The RISC-V Instruction Set Manual

Volume II: Privileged Architecture

Document Version 20190125-Public-Review-*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

January 22, 2019

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, Paolo Bonzini, Ruslan Bukin, Christopher Celio, Chuanhua Chang, David Chisnall, Anthony Coulter, Palmer Dabbelt, Monte Dalrymple, Dennis Ferguson, Mike Frysinger, John Hauser, David Horner, Olof Johansson, Yunsup Lee, Andrew Lutomirski, Prashanth Mundkur, Jonathan Neuschäfer, Rishiyur Nikhil, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Dmitri Pavlov, Kade Phillips, 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 20190125-Public-Review-*draft*", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, January 2019.

Preface

This document describes the RISC-V privileged architecture. This release, version 20190125-Public-Review-*draft*, will be used in ratifying the machine and supervisor ISA modules.

The document contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	**1.11**	**Ratification**
Supervisor ISA	**1.11**	**Ratification**
Hypervisor ISA	*0.2*	*Draft*

The changes in this version of the document include:

- Improvements to the description and commentary.
- Added a draft proposal for a hypervisor extension.
- Specified which interrupt sources are reserved for standard use.
- 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.
- 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.
- 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.

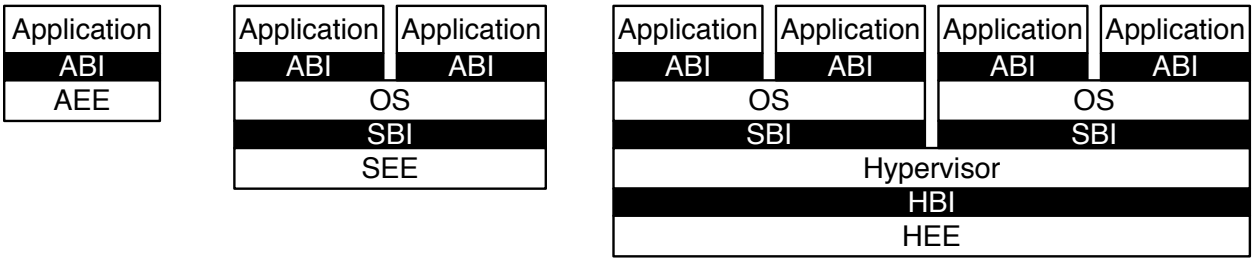


Figure 1.1: Different implementation stacks supporting various forms of privileged execution.

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 exception to be raised. These exceptions will normally cause traps into an underlying execution environment.

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.

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 RV64IMAC dialect, the user, supervisor and machine privilege levels, and the Sv39 address translation mode. It passes the `riscv-tests` test suite, and is capable of booting Linux and seL4. The model currently does not model event and performance counters, address translation other than Sv39, PMP/PMA, the N Standard Extension for User-Level Interrupts, 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 91, 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.

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.

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.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:6]		
User CSRs				
00	00	XX	0x000-0x0FF	Standard read/write
01	00	XX	0x400-0x4FF	Standard read/write
10	00	XX	0x800-0x8FF	Custom read/write
11	00	00-10	0xC00-0xCBF	Standard read-only
11	00	11	0xCC0-0xCFF	Custom read-only
Supervisor CSRs				
00	01	XX	0x100-0x1FF	Standard read/write
01	01	00-10	0x500-0x5BF	Standard read/write
01	01	11	0x5C0-0x5FF	Custom read/write
10	01	00-10	0x900-0x9BF	Standard read/write
10	01	11	0x9C0-0x9FF	Custom read/write
11	01	00-10	0xD00-0xDBF	Standard read-only
11	01	11	0xDC0-0xDFF	Custom read-only
Reserved CSRs				
XX	10	XX	Reserved	
Machine CSRs				
00	11	XX	0x300-0x3FF	Standard read/write
01	11	00-10	0x700-0x79F	Standard read/write
01	11	10	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	10	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11	0x7C0-0x7FF	Custom read/write
10	11	00-10	0xB00-0xBBF	Standard read/write
10	11	11	0xBC0-0xBFF	Custom read/write
11	11	00-10	0xF00-0xFBF	Standard read-only
11	11	11	0xFC0-0xFFF	Custom read-only

Table 2.1: Allocation of RISC-V CSR address ranges.

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 2.2–2.5 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.

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, RV32I only.
0xC81	URO	timeh	Upper 32 bits of time, RV32I only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32I only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32I only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32I only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32I 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
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.
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.
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.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

Table 2.4: 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, RV32I only.
0xB82	MRW	minstreth	Upper 32 bits of minstret, RV32I only.
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3, RV32I only.
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4, RV32I only.
		⋮	
0xB9F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter31, RV32I 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	dscratch	Debug scratch register.

Table 2.5: 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 value of the field prior to the write.

2.4 CSR Width Modulation

If the width of a CSR is changed (for example, by changing MXLEN or UXLEN, as described in Section 3.1.8), 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 1.11

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.

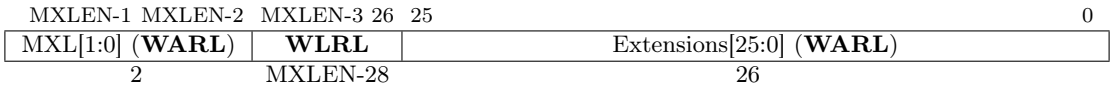


Figure 3.1: Machine ISA register (`misa`).

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.

The `misa` CSR is MXLEN bits wide. If the value read from `misa` is nonzero, field MXL of that value

MXL	XLEN
1	32
2	64
3	128

Table 3.1: Encoding of MXL field in `misa`

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 should contain the maximal set of supported extensions, and I should 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 "G" bit is used as an escape to allow expansion to a larger space of standard extension names.

G is used to indicate the combination IMAFD, so is redundant in the `misa` *CSR, hence we reserve the bit to indicate that additional standard extensions are present.*

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.

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	Additional standard extensions present
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.

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

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

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

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.

