The RISC-V Instruction Set Manual
Volume II: Privileged Architecture (with Sail formal model)
Document Version 1.12-draft

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
`andrew@sifive.com, krste@berkeley.edu`
June 4, 2020

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, Dennis Ferguson, Marc Gauthier, Andy Glew, Gary Guo, Mike Frysinger, John Hauser, David Horner, Olof Johansson, David Kruckemyer, Yunsup Lee, Andrew Lutomirski, 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, Clifford Wolf, and Reinoud Zandijk.

Please cite as: "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, June 2019.

Preface

This version of the document is a draft containing annotations from the Sail formal model of RISC-V (git version ea09b9a). See Section 1.4.

This is **a draft of** version 1.12 of the RISC-V privileged architecture proposal. The document contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	*1.12*	*Draft*
Supervisor ISA	*1.12*	*Draft*
Hypervisor ISA	*0.6*	*Draft*
N Extension	*1.1*	*Draft*

The Machine and Supervisor ISAs, version 1.11, have been ratified by the RISC-V Foundation. Version 1.12 of these modules, described in this document, is a minor revision to version 1.11.

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 reset values are now platform-defined.
- An additional 48 optional PMP registers have been defined.

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

- Defined the RV32-only CSR `mstatush`, which contains most of the same fields as the upper 32 bits of RV64's `mstatus`.
- Permitted the unconditional delegation of less-privileged interrupts.
- Added optional big-endian and bi-endian support.

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.
- Software is strongly recommended to allocate ASIDs globally, so that a future extension can globalize ASIDs for improved performance and hardware flexibility.
- SFENCE.VMA semantics have been clarified.
- Made the `mstatus`.MPP field **WARL**, rather than **WLRL**.
- Made the unused *x*`ip` fields **WPRI**, rather than **WIRI**.
- Made the unused `misa` fields **WLRL**, 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`.*x*IE is written upon an exception.
- Described scheme for emulating misaligned AMOs.
- Specified the behavior of the `misa` and *x*`epc` 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 exceptions. 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.

Contents

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 1.1 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.

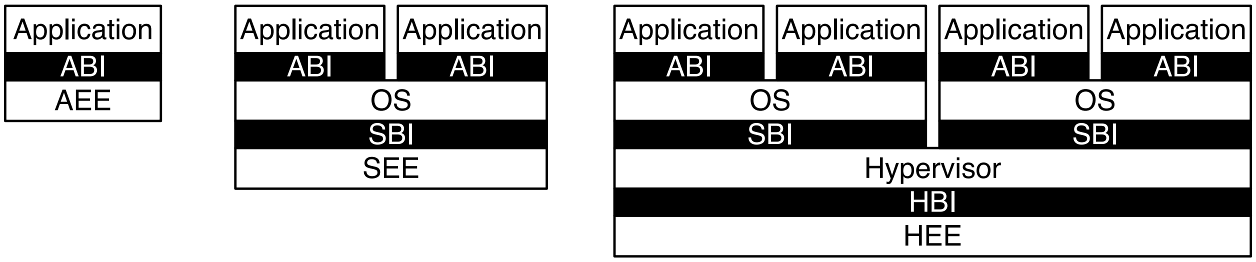

Figure 1.1: Different implementation stacks supporting various forms of privileged execution.

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 1.1.

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

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

Table 1.1: RISC-V privilege levels.

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 5.

Implementations might provide anywhere from 1 to 3 privilege modes trading off reduced isolation for lower implementation cost, as shown in Table 1.2.

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

Table 1.2: Supported combinations of privilege modes.

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.

1.4 Formal Specification in Sail

This version of the document includes selected sections of the formal description of privileged architecture as specified in the Sail ISA description language[1]. This description is generated directly from the Sail sources of the RISC-V model, available at `https://github.com/rems-project/sail-riscv`. More information on Sail is provided in the Sail manual provided with the Sail distribution at `https://github.com/rems-project/sail`.

The Sail model of RISC-V currently implements the IMAC subset of instructions for RV32 and RV64, the user, supervisor and machine privilege levels, and the Sv32, Sv39 and Sv48 address translation modes. It passes the `riscv-tests` test suite, the `riscv-compliance` suite, and is capable of booting Linux and seL4. The model currently does not model the floating point extensions, event and performance counters, PMP/PMA, and the hypervisor mode.

Some selected portions of the Sail model of the privileged architecture are included in an appendix of this volume on page 123, and include model definitions that were omitted from the unprivileged specification volume. The selected portions are not intended to be complete, but instead to highlight formalizations of some of the more involved aspects of the ISA. The complete Sail specification is available at the above url. It includes a reading guide as well as a guide to extending the model.

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), and all other privileged instructions. In addition to the user-level 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 the user-level manual. 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.

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 2.1. 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.

CSR handling in the Sail formal model is described in §A.2.

Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also 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 2.1 also indicates the convention to allocate CSR addresses between standard and custom uses. The CSR addresses reserved 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 [2]. 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

Tables 2.2–2.6 list the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are standard user-level CSRs, as well as the additional user trap registers added by the N extension. The other registers are used by privileged code, as described in the following chapters. Note that not all registers are required on all implementations.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
		User 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 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 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 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 2.1: Allocation of RISC-V CSR address ranges.

Number	Privilege	Name	Description
			User Trap Setup
`0x000`	URW	`ustatus`	User status register.
`0x004`	URW	`uie`	User interrupt-enable register.
`0x005`	URW	`utvec`	User trap handler base address.
			User Trap Handling
`0x040`	URW	`uscratch`	Scratch register for user trap handlers.
`0x041`	URW	`uepc`	User exception program counter.
`0x042`	URW	`ucause`	User trap cause.
`0x043`	URW	`utval`	User bad address or instruction.
`0x044`	URW	`uip`	User interrupt pending.
			User 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`).
			User 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 2.2: Currently allocated RISC-V user-level CSR addresses.

Number	Privilege	Name	Description
		Supervisor Trap Setup	
`0x100`	SRW	`sstatus`	Supervisor status register.
`0x102`	SRW	`sedeleg`	Supervisor exception delegation register.
`0x103`	SRW	`sideleg`	Supervisor interrupt delegation register.
`0x104`	SRW	`sie`	Supervisor interrupt-enable register.
`0x105`	SRW	`stvec`	Supervisor trap handler base address.
`0x106`	SRW	`scounteren`	Supervisor counter enable.
		Supervisor Trap Handling	
`0x140`	SRW	`sscratch`	Scratch register for supervisor trap handlers.
`0x141`	SRW	`sepc`	Supervisor exception program counter.
`0x142`	SRW	`scause`	Supervisor trap cause.
`0x143`	SRW	`stval`	Supervisor bad address or instruction.
`0x144`	SRW	`sip`	Supervisor interrupt pending.
		Supervisor Protection and Translation	
`0x180`	SRW	`satp`	Supervisor address translation and protection.

Table 2.3: Currently allocated RISC-V supervisor-level 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.
Hypervisor Trap Handling			
`0x643`	HRW	`htval`	Hypervisor bad guest physical address.
`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 Protection and Translation			
`0x680`	HRW	`hgatp`	Hypervisor guest address translation and protection.
Hypervisor Counter/Timer Virtualization Registers			
`0x605`	HRW	`htimedelta`	Delta for VS/VU-mode timer.
`0x615`	HRW	`htimedeltah`	Upper 32 bits of `htimedelta`, 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 bad address or instruction.
`0x244`	HRW	`vsip`	Virtual supervisor interrupt pending.
`0x280`	HRW	`vsatp`	Virtual supervisor address translation and protection.

Table 2.4: Currently allocated RISC-V hypervisor-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.
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.
Machine Trap Handling			
`0x340`	MRW	`mscratch`	Scratch register for machine trap handlers.
`0x341`	MRW	`mepc`	Machine exception program counter.
`0x342`	MRW	`mcause`	Machine trap cause.
`0x343`	MRW	`mtval`	Machine bad address or instruction.
`0x344`	MRW	`mip`	Machine interrupt pending.
`0x34A`	MRW	`mtinst`	Machine trap instruction (transformed).
`0x34B`	MRW	`mtval2`	Machine bad guest physical address.
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.

Table 2.5: Currently allocated RISC-V machine-level CSR addresses.

Number	Privilege	Name	Description
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.
		⋮	
`0xB9F`	MRW	`mhpmcounter31h`	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.
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.
Debug Mode Registers			
`0x7B0`	DRW	`dcsr`	Debug control and status register.
`0x7B1`	DRW	`dpc`	Debug PC.
`0x7B2`	DRW	`dscratch0`	Debug scratch register 0.
`0x7B3`	DRW	`dscratch1`	Debug scratch register 1.

Table 2.6: Currently allocated RISC-V machine-level CSR addresses.

2.3 CSR Field Specifications

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

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 hardwire them to 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.

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.

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 Width Modulation

If the width of a CSR is changed (for example, by changing MXLEN or UXLEN, as described in Section 3.1.6.2), 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.

Chapter 3

Machine-Level ISA, Version 1.12

This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V system. 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 Register `misa`

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.

MXLEN-1 MXLEN-2	MXLEN-3 26	25 0
MXL[1:0] (**WARL**)	***WLRL***	Extensions[25:0] (**WARL**)
2	MXLEN-28	26

Figure 3.1: Machine ISA register (`misa`).

The MXL (Machine XLEN) field encodes the native base integer ISA width as shown in Table 3.1. The MXL field may be writable in implementations that support multiple base ISA widths. The effective XLEN in M-mode, *MXLEN*, is given by the setting of MXL, or has a fixed value if `misa` is zero. The MXL field is always set to the widest supported ISA variant at reset.

MXL	XLEN
1	32
2	64
3	128

Table 3.1: Encoding of MXL field in `misa`

The `misa` CSR is MXLEN bits wide. If the value read from `misa` is nonzero, field MXL of that value always denotes the current MXLEN. If a write to `misa` causes MXLEN to change, the position of MXL moves to the most-significant two bits of `misa` at the new width.

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 (XLEN) of the machine. The base width is given by* $XLEN = 2^{MXL+4}$.

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 machine is RV32. If zero after two shifts, then the machine 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, RV128I base ISAs, and the "E" bit will be set for RV32E. 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.

The RV128I base ISA is not yet frozen, 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.

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 hardwired to zero, the "E" bit always reads as the complement of the "I" bit. An implementation that supports both RV32E and RV32I 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.

Bit	Character	Description
0	A	Atomic extension
1	B	*Tentatively reserved for Bit-Manipulation extension*
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E 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	*Tentatively reserved for Dynamically Translated Languages extension*
10	K	*Reserved*
11	L	*Tentatively reserved for Decimal Floating-Point extension*
12	M	Integer Multiply/Divide extension
13	N	User-level interrupts supported
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	*Tentatively reserved for Transactional Memory extension*
20	U	User mode implemented
21	V	*Tentatively reserved for Vector extension*
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

Table 3.2: Encoding of Extensions field in `misa`. All bits that are reserved for future use must return zero when read.

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.

3.1.2 Machine Vendor ID Register `mvendorid`

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.

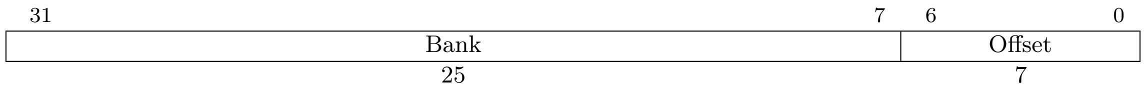

Figure 3.2: Vendor ID register (`mvendorid`).

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 0x7f 0x8a` (twelve continuation codes followed by `0x8a`) would be encoded in the `mvendorid` field 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 the RISC-V Foundation, 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 Register `marchid`

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.

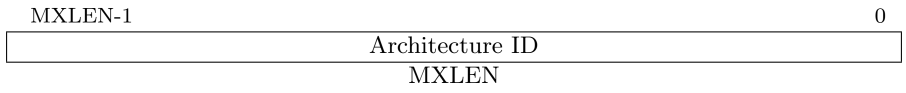

Figure 3.3: Machine Architecture ID register (`marchid`).

Open-source project architecture IDs are allocated globally by the RISC-V Foundation, 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 should be administered by the Foundation 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 Register `mimpid`

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.

MXLEN-1 0
Implementation
MXLEN

Figure 3.4: Machine Implementation ID register (`mimpid`).

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 Register `mhartid`

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.

MXLEN-1 0
Hart ID
MXLEN

Figure 3.5: Hart ID register (`mhartid`).

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 Registers (`mstatus` and `mstatush`)

The `mstatus` register is an MXLEN-bit read/write register formatted as shown in Figure 3.6 for RV64 and Figure 3.7 for RV32. 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.

MXLEN-1	MXLEN-2 38	37	36	35 34	33 32	31 23	22	21	20	19	18
SD	**WPRI**	MBE	SBE	SXL[1:0]	UXL[1:0]	**WPRI**	TSR	TW	TVM	MXR	SUM
1	MXLEN-39	1	1	2	2	9	1	1	1	1	1

17	16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
MPRV	XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	UBE	SPIE	**WPRI**	MIE	**WPRI**	SIE	**WPRI**
1	2	2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.6: Machine-mode status register (`mstatus`) for RV64.

31	30 23	22	21	20	19	18	17
SD	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV
1	8	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	UBE	SPIE	**WPRI**	MIE	**WPRI**	SIE	**WPRI**
2	2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.7: Machine-mode status register (`mstatus`) for RV32.

For RV32 only, `mstatush` is a 32-bit read/write register formatted as shown in Figure 3.8. 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`.

The `mstatush` register is not required to be implemented if every field would be hardwired to zero.

31 6	5	4	3 0
WPRI	MBE	SBE	**WPRI**
26	1	1	4

Figure 3.8: Additional machine-mode status register (`mstatush`) 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.

The role played by `mstatus` *in traps and privilege transition in the formal Sail model is described in §A.5.*

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 U (or M if user-mode is not supported). If xPP$\neq$M, xRET also sets MPRV=0.

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 xPP must be hardwired to 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 Base ISA Control in `mstatus` Register

For RV64 systems, 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 3.1. The effective XLEN in S-mode and U-mode are termed *SXLEN* and *UXLEN*, respectively.

For RV32 systems, the SXL and UXL fields do not exist, and SXLEN=32 and UXLEN=32.

For RV64 systems, if S-mode is not supported, then SXL is hardwired to zero. Otherwise, it is a **WARL** field that encodes the current value of SXLEN. In particular, the implementation may hardwire SXL so that SXLEN=MXLEN.

For RV64 systems, if U-mode is not supported, then UXL is hardwired to zero. Otherwise, it is a **WARL** field that encodes the current value of UXLEN. In particular, the implementation may hardwire UXL so that UXLEN=MXLEN or UXLEN=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.

If MXLEN is changed from 32 to a wider width, each of `mstatus` fields SXL and UXL, if not hardwired to a forced value, gets the value corresponding to the widest supported width not wider than the new MXLEN.

3.1.6.3 Memory Privilege in `mstatus` Register

The MPRV (Modify PRiVilege) bit modifies the privilege level at which loads and stores execute in all privilege modes. 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 hardwired to 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 4.17) 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 hardwired to 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 3.1.6.2.

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 4.17) 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 hardwired to 0 if S-mode is not supported.

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.

3.1.6.4 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 hardwired to 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 hardwired to 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 hardwire SBE so that SBE=MBE and writes to SBE are ignored. If U-mode is supported, an implementation may hardwire UBE so that UBE=MBE or UBE=SBE and writes to UBE are ignored.

An implementation supports only little-endian memory accesses if fields MBE, SBE, and UBE are all hardwired to 0. An implementation supports only big-endian memory accesses (aside from instruction fetches) if MBE is hardwired to 1 and SBE and UBE are each hardwired to 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 nonstandard 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.

The handling of memory privilege in `mstatus` *in the Sail formal model can be seen in §A.6.1 and in the translateAddr function in §A.6.2.*

3.1.6.5 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 the SFENCE.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 hard-wired to 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 instruction 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 3.2.3). 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. The time limit may always be 0, in which case WFI always causes an illegal instruction exception in less-privileged modes when TW=1. TW is hard-wired to 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 hard-wired to 0 when S-mode is not supported.

Trapping SRET is necessary to emulate the hypervisor extension (see Chapter 5) on implementations that do not provide it.

The role played by TW and TSR in the Sail model can be seen in §A.5, and that of TVM in §A.2.

3.1.6.6 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] **WARL** field 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, including the CSR `fcsr` and floating-point data registers `f0`–`f31`, while 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 and XS fields use the same status encoding as shown in Table 3.3, with the four possible status values being Off, Initial, Clean, and Dirty.

Status	FS 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

Table 3.3: Encoding of FS[1:0] and XS[1:0] status fields.

In systems that do not implement S-mode and do not have a floating-point unit, the FS field is hardwired to zero.

