provide a mechanism to modify the counter values.

The implementation must provide a facility for scheduling timer interrupts in terms of the real-time counter, `time`.

11.1.5. Counter-Enable (`scounteren`) Register

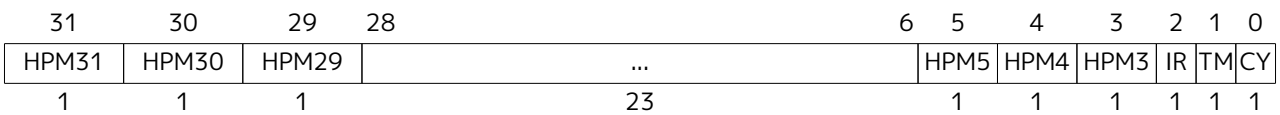


Figure 49. Counter-enable (`scounteren`) register

The counter-enable (`scounteren`) CSR is a 32-bit register that controls the availability of the hardware performance monitoring counters to U-mode.

When the CY, TM, IR, or HPM*n* bit in the `scounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcountern` register while executing in U-mode will cause an illegal-instruction exception. When one of these bits is set, access to the corresponding register is permitted.

`scounteren` must be implemented. However, any of the bits may be read-only zero, indicating reads to the corresponding counter will cause an exception when executing in U-mode. Hence, they are effectively **WARL** fields.

The setting of a bit in `mcounteren` *does not affect whether the corresponding bit in* `scounteren` *is writable. However, U-mode may only access a counter if the corresponding bits in* `scounteren` *and* `mcounteren` *are both set.*

11.1.6. Supervisor Scratch (`sscratch`) Register

The `sscratch` CSR is an SXLEN-bit read/write register, dedicated for use by the supervisor. Typically, `sscratch` is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, `sscratch` is swapped with a user register to provide an initial working register.

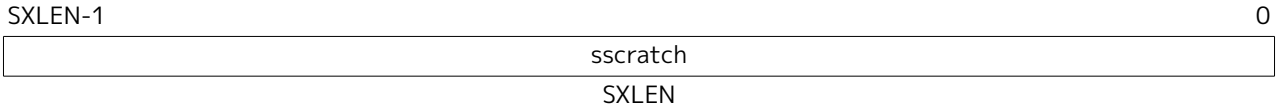


Figure 50. Supervisor Scratch Register

11.1.7. Supervisor Exception Program Counter (`sepc`) Register

`sepc` is an SXLEN-bit read/write CSR formatted as shown in Figure 51. The low bit of `sepc` (`sepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`sepc[1:0]`) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR `misa`, for example), then, whenever IALIGN=32, bit `sepc[1]` is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the SRET instruction. Though masked, `sepc[1]` remains writable when IALIGN=32.

`sepc` is a **WARL** register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing `sepc`, implementations may convert an invalid address into some other invalid address that `sepc` is capable of holding.

When a trap is taken into S-mode, `sepc` is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, `sepc` is never written by the implementation, though it may be explicitly written by software.

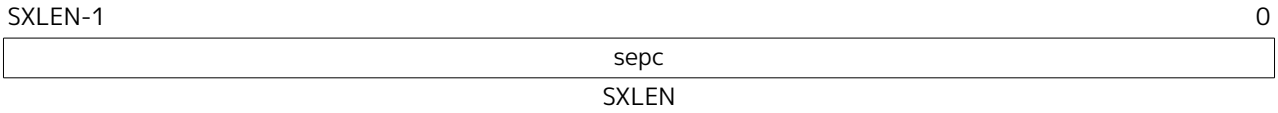


Figure 51. Supervisor exception program counter register.

11.1.8. Supervisor Cause (`scause`) Register

The `scause` CSR is an SXLEN-bit read-write register formatted as shown in Figure 52. When a trap is taken into S-mode, `scause` is written with a code indicating the event that caused the trap. Otherwise, `scause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `scause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Table 22 lists the possible exception codes for the current supervisor ISAs. The Exception Code is a **WLRL** field. It is required to hold the values 0–31 (i.e., bits 4–0 must be implemented), but otherwise it is only guaranteed to hold supported exception codes.

Figure 52. Supervisor Cause (`scause`) register.

Table 22. Supervisor cause (`scause`) register values after trap. Synchronous exception priorities are given by Table 15.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2-4	*Reserved*
1	5	Supervisor timer interrupt
1	6-8	*Reserved*
1	9	Supervisor external interrupt
1	10-12	*Reserved*
1	13	Counter-overflow interrupt
1	14-15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16-17	*Reserved*
0	18	Software check
0	19	Hardware error
0	20-23	*Reserved*
0	24-31	*Designated for custom use*
	32-47	*Reserved*
	48-63	*Designated for custom use*
	≥64	*Reserved*

11.1.9. Supervisor Trap Value (`stval`) Register

The `stval` CSR is an SXLEN-bit read-write register formatted as shown in Figure 53. When a trap is taken into S-mode, `stval` is written with exception-specific information to assist software in handling the trap. Otherwise, `stval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `stval` informatively, which may unconditionally set it to zero, and which may exhibit either behavior, depending on the underlying event that caused the exception.

If `stval` is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, then `stval` will contain the faulting virtual address.

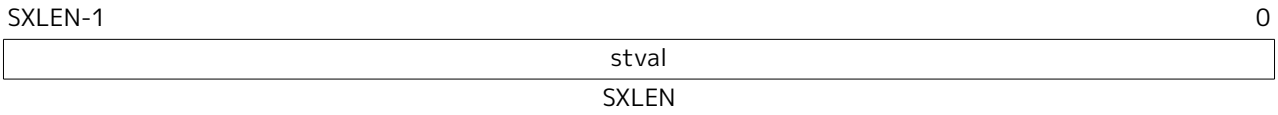


Figure 53. Supervisor Trap Value register.

If `stval` is written with a nonzero value when a misaligned load or store causes an access-fault or page-fault exception, then `stval` will contain the virtual address of the portion of the access that caused the fault.

If `stval` is written with a nonzero value when an instruction access-fault or page-fault exception occurs on a system with variable-length instructions, then `stval` will contain the virtual address of the portion of the instruction that caused the fault, while `sepc` will point to the beginning of the instruction.

The `stval` register can optionally also be used to return the faulting instruction bits on an illegal-instruction exception (`sepc` points to the faulting instruction in memory). If `stval` is written with a nonzero value when an illegal-instruction exception occurs, then `stval` will contain the shortest of:

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first SXLEN bits of the faulting instruction

The value loaded into `stval` on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

On a trap caused by a software check exception, the `stval` register holds the cause for the exception. The following encodings are defined:

- 0 - No information provided.
- 2 - Landing Pad Fault. Defined by the Zicfilp extension (Section 20.1).
- 3 - Shadow Stack Fault. Defined by the Zicfiss extension (Section 20.2).

For other traps, `stval` is set to zero, but a future standard may redefine `stval`'s setting for other traps.

`stval` is a **WARL** register that must be able to hold all valid virtual addresses and the value 0. It need not be capable of holding all possible invalid addresses. Prior to writing `stval`, implementations may convert an invalid address into some other invalid address that `stval` is capable of holding. If the feature to return the faulting instruction bits is implemented, `stval` must also be able to hold all values less than 2^N, where N is the smaller of SXLEN and ILEN.

11.1.10. Supervisor Environment Configuration (`senvcfg`) Register

The `senvcfg` CSR is an SXLEN-bit read/write register, formatted as shown in Figure 54, that controls certain characteristics of the U-mode execution environment.

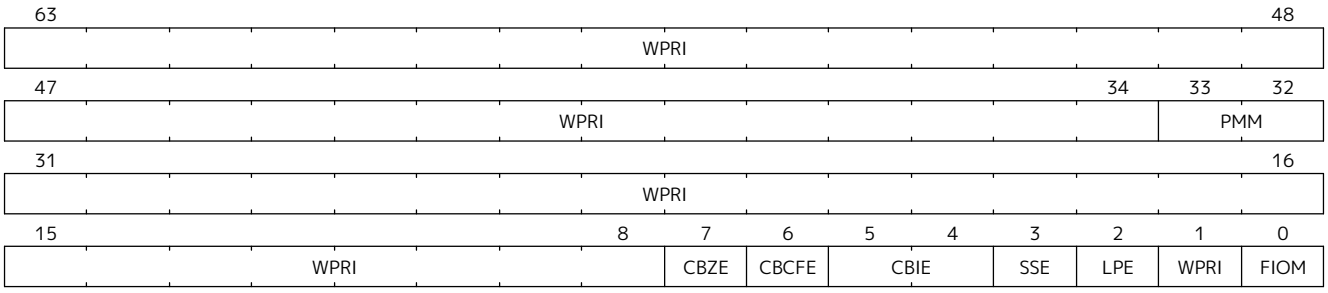


Figure 54. Supervisor environment configuration register (`senvcfg`) for RV64.

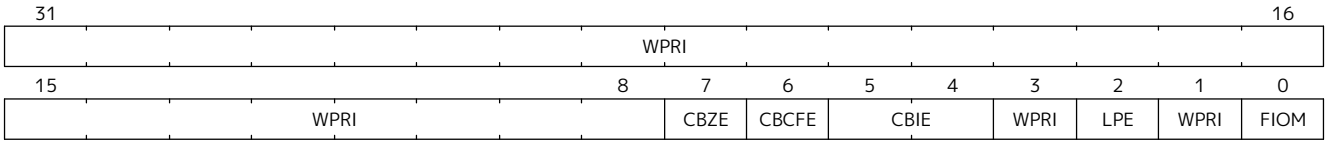


Figure 55. Supervisor environment configuration register (`senvcfg`) for RV32.

If bit FIOM (Fence of I/O implies Memory) is set to one in `senvcfg`, FENCE instructions executed in U-mode are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. Table 23 details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO in U-mode when FIOM=1.

Similarly, for 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.

If `satp`.MODE is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 23. Modified interpretation of FENCE predecessor and successor sets in U-mode when FIOM=1.

Instruction bit	Meaning when set
PI PO	Predecessor device input and memory reads (PR implied) Predecessor device output and memory writes (PW implied)
SI SO	Successor device input and memory reads (SR implied) Successor device output and memory writes (SW implied)

Bit FIOM exists for a specific circumstance when an I/O device is being emulated for U-mode and both of the following are true: (a) the emulated device has a memory buffer that should be I/O space but is actually mapped to main memory via address translation, and (b) multiple physical harts are involved in accessing this emulated device from U-mode.

A hypervisor running in S-mode without the benefit of the hypervisor extension of Chapter 19 may need to emulate a device for U-mode if paravirtualization cannot be employed. If the same hypervisor provides a virtual machine (VM) with multiple virtual harts, mapped one-to-one to real harts, then multiple harts may concurrently access the emulated device, perhaps because: (a) the guest OS within the VM assigns device interrupt handling to one hart while the device is also accessed by a different hart outside of an interrupt handler, or (b) control of the device (or partial control) is being migrated from one hart to another, such as for interrupt load balancing within the VM. For such cases, guest software within the VM is expected to properly coordinate access to the (emulated) device across multiple harts using mutex locks and/or interprocessor interrupts as usual, which in part entails executing I/O fences. But those I/O fences may not be sufficient if some of the device "I/O" is actually main memory, unknown to the guest. Setting FIOM=1 modifies those fences (and all other I/O fences executed in U-mode) to include main memory, too.

Software can always avoid the need to set FIOM by never using main memory to emulate a

device memory buffer that should be I/O space. However, this choice usually requires trapping all U-mode accesses to the emulated buffer, which might have a noticeable impact on performance. The alternative offered by FIOM is sufficiently inexpensive to implement that we consider it worth supporting even if only rarely enabled.

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 is furnished by the Ssnpm extension.

The Zicfilp extension adds the `LPE` field in `senvcfg`. When the `LPE` field is set to 1, the Zicfilp extension is enabled in VU/U-mode. When the `LPE` field is 0, the Zicfilp extension is not enabled in VU/U-mode and the following rules apply to VU/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 in `senvcfg`. When the `SSE` field is set to 1, the Zicfiss extension is activated in VU/U-mode. When the `SSE` field is 0, the Zicfiss extension remains inactive in VU/U-mode, and the following rules apply:

- 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.
- When `menvcfg.SSE` is one, `SSAMOSWAP.W/D` raises an illegal-instruction exception in U-mode and a virtual instruction exception in VU-mode.

11.1.11. Supervisor Address Translation and Protection (`satp`) Register

The `satp` CSR is an SXLEN-bit read/write register, formatted as shown in Figure 56 for SXLEN=32 and Figure 57 for SXLEN=64, which controls supervisor-mode address translation and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the MODE field, which selects the current address-translation scheme. Further details on the access to this register are described in Section 3.1.6.6.

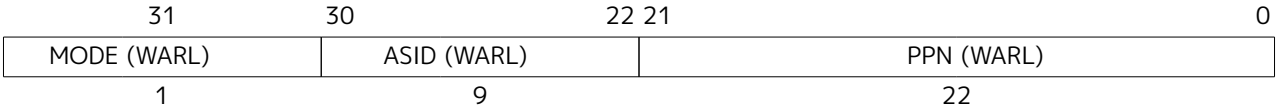


Figure 56. Supervisor address translation and protection (`satp`) register when SXLEN=32.

Storing a PPN in `satp`, rather than a physical address, supports a physical address space larger than 4 GiB for RV32.

The `satp`.PPN field might not be capable of holding all physical page numbers. Some platform standards might place constraints on the values `satp`.PPN may assume, e.g., by requiring that all physical page numbers corresponding to main memory be representable.

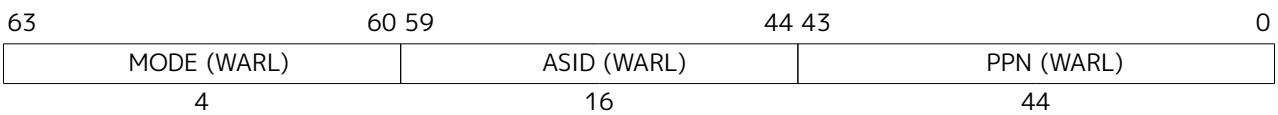


Figure 57. Supervisor address translation and protection (`satp`) register when SXLEN=64, for MODE values Bare, Sv39, Sv48, and Sv57.

We store the ASID and the page table base address in the same CSR to allow the pair to be changed atomically on a context switch. Swapping them non-atomically could pollute the old virtual address space with new translations, or vice-versa. This approach also slightly reduces the cost of a context switch.

Table 24 shows the encodings of the MODE field when SXLEN=32 and SXLEN=64. When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 3.7. To select MODE=Bare, software must write zero to the remaining fields of `satp` (bits 30–0 when SXLEN=32, or bits 59–0 when SXLEN=64). Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an UNSPECIFIED effect on the value that the remaining fields assume and an UNSPECIFIED effect on address translation and protection behavior.

When SXLEN=32, the `satp` encodings corresponding to MODE=Bare and ASID[8:7]=3 are designated for custom use, whereas the encodings corresponding to MODE=Bare and ASID[8:7]≠3 are reserved for future standard use. When SXLEN=64, all `satp` encodings corresponding to MODE=Bare are reserved for future standard use.

Version 1.11 of this standard stated that the remaining fields in `satp` had no effect when MODE=Bare. Making these fields reserved facilitates future definition of additional translation and protection modes, particularly in RV32, for which all patterns of the existing MODE field have already been allocated.

When SXLEN=32, the only other valid setting for MODE is Sv32, a paged virtual-memory scheme described in Section 11.3.

When SXLEN=64, three paged virtual-memory schemes are defined: Sv39, Sv48, and Sv57, described in Section 11.4, Section 11.5, and Section 11.6, respectively. One additional scheme, Sv64, will be defined in a later version of this specification. The remaining MODE settings are reserved for future use and may define different interpretations of the other fields in `satp`.

Implementations are not required to support all MODE settings, and if `satp` is written with an unsupported MODE, the entire write has no effect; no fields in `satp` are modified.

The number of ASID bits is UNSPECIFIED and may be zero. The number of implemented ASID bits, termed *ASIDLEN*, may be determined by writing one to every bit position in the ASID field, then reading back the value in `satp` to see which bit positions in the ASID field hold a one. The least-significant bits of ASID are implemented first: that is, if ASIDLEN > 0, ASID[ASIDLEN-1:0] is writable. The maximal value of ASIDLEN, termed ASIDMAX, is 9 for Sv32 or 16 for Sv39, Sv48, and Sv57.

Table 24. Encoding of `satp` *MODE field.*

SXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 11.3).
SXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	-	*Reserved for standard use*
8	Sv39	Page-based 39-bit virtual addressing (see Section 11.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 11.5).
10	Sv57	Page-based 57-bit virtual addressing (see Section 11.6).
11	Sv64	*Reserved for page-based 64-bit virtual addressing.*
12-13	-	*Reserved for standard use*
14-15	-	*Designated for custom use*

For many applications, the choice of page size has a substantial performance impact. A large page size increases TLB reach and loosens the associativity constraints on virtually indexed, physically tagged caches. At the same time, large pages exacerbate internal fragmentation, wasting physical memory and possibly cache capacity.

After much deliberation, we have settled on a conventional page size of 4 KiB for both RV32 and RV64. We expect this decision to ease the porting of low-level runtime software and device drivers. The TLB reach problem is ameliorated by transparent superpage support in modern operating systems. (Navarro et al., 2002) Additionally, multi-level TLB hierarchies are quite inexpensive relative to the multi-level cache hierarchies whose address space they map.

The `satp` CSR is considered *active* when the effective privilege mode is S-mode or U-mode. Executions of the address-translation algorithm may only begin using a given value of `satp` when `satp` is active.

Translations that began while `satp` *was active are not required to complete or terminate when* `satp` *is no longer active, unless an SFENCE.VMA instruction matching the address and ASID is executed. The SFENCE.VMA instruction must be used to ensure that updates to the address-translation data structures are observed by subsequent implicit reads to those structures by a hart.*

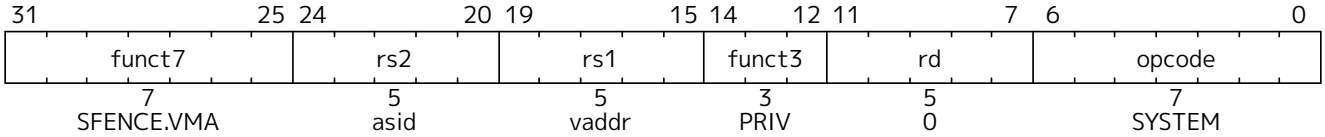
Note that writing `satp` does not imply any ordering constraints between page-table updates and subsequent address translations, nor does it imply any invalidation of address-translation caches. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an SFENCE.VMA instruction (see Section 11.2.1) after, or in some cases before, writing `satp`.

Not imposing upon implementations to flush address-translation caches upon `satp` *writes reduces the cost of context switches, provided a sufficiently large ASID space.*

11.2. Supervisor Instructions

In addition to the SRET instruction defined in Section 3.3.2, one new supervisor-level instruction is provided.

11.2.1. Supervisor Memory-Management Fence Instruction



The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by *rs1* and *rs2*, as described below. SFENCE.VMA is also used to invalidate entries in the address-translation cache associated with a hart (see Section 11.3.2). Further details on the behavior of this instruction are described in Section 3.1.6.6 and Section 3.7.2.

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

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

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

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

- If *rs1*=`x0` and *rs2*=`x0`, the fence orders all reads and writes made to any level of the page tables, for all address spaces. The fence also invalidates all address-translation cache entries, for all address spaces.
- If *rs1*=`x0` and *rs2*≠`x0`, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register *rs2*. Accesses to *global* mappings (see Section 11.3.1) are not ordered. The fence also invalidates all address-translation cache entries matching the address space identified by integer register *rs2*, except for entries containing global mappings.
- If *rs1*≠`x0` and *rs2*=`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces.
- If *rs1*≠`x0` and *rs2*≠`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for the address space identified by integer register *rs2*. Accesses to global mappings are not ordered. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1* and that match the address space identified by integer register *rs2*, except for entries containing global mappings.

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

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

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

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

A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. In particular, if a leaf PTE is modified but a subsuming SFENCE.VMA is not executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.

In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with rs1=`x0`. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.

Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.

This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

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

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

The following common situations typically require executing an SFENCE.VMA instruction:

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

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

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

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

11.3. Sv32: Page-Based 32-bit Virtual-Memory Systems

When Sv32 is written to the MODE field in the `satp` register (see Section 11.1.11), the supervisor operates in a 32-bit paged virtual-memory system. In this mode, supervisor and user virtual addresses are translated into supervisor physical addresses by traversing a radix-tree page table. Sv32 is supported when SXLEN=32 and is designed to include mechanisms sufficient for supporting modern Unix-based operating systems.

The initial RISC-V paged virtual-memory architectures have been designed as straightforward implementations to support existing operating systems. We have architected page table layouts to support a hardware page-table walker. Software TLB refills are a performance bottleneck on high-performance systems, and are especially troublesome with decoupled specialized coprocessors. An implementation can choose to implement software TLB refills using a machine-mode trap handler as an extension to M-mode.

Some ISAs architecturally expose virtually indexed, physically tagged caches, in that accesses to the same physical address via different virtual addresses might not be coherent unless the virtual addresses lie within the same cache set. Implicitly, this specification does not permit such behavior to be architecturally exposed.

11.3.1. Addressing and Memory Protection

Sv32 implementations support a 32-bit virtual address space, divided into pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown in Figure 58. When Sv32 virtual memory mode is selected in the MODE field of the `satp` register, supervisor virtual addresses are translated into supervisor physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12-bit page offset is untranslated. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures (Section 3.7), before being directly converted to machine-level physical addresses. If necessary, supervisor-level physical addresses are zero-extended to the number of physical address bits found in the implementation.

For example, consider an RV32 system supporting 34 bits of physical address. When the value of `satp`.MODE is Sv32, a 34-bit physical address is produced directly, and therefore no zero extension is needed. When the value of `satp`.MODE is Bare, the 32-bit virtual address is translated (unmodified) into a 32-bit physical address, and then that physical address is zero-extended into a 34-bit machine-level physical address.

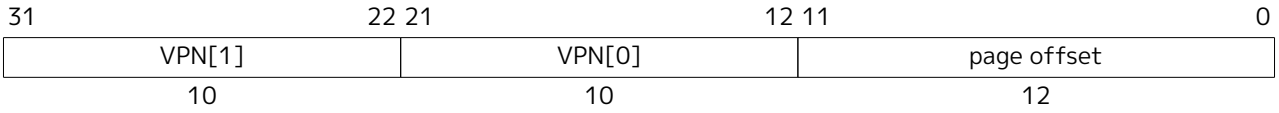


Figure 58. Sv32 virtual address.

Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register.

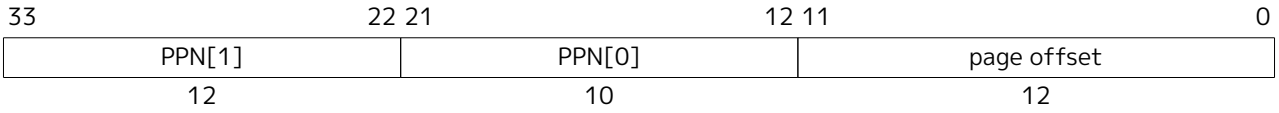


Figure 59. SV32 physical address.

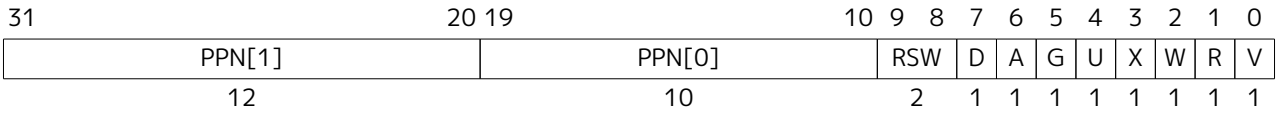


Figure 60. Sv32 page table entry.

The PTE format for Sv32 is shown in Figure 60. The V bit indicates whether the PTE is valid; if it is 0, all other bits in the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use. Table 25 summarizes the encoding of the permission bits.

Table 25. Encoding of PTE R/W/X fields.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load or load-reserved instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional, or AMO instruction whose effective address lies within a page without write permissions raises a store page-fault exception.

AMOs never raise load page-fault exceptions. Since any unreadable page is also unwritable, attempting to perform an AMO on an unreadable page always raises a store page-fault exception.

The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the SUM bit in the `sstatus` register is set, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the SUM bit clear, in which case, supervisor code will fault on accesses to user-mode pages. Irrespective of SUM, the supervisor may not execute code on pages with U=1.

An alternative PTE format would support different permissions for supervisor and user. We omitted this feature because it would be largely redundant with the SUM mechanism (see Section 11.1.1.2) and would require more encoding space in the PTE.

The G bit designates a *global* mapping. Global mappings are those that exist in all address spaces. For non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global. Note that failing to mark a global mapping as global merely reduces performance, whereas marking a non-global mapping as global is a software bug that, after switching to an address space with a different non-global mapping for that address range, can unpredictably result in either mapping being used.

Global mappings need not be stored redundantly in address-translation caches for multiple ASIDs. Additionally, they need not be flushed from local address-translation caches when an SFENCE.VMA instruction is executed with rs2≠`x0`.

The RSW field is reserved for use by supervisor software; the implementation shall ignore this field.

Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared.

Two schemes to manage the A and D bits are defined:

- The *Svade* extension: when a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- When the Svade extension is not implemented, the following scheme applies.

  When a virtual page is accessed and the A bit is clear, the PTE is updated to set the A bit. When the virtual page is written and the D bit is clear, the PTE is updated to set the D bit. When G-stage address translation is in use and is not Bare, the G-stage virtual pages may be accessed or written by implicit

accesses to VS-level memory management data structures, such as page tables.

When two-stage address translation is in use, an explicit access may cause both VS-stage and G-stage PTEs to be updated. The following rules apply to all PTE updates caused by an explicit or an implicit memory accesses.