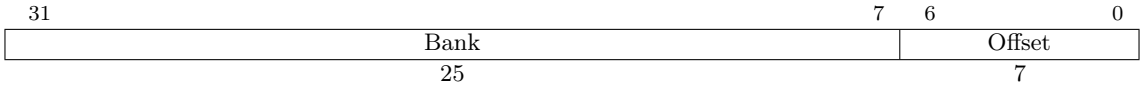


Figure 3.2: Vendor ID register (`mvendorid`).

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`.

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.

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 be 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.

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 Register (`mstatus`)

The `mstatus` register is an MXLEN-bit read/write register formatted as shown in Figure 3.6 for RV32 and Figure 3.7 for RV64. The `mstatus` register keeps track of and controls the hart's current operating state. Restricted views of the `mstatus` register appear as the `sstatus` and `ustatus` registers in the S-level and U-level ISAs respectively.

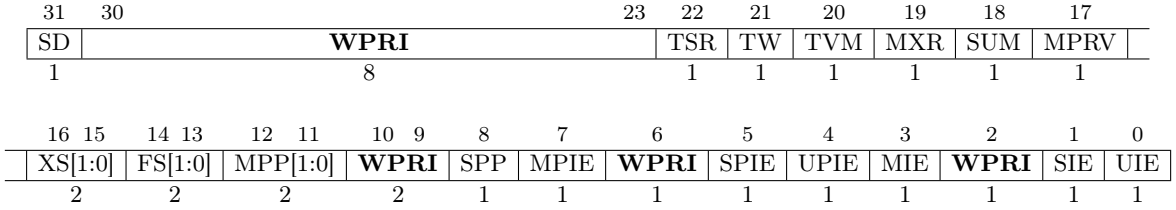


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

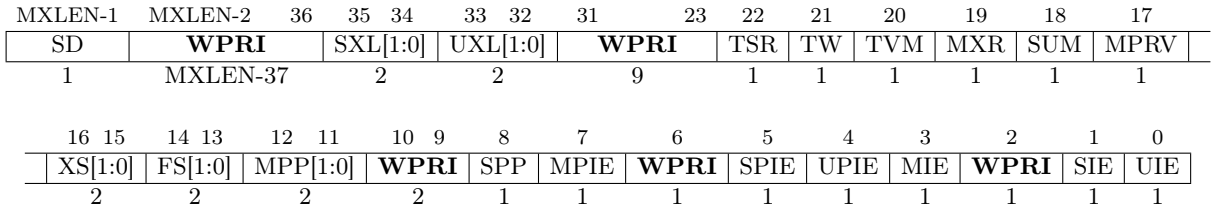


Figure 3.7: Machine-mode status register (`mstatus`) for RV64.

3.1.7 Privilege and Global Interrupt-Enable Stack in `mstatus` register

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

The global x*IE 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 the lower-privilege mode's global wIE bit. Interrupts for higher-privilege modes, $y>x$, are always globally enabled regardless of the setting of the higher-privilege mode's global yIE bit. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected high-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 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, SPP is one bit wide, and UPP is implicitly zero. 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 MRET, SRET, or URET instructions are used to return from traps in M-mode, S-mode, or U-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).

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.

User-level interrupts are an optional extension and have been allocated the ISA extension letter N. If user-level interrupts are omitted, the UIE and UPIE bits are hardwired to zero. For all other supported privilege modes x, the xIE and xPIE must not be hardwired.

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.

3.1.8 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.

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 an error, 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.

3.1.9 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 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.

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 4.15) 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.

For simplicity, MPRV and MXR are in effect regardless of privilege mode, but in normal use will only be enabled for short sequences in machine mode.

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 3.1.8.

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 4.15) 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.

3.1.10 Virtualization Support in `mstatus` Register

The TVM (Trap Virtual Memory) bit 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 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 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.

3.1.11 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] read/write 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 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.

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.

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.

In a supervisor-level OS, any additional user-mode state should be initialized, saved, and restored using SBI calls that treats the additional context as an opaque object of a fixed maximum size. The implementation of the SBI initialize, save, and restore calls might require additional implementation-dependent privileged instructions to initialize, save, and restore microarchitectural state inside a coprocessor.

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.

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

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

Figure 3.8: Machine trap-vector base-address register (`mtvec`).

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 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) 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 a less-privileged 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.

3.1.13 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 all three privilege modes (M/S/U), setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap in S-mode or U-mode to the S-mode trap handler. If U-mode traps are supported, S-mode may in turn set corresponding bits in the `sedeleg` and `sideleg` registers to delegate traps that occur in U-mode to the U-mode trap handler.

In systems with two privilege modes (M/U) and support for U-mode traps, setting a bit in `medeleg` or `mideleg` will delegate the corresponding trap in U-mode to the U-mode trap handler.

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 should not exist.

In versions 1.9.1 and earlier , these registers existed but were hardwired to zero in M-mode only, or M/U without N 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 a less-privileged mode x, the x`cause` register is written with the trap cause; the x`epc` register is written with the virtual address of the instruction that took the trap; the x`tval` register is written with an exception-specific datum; the xPP field of `mstatus` is written with the active privilege mode at the time of the trap; the xPIE field of `mstatus` is written with the value of the xIE field at the time of the trap; and the xIE field of `mstatus` is cleared. The `mcause` and `mepc` registers and the MPP and MPIE fields of `mstatus` are not written.

An implementation shall not hardwire any delegation bits to one, i.e., any trap that can be delegated must support not being delegated. 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.

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.

`medeleg` has a bit position allocated for every synchronous exception shown in Table 3.6, 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 3.9: Machine Exception Delegation Register `medeleg`.

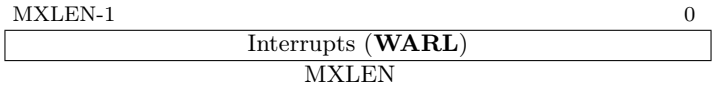


Figure 3.10: Machine Interrupt Delegation Register `mideleg`.

`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).

Some exceptions cannot occur at less privileged modes, and corresponding x`edeleg` bits should be hardwired to zero. In particular, `medeleg`[11] and `sedeleg`[11:9] are all hardwired to zero.