In systems without additional user extensions requiring new state, the XS field is hardwired to 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 3.3.

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 field or XS field signals the presence of some dirty state that will require saving extended user context to memory. If both XS and FS are hardwired to 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. Other extensions 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.

Table 3.4 shows all the possible state transitions for the FS or XS status bits. Note that the standard floating-point extensions do not support user-mode unconfigure or disable/enable instructions.

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 or XS bits encode a Dirty state (i.e., SD=((FS==11) OR (XS==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.

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 to modify 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

Table 3.4: FS and XS state transitions.

All privileged modes share a single copy of the FS and XS bits. In a system with more than one privileged mode, supervisor mode would normally use the FS and XS bits directly to record the status with respect to the supervisor-level saved context. Other more-privileged active modes 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.

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

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).

The `mtvec` register must always be implemented, but can contain a hardwired read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in

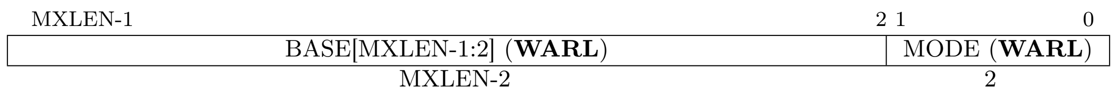

Figure 3.9: Machine trap-vector base-address register (`mtvec`).

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.

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

Table 3.5: Encoding of `mtvec` MODE field.

The encoding of the MODE field is shown in Table 3.5. 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 3.6 on page 40) causes the `pc` to be set to BASE+`0x1c`.

When vectored interrupts are enabled, interrupt cause 0, which corresponds to user-mode software interrupts, are vectored to the same location as synchronous exceptions. This ambiguity does not arise in practice, since user-mode software interrupts are either disabled or delegated to user mode.

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.

The Sail specification of the handling of the `mtvec` *register is in §A.4.*

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

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 3.2.2).

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`) and machine interrupt delegation register (`mideleg`) are MXLEN-bit read/write registers.

In systems 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 systems without S-mode, the `medeleg` and `mideleg` registers should not exist (unless the N extension for user-mode interrupts is implemented).

In versions 1.9.1 and earlier , these registers existed but were hardwired to zero in M-mode only, or M/U without N systems. 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 hardwire any bits of `medeleg` to one, i.e., any synchronous trap that can be delegated must support not being delegated. Similarly, an implementation shall not hardwire to one any bits of `mideleg` corresponding to machine-level interrupts (but may do so for lower-level interrupts).

Version 1.11 and earlier prohibited hardwiring any bits of `mideleg` *to 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.

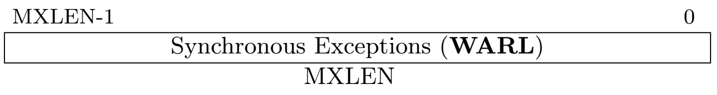

Figure 3.10: Machine Exception Delegation Register `medeleg`.

`medeleg` has a bit position allocated for every synchronous exception shown in Table 3.6 on page 40, 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).

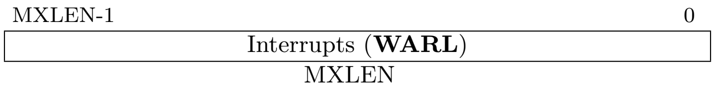

Figure 3.11: Machine Interrupt Delegation Register `mideleg`.

`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 hardwired to zero. In particular, `medeleg`[11] is hardwired to zero.

The specification of interrupt and exception delegation in the Sail formal model is in §A.3 and §A.5.

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

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 3.1.16) 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 available for platform or custom use.

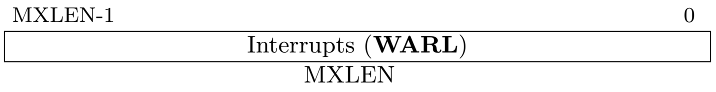

Figure 3.12: Machine Interrupt-Pending Register (`mip`).

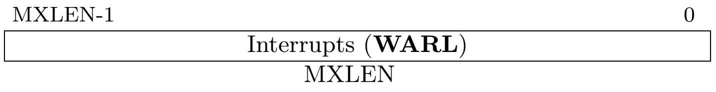

Figure 3.13: Machine Interrupt-Enable Register (`mie`).

An interrupt i will be taken if bit i is set in both `mip` and `mie`, and if interrupts are globally enabled. By default, M-mode interrupts are globally enabled if the hart's current privilege mode is less than M, or if the current privilege mode is M and the MIE bit in the `mstatus` register is set. If bit i in `mideleg` is set, however, interrupts are considered to be globally enabled if the hart's current privilege mode equals the delegated privilege mode and that mode's interrupt enable bit (xIE in `mstatus` for mode x) is set, or if the current privilege mode is less than the delegated privilege mode.

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 hardwired to zero.

The standard portions (bits 15:0) of registers `mip` and `mie` are formatted as shown in Figures 3.14 and 3.15 respectively.

15 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.14: Standard portion (bits 15:0) of `mip`.

15 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.15: 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 `mip`, 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 supervisor mode is not implemented, bits SEIP, STIP, and SSIP of `mip` and SEIE, STIE, and SSIE of `mie` are hardwired to 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. However, the value used in the read-modify-write sequence of a CSRRS or CSRRC instruction contains only the software-writable SEIP bit, ignoring the interrupt value from the external interrupt controller.

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`.

Interprocessor interrupts at supervisor level are implemented through implementation-specific mechanisms, e.g., via calls to an SEE, which might ultimately result in a machine-mode write to the receiving hart's MSIP bit.

We allow a hart to directly write only its own SSIP bit, not those of other harts, as other harts might be virtualized and possibly descheduled by higher privilege levels. We rely on calls to the SEE to provide interprocessor interrupts for this reason. Machine-mode harts are not virtualized and can directly interrupt other harts by setting their MSIP bits, typically using uncached I/O writes to memory-mapped control registers depending on the platform specification.

Multiple simultaneous interrupts destined for different privilege modes are handled in decreasing order of destined privilege mode. Multiple simultaneous interrupts destined for the same privilege mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI. Synchronous exceptions are of lower priority than all interrupts.

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.*

Synchronous exceptions are given the lowest priority to minimize worst-case interrupt latency.

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` appear to be hardwired to zero.

The processing of the `mip` *and* `mie` *registers in the Sail model is specified in §A.3.*

3.1.10 Machine Timer Registers (`mtime` and `mtimecmp`)

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 timebase of `mtime`. 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 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.

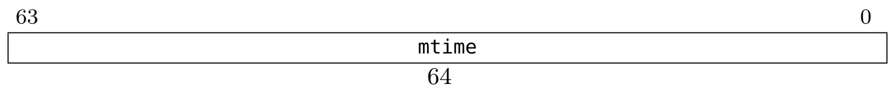

Figure 3.16: Machine time register (memory-mapped control register).

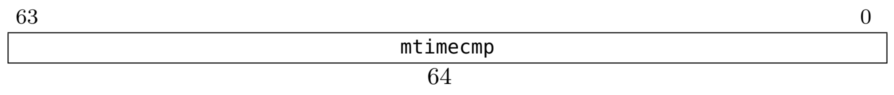

Figure 3.17: Machine time compare register (memory-mapped control register).

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.

Writes to `mtime` and `mtimecmp` are 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:

```
# 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.
```

Figure 3.18: 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.

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

3.1.11 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 systems.

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 MXLEN-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 hard-wire both the counter and its corresponding event selector to 0.

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

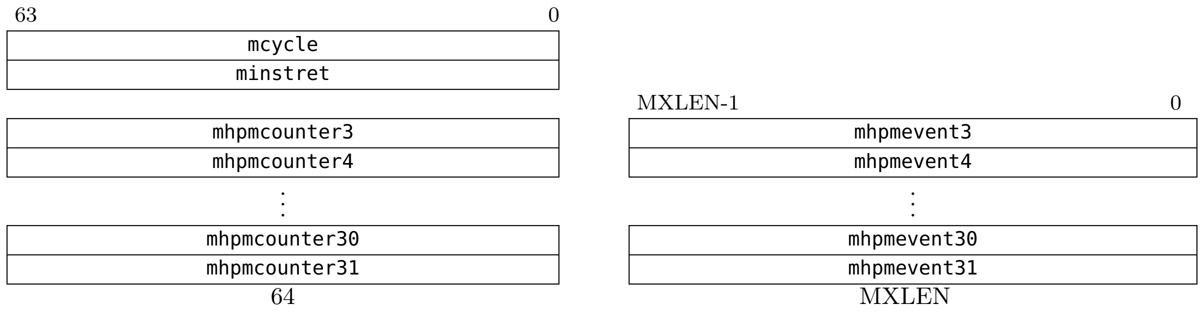

Figure 3.19: Hardware performance monitor counters.

A future revision of this specification will define a mechanism to generate an interrupt when a hardware performance monitor counter overflows.

On RV32 only, reads of the `mcycle`, `minstret`, and `mhpmcountern` CSRs return the low 32 bits, while reads of the `mcycleh`, `minstreth`, and `mhpmcounternh` CSRs return bits 63–32 of the corresponding counter.

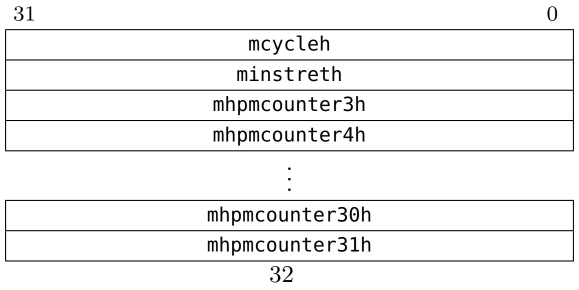

Figure 3.20: Upper 32 bits of hardware performance monitor counters, RV32 only.

3.1.12 Machine Counter-Enable Register (`mcounteren`)

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

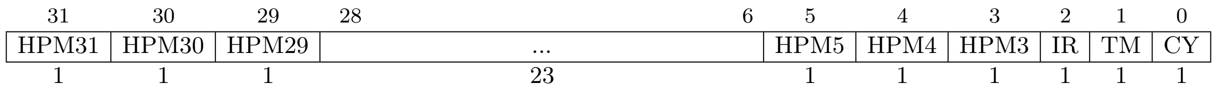

Figure 3.21: Counter-enable register (`mcounteren`).

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 systems that do not need high-performance timers and counters, machine-mode software can trap accesses and implement all features in software. For systems 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 `hpmcounter`*`n`* 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, on RV32I the `cycleh`, `instreth` and `hpmcounter`*`n`* CSRs are read-only shadows of `mcycleh`, `minstreth` and `mhpmcounter`*`n`*`h`, respectively. On RV32I the `timeh` CSR is a read-only shadow of the upper 32 bits of the memory-mapped `mtime` register, while `time` shadows only the lower 32 bits of `mtime`.

Implementations can convert reads of the `time` *and* `timeh` *CSRs into loads to the memory-mapped* `mtime` *register, or emulate this functionality in M-mode software.*

3.1.13 Machine Counter-Inhibit CSR (`mcountinhibit`)

31	30	29	28 … 6	5	4	3	2	1	0
HPM31	HPM30	HPM29	...	HPM5	HPM4	HPM3	IR	0	CY
1	1	1	23	1	1	1	1	1	1

Figure 3.22: Counter-inhibit register `mcountinhibit`.

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 HPMn bit in the `mcountinhibit` register is clear, the `cycle`, `instret`, or `hpmcounter`*`n`* register increments as usual. When the CY, IR, or HPMn 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 `cycle` *and* `instret` *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 `time` *counter can be shared between multiple cores, it cannot be inhibited with the* `mcountinhibit` *mechanism.*

3.1.14 Machine Scratch Register (`mscratch`)

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.

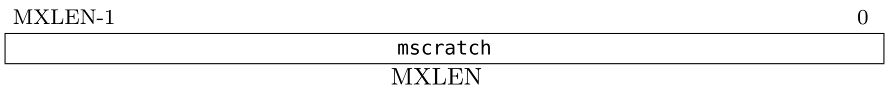

Figure 3.23: Machine-mode scratch register.

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.

3.1.15 Machine Exception Program Counter (`mepc`)

`mepc` is an MXLEN-bit read/write register formatted as shown in Figure 3.24. 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. Implementations may convert some invalid address patterns into other invalid addresses prior to writing them to `mepc`.

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.

The use of `mepc` in the Sail model is specified in §A.4 and §A.5.

MXLEN-1 0
`mepc`
MXLEN

Figure 3.24: Machine exception program counter register.

3.1.16 Machine Cause Register (`mcause`)

The `mcause` register is an MXLEN-bit read-write register formatted as shown in Figure 3.25. 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. Table 3.6 lists the possible machine-level exception codes. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

MXLEN-1	MXLEN-2 0
Interrupt	Exception Code (**WLRL**)
1	MXLEN-1

Figure 3.25: Machine Cause register `mcause`.

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 opcode 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 raises multiple synchronous exceptions, the decreasing priority order of Table 3.7 indicates which exception is taken and reported in `mcause`. The priority of any custom synchronous exceptions is implementation-defined.

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.

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	12–15	*Reserved*
1	≥16	*Available 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	16–23	*Reserved*
0	24–31	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

Table 3.6: Machine cause register (`mcause`) values after trap.

Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12	Instruction page fault
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8, 9, 11	Environment call
	3	Environment break
	3	Load/Store/AMO address breakpoint
	6	Store/AMO address misaligned
	4	Load address misaligned
	15	Store/AMO page fault
	13	Load page fault
	7	Store/AMO access fault
Lowest	5	Load access fault

Table 3.7: Synchronous exception priority in decreasing priority order.

3.1.17 Machine Trap Value Register (`mtval`)

The `mtval` register is an MXLEN-bit read-write register formatted as shown in Figure 3.26. 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 and which may unconditionally set it to zero.

When a hardware breakpoint is triggered, or an address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, `mtval` is written with the faulting virtual address. On an illegal instruction trap, `mtval` may be written with the first XLEN or ILEN bits of the faulting instruction as described below. For other traps, `mtval` is set to zero, but a future standard may redefine `mtval`'s setting for other traps.

The `mtval` *register replaces the* `mbadaddr` *register in the previous specification. In addition to providing bad addresses, the register can now provide the bad instruction that triggered an illegal instruction trap (and may in future be used to return other information). Returning the instruction bits accelerates instruction emulation and also removes some races that might be present when trying to emulate illegal instructions.*

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.*

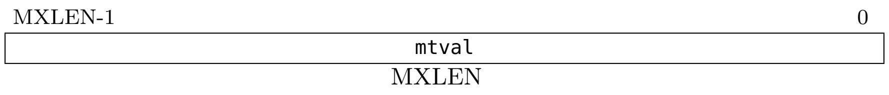

Figure 3.26: Machine Trap Value register.

For misaligned loads and stores that cause access-fault or page-fault exceptions, `mtval` will contain the virtual address of the portion of the access that caused the fault. For instruction access-fault or page-fault exceptions on systems with variable-length instructions, `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 this feature is not provided, then `mtval` is set to zero on an illegal instruction fault.

If this feature is provided, after an illegal instruction trap, `mtval` will contain the shortest of:

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

The value loaded into `mtval` 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 XLEN 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).*

If the hardware platform specifies that no exceptions set `mtval` to a nonzero value, then it may be hardwired to zero. Otherwise, `mtval` 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. Implementations may convert some invalid address patterns into other invalid addresses prior to writing them to `mtval`. 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 XLEN and ILEN.

The use of `mtval` *in the Sail model is specified in §A.4 and §A.5.*

3.2 Machine-Mode Privileged Instructions

3.2.1 Environment Call and Breakpoint

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

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.

```
function clause execute ECALL() = {
  let t : sync_exception =
    struct { trap = match (cur_privilege) {
                      User => E_U_EnvCall(),
                      Supervisor => E_S_EnvCall(),
                      Machine => E_M_EnvCall()
                    },
             excinfo = (None() : option(xlenbits)),
             ext = None() };
  set_next_pc(exception_handler(cur_privilege, CTL_TRAP(t), PC));
  RETIRE_FAIL
}
```

```
function clause execute EBREAK() = {
  handle_exception(E_Breakpoint());
  RETIRE_FAIL
}
```

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. It generates 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.

3.2.2 Trap-Return Instructions

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

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRET/SRET	0	PRIV	0	SYSTEM

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 3.1.6.5. An xRET instruction can be executed in privilege mode x or higher, where executing a lower-privilege xRET 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, xRET sets the `pc` to the value stored in the x`epc` register.