The PTE update must be atomic with respect to other accesses to the PTE, and must atomically perform all tablewalk checks for that leaf PTE as part of, and before, conditionally updating the PTE value. Updates of the A bit may be performed as a result of speculation, even if the associated memory access ultimately is not performed architecturally. However, updates to the D bit, resulting from an explicit store, must be exact (i.e., non-speculative), and observed in program order by the local hart. When two-stage address translation is active, updates to the D bit in G-stage PTEs may be performed by an implicit access to a VS-stage PTE, if the G-stage PTE provides write permission, before any speculative access to the VS-stage PTE.

The PTE update must appear in the global memory order before the memory access that caused the PTE update and before any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

The PTE update is not required to be atomic with respect to the memory access that caused the update and a trap may occur between the PTE update and the memory access that caused the PTE update. If a trap occurs then the A and/or D bit may be updated but the memory access that caused the PTE update might not occur. The hart must not perform the memory access that caused the PTE update before the PTE update is globally visible.

The page tables must be located in memory with hardware page-table write access and *RsrvEventual* PMA.

All harts in a system must employ the same PTE-update scheme as each other.

The PTE updates due to memory accesses ordered-after a FENCE are not themselves ordered by the FENCE.

Simpler implementations may order the Page Table Entry (PTE) update to precede all subsequent explicit memory accesses, as opposed to ensuring that the PTE update is precisely sequenced before subsequent explicit memory accesses to the associated virtual page.

Prior versions of this specification required PTE A bit updates to be exact, but allowing the A bit to be updated as a result of speculation simplifies the implementation of address translation prefetchers. System software typically uses the A bit as a page replacement policy hint, but does not require exactness for functional correctness. On the other hand, D bit updates are still required to be exact and performed in program order, as the D bit affects the functional correctness of page eviction.

Implementations are of course still permitted to perform both A and D bit updates only in an exact manner.

In both cases, requiring atomicity ensures that the PTE update will not be interrupted by other intervening writes to the page table, as such interruptions could lead to A/D bits being set on PTEs that have been reused for other purposes, on memory that has been reclaimed for other purposes, and so on. Simple implementations may instead generate page-fault exceptions.

The A and D bits are never cleared by the implementation. If the supervisor software does not

> *rely on accessed and/or dirty bits, e.g. if it does not swap memory pages to secondary storage or if the pages are being used to map I/O space, it should always set them to 1 in the PTE to improve performance.*

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv32 supports 4 MiB *megapages*. A megapage must be virtually and physically aligned to a 4 MiB boundary; a page-fault exception is raised if the physical address is insufficiently aligned.

For non-leaf PTEs, the D, A, and U bits are reserved for future standard use. Until their use is defined by a standard extension, they must be cleared by software for forward compatibility.

For implementations with both page-based virtual memory and the "A" standard extension, the LR/SC reservation set must lie completely within a single base physical page (i.e., a naturally aligned 4 KiB physical-memory region).

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before a page-fault exception occurs. In particular, a portion of a misaligned store that passes the exception check may become visible, even if another portion fails the exception check. The same behavior may manifest for stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

11.3.2. Virtual Address Translation Process

A virtual address *va* is translated into a physical address *pa* as follows:

1. Let a be `satp`.$ppn \times$PAGESIZE, and let i=LEVELS-1. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.) The `satp` register must be *active*, i.e., the effective privilege mode must be S-mode or U-mode.
2. Let *pte* be the value of the PTE at address $a+va.vpn[i] \times$PTESIZE. (For Sv32, PTESIZE=4.) If accessing *pte* violates a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
3. If $pte.v$=0, or if $pte.r$=0 and $pte.w$=1, or if any bits or encodings that are reserved for future standard use are set within *pte*, stop and raise a page-fault exception corresponding to the original access type.
4. Otherwise, the PTE is valid. If $pte.r$=1 or $pte.x$=1, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let i=i-1. If i<0, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let a=$pte.ppn \times$PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the *pte.r*, *pte.w*, *pte.x*, and *pte.u* bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception corresponding to the original access type.
6. If $i>0$ and $pte.ppn[i\text{-}1\text{:}0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
7. If $pte.a$=0, or if the original memory access is a store and $pte.d$=0:
   - If the Svade extension is implemented, stop and raise a page-fault exception corresponding to the original access type.
   - If a store to *pte* would violate a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
   - Perform the following steps atomically:
     - Compare *pte* to the value of the PTE at address $a+va.vpn[i] \times$PTESIZE.
     - If the values match, set $pte.a$ to 1 and, if the original memory access is a store, also set $pte.d$ to 1.
     - If the comparison fails, return to step 2.

8. The translation is successful. The translated physical address is given as follows:
   - *pa.pgoff* = *va.pgoff*.
   - If *i*>0, then this is a superpage translation and *pa.ppn*[*i*-1:0] = *va.vpn*[*i*-1:0].
   - *pa.ppn*[LEVELS-1:*i*] = *pte.ppn*[LEVELS-1:*i*].

All implicit accesses to the address-translation data structures in this algorithm are performed using width PTESIZE.

This implies, for example, that an Sv48 implementation may not use two separate 4B reads to non-atomically access a single 8B PTE, and that A/D bit updates performed by the implementation are treated as atomically updating the entire PTE, rather than just the A and/or D bit alone (even though the PTE value does not otherwise change).

The results of implicit address-translation reads in step 2 may be held in a read-only, incoherent *address-translation cache* but not shared with other harts. The address-translation cache may hold an arbitrary number of entries, including an arbitrary number of entries for the same address and ASID. Entries in the address-translation cache may then satisfy subsequent step 2 reads if the ASID associated with the entry matches the ASID loaded in step 0 or if the entry is associated with a *global* mapping. To ensure that implicit reads observe writes to the same memory locations, an SFENCE.VMA instruction must be executed after the writes to flush the relevant cached translations.

The address-translation cache cannot be used in step 7; accessed and dirty bits may only be updated in memory directly.

It is permitted for multiple address-translation cache entries to co-exist for the same address. This represents the fact that in a conventional TLB hierarchy, 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`, or if multiple TLBs exist in parallel at a given level of the hierarchy. In this case, just as if an SFENCE.VMA is not executed between a write to the memory-management tables and subsequent implicit read of the same address: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.

Implementations may also execute the address-translation algorithm speculatively at any time, for any virtual address, as long as `satp` is active (as defined in Section 11.1.11). Such speculative executions have the effect of pre-populating the address-translation cache.

Speculative executions of the address-translation algorithm behave as non-speculative executions of the algorithm do, except that they must not set the dirty bit for a PTE, they must not trigger an exception, and they must not create address-translation cache entries if those entries would have been invalidated by any SFENCE.VMA instruction executed by the hart since the speculative execution of the algorithm began.

For instance, it is illegal for both non-speculative and speculative executions of the translation algorithm to begin, read the level 2 page table, pause while the hart executes an SFENCE.VMA with rs1=rs2=`x0`, then resume using the now-stale level 2 PTE, as subsequent implicit reads could populate the address-translation cache with stale PTEs.

In many implementations, an SFENCE.VMA instruction with rs1=`x0` will therefore either terminate all previously-launched speculative executions of the address-translation algorithm (for the specified ASID, if applicable), or simply wait for them to complete (in which case any address-translation cache entries created will be invalidated by the SFENCE.VMA as appropriate). Likewise, an SFENCE.VMA instruction with rs1≠`x0` generally must either ensure that previously-launched speculative executions of the address-translation algorithm (for the specified ASID, if applicable) are prevented from creating new address-translation cache

entries mapping leaf PTEs, or wait for them to complete.

A consequence of implementations being permitted to read the translation data structures arbitrarily early and speculatively is that at any time, all page table entries reachable by executing the algorithm may be loaded into the address-translation cache.

Although it would be uncommon to place page tables in non-idempotent memory, there is no explicit prohibition against doing so. Since the algorithm may only touch page tables reachable from the root page table indicated in `satp`, the range of addresses that an implementation's page table walker will touch is fully under supervisor control.

The algorithm does not admit the possibility of ignoring high-order PPN bits for implementations with narrower physical addresses.

11.4. Sv39: Page-Based 39-bit Virtual-Memory System

This section describes a simple paged virtual-memory system for SXLEN=64, which supports 39-bit virtual address spaces. The design of Sv39 follows the overall scheme of Sv32, and this section details only the differences between the schemes.

We specified multiple virtual memory systems for RV64 to relieve the tension between providing a large address space and minimizing address-translation cost. For many systems, 39 bits of virtual-address space is ample, and so Sv39 suffices. Sv48 increases the virtual address space to 48 bits, but increases the physical memory capacity dedicated to page tables, the latency of page-table traversals, and the size of hardware structures that store virtual addresses. Sv57 increases the virtual address space, page table capacity requirement, and translation latency even further.

11.4.1. Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into pages. An Sv39 address is partitioned as shown in Figure 61. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else a page-fault exception will occur. The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

When mapping between narrower and wider addresses, RISC-V zero-extends a narrower physical address to a wider size. The mapping between 64-bit virtual addresses and the 39-bit usable address space of Sv39 is not based on zero extension but instead follows an entrenched convention that allows an OS to use one or a few of the most-significant bits of a full-size (64-bit) virtual address to quickly distinguish user and supervisor address regions.

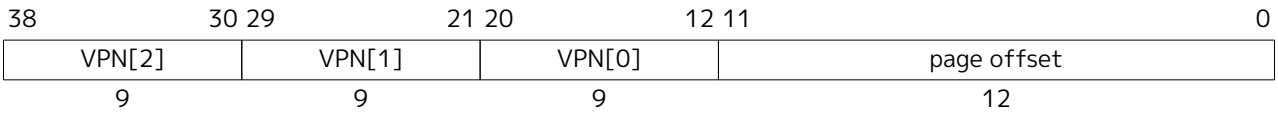


Figure 61. Sv39 virtual address.

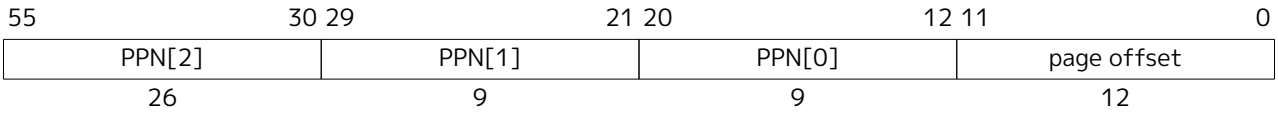


Figure 62. Sv39 physical address.

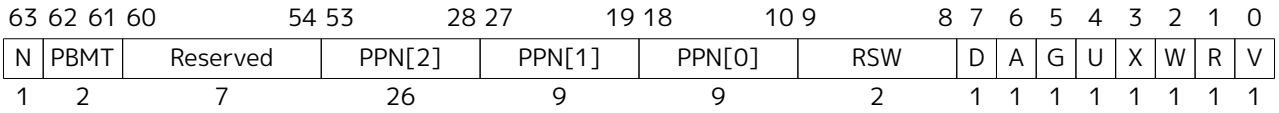


Figure 63. Sv39 page table entry.

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register's PPN field.

The PTE format for Sv39 is shown in Figure 63. Bits 9-0 have the same meaning as for Sv32. Bit 63 is reserved for use by the Svnapot extension in Chapter 12. If Svnapot is not implemented, bit 63 remains reserved and must be zeroed by software for forward compatibility, or else a page-fault exception is raised. Bits 62-61 are reserved for use by the Svpbmt extension in Chapter 13. If Svpbmt is not implemented, bits 62-61 remain reserved and must be zeroed by software for forward compatibility, or else a page-fault exception is raised. Bits 60-54 are reserved for future standard use and, until their use is defined by some standard extension, must be zeroed by software for forward compatibility. If any of these bits are set, a page-fault exception is raised.

We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency. These reserved bits may also be used to facilitate research experimentation. The cost is reducing the physical address space, but is presently ample. When it no longer suffices, the reserved bits that remain unallocated could be used to expand the physical address space.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB *megapages* and 1 GiB *gigapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 11.3.2, except LEVELS equals 3 and PTESIZE equals 8.

11.5. Sv48: Page-Based 48-bit Virtual-Memory System

This section describes a simple paged virtual-memory system for SXLEN=64, which supports 48-bit virtual address spaces. Sv48 is intended for systems for which a 39-bit virtual address space is insufficient. It closely follows the design of Sv39, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv48 must also support Sv39.

Systems that support Sv48 can also support Sv39 at essentially no cost, and so should do so to maintain compatibility with supervisor software that assumes Sv39.

11.5.1. Addressing and Memory Protection

Sv48 implementations support a 48-bit virtual address space, divided into pages. An Sv48 address is partitioned as shown in Figure 64. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else a page-fault exception will occur. The 36-bit VPN is translated into a 44-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

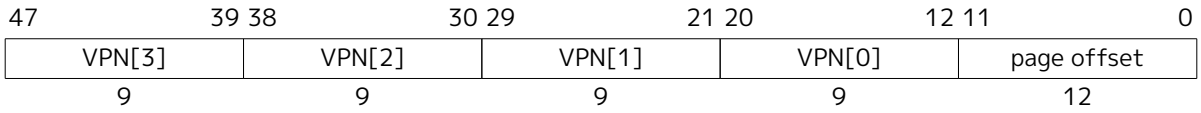


Figure 64. Sv48 virtual address.

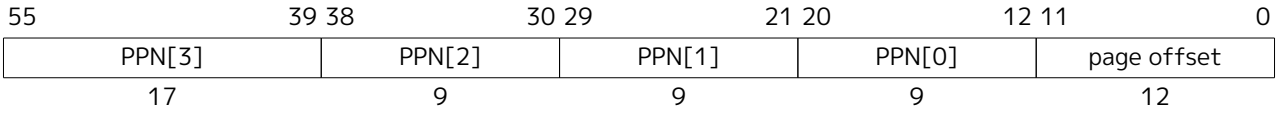


Figure 65. Sv48 physical address.

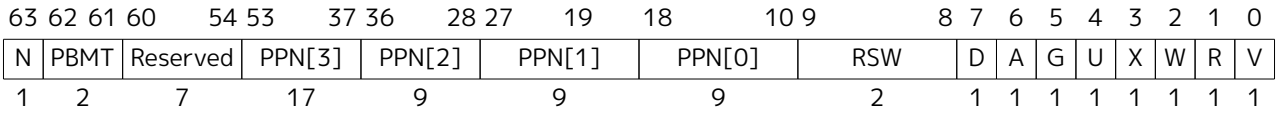


Figure 66. Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 66. Bits 63-54 and 9-0 have the same meaning as for Sv39. Any level of PTE may be a leaf PTE, so in addition to pages, Sv48 supports *megapages*, *gigapages*, and *terapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 11.3.2, except LEVELS equals 4 and PTESIZE equals 8.

11.6. Sv57: Page-Based 57-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, which supports 57-bit virtual address spaces. Sv57 is intended for systems for which a 48-bit virtual address space is insufficient. It closely follows the design of Sv48, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv57 must also support Sv48.

Systems that support Sv57 can also support Sv48 at essentially no cost, and so should do so to maintain compatibility with supervisor software that assumes Sv48.

11.6.1. Addressing and Memory Protection

Sv57 implementations support a 57-bit virtual address space, divided into pages. An Sv57 address is partitioned as shown in Figure 67. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–57 all equal to bit 56, or else a page-fault exception will occur. The 45-bit VPN is translated into a 44-bit PPN via a five-level page table, while the 12-bit page offset is untranslated.

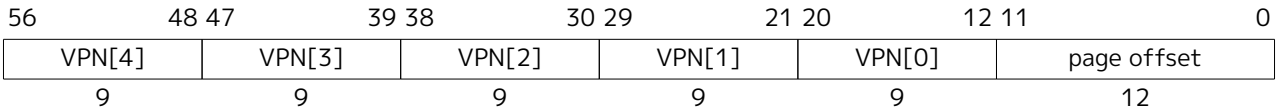


Figure 67. Sv57 virtual address.

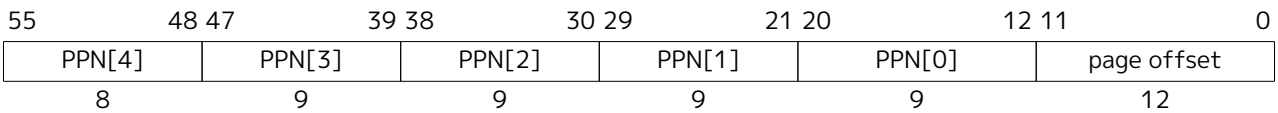


Figure 68. Sv57 physical address.

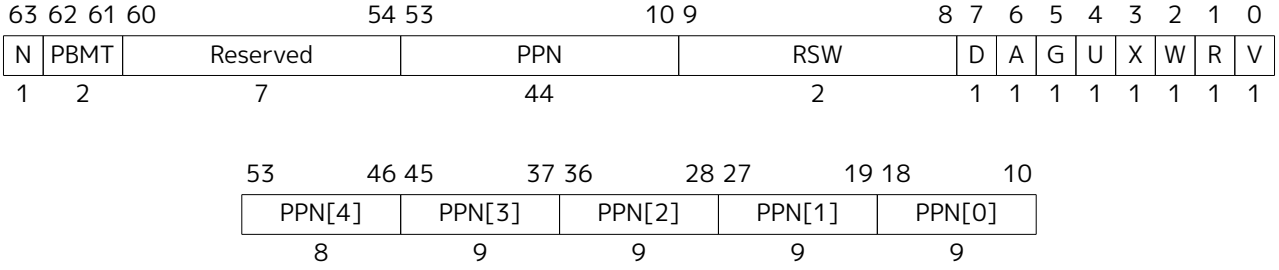


Figure 69. Sv57 page table entry.

The PTE format for Sv57 is shown in Figure 69. Bits 63–54 and 9–0 have the same meaning as for Sv39. Any level of PTE may be a leaf PTE, so in addition to pages, Sv57 supports *megapages*, *gigapages*, *terapages*, and *petapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 11.3.2, except LEVELS equals 5 and PTESIZE equals 8.

Chapter 12. "Svnapot" Extension for NAPOT Translation Contiguity, Version 1.0

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 26. 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 Section 11.3.2, except that:

- If the encoding in *pte* is valid according to Table 26, 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 26, 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=`x0`, or with multiple SFENCE.VMA instructions with rs1≠`x0`), then update the PTE(s), as described in Section 11.2.1, 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 Section 11.3.2 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.

Chapter 13. "Svpbmt" Extension for Page-Based Memory Types, Version 1.0

In Sv39, Sv48, and Sv57, bits 62-61 of a leaf page table entry indicate the use of page-based memory types that override the PMA(s) for the associated memory pages. The encoding for the PBMT bits is captured in Table 27.

The Svpbmt extension depends on Sv39.

Table 27. Encodings for PBMT field in Sv39, Sv48, and Sv57 PTEs.

Mode	Value	Requested Memory Attributes
PMA	0	None
NC	1	Non-cacheable, idempotent, weakly-ordered (RVWMO), main memory
IO	2	Non-cacheable, non-idempotent, strongly-ordered (I/O ordering), I/O
-	3	*Reserved for future standard use*

Implementations may override additional PMAs not explicitly listed in Table 27. For example, to be consistent with the characteristics of a typical I/O region, a misaligned memory access to a page with PBMT=IO might raise an exception, even if the underlying region were main memory and the same access would have succeeded for PBMT=PMA.

Future extensions may provide more and/or finer-grained control over which PMAs can be overridden.

For non-leaf PTEs, bits 62-61 are reserved for future standard use. Until their use is defined by a standard extension, they must be cleared by software for forward compatibility, or else a page-fault exception is raised.

For leaf PTEs, setting bits 62-61 to the value 3 is reserved for future standard use. Until this value is defined by a standard extension, using this reserved value in a leaf PTE raises a page-fault exception.

When PBMT settings override a main memory page into I/O or vice versa, memory accesses to such pages obey the memory ordering rules of the final effective attribute, as follows.

If the underlying physical memory attribute for a page is I/O, and the page has PBMT=NC, then accesses to that page obey RVWMO. However, accesses to such pages are considered to be *both* I/O and main memory accesses for the purposes of FENCE, *.aq*, and *.rl*.

If the underlying physical memory attribute for a page is main memory, and the page has PBMT=IO, then accesses to that page obey strong channel 0 I/O ordering rules. However, accesses to such pages are considered to be *both* I/O and main memory accesses for the purposes of FENCE, *.aq*, and *.rl*.

A device driver written to rely on I/O strong ordering rules will not operate correctly if the address range is mapped with PBMT=NC. As such, this configuration is discouraged.

It will often still be useful to map physical I/O regions using PBMT=NC so that write combining and speculative accesses can be performed. Such optimizations will likely improve performance when applied with adequate care.

When Svpbmt is used with non-zero PBMT encodings, it is possible for multiple virtual aliases of the same physical page to exist simultaneously with different memory attributes. It is also possible for a U-mode or S-mode mapping through a PTE with Svpbmt enabled to observe different memory attributes for a given region of physical memory than a concurrent access to the same page performed by M-mode or when MODE=Bare. In such cases, the behaviors dictated by the attributes (including coherence, which is

otherwise unaffected) may be violated.

Accessing the same location using different attributes that are both non-cacheable (e.g., NC and IO) does not cause loss of coherence, but might result in weaker memory ordering than the stricter attribute ordinarily guarantees. Executing a `fence iorw, iorw` instruction between such accesses suffices to prevent loss of memory ordering.

Accessing the same location using different cacheability attributes may cause loss of coherence. Executing the following sequence between such accesses prevents both loss of coherence and loss of memory ordering: `fence iorw, iorw`, followed by `cbo.flush` to an address of that location, followed by a `fence iorw, iorw`.

It follows that, if the same location might later be referenced using the original attributes, then this sequence must be repeated beforehand.

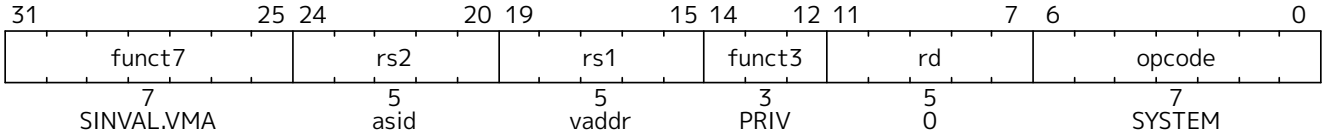
In certain cases, a weaker sequence might suffice to prevent loss of coherence. These situations will be detailed following the forthcoming formalization of the interaction of the RVWMO memory model with the instructions in the Zicbom extension.

When two-stage address translation is enabled within the H extension, the page-based memory types are also applied in two stages. First, if `hgatp`.MODE is not equal to zero, non-zero G-stage PTE PBMT bits override the attributes in the PMA to produce an intermediate set of attributes. Otherwise, the PMAs serve as the intermediate attributes. Second, if `vsatp`.MODE is not equal to zero, non-zero VS-stage PTE PBMT bits override the intermediate attributes to produce the final set of attributes used by accesses to the page in question. Otherwise, the intermediate attributes are used as the final set of attributes.

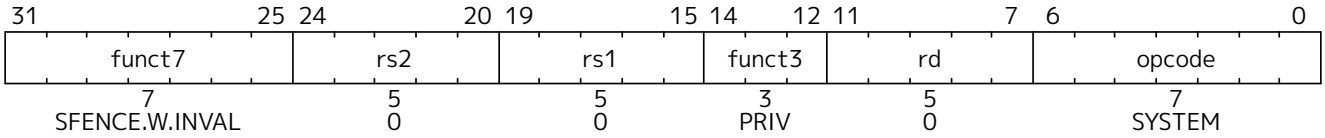
These final attributes apply to implicit and explicit accesses that are subject to both stages of address translation. For accesses that are not subject to the first stage of address translation, e.g. VS-stage page-table accesses, the intermediate attributes apply instead.

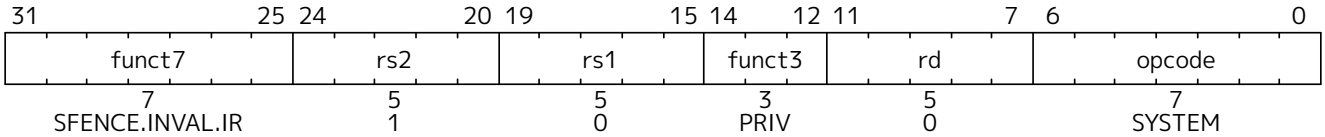
Chapter 14. "Svinval" Extension for Fine-Grained Address-Translation Cache Invalidation, Version 1.0

The Svinval extension splits SFENCE.VMA, HFENCE.VVMA, and HFENCE.GVMA instructions into finer-grained invalidation and ordering operations that can be more efficiently batched or pipelined on certain classes of high-performance implementation.



The SINVAL.VMA instruction invalidates any address-translation cache entries that an SFENCE.VMA instruction with the same values of *rs1* and *rs2* would invalidate. However, unlike SFENCE.VMA, SINVAL.VMA instructions are only ordered with respect to SFENCE.VMA, SFENCE.W.INVAL, and SFENCE.INVAL.IR instructions as defined below.

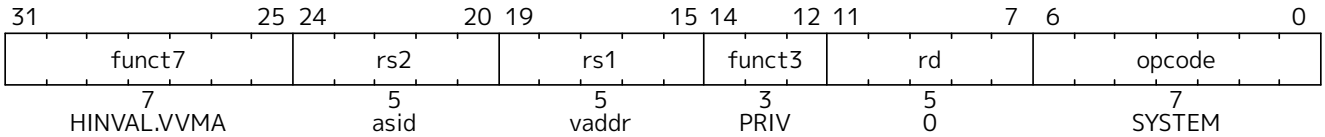


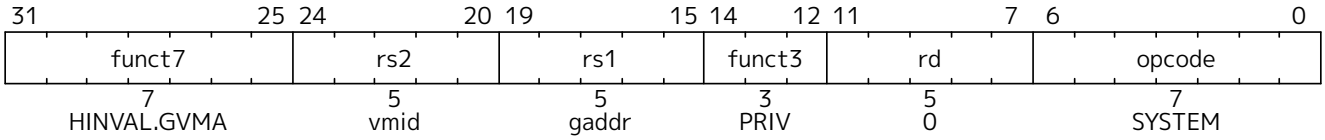


The SFENCE.W.INVAL instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before subsequent SINVAL.VMA instructions executed by the same hart. The SFENCE.INVAL.IR instruction guarantees that any previous SINVAL.VMA instructions executed by the current hart are ordered before subsequent implicit references by that hart to the memory-management data structures.