3.1.14 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. Only the bits corresponding to lower-privilege software interrupts (USIP, SSIP), timer interrupts (UTIP, STIP), and external interrupts (UEIP, SEIP) in `mip` are writable through this CSR address; the remaining bits are read-only.

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.

Restricted views of the `mip` and `mie` registers appear as the `sip`/`sie`, and `uip`/`uie` registers in S-mode and U-mode respectively. If an interrupt is delegated to privilege mode x by setting a bit in the `mideleg` register, it becomes visible in the x`ip` register and is maskable using the x`ie` register. Otherwise, the corresponding bits in x`ip` and x`ie` appear to be hardwired to zero.

MXLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI		MEIP	**WPRI**	SEIP	UEIP	MTIP	**WPRI**	STIP	UTIP	MSIP	**WPRI**	SSIP	USIP
MXLEN-12		1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.11: Machine interrupt-pending register (`mip`).

MXLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI		MEIE	**WPRI**	SEIE	UEIE	MTIE	**WPRI**	STIE	UTIE	MSIE	**WPRI**	SSIE	USIE
MXLEN-12		1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.12: Machine interrupt-enable register (`mie`).

The MTIP, STIP, UTIP bits correspond to timer interrupt-pending bits for machine, supervisor, and user timer interrupts, respectively. The MTIP bit is read-only and is cleared by writing to the

memory-mapped machine-mode timer compare register. The UTIP and STIP bits may be written by M-mode software to deliver timer interrupts to lower privilege levels. User and supervisor software may clear the UTIP and STIP bits with calls to the AEE and SEE respectively.

There is a separate timer interrupt-enable bit, named MTIE, STIE, and UTIE for M-mode, S-mode, and U-mode timer interrupts respectively.

Each lower privilege level has a separate software interrupt-pending bit (SSIP, USIP), which can be both read and written by CSR accesses from code running on the local hart at the associated or any higher privilege level. The machine-level MSIP bits are written by accesses to memory-mapped control registers, which are used by remote harts to provide machine-mode interprocessor interrupts. Interprocessor interrupts for lower privilege levels are implemented through ABI and SBI calls to the AEE or SEE respectively, which might ultimately result in a machine-mode write to the receiving hart's MSIP bit. A hart can write its own MSIP bit using the same memory-mapped control register.

The MSIE, SSIE, and USIE fields in the `mie` CSR enable M-mode software interrupts, S-mode software interrupts, and U-mode software interrupts, respectively.

We only allow a hart to directly write its own SSIP or USIP bits when running in the appropriate mode, as other harts might be virtualized and possibly descheduled by higher privilege levels. We rely on ABI and SBI calls 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.

The MEIP field in `mip` is a read-only bit that indicates a machine-mode external interrupt is pending. MEIP is set and cleared by a platform-specific interrupt controller. The MEIE field in `mie` enables machine external interrupts when set.

The SEIP field in `mip` contains a single read-write bit. SEIP 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. The logical-OR of the software-writable bit and the signal from the external interrupt controller is used to generate external interrupts to the supervisor. When the SEIP bit is read with a CSRRW, CSRRS, or CSRRC instruction, the value returned in the `rd` destination register contains 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 is 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.

The UEIP field in `mip` provides user-mode external interrupts when the N extension for user-mode interrupts is implemented. It is defined analogously to SEIP.

The MEIE, SEIE, and UEIE fields in the `mie` CSR enable M-mode external interrupts, S-mode external interrupts, and U-mode external interrupts, respectively.

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

For all the various interrupt types (software, timer, and external), if a privilege level is not supported, or if U-mode is supported but the N extension is not supported, than the associated pending and interrupt-enable bits are hardwired to zero in the `mip` and `mie` registers respectively. Hence, these are all effectively **WARL** fields.

Implementations may add additional platform-specific interrupt sources to bits 16 and above of the `mip` and `mie` registers. Some platforms may avail these interrupts for custom use. The other unallocated interrupt sources (15–12, 10, 6, and 2) are reserved for future standard use.

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 (S or U) and that mode's interrupt enable bit (SIE or UIE in `mstatus`) is set, or if the current privilege mode is less than the delegated privilege mode.

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, UEI, USI, UTI. 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.

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

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode register, `mtime`. `mtime` must run at constant frequency, and the platform must provide a mechanism for determining the timebase of `mtime`.

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`), which causes a timer interrupt to be posted when the `mtime` register contains a value greater than or equal to the value in the

`mtimecmp` register. The interrupt remains posted until it is cleared by writing the `mtimecmp` register. The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

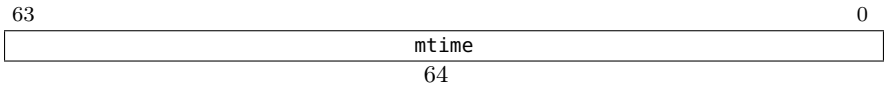


Figure 3.13: Machine time register (memory-mapped control register).

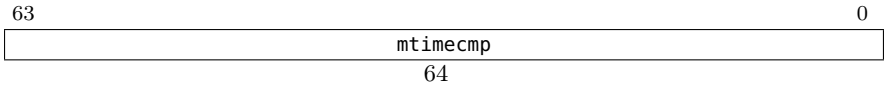


Figure 3.14: Machine time compare register (memory-mapped control register).

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:

3.1.16 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.

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

Figure 3.15: Sample code for setting the 64-bit time comparand in RV32 assuming the registers live in a strongly ordered I/O region.

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

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.