Previously, there was only a single ERET instruction (which was also earlier known as SRET). To support the addition of user-level interrupts, we needed to add a separate URET instruction to continue to allow classic virtualization of OS code using the ERET instruction. It then became more orthogonal to support a different x*RET instruction per privilege level.*

```
function clause execute MRET() = {
 if cur_privilege != Machine
 then handle_illegal()
 else if ~(ext_check_xret_priv (Machine))
 then ext_fail_xret_priv ()
 else set_next_pc(exception_handler(cur_privilege, CTL_MRET(), PC));
 RETIRE_FAIL
}
```

```
function clause execute SRET() = {
 let sret_illegal : bool = match cur_privilege {
  User => true,
  Supervisor => ~ (haveSupMode ()) | mstatus.TSR() == 0b1,
  Machine => ~ (haveSupMode ())
 };
 if sret_illegal
 then handle_illegal()
 else if ~(ext_check_xret_priv (Supervisor))
 then ext_fail_xret_priv ()
 else set_next_pc(exception_handler(cur_privilege, CTL_SRET(), PC));
 RETIRE_FAIL
}
```

```
function clause execute URET() = {
 if (~ (haveUsrMode()))
 then handle_illegal()
 else if ~ (ext_check_xret_priv (User))
 then ext_fail_xret_priv ()
```

```
  else set_next_pc(exception_handler(cur_privilege, CTL_URET(), PC));
  RETIRE_FAIL
}
```

If the A extension is supported, the xRET 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 xRET.

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

3.2.3 Wait for Interrupt

The Wait for Interrupt instruction (WFI) provides a hint to 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 3.1.6.5.

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
WFI	0	PRIV	0	SYSTEM

If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt exception 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 exception and trap, so that a simple return from the trap handler will execute code after the WFI instruction.

The purpose of the WFI instruction is to provide a hint to the implementation, and so a legal implementation is to simply implement WFI 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`*,* `sip`*, or* `uip` *registers can be interrogated to determine the presence of any interrupt in machine, supervisor, or user 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.

```
function clause execute WFI() =
 match cur_privilege {
  Machine => { platform_wfi(); RETIRE_SUCCESS },
  Supervisor => if mstatus.TW() == 0b1
              then { handle_illegal(); RETIRE_FAIL }
              else { platform_wfi(); RETIRE_SUCCESS },
  User => { handle_illegal(); RETIRE_FAIL }
 }
```

The $platform\_wfi$ function here implements a platform-specific WFI semantics.

3.3 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 and widest MXLEN, as described in Section 3.1.1. 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. 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 (e.g., hard 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.*

3.4 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 address of the next instruction to be executed at the time the NMI was taken, 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.

3.5 Physical Memory Attributes

The physical memory map for a complete system includes various address ranges, some corresponding to memory regions, some to memory-mapped control registers, and some to empty holes in the address space. 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 3.6, 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 slave 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.

3.5.1 Main Memory versus I/O versus Empty Regions

The most important characterization of a given memory address range is whether it holds regular main memory, or I/O devices, or is empty. 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. Empty regions are also classified as I/O regions but with attributes specifying that no accesses are supported.

3.5.2 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.

3.5.3 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.

3.5.3.1 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.

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

Table 3.8: Classes of AMOs supported by I/O regions.

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

3.5.3.2 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.

3.5.3.3 Alignment

Memory regions that support aligned LR/SC or aligned AMOs might also support misaligned LR/SC or misaligned AMOs for some addresses and access widths. If, for a given address and access width, a misaligned LR/SC or AMO generates an address-misaligned exception, then *all* loads, stores, LRs/SCs, and AMOs using that address and access width must generate address-misaligned exceptions.

The standard "A" extension does not support misaligned AMOs or LR/SC pairs. Support for misaligned AMOs is provided by the standard "Zam" extension. Support for misaligned LR/SC sequences is not currently standardized, so LR and SC to misaligned addresses must raise an exception.

Mandating that misaligned loads and stores raise address-misaligned exceptions wherever misaligned AMOs raise address-misaligned exceptions permits the emulation of misaligned AMOs in an M-mode trap handler. The handler guarantees atomicity by acquiring a global mutex and emulating the access within the critical section. Provided that the handler for misaligned loads and stores uses the same mutex, all accesses to a given address that use the same word size will be mutually 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. If, for a given address and access width, all misaligned LRs/SCs and AMOs generate access-fault exceptions, then regular misaligned loads and stores using the same address and access width are not required to execute atomically.

3.5.4 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 targeted slave 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.

3.5.5 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 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.

We categorize RISC-V caches into three types: master-private*,* shared*, and* slave-private*. Master-private caches are attached to a single master agent, i.e., one that issues read/write requests to the memory system. Shared caches are located between masters and slaves and may be hierarchically organized. Slave-private caches do not impact coherence, as they are local to a single slave and do not affect other PMAs at a master, so are not considered further here. We use* private cache *to mean a master-private cache in the following section, unless explicitly stated otherwise.*

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.

3.5.6 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.

3.6 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 3.5.

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 in S and U mode, data accesses in S and U mode when the MPRV bit in the `mstatus` register is clear, and data accesses in any 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.

3.6.1 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 CSRs. All PMP CSR fields are **WARL** and may be hardwired to 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 3.27. For RV64, eight even-numbered CSRs, `pmpcfg0`, `pmpcfg2`, . . . , `pmpcfg14`, hold the configurations for the 64 PMP entries, as shown in Figure 3.28. For RV64, the odd-numbered configuration registers, `pmpcfg1`, `pmpcfg3`, . . . , `pmpcfg15`, are illegal.

RV64 systems 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.*

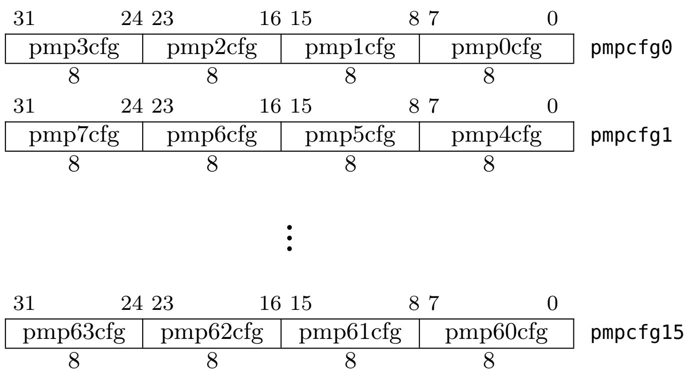

Figure 3.27: RV32 PMP configuration CSR layout.

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 3.29. For RV64, each PMP address register encodes bits 55–2 of a 56-bit physical address, as shown in Figure 3.30. 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 4.3 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 Sections 4.4 and 4.5 support a 56-bit physical address space, so the RV64 PMP address registers impose the same limit.

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
pmp7cfg	pmp6cfg	pmp5cfg	pmp4cfg	pmp3cfg	pmp2cfg	pmp1cfg	pmp0cfg	`pmpcfg0`
8	8	8	8	8	8	8	8	

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
pmp15cfg	pmp14cfg	pmp13cfg	pmp12cfg	pmp11cfg	pmp10cfg	pmp9cfg	pmp8cfg	`pmpcfg2`
8	8	8	8	8	8	8	8	

⋮

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
pmp63cfg	pmp62cfg	pmp61cfg	pmp60cfg	pmp59cfg	pmp58cfg	pmp57cfg	pmp56cfg	`pmpcfg14`
8	8	8	8	8	8	8	8	

Figure 3.28: RV64 PMP configuration CSR layout.

31 0
address[33:2] (**WARL**)
32

Figure 3.29: PMP address register format, RV32.

63 54	53 0
0 (**WARL**)	address[55:2] (**WARL**)
10	54

Figure 3.30: PMP address register format, RV64.

Figure 3.31 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 combination R=0 and W=1 is reserved for future use. The remaining two fields, A and L, are described in the following sections.

7	6 5	4 3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)
1	2	2	1	1	1

Figure 3.31: 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 whose effective address lies within a PMP region without read permissions raises a load access-fault exception. Attempting to execute a store, store-conditional (regardless of success), or AMO instruction whose effective address lies within a PMP region without write permissions raises a store access-fault exception.

If MXLEN is changed, the contents of the `pmp`*`x`*`cfg` fields are preserved, but appear in the `pmpcfg`*`y`* CSR prescribed by the new setting of MXLEN. For example, when MXLEN is changed from 64 to 32, `pmp4cfg` moves from `pmpcfg0`[39:32] to `pmpcfg1`[7:0]. The `pmpaddr` CSRs follow the usual CSR width modulation rules described in Section 2.4.

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 3.9. 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.

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, $\geq$8 bytes

Table 3.9: Encoding of A field in PMP configuration registers.

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 3.10.

`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

Table 3.10: NAPOT range encoding in PMP address and configuration registers.

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 $\texttt{pmpaddr}_{i-1} \leq y < \texttt{pmpaddr}_i$. If PMP entry 0's A field is set to TOR, zero is used for the lower bound, and so it matches any address $y < \texttt{pmpaddr}_0$.

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 $G \geq 1$, the NA4 mode is not selectable. When $G \geq 2$ and $\texttt{pmpcfg}_i$.A[1] is set, i.e. the mode is NAPOT, then bits $\texttt{pmpaddr}_i$[G-2:0] read as all ones. When $G \geq 1$ and $\texttt{pmpcfg}_i$.A[1] is clear, i.e. the mode is OFF or TOR, then bits $\texttt{pmpaddr}_i$[G-1:0] read as all zeros. Bits $\texttt{pmpaddr}_i$[G-1:0] do not affect the TOR address-matching logic. Although changing $\texttt{pmpcfg}_i$.A[1] affects the value read from $\texttt{pmpaddr}_i$, it does not affect the underlying value stored in that register—in particular, $\texttt{pmpaddr}_i$[G-1] retains its original value when $\texttt{pmpcfg}_i$.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.

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 `pmp`i`cfg` and `pmpaddr`i are ignored. Additionally, if `pmp`i`cfg`.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.

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 floating-point stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

3.6.2 Physical Memory Protection and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in Chapter 4. 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 virtual-memory access. The PMP settings for the resulting physical address may be checked at any point between the address translation and the explicit virtual-memory access. Hence, when the PMP settings are modified in a manner that affects either the physical memory that holds the page tables or the physical memory to which the page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. This is accomplished by executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0`, after the PMP CSRs are written.

If page-based virtual memory is not implemented, or when it is disabled, memory accesses check the PMP settings synchronously, so no fence is needed.

Chapter 4

Supervisor-Level ISA, Version 1.12

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.

4.1 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.

4.1.1 Supervisor Status Register (`sstatus`)

The `sstatus` register is an SXLEN-bit read/write register formatted as shown in Figure 4.1 for RV32 and Figure 4.2 for RV64. The `sstatus` register keeps track of the processor's current operating state.

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 3.2.2) is executed to return from the trap handler, the

31	30 20	19	18	17	16 15	14 13	12 9	8	7	6	5	4 2	1	0
SD	**WPRI**	MXR	SUM	**WPRI**	XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	UBE	SPIE	**WPRI**	SIE	**WPRI**
1	11	1	1	1	2	2	4	1	1	1	1	3	1	1

Figure 4.1: Supervisor-mode status register (`sstatus`) for RV32.

SXLEN-1	SXLEN-2 34	33 32	31 20	19	18	17
SD	**WPRI**	UXL[1:0]	**WPRI**	MXR	SUM	**WPRI**
1	SXLEN-35	2	12	1	1	1

16 15	14 13	12 9	8	7	6	5	4 2	1	0
XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	UBE	SPIE	**WPRI**	SIE	**WPRI**
2	2	4	1	1	1	1	3	1	1

Figure 4.2: Supervisor-mode status register (`sstatus`) for RV64.

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`*.*

4.1.1.1 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 3.1.

For RV32 systems, the UXL field does not exist, and UXLEN=32. For RV64 systems, it is a **WARL** field that encodes the current value of UXLEN. In particular, the implementation may hardwire UXL so that UXLEN=SXLEN.

If UXLEN $\neq$ 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.

4.1.1.2 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 4.17) 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 4.17) 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.

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.

4.1.1.3 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 hardwire UBE to specify always 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.

4.1.2 Supervisor Trap Vector Base Address Register (`stvec`)

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).

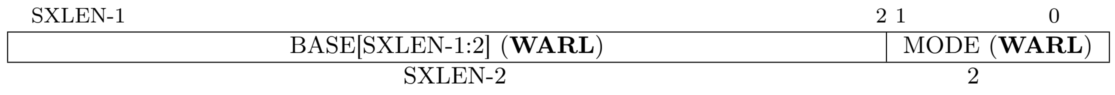

Figure 4.3: Supervisor trap vector base address register (`stvec`).

The BASE field in `stvec` is a **WARL** 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.

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

Table 4.1: Encoding of `stvec` MODE field.

The encoding of the MODE field is shown in Table 4.1. 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 4.2) causes the `pc` to be set to BASE+`0x14`. Setting MODE=Vectored may impose a stricter alignment constraint on BASE.

4.1.3 Supervisor Interrupt Registers (`sip` and `sie`)

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 4.1.8) 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 available for platform or custom use.

An interrupt i will be taken if bit i is set in both `sip` and `sie`, and if supervisor-level interrupts are globally enabled. Supervisor-level interrupts are globally enabled if the hart's current privilege

SXLEN-1 … 0
Interrupts (**WARL**)
SXLEN

Figure 4.4: Supervisor interrupt-pending register (`sip`).

SXLEN-1 … 0
Interrupts (**WARL**)
SXLEN

Figure 4.5: Supervisor interrupt-enable register (`sie`).

mode is less than S, or if the current privilege mode is S and the SIE bit in the `sstatus` register is set.

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 must be hardwired to zero.

The standard portions (bits 15:0) of registers `sip` and `sie` are formatted as shown in Figures 4.6 and 4.7 respectively.

15 … 10	9	8 … 6	5	4 … 2	1	0
0	SEIP	0	STIP	0	SSIP	0
6	1	3	1	3	1	1

Figure 4.6: Standard portion (bits 15:0) of `sip`.

15 … 10	9	8 … 6	5	4 … 2	1	0
0	SEIE	0	STIE	0	SSIE	0
6	1	3	1	3	1	1

Figure 4.7: 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`. A supervisor-level software interrupt is triggered on the current hart by writing 1 to SSIP, while a pending supervisor-level software interrupt can be cleared by writing 0 to SSIP.

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, or SSI) may not be implemented, in which case the corresponding interrupt-pending and interrupt-enable bits are hardwired to 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 hardwired to 0 in Figures 4.6 and 4.7.*

Multiple simultaneous interrupts destined for supervisor mode are handled in the following decreasing priority order: SEI, SSI, STI. Synchronous exceptions are of lower priority than all interrupts.

4.1.4 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`.

4.1.5 Counter-Enable Register (`scounteren`)

31	30	29	28 6	5	4	3	2	1	0
HPM31	HPM30	HPM29	...	HPM5	HPM4	HPM3	IR	TM	CY
1	1	1	23	1	1	1	1	1	1

Figure 4.8: Counter-enable register (`scounteren`).

The counter-enable register `scounteren` is a 32-bit register that controls the availability of the hardware performance monitoring counters to U-mode.

When the CY, TM, IR, or HPMn bit in the `scounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcounter`*n* 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 contain a hardwired value of 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.*

4.1.6 Supervisor Scratch Register (`sscratch`)

The `sscratch` register 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.

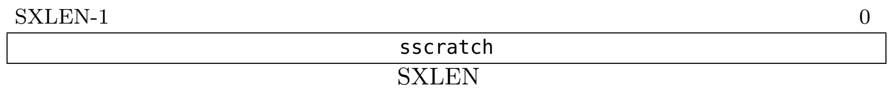

Figure 4.9: Supervisor Scratch Register.

4.1.7 Supervisor Exception Program Counter (`sepc`)

`sepc` is an SXLEN-bit read/write register formatted as shown in Figure 4.10. 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. Implementations may convert some invalid address patterns into other invalid addresses prior to writing them to `sepc`.

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.

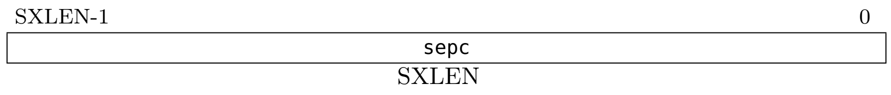

Figure 4.10: Supervisor exception program counter register.

4.1.8 Supervisor Cause Register (`scause`)

The `scause` register is an SXLEN-bit read-write register formatted as shown in Figure 4.11. 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. Table 4.2 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.

SXLEN-1	SXLEN-2 0
Interrupt	Exception Code (**WLRL**)
1	SXLEN-1

Figure 4.11: Supervisor Cause register `scause`.

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	10–15	*Reserved*
1	≥16	*Available 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	16–23	*Reserved*
0	24–31	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

Table 4.2: Supervisor cause register (`scause`) values after trap. Synchronous exception priorities are given by Table 3.7.

4.1.9 Supervisor Trap Value (`stval`) Register

The `stval` register is an SXLEN-bit read-write register formatted as shown in Figure 4.12. 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 and which may unconditionally set it to zero.