When executed in order (but not necessarily consecutively) by a single hart, the sequence SFENCE.W.INVAL, SINVAL.VMA, and SFENCE.INVAL.IR has the same effect as a hypothetical SFENCE.VMA instruction in which:

- the values of *rs1* and *rs2* for the SFENCE.VMA are the same as those used in the SINVAL.VMA,
- reads and writes prior to the SFENCE.W.INVAL are considered to be those prior to the SFENCE.VMA, and
- reads and writes following the SFENCE.INVAL.IR are considered to be those subsequent to the SFENCE.VMA.





If the hypervisor extension is implemented, the Svinval extension also provides two additional instructions: HINVAL.VVMA and HINVAL.GVMA. These have the same semantics as SINVAL.VMA, except

that they combine with SFENCE.W.INVAL and SFENCE.INVAL.IR to replace HFENCE.VVMA and HFENCE.GVMA, respectively, instead of SFENCE.VMA. In addition, HINVAL.GVMA uses VMIDs instead of ASIDs.

SINVAL.VMA, HINVAL.VVMA, and HINVAL.GVMA require the same permissions and raise the same exceptions as SFENCE.VMA, HFENCE.VVMA, and HFENCE.GVMA, respectively. In particular, an attempt to execute any of these instructions in U-mode always raises an illegal-instruction exception, and an attempt to execute SINVAL.VMA or HINVAL.GVMA in S-mode or HS-mode when `mstatus`.TVM=1 also raises an illegal-instruction exception. An attempt to execute HINVAL.VVMA or HINVAL.GVMA in VS-mode or VU-mode, or to execute SINVAL.VMA in VU-mode, raises a virtual-instruction exception. When `hstatus`.VTVM=1, an attempt to execute SINVAL.VMA in VS-mode also raises a virtual instruction exception.

Attempting to execute SFENCE.W.INVAL or SFENCE.INVAL.IR in U-mode raises an illegal-instruction exception. Doing so in VU-mode raises a virtual-instruction exception. SFENCE.W.INVAL and SFENCE.INVAL.IR are unaffected by the `mstatus`.TVM and `hstatus`.VTVM fields and hence are always permitted in S-mode and VS-mode.

SFENCE.W.INVAL and SFENCE.INVAL.IR instructions do not need to be trapped when `mstatus`*.TVM=1 or when* `hstatus`*.VTVM=1, as they only have ordering effects but no visible side effects. Trapping of the SINVAL.VMA instruction is sufficient to enable emulation of the intended overall TLB maintenance functionality.*

In typical usage, software will invalidate a range of virtual addresses in the address-translation caches by executing an SFENCE.W.INVAL instruction, executing a series of SINVAL.VMA, HINVAL.VVMA, or HINVAL.GVMA instructions to the addresses (and optionally ASIDs or VMIDs) in question, and then executing an SFENCE.INVAL.IR instruction.

High-performance implementations will be able to pipeline the address-translation cache invalidation operations, and will defer any pipeline stalls or other memory ordering enforcement until an SFENCE.W.INVAL, SFENCE.INVAL.IR, SFENCE.VMA, HFENCE.GVMA, or HFENCE.VVMA instruction is executed.

Simpler implementations may implement SINVAL.VMA, HINVAL.VVMA, and HINVAL.GVMA identically to SFENCE.VMA, HFENCE.VVMA, and HFENCE.GVMA, respectively, while implementing SFENCE.W.INVAL and SFENCE.INVAL.IR instructions as no-ops.

Chapter 15. "Svadu" Extension for Hardware Updating of A/D Bits, Version 1.0

The Svadu extension adds support and CSR controls for hardware updating of PTE A/D bits.

If the Svadu extension is implemented, the `menvcfg`.ADUE field is writable. If the hypervisor extension is additionally implemented, the `henvcfg`.ADUE field is also writable. See Section 3.1.18 and Section 19.2.5 for the definitions of those fields.

Section 11.3.1 defines the semantics of hardware updating of A/D bits. When hardware updating of A/D bits is disabled, the Svade extension, which mandates exceptions when A/D bits need be set, instead takes effect. The Svade extension is also defined in Section 11.3.1.

Chapter 16. "Svvptc" Extension for Obviating Memory-Management Instructions after Marking PTEs Valid, Version 1.0

When the Svvptc extension is implemented, explicit stores by a hart that update the Valid bit of leaf and/or non-leaf PTEs from 0 to 1 and are visible to a hart will eventually become visible within a bounded timeframe to subsequent implicit accesses by that hart to such PTEs.

Svvptc relieves an operating system from executing certain memory-management instructions, such as `SFENCE.VMA` *or* `SINVAL.VMA`*, which would normally be used to synchronize the hart's address-translation caches when a memory-resident PTE is changed from Invalid to Valid. Synchronizing the hart's address-translation caches with other forms of updates to a memory-resident PTE, including when a PTE is changed from Valid to Invalid, requires the use of suitable memory-management instructions. Svvptc guarantees that a change to a PTE from Invalid to Valid is made visible within a bounded time, thereby making the execution of these memory-management instructions redundant. The performance benefit of eliding these instructions outweighs the cost of an occasional gratuitous additional page fault that may occur.*

Depending on the microarchitecture, some possible ways to facilitate implementation of Svvptc include: not having any address-translation caches, not storing Invalid PTEs in the address-translation caches, automatically evicting Invalid PTEs using a bounded timer, or making address-translation caches coherent with store instructions that modify PTEs.

Chapter 17. "Sstc" Extension for Supervisor-mode Timer Interrupts, Version 1.0

The current Privileged arch specification only defines a hardware mechanism for generating machine-mode timer interrupts (based on the mtime and mtimecmp registers). With the resultant requirement that timer services for S-mode/HS-mode (and for VS-mode) have to all be provided by M-mode - via SBI calls from S/HS-mode up to M-mode (or VS-mode calls to HS-mode and then to M-mode). M-mode software then multiplexes these multiple logical timers onto its one physical M-mode timer facility, and the M-mode timer interrupt handler passes timer interrupts back down to the appropriate lower privilege mode.

This extension serves to provide supervisor mode with its own CSR-based timer interrupt facility that it can directly manage to provide its own timer service (in the form of having its own stimecmp register) - thus eliminating the large overheads for emulating S/HS-mode timers and timer interrupt generation up in M-mode. Further, this extension adds a similar facility to the Hypervisor extension for VS-mode.

To make it easy to understand the deltas from the current Priv 1.11/1.12 specs, this is written as the actual exact changes to be made to existing paragraphs of Priv spec text (or additional paragraphs within the existing text).

The extension name is "Sstc" ('Ss' for Privileged arch and Supervisor-level extensions, and 'tc' for timecmp). This extension adds the S-level stimecmp CSR and the VS-level vstimecmp CSR.

17.1. Machine and Supervisor Level Additions

17.1.1. Supervisor Timer (`stimecmp`) Register

This extension adds this new CSR.

The `stimecmp` CSR is a 64-bit register and has 64-bit precision on all RV32 and RV64 systems. In RV32 only, accesses to the `stimecmp` CSR access the low 32 bits, while accesses to the `stimecmph` CSR access the high 32 bits of `stimecmp`.

The CSR numbers for `stimecmp` / `stimecmph` are 0x14D / 0x15D (within the Supervisor Trap Setup block of CSRs).

A supervisor timer interrupt becomes pending, as reflected in the STIP bit in the `mip` and `sip` registers whenever `time` contains a value greater than or equal to `stimecmp`, treating the values as unsigned integers. If the result of this comparison changes, it is guaranteed to be reflected in STIP eventually, but not necessarily immediately. The interrupt remains posted until `stimecmp` becomes greater than `time`, typically as a result of writing `stimecmp`. The interrupt will be taken based on the standard interrupt enable and delegation rules.

A spurious timer interrupt might occur if an interrupt handler advances `stimecmp` then immediately returns, because STIP might not yet have fallen in the interim. All software should be written to assume this event is possible, but most software should assume this event is extremely unlikely. It is almost always more performant to incur an occasional spurious timer interrupt than to poll STIP until it falls.

In systems in which a supervisor execution environment (SEE) provides timer facilities via an SBI function call, this SBI call will continue to support requests to schedule a timer interrupt. The SEE will simply make use of stimecmp, changing its value as appropriate. This ensures compatibility with existing S-mode software that uses this SEE facility, while new S-mode software takes advantage of stimecmp directly.)

17.1.2. Machine Interrupt (`mip` and `mie`) Registers

This extension modifies the description of the STIP/STIE bits in these registers as follows:

If supervisor mode is implemented, its mip.STIP and mie.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If the stimecmp register is not implemented, STIP is writable in mip, and may be written by M-mode software to deliver timer interrupts to S-mode. If the `stimecmp` (supervisor-mode timer compare) register is implemented, STIP is read-only in mip and reflects the supervisor-level timer interrupt signal resulting from stimecmp. This timer interrupt signal is cleared by writing `stimecmp` with a value greater than the current time value.

17.1.3. Supervisor Interrupt (`sip` and `sie`) Registers

This extension modifies the description of the STIP/STIE bits in these registers as follows:

Bits `sip`.STIP and `sie`.STIE are the interrupt-pending and interrupt-enable bits for supervisor level timer interrupts. If implemented, STIP is read-only in sip, and is either set and cleared by the execution environment (if `stimecmp` is not implemented), or reflects the timer interrupt signal resulting from `stimecmp` (if `stimecmp` is implemented). The `sip`.STIP bit, in response to timer interrupts generated by `stimecmp`, is set and cleared by writing `stimecmp` with a value that respectively is less than or equal to, or greater than, the current time value.

17.1.4. Machine Counter-Enable (`mcounteren`) Register

This extension adds to the description of the TM bit in this register as follows:

In addition, when the TM bit in the mcounteren register is clear, attempts to access the `stimecmp` or `vstimecmp` register while executing in a mode less privileged than M will cause an illegal instruction exception. When this bit is set, access to the `stimecmp` or `vstimecmp` register is permitted in S-mode if implemented, and access to the `vstimecmp` register (via `stimecmp`) is permitted in VS-mode if implemented and not otherwise prevented by the TM bit in `hcounteren`.

17.2. Hypervisor Extension Additions

17.2.1. Virtual Supervisor Timer (`vstimecmp`) Register

This extension adds this new CSR.

The `vstimecmp` CSR is a 64-bit register and has 64-bit precision on all RV32 and RV64 systems. In RV32 only, accesses to the `vstimecmp` CSR access the low 32 bits, while accesses to the `vstimecmph` CSR access the high 32 bits of vstimecmp.

The proposed CSR numbers for `vstimecmp` / `vstimecmph` are 0x24D / 0x25D (within the Virtual Supervisor Registers block of CSRs, and mirroring the CSR numbers for stimecmp/stimecmph).

A virtual supervisor timer interrupt becomes pending, as reflected in the VSTIP bit in the `hip` register, whenever (`time` + `htimedelta`), truncated to 64 bits, contains a value greater than or equal to `vstimecmp`, treating the values as unsigned integers. If the result of this comparison changes, it is guaranteed to be reflected in VSTIP eventually, but not necessarily immediately. The interrupt remains posted until `vstimecmp` becomes greater than (`time` + `htimedelta`), typically as a result of writing `vstimecmp`. The interrupt will be taken based on the standard interrupt enable and delegation rules while V=1.

In systems in which a supervisor execution environment (SEE) implemented by an HS-mode hypervisor provides timer facilities via an SBI function call, this SBI call will continue to support requests to schedule a timer interrupt. The SEE will simply make use of vstimecmp, changing its value as appropriate. This ensures compatibility with existing guest VS-mode software that uses this SEE facility, while new VS-mode software takes advantage of vstimecmp directly.)

17.2.2. Hypervisor Interrupt (hvip, hip, and hie) Registers

This extension modifies the description of the VSTIP/VSTIE bits in the hip/hie registers as follows:

Bits hip.VSTIP and hie.VSTIE are the interrupt-pending and interrupt-enable bits for VS-level timer interrupts. VSTIP is read-only in hip, and is the logical-OR of hvip.VSTIP and the timer interrupt signal resulting from `vstimecmp` (if `vstimecmp` is implemented). The `hip`.VSTIP bit, in response to timer interrupts generated by `vstimecmp`, is set and cleared by writing `vstimecmp` with a value that respectively is less than or equal to, or greater than, the current (`time` + `htimedelta`) value. The `hip`.VSTIP bit remains defined while V=0 as well as V=1.

17.2.3. Hypervisor Counter-Enable (hcounteren) Register

This extension adds to the description of the TM bit in this register as follows:

In addition, when the TM bit in the `hcounteren` register is clear, attempts to access the `vstimecmp` register (via stimecmp) while executing in VS-mode will cause a virtual instruction exception if the same bit in `mcounteren` is set. When this bit and the same bit in `mcounteren` are both set, access to the `vstimecmp` register (if implemented) is permitted in VS-mode.

17.3. Environment Config (menvcfg and henvcfg) Support

Enable/disable bits for this extension are provided in the new `menvcfg` / `henvcfg` CSRs.

Bit 63 of `menvcfg` (or bit 31 of `menvcfgh`) - named STCE (STimecmp Enable) - enables `stimecmp` for S-mode when set to one, and the same bit of henvcfg enables `vstimecmp` for VS-mode. These STCE bits are WARL and are hard-wired to 0 when this extension is not implemented.

When this extension is implemented and STCE in `menvcfg` is zero, an attempt to access `stimecmp` or `vstimecmp` in a mode other than M-mode raises an illegal instruction exception, STCE in `henvcfg` is read-only zero, and STIP in `mip` and `sip` reverts to its defined behavior as if this extension is not implemented. Further, if the H extension is implemented, then hip.VSTIP also reverts its defined behavior as if this extension is not implemented.

But when STCE in `menvcfg` is one and STCE in `henvcfg` is zero, an attempt to access `stimecmp` (really `vstimecmp`) when V = 1 raises a virtual instruction exception, and VSTIP in hip reverts to its defined behavior as if this extension is not implemented.

Chapter 18. "Sscofpmf" Extension for Count Overflow and Mode-Based Filtering, Version 1.0

The current Privileged specification defines mhpmevent CSRs to select and control event counting by the associated hpmcounter CSRs, but provides no standardization of any fields within these CSRs. For at least Linux-class rich-OS systems it is desirable to standardize certain basic features that are broadly desired (and have come up over the past year plus on RISC-V lists, as well as have been the subject of past proposals). This enables there to be standard upstream software support that eliminates the need for implementations to provide their own custom software support.

This extension serves to accomplish exactly this within the existing mhpmevent CSRs (and correspondingly avoids the unnecessary creation of whole new sets of CSRs - past just one new CSR).

This extension sticks to addressing two basic well-understood needs that have been requested by various people. To make it easy to understand the deltas from the current Priv 1.11/1.12 specs, this is written as the actual exact changes to be made to existing paragraphs of Priv spec text (or additional paragraphs within the existing text).

The extension name is "Sscofpmf" ('Ss' for Privileged arch and Supervisor-level extensions, and 'cofpmf' for Count OverFlow and Privilege Mode Filtering).

Note that the new count overflow interrupt will be treated as a standard local interrupt that is assigned to bit 13 in the mip/mie/sip/sie registers.

18.1. Count Overflow Control

The following bits are added to `mhpmevent`:

63	62	61	60	59	58	57	56
OF	MINH	SINH	UINH	VSINH	VUINH	*WPRI*	*WPRI*

Field	Description
OF	Overflow status and interrupt disable bit that is set when counter overflows
MINH	If set, then counting of events in M-mode is inhibited
SINH	If set, then counting of events in S/HS-mode is inhibited
UINH	If set, then counting of events in U-mode is inhibited
VSINH	If set, then counting of events in VS-mode is inhibited
VUINH	If set, then counting of events in VU-mode is inhibited
WPRI	Reserved
WPRI	Reserved

For each `x`INH bit, if the associated privilege mode is not implemented, the bit is read-only zero.

Each of the five `x`INH bits, when set, inhibit counting of events while in privilege mode `x`. All-zeroes for these bits results in counting of events in all modes.

The OF bit is set when the corresponding hpmcounter overflows, and remains set until written by software. Since hpmcounter values are unsigned values, overflow is defined as unsigned overflow of the implemented counter bits. Note that there is no loss of information after an overflow since the counter wraps around and keeps counting while the sticky OF bit remains set.

If supervisor mode is implemented, the 32-bit scountovf register contains read-only shadow copies of the OF bits in all 32 mhpmevent registers.

If an hpmcounter overflows while the associated OF bit is zero, then a "count overflow interrupt request" is generated. If the OF bit is one, then no interrupt request is generated. Consequently the OF bit also functions as a count overflow interrupt disable for the associated hpmcounter.

Count overflow never results from writes to the mhpmcounter*n* or mhpmevent*n* registers, only from hardware increments of counter registers.

This count-overflow-interrupt-request signal is treated as a standard local interrupt that corresponds to bit 13 in the `mip`/`mie`/`sip`/`sie` registers. The `mip`/`sip` LCOFIP and `mie`/`sie` LCOFIE bits are, respectively, the interrupt-pending and interrupt-enable bits for this interrupt. ('LCOFI' represents 'Local Count Overflow Interrupt'.)

Generation of a count-overflow-interrupt request by an `hpmcounter` sets the associated OF bit. When an OF bit is set, it eventually, but not necessarily immediately, sets the LCOFIP bit in the `mip`/`sip` registers. The LCOFIP bit is cleared by software before servicing the count overflow interrupt resulting from one or more count overflows. The `mideleg` register controls the delegation of this interrupt to S-mode versus M-mode.

There are not separate overflow status and overflow interrupt enable bits. In practice, enabling overflow interrupt generation (by clearing the OF bit) is done in conjunction with initializing the counter to a starting value. Once a counter has overflowed, it and the OF bit must be reinitialized before another overflow interrupt can be generated.

Software can distinguish newly overflowed counters (yet to be serviced by an overflow interrupt handler) from overflowed counters that have already been serviced or that are configured to not generate an interrupt on overflow, by maintaining a bit mask reflecting which counters are active and due to eventually overflow.

18.2. Supervisor Count Overflow (`scountovf`) Register

This extension adds the `scountovf` CSR, a 32-bit read-only register that contains shadow copies of the OF bits in the 29 mhpmevent CSRs (mhpmevent*3* - mhpmevent*31*) - where scountovf bit *X* corresponds to mhpmevent*X*.

This register enables supervisor-level overflow interrupt handler software to quickly and easily determine which counter(s) have overflowed (without needing to make an execution environment call or series of calls ultimately up to M-mode).

Read access to bit *X* is subject to the same mcounteren (or mcounteren and hcounteren) CSRs that mediate access to the hpmcounter CSRs by S-mode (or VS-mode). In M-mode, scountovf bit *X* is always readable. In S/HS-mode, scountovf bit *X* is readable when mcounteren bit *X* is set, and otherwise reads as zero. Similarly, in VS mode, scountovf bit *X* is readable when mcounteren bit *X* and hcounteren bit *X* are both set, and otherwise reads as zero.

Chapter 19. "H" Extension for Hypervisor Support, Version 1.0

This chapter describes the RISC-V hypervisor extension, which virtualizes the supervisor-level architecture to support the efficient hosting of guest operating systems atop a type-1 or type-2 hypervisor. The hypervisor extension changes supervisor mode into *hypervisor-extended supervisor mode* (HS-mode, or *hypervisor mode* for short), where a hypervisor or a hosting-capable operating system runs. The hypervisor extension also adds another stage of address translation, from *guest physical addresses* to supervisor physical addresses, to virtualize the memory and memory-mapped I/O subsystems for a guest operating system. HS-mode acts the same as S-mode, but with additional instructions and CSRs that control the new stage of address translation and support hosting a guest OS in virtual S-mode (VS-mode). Regular S-mode operating systems can execute without modification either in HS-mode or as VS-mode guests.

In HS-mode, an OS or hypervisor interacts with the machine through the same SBI as an OS normally does from S-mode. An HS-mode hypervisor is expected to implement the SBI for its VS-mode guest.

The hypervisor extension depends on an "I" base integer ISA with 32 `x` registers (RV32I or RV64I), not RV32E or RV64E, which have only 16 `x` registers. CSR `mtval` must not be read-only zero, and standard page-based address translation must be supported, either Sv32 for RV32, or a minimum of Sv39 for RV64.

The hypervisor extension is enabled by setting bit 7 in the `misa` CSR, which corresponds to the letter H. RISC-V harts that implement the hypervisor extension are encouraged not to hardwire `misa`[7], so that the extension may be disabled.

The baseline privileged architecture is designed to simplify the use of classic virtualization techniques, where a guest OS is run at user-level, as the few privileged instructions can be easily detected and trapped. The hypervisor extension improves virtualization performance by reducing the frequency of these traps.

The hypervisor extension has been designed to be efficiently emulable on platforms that do not implement the extension, by running the hypervisor in S-mode and trapping into M-mode for hypervisor CSR accesses and to maintain shadow page tables. The majority of CSR accesses for type-2 hypervisors are valid S-mode accesses so need not be trapped. Hypervisors can support nested virtualization analogously.

19.1. Privilege Modes

The current *virtualization mode*, denoted V, indicates whether the hart is currently executing in a guest. When V=1, the hart is either in virtual S-mode (VS-mode), or in virtual U-mode (VU-mode) atop a guest OS running in VS-mode. When V=0, the hart is either in M-mode, in HS-mode, or in U-mode atop an OS running in HS-mode. The virtualization mode also indicates whether two-stage address translation is active (V=1) or inactive (V=0). Table 28 lists the possible privilege modes of a RISC-V hart with the hypervisor extension.

Table 28. Privilege modes with the hypervisor extension.

Virtualization Mode (V)	Nominal Privilege	Abbreviation	Name	Two-Stage Translation
0 0 0	U S M	U-mode HS-mode M-mode	User mode Hypervisor-extended supervisor mode Machine mode	Off Off Off
1 1	U S	VU-mode VS-mode	Virtual user mode Virtual supervisor mode	On On

For privilege modes U and VU, the *nominal privilege mode* is U, and for privilege modes HS and VS, the nominal privilege mode is S.

HS-mode is more privileged than VS-mode, and VS-mode is more privileged than VU-mode. VS-mode interrupts are globally disabled when executing in U-mode.

This description does not consider the possibility of U-mode or VU-mode interrupts and will be revised if an extension for user-level interrupts is adopted.

19.2. Hypervisor and Virtual Supervisor CSRs

An OS or hypervisor running in HS-mode uses the supervisor CSRs to interact with the exception, interrupt, and address-translation subsystems. Additional CSRs are provided to HS-mode, but not to VS-mode, to manage two-stage address translation and to control the behavior of a VS-mode guest: `hstatus`, `hedeleg`, `hideleg`, `hvip`, `hip`, `hie`, `hgeip`, `hgeie`, `henvcfg`, `henvcfgh`, `hcounteren`, `htimedelta`, `htimedeltah`, `htval`, `htinst`, and `hgatp`.

Furthermore, several *virtual supervisor* CSRs (VS CSRs) are replicas of the normal supervisor CSRs. For example, `vsstatus` is the VS CSR that duplicates the usual `sstatus` CSR.

When V=1, the VS CSRs substitute for the corresponding supervisor CSRs, taking over all functions of the usual supervisor CSRs except as specified otherwise. Instructions that normally read or modify a supervisor CSR shall instead access the corresponding VS CSR. When V=1, an attempt to read or write a VS CSR directly by its own separate CSR address causes a virtual-instruction exception. (Attempts from U-mode cause an illegal-instruction exception as usual.) The VS CSRs can be accessed as themselves only from M-mode or HS-mode.

While V=1, the normal HS-level supervisor CSRs that are replaced by VS CSRs retain their values but do not affect the behavior of the machine unless specifically documented to do so. Conversely, when V=0, the VS CSRs do not ordinarily affect the behavior of the machine other than being readable and writable by CSR instructions.

Some standard supervisor CSRs (`senvcfg`, `scounteren`, and `scontext`, possibly others) have no matching VS CSR. These supervisor CSRs continue to have their usual function and accessibility even when V=1, except with VS-mode and VU-mode substituting for HS-mode and U-mode. Hypervisor software is expected to manually swap the contents of these registers as needed.

Matching VS CSRs exist only for the supervisor CSRs that must be duplicated, which are mainly those that get automatically written by traps or that impact instruction execution immediately after trap entry and/or right before SRET, when software alone is unable to swap a CSR at exactly the right moment. Currently, most supervisor CSRs fall into this category, but future ones might not.

In this chapter, we use the term *HSXLEN* to refer to the effective XLEN when executing in HS-mode, and *VSXLEN* to refer to the effective XLEN when executing in VS-mode.

19.2.1. Hypervisor Status (`hstatus`) Register

The `hstatus` register is an HSXLEN-bit read/write register formatted as shown in Figure 70 when HSXLEN=32 and Figure 71 when HSXLEN=64. The `hstatus` register provides facilities analogous to the `mstatus` register for tracking and controlling the exception behavior of a VS-mode guest.

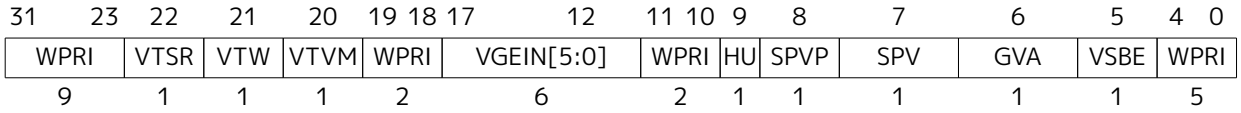


Figure 70. Hypervisor status register (`hstatus`) when HSLEN=32

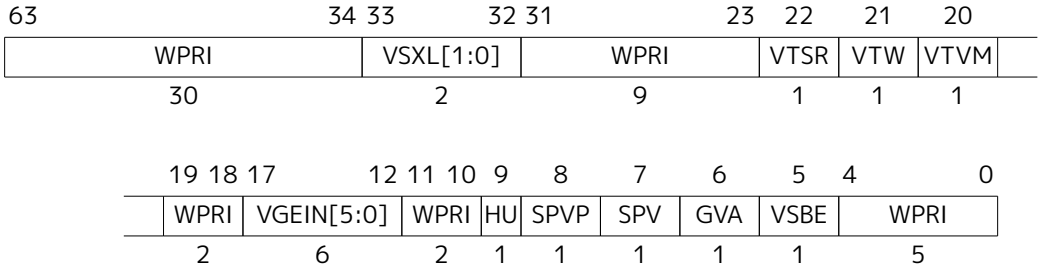


Figure 71. Hypervisor status register (`hstatus`) when HSXLEN=64.