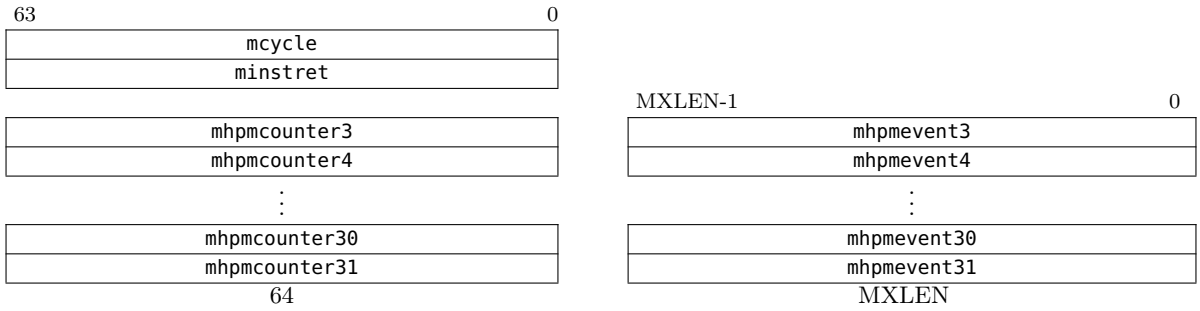


Figure 3.16: Hardware performance monitor counters.

All of these counters have 64-bit precision on RV32 and RV64.

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

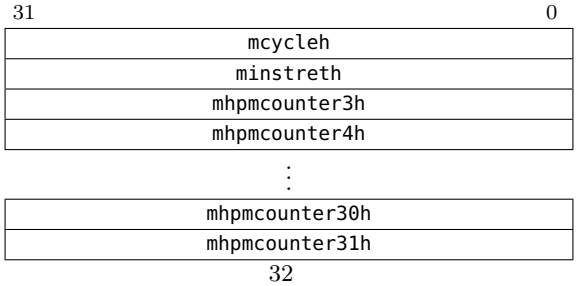


Figure 3.17: Upper 32 bits of hardware performance monitor counters, RV32 only.

3.1.17 Counter-Enable Registers (`[m|s]counteren`)

The counter-enable registers `mcounteren` and `scounteren` are 32-bit registers that control the availability of the hardware performance-monitoring counters to the next-lowest privileged mode.

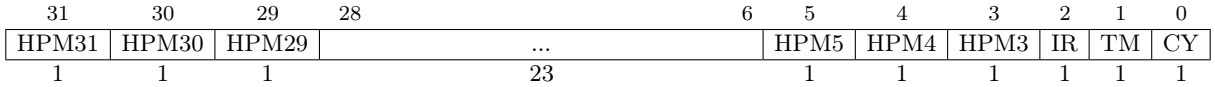


Figure 3.18: Counter-enable registers (`mcounteren` and `scounteren`).

The settings in these registers only control accessibility. The act of reading or writing these registers 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 `hpmcounter`*n* 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).

If S-mode is implemented, the same bit positions in the `scounteren` register analogously control access to these registers while executing in U-mode. If S-mode is permitted to access a counter register and the corresponding bit is set in `scounteren`, then U-mode is also permitted to access that register.

Registers `mcounteren` and `scounteren` are **WARL** registers that must be implemented if U-mode and S-mode are implemented, Any of the bits may contain a hardwired value of zero, indicating reads to the corresponding counter will cause an exception when executing in a less-privileged 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.

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

3.1.18 Machine Counter-Inhibit CSR (`mcountinhibit`)

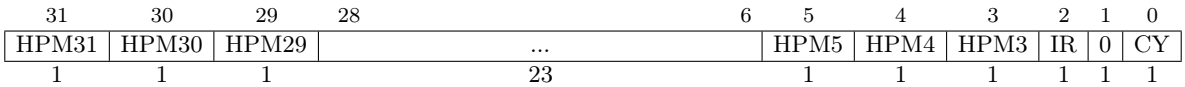


Figure 3.19: Counter-inhibit register `mcountinhibit`.

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 HPM*n* bit in the `mcountinhibit` register is clear, the `cycle`, `instret`, or `hpmcounter`*n* register increments as usual. When the CY, IR, or HPM*n* bit is set, the corresponding counter does not increment.

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.*

As all the harts on a processor core share a `cycle` *counter, so they share an* `mcountinhibit`*.CY bit.*

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

3.1.19 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 3.20: Machine-mode scratch register.

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

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

3.1.20 Machine Exception Program Counter (`mepc`)

`mepc` is an MXLEN-bit read/write register formatted as shown in Figure 3.21. The low bit of `mepc` (`mepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`mepc[1:0]`) are always zero.

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

`mepc` is a **WARL** register that must be able to hold all valid physical and 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 a trap is taken into M-mode, `mepc` is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, `mepc` is never written by the implementation, though it may be explicitly written by software.

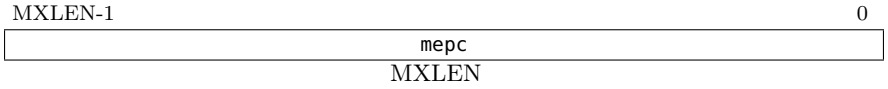


Figure 3.21: Machine exception program counter register.

3.1.21 Machine Cause Register (`mcause`)

The `mcause` register is an MXLEN-bit read-write register formatted as shown in Figure 3.22. When a trap is taken into M-mode, `mcause` is written with a code indicating the event that caused the trap. Otherwise, `mcause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `mcause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. 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.

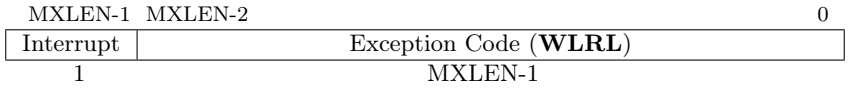


Figure 3.22: Machine Cause register `mcause`.

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.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	≥12	*Reserved*
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	*Reserved*

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

3.1.22 Machine Trap Value (`mtval`) Register

The `mtval` register is an MXLEN-bit read-write register formatted as shown in Figure 3.23. When a trap is taken into M-mode, `mtval` is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, `mtval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `mtval` informatively and which may unconditionally set it to zero.

When a hardware breakpoint is triggered, or an instruction-fetch, load, or store address-misaligned, access, or page-fault exception occurs, `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 exceptions. This design reduces datapath cost for most implementations, particularly those with hardware page-table walkers.*

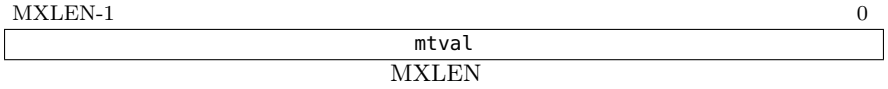


Figure 3.23: Machine Trap Value register.