When a hardware breakpoint is triggered, or an instruction, load, or store address-misaligned, access-fault, or page-fault exception occurs, `stval` is written with the faulting virtual address. On an illegal instruction trap, `stval` may be written with the first XLEN or ILEN bits of the faulting instruction as described below. For other exceptions, `stval` is set to zero, but a future standard may redefine `stval`'s setting for other exceptions.

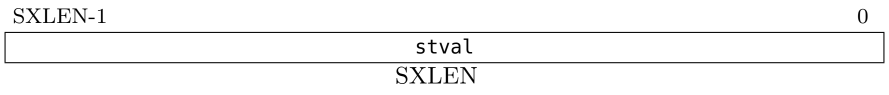

Figure 4.12: Supervisor Trap Value register.

For misaligned loads and stores that cause access-fault or page-fault exceptions, `stval` will contain the virtual address of the portion of the access that caused the fault. For instruction access-fault or page-fault exceptions on systems with variable-length instructions, `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 this feature is not provided, then `stval` is set to zero on an illegal instruction fault.

If this feature is provided, after an illegal instruction trap, `stval` will contain the shortest of:

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

The value loaded into `stval` is right-justified and all unused upper bits are cleared to zero.

`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. Implementations may convert some invalid address patterns into other invalid addresses prior to writing them to `stval`. 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 XLEN and ILEN.

4.1.10 Supervisor Address Translation and Protection (`satp`) Register

The `satp` register is an SXLEN-bit read/write register, formatted as shown in Figure 4.13 for SXLEN=32 and Figure 4.14 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 3.1.6.5.

31	30 22	21 0
`MODE` (**WARL**)	`ASID` (**WARL**)	`PPN` (**WARL**)
1	9	22

Figure 4.13: RV32 Supervisor address translation and protection register `satp`.

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.*

63 60	59 44	43 0
`MODE` (**WARL**)	`ASID` (**WARL**)	`PPN` (**WARL**)
4	16	44

Figure 4.14: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

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 4.3 shows the encodings of the MODE field for RV32 and RV64. 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 3.6. When MODE=Bare, patterns other than zero in the remaining fields of `satp` (bits 30–0 for RV32, or bits 59–0 for RV64) are reserved for future use. To select MODE=Bare, software must write zero to these fields. 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.

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.*

For RV32, the only other valid setting for MODE is Sv32, a paged virtual-memory scheme described in Section 4.3.

For RV64, two paged virtual-memory schemes are defined: Sv39 and Sv48, described in Sections 4.4 and 4.5, respectively. Two additional schemes, Sv57 and 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.

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 4.3).

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39	Page-based 39-bit virtual addressing (see Section 4.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 4.5).
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–15	—	*Reserved*

Table 4.3: Encoding of `satp` MODE field.

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 and Sv48.

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 [3]. Additionally, multi-level TLB hierarchies are quite inexpensive relative to the multi-level cache hierarchies whose address space they map.

Note that writing `satp` does not imply any ordering constraints between page-table updates and subsequent address translations. 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 4.2.1) after 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.*

4.2 Supervisor Instructions

In addition to the SRET instruction defined in Section 3.2.2, one new supervisor-level instruction is provided.

4.2.1 Supervisor Memory-Management Fence Instruction

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
SFENCE.VMA	asid	vaddr	PRIV	0	SYSTEM

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 all subsequent implicit references from that hart to the memory-management data structures. Further details on the behavior of this instruction are described in Section 3.1.6.5 and Section 3.6.2.

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.

- 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 4.3.1) are not ordered.
- 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.
- 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.

When *rs2*≠`x0`, bits SXLEN-1:ASIDMAX of the value held in *rs2* are reserved for future use and 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 can ignore the virtual address in rs1 *and the ASID value in* rs2 *and always perform a global fence.*

Implementations may perform implicit reads of the translation data structures pointed to by the current `satp` register arbitrarily early and speculatively. The results of these reads may be held in an incoherent cache but not shared with other harts. Cache entries may only be established for the ASID currently loaded into the `satp` register, or for global entries. The cache may only satisfy implicit reads for entries that have been established for the ASID currently loaded into `satp`, or for global entries. Changes in the `satp` register do not necessarily flush any such translation caches. To ensure the 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.

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.*

Implementations must only perform implicit reads of the translation data structures pointed to by the current contents of the `satp` register, 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.

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.*

For forward compatibility with future versions of this specification, supervisor software should adhere to the following constraints on ASID usage. Supervisor software that uses ASIDs should use a nonzero ASID value to refer to the same address space across all harts in the SEE and should not use an ASID value of 0. Supervisor software that does not use ASIDs should always set the ASID field in the `satp` CSR to 0.

A future extension may define ASIDs to be global across the SEE, facilitating such optimizations as multicast TLB shootdown and sharing translation caches between harts.

The extension will provide a mechanism so that existing supervisor software that always uses ASID 0 will continue to function as it currently does.

```
function clause execute SFENCE_VMA(rs1, rs2) = {
 let addr : option(xlenbits) = if rs1 == 0b00000 then None() else Some(X(rs1));
 let asid : option(xlenbits) = if rs2 == 0b00000 then None() else Some(X(rs2));
 match cur_privilege {
  User => { handle_illegal(); RETIRE_FAIL },
  Supervisor => match (architecture(get_mstatus_SXL(mstatus)), mstatus.TVM()) {
               (Some(_), 0b1) => { handle_illegal(); RETIRE_FAIL },
               (Some(_), 0b0) => { flush_TLB(asid, addr); RETIRE_SUCCESS },
               (_, _) => internal_error("unimplemented sfence architecture")
             },
  Machine => { flush_TLB(asid, addr); RETIRE_SUCCESS }
 }
}
```

The flush_TLB function is not described in the Sail model included in this document, since the specification leaves the details of a TLB unspecified.

4.3 Sv32: Page-Based 32-bit Virtual-Memory Systems

When Sv32 is written to the MODE field in the `satp` register (see Section 4.1.10), 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 on RV32 systems 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.

4.3.1 Addressing and Memory Protection

Sv32 implementations support a 32-bit virtual address space, divided into 4 KiB pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown in Figure 4.15. 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 (Sections 3.6), before being directly converted to machine-level physical addresses.

31 22	21 12	11 0
VPN[1]	VPN[0]	page offset
10	10	12

Figure 4.15: Sv32 virtual address.

33 22	21 12	11 0
PPN[1]	PPN[0]	page offset
12	10	12

Figure 4.16: Sv32 physical address.

31 20	19 10	9 8	7	6	5	4	3	2	1	0
PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
12	10	2	1	1	1	1	1	1	1	1

Figure 4.17: Sv32 page table entry.

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.

The PTE format for Sv32 is shown in Figures 4.17. 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 4.4 summarizes the encoding of the permission bits.

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.

Table 4.4: Encoding of PTE R/W/X fields.

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 (regardless of success), 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 4.1.1.2) 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 permitted:

- 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 a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, the implementation sets the corresponding bit(s) in the PTE. The PTE update must be atomic with respect to other accesses to the PTE, and must atomically check that the PTE is valid and grants sufficient permissions. The PTE update must be exact (i.e., not speculative), and observed in program order by the local hart. Furthermore, the PTE update must appear in the global memory order no later than the explicit memory access, or 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 explicit memory access that caused the update, and the sequence is interruptible. However, the hart must not perform the explicit memory access before the PTE update is globally visible.

All harts in a system must employ the same PTE-update scheme as each other.

Mandating that the PTE updates to be exact, atomic, and in program order simplifies the specification, and makes the feature more useful for system software. 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 and must be cleared by software for forward compatibility.

The handling of PTEs in the Sail model is specified in §A.6.1.

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 page (i.e., a naturally aligned 4 KiB region).

4.3.2 Virtual Address Translation Process

A virtual address *va* is translated into a physical address *pa* as follows:

1. If XLEN equals VALEN, proceed. (For Sv32, VALEN=32.) Otherwise, check whether each bit of *va*[XLEN-1:VALEN] is equal to *va*[VALEN-1]. If not, stop and raise a page-fault exception corresponding to the original access type.
2. Let a be `satp`.*ppn* × PAGESIZE, and let $i = \text{LEVELS} - 1$. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.)
3. Let *pte* be the value of the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$. (For Sv32, PTESIZE=4.) If accessing *pte* violates a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
4. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception corresponding to the original access type.
5. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 6. 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 $a = pte.ppn \times \text{PAGESIZE}$ and go to step 3.
6. 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.
7. If $i > 0$ and $pte.ppn[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
8. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, either raise a page-fault exception corresponding to the original access type, or:
   - Set *pte.a* to 1 and, if the memory access is a store, also set *pte.d* to 1.
   - If this access violates a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
   - This update and the loading of *pte* in step 3 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.
9. 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 $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

4.4 Sv39: Page-Based 39-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, 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, 512 GiB of virtual-address space is ample, and so Sv39 suffices. Sv48 increases the virtual address space to 256 TiB, 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.

4.4.1 Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into 4 KiB pages. An Sv39 address is partitioned as shown in Figure 4.18. 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 usually zero-extends a narrower 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.

38 30	29 21	20 12	11 0
VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	12

Figure 4.18: Sv39 virtual address.

55 30	29 21	20 12	11 0
PPN[2]	PPN[1]	PPN[0]	page offset
26	9	9	12

Figure 4.19: Sv39 physical address.

63 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

Figure 4.20: 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 4.20. Bits 9–0 have the same meaning as for Sv32. Bits 63–54 are reserved for future standard use and must be zeroed by software for forward compatibility.

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 64 PiB 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 4.3.2, except VALEN equals 39, LEVELS equals 3, and PTESIZE equals 8.

The Sail specification of the Sv39 address translation scheme is in §A.6.2. We do not include the Sail specifications of the Sv32 and Sv48 schemes in this document since they have a very similar structure.

4.5 Sv48: Page-Based 48-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, 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.

4.5.1 Addressing and Memory Protection

Sv48 implementations support a 48-bit virtual address space, divided into 4 KiB pages. An Sv48 address is partitioned as shown in Figure 4.21. 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.

47 39	38 30	29 21	20 12	11 0
VPN[3]	VPN[2]	VPN[1]	VPN[0]	page offset
9	9	9	9	12

Figure 4.21: Sv48 virtual address.

The PTE format for Sv48 is shown in Figure 4.23. Bits 9–0 have the same meaning as for Sv32. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv48 supports 2 MiB *megapages*,

55 39	38 30	29 21	20 12	11 0
PPN[3]	PPN[2]	PPN[1]	PPN[0]	page offset
17	9	9	9	12

Figure 4.22: Sv48 physical address.

63 54	53 37	36 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[3]	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	17	9	9	9	2	1	1	1	1	1	1	1	1

Figure 4.23: Sv48 page table entry.

1 GiB *gigapages*, and 512 GiB *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 4.3.2, except VALEN equals 39, LEVELS equals 4, and PTESIZE equals 8.

Chapter 5

Hypervisor Extension, Version 0.6.1

Warning! This draft specification may change before being accepted as standard by the RISC-V Foundation.

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 is enabled by setting bit 7 in the `misa` CSR, which corresponds to the letter H. When `misa`[7] is clear, the hart behaves as though this extension were not implemented, and attempts to use hypervisor CSRs or instructions raise an illegal instruction exception. Implementations that include 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.

5.1 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 5.1 lists the possible operating modes of a RISC-V hart with the hypervisor extension.

Virtualization Mode (V)	Privilege Encoding	Abbreviation	Name	Two-Stage Translation
0	0	U-mode	User mode	Off
0	1	HS-mode	Hypervisor-extended supervisor mode	Off
0	3	M-mode	Machine mode	Off
1	0	VU-mode	Virtual user mode	On
1	1	VS-mode	Virtual supervisor mode	On

Table 5.1: Operating modes with the hypervisor extension.

5.2 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`, `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.

A few standard supervisor CSRs (`scounteren` and, if the N extension is implemented, `sedeleg` and `sideleg`) 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.

5.2.1 Hypervisor Status Register (`hstatus`)

The `hstatus` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.1 when HSXLEN=32 and Figure 5.2 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.

31 23	22	21	20	19 18	17 12	11 10	9	8	7	6	5	4 0
WPRI	VTSR	VTW	VTVM	**WPRI**	VGEIN[5:0]	**WPRI**	HU	SPVP	SPV	GVA	VSBE	**WPRI**
9	1	1	1	2	6	2	1	1	1	1	1	5

Figure 5.1: Hypervisor status register (`hstatus`) for RV32.

HSXLEN-1 34	33 32	31 23	22	21	20
WPRI	VSXL[1:0]	**WPRI**	VTSR	VTW	VTVM
HSXLEN-34	2	9	1	1	1

19 18	17 12	11 10	9	8	7	6	5	4 0
WPRI	VGEIN[5:0]	**WPRI**	HU	SPVP	SPV	GVA	VSBE	**WPRI**
2	6	2	1	1	1	1	1	5

Figure 5.2: Hypervisor status register (`hstatus`) for RV64.

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 3.1 on page 16. In particular, the implementation may make VSXL a read-only field forcing VSXLEN=HSXLEN.

If HSXLEN is changed from 32 to a wider width, and if field VSXL is not hardwired to a forced 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. When VTVM=1, an attempt in VS-mode to execute SFENCE.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 hardwired to zero. Guest external interrupts are explained in Section 5.2.4, and the use of VGEIN is covered further in Section 5.2.3.

Field HU (Hypervisor User 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 trap 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 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 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 (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 memory faults, 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.

5.2.2 Hypervisor Trap Delegation Registers (`hedeleg` and `hideleg`)

Registers `hedeleg` and `hideleg` are HSXLEN-bit read/write registers, formatted as shown in Figures 5.3 and 5.4 respectively. 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`.

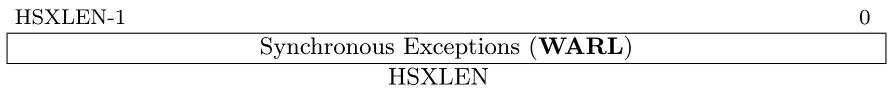

Figure 5.3: Hypervisor exception delegation register (`hedeleg`).

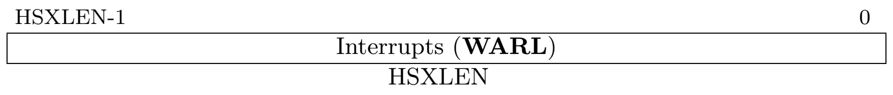

Figure 5.4: Hypervisor interrupt delegation register (`hideleg`).

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
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
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

Table 5.2: Bits of `hedeleg` that must be writable or must be hardwired to zero.

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 hardwired to zero. Many bits of `hedeleg` are required specifically to be writable or zero, as enumerated in Table 5.2. 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.*

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`, only bits 10, 6, and 2 (corresponding to the standard VS-level interrupts) shall be writable, and the others shall be hardwired to zero.

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 or custom interrupt causes (codes 16 and above).

5.2.3 Hypervisor Interrupt Registers (`hvip`, `hip`, and `hie`)

Register `hvip` is an HSXLEN-bit read/write register that a hypervisor can write to indicate virtual interrupts intended for VS-mode. The bit positions writable in `hideleg` shall also be writable in `hvip`, and the other bits of `hvip` shall be hardwired to zeros.

HSXLEN-1   0
Virtual Interrupts (**WARL**)
HSXLEN

Figure 5.5: Hypervisor virtual-interrupt-pending register (`hvip`).

The standard portion (bits 15:0) of `hvip` is formatted as shown in Figure 5.6. 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.

15 11	10	9 7	6	5 3	2	1 0
0	VSEIP	0	VSTIP	0	VSSIP	0
5	1	3	1	3	1	2

Figure 5.6: 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. As with `sip` and `sie`, an interrupt i will be taken in HS-mode if bit i is set in both `hip` and `hie`, and if supervisor-level interrupts are globally enabled.

HSXLEN-1   0
Interrupts (**WARL**)
HSXLEN

Figure 5.7: Hypervisor interrupt-pending register (`hip`).

HSXLEN-1 … 0
Interrupts (**WARL**)
HSXLEN

Figure 5.8: Hypervisor interrupt-enable register (`hie`).

For each writable bit in `sie`, the corresponding bit shall be hardwired to 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.*

If bit i of `sie` is hardwired to 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 hardwired to zero.

The standard portions (bits 15:0) of registers `hip` and `hie` are formatted as shown in Figures 5.9 and 5.10 respectively.

15 13	12	11	10	9 7	6	5 3	2	1 0
0	SGEIP	0	VSEIP	0	VSTIP	0	VSSIP	0
3	1	1	1	3	1	3	1	2

Figure 5.9: Standard portion (bits 15:0) of `hip`.