The VSXL field controls the effective XLEN for VS-mode (known as VSXLEN), which may differ from the XLEN for HS-mode (HSXLEN). When HSXLEN=32, the VSXL field does not exist, and VSXLEN=32. When HSXLEN=64, VSXL is a **WARL** field that is encoded the same as the MXL field of `misa`, shown in Table 9. In particular, an implementation may make VSXL be a read-only field whose value always ensures that VSXLEN=HSXLEN.

If HSXLEN is changed from 32 to a wider width, and if field VSXL is not restricted to a single value, it gets the value corresponding to the widest supported width not wider than the new HSXLEN.

The `hstatus` fields VTSR, VTW, and VTVM are defined analogously to the `mstatus` fields TSR, TW, and TVM, but affect execution only in VS-mode, and cause virtual-instruction exceptions instead of illegal-instruction exceptions. When VTSR=1, an attempt in VS-mode to execute SRET raises a virtual-instruction exception. When VTW=1 (and assuming `mstatus`.TW=0), an attempt in VS-mode to execute WFI raises a virtual-instruction exception if the WFI does not complete within an implementation-specific, bounded time limit. An implementation may have WFI always raise a virtual-instruction exception in VS-mode when VTW=1 (and `mstatus`.TW=0), even if there are pending globally-disabled interrupts when the instruction is executed. When VTVM=1, an attempt in VS-mode to execute SFENCE.VMA or SINVAL.VMA or to access CSR `satp` raises a virtual-instruction exception.

The VGEIN (Virtual Guest External Interrupt Number) field selects a guest external interrupt source for VS-level external interrupts. VGEIN is a **WLRL** field that must be able to hold values between zero and the maximum guest external interrupt number (known as GEILEN), inclusive. When VGEIN=0, no guest external interrupt source is selected for VS-level external interrupts. GEILEN may be zero, in which case VGEIN may be read-only zero. Guest external interrupts are explained in Section 19.2.4, and the use of VGEIN is covered further in Section 19.2.3.

Field HU (Hypervisor in U-mode) controls whether the virtual-machine load/store instructions, HLV, HLVX, and HSV, can be used also in U-mode. When HU=1, these instructions can be executed in U-mode the same as in HS-mode. When HU=0, all hypervisor instructions cause an illegal-instruction exception in U-mode.

The HU bit allows a portion of a hypervisor to be run in U-mode for greater protection against software bugs, while still retaining access to a virtual machine's memory.

The SPV bit (Supervisor Previous Virtualization mode) is written by the implementation whenever a trap is taken into HS-mode. Just as the SPP bit in `sstatus` is set to the (nominal) privilege mode at the time of the trap, the SPV bit in `hstatus` is set to the value of the virtualization mode V at the time of the trap. When an SRET instruction is executed when V=0, V is set to SPV.

When V=1 and a trap is taken into HS-mode, bit SPVP (Supervisor Previous Virtual Privilege) is set to the nominal privilege mode at the time of the trap, the same as `sstatus`.SPP. But if V=0 before a trap, SPVP is left unchanged on trap entry. SPVP controls the effective privilege of explicit memory accesses made by the virtual-machine load/store instructions, HLV, HLVX, and HSV.

Without SPVP, if instructions HLV, HLVX, and HSV looked instead to `sstatus`*.SPP for the effective privilege of their memory accesses, then, even with HU=1, U-mode could not access virtual machine memory at VS-level, because to enter U-mode using SRET always leaves SPP=0. Unlike SPP, field SPVP is untouched by transitions back-and-forth between HS-mode and U-mode.*

Field GVA (Guest Virtual Address) is written by the implementation whenever a trap is taken into HS-mode. For any trap (breakpoint, address misaligned, access fault, page fault, or guest-page fault) that writes a guest virtual address to `stval`, GVA is set to 1. For any other trap into HS-mode, GVA is set to 0.

For breakpoint and memory access traps that write a nonzero value to `stval`*, GVA is redundant with field SPV (the two bits are set the same) except when the explicit memory access of an HLV, HLVX, or HSV instruction causes a fault. In that case, SPV=0 but GVA=1.*

The VSBE bit is a **WARL** field that controls the endianness of explicit memory accesses made from VS-mode. If VSBE=0, explicit load and store memory accesses made from VS-mode are little-endian, and if VSBE=1, they are big-endian. VSBE also controls the endianness of all implicit accesses to VS-level memory management data structures, such as page tables. An implementation may make VSBE a read-only field that always specifies the same endianness as HS-mode.

19.2.2. Hypervisor Trap Delegation (`hedeleg` and `hideleg`) Registers

Register `hedeleg` is a 64-bit read/write register, formatted as shown in Figure 72. Register `hideleg` is an HSXLEN-bit read/write register, formatted as shown in Figure 73. 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`.

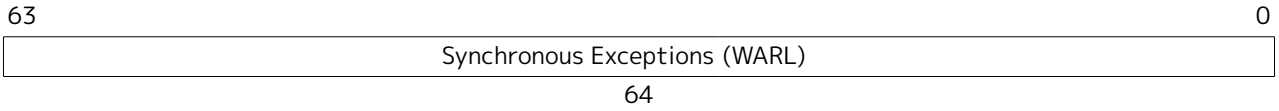


Figure 72. Hypervisor exception delegation register (`hedeleg`).

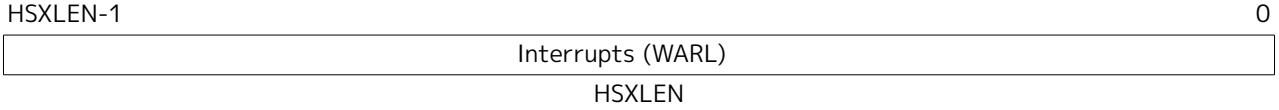


Figure 73. Hypervisor exception delegation register (`hideleg`).

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, as enumerated in Table

29. 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.

An interrupt that has been delegated to HS-mode (using `mideleg`) is further delegated to VS-mode if the corresponding `hideleg` bit is set. Among bits 15:0 of `hideleg`, bits 10, 6, and 2 (corresponding to the standard VS-level interrupts) are writable, and bits 12, 9, 5, and 1 (corresponding to the standard S-level interrupts) are read-only zeros.

When a virtual supervisor external interrupt (code 10) is delegated to VS-mode, it is automatically translated by the machine into a supervisor external interrupt (code 9) for VS-mode, including the value written to `vscause` on an interrupt trap. Likewise, a virtual supervisor timer interrupt (6) is translated into a supervisor timer interrupt (5) for VS-mode, and a virtual supervisor software interrupt (2) is translated into a supervisor software interrupt (1) for VS-mode. Similar translations may or may not be done for platform interrupt causes (codes 16 and above).

Table 29. Bits of `hedeleg` *that must be writable or must be read-only zero.*

Bit	Attribute	Corresponding Exception
0	(See text)	Instruction address misaligned
1	Writable	Instruction access fault
2	Writable	Illegal instruction
3	Writable	Breakpoint
4	Writable	Load address misaligned
5	Writable	Load access fault
6	Writable	Store/AMO address misaligned
7	Writable	Store/AMO access fault
8	Writable	Environment call from U-mode or VU-mode
9	Read-only 0	Environment call from HS-mode
10	Read-only 0	Environment call from VS-mode
11	Read-only 0	Environment call from M-mode
12	Writable	Instruction page fault
13	Writable	Load page fault
15	Writable	Store/AMO page fault
16	Read-only 0	Double trap
18	Writable	Software check
19	Writable	Hardware error
20	Read-only 0	Instruction guest-page fault
21	Read-only 0	Load guest-page fault
22	Read-only 0	Virtual instruction
23	Read-only 0	Store/AMO guest-page fault

19.2.3. Hypervisor Interrupt (`hvip`, `hip`, and `hie`) Registers

Register `hvip` is an HSXLEN-bit read/write register that a hypervisor can write to indicate virtual interrupts intended for VS-mode. Bits of `hvip` that are not writable are read-only zeros.

Figure 74. Hypervisor virtual-interrupt-pending register(`hvip`*).*

The standard portion (bits 15:0) of `hvip` is formatted as shown in Figure 75. Bits VSEIP, VSTIP, and VSSIP of `hvip` are writable. Setting VSEIP=1 in `hvip` asserts a VS-level external interrupt; setting VSTIP asserts a VS-level timer interrupt; and setting VSSIP asserts a VS-level software interrupt.

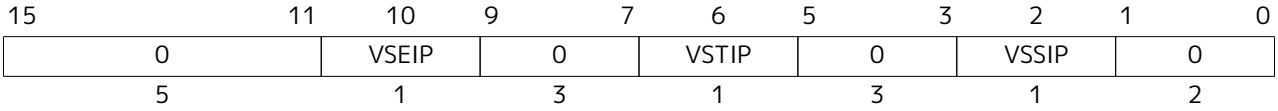


Figure 75. Standard portion (bits 15:0) of `hvip`.

Registers `hip` and `hie` are HSXLEN-bit read/write registers that supplement HS-level's `sip` and `sie` respectively. The `hip` register indicates pending VS-level and hypervisor-specific interrupts, while `hie` contains enable bits for the same interrupts.

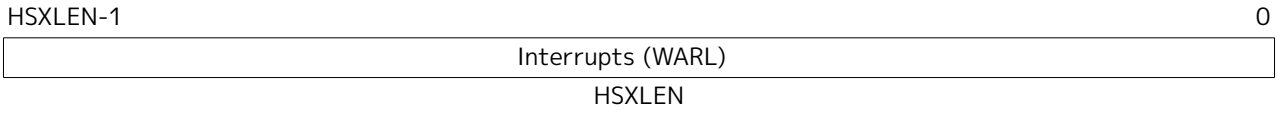


Figure 76. Hypervisor interrupt-pending register (`hip`*).*

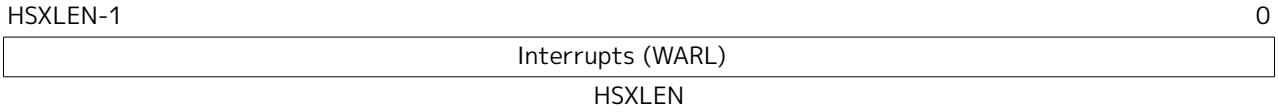


Figure 77. Hypervisor interrupt-enable register (`hie`*).*

For each writable bit in `sie`, the corresponding bit shall be read-only zero in both `hip` and `hie`. Hence, the nonzero bits in `sie` and `hie` are always mutually exclusive, and likewise for `sip` and `hip`.

The active bits of `hip` *and* `hie` *cannot be placed in HS-level's* `sip` *and* `sie` *because doing so would make it impossible for software to emulate the hypervisor extension on platforms that do not implement it in hardware.*

An interrupt *i* will trap to HS-mode whenever all of the following are true: (a) either the current operating mode is HS-mode and the SIE bit in the `sstatus` register is set, or the current operating mode has less privilege than HS-mode; (b) bit *i* is set in both `sip` and `sie`, or in both `hip` and `hie`; and (c) bit *i* is not set in `hideleg`.

If bit *i* of `sie` is read-only zero, the same bit in register `hip` may be writable or may be read-only. When bit *i* in `hip` is writable, a pending interrupt *i* can be cleared by writing 0 to this bit. If interrupt *i* can become pending in `hip` but bit *i* in `hip` is read-only, then either the interrupt can be cleared by clearing bit *i* of `hvip`, or the implementation must provide some other mechanism for clearing the pending interrupt (which may involve a call to the execution environment).

A bit in `hie` shall be writable if the corresponding interrupt can ever become pending in `hip`. Bits of `hie` that are not writable shall be read-only zero.

The standard portions (bits 15:0) of registers `hip` and `hie` are formatted as shown in Figure 78 and Figure 79 respectively.

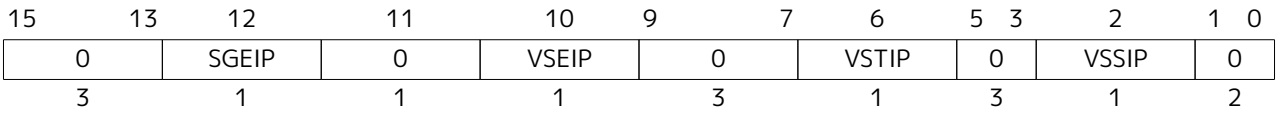


Figure 78. Standard portion (bits 15:0) of `hip`.

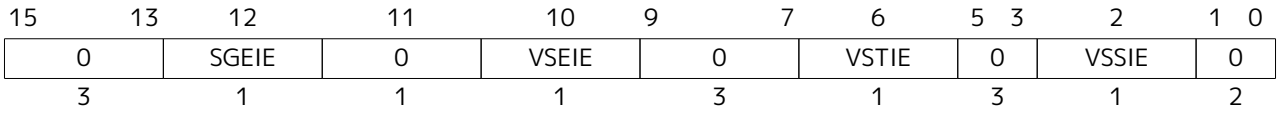


Figure 79. Standard portion (bits 15:0) of `hie`.

Bits `hip`.SGEIP and `hie`.SGEIE are the interrupt-pending and interrupt-enable bits for guest external interrupts at supervisor level (HS-level). SGEIP is read-only in `hip`, and is 1 if and only if the bitwise logical-AND of CSRs `hgeip` and `hgeie` is nonzero in any bit. (See Section 19.2.4.)

Bits `hip`.VSEIP and `hie`.VSEIE are the interrupt-pending and interrupt-enable bits for VS-level external interrupts. VSEIP is read-only in `hip`, and is the logical-OR of these interrupt sources:

- bit VSEIP of `hvip`;
- the bit of `hgeip` selected by `hstatus`.VGEIN; and
- any other platform-specific external interrupt signal directed to VS-level.

Bits `hip`.VSTIP and `hie`.VSTIE are the interrupt-pending and interrupt-enable bits for VS-level timer interrupts. VSTIP is read-only in `hip`, and is the logical-OR of `hvip`.VSTIP and any other platform-specific timer interrupt signal directed to VS-level.

Bits `hip`.VSSIP and `hie`.VSSIE are the interrupt-pending and interrupt-enable bits for VS-level software interrupts. VSSIP in `hip` is an alias (writable) of the same bit in `hvip`.

Multiple simultaneous interrupts destined for HS-mode are handled in the following decreasing priority order: SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI.

19.2.4. Hypervisor Guest External Interrupt Registers (`hgeip` and `hgeie`)

The `hgeip` register is an HSXLEN-bit read-only register, formatted as shown in Figure 80, that indicates pending guest external interrupts for this hart. The `hgeie` register is an HSXLEN-bit read/write register, formatted as shown in Figure 81, that contains enable bits for the guest external interrupts at this hart. Guest external interrupt number *i* corresponds with bit *i* in both `hgeip` and `hgeie`.

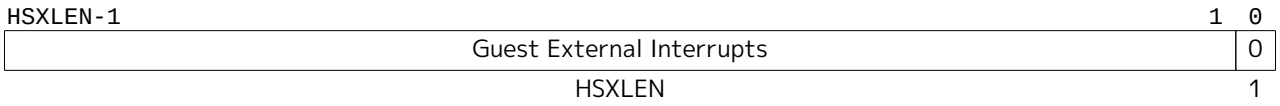


Figure 80. Hypervisor guest external interrupt-pending register (`hgeip`*).*

Figure 81. Hypervisor guest external interrupt-enable register (`hgeie`*).*

Guest external interrupts represent interrupts directed to individual virtual machines at VS-level. If a RISC-V platform supports placing a physical device under the direct control of a guest OS with minimal hypervisor intervention (known as *pass-through* or *direct assignment* between a virtual machine and the physical device), then, in such circumstance, interrupts from the device are intended for a specific virtual machine. Each bit of `hgeip` summarizes *all* pending interrupts directed to one virtual hart, as collected and reported by an interrupt controller. To distinguish specific pending interrupts from multiple devices, software must query the interrupt controller.

Support for guest external interrupts requires an interrupt controller that can collect virtual-machine-directed interrupts separately from other interrupts.

The number of bits implemented in `hgeip` and `hgeie` for guest external interrupts is UNSPECIFIED and may be zero. This number is known as *GEILEN*. The least-significant bits are implemented first, apart from bit 0. Hence, if GEILEN is nonzero, bits GEILEN:1 shall be writable in `hgeie`, and all other bit positions shall be read-only zeros in both `hgeip` and `hgeie`.

The set of guest external interrupts received and handled at one physical hart may differ from those received at other harts. Guest external interrupt number i at one physical hart is typically expected not to be the same as guest external interrupt i at any other hart. For any one physical hart, the maximum number of virtual harts that may directly receive guest external interrupts is limited by GEILEN. The maximum this number can be for any implementation is 31 for RV32 and 63 for RV64, per physical hart.

A hypervisor is always free to emulate devices for any number of virtual harts without being limited by GEILEN. Only direct pass-through (direct assignment) of interrupts is affected by the GEILEN limit, and the limit is on the number of virtual harts receiving such interrupts, not the number of distinct interrupts received. The number of distinct interrupts a single virtual hart may receive is determined by the interrupt controller.

Register `hgeie` selects the subset of guest external interrupts that cause a supervisor-level (HS-level) guest external interrupt. The enable bits in `hgeie` do not affect the VS-level external interrupt signal selected from `hgeip` by `hstatus`.VGEIN.

19.2.5. Hypervisor Environment Configuration Register (`henvcfg`)

The `henvcfg` CSR is a 64-bit read/write register, formatted as shown in Figure 82, that controls certain characteristics of the execution environment when virtualization mode V=1.

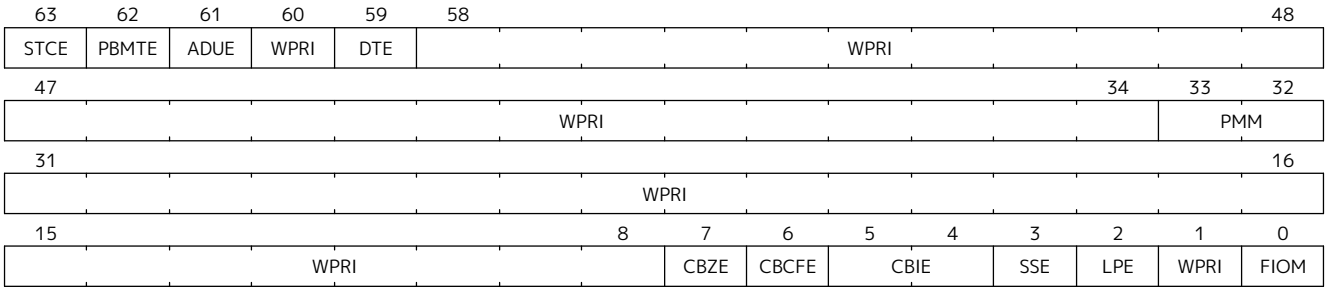


Figure 82. Hypervisor environment configuration register (`henvcfg`).

If bit 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 30 details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO when FIOM=1 and V=1.

Similarly, when 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 30. Modified interpretation of FENCE predecessor and successor sets when FIOM=1 and virtualization mode V=1.

Instruction bit	Meaning when set
PI PO	Predecessor device input and memory reads (PR implied) Predecessor device output and memory writes (PW implied)
SI SO	Successor device input and memory reads (SR implied) 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 is furnished by the Ssnpm 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.SSE` field will read as zero and is read-only.
- When `menvcfg.SSE` is one, `SSAMOSWAP.W/D` raises a virtual instruction exception.

The Ssdbltrp extension adds the double-trap-enable (`DTE`) field in `henvcfg`. When `henvcfg.DTE` is zero, the implementation behaves as though Ssdbltrp is not implemented for VS-mode and the `vsstatus.SDT` 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.

19.2.6. Hypervisor Counter-Enable (`hcounteren`) Register

The counter-enable register `hcounteren` is a 32-bit register that controls the availability of the hardware performance monitoring counters to the guest virtual machine.

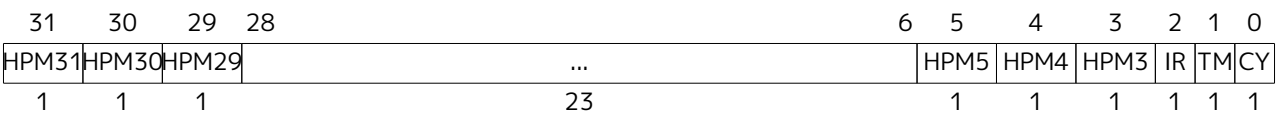


Figure 83. Hypervisor counter-enable register (`hcounteren`).

When the CY, TM, IR, or HPM_n_ bit in the `hcounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcounter` *n* register while V=1 will cause a virtual-instruction exception if the same bit in `mcounteren` is 1. When one of these bits is set, access to the corresponding register is permitted when V=1, unless prevented for some other reason. In VU-mode, a counter is not readable unless the applicable bits are set in both `hcounteren` and `scounteren`.

`hcounteren` must be implemented. However, any of the bits may be read-only zero, indicating reads to the corresponding counter will cause an exception when V=1. Hence, they are effectively **WARL** fields.

19.2.7. Hypervisor Time Delta (`htimedelta`) Register

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.*

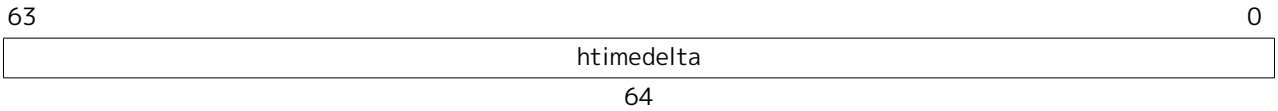


Figure 84. Hypervisor time delta register.

When XLEN=32, `htimedeltah` is a 32-bit read/write register that aliases bits 63:32 of `htimedelta`. Register `htimedeltah` does not exist when XLEN=64.

If the `time` CSR is implemented, `htimedelta` (and `htimedeltah` for XLEN=32) must be implemented.

19.2.8. Hypervisor Trap Value (`htval`) Register

The `htval` register is an HSXLEN-bit read/write register formatted as shown in Figure 85. 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.

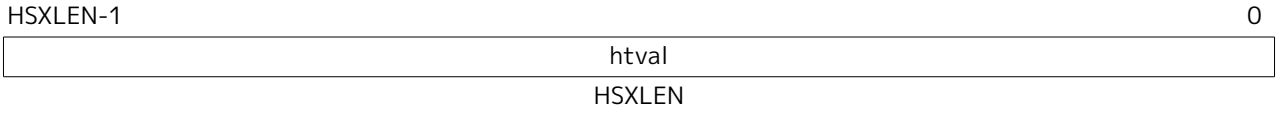


Figure 85. Hypervisor trap value register (`htval`).

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`.

A guest physical address written to `htval` *is shifted right by 2 bits to accommodate addresses wider than the current XLEN. For RV32, the hypervisor extension permits guest physical addresses as wide as 34 bits, and* `htval` *reports bits 33:2 of the address. This shift-by-2 encoding of guest physical addresses matches the encoding of physical addresses in PMP address registers (Section 3.7) and in page table entries (Section 11.3, Section 11.4, Section 11.5, and Section 11.6).*

If the least-significant two bits of a faulting guest physical address are needed, these bits are ordinarily the same as the least-significant two bits of the faulting virtual address in `stval`*. For faults due to implicit memory accesses for VS-stage address translation, the least-significant two bits are instead zeros. These cases can be distinguished using the value provided in register* `htinst`*.*

`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.

Unless it has reason to assume otherwise (such as a platform standard), software that writes a value to `htval` *should read back from* `htval` *to confirm the stored value.*

19.2.9. Hypervisor Trap Instruction (`htinst`) Register

The `htinst` register is an HSXLEN-bit read/write register formatted as shown in Figure 86. When a trap is taken into HS-mode, `htinst` 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 `htinst` on a trap are documented in Section 19.6.3.

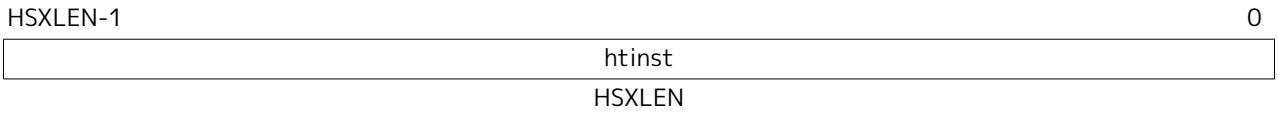


Figure 86. Hypervisor trap instruction (`htinst`) register.

`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.

19.2.10. Hypervisor Guest Address Translation and Protection (`hgatp`) Register

The `hgatp` register is an HSXLEN-bit read/write register, formatted as shown in Figure 87 for HSXLEN=32 and Figure 88 for HSXLEN=64, which controls G-stage address translation and protection, the second stage of two-stage translation for guest virtual addresses (see Section 19.5). 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 `hgatp` while executing in HS-mode will raise an illegal-instruction exception.

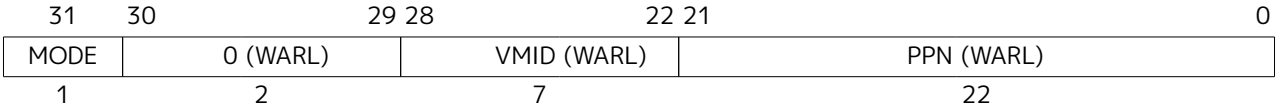


Figure 87. Hypervisor guest address translation and protection register `hgatp` *when HSXLEN=32.*

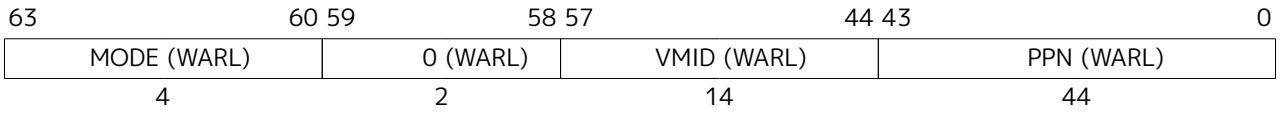


Figure 88. Hypervisor guest address translation and protection register `hgatp` *when HSXLEN=64 for MODE values Bare, Sv39x4, and Sv57x4.*

Table 31 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 Section 3.7. 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. All of these paged virtual-memory schemes are described in Section 19.5.1.