For misaligned loads and stores that cause access or page-fault exceptions, `mtval` will contain the virtual address of the portion of the access that caused the fault. For instruction-fetch access 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

physical and 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.

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.

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.

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/URET	0	PRIV	0	SYSTEM

To return after handling a trap, there are separate trap return instructions per privilege level: MRET, SRET, and URET. 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 3.1.10. URET is only provided if user-mode traps are supported, and should raise an illegal instruction otherwise. 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 3.1.7, xRET sets the `pc` to the value stored in the x`epc` register.

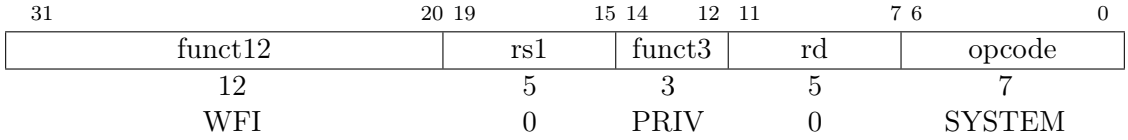
Previously, there was only a single ERET instruction (which was also earlier known as SRET). To support the addition of user-level interrupts, we needed to add a separate URET instruction to continue to allow classic virtualization of OS code using the ERET instruction. It then became more orthogonal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

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

If x*,RET 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 3.1.10.



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/SIE/UIE) and the delegation registers `[m|s]ideleg` (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.

3.3 Reset

Upon reset, a hart's privilege mode is set to M. The `mstatus` fields MIE and MPRV are reset to 0. 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. 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, but a value of 0 is reserved to mean "unknown cause" and implementations that do not distinguish sources of NMIs via the `mcause` register should return 0.

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 load, store, or instruction-fetch access 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, write, and execute of all access widths required by the attached devices.

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.

3.5.3 Atomicity PMAs

Atomicity PMAs describes which atomic instructions are supported in this address region. Main memory regions must support the atomic operations required by the processors attached. I/O regions may only support a subset or none of the processor-supported atomic operations.

Support for atomic instructions is divided into two categories: *LR/SC* and *AMOs*. 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.

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

Table 3.7: Classes of AMOs supported by I/O regions. Main memory regions must always support all AMOs required by the processor.

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

Memory regions that support LR/SC will define the conditions under which LR/SC sequences must succeed or must fail. Main memory must guarantee the eventual success of any LR/SC sequence that meets the requirements described in the user ISA specification.

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 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 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). Main memory regions always have either the RVWMO or RVTSO memory model.

Accesses by one hart to the I/O space are observable not only by other harts and bus mastering devices, but also by targeted slave I/O devices. Within I/O, regions may further be classified as implementing either *relaxed* or *strong* ordering. A relaxed I/O region has no ordering guarantees on how memory accesses made by one hart are observable by different harts or I/O devices beyond those enforced by FENCE and AMO instructions. A strongly ordered I/O region ensures that all accesses made by a hart to that region are only observable in program order by all other harts or I/O devices.

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 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 and within a platform may vary by physical memory region, 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 when the hart is running in S or U modes, and for loads and stores when the MPRV bit is set in the `mstatus` register and the MPP field in the `mstatus` register 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 without a system 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 16 PMP entries are supported. If any PMP entries are implemented, then all PMP CSRs must be implemented, but 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, four CSRs, `pmpcfg0`–`pmpcfg3`, hold the configurations `pmp0cfg`–`pmp15cfg` for the 16 PMP entries, as shown in Figure 3.24. For RV64, `pmpcfg0` and `pmpcfg2` hold the configurations for the 16 PMP entries, as shown in Figure 3.25; `pmpcfg1` and `pmpcfg3` 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.*

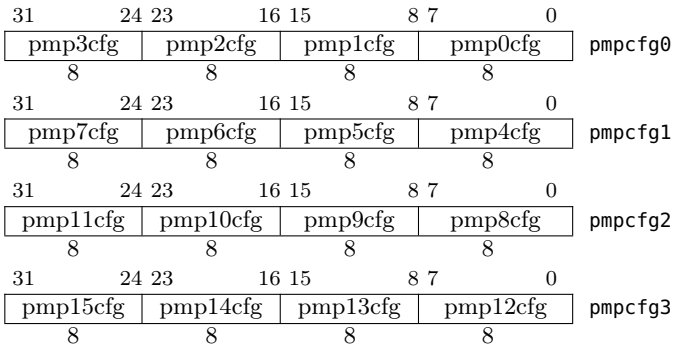


Figure 3.24: RV32 PMP configuration CSR layout.

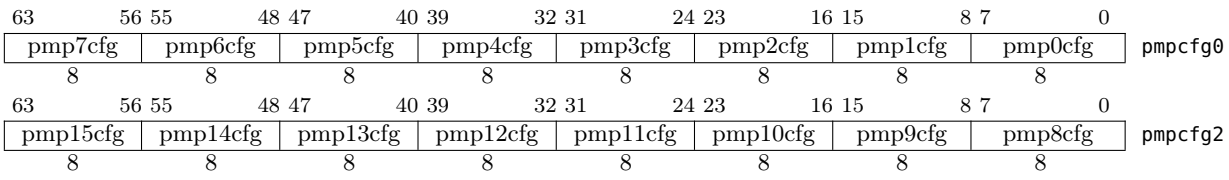


Figure 3.25: RV64 PMP configuration CSR layout.

The PMP address registers are CSRs named `pmpaddr0`–`pmpaddr15`. Each PMP address register encodes bits 33–2 of a 34-bit physical address for RV32, as shown in Figure 3.26. For RV64, each PMP address register encodes bits 55–2 of a 56-bit physical address, as shown in Figure 3.27. 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.

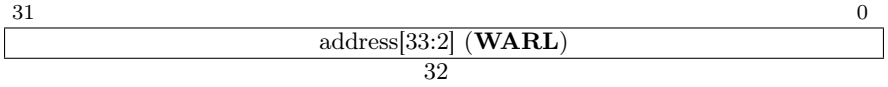


Figure 3.26: PMP address register format, RV32.