15 13	12	11	10	9 7	6	5 3	2	1 0
0	SGEIE	0	VSEIE	0	VSTIE	0	VSSIE	0
3	1	1	1	3	1	3	1	2

Figure 5.10: 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 5.2.4.)

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.

5.2.4 Hypervisor Guest External Interrupt Registers (`hgeip` and `hgeie`)

The `hgeip` register is an HSXLEN-bit read-only register, formatted as shown in Figure 5.11, that indicates pending guest external interrupts for this hart. The `hgeie` register is an HSXLEN-bit read/write register, formatted as shown in Figure 5.12, 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`.

HSXLEN-1 1	0
Guest External Interrupts	0
HSXLEN-1	1

Figure 5.11: Hypervisor guest external interrupt-pending register (`hgeip`).

HSXLEN-1 1	0
Guest External Interrupts (**WARL**)	0
HSXLEN-1	1

Figure 5.12: 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 hardwired to 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.

5.2.5 Hypervisor Counter-Enable Register (`hcounteren`)

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.

31	30	29	28 6	5	4	3	2	1	0
HPM31	HPM30	HPM29	...	HPM5	HPM4	HPM3	IR	TM	CY
1	1	1	23	1	1	1	1	1	1

Figure 5.13: Hypervisor counter-enable register (`hcounteren`).

When the CY, TM, IR, or HPMn 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 contain a hardwired value of zero, indicating reads to the corresponding counter will cause an exception when V=1. Hence, they are effectively **WARL** fields.

5.2.6 Hypervisor Time Delta Registers (`htimedelta`, `htimedeltah`)

The `htimedelta` CSR is a 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.*

63 0
`htimedelta`
64

Figure 5.14: Hypervisor time delta register, HSXLEN=64.

For HSXLEN=32 only, `htimedelta` holds the lower 32 bits of the delta, and `htimedeltah` holds the upper 32 bits of the delta.

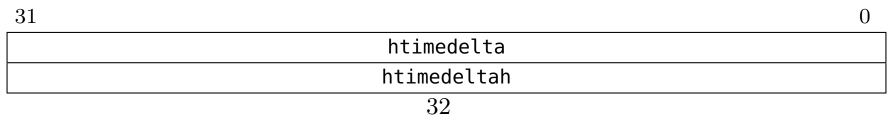

Figure 5.15: Hypervisor time delta registers, HSXLEN=32.

5.2.7 Hypervisor Trap Value Register (`htval`)

The `htval` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.16. 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.

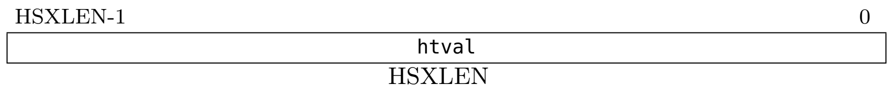

Figure 5.16: Hypervisor trap value register (`htval`).

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 3.6) and in page table entries (Sections 4.3, 4.4, and 4.5).*

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.*

5.2.8 Hypervisor Trap Instruction Register (`htinst`)

The `htinst` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.17. 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 5.6.3.

HSXLEN-1 0
`htinst`
HSXLEN

Figure 5.17: Hypervisor trap instruction register (`htinst`).

`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.

5.2.9 Hypervisor Guest Address Translation and Protection Register (`hgatp`)

The `hgatp` register is an HSXLEN-bit read/write register, formatted as shown in Figure 5.18 for HSXLEN=32 and Figure 5.19 for HSXLEN=64, which controls G-stage address translation and protection, the second stage of two-stage translation for guest virtual addresses (see Section 5.5). 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.

31	30 29	28 22	21 0
MODE	0 (**WARL**)	VMID (**WARL**)	PPN (**WARL**)
1	2	7	22

Figure 5.18: RV32 Hypervisor guest address translation and protection register `hgatp`.

63 60	59 58	57 44	43 0
MODE (**WARL**)	0 (**WARL**)	VMID (**WARL**)	PPN (**WARL**)
4	2	14	44

Figure 5.19: RV64 Hypervisor guest address translation and protection register `hgatp`, for MODE values Bare, Sv39x4, and Sv48x4.

Table 5.3 shows the encodings of the MODE field for RV32 and RV64. 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 3.6. In this case, the remaining fields in `hgatp` must be set to zeros.

For RV32, 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. For RV64, modes Sv39x4 and Sv48x4 are defined as modifications of the Sv39 and Sv48 paged virtual-memory schemes. All of these paged virtual-memory schemes are described in Section 5.5.1. An additional RV64 scheme, Sv57x4, may be defined in a later version of this specification.

The remaining MODE settings for RV64 are reserved for future use and may define different interpretations of the other fields in `hgatp`.

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32x4	Page-based 34-bit virtual addressing (2-bit extension of Sv32).

RV64		
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*	*Reserved for page-based 59-bit virtual addressing.*
11–15	—	*Reserved*

Table 5.3: Encoding of `hgatp` MODE field.

RV64 implementations are not required to support all defined RV64 MODE settings.

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 5.5.1, for the paged virtual-memory schemes (Sv32x4, Sv39x4, and Sv48x4), 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 hardwire PPN[1:0] to 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 and Sv48x4.

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, it may be necessary to execute an HFENCE.GVMA instruction (see Section 5.3.2) before or after writing `hgatp`.

5.2.10 Virtual Supervisor Status Register (`vsstatus`)

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 5.20 when VSXLEN=32 and Figure 5.21 when VSXLEN=64. When V=1, `vsstatus` substitutes for the usual `sstatus`, so instructions that normally read or modify `sstatus` actually access `vsstatus` instead.

31	30 20	19	18	17	16 15	14 13	12 9	8	7	6	5	4 2	1	0
SD	**WPRI**	MXR	SUM	**WPRI**	XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	UBE	SPIE	**WPRI**	SIE	**WPRI**
1	11	1	1	1	2	2	4	1	1	1	1	3	1	1

Figure 5.20: Virtual supervisor status register (`vsstatus`) for RV32.

VSXLEN-1	VSXLEN-2 34	33 32	31 20	19	18	17
SD	**WPRI**	UXL[1:0]	**WPRI**	MXR	SUM	**WPRI**
1	VSXLEN-35	2	12	1	1	1

16 15	14 13	12 9	8	7	6	5	4 2	1	0
XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	UBE	SPIE	**WPRI**	SIE	**WPRI**
2	2	4	1	1	1	1	3	1	1

Figure 5.21: Virtual supervisor status register (`vsstatus`) for RV64.

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 3.1 on page 16. In particular, the implementation may hardwire field UXL so that VU-mode XLEN=VSXLEN.

If VSXLEN is changed from 32 to a wider width, and if field UXL is not hardwired to a forced 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.*

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 a read-only alias 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.

5.2.11 Virtual Supervisor Interrupt Registers (`vsip` and `vsie`)

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 Figures 5.22 and 5.23 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.

VSXLEN-1 … 0
Interrupts (**WARL**)
VSXLEN

Figure 5.22: Virtual supervisor interrupt-pending register (`vsip`).

VSXLEN-1 … 0
Interrupts (**WARL**)
VSXLEN

Figure 5.23: Virtual supervisor interrupt-enable register (`vsie`).

The standard portions (bits 15:0) of registers `vsip` and `vsie` are formatted as shown in Figures 5.24 and 5.25 respectively.

15 … 10	9	8 … 6	5	4 … 2	1	0
0	SEIP	0	STIP	0	SSIP	0
6	1	3	1	3	1	1

Figure 5.24: Standard portion (bits 15:0) of `vsip`.

15 … 10	9	8 … 6	5	4 … 2	1	0
0	SEIE	0	STIE	0	SSIE	0
6	1	3	1	3	1	1

Figure 5.25: Standard portion (bits 15:0) of `vsie`.

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.

5.2.12 Virtual Supervisor Trap Vector Base Address Register (`vstvec`)

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 5.26. 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.

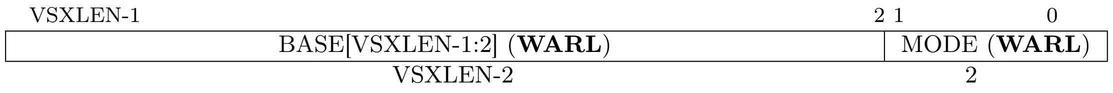

Figure 5.26: Virtual supervisor trap vector base address register (`vstvec`).

5.2.13 Virtual Supervisor Scratch Register (`vsscratch`)

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 5.27. 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.

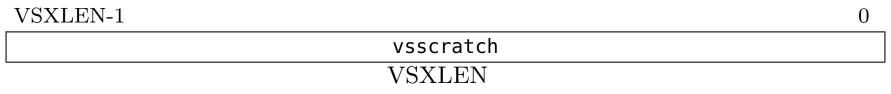

Figure 5.27: Virtual supervisor scratch register (`vsscratch`).

5.2.14 Virtual Supervisor Exception Program Counter (`vsepc`)

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 5.28. 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.

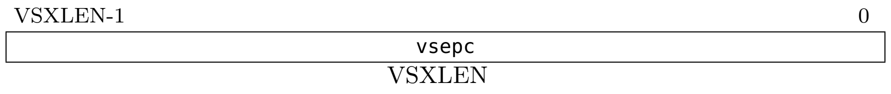

Figure 5.28: Virtual supervisor exception program counter (`vsepc`).

5.2.15 Virtual Supervisor Cause Register (`vscause`)

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 5.29. 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.

VSXLEN-1	VSXLEN-2 … 0
Interrupt	Exception Code (**WLRL**)
1	VSXLEN-1

Figure 5.29: Virtual supervisor cause register (`vscause`).

5.2.16 Virtual Supervisor Trap Value Register (`vstval`)

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 5.30. 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.

VSXLEN-1 … 0
`vstval`
VSXLEN

Figure 5.30: Virtual supervisor trap value register (`vstval`).

5.2.17 Virtual Supervisor Address Translation and Protection Register (`vsatp`)

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 5.31 for VSXLEN=32 and Figure 5.32 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 5.5).

31	30 … 22	21 … 0
MODE (**WARL**)	ASID (**WARL**)	PPN (**WARL**)
1	9	22

Figure 5.31: RV32 virtual supervisor address translation and protection register `vsatp`.

63 … 60	59 … 44	43 … 0
MODE (**WARL**)	ASID (**WARL**)	PPN (**WARL**)
4	16	44

Figure 5.32: RV64 virtual supervisor address translation and protection register `vsatp`, for MODE values Bare, Sv39, and Sv48.

When V=0, a write to `vsatp` with an unsupported MODE value is not ignored as it is for `satp`. Instead, the fields of `vsatp` are **WARL** in the normal way.

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.

5.3 Hypervisor Instructions

The hypervisor extension adds virtual-machine load and store instructions and two privileged fence instructions.

5.3.1 Hypervisor Virtual-Machine Load and Store Instructions

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
HLV.*width*	[U]	addr	PRIVM	dest	SYSTEM
HLVX.HU/WU	HLVX	addr	PRIVM	dest	SYSTEM
HSV.*width*	src	addr	PRIVM	0	SYSTEM

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 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. 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.)

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.

Attempts to execute a virtual-machine load/store instruction (HLV, HLVX, or HSV) when V=1 cause a virtual instruction trap. Attempts to execute one of these same instructions from U-mode when `hstatus`.HU=0 cause an illegal instruction trap.

5.3.2 Hypervisor Memory-Management Fence Instructions

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
HFENCE.VVMA	asid	vaddr	PRIV	0	SYSTEM
HFENCE.GVMA	vmid	gaddr	PRIV	0	SYSTEM

The hypervisor memory-management fence instructions, HFENCE.VVMA and HFENCE.GVMA, perform a function similar to SFENCE.VMA (Section 4.2.1), 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 subsequent implicit reads by that hart of the VS-level memory-management data structures, when those implicit reads are 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 use and 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 subsequent implicit reads by that hart of guest-physical memory-management data structures done 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).

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 use and 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.*

Attempts to execute HFENCE.VVMA or HFENCE.GVMA when V=1 cause a virtual instruction trap, while attempts to do the same in U-mode cause an illegal instruction trap. Attempting to execute HFENCE.GVMA in HS-mode when `mstatus`.TVM=1 also causes an illegal instruction trap.

5.4 Machine-Level CSRs

The hypervisor extension augments or modifies machine CSRs `mstatus`, `mstatush`, `mideleg`, `mip`, and `mie`, and adds CSRs `mtval2` and `mtinst`.

5.4.1 Machine Status Registers (`mstatus` and `mstatush`)

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 5.33 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`, which must exist. Figure 5.34 shows the `mstatush` register when the hypervisor extension is implemented and MXLEN=32.

MXLEN-1	MXLEN-2 … 40	39	38	37	36	35 34	33 32
SD	**WPRI**	MPV	GVA	MBE	SBE	SXL[1:0]	UXL[1:0]
1	MXLEN-41	1	1	1	1	2	2

31 … 23	22	21	20	19	18	17	16 15	14 13
WPRI	TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]	FS[1:0]
9	1	1	1	1	1	1	2	2

12 11	10 9	8	7	6	5	4	3	2	1	0
MPP[1:0]	**WPRI**	SPP	MPIE	UBE	SPIE	**WPRI**	MIE	**WPRI**	SIE	**WPRI**
2	2	1	1	1	1	1	1	1	1	1

Figure 5.33: Machine status register (`mstatus`) for RV64 when the hypervisor extension is implemented.

31 … 8	7	6	5	4	3 … 0
WPRI	MPV	GVA	MBE	SBE	**WPRI**
24	1	1	1	1	4

Figure 5.34: Additional machine status register (`mstatush`) 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 bit is set to the 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 (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, but has no effect on accesses to `vsatp` or instruction HFENCE.VVMA.

The hypervisor extension changes the behavior of the 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 privilege mode were set to MPP. Table 5.4 enumerates the cases.

MPRV	MPV	MPP	Effect
0	–	–	Normal access; current privilege and virtualization modes apply.
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.

Table 5.4: Effect of MPRV on the translation and protection of explicit memory accesses.

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 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`.

5.4.2 Machine Interrupt Delegation Register (`mideleg`)

When the hypervisor extension is implemented, bits 10, 6, and 2 of `mideleg` (corresponding to the standard VS-level interrupts) are each hardwired to 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 hardwired to one. VS-level interrupts and guest external interrupts are always delegated past M-mode to HS-mode.

5.4.3 Machine Interrupt Registers (`mip` and `mie`)

The hypervisor extension gives registers `mip` and `mie` additional active bits for the hypervisor-added interrupts. Figures 5.35 and 5.36 show the standard portions (bits 15:0) of registers `mip` and `mie` when the hypervisor extension is implemented.

15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	SGEIP	MEIP	VSEIP	SEIP	0	MTIP	VSTIP	STIP	0	MSIP	VSSIP	SSIP	0
3	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 5.35: Standard portion (bits 15:0) of `mip`.

15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	SGEIE	MEIE	VSEIE	SEIE	0	MTIE	VSTIE	STIE	0	MSIE	VSSIE	SSIE	0
3	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 5.36: 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`.

5.4.4 Machine Second Trap Value Register (`mtval2`)

The `mtval2` register is an MXLEN-bit read/write register formatted as shown in Figure 5.37. 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.

MXLEN-1 0
`mtval2`
MXLEN

Figure 5.37: 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.

5.4.5 Machine Trap Instruction Register (`mtinst`)

The `mtinst` register is an MXLEN-bit read/write register formatted as shown in Figure 5.38. 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 5.6.3.

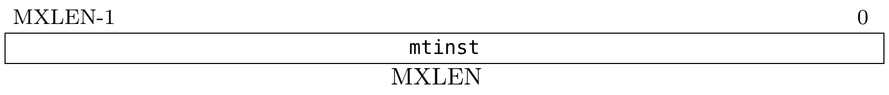

Figure 5.38: Machine trap instruction register (`mtinst`).

`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.

5.5 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, 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.

5.5.1 Guest Physical Address Translation

The mapping of guest physical addresses to supervisor physical addresses is controlled by CSR `hgatp` (Section 5.2.9).

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, or Sv48x4, G-stage address translation is a variation on the usual page-based virtual address translation scheme of Sv32, Sv39, or Sv48, respectively. In each case, the size of the incoming address is widened by 2 bits (to 34, 41, or 50 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, or Sv48 are adopted unchanged for G-stage translation. Non-root page tables and all page table entries (PTEs) have the same formats as documented in Sections 4.3, 4.4, and 4.5.

For Sv32x4, an incoming guest physical address is partitioned into a virtual page number (VPN) and page offset as shown in Figure 5.39. This partitioning is identical to that for an Sv32 virtual address as depicted in Figure 4.15 (page 73), 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 4.16.)

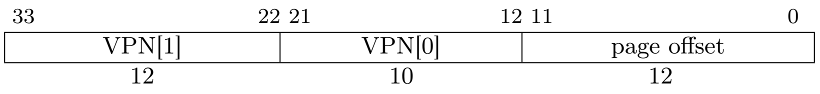

Figure 5.39: Sv32x4 virtual address (guest physical address).