The remaining MODE settings when HSXLEN=64 are reserved for future use and may define different interpretations of the other fields in `hgatp`.

Table 31. Encoding of `hgatp` *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	No translation or protection.
1-7	—	*Reserved*
8	Sv39x4	Page-based 41-bit virtual addressing (2-bit extension of Sv39).
9	Sv48x4	Page-based 50-bit virtual addressing (2-bit extension of Sv48).
10	Sv57x4	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 `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.

As explained in Section 19.5.1, 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 `hgatp` 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 `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 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 `hgatp` 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 `hgatp` 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 Section 19.3.2) before or after writing `hgatp`.

19.2.11. Virtual Supervisor Status (`vsstatus`) Register

The `vsstatus` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `sstatus`, formatted as shown in Figure 89 when VSXLEN=32 and Figure 90 when VSXLEN=64. When V=1, `vsstatus` substitutes for the usual `sstatus`, so instructions that normally read or modify `sstatus` actually access `vsstatus` instead.

Figure 89. Virtual supervisor status (`vsstatus`) register when VSXLEN=32.

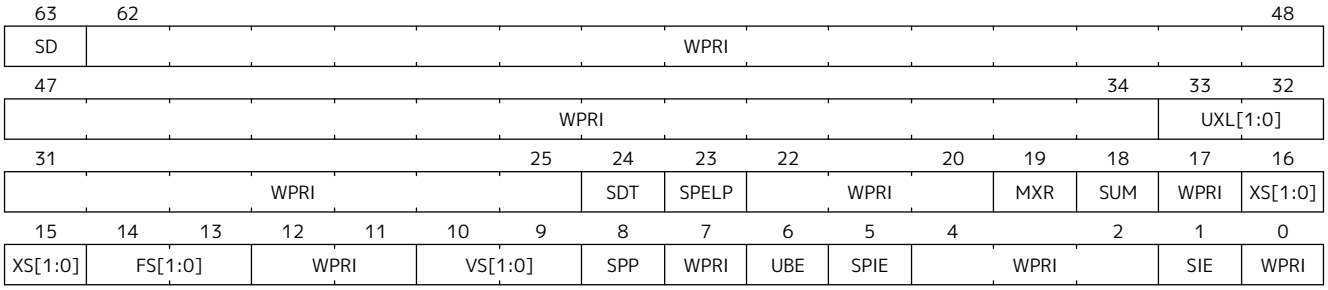


Figure 90. Virtual supervisor status (`vsstatus`) register when VSXLEN=64.

The UXL field controls the effective XLEN for VU-mode, which may differ from the XLEN for VS-mode (VSXLEN). When VSXLEN=32, the UXL field does not exist, and VU-mode XLEN=32. When VSXLEN=64, UXL is a **WARL** field that is encoded the same as the MXL field of `misa`, shown in Table 9. In particular, an implementation may make UXL be a read-only copy of field VSXL of `hstatus`, forcing VU-mode XLEN=VSXLEN.

If VSXLEN is changed from 32 to a wider width, and if field UXL is not restricted to a single value, it gets the value corresponding to the widest supported width not wider than the new VSXLEN.

When V=1, both `vsstatus`.FS and the HS-level `sstatus`.FS are in effect. Attempts to execute a floating-point instruction when either field is 0 (Off) raise an illegal-instruction exception. Modifying the floating-point state when V=1 causes both fields to be set to 3 (Dirty).

For a hypervisor to benefit from the extension context status, it must have its own copy in the HS-level `sstatus`, maintained independently of a guest OS running in VS-mode. While a version of the extension context status obviously must exist in `vsstatus` for VS-mode, a hypervisor cannot rely on this version being maintained correctly, given that VS-level software can change `vsstatus`.FS arbitrarily. If the HS-level `sstatus`.FS were not independently active and maintained by the hardware in parallel with `vsstatus`.FS while V=1, hypervisors would always be forced to conservatively swap all floating-point state when context-switching between virtual machines.

Similarly, when V=1, both `vsstatus`.VS and the HS-level `sstatus`.VS are in effect. Attempts to execute a vector instruction when either field is 0 (Off) raise an illegal-instruction exception. Modifying the vector

state when V=1 causes both fields to be set to 3 (Dirty).

Read-only fields SD and XS summarize the extension context status as it is visible to VS-mode only. For example, the value of the HS-level `sstatus`.FS does not affect `vsstatus`.SD.

An implementation may make field UBE be a read-only copy of `hstatus`.VSBE.

When V=0, `vsstatus` does not directly affect the behavior of the machine, unless a virtual-machine load/store (HLV, HLVX, or HSV) or the MPRV feature in the `mstatus` register is used to execute a load or store *as though* V=1.

The Zicfilp extension adds the `SPELP` field that holds the previous `ELP`, and is updated as specified in Section 20.1.2. The `SPELP` field is encoded as follows:

- 0 - `NO_LP_EXPECTED` - no landing pad instruction expected.
- 1 - `LP_EXPECTED` - a landing pad instruction is expected.

The Ssdbltrp adds an S-mode-disable-trap (`SDT`) field extension to address double trap (See Section 11.1.1.5) in VS-mode.

19.2.12. Virtual Supervisor Interrupt (`vsip` and `vsie`) Registers

The `vsip` and `vsie` registers are VSXLEN-bit read/write registers that are VS-mode's versions of supervisor CSRs `sip` and `sie`, formatted as shown in Figure 91 and Figure 92 respectively. When V=1, `vsip` and `vsie` substitute for the usual `sip` and `sie`, so instructions that normally read or modify `sip`/`sie` actually access `vsip`/`vsie` instead. However, interrupts directed to HS-level continue to be indicated in the HS-level `sip` register, not in `vsip`, when V=1.

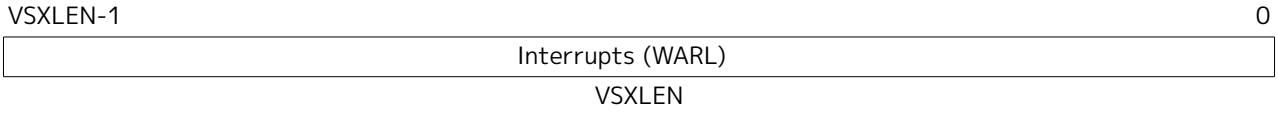


Figure 91. Virtual supervisor interrupt-pending register (`vsip`).

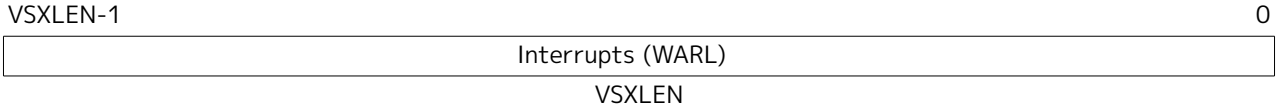


Figure 92. Virtual supervisor interrupt-enable register (`vsie`).

The standard portions (bits 15:0) of registers `vsip` and `vsie` are formatted as shown in Figure 93 and Figure 94 respectively.

Figure 93. Standard portion (bits 15:0) of `vsip`.

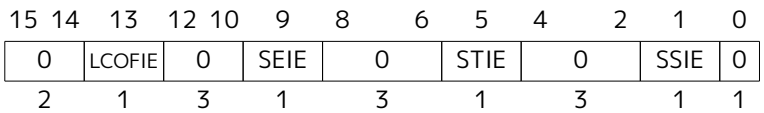


Figure 94. Standard portion (bits 15:0) of `vsie`.

Extension Shlcofideleg supports delegating LCOFI interrupts to VS-mode. If the Shlcofideleg extension is implemented, `hideleg` bit 13 is writable; otherwise, it is read-only zero. When bit 13 of `hideleg` is zero,

`vsip`.LCOFIP and `vsie`.LCOFIE are read-only zeros. Else, `vsip`.LCOFIP and `vsie`.LCOFIE are aliases of `sip`.LCOFIP and `sie`.LCOFIE.

When bit 10 of `hideleg` is zero, `vsip`.SEIP and `vsie`.SEIE are read-only zeros. Else, `vsip`.SEIP and `vsie`.SEIE are aliases of `hip`.VSEIP and `hie`.VSEIE.

When bit 6 of `hideleg` is zero, `vsip`.STIP and `vsie`.STIE are read-only zeros. Else, `vsip`.STIP and `vsie`.STIE are aliases of `hip`.VSTIP and `hie`.VSTIE.

When bit 2 of `hideleg` is zero, `vsip`.SSIP and `vsie`.SSIE are read-only zeros. Else, `vsip`.SSIP and `vsie`.SSIE are aliases of `hip`.VSSIP and `hie`.VSSIE.

19.2.13. Virtual Supervisor Trap Vector Base Address (`vstvec`) Register

The `vstvec` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `stvec`, formatted as shown in Figure 95. When V=1, `vstvec` substitutes for the usual `stvec`, so instructions that normally read or modify `stvec` actually access `vstvec` instead. When V=0, `vstvec` does not directly affect the behavior of the machine.

Figure 95. Virtual supervisor trap vector base address register `vstvec`.

19.2.14. Virtual Supervisor Scratch (`vsscratch`) Register

The `vsscratch` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `sscratch`, formatted as shown in Figure 96. When V=1, `vsscratch` substitutes for the usual `sscratch`, so instructions that normally read or modify `sscratch` actually access `vsscratch` instead. The contents of `vsscratch` never directly affect the behavior of the machine.

Figure 96. Virtual supervisor scratch register `vsscratch`.

19.2.15. Virtual Supervisor Exception Program Counter (`vsepc`) Register

The `vsepc` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `sepc`, formatted as shown in Figure 97. When V=1, `vsepc` substitutes for the usual `sepc`, so instructions that normally read or modify `sepc` actually access `vsepc` instead. When V=0, `vsepc` does not directly affect the behavior of the machine.

`vsepc` is a **WARL** register that must be able to hold the same set of values that `sepc` can hold.

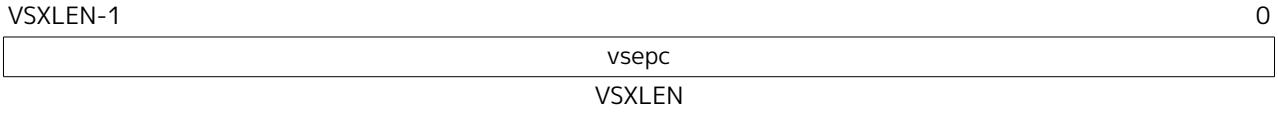


Figure 97. Virtual supervisor exception program counter (`vsepc`*).*

19.2.16. Virtual Supervisor Cause (`vscause`) Register

The `vscause` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `scause`, formatted as shown in Figure 98. When V=1, `vscause` substitutes for the usual `scause`, so instructions that normally read or modify `scause` actually access `vscause` instead. When V=0, `vscause` does not directly affect the behavior of the machine.

`vscause` is a **WLRL** register that must be able to hold the same set of values that `scause` can hold.

Figure 98. Virtual supervisor cause register (`vscause`).

19.2.17. Virtual Supervisor Trap Value (`vstval`) Register

The `vstval` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `stval`, formatted as shown in Figure 99. When V=1, `vstval` substitutes for the usual `stval`, so instructions that normally read or modify `stval` actually access `vstval` instead. When V=0, `vstval` does not directly affect the behavior of the machine.

`vstval` is a **WARL** register that must be able to hold the same set of values that `stval` can hold.

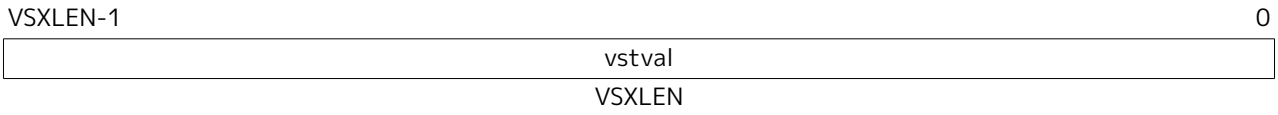


Figure 99. Virtual supervisor trap value register (`vstval`).

19.2.18. Virtual Supervisor Address Translation and Protection (`vsatp`) Register

The `vsatp` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `satp`, formatted as shown in Figure 100 for VSXLEN=32 and Figure 101 for VSXLEN=64. 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 (see Section 19.5).

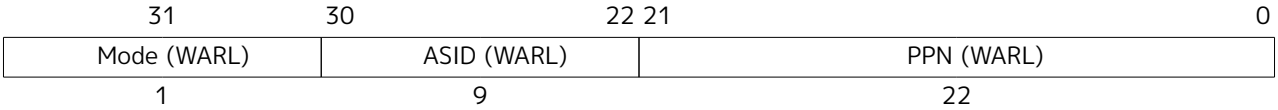


Figure 100. Virtual supervisor address translation and protection `vsatp` register when VSXLEN=32.

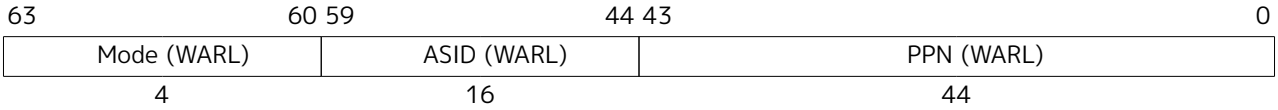


Figure 101. Virtual supervisor address translation and protection `vsatp` register when VSXLEN=64.

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.

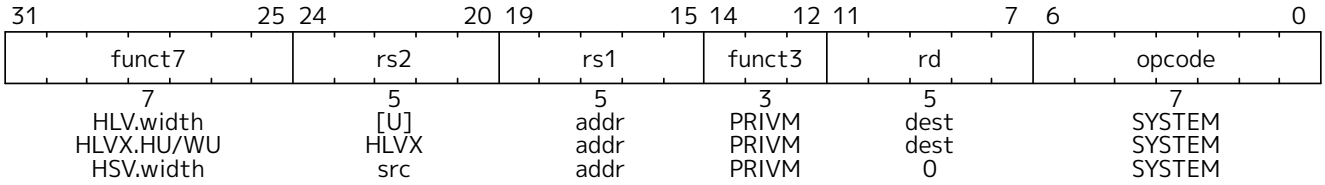
When V=0, a write to `vsatp` with an unsupported MODE value is either ignored as it is for `satp`, or the fields of `vsatp` are treated as **WARL** in the normal way. However, when V=1, a write to `satp` with an unsupported MODE value *is* ignored and no write to `vsatp` is effected.

When V=0, `vsatp` does not directly affect the behavior of the machine, unless a virtual-machine load/store (HLV, HLVX, or HSV) or the MPRV feature in the `mstatus` register is used to execute a load or store *as though* V=1.

19.3. Hypervisor Instructions

The hypervisor extension adds virtual-machine load and store instructions and two privileged fence instructions.

19.3.1. Hypervisor Virtual-Machine Load and Store Instructions



The hypervisor virtual-machine load and store instructions are valid only in M-mode or HS-mode, or in U-mode when `hstatus`.HU=1. Each instruction performs an explicit memory access as though V=1; i.e., with the address translation and protection, and the endianness, that apply to memory accesses in either VS-mode or VU-mode. Field SPVP of `hstatus` controls the privilege level of the access. The explicit memory access is done as though in VU-mode when SPVP=0, and as though in VS-mode when SPVP=1. As usual when V=1, two-stage address translation is applied, and the HS-level `sstatus`.SUM is ignored. HS-level `sstatus`.MXR makes execute-only pages readable by explicit loads for both stages of address translation (VS-stage and G-stage), whereas `vsstatus`.MXR affects only the first translation stage (VS-stage).

For every RV32I or RV64I load instruction, LB, LBU, LH, LHU, LW, LWU, and LD, there is a corresponding virtual-machine load instruction: HLV.B, HLV.BU, HLV.H, HLV.HU, HLV.W, HLV.WU, and HLV.D. For every RV32I or RV64I store instruction, SB, SH, SW, and SD, there is a corresponding virtual-machine store instruction: HSV.B, HSV.H, HSV.W, and HSV.D. Instructions HLV.WU, HLV.D, and HSV.D are not valid for RV32, of course.

Instructions HLVX.HU and HLVX.WU are the same as HLV.HU and HLV.WU, except that *execute* permission takes the place of *read* permission during address translation. That is, the memory being read must be executable in both stages of address translation, but read permission is not required. For the supervisor physical address that results from address translation, the supervisor physical memory attributes must grant both *execute* and *read* permissions. (The *supervisor physical memory attributes* are the machine's physical memory attributes as modified by physical memory protection, Section 3.7, for supervisor level.)

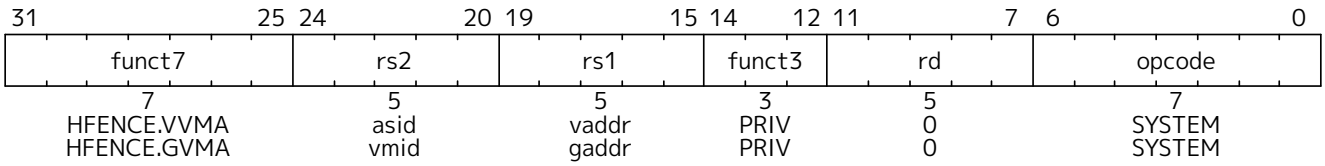
HLVX cannot override machine-level physical memory protection (PMP), so attempting to read memory that PMP designates as execute-only still results in an access-fault exception.

Although HLVX instructions' explicit memory accesses require execute permissions, they still raise the same exceptions as other load instructions, rather than raising fetch exceptions instead.

HLVX.WU is valid for RV32, even though LWU and HLV.WU are not. (For RV32, HLVX.WU can be considered a variant of HLV.W, as sign extension is irrelevant for 32-bit values.)

Attempts to execute a virtual-machine load/store instruction (HLV, HLVX, or HSV) when V=1 cause a virtual-instruction exception. Attempts to execute one of these same instructions from U-mode when `hstatus`.HU=0 cause an illegal-instruction exception.

19.3.2. Hypervisor Memory-Management Fence Instructions



The hypervisor memory-management fence instructions, HFENCE.VVMA and HFENCE.GVMA, perform a function similar to SFENCE.VMA (Section 11.2.1), except applying to the VS-level memory-management data structures controlled by CSR `vsatp` (HFENCE.VVMA) or the guest-physical memory-management data structures controlled by CSR `hgatp` (HFENCE.GVMA). Instruction SFENCE.VMA applies only to the memory-management data structures controlled by the current `satp` (either the HS-level `satp` when V=0 or `vsatp` when V=1).

HFENCE.VVMA is valid only in M-mode or HS-mode. Its effect is much the same as temporarily entering VS-mode and executing SFENCE.VMA. Executing an HFENCE.VVMA guarantees that any previous stores already visible to the current hart are ordered before all implicit reads by that hart done for VS-stage address translation for instructions that

- are subsequent to the HFENCE.VVMA, and
- execute when `hgatp`.VMID has the same setting as it did when HFENCE.VVMA executed.

Implicit reads need not be ordered when `hgatp`.VMID is different than at the time HFENCE.VVMA executed. If operand *rs1*≠`x0`, it specifies a single guest virtual address, and if operand *rs2*≠`x0`, it specifies a single guest address-space identifier (ASID).

An HFENCE.VVMA instruction applies only to a single virtual machine, identified by the setting of `hgatp`*.VMID when HFENCE.VVMA executes.*

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

Simpler implementations of HFENCE.VVMA can ignore the guest virtual address in rs1 and the guest ASID value in rs2, as well as `hgatp`*.VMID, and always perform a global fence for the VS-level memory management of all virtual machines, or even a global fence for all memory-management data structures.*

Neither `mstatus`.TVM nor `hstatus`.VTVM causes HFENCE.VVMA to trap.

HFENCE.GVMA is valid only in HS-mode when `mstatus`.TVM=0, or in M-mode (irrespective of `mstatus`.TVM). Executing an HFENCE.GVMA instruction guarantees that any previous stores already visible to the current hart are ordered before all implicit reads by that hart done for G-stage address translation for instructions that follow the HFENCE.GVMA. If operand *rs1*≠`x0`, it specifies a single guest physical address, shifted right by 2 bits, and if operand *rs2*≠`x0`, it specifies a single virtual machine identifier (VMID).

Conceptually, an implementation might contain two address-translation caches: one that maps guest virtual addresses to guest physical addresses, and another that maps guest

physical addresses to supervisor physical addresses. HFENCE.GVMA need not flush the former cache, but it must flush entries from the latter cache that match the HFENCE.GVMA's address and VMID arguments.

More commonly, implementations contain address-translation caches that map guest virtual addresses directly to supervisor physical addresses, removing a level of indirection. For such implementations, any entry whose guest virtual address maps to a guest physical address that matches the HFENCE.GVMA's address and VMID arguments must be flushed. Selectively flushing entries in this fashion requires tagging them with the guest physical address, which is costly, and so a common technique is to flush all entries that match the HFENCE.GVMA's VMID argument, regardless of the address argument.

Like for a guest physical address written to `htval` *on a trap, a guest physical address specified in rs1 is shifted right by 2 bits to accommodate addresses wider than the current XLEN.*

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

Simpler implementations of HFENCE.GVMA can ignore the guest physical address in rs1 and the VMID value in rs2 and always perform a global fence for the guest-physical memory management of all virtual machines, or even a global fence for all memory-management data structures.

If `hgatp`.MODE is changed for a given VMID, an HFENCE.GVMA with *rs1*=`x0` (and *rs2* set to either `x0` or the VMID) must be executed to order subsequent guest translations with the MODE change—even if the old MODE or new MODE is Bare.

Attempts to execute HFENCE.VVMA or HFENCE.GVMA when V=1 cause a virtual-instruction exception, while attempts to do the same in U-mode cause an illegal-instruction exception. Attempting to execute HFENCE.GVMA in HS-mode when `mstatus`.TVM=1 also causes an illegal-instruction exception.

19.4. Machine-Level CSRs

The hypervisor extension augments or modifies machine CSRs `mstatus`, `mstatush`, `mideleg`, `mip`, and `mie`, and adds CSRs `mtval2` and `mtinst`.

19.4.1. Machine Status (`mstatus` and `mstatush`) Registers

The hypervisor extension adds two fields, MPV and GVA, to the machine-level `mstatus` or `mstatush` CSR, and modifies the behavior of several existing `mstatus` fields. Figure 102 shows the modified `mstatus` register when the hypervisor extension is implemented and MXLEN=64. When MXLEN=32, the hypervisor extension adds MPV and GVA not to `mstatus` but to `mstatush`. Figure 103 shows the `mstatush` register when the hypervisor extension is implemented and MXLEN=32.

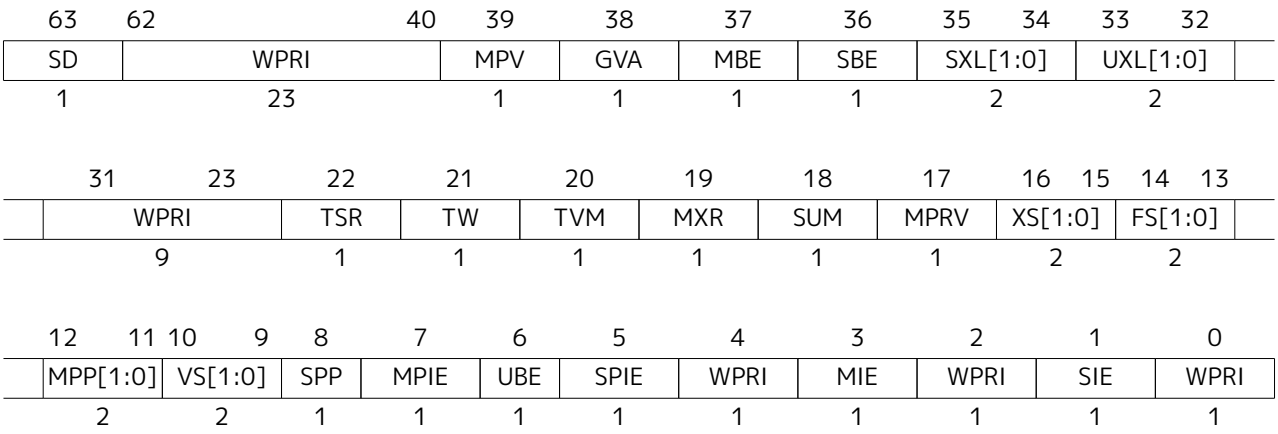


Figure 102. Machine status (`mstatus`) register for RV64 when the hypervisor extension is implemented.

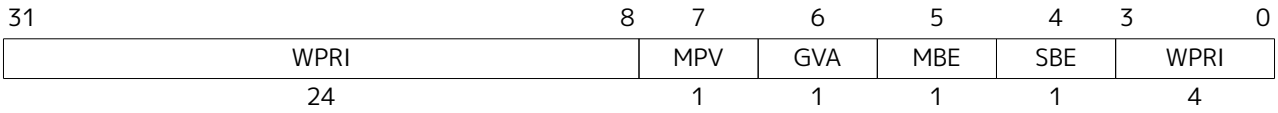


Figure 103. Additional machine status (`mstatush`) register for RV32 when the hypervisor extension is implemented. The format of `mstatus` is unchanged for RV32.

The MPV bit (Machine Previous Virtualization Mode) is written by the implementation whenever a trap is taken into M-mode. Just as the MPP field is set to the (nominal) privilege mode at the time of the trap, the MPV bit is set to the value of the virtualization mode V at the time of the trap. When an MRET instruction is executed, the virtualization mode V is set to MPV, unless MPP=3, in which case V remains 0.