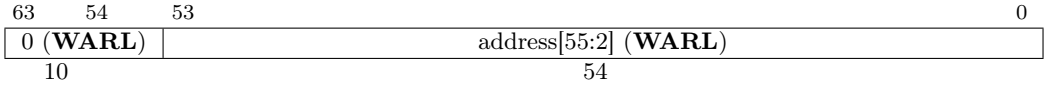


Figure 3.27: PMP address register format, RV64.

Figure 3.28 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.

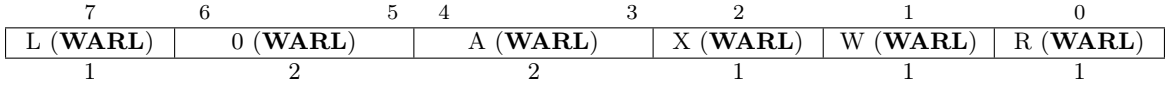


Figure 3.28: PMP configuration register format.

Attempting to fetch an instruction from a PMP region that does not have execute permissions raises a fetch access 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 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 exception.

If MXLEN is changed, the contents of the `pmpxcfg` fields are preserved, but appear in the `pmpcfgy` 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 3.8. 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, ≥8 bytes

Table 3.8: 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 3.9.

`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 3.9: 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 may only be unlocked with a system 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.

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.

Failed accesses generate a load, store, or instruction access exception. Note that a single instruction may generate multiple accesses, which may not be mutually atomic. An access 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 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 1.11

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-level code relies on a supervisor execution environment to initialize the environment and enter the supervisor code at an entry point defined by the supervisor binary interface (SBI). The SBI also defines function entry points that provide supervisor environment services for supervisor-level code.

Supervisor mode is deliberately restricted in terms of interactions with underlying physical hardware, such as physical memory and device interrupts, to support clean virtualization.

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 (hypervisor or machine) 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.

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

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

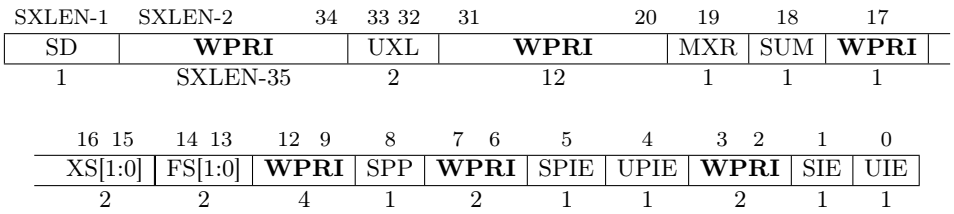


Figure 4.2: Supervisor-mode status register (`sstatus`) for RV64.

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 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 UIE bit enables or disables user-mode interrupts. User-level interrupts are enabled only if UIE is set and the hart is running in user-mode. The UPIE bit indicates whether user-level interrupts were enabled prior to taking a user-level trap. When a URET instruction is executed, UIE is set to UPIE, and UPIE is set to 1. User-level interrupts are optional. If omitted, the UIE and UPIE bits are hardwired to zero.

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`*.*

4.1.2 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.

4.1.3 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 4.15) 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 4.15) 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.

4.1.4 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 always be at least 4-byte aligned, and the MODE setting may impose additional alignment constraints on the value in the BASE 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,

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.

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.

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 for supervisor software, since user-mode software interrupts are either disabled or delegated to user mode.

4.1.5 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.

SXLEN-1 … 10	9	8	7 … 6	5	4	3 … 2	1	0
WPRI	SEIP	UEIP	**WPRI**	STIP	UTIP	**WPRI**	SSIP	USIP
SXLEN-10	1	1	2	1	1	2	1	1

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

SXLEN-1 … 10	9	8	7 … 6	5	4	3 … 2	1	0
WPRI	SEIE	UEIE	**WPRI**	STIE	UTIE	**WPRI**	SSIE	USIE
SXLEN-10	1	1	2	1	1	2	1	1

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

Three types of interrupts are defined: software interrupts, timer interrupts, and external interrupts. A supervisor-level software interrupt is triggered on the current hart by writing 1 to its supervisor software interrupt-pending (SSIP) bit in the `sip` register. A pending supervisor-level software interrupt can be cleared by writing 0 to the SSIP bit in `sip`. Supervisor-level software interrupts are disabled when the SSIE bit in the `sie` register is clear.

Interprocessor interrupts are sent to other harts by means of SBI calls, which will ultimately cause the SSIP bit to be set in the recipient hart's `sip` register.

A user-level software interrupt is triggered on the current hart by writing 1 to its user software interrupt-pending (USIP) bit in the `sip` register. A pending user-level software interrupt can be cleared by writing 0 to the USIP bit in `sip`. User-level software interrupts are disabled when the

USIE bit in the `sie` register is clear. If user-level interrupts are not supported, USIP and USIE are hardwired to zero.

All bits besides SSIP, USIP, and UEIP in the `sip` register are read-only.

A supervisor-level timer interrupt is pending if the STIP bit in the `sip` register is set. Supervisor-level timer interrupts are disabled when the STIE bit in the `sie` register is clear. An SBI call to the SEE may be used to clear the pending timer interrupt.

A user-level timer interrupt is pending if the UTIP bit in the `sip` register is set. User-level timer interrupts are disabled when the UTIE bit in the `sie` register is clear. If user-level interrupts are supported, the ABI should provide a facility for scheduling timer interrupts in terms of real-time counter values. If user-level interrupts are not supported, UTIP and UTIE are hardwired to zero.

A supervisor-level external interrupt is pending if the SEIP bit in the `sip` register is set. Supervisor-level external interrupts are disabled when the SEIE bit in the `sie` register is clear. The SBI should provide facilities to mask, unmask, and query the cause of external interrupts.