For Sv39x4, an incoming guest physical address is partitioned as shown in Figure 5.40. This partitioning is identical to that for an Sv39 virtual address as depicted in Figure 4.18 (page 77), 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.

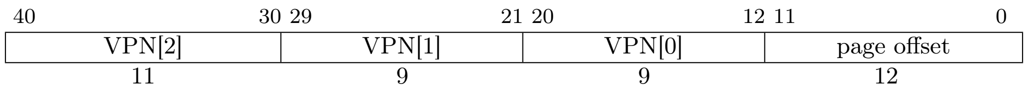

Figure 5.40: Sv39x4 virtual address (guest physical address).

For Sv48x4, an incoming guest physical address is partitioned as shown in Figure 5.41. This partitioning is identical to that for an Sv48 virtual address as depicted in Figure 4.21 (page 78), 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.

49 39	38 30	29 21	20 12	11 0
VPN[3]	VPN[2]	VPN[1]	VPN[0]	page offset
11	9	9	9	12

Figure 5.41: Sv48x4 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, or Sv48x4 guest physical address is accomplished with the same algorithm used for Sv32, Sv39, or Sv48, as presented in Section 4.3.2, except that:

- in step 1, $a =$ `hgatp`.PPN $\times$ PAGESIZE;
- 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 are checked as though for a 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).

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.

5.5.2 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 operating 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 5.6.3.

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`.

5.5.3 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 5.3.2.

Section 3.6.2 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. If G-stage address translation is in use and is not Bare, synchronization with its data structures is also needed. When PMP settings are modified in a manner that affects either the physical memory that holds guest-physical page tables or the physical memory to which guest-physical page tables point, an HFENCE.GVMA instruction with *rs1*=`x0` and *rs2*=`x0` must be executed in M-mode after the PMP CSRs are written. An HFENCE.VVMA instruction is not required.

5.6 Traps

5.6.1 Trap Cause Codes

The hypervisor extension augments the trap cause encoding. Table 5.5 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.

HS-mode and VS-mode ECALLs use different cause values so they can be delegated separately.

When V=1, a virtual instruction trap (not an illegal instruction trap) is taken for:

- attempts to access a counter CSR when the corresponding bit in `hcounteren` is 0 and the same bit in `mcounteren` is 1;
- attempts to execute a hypervisor instruction (HLV, HLVX, HSV, or HFENCE) or to access an implemented hypervisor CSR or VS CSR;
- in VU-mode, attempts to execute WFI or a supervisor instruction (SRET or SFENCE), or to access an implemented supervisor CSR;
- 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; or
- in VS-mode, attempts to execute an SFENCE instruction or to access `satp`, when `hstatus`.VTVM=1.

On a virtual instruction trap, `mtval` or `stval` is written the same as for an illegal instruction trap.

When V=1, privileged instructions that are invalid in VS-mode or VU-mode generally cause a virtual instruction trap instead of an illegal instruction trap. The same goes for attempts to access hypervisor- or supervisor-level CSRs that fail due to insufficient privilege when V=1, or attempts to access CSRs to which access has been expressly disabled by a hypervisor CSR (e.g. `hcounteren`*). It is not unusual that hypervisors must emulate such instructions, to support nested hypervisors or for other reasons. When not emulating an instruction, a hypervisor should convert a virtual instruction trap into an illegal instruction exception for the guest virtual machine.*

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.

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	13–15	*Reserved*
1	≥16	*Available for platform or custom 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 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	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

Table 5.5: Machine and supervisor cause register (`mcause` and `scause`) values when the hypervisor extension is implemented.

5.6.2 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 5.6. A trap into M-mode also writes fields GVA, MPIE, and MIE in `mstatus`/`mstatush` and writes CSRs `mepc`, `mcause`, `mtval`, `mtval2`, and `mtinst`.

Previous Mode	MPV	MPP
U-mode	0	0
HS-mode	0	1
M-mode	0	3
VU-mode	1	0
VS-mode	1	1

Table 5.6: Value of `mstatus`/`mstatush` fields MPV and MPP after a trap into M-mode. Upon trap return, MPV is ignored when MPP=3.

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 5.7. 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`.

Previous Mode	SPV	SPP
U-mode	0	0
HS-mode	0	1
VU-mode	1	0
VS-mode	1	1

Table 5.7: Value of `hstatus` field SPV and `sstatus` field SPP after a trap into HS-mode.

When a trap is taken into VS-mode, `vsstatus`.SPP is set according to Table 5.8. 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`.

Previous Mode	SPP
VU-mode	0
VS-mode	1

Table 5.8: Value of `vsstatus` field SPP after a trap into VS-mode.

5.6.3 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 nonstandard); 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 available for a custom instruction (*not* an unused reserved encoding).

  This is a *custom value*. The instruction that trapped is a nonstandard 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 5.9 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 nonstandard. A future standard or extension may permit other values to be written, chosen from the set of allowed values established earlier.

Exception	Zero	Transformed Standard Instruction	Custom Value	Pseudo-instruction 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

Table 5.9: Values that may be automatically written to the trap instruction register (`mtinst` or `htinst`) on an exception trap.

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, or FLQ, the transformed instruction has the format shown in Figure 5.42.

31 25	24 20	19 15	14 12	11 7	6 0
0	0	Addr. Offset	funct3	rd	opcode
7	5	5	3	5	7

Figure 5.42: Transformed noncompressed load instruction (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, or FLQ). 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, or FSQ, the transformed instruction has the format shown in Figure 5.43.

31 25	24 20	19 15	14 12	11 7	6 0
0	rs2	Addr. Offset	funct3	0	opcode
7	5	5	3	5	7

Figure 5.43: Transformed noncompressed store instruction (SB, SH, SW, SD, FSW, FSD, or FSQ). 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 5.44.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	Addr. Offset	funct3	rd	opcode
5	1	1	5	5	3	5	7

Figure 5.44: Transformed atomic instruction (load-reserved, store-conditional, or AMO instruction). 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 5.45.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	Addr. Offset	funct3	rd	opcode
7	5	5	3	5	7

Figure 5.45: 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, or FLQ) or basic store (SB, SH, SW, SD, FSW, FSD, or FSQ), 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 compatiblity, 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 5.10; zero is not allowed.

Value	Meaning
`0x00002000`	32-bit read for VS-stage address translation (RV32)
`0x00002020`	32-bit write for VS-stage address translation (RV32)
`0x00003000`	64-bit read for VS-stage address translation (RV64)
`0x00003020`	64-bit write for VS-stage address translation (RV64)

Table 5.10: Special pseudoinstruction values for guest-page faults. The RV32 values are used when VSXLEN=32, and the RV64 values when VSXLEN=64.

The defined pseudoinstruction values are designed to correspond closely with the encodings of basic loads and stores, as illustrated by Table 5.11.

Encoding	Instruction
`0x00002003`	`lw x0,0(x0)`
`0x00002023`	`sw x0,0(x0)`
`0x00003003`	`ld x0,0(x0)`
`0x00003023`	`sd x0,0(x0)`

Table 5.11: Standard instructions corresponding to the special pseudoinstructions of Table 5.10.

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 hardwired entirely to 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.

5.6.4 Trap Return

The MRET instruction is used to return from a trap taken into M-mode. MRET first determines what the new operating mode will be according to the values of MPP and MPV in `mstatus` or `mstatush`, as encoded in Table 5.6. MRET then in `mstatus`/`mstatush` sets MPV=0, MPP=0, MIE=MPIE, and MPIE=1. Lastly, MRET sets the virtualization and privilege modes 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 operating mode will be according to the values in `hstatus`.SPV and `sstatus`.SPP, as encoded in Table 5.7. SRET then sets `hstatus`.SPV=0, and in `sstatus` sets SPP=0, SIE=SPIE, and SPIE=1. Lastly, SRET sets the virtualization and privilege modes as previously determined, and sets `pc`=`sepc`.

When executed in VS-mode (i.e., V=1), SRET sets the privilege mode according to Table 5.8, in `vsstatus` sets SPP=0, SIE=SPIE, and SPIE=1, and lastly sets `pc`=`vsepc`.

Chapter 6

"N" Standard Extension for User-Level Interrupts, Version 1.1

This is a placeholder for a more complete writeup of the N extension, and to form a basis for discussion.

An ongoing topic of discussion is whether, for systems needing only M and U privilege modes, the N extension should be supplanted by S-mode without virtual memory (i.e., with `satp` *hardwired to zero). This approach would have similar hardware cost and would simplify the architecture.*

This chapter presents a proposal for adding RISC-V user-level interrupt and exception handling. When the N extension is present, and the outer execution environment has delegated designated interrupts and exceptions to user-level, then hardware can transfer control directly to a user-level trap handler without invoking the outer execution environment.

User-level interrupts are primarily intended to support secure embedded systems with only M-mode and U-mode present, but can also be supported in systems running Unix-like operating systems to support user-level trap handling.

When used in an Unix environment, the user-level interrupts would likely not replace conventional signal handling, but could be used as a building block for further extensions that generate user-level events such as garbage collection barriers, integer overflow, floating-point traps.

6.1 Additional CSRs

New user-visible CSRs are added to support the N extension. Their encodings are listed in Table 2.2 in Chapter 2.

6.1.1 User Status Register (`ustatus`)

The `ustatus` register is a UXLEN-bit read/write register formatted as shown in Figure 6.1. The `ustatus` register keeps track of and controls the hart's current operating state.

UXLEN 5	4	3 1	0
WPRI	UPIE	**WPRI**	UIE
UXLEN-5	1	3	1

Figure 6.1: User-mode status register (`ustatus`).

The user interrupt-enable bit UIE disables user-level interrupts when clear. The value of UIE is copied into UPIE when a user-level trap is taken, and the value of UIE is set to zero to provide atomicity for the user-level trap handler.

The UIE and UPIE bits are mirrored in the `mstatus` and `sstatus` registers in the same bit positions.

There is no UPP bit to hold the previous privilege mode as it can only be user mode.

A new instruction, URET, is used to return from traps in U-mode. URET copies UPIE into UIE, then sets UPIE, before copying `uepc` to the `pc`.

UPIE is set after the UPIE/UIE stack is popped to enable interrupts and help catch coding errors.

6.1.2 User Interrupt Registers (`uip` and `uie`)

The `uip` register is a UXLEN-bit read/write register containing information on pending interrupts, while `uie` is the corresponding UXLEN-bit read/write register containing interrupt enable bits.

UXLEN-1 9	8	7 5	4	3 1	0
WPRI	UEIP	**WPRI**	UTIP	**WPRI**	USIP
UXLEN-9	1	3	1	3	1

Figure 6.2: User interrupt-pending register (`uip`).

UXLEN-1 9	8	7 5	4	3 1	0
WPRI	UEIE	**WPRI**	UTIE	**WPRI**	USIE
UXLEN-9	1	3	1	3	1

Figure 6.3: User interrupt-enable register (`uie`).

Three types of interrupts are defined: software interrupts, timer interrupts, and external interrupts. A user-level software interrupt is triggered on the current hart by writing 1 to its user software interrupt-pending (USIP) bit in the `uip` register. A pending user-level software interrupt can be cleared by writing 0 to the USIP bit in `uip`. User-level software interrupts are disabled when the USIE bit in the `uie` register is clear.

The ABI should provide a mechanism to send interprocessor interrupts to other harts, which will ultimately cause the USIP bit to be set in the recipient hart's `uip` register.

All bits besides USIP in the `uip` register are read-only.

A user-level timer interrupt is pending if the UTIP bit in the `uip` register is set. User-level timer interrupts are disabled when the UTIE bit in the `uie` register is clear. The ABI should provide a mechanism to clear a pending timer interrupt.

A user-level external interrupt is pending if the UEIP bit in the `uip` register is set. User-level external interrupts are disabled when the UEIE bit in the `uie` register is clear. The ABI should provide facilities to mask, unmask, and query the cause of external interrupts.

The `uip` and `uie` registers are subsets of the `mip` and `mie` registers. Reading any field, or writing any writable field, of `uip`/`uie` effects a read or write of the homonymous field of `mip`/`mie`. If S-mode is implemented, the `uip` and `uie` registers are also subsets of the `sip` and `sie` registers.

6.1.3 Machine Trap Delegation Registers (`medeleg` and `mideleg`)

In systems with the N extension, the `medeleg` and `mideleg` registers, described in Chapter 3, must be implemented.

In systems that implement S-mode, `medeleg` and `mideleg` behave as described in Chapter 3. In systems with only M and U privilege modes, setting a bit in `medeleg` or `mideleg` delegates the corresponding trap in U-mode to the U-mode trap handler.

6.1.4 Supervisor Trap Delegation Registers (`sedeleg` and `sideleg`)

For systems with both S-mode and the N extension, new CSRs `sedeleg` and `sideleg` are added. These registers have the same layout as the machine trap delegation registers, `medeleg` and `mideleg`.

`sedeleg` and `sideleg` allow S-mode to delegate traps to U-mode. Only bits corresponding to traps that have been delegated to S-mode are writable; the others are hardwired to zero. Setting a bit in `sedeleg` or `sideleg` delegates the corresponding trap in U-mode to the U-mode trap handler.

6.1.5 Other CSRs

The `uscratch`, `uepc`, `ucause`, `utvec`, and `utval` CSRs are defined analogously to the `mscratch`, `mepc`, `mcause`, `mtvec`, and `mtval` CSRs.

A more complete writeup is to follow.

6.2 N Extension Instructions

The URET instruction is added to perform the analogous function to MRET and SRET.

6.3 Reducing Context-Swap Overhead

The user-level interrupt-handling registers add considerable state to the user-level context, yet will usually rarely be active in normal use. In particular, `uepc`, `ucause`, and `utval` are only valid during execution of a trap handler.

An NS field can be added to `mstatus` and `sstatus` following the format of the FS and XS fields to reduce context-switch overhead when the values are not live. Execution of URET will place the `uepc`, `ucause`, and `utval` back into initial state.

Chapter 7

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

0000000	00010	00000	000	00000	1110011	URET
0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET

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
0110010	00011	rs1	100	rd	1110011	HLVX.HU
0110100	00000	rs1	100	rd	1110011	HLV.W
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

Table 7.1: RISC-V Privileged Instructions

Chapter 8

History

8.1 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.

Appendix A

Formal Specification in Sail

This appendix includes selections from the Sail formal model of the RISC-V ISA that highlight aspects of privileged mode operation. Please refer to the complete Sail model at https://github.com/rems-project/sail-riscv for details not covered in this section.