Field GVA (Guest Virtual Address) is written by the implementation whenever a trap is taken into M-mode. For any trap (breakpoint, address misaligned, access fault, page fault, or guest-page fault) that writes a guest virtual address to `mtval`, GVA is set to 1. For any other trap into M-mode, GVA is set to 0.

The TSR and TVM fields of `mstatus` affect execution only in HS-mode, not in VS-mode. The TW field affects execution in all modes except M-mode.

Setting TVM=1 prevents HS-mode from accessing `hgatp` or executing HFENCE.GVMA or HINVAL.GVMA, but has no effect on accesses to `vsatp` or instructions HFENCE.VVMA or HINVAL.VVMA.

TVM exists in `mstatus` to allow machine-level software to modify the address translations managed by a supervisor-level OS, usually for the purpose of inserting another stage of address translation below that controlled by the OS. The instruction traps enabled by TVM=1 permit machine level to co-opt both `satp` and `hgatp` and substitute shadow page tables that merge the OS's chosen page translations with M-level's lower-stage translations, all without the OS being aware. M-level software needs this ability not only to emulate the hypervisor extension if not already supported, but also to emulate any future RISC-V extensions that may modify or add address translation stages, perhaps, for example, to improve support for nested hypervisors, i.e., running hypervisors atop other hypervisors.

However, setting TVM=1 does not cause traps for accesses to `vsatp` or instructions HFENCE.VVMA or HINVAL.VVMA, or for any actions taken in VS-mode, because M-level software is not expected to need to involve itself in VS-stage address translation. For virtual machines, it should be sufficient, and in all likelihood faster as well, to leave VS-stage address translation alone and merge all other translation stages into G-stage shadow page tables controlled by `hgatp`. This assumption does place some constraints on possible future RISC-V extensions that current machines will be able to emulate efficiently.

The hypervisor extension changes the behavior of the Modify Privilege field, MPRV, of `mstatus`. When

MPRV=0, translation and protection behave as normal. When MPRV=1, explicit memory accesses are translated and protected, and endianness is applied, as though the current virtualization mode were set to MPV and the current nominal privilege mode were set to MPP. Table 32 enumerates the cases.

Table 32. Effect of MPRV on the translation and protection of explicit memory accesses.

MPRV	MPV	MPP	Effect
0	-	-	Normal access; current privilege mode applies.
1	0	0	U-level access with HS-level translation and protection only.
1	0	1	HS-level access with HS-level translation and protection only.
1	-	3	M-level access with no translation.
1	1	0	VU-level access with two-stage translation and protection. The HS-level MXR bit makes any executable page readable. `vsstatus`.MXR makes readable those pages marked executable at the VS translation stage, but only if readable at the guest-physical translation stage.
1	1	1	VS-level access with two-stage translation and protection. The HS-level MXR bit makes any executable page readable. `vsstatus`.MXR makes readable those pages marked executable at the VS translation stage, but only if readable at the guest-physical translation stage. `vsstatus`.SUM applies instead of the HS-level SUM bit.

MPRV does not affect the virtual-machine load/store instructions, HLV, HLVX, and HSV. The explicit loads and stores of these instructions always act as though V=1 and the nominal privilege mode were `hstatus`.SPVP, overriding MPRV.

The `mstatus` register is a superset of the HS-level `sstatus` register but is not a superset of `vsstatus`.

19.4.2. Machine Interrupt Delegation (`mideleg`) Register

When the hypervisor extension is implemented, bits 10, 6, and 2 of `mideleg` (corresponding to the standard VS-level interrupts) are each read-only one. Furthermore, if any guest external interrupts are implemented (GEILEN is nonzero), bit 12 of `mideleg` (corresponding to supervisor-level guest external interrupts) is also read-only one. VS-level interrupts and guest external interrupts are always delegated past M-mode to HS-mode.

For bits of `mideleg` that are zero, the corresponding bits in `hideleg`, `hip`, and `hie` are read-only zeros.

19.4.3. Machine Interrupt (`mip` and `mie`) Registers

The hypervisor extension gives registers `mip` and `mie` additional active bits for the hypervisor-added interrupts. Figure 104 and Figure 105 show the standard portions (bits 15:0) of registers `mip` and `mie` when the hypervisor extension is implemented.

Figure 104. Standard portion (bits 15:0) of `mip`.

Figure 105. Standard portion (bits 15:0) of `mie`.

Bits SGEIP, VSEIP, VSTIP, and VSSIP in `mip` are aliases for the same bits in hypervisor CSR `hip`, while SGEIE, VSEIE, VSTIE, and VSSIE in `mie` are aliases for the same bits in `hie`.

19.4.4. Machine Second Trap Value (`mtval2`) Register

The `mtval2` register is an MXLEN-bit read/write register formatted as shown in Figure 106. When a trap is taken into M-mode, `mtval2` is written with additional exception-specific information, alongside `mtval`, to assist software in handling the trap.

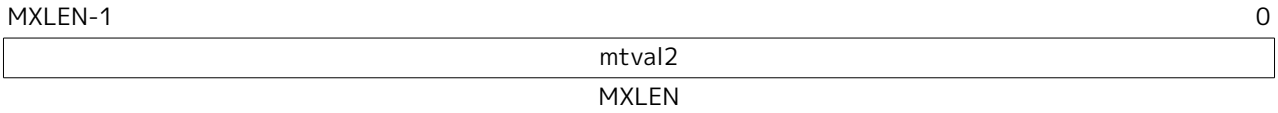


Figure 106. Machine second trap value register (`mtval2`).

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.

The Ssdbltrap extension (See Chapter 21) requires the implementation of the `mtval2` CSR.

19.4.5. Machine Trap Instruction (`mtinst`) Register

The `mtinst` register is an MXLEN-bit read/write register formatted as shown in Figure 107. 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 Section 19.6.3.

Figure 107. Machine trap instruction (`mtinst`) register.

`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.

19.5. Two-Stage Address Translation

Whenever the current virtualization mode V is 1, two-stage address translation and protection is in effect. For any virtual memory access, the original virtual address is converted in the first stage by VS-level address translation, as controlled by the `vsatp` register, into a *guest physical address*. The guest physical

address is then converted in the second stage by guest physical address translation, as controlled by the `hgatp` register, into a supervisor physical address. The two stages are known also as VS-stage and G-stage translation. Although there is no option to disable two-stage address translation when V=1, either stage of translation can be effectively disabled by zeroing the corresponding `vsatp` or `hgatp` register.

The `vsstatus` field MXR, which makes execute-only pages readable by explicit loads, only overrides VS-stage page protection. Setting MXR at VS-level does not override guest-physical page protections. Setting MXR at HS-level, however, overrides both VS-stage and G-stage execute-only permissions.

When V=1, memory accesses that would normally bypass address translation are subject to G-stage address translation alone. This includes memory accesses made in support of VS-stage address translation, such as reads and writes of VS-level page tables.

Machine-level physical memory protection applies to supervisor physical addresses and is in effect regardless of virtualization mode.

19.5.1. Guest Physical Address Translation

The mapping of guest physical addresses to supervisor physical addresses is controlled by CSR `hgatp` (Section 19.2.10).

When the address translation scheme selected by the MODE field of `hgatp` is Bare, guest physical addresses are equal to supervisor physical addresses without modification, and no memory protection applies in the trivial translation of guest physical addresses to supervisor physical addresses.

When `hgatp`.MODE specifies a translation scheme of Sv32x4, Sv39x4, Sv48x4, or Sv57x4, G-stage address translation is a variation on the usual page-based virtual address translation scheme of Sv32, Sv39, Sv48, or Sv57, respectively. In each case, the size of the incoming address is widened by 2 bits (to 34, 41, 50, or 59 bits). To accommodate the 2 extra bits, the root page table (only) is expanded by a factor of four to be 16 KiB instead of the usual 4 KiB. Matching its larger size, the root page table also must be aligned to a 16 KiB boundary instead of the usual 4 KiB page boundary. Except as noted, all other aspects of Sv32, Sv39, Sv48, or Sv57 are adopted unchanged for G-stage translation. Non-root page tables and all page table entries (PTEs) have the same formats as documented in Section 11.3, Section 11.4, Section 11.5, and Section 11.6.

For Sv32x4, an incoming guest physical address is partitioned into a virtual page number (VPN) and page offset as shown in Figure 108. This partitioning is identical to that for an Sv32 virtual address as depicted in Figure 58, except with 2 more bits at the high end in VPN[1]. (Note that the fields of a partitioned guest physical address also correspond one-for-one with the structure that Sv32 assigns to a physical address, depicted in Figure 58.)

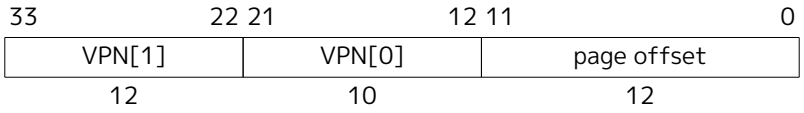


Figure 108. Sv32x4 virtual address (guest physical address).

For Sv39x4, an incoming guest physical address is partitioned as shown in Figure 109. This partitioning is identical to that for an Sv39 virtual address as depicted in Figure 61, except with 2 more bits at the high end in VPN[2]. Address bits 63:41 must all be zeros, or else a guest-page-fault exception occurs.

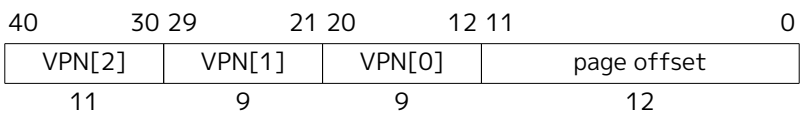


Figure 109. Sv39x4 virtual address (guest physical address).

For Sv48x4, an incoming guest physical address is partitioned as shown in Figure 110. This partitioning is identical to that for an Sv48 virtual address as depicted in Figure 64, except with 2 more bits at the high end in VPN[3]. Address bits 63:50 must all be zeros, or else a guest-page-fault exception occurs.

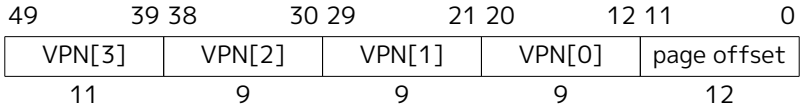


Figure 110. Sv48x4 virtual address (guest physical address).

For Sv57x4, an incoming guest physical address is partitioned as shown in Figure 111. This partitioning is identical to that for an Sv57 virtual address as depicted in Figure 67, except with 2 more bits at the high end in VPN[4]. Address bits 63:59 must all be zeros, or else a guest-page-fault exception occurs.

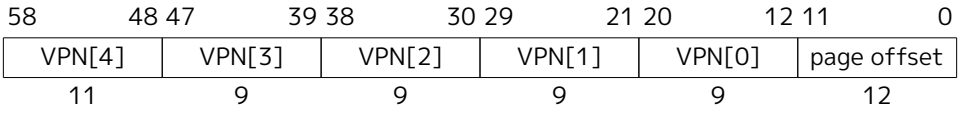


Figure 111. Sv57x4 virtual address (guest physical address).

The page-based G-stage address translation scheme for RV32, Sv32x4, is defined to support a 34-bit guest physical address so that an RV32 hypervisor need not be limited in its ability to virtualize real 32-bit RISC-V machines, even those with 33-bit or 34-bit physical addresses. This may include the possibility of a machine virtualizing itself, if it happens to use 33-bit or 34-bit physical addresses. Multiplying the size and alignment of the root page table by a factor of four is the cheapest way to extend Sv32 to cover a 34-bit address. The possible wastage of 12 KiB for an unnecessarily large root page table is expected to be of negligible consequence for most (maybe all) real uses.

A consistent ability to virtualize machines having as much as four times the physical address space as virtual address space is believed to be of some utility also for RV64. For a machine implementing 39-bit virtual addresses (Sv39), for example, this allows the hypervisor extension to support up to a 41-bit guest physical address space without either necessitating hardware support for 48-bit virtual addresses (Sv48) or falling back to emulating the larger address space using shadow page tables.

The conversion of an Sv32x4, Sv39x4, Sv48x4, or Sv57x4 guest physical address is accomplished with the same algorithm used for Sv32, Sv39, Sv48, or Sv57, as presented in Section 11.3.2, except that:

- `hgatp` substitutes for the usual `satp`;
- for the translation to begin, the effective privilege mode must be VS-mode or VU-mode;
- when checking the U bit, the current privilege mode is always taken to be U-mode; and
- guest-page-fault exceptions are raised instead of regular page-fault exceptions.

For G-stage address translation, all memory accesses (including those made to access data structures for VS-stage address translation) are considered to be user-level accesses, as though executed in U-mode. Access type permissions—readable, writable, or executable—are checked during G-stage translation the same as for VS-stage translation. For a memory access made to support VS-stage address translation (such as to read/write a VS-level page table), permissions and the need to set A and/or D bits at the G-stage level are checked as though for an implicit load or store, not for the original access type. However, any exception is always reported for the original access type (instruction, load, or store/AMO).

The G bit in all G-stage PTEs is reserved for future standard use. Until its use is defined by a standard extension, it should be cleared by software for forward compatibility, and must be ignored by hardware.

G-stage address translation uses the identical format for PTEs as regular address translation, even including the U bit, due to the possibility of sharing some (or all) page tables between G-stage translation and regular HS-level address translation. Regardless of whether this usage will ever become common, we chose not to preclude it.

19.5.2. Guest-Page Faults

Guest-page-fault traps may be delegated from M-mode to HS-mode under the control of CSR `medeleg`, but cannot be delegated to other privilege modes. On a guest-page fault, CSR `mtval` or `stval` is written with the faulting guest virtual address as usual, and `mtval2` or `htval` is written either with zero or with the faulting guest physical address, shifted right by 2 bits. CSR `mtinst` or `htinst` may also be written with information about the faulting instruction or other reason for the access, as explained in Section 19.6.3.

When an instruction fetch or a misaligned memory access straddles a page boundary, two different address translations are involved. When a guest-page fault occurs in such a circumstance, the faulting virtual address written to `mtval`/`stval` is the same as would be required for a regular page fault. Thus, the faulting virtual address may be a page-boundary address that is higher than the instruction's original virtual address, if the byte at that page boundary is among the accessed bytes.

When a guest-page fault is not due to an implicit memory access for VS-stage address translation, a nonzero guest physical address written to `mtval2`/`htval` shall correspond to the exact virtual address written to `mtval`/`stval`.

19.5.3. Memory-Management Fences

The behavior of the SFENCE.VMA instruction is affected by the current virtualization mode V. When V=0, the virtual-address argument is an HS-level virtual address, and the ASID argument is an HS-level ASID. The instruction orders stores only to HS-level address-translation structures with subsequent HS-level address translations.

When V=1, the virtual-address argument to SFENCE.VMA is a guest virtual address within the current virtual machine, and the ASID argument is a VS-level ASID within the current virtual machine. The current virtual machine is identified by the VMID field of CSR `hgatp`, and the effective ASID can be considered to be the combination of this VMID with the VS-level ASID. The SFENCE.VMA instruction orders stores only to the VS-level address-translation structures with subsequent VS-stage address translations for the same virtual machine, i.e., only when `hgatp`.VMID is the same as when the SFENCE.VMA executed.

Hypervisor instructions HFENCE.VVMA and HFENCE.GVMA provide additional memory-management fences to complement SFENCE.VMA. These instructions are described in Section 19.3.2.

Section 3.7.2 discusses the intersection between physical memory protection (PMP) and page-based address translation. It is noted there that, when PMP settings are modified in a manner that affects either the physical memory that holds page tables or the physical memory to which page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. For HS-level address translation, this is accomplished by executing in M-mode an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0`, after the PMP CSRs are written. Synchronization with G-stage and VS-stage data structures is also needed. Executing an HFENCE.GVMA instruction with *rs1*=`x0` and *rs2*=`x0` suffices to flush all G-stage or VS-stage address-translation cache entries that have cached PMP settings corresponding to the final translated supervisor physical address. An HFENCE.VVMA instruction is not required.

Similarly, 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.

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.*

19.6. Traps

19.6.1. Trap Cause Codes

The hypervisor extension augments the trap cause encoding. Table 33 lists the possible M-mode and HS-mode trap cause codes when the hypervisor extension is implemented. Codes are added for VS-level interrupts (interrupts 2, 6, 10), for supervisor-level guest external interrupts (interrupt 12), for virtual-instruction exceptions (exception 22), and for guest-page faults (exceptions 20, 21, 23). Furthermore, environment calls from VS-mode are assigned cause 10, whereas those from HS-mode or S-mode use cause 9 as usual.

Table 33. Machine and supervisor cause register (`mcause` *and* `scause`*) values when the hypervisor extension is implemented.*

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	Virtual supervisor software interrupt
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	Virtual supervisor timer interrupt
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	Virtual supervisor external interrupt
1	11	Machine external interrupt
1	12	Supervisor guest external interrupt
1	13	*Reserved for counter-overflow interrupt*
1	14-15	*Reserved*
1	≥16	*Designated for platform use*

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode or VU-mode
0	9	Environment call from HS-mode
0	10	Environment call from VS-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16-19	*Reserved*
0	20	Instruction guest-page fault
0	21	Load guest-page fault
0	22	Virtual instruction
0	23	Store/AMO guest-page fault
0	24-31	*Designated for custom use*
0	32-47	*Reserved*
0	48-63	*Designated for custom use*
0	≥64	*Reserved*

HS-mode and VS-mode ECALLs use different cause values so they can be delegated separately.

When V=1, a virtual-instruction exception (code 22) is normally raised instead of an illegal-instruction exception if the attempted instruction is *HS-qualified* but is prevented from executing when V=1 either due to insufficient privilege or because the instruction is expressly disabled by a supervisor or hypervisor CSR such as `scounteren` or `hcounteren`. An instruction is *HS-qualified* if it would be valid to execute in HS-mode (for some values of the instruction's register operands), assuming fields TSR and TVM of CSR `mstatus` are both zero.

A special rule applies for CSR instructions that access 32-bit high-half CSRs such as `cycleh` and `htimedeltah`. When V=1 and XLEN=32, an invalid attempt to access a high-half CSR raises a virtual-instruction exception instead of an illegal-instruction exception if the same CSR instruction for the corresponding *low-half* CSR (e.g.`cycle` or `htimedelta`) is HS-qualified.

When XLEN>32, an attempt to access a high-half CSR always raises an illegal-instruction exception.

Specifically, a virtual-instruction exception is raised for the following cases:

- in VS-mode, attempts to access a non-high-half counter CSR when the corresponding bit in `hcounteren` is 0 and the same bit in `mcounteren` is 1;
- in VS-mode, if XLEN=32, attempts to access a high-half counter CSR when the corresponding bit in `hcounteren` is 0 and the same bit in `mcounteren` is 1;
- in VU-mode, attempts to access a non-high-half counter CSR when the corresponding bit in either `hcounteren` or `scounteren` is 0 and the same bit in `mcounteren` is 1;
- in VU-mode, if XLEN=32, attempts to access a high-half counter CSR when the corresponding bit in either `hcounteren` or `scounteren` is 0 and the same bit in `mcounteren` is 1;
- in VS-mode or VU-mode, attempts to execute a hypervisor instruction (HLV, HLVX, HSV, or HFENCE);

- in VS-mode or VU-mode, attempts to access an implemented non-high-half hypervisor CSR or VS CSR when the same access (read/write) would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VS-mode or VU-mode, if XLEN=32, attempts to access an implemented high-half hypervisor CSR or high-half VS CSR when the same access (read/write) to the CSR"s low-half partner would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VU-mode, attempts to execute WFI when `mstatus`.TW=0, or to execute a supervisor instruction (SRET or SFENCE);
- in VU-mode, attempts to access an implemented non-high-half supervisor CSR when the same access (read/write) would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VU-mode, if XLEN=32, attempts to access an implemented high-half supervisor CSR when the same access to the CSR's low-half partner would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VS-mode, attempts to execute WFI when `hstatus`.VTW=1 and `mstatus`.TW=0, unless the instruction completes within an implementation-specific, bounded time;
- in VS-mode, attempts to execute SRET when `hstatus`.VTSR=1; and
- in VS-mode, attempts to execute an SFENCE.VMA or SINVAL.VMA instruction or to access `satp`, when `hstatus`.VTVM=1.

Other extensions to the RISC-V Privileged Architecture may add to the set of circumstances that cause a virtual-instruction exception when V=1.

On a virtual-instruction trap, `mtval` or `stval` is written the same as for an illegal-instruction trap.

It is not unusual that hypervisors must emulate the instructions that raise virtual-instruction exceptions, to support nested hypervisors or for other reasons. Machine level is expected ordinarily to delegate virtual-instruction traps directly to HS-level, whereas illegal-instruction traps are likely to be processed first in M-mode before being conditionally delegated (by software) to HS-level. Consequently, virtual-instruction traps are expected typically to be handled faster than illegal-instruction traps.

When not emulating the trapping instruction, a hypervisor should convert a virtual-instruction trap into an illegal-instruction exception for the guest virtual machine.

Because TSR and TVM in `mstatus` are intended to impact only S-mode (HS-mode), they are ignored for determining exceptions in VS-mode.

Fields FS and VS in registers `sstatus` and `vsstatus` deviate from the usual *HS-qualified* rule. If an instruction is prevented from executing because FS or VS is zero in either `sstatus` or `vsstatus`, the exception raised is always an illegal-instruction exception, never a virtual-instruction exception.

Early implementations of the H extension treated FS and VS in `sstatus` and `vsstatus` specially this way, and the behavior has been codified to maintain compatibility for software.

Table 34. Synchronous exception priority when the hypervisor extension is implemented.

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
	12, 20, 1	During instruction address translation: First encountered page fault, guest-page fault, or access fault
	1	With physical address for instruction: Instruction access fault
	2 22 0 8, 9, 10, 11 3 3	Illegal instruction Virtual instruction Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	4,6	Optionally: Load/store/AMO address misaligned
	13, 15, 21, 23, 5, 7	During address translation for an explicit memory access: First encountered page fault, guest-page fault, or access fault
	5, 7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4, 6	If not higher priority: Load/store/AMO address misaligned

If an instruction may raise multiple synchronous exceptions, the decreasing priority order of Table 34 indicates which exception is taken and reported in `mcause` or `scause`.

19.6.2. Trap Entry

When a trap occurs in HS-mode or U-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode. When a trap occurs in VS-mode or VU-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode, unless further delegated by `hedeleg` or `hideleg`, in which case it goes to VS-mode.

When a trap is taken into M-mode, virtualization mode V gets set to 0, and fields MPV and MPP in `mstatus` (or `mstatush`) are set according to Table 35. A trap into M-mode also writes fields GVA, MPIE, and MIE in `mstatus`/`mstatush` and writes CSRs `mepc`, `mcause`, `mtval`, `mtval2`, and `mtinst`.

Table 35. Value of `mstatus`/`mstatush` fields MPV and MPP after a trap into M-mode. Upon trap return, MPV is ignored when MPP=3.

Previous Mode	MPV	MPP
U-mode	0	0
HS-mode	0	1
M-mode	0	3
VU-mode	1	0
VS-mode	1	1

When a trap is taken into HS-mode, virtualization mode V is set to 0, and `hstatus`.SPV and `sstatus`.SPP are set according to Table 36. If V was 1 before the trap, field SPVP in `hstatus` is set the same as `sstatus`.SPP; otherwise, SPVP is left unchanged. A trap into HS-mode also writes field GVA in `hstatus`, fields SPIE and SIE in `sstatus`, and CSRs `sepc`, `scause`, `stval`, `htval`, and `htinst`.

Table 36. Value of `hstatus` field SPV and `sstatus` field SPP after a trap into HS-mode.

Previous Mode	SPV	SPP
U-mode	0	0
HS-mode	0	1
VU-mode	1	0
VS-mode	1	1

When a trap is taken into VS-mode, `vsstatus`.SPP is set according to Table 37. Register `hstatus` and the HS-level `sstatus` are not modified, and the virtualization mode V remains 1. A trap into VS-mode also writes fields SPIE and SIE in `vsstatus` and writes CSRs `vsepc`, `vscause`, and `vstval`.

Table 37. Value of `vsstatus` field SPP after a trap into VS-mode.

Previous Mode	SPP
VU-mode	0
VS-mode	1

19.6.3. Transformed Instruction or Pseudoinstruction for `mtinst` or `htinst`

On any trap into M-mode or HS-mode, one of these values is written automatically into the appropriate trap instruction CSR, `mtinst` or `htinst`:

- zero;
- a transformation of the trapping instruction;
- a custom value (allowed only if the trapping instruction is non-standard); or
- a special pseudoinstruction.

Except when a pseudoinstruction value is required (described later), the value written to `mtinst` or `htinst` may always be zero, indicating that the hardware is providing no information in the register for this particular trap.

The value written to the trap instruction CSR serves two purposes. The first is to improve the speed of instruction emulation in a trap handler, partly by allowing the handler to skip loading the trapping instruction from memory, and partly by obviating some of the work of decoding and executing the instruction. The second purpose is to supply, via pseudoinstructions, additional information about guest-page-fault exceptions caused by implicit memory accesses done for VS-stage address translation.

A transformation of the trapping instruction is written instead of simply a copy of the original instruction in order to minimize the burden for hardware yet still provide to a trap handler the information needed to emulate the instruction. An implementation may at any time reduce its effort by substituting zero in place of the transformed instruction.

On an interrupt, the value written to the trap instruction register is always zero. On a synchronous exception, if a nonzero value is written, one of the following shall be true about the value:

- Bit 0 is `1`, and replacing bit 1 with `1` makes the value into a valid encoding of a standard instruction.

  In this case, the instruction that trapped is the same kind as indicated by the register value, and the register value is the transformation of the trapping instruction, as defined later. For example, if bits 1:0 are binary `11` and the register value is the encoding of a standard LW (load word) instruction, then the trapping instruction is LW, and the register value is the transformation of the trapping LW instruction.

- Bit 0 is `1`, and replacing bit 1 with `1` makes the value into an instruction encoding that is explicitly

designated for a custom instruction (*not* an unused reserved encoding).

This is a *custom value*. The instruction that trapped is a non-standard instruction. The interpretation of a custom value is not otherwise specified by this standard.