The UEIP field in `sip` contains a single read-write bit. UEIP may be written by S-mode software to indicate to U-mode that an external interrupt is pending. Additionally, the platform-level interrupt controller may generate user-level external interrupts. The logical-OR of the software-writable bit and the signal from the external interrupt controller are used to generate external interrupts for user mode. When the UEIP bit is read with a CSRRW, CSRRS, or CSRRC instruction, the value returned in the `rd` destination register contains 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 is only the software-writable UEIP bit, ignoring the interrupt value from the external interrupt controller.

Analogous to SEIP, the UEIP field behavior is designed to allow a higher privilege layer to mimic external interrupts without losing any real external interrupts.

User-level external interrupts are disabled when the UEIE bit in the `sie` register is clear. If the N extension for user-level interrupts is not implemented, UEIP and UEIE are hardwired to zero.

The `sip` *and* `sie` *registers are subsets of the* `mip` *and* `mie` *registers. Reading any 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 marked* ***WPRI*** *in Figures 4.4 and 4.5.*

4.1.6 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 SBI should provide a mechanism to modify the counter values.

The SBI must provide a facility for scheduling timer interrupts in terms of the real-time counter, `time`.

4.1.7 Counter-Enable Register (scounteren)

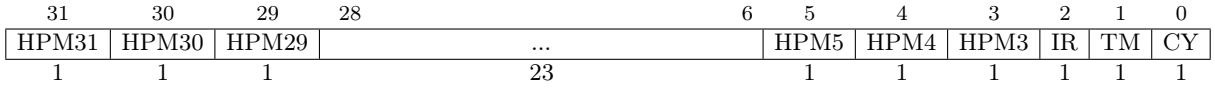


Figure 4.6: Counter-enable register (scounteren).

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 HPM*n* 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.

4.1.8 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 4.7: Supervisor Scratch Register.

4.1.9 Supervisor Exception Program Counter (sepc)

sepc is an SXLEN-bit read/write register formatted as shown in Figure 4.8. 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 physical and 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.

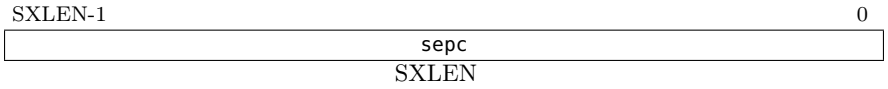


Figure 4.8: Supervisor exception program counter register.

4.1.10 Supervisor Cause Register (`scause`)

The `scause` register is an SXLEN-bit read-write register formatted as shown in Figure 4.9. 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, in descending order of priority. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

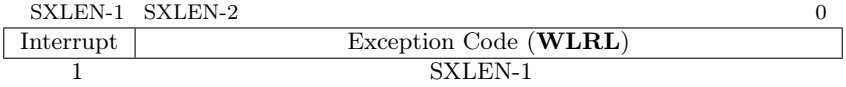


Figure 4.9: Supervisor Cause register `scause`.

4.1.11 Supervisor Trap Value (`stval`) Register

The `stval` register is an SXLEN-bit read-write register formatted as shown in Figure 4.10. 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-fetch, load, or store address-misaligned, access, 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.

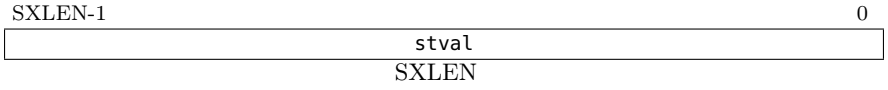


Figure 4.10: Supervisor Trap Value register.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	≥10	*Reserved*
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	*Reserved*

Table 4.2: Supervisor cause register (`scause`) values after trap.

For misaligned loads and stores that cause access or page-fault exceptions, `stval` will contain the virtual address of the portion of the access that caused the fault. For instruction-fetch access 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 physical and 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.

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

The `satp` register is an SXLEN-bit read/write register, formatted as shown in Figure 4.11 for SXLEN=32 and Figure 4.12 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 3.1.10.

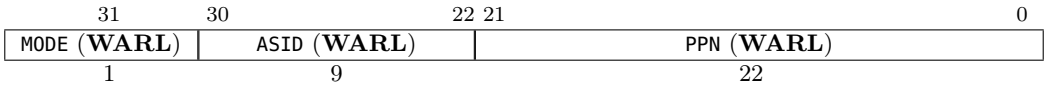


Figure 4.11: RV32 Supervisor address translation and protection register `satp`.

Storing a PPN in `satp`*, rather than a physical address, supports a physical address space larger than 4 GiB for RV32.*

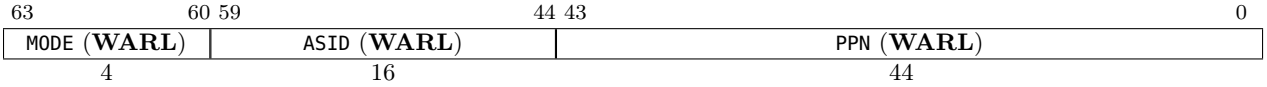


Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

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. In this case, the remaining fields in `satp` have no effect.

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 supervisor physical address bits is implementation-defined; any unimplemented address bits are hardwired to zero in the `satp` register. The number of ASID bits is also implementation-defined 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.

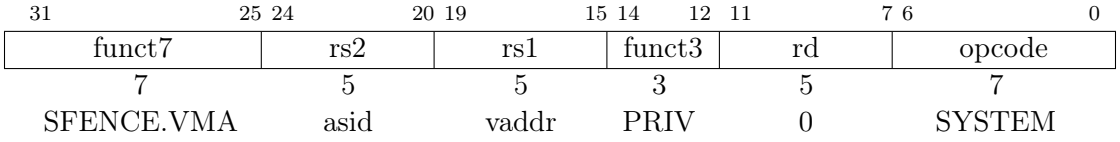
Note that writing `satp` does not imply any ordering constraints between page-table updates and subsequent address translations. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an SFENCE.VMA instruction (see Section 4.2.1) after writing `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



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 loads and stores in the instruction stream. Executing an SFENCE.VMA instruction guarantees that any stores in the instruction stream prior to the SFENCE.VMA are ordered before all implicit references subsequent to the SFENCE.VMA. Further details on the behavior of this instruction are described in Section 3.1.10 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.