A.1 Basic Definitions

For clarity, this section defines types and functions that commonly occur in the following sections. *ctl_result* defines the type of control flow changes involving a privilege transition, where *ExceptionType* is an enumeration of the values in Table 3.6 on page 40. *tval* unwraps the value in optional *excinfo* embedded in a *sync_exception*. *tvec_addr* generates the trap vector address from the current value of the *Mtvec* register. *checked_mem_read* and *checked_mem_write* perform checked accesses to physical addresses.

```
function privLevel_to_bits (p) =
 match (p) {
  User => 0b00,
  Supervisor => 0b01,
  Machine => 0b11
 }
```

```
struct sync_exception = {
 trap : ExceptionType,
 excinfo : option(xlenbits),
 ext : option(ext_exception) /* for extensions */
}
```

```
union ctl_result = {
 CTL_TRAP : sync_exception,
 CTL_SRET : unit,
 CTL_MRET : unit,
 CTL_URET : unit
}
```

```
function tval(excinfo : option(xlenbits)) -> xlenbits = {
  match (excinfo) {
    Some(e) => e,
    None() => EXTZ(0b0)
  }
}
```

```
function tvec_addr(m : Mtvec, c : Mcause) -> option(xlenbits) = {
  let base : xlenbits = m.Base() @ 0b00;
  match (trapVectorMode_of_bits(m.Mode())) {
    TV_Direct => Some(base),
    TV_Vector => if c.IsInterrupt() == 0b1
                 then Some(base + (EXTZ(c.Cause()) << 2))
                 else Some(base),
    TV_Reserved => None()
  }
}
```

```
function pc_alignment_mask() -> xlenbits =
  ~(EXTZ(if misa.C() == 0b1 then 0b00 else 0b10))
```

```
function checked_mem_read forall 'n, 0 < 'n <= max_mem_access . (t : AccessType(
    ↪ ext_access_type), paddr : xlenbits, width : atom('n), aq : bool, rl : bool,
    ↪ res: bool, meta : bool) -> MemoryOpResult((bits(8 * 'n), mem_meta)) =
  if within_mmio_readable(paddr, width)
  then MemoryOpResult_add_meta(mmio_read(t, paddr, width), default_meta)
  else if within_phys_mem(paddr, width)
  then phys_mem_read(t, paddr, width, aq, rl, res, meta)
  else match t {
    Execute() => MemException(E_Fetch_Access_Fault()),
    Read(Data) => MemException(E_Load_Access_Fault()),
    _ => MemException(E_SAMO_Access_Fault())
  }
```

```
function checked_mem_write forall 'n, 0 < 'n <= max_mem_access . (wk : write_kind,
    ↪ paddr : xlenbits, width : atom('n), data: bits(8 * 'n), meta: mem_meta) ->
    ↪ MemoryOpResult(bool) =
  if within_mmio_writable(paddr, width)
  then mmio_write(paddr, width, data)
  else if within_phys_mem(paddr, width)
  then phys_mem_write(wk, paddr, width, data, meta)
  else MemException(E_SAMO_Access_Fault())
```

A.2 CSR access control

These functions specify the access control checks for CSR registers when they are accessed by CSR instructions, and collect together in one location the CSR access checks described in various earlier sections of this volume.

```
function csrAccess(csr : csreg) -> csrRW = csr[11..10]
```

```
function csrPriv(csr : csreg) -> priv_level = csr[9..8]
function is_CSR_defined (csr : csreg, p : Privilege) -> bool =
 match (csr) {
  /* machine mode: informational */
  0xf11 => p == Machine, // mvendorid
  0xf12 => p == Machine, // marchdid
  0xf13 => p == Machine, // mimpid
  0xf14 => p == Machine, // mhartid
  /* machine mode: trap setup */
  0x300 => p == Machine, // mstatus
  0x301 => p == Machine, // misa
  0x302 => p == Machine, // medeleg
  0x303 => p == Machine, // mideleg
  0x304 => p == Machine, // mie
  0x305 => p == Machine, // mtvec
  0x306 => p == Machine, // mcounteren
  /* machine mode: trap handling */
  0x340 => p == Machine, // mscratch
  0x341 => p == Machine, // mepc
  0x342 => p == Machine, // mcause
  0x343 => p == Machine, // mtval
  0x344 => p == Machine, // mip

  0x3A0 => p == Machine, // pmpcfg0
  0x3A1 => p == Machine & (sizeof(xlen) == 32), // pmpcfg1
  0x3A2 => p == Machine, // pmpcfg2
  0x3A3 => p == Machine & (sizeof(xlen) == 32), // pmpcfg3

  0x3B0 => p == Machine, // pmpaddr0
  0x3B1 => p == Machine, // pmpaddr1
  0x3B2 => p == Machine, // pmpaddr2
  0x3B3 => p == Machine, // pmpaddr3
  0x3B4 => p == Machine, // pmpaddr4
  0x3B5 => p == Machine, // pmpaddr5
  0x3B6 => p == Machine, // pmpaddr6
  0x3B7 => p == Machine, // pmpaddr7
  0x3B8 => p == Machine, // pmpaddr8
  0x3B9 => p == Machine, // pmpaddr9
  0x3BA => p == Machine, // pmpaddrA
  0x3BB => p == Machine, // pmpaddrB
  0x3BC => p == Machine, // pmpaddrC
  0x3BD => p == Machine, // pmpaddrD
  0x3BE => p == Machine, // pmpaddrE
  0x3BF => p == Machine, // pmpaddrF

  /* counters */
  0xB00 => p == Machine, // mcycle
  0xB02 => p == Machine, // minstret

  0xB80 => p == Machine & (sizeof(xlen) == 32), // mcycleh
  0xB82 => p == Machine & (sizeof(xlen) == 32), // minstreth

  /* disabled trigger/debug module */
  0x7a0 => p == Machine,

  /* supervisor mode: trap setup */
  0x100 => haveSupMode() & (p == Machine | p == Supervisor), // sstatus
  0x102 => haveSupMode() & (p == Machine | p == Supervisor), // sedeleg
  0x103 => haveSupMode() & (p == Machine | p == Supervisor), // sideleg
  0x104 => haveSupMode() & (p == Machine | p == Supervisor), // sie
  0x105 => haveSupMode() & (p == Machine | p == Supervisor), // stvec
  0x106 => haveSupMode() & (p == Machine | p == Supervisor), // scounteren

  /* supervisor mode: trap handling */
  0x140 => haveSupMode() & (p == Machine | p == Supervisor), // sscratch
  0x141 => haveSupMode() & (p == Machine | p == Supervisor), // sepc
  0x142 => haveSupMode() & (p == Machine | p == Supervisor), // scause
  0x143 => haveSupMode() & (p == Machine | p == Supervisor), // stval
  0x144 => haveSupMode() & (p == Machine | p == Supervisor), // sip

  /* supervisor mode: address translation */
  0x180 => haveSupMode() & (p == Machine | p == Supervisor), // satp

  /* user mode: counters */
  0xC00 => p == User, // cycle
  0xC01 => p == User, // time
  0xC02 => p == User, // instret

  0xC80 => p == User & (sizeof(xlen) == 32), // cycleh
  0xC81 => p == User & (sizeof(xlen) == 32), // timeh
  0xC82 => p == User & (sizeof(xlen) == 32), // instreth

  /* check extensions */
  _ => ext_is_CSR_defined(csr, p)
 }
```

```
function check_CSR_access(csrrw, csrpr, p, isWrite) =
   (~ (isWrite == true & csrrw == 0b11)) /* read/write */
 & (privLevel_to_bits(p) >=_u csrpr) /* privilege */
```

```
function check_TVM_SATP(csr : csreg, p : Privilege) -> bool =
 ~ (csr == 0x180 & p == Supervisor & mstatus.TVM() == 0b1)
```

```
function check_Counteren(csr : csreg, p : Privilege) -> bool =
 match(csr, p) {
   (0xC00, Supervisor) => mcounteren.CY() == 0b1,
   (0xC01, Supervisor) => mcounteren.TM() == 0b1,
   (0xC02, Supervisor) => mcounteren.IR() == 0b1,

   (0xC00, User) => mcounteren.CY() == 0b1 & ((~ (haveSupMode())) | scounteren.CY()
       ↪ == 0b1),
   (0xC01, User) => mcounteren.TM() == 0b1 & ((~ (haveSupMode())) | scounteren.TM()
       ↪ == 0b1),
   (0xC02, User) => mcounteren.IR() == 0b1 & ((~ (haveSupMode())) | scounteren.IR()
       ↪ == 0b1),

   (_, _) => /* no HPM counters for now */
           if 0xC03 <=_u csr & csr <=_u 0xC1F
           then false
           else true
 }
```

```
function check_CSR(csr : csreg, p : Privilege, isWrite : bool) -> bool =
   is_CSR_defined(csr, p)
 & check_CSR_access(csrAccess(csr), csrPriv(csr), p, isWrite)
 & check_TVM_SATP(csr, p)
 & check_Counteren(csr, p)
```

A.3 Interrupt Delegation and Dispatch

dispatchInterrupt determines the interrupt that should be dispatched, if any, and the delegated dispatch privilege level, from the current values of the *mip*, *mie* and *mstatus* registers and the current privilege level. The model currently assumes that the S-mode is always enabled and does not model the N standard extension.

```
function findPendingInterrupt(ip : xlenbits) -> option(InterruptType) = {
  let ip = Mk_Minterrupts(ip);
  if ip.MEI() == 0b1 then Some(I_M_External)
  else if ip.MSI() == 0b1 then Some(I_M_Software)
  else if ip.MTI() == 0b1 then Some(I_M_Timer)
  else if ip.SEI() == 0b1 then Some(I_S_External)
  else if ip.SSI() == 0b1 then Some(I_S_Software)
  else if ip.STI() == 0b1 then Some(I_S_Timer)
  else if ip.UEI() == 0b1 then Some(I_U_External)
  else if ip.USI() == 0b1 then Some(I_U_Software)
  else if ip.UTI() == 0b1 then Some(I_U_Timer)
  else None()
}
```

```
union interrupt_set = {
  Ints_Pending : xlenbits,
  Ints_Delegated : xlenbits,
  Ints_Empty : unit
}
```

```
function processPending(xip : Minterrupts, xie : Minterrupts, xideleg : xlenbits,
                        priv_enabled : bool) -> interrupt_set = {
  /* interrupts that are enabled but not delegated are pending */
  let effective_pend = xip.bits() & xie.bits() & (~ (xideleg));
  /* the others are delegated */
  let effective_delg = xip.bits() & xideleg;
  /* we have pending interrupts if this privilege is enabled */
  if priv_enabled & (effective_pend != EXTZ(0b0))
  then Ints_Pending(effective_pend)
  else if effective_delg != EXTZ(0b0)
  then Ints_Delegated(effective_delg)
  else Ints_Empty()
}
```

```
function getPendingSet(priv : Privilege) -> option((xlenbits, Privilege)) = {
  assert(haveUsrMode(), "no user mode: M/U or M/S/U system required");
  let effective_pending = mip.bits() & mie.bits();
  if effective_pending == EXTZ(0b0) then None() /* fast path */
  else {
    /* Higher privileges than the current one are implicitly enabled,
     * while lower privileges are blocked. An unsupported privilege is
     * considered blocked.
     */
    let mIE = priv != Machine | (priv == Machine & mstatus.MIE() == 0b1);
    let sIE = haveSupMode() & (priv == User | (priv == Supervisor & mstatus.SIE() ==
        ↪ 0b1));
    let uIE = haveNExt() & (priv == User & mstatus.UIE() == 0b1);
    match processPending(mip, mie, mideleg.bits(), mIE) {
```

```
    Ints_Empty() => None(),
    Ints_Pending(p) => let r = (p, Machine) in Some(r),
    Ints_Delegated(d) =>
      if (~ (haveSupMode())) then {
        if uIE then let r = (d, User) in Some(r)
        else None()
      } else {
        /* the delegated bits are pending for S-mode */
        match processPending(Mk_Minterrupts(d), mie, sideleg.bits(), sIE) {
          Ints_Empty() => None(),
          Ints_Pending(p) => let r = (p, Supervisor) in Some(r),
          Ints_Delegated(d) => if uIE
                               then let r = (d, User) in Some(r)
                               else None()
        }
      }
    }
  }
}
```

```
function dispatchInterrupt(priv : Privilege) -> option((InterruptType, Privilege)) =
    ↪ {
  /* If we don't have different privilege levels, we don't need to check delegation.
   * Absence of U-mode implies absence of S-mode.
   */
  if (~ (haveUsrMode())) | ((~ (haveSupMode())) & (~ (haveNExt()))) then {
    assert(priv == Machine, "invalid current privilege");
    let enabled_pending = mip.bits() & mie.bits();
    match findPendingInterrupt(enabled_pending) {
      Some(i) => let r = (i, Machine) in Some(r),
      None() => None()
    }
  } else {
    match getPendingSet(priv) {
      None() => None(),
      Some(ip, p) => match findPendingInterrupt(ip) {
                       None() => None(),
                       Some(i) => let r = (i, p) in Some(r)
                     }
    }
  }
}
```

A.4 Trap vector and trap return addresses

tvec_addr describes how the `xtvec` registers are interpreted, while *prepare_trap_vector* describes how the addresses in the `xtvec` registers are used. The `xepc` registers are used by *get_trap_vector* and *set_trap_vector* for computing the return addresses from traps.

```
function prepare_trap_vector(p : Privilege, cause : Mcause) -> xlenbits = {
  let tvec : Mtvec = match p {
                       Machine => mtvec,
                 Supervisor => stvec,
                 User => utvec
               };
  match tvec_addr(tvec, cause) {
    Some(epc) => epc,
    None() => internal_error("Invalid tvec mode")
  }
}
```

```
function get_xret_target(p) =
  match p {
    Machine => mepc,
    Supervisor => sepc,
    User => uepc
  }
```

```
function set_xret_target(p, value) = {
  let target = legalize_xepc(value);
  match p {
    Machine => mepc = target,
    Supervisor => sepc = target,
    User => uepc = target
  };
  target
}
```

A.5 Trap handling and privilege transition

exception_delegatee computes the delegated privilege level at which an exception should be handled given the current privilege level. *exception_handler* processes traps and privilege transition instructions, and delegates trap handling to *trap_handler*. *handle_interrupt*, *handle_mem_exception* and *handle_illegal* are convenient single-purpose helper wrapper functions.

```
function exception_delegatee(e : ExceptionType, p : Privilege) -> Privilege = {
  let idx = num_of_ExceptionType(e);
  let super = bit_to_bool(medeleg.bits()[idx]);
  /* if S-mode is absent, medeleg delegates to U-mode if 'N' is supported. */
  let user = if haveSupMode()
             then super & haveNExt() & bit_to_bool(sedeleg.bits()[idx])
             else super & haveNExt();
  let deleg = if haveUsrMode() & user then User
              else if haveSupMode() & super then Supervisor
              else Machine;
  /* We cannot transition to a less-privileged mode. */
  if privLevel_to_bits(deleg) <_u privLevel_to_bits(p)
  then p else deleg
}
```

```
function trap_handler(del_priv : Privilege, intr : bool, c : exc_code, pc : xlenbits
    ↪ , info : option(xlenbits), ext : option(ext_exception))
               -> xlenbits = {
  rvfi_trap();
```

```
if get_config_print_platform()
then print_platform("handling " ^ (if intr then "int" else "exc")
                ^ BitStr(c) ^ " at priv " ^ to_str(del_priv)
                ^ " with tval " ^ BitStr(tval(info)));

cancel_reservation();

match (del_priv) {
  Machine => {
    mcause->IsInterrupt() = bool_to_bits(intr);
    mcause->Cause() = EXTZ(c);

    mstatus->MPIE() = mstatus.MIE();
    mstatus->MIE() = 0b0;
    mstatus->MPP() = privLevel_to_bits(cur_privilege);
    mtval = tval(info);
    mepc = pc;

    cur_privilege = del_priv;

    handle_trap_extension(del_priv, pc, ext);

    if get_config_print_reg()
    then print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()));

    prepare_trap_vector(del_priv, mcause)
  },
  Supervisor => {
    assert (haveSupMode(), "no supervisor mode present for delegation");

    scause->IsInterrupt() = bool_to_bits(intr);
    scause->Cause() = EXTZ(c);

    mstatus->SPIE() = mstatus.SIE();
    mstatus->SIE() = 0b0;
    mstatus->SPP() = match cur_privilege {
                    User => 0b0,
                    Supervisor => 0b1,
                    Machine => internal_error("invalid privilege for s-mode trap")
                  };
    stval = tval(info);
    sepc = pc;

    cur_privilege = del_priv;

    handle_trap_extension(del_priv, pc, ext);

    if get_config_print_reg()
    then print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()));

    prepare_trap_vector(del_priv, scause)
  },
  User => {
    assert(haveUsrMode(), "no user mode present for delegation");

    ucause->IsInterrupt() = bool_to_bits(intr);
    ucause->Cause() = EXTZ(c);

    mstatus->UPIE() = mstatus.UIE();
    mstatus->UIE() = 0b0;
```

```
      utval = tval(info);
      uepc = pc;

      cur_privilege = del_priv;

      handle_trap_extension(del_priv, pc, ext);

      if get_config_print_reg()
      then print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()));

      prepare_trap_vector(del_priv, ucause)
    }
  };
}
```

```
function exception_handler(cur_priv : Privilege, ctl : ctl_result,
                           pc: xlenbits) -> xlenbits = {
  match (cur_priv, ctl) {
    (_, CTL_TRAP(e)) => {
      let del_priv = exception_delegatee(e.trap, cur_priv);
      if get_config_print_platform()
      then print_platform("trapping from " ^ to_str(cur_priv) ^ " to " ^ to_str(
          ↪ del_priv)
                          ^ " to handle " ^ to_str(e.trap));
      trap_handler(del_priv, false, exceptionType_to_bits(e.trap), pc, e.excinfo, e.
          ↪ ext)
    },
    (_, CTL_MRET()) => {
      let prev_priv = cur_privilege;
      mstatus->MIE() = mstatus.MPIE();
      mstatus->MPIE() = 0b1;
      cur_privilege = privLevel_of_bits(mstatus.MPP());
      mstatus->MPP() = privLevel_to_bits(if haveUsrMode() then User else Machine);
      if cur_privilege != Machine
      then mstatus->MPRV() = 0b0;

      if get_config_print_reg()
      then print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()));
      if get_config_print_platform()
      then print_platform("ret-ing from " ^ to_str(prev_priv) ^ " to " ^ to_str(
          ↪ cur_privilege));

      cancel_reservation();
      prepare_xret_target(Machine) & pc_alignment_mask()
    },
    (_, CTL_SRET()) => {
      let prev_priv = cur_privilege;
      mstatus->SIE() = mstatus.SPIE();
      mstatus->SPIE() = 0b1;
      cur_privilege = if mstatus.SPP() == 0b1 then Supervisor else User;
      mstatus->SPP() = 0b0;
      if cur_privilege != Machine
      then mstatus->MPRV() = 0b0;

      if get_config_print_reg()
      then print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()));
      if get_config_print_platform()
      then print_platform("ret-ing from " ^ to_str(prev_priv)
                          ^ " to " ^ to_str(cur_privilege));

      cancel_reservation();
      prepare_xret_target(Supervisor) & pc_alignment_mask()
    },
    (_, CTL_URET()) => {
      let prev_priv = cur_privilege;
      mstatus->UIE() = mstatus.UPIE();
      mstatus->UPIE() = 0b1;
      cur_privilege = User;

      if get_config_print_reg()
      then print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()));
      if get_config_print_platform()
      then print_platform("ret-ing from " ^ to_str(prev_priv) ^ " to " ^ to_str(
          ↪ cur_privilege));

      cancel_reservation();
      prepare_xret_target(User) & pc_alignment_mask()
    }
  }
}
```

```
function handle_mem_exception(addr : xlenbits, e : ExceptionType) -> unit = {
 let t : sync_exception = struct { trap = e,
                                   excinfo = Some(addr),
                                   ext = None() } in
 set_next_pc(exception_handler(cur_privilege, CTL_TRAP(t), PC))
}
```

```
function handle_interrupt(i : InterruptType, del_priv : Privilege) -> unit =
 set_next_pc(trap_handler(del_priv, true, interruptType_to_bits(i), PC, None(),
    ↪ None()))
```

```
function handle_illegal() -> unit = {
 let info = if plat_mtval_has_illegal_inst_bits ()
            then Some(instbits)
            else None();
 let t : sync_exception = struct { trap = E_Illegal_Instr(),
                                   excinfo = info,
                                   ext = None() };
 set_next_pc(exception_handler(cur_privilege, CTL_TRAP(t), PC))
}
```

A.6 Virtual Memory

This section covers the support of virtual memory in supervisor mode. We specify only the Sv39 address translation scheme below, since the Sv32 and Sv48 schemes follow a very similar structure.