- The value is one of the special pseudoinstructions defined later, all of which have bits 1:0 equal to `00`.

These three cases exclude a large number of other possible values, such as all those having bits 1:0 equal to binary `10`. A future standard or extension may define additional cases, thus allowing values that are currently excluded. Software may safely treat an unrecognized value in a trap instruction register the same as zero.

To be forward-compatible with future revisions of this standard, software that interprets a nonzero value from `mtinst` *or* `htinst` *must fully verify that the value conforms to one of the cases listed above. For instance, for RV64, discovering that bits 6:0 of* `mtinst` *are* `0000011` *and bits 14:12 are* `010` *is not sufficient to establish that the first case applies and the trapping instruction is a standard LW instruction; rather, software must also confirm that bits 63:32 of* `mtinst` *are all zeros. A future standard might define new values for 64-bit* `mtinst` *that are nonzero in bits 63:32 yet may coincidentally have in bits 31:0 the same bit patterns as standard RV64 instructions.*

Unlike for standard instructions, there is no requirement that the instruction encoding of a custom value be of the same `kind" as the instruction that trapped (or even have any correlation with the trapping instruction).

Table 38 shows the values that may be automatically written to the trap instruction register for each standard exception cause. For exceptions that prevent the fetching of an instruction, only zero or a pseudoinstruction value may be written. A custom value may be automatically written only if the instruction that traps is non-standard. A future standard or extension may permit other values to be written, chosen from the set of allowed values established earlier.

Table 38. Values that may be automatically written to the trap instruction (`mtinst` or `htinst`) register on an exception trap.

Exception	Zero	Transformed Standard Instruction	Custom Value	Pseudoinstruction Value
Instruction address misaligned	Yes	No	Yes	No
Instruction access fault	Yes	No	No	No
Illegal instruction	Yes	No	No	No
Breakpoint	Yes	No	Yes	No
Virtual instruction	Yes	No	Yes	No
Load address misaligned	Yes	Yes	Yes	No
Load access fault	Yes	Yes	Yes	No
Store/AMO address misaligned	Yes	Yes	Yes	No
Store/AMO access fault	Yes	Yes	Yes	No
Environment call	Yes	No	Yes	No
Instruction page fault	Yes	No	No	No
Load page fault	Yes	Yes	Yes	No
Store/AMO page fault	Yes	Yes	Yes	No
Instruction guest-page fault	Yes	No	No	Yes
Load guest-page fault	Yes	Yes	Yes	Yes
Store/AMO guest-page fault	Yes	Yes	Yes	Yes

As enumerated in the table, a synchronous exception may write to the trap instruction register a standard transformation of the trapping instruction only for exceptions that arise from explicit memory accesses (from loads, stores, and AMO instructions). Accordingly, standard transformations are currently defined only for these memory-access instructions. If a synchronous trap occurs for a standard instruction for which no transformation has been defined, the trap instruction register shall be written with zero (or, under certain circumstances, with a special pseudoinstruction value).

For a standard load instruction that is not a compressed instruction and is one of LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, FLQ, or FLH, the transformed instruction has the format shown in Figure 112.

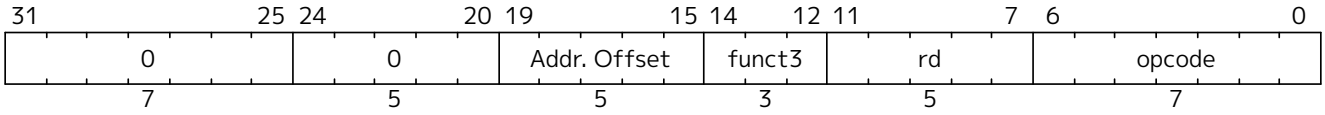


Figure 112. Transformed noncompressed load instruction (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, FLQ, or FLH). Fields funct3, rd, and opcode are the same as the trapping load instruction.

For a standard store instruction that is not a compressed instruction and is one of SB, SH, SW, SD, FSW, FSD, FSQ, or FSH, the transformed instruction has the format shown in Figure 113.

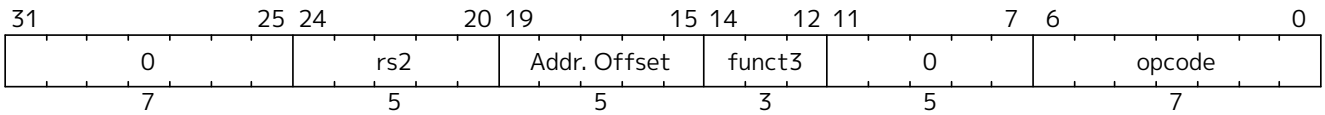


Figure 113. Transformed noncompressed store instruction (SB, SH, SW, SD, FSW, FSD, FSQ, or FSH). Fields rs2, funct3, and opcode are the same as the trapping store instruction.

For a standard atomic instruction (load-reserved, store-conditional, or AMO instruction), the transformed instruction has the format shown in Figure 114.

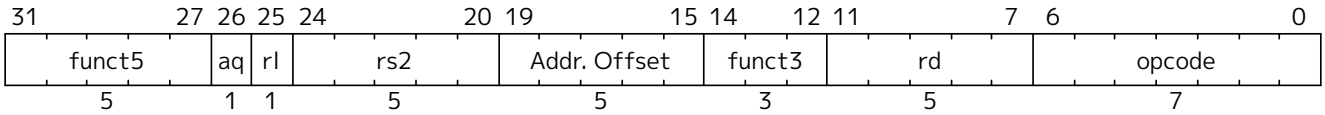


Figure 114. Transformed atomic instruction (load-reserved, store-conditional, or AMO instruc-tion). All fields are the same as the trapping instruction except bits 19:15, Addr. Offset.

For a standard virtual-machine load/store instruction (HLV, HLVX, or HSV), the transformed instruction has the format shown in Figure 115.

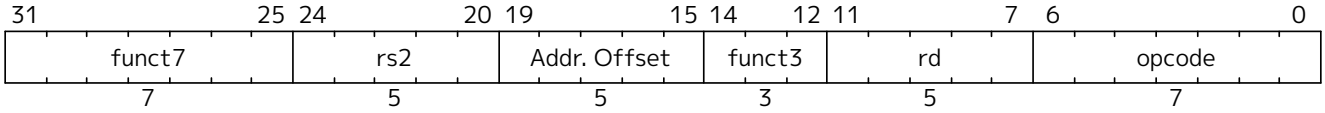


Figure 115. Transformed virtual-machine load/store instruction (HLV, HLVX, HSV). All fields are the same as the trapping instruction except bits 19:15, Addr. Offset

In all the transformed instructions above, the Addr. Offset field that replaces the instruction's rs1 field in bits 19:15 is the positive difference between the faulting virtual address (written to `mtval` or `stval`) and the original virtual address. This difference can be nonzero only for a misaligned memory access. Note also that, for basic loads and stores, the transformations replace the instruction's immediate offset fields with zero.

For a standard compressed instruction (16-bit size), the transformed instruction is found as follows:

1. Expand the compressed instruction to its 32-bit equivalent.
2. Transform the 32-bit equivalent instruction.
3. Replace bit 1 with a `0`.

Bits 1:0 of a transformed standard instruction will be binary `01` if the trapping instruction is compressed and `11` if not.

In decoding the contents of `mtinst` or `htinst`, once software has determined that the register contains the encoding of a standard basic load (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, FLQ, or FLH) or basic store (SB, SH, SW, SD, FSW, FSD, FSQ, or FSH), it is not necessary to confirm also that the immediate offset fields (31:25, and 24:20 or 11:7) are zeros. The knowledge that the register's value is the encoding of a basic load/store is sufficient to prove that the trapping instruction is of the same kind.

A future version of this standard may add information to the fields that are currently zeros. However, for backwards compatibility, any such information will be for performance purposes only and can safely be ignored.

For guest-page faults, the trap instruction register is written with a special pseudoinstruction value if: (a) the fault is caused by an implicit memory access for VS-stage address translation, and (b) a nonzero value (the faulting guest physical address) is written to `mtval2` or `htval`. If both conditions are met, the value written to `mtinst` or `htinst` must be taken from Table 39; zero is not allowed.

Table 39. Special pseudoinstruction values for guest-page faults. The RV32 values are used when VSXLEN=32, and the RV64 values when VSXLEN=64.

Value	Meaning
`0x00002000` `0x00002020`	32-bit read for VS-stage address translation (RV32) 32-bit write for VS-stage address translation (RV32)
`0x00003000` `0x00003020`	64-bit read for VS-stage address translation (RV64) 64-bit write for VS-stage address translation (RV64)

The defined pseudoinstruction values are designed to correspond closely with the encodings of basic loads and stores, as illustrated by Table 40.

Table 40. Standard instructions corresponding to the special pseudoinstructions of Table 39.

Encoding	Instruction
`0x00002003` `0x00002023`	`lw x0,0(x0)` `sw x0,0(x0)`
`0x00003003` `0x00003023`	`ld x0,0(x0)` `sd x0,0(x0)`

A *write* pseudoinstruction (`0x00002020` or `0x00003020`) is used for the case that the machine is attempting automatically to update bits A and/or D in VS-level page tables. All other implicit memory accesses for VS-stage address translation will be reads. If a machine never automatically updates bits A or D in VS-level page tables (leaving this to software), the *write* case will never arise. The fact that such a page table update must actually be atomic, not just a simple write, is ignored for the pseudoinstruction.

> *If the conditions that necessitate a pseudoinstruction value can ever occur for M-mode, then* `mtinst` *cannot be entirely read-only zero; and likewise for HS-mode and* `htinst`*. However, in that case, the trap instruction registers may minimally support only values 0 and* `0x00002000` *or* `0x00003000`*, and possibly* `0x00002020` *or* `0x00003020`*, requiring as few as one or two flip-flops in hardware, per register.*

> *There is no harm here in ignoring the atomicity requirement for page table updates, because a hypervisor is not expected in these circumstances to emulate an implicit memory access that fails. Rather, the hypervisor is given enough information about the faulting access to be able to make the memory accessible (e.g. by restoring a missing page of virtual memory) before resuming execution by retrying the faulting instruction.*

19.6.4. Trap Return

The MRET instruction is used to return from a trap taken into M-mode. MRET first determines what the new privilege mode will be according to the values of MPP and MPV in `mstatus` or `mstatush`, as encoded in Table 35. MRET then in `mstatus`/`mstatush` sets MPV=0, MPP=0, MIE=MPIE, and MPIE=1. Lastly, MRET sets the privilege mode as previously determined, and sets `pc`=`mepc`.

The SRET instruction is used to return from a trap taken into HS-mode or VS-mode. Its behavior depends on the current virtualization mode.

When executed in M-mode or HS-mode (i.e., V=0), SRET first determines what the new privilege mode will be according to the values in `hstatus`.SPV and `sstatus`.SPP, as encoded in Table 36. SRET then sets `hstatus`.SPV=0, and in `sstatus` sets SPP=0, SIE=SPIE, and SPIE=1. Lastly, SRET sets the privilege mode as previously determined, and sets `pc`=`sepc`.

When executed in VS-mode (i.e., V=1), SRET sets the privilege mode according to Table 37, in `vsstatus` sets SPP=0, SIE=SPIE, and SPIE=1, and lastly sets `pc`=`vsepc`.

If the Ssdbltrp extension is implemented, when `SRET` is executed in HS-mode, if the new privilege mode is VU, the `SRET` instruction sets `vsstatus.SDT` to 0. When executed in VS-mode, `vsstatus.SDT` is set to 0.

Chapter 20. Control-flow Integrity (CFI)

Control-flow Integrity (CFI) capabilities help defend against Return-Oriented Programming (ROP) and Call/Jump-Oriented Programming (COP/JOP) style control-flow subversion attacks. The Zicfiss and Zicfilp extensions provide backward-edge and forward-edge control flow integrity respectively. Please see the Control-flow Integrity chapter of the Unprivileged ISA specification for further details on these CFI capabilities and the associated Unprivileged ISA.

20.1. Landing Pad (Zicfilp)

This section specifies the Privileged ISA for the Zicfilp extension.

20.1.1. Landing-Pad-Enabled (LPE) State

The term `xLPE` is used to determine if forward-edge CFI using landing pads provided by the Zicfilp extension is enabled at a privilege mode.

When S-mode is implemented, it is determined as follows:

Table 41. `xLPE` determination when S-mode is implemented

Privilege Mode	xLPE
M	`mseccfg.MLPE`
S or HS	`menvcfg.LPE`
VS	`henvcfg.LPE`
U or VU	`senvcfg.LPE`

When S-mode is not implemented, it is determined as follows:

Table 42. `xLPE` determination when S-mode is not implemented

Privilege Mode	xLPE
M	`mseccfg.MLPE`
U	`menvcfg.LPE`

The Zicfilp must be explicitly enabled for use at each privilege mode.

Programs compiled with the `LPAD` instruction continue to function correctly, but without forward-edge CFI protection, when the Zicfilp extension is not implemented or is not enabled.

20.1.2. Preserving Expected Landing Pad State on Traps

A trap may need to be delivered to the same or to a higher privilege mode upon completion of `JALR`/`C.JALR`/`C.JR`, but before the instruction at the target of indirect call/jump was decoded, due to:

- Asynchronous interrupts.
- Synchronous exceptions with priority higher than that of a software-check exception with `xtval` set to "landing pad fault (code=2)" (See Table 15 of Privileged Specification).

The software-check exception caused by Zicfilp has higher priority than an illegal-instruction exception but lower priority than instruction access-fault.

The software-check exception due to the instruction not being an `LPAD` instruction when `ELP` is `LP_EXPECTED` or an software-check exception caused by the `LPAD` instruction itself (See [LP_INST]) leads to a trap being delivered to the same or to a higher privilege mode.

In such cases, the `ELP` prior to the trap, the previous `ELP`, must be preserved by the trap delivery such that it can be restored on a return from the trap. To store the previous `ELP` state on trap delivery to M-mode, an `MPELP` bit is provided in the `mstatus` CSR. To store the previous `ELP` state on trap delivery to S/HS-mode, an `SPELP` bit is provided in the `mstatus` CSR. The `SPELP` bit in `mstatus` can be accessed through the `sstatus` CSR. To store the previous `ELP` state on traps to VS-mode, a `SPELP` bit is defined in the `vsstatus` (VS-modes version of `sstatus`). To store the previous `ELP` state on transition to Debug Mode, a `pelp` bit is defined in the `dcsr` register.

When a trap is taken into privilege mode `x`, the *`x`*`PELP` is set to `ELP` and `ELP` is set to `NO_LP_EXPECTED`.

An `MRET` or `SRET` instruction is used to return from a trap in M-mode or S-mode, respectively. When executing an *`x`*`RET` instruction, if the new privilege mode is `y`, then `ELP` is set to the value of *`x`*`PELP` if *`y`*`LPE` (see Section 20.1.1) is 1; otherwise, it is set to `NO_LP_EXPECTED`; *`x`*`PELP` is set to `NO_LP_EXPECTED`.

Upon entry into Debug Mode, the `pelp` bit in `dcsr` is updated with the `ELP` at the privilege level the hart was previously in, and the `ELP` is set to `NO_LP_EXPECTED`. When a hart resumes from Debug Mode, if the new privilege mode is `y`, then `ELP` is set to the value of `pelp` if *`y`*`LPE` (see Section 20.1.1) is 1; otherwise, it is set to `NO_LP_EXPECTED`.

See also Chapter 8 for semantics added to the RNMI trap and the MNRET instruction when this extension is implemented.

The trap handler in privilege mode `x` *must save the* *`x`*`PELP` *bit and the* `x7` *register before performing an indirect call/jump if* `xLPE=1`*. If the privilege mode* `x` *can respond to interrupts and* `xLPE=1`*, then the trap handler should also save these values before enabling interrupts.*

The trap handler in privilege mode `x` *must restore the saved* *`x`*`PELP` *bit and the* `x7` *register before executing the* *`x`*`RET` *instruction to return from a trap.*

20.2. Shadow Stack (Zicfiss)

This section specifies the Privileged ISA for the Zicfiss extension.

20.2.1. Shadow Stack Pointer (ssp) CSR access control

Attempts to access the `ssp` CSR may result in either an illegal-instruction exception or a virtual instruction exception, contingent upon the state of the *x*`envcfg.SSE` fields. The conditions are specified as follows:

- If the privilege mode is less than M and `menvcfg.SSE` is 0, an illegal-instruction exception is raised.
- Otherwise, if in U-mode and `senvcfg.SSE` is 0, an illegal-instruction exception is raised.
- Otherwise, if in VS-mode and `henvcfg.SSE` is 0, a virtual instruction exception is raised.
- Otherwise, if in VU-mode and either `henvcfg.SSE` or `senvcfg.SSE` is 0, a virtual instruction exception is raised.
- Otherwise, the access is allowed.

20.2.2. Shadow-Stack-Enabled (SSE) State

The term `xSSE` is used to determine if backward-edge CFI using shadow stacks provided by the Zicfiss extension is enabled at a privilege mode.

When S-mode is implemented, it is determined as follows:

Table 43. `xSSE` *determination when S-mode is implemented*

Privilege Mode	xSSE
M	0
S or HS	menvcfg.SSE
VS	henvcfg.SSE
U or VU	senvcfg.SSE

When S-mode is not implemented, then `xSSE` is 0 at both M and U privilege modes.

Activating Zicfiss in U-mode must be done explicitly per process. Not activating Zicfiss at U-mode for a process when that application is not compiled with Zicfiss allows it to invoke shared libraries that may contain Zicfiss instructions. The Zicfiss instructions in the shared library revert to their Zimop/Zcmop-defined behavior in this case.

When Zicfiss is enabled in S-mode it is benign to use an operating system that is not compiled with Zicfiss instructions. Such an operating system that does not use backward-edge CFI for S-mode execution may still activate Zicfiss for U-mode applications.

When programs that use Zicfiss instructions are installed on a processor that supports the Zicfiss extension but the extension is not enabled at the privilege mode where the program executes, the program continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.

When programs that use Zicfiss instructions are installed on a processor that does not support the Zicfiss extension but supports the Zimop and Zcmop extensions, the programs continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.

On processors that do not support Zimop/Zcmop extensions, all Zimop/Zcmop code points including those used for Zicfiss instructions may cause an illegal-instruction exception. Execution of programs that use these instructions on such machines is not supported.

Activating Zicfiss in M-mode is currently not supported. Additionally, when S-mode is not implemented, activation in U-mode is also not supported. These functionalities may be introduced in a future standard extension.

Changes to `xSSE` *take effect immediately; address-translation caches need not be synchronized with SFENCE.VMA, HFENCE.GVMA, or HFENCE.VVMA instructions.*

20.2.3. Shadow Stack Memory Protection

To protect shadow stack memory, the memory is associated with a new page type – the Shadow Stack (SS) page – in the single-stage and VS-stage page tables. The encoding `R=0`, `W=1`, and `X=0`, is defined to represent an SS page. When `menvcfg.SSE=0`, this encoding remains reserved. Similarly, when `V=1` and `henvcfg.SSE=0`, this encoding remains reserved at `VS` and `VU` levels.

If `satp.MODE` (or `vsatp.MODE` when `V=1`) is set to `Bare` and the effective privilege mode is below M, shadow stack memory accesses are prohibited, and shadow stack instructions will raise a store/AMO access-fault exception. When the effective privilege mode is M, any memory access by an `SSAMOSWAP.W/D` instruction will result in a store/AMO access-fault exception.

Memory mapped as an SS page cannot be written to by instructions other than `SSAMOSWAP.W/D`, `SSPUSH`, and `C.SSPUSH`. Attempts will raise a store/AMO access-fault exception. Access to a SS page using *cache-block operation* (`CBO.*`) instructions is not permitted. Such accesses will raise a store/AMO access-fault exception. Implicit accesses, including instruction fetches to an SS page, are not permitted. Such accesses will raise an access-fault exception appropriate to the access type. However, the shadow stack is readable by all instructions that only load from memory.

Stores to shadow stack pages by instructions other than `SSAMOSWAP`*,* `SSPUSH`*, and* `C.SSPUSH` *will trigger a store/AMO access-fault exception, not a store/AMO page-fault exception, signaling a fatal error. A store/AMO page-fault suggests that the operating system could address and rectify the fault, which is not feasible in this scenario. Hence, the page fault handler must decode the opcode of the faulting instruction to discern whether the fault was caused by a non-shadow-stack instruction writing to an SS page (a fatal condition) or by a shadow stack instruction to a non-resident page (a recoverable condition). The performance-critical nature of operating system page fault handlers necessitates triggering an access-fault instead of a page fault, allowing for a straightforward distinction between fatal conditions and recoverable faults.*

Operating systems must ensure that no writable, non-shadow-stack alias virtual address mappings exist for the physical memory backing the shadow stack. Furthermore, in systems where an address-misaligned exception supersedes the access-fault exception, handlers emulating misaligned stores must be designed to cause an access-fault exception when the store is directed to a shadow stack page.

All instructions that perform load operations are allowed to read from the shadow stack. This feature facilitates debugging and performance profiling by allowing examination of the link register values backed up in the shadow stack.

As of the drafting of this specification, instruction fetches are the sole type of implicit access subjected to single- or VS-stage address translation.

If a shadow stack (SS) instruction raises an access-fault, page-fault, or guest-page-fault exception that is supposed to indicate the original instruction type (load or store/AMO), then the reported exception cause is respectively a store/AMO access fault (code 7), a store/AMO page fault (code 15), or a store/AMO guest-page fault (code 23). For shadow stack instructions, the reported instruction type is always as though it were a store or AMO, even for instructions `SSPOPCHK` and `C.SSPOPCHK` that only read from memory and do not write to it.

When Zicfiss is implemented, the existing "store/AMO" exceptions can be thought of as "store/AMO/SS" exceptions, indicating that the trapping instruction is either a store, an AMO, or a shadow stack instruction.

Shadow stack instructions are restricted to accessing shadow stack (`pte.xwr=010b`) pages. Should a shadow stack instruction access a page that is not designated as a shadow stack page and is not marked as read-only (`pte.xwr=001`), a store/AMO access-fault exception will be invoked. Conversely, if the page being accessed by a shadow stack instruction is a read-only page, a store/AMO page-fault exception will be triggered.

Shadow stack loads and stores will trigger a store/AMO page-fault if the accessed page is read-only, to support copy-on-write (COW) of a shadow stack page. If the page has been marked read-only for COW tracking, the page fault handler responds by creating a copy of the page and updates the `pte.xwr` *to* `010b`*, thereby designating each copy as a shadow stack page. Conversely, if the access targets a genuinely read-only page, the fault being reported as a store/AMO page-fault signals to the operating system that the fault is fatal and non-recoverable. Reporting the fault as a store/AMO page-fault, even for* `SSPOPCHK` *initiated memory access, aids in the determination of fatality; if these were reported as load page-faults, access to a truly read-only page might be mistakenly treated as a recoverable fault, leading to the faulting instruction being retried indefinitely. The PTE does not provide a read-only shadow stack encoding.*

Attempts by shadow stack instructions to access pages marked as read-write, read-write-execute, read-execute, or execute-only result in a store/AMO access-fault exception, similarly indicating a fatal condition.

Shadow stacks should be bounded at each end by guard pages to prevent accidental underflows or overflows from one shadow stack into another. Conventionally, a guard page for a stack is a page that is not accessible by the process that owns the stack.

If the virtual address in `ssp` is not `XLEN` aligned, then the `SSPUSH`/ `C.SSPUSH`/`SSPOPCHK`/`C.SSPOPCHK` instructions cause a store/AMO access-fault exception.

Misaligned accesses to shadow stack are not required and enforcing alignment is more secure to detect errors in the program. An access-fault exception is raised instead of address-misaligned exception in such cases to indicate fatality and that the instruction must not be emulated by a trap handler.

Correct execution of shadow stack instructions that access memory requires the the accessed memory to be idempotent. If the memory referenced by `SSPUSH`/`C.SSPUSH`/`SSPOPCHK`/`C.SSPOPCHK`/`SSAMOSWAP.W/D` instructions is not idempotent, then the instructions cause a store/AMO access-fault exception.

The `SSPOPCHK` instruction performs a load followed by a check of the loaded data value with the link register as source. If the check against the link register faults, and the instruction is restarted by the trap handler, then the instruction will perform a load again. If the memory from which the load is performed is non-idempotent, then the second load may cause unexpected side effects. Shadow stack instructions that access the shadow stack require the memory referenced by `ssp` to be idempotent to avoid such concerns. Locating shadow stacks in non-idempotent memory, such as non-idempotent device memory, is not an expected usage, and requiring memory referenced to be idempotent does not pose a significant restriction.

The `U` and `SUM` bit enforcement is performed normally for shadow stack instruction initiated memory accesses. The state of the `MXR` bit does not affect read access to a shadow stack page as the shadow stack page is always readable by all instructions that load from memory.

The G-stage address translation and protections remain unaffected by the Zicfiss extension. The `xwr == 010b` encoding in the G-stage PTE remains reserved. When G-stage page tables are active, the shadow stack instructions that access memory require the G-stage page table to have read-write permission for the accessed memory; else a store/AMO guest-page fault exception is raised.

A future extension may define a shadow stack encoding in the G-stage page table to support use cases such as a hypervisor enforcing shadow stack protections for its guests.

Svpbmt and Svnapot extensions are supported for shadow stack pages.

The PMA checks are extended to require memory referenced by shadow stack instructions to be idempotent. The PMP checks are extended to require read-write permission for memory accessed by shadow stack instructions. If the PMP does not provide read-write permissions or if the accessed memory is not idempotent then a store/AMO access-fault exception is raised.

The `SSAMOSWAP.W/D` instructions require the PMA of the accessed memory range to provide AMOSwap level support.

Chapter 21. "Ssdbltrp" Double Trap Extension, Version 1.0

The Ssdbltrp extension addresses a double trap (See Section 3.1.6.2) privilege modes lower than M. It enables HS-mode to invoke a critical error handler in a virtual machine on a double trap in VS-mode. It also allows M-mode to invoke a critical error handler in the OS/Hypervisor on a double trap in S/HS-mode.