A.6.1 PTE handling

These types and functions describe the specification and handling of page-table entries, used in the specification of address translation described next.

```
type pteAttribs = bits(8)
```

```
struct PTE_Bits = {PTE_Bits_chunk_0 : bitvector(8, dec)}
```

```
function isPTEPtr(p : pteAttribs, ext : extPte) -> bool = {
  let a = Mk_PTE_Bits(p);
  a.R() == 0b0 & a.W() == 0b0 & a.X() == 0b0
}
```

```
function isInvalidPTE(p : pteAttribs, ext : extPte) -> bool = {
  let a = Mk_PTE_Bits(p);
  a.V() == 0b0 | (a.W() == 0b1 & a.R() == 0b0)
}
```

```
function checkPTEPermission(ac : AccessType(ext_access_type), priv : Privilege, mxr
    ↪ : bool, do_sum : bool, p : PTE_Bits, ext : extPte, ext_ptw : ext_ptw) ->
    ↪ PTE_Check = {
  match (ac, priv) {
    (Read(Data), User) => to_pte_check(p.U() == 0b1 & (p.R() == 0b1 | (p.X() == 0b1 &
        ↪  mxr))),
    (Write(Data), User) => to_pte_check(p.U() == 0b1 & p.W() == 0b1),
    (ReadWrite(Data), User) => to_pte_check(p.U() == 0b1 & p.W() == 0b1 & (p.R() == 0
        ↪ b1 | (p.X() == 0b1 & mxr))),
    (Execute(), User) => to_pte_check(p.U() == 0b1 & p.X() == 0b1),

    (Read(Data), Supervisor) => to_pte_check((p.U() == 0b0 | do_sum) & (p.R() == 0b1
        ↪ | (p.X() == 0b1 & mxr))),
    (Write(Data), Supervisor) => to_pte_check((p.U() == 0b0 | do_sum) & p.W() == 0b1)
        ↪ ,
    (ReadWrite(Data), Supervisor) => to_pte_check((p.U() == 0b0 | do_sum) & p.W() ==
        ↪ 0b1 & (p.R() == 0b1 | (p.X() == 0b1 & mxr))),
    (Execute(), Supervisor) => to_pte_check(p.U() == 0b0 & p.X() == 0b1),

    (_, Machine) => internal_error("m-mode mem perm check")
  }
}
```

```
function update_PTE_Bits(p : PTE_Bits, a : AccessType(ext_access_type), ext : extPte
    ↪ ) -> option((PTE_Bits, extPte)) = {
  let update_d = (a == Write(Data) | a == ReadWrite(Data)) & p.D() == 0b0; // dirty-
      ↪ bit
  let update_a = p.A() == 0b0; // accessed-bit
  if update_d | update_a then {
    let np = update_A(p, 0b1);
    let np = if update_d then update_D(np, 0b1) else np;
    Some(np, ext)
  } else None()
}
```

A.6.2 Sv39 Address Translation

The specification of address translation below occasionally refer to a simplistic TLB implemented in the formal model, but we do not include the definitions for this TLB since it is not actually part of the ISA specification. The model uses *PTW_Result* to capture the result of the page-table walk implemented in *walk*39, and converts any resulting error into an architectural exception using *translationException*. *translateAddr*is the entry point for address translation, and is called by the instructions that access memory as described in the user-level specification. *translateAddr* uses *translationMode* to dispatch to the appropriate translation mode.

plat_enable_dirty_update is an external platform-level setting that controls whether PTEs are updated during page-table walks.

```
union PTW_Error = {
  PTW_Access : unit, /* physical memory access error for a PTE */
  PTW_Invalid_PTE : unit,
  PTW_No_Permission : unit,
  PTW_Misaligned : unit, /* misaligned superpage */
  PTW_PTE_Update : unit, /* PTE update needed but not enabled */
  PTW_Ext_Error : ext_ptw_error /* parameterized for errors from extensions */
}
```

```
union PTW_Result('paddr : Type, 'pte : Type) = {
  PTW_Success: ('paddr, 'pte, 'paddr, nat, bool, ext_ptw),
  PTW_Failure: (PTW_Error, ext_ptw)
}
```

```
function translationException(a : AccessType(ext_access_type), f : PTW_Error) ->
    ↪ ExceptionType = {
  let e : ExceptionType =
  match (a, f) {
    (_, PTW_Ext_Error(e)) => E_Extension(ext_translate_exception(e)),
    (ReadWrite(Data), PTW_Access()) => E_SAMO_Access_Fault(),
    (ReadWrite(Data), _) => E_SAMO_Page_Fault(),
    (Read(Data), PTW_Access()) => E_Load_Access_Fault(),
    (Read(Data), _) => E_Load_Page_Fault(),
    (Write(Data), PTW_Access()) => E_SAMO_Access_Fault(),
    (Write(Data), _) => E_SAMO_Page_Fault(),
    (Execute(), PTW_Access()) => E_Fetch_Access_Fault(),
    (Execute(), _) => E_Fetch_Page_Fault()
  } in {
/* print_mem("translationException(" ^ a ^ ", " ^ f ^ ") -> " ^ e); */
    e
  }
}
```

```
function walk39(vaddr, ac, priv, mxr, do_sum, ptb, level, global, ext_ptw) = {
  let va = Mk_SV39_Vaddr(vaddr);
  let pt_ofs : paddr64 = shiftl(EXTZ(shiftr(va.VPNi(), (level * SV39_LEVEL_BITS))[(
      ↪ SV39_LEVEL_BITS - 1) .. 0]),
                             PTE39_LOG_SIZE);
  let pte_addr = ptb + pt_ofs;
  match (mem_read(ac, EXTZ(pte_addr), 8, false, false, false)) {
    MemException(_) => {
/* print("walk39(vaddr=" ^ BitStr(vaddr) ^ " level=" ^ string_of_int(level)
            ^ " pt_base=" ^ BitStr(ptb)
            ^ " pt_ofs=" ^ BitStr(pt_ofs)
            ^ " pte_addr=" ^ BitStr(pte_addr)
            ^ ": invalid pte address"); */
      PTW_Failure(PTW_Access(), ext_ptw)
    },
    MemValue(v) => {
```

```
    let pte = Mk_SV39_PTE(v);
    let pbits = pte.BITS();
    let ext_pte = pte.Ext();
    let pattr = Mk_PTE_Bits(pbits);
    let is_global = global | (pattr.G() == 0b1);
/* print("walk39(vaddr=" ^ BitStr(vaddr) ^ " level=" ^ string_of_int(level)
          ^ " pt_base=" ^ BitStr(ptb)
          ^ " pt_ofs=" ^ BitStr(pt_ofs)
          ^ " pte_addr=" ^ BitStr(pte_addr)
          ^ " pte=" ^ BitStr(v)); */
    if isInvalidPTE(pbits, ext_pte) then {
/* print("walk39: invalid pte"); */
      PTW_Failure(PTW_Invalid_PTE(), ext_ptw)
    } else {
      if isPTEPtr(pbits, ext_pte) then {
        if level > 0 then {
          /* walk down the pointer to the next level */
          walk39(vaddr, ac, priv, mxr, do_sum, shiftl(EXTZ(pte.PPNi()), PAGESIZE_BITS), level - 1, is_global, ext_ptw)
        } else {
          /* last-level PTE contains a pointer instead of a leaf */
/* print("walk39: last-level pte contains a ptr"); */
          PTW_Failure(PTW_Invalid_PTE(), ext_ptw)
        }
      } else { /* leaf PTE */
        match checkPTEPermission(ac, priv, mxr, do_sum, pattr, ext_pte, ext_ptw) {
          PTE_Check_Failure(ext_ptw) => {
/* print("walk39: pte permission check failure"); */
            PTW_Failure(PTW_No_Permission(), ext_ptw)
          },
          PTE_Check_Success(ext_ptw) => {
            if level > 0 then { /* superpage */
              /* fixme hack: to get a mask of appropriate size */
              let mask = shiftl(pte.PPNi() ^ pte.PPNi() ^ EXTZ(0b1), level * SV39_LEVEL_BITS) - 1;
              if (pte.PPNi() & mask) != EXTZ(0b0) then {
                /* misaligned superpage mapping */
/* print("walk39: misaligned superpage mapping"); */
                PTW_Failure(PTW_Misaligned(), ext_ptw)
              } else {
                /* add the appropriate bits of the VPN to the superpage PPN */
                let ppn = pte.PPNi() | (EXTZ(va.VPNi()) & mask);
/* let res = append(ppn, va.PgOfs());
                print("walk39: using superpage: pte.ppn=" ^ BitStr(pte.PPNi())
                      ^ " ppn=" ^ BitStr(ppn) ^ " res=" ^ BitStr(res)); */
                PTW_Success(append(ppn, va.PgOfs()), pte, pte_addr, level, is_global, ext_ptw)
              }
            } else {
              /* normal leaf PTE */
/* let res = append(pte.PPNi(), va.PgOfs());
              print("walk39: pte.ppn=" ^ BitStr(pte.PPNi()) ^ " ppn=" ^ BitStr(pte.PPNi()) ^ " res=" ^ BitStr(res)); */
              PTW_Success(append(pte.PPNi(), va.PgOfs()), pte, pte_addr, level, is_global, ext_ptw)
            }
          }
        }
      }
    }
    }
  }
}
```

```
function translate39(asid, ptb, vAddr, ac, priv, mxr, do_sum, level, ext_ptw) = {
  match lookup_TLB39(asid, vAddr) {
    Some(idx, ent) => {
/* print("translate39: TLB39 hit for " ^ BitStr(vAddr)); */
      let pte = Mk_SV39_PTE(ent.pte);
      let ext_pte = pte.Ext();
      let pteBits = Mk_PTE_Bits(pte.BITS());
      match checkPTEPermission(ac, priv, mxr, do_sum, pteBits, ext_pte, ext_ptw) {
        PTE_Check_Failure(ext_ptw) => { TR_Failure(PTW_No_Permission(), ext_ptw) },
        PTE_Check_Success(ext_ptw) => {
          match update_PTE_Bits(pteBits, ac, ext_pte) {
            None() => TR_Address(ent.pAddr | EXTZ(vAddr & ent.vAddrMask), ext_ptw),
            Some(pbits, ext) => {
              if ~ (plat_enable_dirty_update ())
              then {
                /* pte needs dirty/accessed update but that is not enabled */
                TR_Failure(PTW_PTE_Update(), ext_ptw)
              } else {
                /* update PTE entry and TLB */
                n_pte = update_BITS(pte, pbits.bits());
                n_pte = update_Ext(n_pte, ext);
                n_ent : TLB39_Entry = ent;
                n_ent.pte = n_pte.bits();
                write_TLB39(idx, n_ent);
                /* update page table */
                match mem_write_value(EXTZ(ent.pteAddr), 8, n_pte.bits(), false, false,
                    ↪ false) {
                  MemValue(_) => (),
                  MemException(e) => internal_error("invalid physical address in TLB")
                };
                TR_Address(ent.pAddr | EXTZ(vAddr & ent.vAddrMask), ext_ptw)
              }
            }
          }
        }
      }
    },
    None() => {
      match walk39(vAddr, ac, priv, mxr, do_sum, ptb, level, false, ext_ptw) {
        PTW_Failure(f, ext_ptw) => TR_Failure(f, ext_ptw),
        PTW_Success(pAddr, pte, pteAddr, level, global, ext_ptw) => {
          match update_PTE_Bits(Mk_PTE_Bits(pte.BITS()), ac, pte.Ext()) {
            None() => {
              add_to_TLB39(asid, vAddr, pAddr, pte, pteAddr, level, global);
              TR_Address(pAddr, ext_ptw)
            },
            Some(pbits, ext) =>
              if ~ (plat_enable_dirty_update ())
              then {
                /* pte needs dirty/accessed update but that is not enabled */
                TR_Failure(PTW_PTE_Update(), ext_ptw)
              } else {
                w_pte : SV39_PTE = update_BITS(pte, pbits.bits());
                w_pte : SV39_PTE = update_Ext(w_pte, ext);
                match mem_write_value(EXTZ(pteAddr), 8, w_pte.bits(), false, false,
                    ↪ false) {
```

```
              MemValue(_) => {
                add_to_TLB39(asid, vAddr, pAddr, w_pte, pteAddr, level, global);
                TR_Address(pAddr, ext_ptw)
              },
              MemException(e) => {
                /* pte is not in valid memory */
                TR_Failure(PTW_Access(), ext_ptw)
              }
            }
          }
        }
      }
    }
  }
 }
}
```

```
function translationMode(priv) = {
  if priv == Machine then Sbare
  else {
    let arch = architecture(get_mstatus_SXL(mstatus));
    match arch {
      Some(RV64) => {
        let mbits : satp_mode = Mk_Satp64(satp).Mode();
        match satp64Mode_of_bits(RV64, mbits) {
          Some(m) => m,
          None() => internal_error("invalid RV64 translation mode in satp")
        }
      },
      Some(RV32) => {
        let s = Mk_Satp32(satp[31..0]);
        if s.Mode() == 0b0 then Sbare else Sv32
      },
      _ => internal_error("unsupported address translation arch")
    }
  }
}
```

```
union TR_Result('paddr : Type, 'failure : Type) = {
  TR_Address : ('paddr, ext_ptw),
  TR_Failure : ('failure, ext_ptw)
}
```

```
function translateAddr(vAddr, ac) = {
  let effPriv : Privilege = effectivePrivilege(ac, mstatus, cur_privilege);
  let mxr : bool = mstatus.MXR() == 0b1;
  let do_sum : bool = mstatus.SUM() == 0b1;
  let mode : SATPMode = translationMode(effPriv);

  let asid = curAsid64(satp);
  let ptb = curPTB64(satp);

  /* PTW extensions: initialize the PTW extension state. */
  let ext_ptw : ext_ptw = init_ext_ptw;

  match mode {
    Sbare => TR_Address(vAddr, ext_ptw),
    Sv39 => match translate39(asid, ptb, vAddr[38 .. 0], ac, effPriv, mxr, do_sum,
       ↪ SV39_LEVELS - 1, ext_ptw) {
            TR_Address(pa, ext_ptw) => TR_Address(EXTZ(pa), ext_ptw),
              TR_Failure(f, ext_ptw) => TR_Failure(translationException(ac, f),
                  ↪ ext_ptw)
            },
    Sv48 => match translate48(asid, ptb, vAddr[47 .. 0], ac, effPriv, mxr, do_sum,
        ↪ SV48_LEVELS - 1, ext_ptw) {
              TR_Address(pa, ext_ptw) => TR_Address(EXTZ(pa), ext_ptw),
              TR_Failure(f, ext_ptw) => TR_Failure(translationException(ac, f),
                  ↪ ext_ptw)
            },
    _ => internal_error("unsupported address translation scheme")
  }
}
```

Bibliography

[1] Sail ISA Description Language. https://www.cl.cam.ac.uk/~pes20/sail/.

[2] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[3] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, December 2002.