The Ssdbltrp extension adds the `menvcfg`.DTE (See Section 3.1.18) and the `sstatus`.SDT fields (See Section 11.1.1). If the hypervisor extension is additionally implemented, then the extension adds the `henvcfg`.DTE (See Section 19.2.5) and the `vsstatus`.SDT fields (See Section 19.2.11).

See Section 11.1.1.5 for the operational details.

Chapter 22. Pointer Masking Extensions, Version 1.0.0

22.1. Introduction

RISC-V Pointer Masking (PM) is a feature that, when enabled, causes the CPU to ignore the upper bits of the effective address (these terms will be defined more precisely in the Background section). This allows these bits to be used in whichever way the application chooses. The version of the extension being described here specifically targets **tag checks**: When an address is accessed, the tag stored in the masked bits can be compared against a range-based tag. This is used for dynamic safety checkers such as HWASAN (Serebryany et al., 2018). Such tools can be applied in all privilege modes (U, S and M).

HWASAN leverages tags in the upper bits of the address to identify memory errors such as use-after-free or buffer overflow errors. By storing a **pointer tag** in the upper bits of the address and checking it against a **memory tag** stored in a side table, it can identify whether a pointer is pointing to a valid location. Doing this without hardware support introduces significant overheads since the pointer tag needs to be manually removed for every conventional memory operation. Pointer masking support reduces these overheads.

Pointer masking only adds the ability to ignore pointer tags during regular memory accesses. The tag checks themselves can be implemented in software or hardware. If implemented in software, pointer masking still provides performance benefits since non-checked accesses do not need to transform the address before every memory access. Hardware implementations are expected to provide even larger benefits due to performing tag checks out-of-band and hardening security guarantees derived from these checks. We anticipate that future extensions may build on pointer masking to support this functionality in hardware.

It is worth mentioning that while HWASAN is the primary use-case for the current pointer masking extension, a number of other hardware/software features may be implemented leveraging Pointer Masking. Some of these use cases include sandboxing, object type checks and garbage collection bits in runtime systems. Note that the current version of the spec does not explicitly address these use cases, but future extensions may build on it to do so.

While we describe the high-level concepts of pointer masking as if it was a single extension, it is, in reality, a family of extensions that implementations or profiles may choose to individually include or exclude (see Section 22.2.7).

22.2. Background

22.2.1. Definitions

We now define basic terms. Note that these rely on the definition of an “ignore” transformation, which is defined in Chapter 2.2.

- **Effective address (as defined in the RISC-V Base ISA):** A load/store effective address sent to the memory subsystem (e.g., as generated during the execution of load/store instructions). This does not include addresses corresponding to implicit accesses, such as page table walks.
- **Masked bits:** The upper PMLEN bits of an address, where PMLEN is a configurable parameter. We will use PMLEN consistently throughout this document to refer to this parameter.
- **Transformed address:** An effective address after the ignore transformation has been applied.
- **Address translation mode:** The MODE of the currently active address translation scheme as defined in the RISC-V privileged specification. This could, for example, refer to Bare, Sv39, Sv48, and Sv57. In accordance with the privileged specification, non-Bare translation modes are referred to as virtual-

memory schemes. For the purpose of this specification, M-mode translation is treated as equivalent to Bare.

- **Address validity:** The RISC-V privileged spec defines validity of addresses based on the address translation mode that is currently in use (e.g., Sv57, Sv48, Sv39, etc.). For a virtual address to be valid, all bits in the unused portion of the address must be the same as the Most Significant Bit (MSB) of the used portion. For example, when page-based 48-bit virtual memory (Sv48) is used, load/store effective addresses, which are 64 bits, must have bits 63–48 all set to bit 47, or else a page-fault exception will occur. For physical addresses, validity means that bits XLEN-1 to PABITS are zero, where PABITS is the number of physical address bits supported by the processor.
- **NVBITS:** The upper bits within a virtual address that have no effect on addressing memory and are only used for validity checks. These bits depend on the currently active address translation mode. For example, in Sv48, these are bits 63-48.
- **VBITS:** The bits within a virtual address that affect which memory is addressed. These are the bits of an address which are used to index into page tables.

22.2.2. The “Ignore” Transformation

The ignore transformation differs depending on whether it applies to a virtual or physical address. For virtual addresses, it replaces the upper PMLEN bits with the sign extension of the PMLEN+1st bit.

```
transformed_effective_address =
  {{PMLEN{effective_address[XLEN-PMLEN-1]}}, effective_address[XLEN-PMLEN-1:0]}
```

Listing 1. "Ignore" Transformation for virtual addresses, expressed in Verilog code.

If PMLEN is less than or equal to NVBITS for the largest supported address translation mode on a given architecture, this is equivalent to ignoring a subset of NVBITS. This enables cheap implementations that modify validity checks in the CPU instead of performing the sign extension.

When applied to a physical address, including guest-physical addresses (i.e., all cases except when the active satp register’s MODE field != Bare), the ignore transformation replaces the upper PMLEN bits with 0. This includes both the case of running in M-mode and running in other privilege modes with Bare address translation mode.

```
transformed_effective_address =
  {{PMLEN{0}}, effective_address[XLEN-PMLEN-1:0]}
```

Listing 2. "Ignore" Transformation for physical addresses, expressed in Verilog code.

This definition is consistent with the way that RISC-V already handles physical and virtual addresses differently. While the unused upper bits of virtual addresses are the sign-extension of the used bits (see the definition of "address validity" in Section 22.2.1), the equivalent bits in physical addresses are zero-extended. This is necessary due to their interactions with other mechanisms such as Physical Memory Protection (PMP).

When pointer masking is enabled, the ignore transformation will be applied to every explicit memory access (e.g., loads/stores, atomics operations, and floating point loads/stores). The transformation **does not** apply to implicit accesses such as page table walks or instruction fetches. The set of accesses that pointer masking applies to is described in Section 22.2.6.

Pointer masking does not change the underlying address generation logic or permission checks. Under a fixed address translation mode, it is semantically equivalent to replacing a subset of instructions (e.g., loads and stores) with an instruction sequence that applies the ignore operation to the target address of this instruction and then applies the instruction to the transformed address. References to address translation and other implementation details in the text are primarily to explain design decisions and common implementation patterns.

Note that pointer masking is purely an arithmetic operation on the address that makes no assumption about the meaning of the addresses it is applied to. Pointer masking with the same value of PMLEN always has the same effect for the same type of address (virtual or physical). This ensures that code that relies on pointer masking does not need to be aware of the environment it runs in once pointer masking has been enabled, as long as the value of PMLEN is known, and whether or not addresses are virtual or physical. For example, the same application or library code can run in user mode, supervisor mode or M-mode (with different address translation modes) without modification.

A common scenario for such code is that addresses are generated by mmap system calls. This abstracts away the details of the underlying address translation mode from the application code. Software therefore needs to be aware of the value of PMLEN to ensure that its minimally required number of tag bits is supported. Section 22.2.4 covers how this value is derived.

22.2.3. Example

Table 1 shows an example of the pointer masking transformation on a virtual address when PM is enabled for RV64 under Sv57 (PMLEN=7).

Table 44. Example of PM address translation for RV64 under Sv57

Page-based profile	Sv57 on RV64
Effective Address	0xABFFFFFF12345678 NVBITS[1010101] VBITS[1111111111111111111111110001...000]
PMLEN	7
Mask	0x01FFFFFFFFFFFFFF NVBITS[0000000] VBITS[1111111111111111111111111111...111]
PMLEN+1st bit from the top (i.e., bit XLEN-PMLEN-1)	1
Transformed effective address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[1111111111111111111111110001...000]

If the address was a physical address rather than a virtual address with Sv57, the transformed address with PMLEN=7 would be 0x1FFFFFF12345678.

22.2.4. Determining the Value of PMLEN

From an implementation perspective, ignoring bits is deeply connected to the maximum virtual and physical address space supported by the processor (e.g., Bare, Sv48, Sv57). In particular, applying the above transformation is cheap if it covers only bits that are not used by **any** supported address translation mode (as it is equivalent to switching off validity checks). Masking NVBITS beyond those bits is more expensive as it requires ignoring them in the TLB tag, and even more expensive if the masked bits extend into the VBITS portion of the address (as it requires performing the actual sign extension). Similarly, when running in Bare or M mode, it is common for implementations to not use a particular number of bits at the top of the physical address range and fix them to zero. Applying the ignore transformation to those bits is cheap as well, since it will result in a valid physical address with all the upper bits fixed to 0.

The current standard only supports PMLEN=XLEN-48 (i.e., PMLEN=16 in RV64) and PMLEN=XLEN-57 (i.e., PMLEN=7 in RV64). A setting has been reserved to potentially support other values of PMLEN in future standards. In such future standards, different supported values of PMLEN may be defined for each privilege mode (U/VU, S/HS, and M).

Future versions of the pointer masking extension may introduce the ability to freely configure the value of PMLEN. The current extension does not define the behavior if PMLEN was different from the values defined above. In particular, there is no guarantee that a future pointer masking extension would define the ignore operation in the same way for those values of PMLEN.

22.2.5. Pointer Masking and Privilege Modes

Pointer masking is controlled separately for different privilege modes. The subset of supported privilege modes is determined by the set of supported pointer masking extensions. Different privilege modes may have different pointer masking settings active simultaneously and the hardware will automatically apply the pointer masking settings of the currently active privilege mode. A privilege mode's pointer masking setting is configured by bits in configuration registers of the next-higher privilege mode.

Note that the pointer masking setting that is applied only depends on the active privilege mode, not on the address that is being masked. Some operating systems (e.g., Linux) may use certain bits in the address to disambiguate between different types of addresses (e.g., kernel and user-mode addresses). Pointer masking *does not* take these semantics into account and is purely an arithmetic operation on the address it is given.

Linux places kernel addresses in the upper half of the address space and user addresses in the lower half of the address space. As such, the MSB is often used to identify the type of a particular address. With pointer masking enabled, this role is now played by bit XLEN-PMLEN-1 and code that checks whether a pointer is a kernel or a user address needs to inspect this bit instead. For backward compatibility, it may be desirable that the MSB still indicates whether an address is a user or a kernel address. An operating system's ABI may mandate this, but it does not affect the pointer masking mechanism itself. For example, the Linux ABI may choose to mandate that the MSB is not used for tagging and replicates bit XLEN-PMLEN-1 bit (note that for such a mechanism to be secure, the kernel needs to check the MSB of any user mode-supplied address and ensure that this invariant holds before using it; alternatively, it can apply the transformation from Listing 1 or 2 to ensure that the MSB is set to the correct value).

22.2.6. Memory Accesses Subject to Pointer Masking

Pointer masking applies to all explicit memory accesses. Currently, in the Base and Privileged ISAs, these are:

- **Base Instruction Set**: LB, LH, LW, LBU, LHU, LWU, LD, SB, SH, SW, SD.
- **Atomics**: All instructions in RV32A and RV64A.
- **Floating Point**: FLW, FLD, FLQ, FSW, FSD, FSQ.
- **Compressed**: All instructions mapping to any of the above, and C.LWSP, C.LDSP, C.LQSP, C.FLWSP, C.FLDSP, C.SWSP, C.SDSP, C.SQSP, C.FSWSP, C.FSDSP.
- **Hypervisor Extension**: HLV.*, HSV.* (in some cases; see Section 22.3.1).
- **Cache Management Operations**: All instructions in Zicbom, Zicbop and Zicboz.
- **Vector Extension**: All vector load and store instructions in the ratified RVV 1.0 spec.

- **Zicfiss Extension**: SSPUSH, C.SSPUSH, SSPOPCHK, C.SSPOPCHK, SSAMOSWAP.W/D.
- **Assorted**: FENCE, FENCE.I (if the currently unused address fields become enabled in the future).

This list will grow over time as new extensions introduce new instructions that perform explicit memory accesses.

For other extensions, pointer masking applies to all explicit memory accesses by default. Future extensions may add specific language to indicate whether particular accesses are or are not included in pointer masking.

It is worth noting that pointer masking is not applied to `SFENCE.*`, `HFENCE.*`, `SINVAL.*`, *or* `HINVAL.*`. *When such an operation is invoked, it is the responsibility of the software to provide the correct address.*

MPRV and SPVP affect pointer masking as well, causing the pointer masking settings of the effective privilege mode to be applied. When MXR is in effect at the effective privilege mode where explicit memory access is performed, pointer masking does not apply.

Note that this includes cases where page-based virtual memory is not in effect; i.e., although MXR has no effect on permissions checks when page-based virtual memory is not in effect, it is still used in determining whether or not pointer masking should be applied.

Cache Management Operations (CMOs) must respect and take into account pointer masking. Otherwise, a few serious security problems can appear, including:

- *CBO.ZERO may work as a STORE operation. If pointer masking is not respected, it would be possible to write to memory bypassing the mask enforcement.*
- *If CMOs did not respect pointer masking, it would be possible to weaponize this in a side-channel attack. For example, U-mode would be able to flush a physical address (without masking) that it should not be permitted to.*

Pointer masking only applies to accesses generated by instructions on the CPU (including CPU extensions such as an FPU). E.g., it does not apply to accesses generated by page table walks, the IOMMU, or devices.

Pointer Masking does not apply to DMA controllers and other devices. It is therefore the responsibility of the software to manually untag these addresses.

Misaligned accesses are supported, subject to the same limitations as in the absence of pointer masking. The behavior is identical to applying the pointer masking transformation to every constituent aligned memory access. In other words, the accessed bytes should be identical to the bytes that would be accessed if the pointer masking transformation was individually applied to every byte of the access without pointer masking. This ensures that both hardware implementations and emulation of misaligned accesses in M-mode behave the same way, and that the M-mode implementation is identical whether or not pointer masking is enabled (e.g., such an implementation may leverage MPRV to apply the correct privilege mode's pointer masking setting).

No pointer masking operations are applied when software reads/writes to CSRs, including those meant to hold addresses. If software stores tagged addresses into such CSRs, data load or data store operations based on those addresses are subject to pointer masking only if they are explicit (Section 22.2.6) and pointer masking is enabled for the privilege mode that performs the access. The implemented WARL width of CSRs is unaffected by pointer masking (e.g., if a CSR supports 52 bits of valid addresses and pointer masking is supported with PMLEN=16, the necessary number of WARL bits remains 52 independently of whether pointer masking is enabled or disabled).

In contrast to software writes, pointer masking **is applied** for hardware writes to a CSR (e.g., when the

hardware writes the transformed address to `stval` when taking an exception). Pointer masking is also applied to the memory access address when matching address triggers in debug.

For example, software is free to write a tagged or untagged address to `stvec`, but on trap delivery (e.g., due to an exception or interrupt), pointer masking **will not be applied** to the address of the trap handler. However, pointer masking **will be applied** by the hardware to any address written into `stval` when delivering an exception.

The rationale for this choice is that delivering the additional bits may add overheads in some hardware implementations. Further, pointer masking is configured per privilege mode, so all trap handlers in supervisor mode would need to be careful to configure pointer masking the same way as user mode or manually unmask (which is expensive).

22.2.7. Pointer Masking Extensions

Pointer masking refers to a number of separate extensions, all of which are privileged. This approach is used to capture optionality of pointer masking features. Profiles and implementations may choose to support an arbitrary subset of these extensions and must define valid ranges for their corresponding values of PMLEN.

Extensions:

- **Ssnpm**: A supervisor-level extension that provides pointer masking for the next lower privilege mode (U-mode), and for VS- and VU-modes if the H extension is present.
- **Smnpm**: A machine-level extension that provides pointer masking for the next lower privilege mode (S/HS if S-mode is implemented, or U-mode otherwise).
- **Smmpm**: A machine-level extension that provides pointer masking for M-mode.

See Section 22.3 for details on how each of these extensions is configured.

In addition, the pointer masking standard defines two extensions that describe an execution environment but have no bearing on hardware implementations. These extensions are intended to be used in profile specifications where a User profile or a Supervisor profile can only reference User level or Supervisor level pointer masking functionality, and not the associated CSR controls that exist at a higher privilege level (i.e., in the execution environment).

- **Sspm**: An extension that indicates that there is pointer-masking support available in supervisor mode, with some facility provided in the supervisor execution environment to control pointer masking.
- **Supm**: An extension that indicates that there is pointer-masking support available in user mode, with some facility provided in the application execution environment to control pointer masking.

The precise nature of these facilities is left to the respective execution environment.

Pointer masking only applies to RV64. In RV32, trying to enable pointer masking will result in an illegal WARL write and not update the pointer masking configuration bits (see Section 22.3 for details). The same is the case on RV64 or larger systems when UXL/SXL/MXL is set to 1 for the corresponding privilege mode. Note that in RV32, the CSR bits introduced by pointer masking are still present, for compatibility between RV32 and larger systems with UXL/SXL/MXL set to 1. Setting UXL/SXL/MXL to 1 will clear the corresponding pointer masking configuration bits.

Note that setting UXL/SXL/MXL to 1 and back to 0 does not preserve the previous values of the PMM bits. This includes the case of entering an RV32 virtual machine from an RV64 hypervisor and returning.

22.3. ISA Extensions

This section describes the pointer masking extensions `Smmpm`, `Smnpm` and `Ssnpm`. All of these extensions are privileged ISA extensions and do not add any new CSRs. For the definitions of `Sspm` and `Supm`, see Section 22.2.7.

Future extensions may introduce additional CSRs to allow different privilege modes to modify their own pointer masking settings. This may be required for future use cases in managed runtime systems that are not currently addressed as part of this extension.

Each extension introduces a 2-bit WARL field (`PMM`) that may take on the following values to set the pointer masking settings for a particular privilege mode.

Table 45. Possible values of `PMM` WARL field.

Value	Description
00	Pointer masking is disabled (PMLEN=0)
01	Reserved
10	Pointer masking is enabled with PMLEN=XLEN-57 (PMLEN=7 on RV64)
11	Pointer masking is enabled with PMLEN=XLEN-48 (PMLEN=16 on RV64)

All of these fields are read-only 0 on RV32 systems.

22.3.1. Ssnpm

`Ssnpm` adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `senvcfg`. Setting `PMM` enables or disables pointer masking for the next lower privilege mode (U/VU mode), according to the values in Table 2.

In systems where the H Extension is present, `Ssnpm` also adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `henvcfg`. Setting `PMM` enables or disables pointer masking for VS-mode, according to the values in Table 2. Further, a 2-bit WARL field (`HUPMM`) is added to bits 49:48 of `hstatus`. Setting `hstatus.HUPMM` enables or disables pointer masking for `HLV.*` and `HSV.*` instructions in U-mode, according to the values in Table 2, when their explicit memory access is performed as though in VU-mode. In HS- and M-modes, pointer masking for these instructions is enabled or disabled by `senvcfg.PMM`, when their explicit memory access is performed as though in VU-mode. Setting `henvcfg.PMM` enables or disables pointer masking for `HLV.*` and `HSV.*` when their explicit memory access is performed as though in VS-mode.

The hypervisor should copy the value written to `senvcfg.PMM` by the guest to the `hstatus.HUPMM` field prior to invoking `HLV.*` or `HSV.*` instructions in U-mode.

The memory accesses performed by the `HLVX.*` instructions are not subject to pointer masking.

`HLVX.*` instructions, designed for emulating implicit access to fetch instructions from guest memory, perform memory accesses that are exempt from pointer masking to facilitate this emulation. For the same reason, pointer masking does not apply when MXR is set.

22.3.2. Smnpm

`Smnpm` adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `menvcfg`. Setting `PMM` enables or disables pointer masking for the next lower privilege mode (S-/HS-mode if S-mode is implemented, or U-mode otherwise), according to the values in Table 2.

The type of address determines which type of pointer masking is applied. For example, when

> *running with virtualization in VS/VU mode with* `vsatp.MODE` *= Bare, physical address pointer masking (zero extension) applies.*

22.3.3. Smmpm

`Smmpm` adds a new 2-bit WARL field (`PMM`) to bits 33:32 of `mseccfg`. The presence of `Smmpm` implies the presence of the `mseccfg` register, even if it would not otherwise be present. Setting `PMM` enables or disables pointer masking for M mode, according to the values in Table 2.

22.3.4. Interaction with SFENCE.VMA

Since pointer masking applies to the effective address only and does not affect any memory-management data structures, no SFENCE.VMA is required after enabling/disabling pointer masking.

22.3.5. Interaction with Two-Stage Address Translation

Guest physical addresses (GPAs) are 2 bits wider than the corresponding virtual address translation modes, resulting in additional address translation schemes Sv32x4, Sv39x4, Sv48x4 and Sv57x4 for translating guest physical addresses to supervisor physical addresses. When running with virtualization in VS/VU mode with `vsatp.MODE` = Bare, this means that those two bits may be subject to pointer masking, depending on `hgatp.MODE` and `senvcfg.PMM`/`henvcfg.PMM` (for VU/VS mode). If `vsatp.MODE` != BARE, this issue does **not** apply.

> *An implementation could mask those two bits on the TLB access path, but this can have a significant timing impact. Alternatively, an implementation may choose to "waste" TLB capacity by having up to 4 duplicate entries for each page. In this case, the pointer masking operation can be applied on the TLB refill path, where it is unlikely to affect timing. To support this approach, some TLB entries need to be flushed when PMLEN changes in a way that may affect these duplicate entries.*

To support implementations where (XLEN-PMLEN) can be less than the GPA width supported by `hgatp.MODE`, hypervisors should execute an `HFENCE.GVMA` with rs1=x0 if the `henvcfg.PMM` is changed from or to a value where (XLEN-PMLEN) is less than GPA width supported by the `hgatp` translation mode of that guest. Specifically, these cases are:

- `PMLEN=7` and `hgatp.MODE=sv57x4`
- `PMLEN=16` and `hgatp.MODE=sv57x4`
- `PMLEN=16` and `hgatp.MODE=sv48x4`

> `Smmpm` *implementations need to satisfy max(largest supported virtual address size, largest supported supervisor physical address size)* ⇐ *(XLEN - PMLEN) bits to avoid any masking logic on the TLB access path.*

Implementation of an address-specific `HFENCE.GVMA` should either ignore the address argument, or should ignore the top masked GPA bits of entries when comparing for an address match.

22.3.6. Number of Masked Bits

As described in Section 22.2.4, the supported values of PMLEN may depend on the effective privilege mode. The current standard only defines PMLEN=XLEN-48 and PMLEN=XLEN-57, but this assumption may be relaxed in future extensions and profiles. Trying to enable pointer masking in an unsupported

scenario represents an illegal write to the corresponding pointer masking enable bit and follows WARL semantics. Future profiles may choose to define certain combinations of privilege modes and supported values of PMLEN as mandatory.

An option that was considered but discarded was to allow implementations to set PMLEN depending on the active addressing mode. For example, PMLEN could be set to 16 for Sv48 and to 25 for Sv39. However, having a single value of PMLEN (e.g., setting PMLEN to 16 for both Sv39 and Sv48 rather than 25) facilitates TLB implementations in designs that support Sv39 and Sv48 but not Sv57. 16 bits are sufficient for current pointer masking use cases but allow for a TLB implementation that matches against the same number of virtual tag bits independently of whether it is running with Sv39 or Sv48. However, if Sv57 is supported, tag matching may need to be conditional on the current address translation mode.

Chapter 23. RISC-V Privileged Instruction Set Listings

This chapter presents instruction-set listings for all instructions defined in the RISC-V Privileged Architecture.

The instruction-set listings for unprivileged instructions, including the ECALL and EBREAK instructions, are provided in Volume I of this manual.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type

Trap-Return Instructions

0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET
0111000	00010	00000	000	00000	1110011	MNRET

Interrupt-Management Instructions

0001000	00101	00000	000	00000	1110011	WFI

Supervisor Memory-Management Instructions

0001001	rs2	rs1	000	00000	1110011	SFENCE.VMA

Hypervisor Memory-Management Instructions

0010001	rs2	rs1	000	00000	1110011	HFENCE.VVMA
0110001	rs2	rs1	000	00000	1110011	HFENCE.GVMA

Hypervisor Virtual-Machine Load and Store Instructions

0110000	00000	rs1	100	rd	1110011	HLV.B
0110000	00001	rs1	100	rd	1110011	HLV.BU
0110010	00000	rs1	100	rd	1110011	HLV.H
0110010	00001	rs1	100	rd	1110011	HLV.HU
0110100	00000	rs1	100	rd	1110011	HLV.W
0110010	00011	rs1	100	rd	1110011	HLVX.HU
0110100	00011	rs1	100	rd	1110011	HLVX.WU
0110001	rs2	rs1	100	00000	1110011	HSV.B
0110011	rs2	rs1	100	00000	1110011	HSV.H
0110101	rs2	rs1	100	00000	1110011	HSV.W

Hypervisor Virtual-Machine Load and Store Instructions, RV64 only

0110100	00001	rs1	100	rd	1110011	HLV.WU
0110110	00000	rs1	100	rd	1110011	HLV.D
0110111	rs2	rs1	100	00000	1110011	HSV.D

Svinval Memory-Management Extension

0001011	rs2	rs1	000	00000	1110011	SINVAL.VMA
0001100	00000	00000	000	00000	1110011	SFENCE.W.INVAL
0001100	00001	00000	000	00000	1110011	SFENCE.INVAL.IR
0010011	rs2	rs1	000	00000	1110011	HINVAL.VVMA
0110011	rs2	rs1	000	00000	1110011	HINVAL.GVMA

Figure 116. RISC-V Privileged Instructions

Chapter 24. History

24.1. Research Funding at UC Berkeley

Development of the RISC-V architecture and implementations has been partially funded by the following sponsors.

- **Par Lab:** Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.
- **Project Isis:** DoE Award DE-SC0003624.
- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. DARPA POEM program Award HR0011-11-C-0100. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support from ASPIRE industrial sponsor, Intel, and ASPIRE affiliates, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

Bibliography

The RISC-V Debug Specification. github.com/riscv/riscv-debug-spec

Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, *7*(6), 34–45.

Navarro, J., Iyer, S., Druschel, P., & Cox, A. (2002). Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.*, *36*(SI), 89–104. doi.org/10.1145/844128.844138

Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyrklevich, V., & Vyukov, D. (2018). Memory Tagging and how it improves C/C++ memory safety. *CoRR*, *abs/1802.09517*. arxiv.org/abs/1802.09517