Note the instruction has no effect on the translations of other harts, which must be notified separately. 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. Alternatively, implementations might provide direct hardware support for remote TLB invalidation. TLB shootdowns are handled by an SBI call to hide implementation details.

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 the leaf page table entry 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 the leaf page table entry 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.*

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

When Sv32 is written to the MODE field in the `satp` register (see Section 4.1.12), 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 4.13. 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.

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 4.15. 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

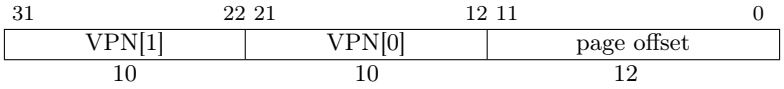


Figure 4.13: Sv32 virtual address.

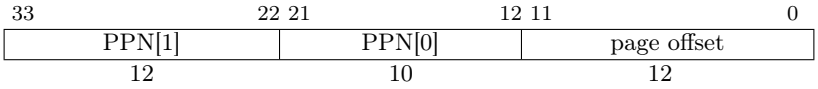


Figure 4.14: Sv32 physical address.

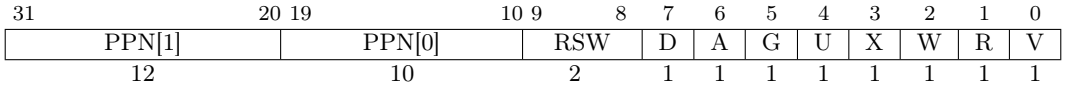


Figure 4.15: Sv32 page table entry.

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 4.1.3) 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 an error.

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. 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.

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 use and must be cleared by software for forward compatibility.

4.3.2 Virtual Address Translation Process

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

1. Let a be `satp`.*ppn* × PAGESIZE, and let i = LEVELS − 1. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.)
2. Let *pte* be the value of the PTE at address $a+va.vpn[i]\times$PTESIZE. (For Sv32, PTESIZE=4.) If accessing *pte* violates a PMA or PMP check, raise an access exception corresponding to the original access type.
3. 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.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the *pte.r*, *pte.w*, *pte.x*, and *pte.u* bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception corresponding to the original access type.
6. If $i > 0$ and $pte.ppn[i-1:0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
7. If $pte.a = 0$, or if the 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 exception corresponding to the original access type.
   - This update and the loading of *pte* in step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.
8. The translation is successful. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i-1:0] = va.vpn[i-1:0]$.
   - $pa.ppn[\text{LEVELS}-1:i] = pte.ppn[\text{LEVELS}-1:i]$.

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 4.16. 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.

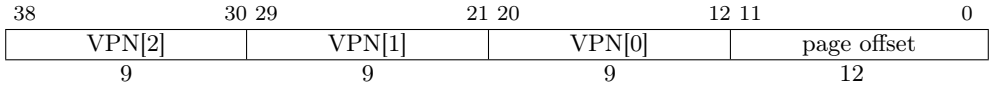


Figure 4.16: Sv39 virtual address.

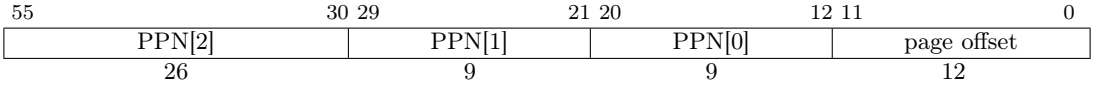


Figure 4.17: Sv39 physical address.

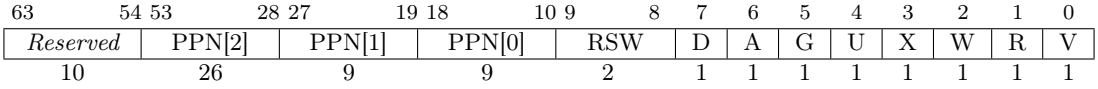


Figure 4.18: Sv39 page table entry.

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 4.18. Bits 9–0 have the same meaning as for Sv32. Bits 63–54 are reserved for future 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 LEVELS equals 3 and PTESIZE equals 8.

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 4.19. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else a page-fault exception will occur. The 36-bit VPN is translated into a 44-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

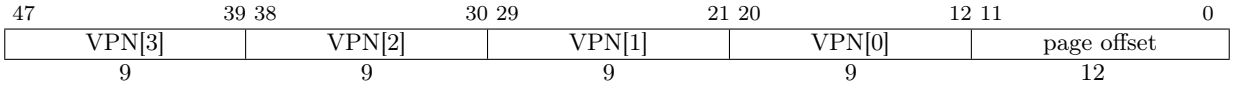


Figure 4.19: Sv48 virtual address.

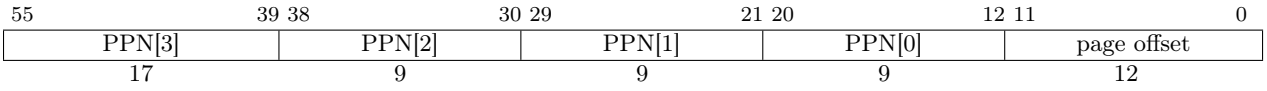


Figure 4.20: Sv48 physical address.

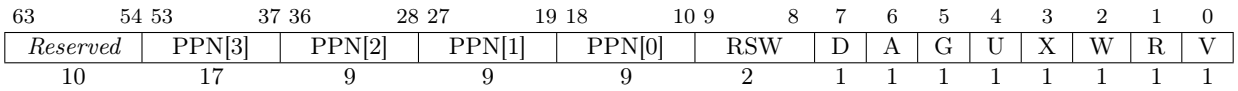


Figure 4.21: Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 4.21. 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*, 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 LEVELS equals 4 and PTESIZE equals 8.

Chapter 5

Hypervisor Extension, Version 0.2

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 level 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 level 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.

This draft is based on earlier proposals by John Hauser and Paolo Bonzini.

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) under a guest OS running in VS-mode. When V=0, the hart is either in M-mode, in HS-mode, or in U-mode under an OS running in HS-mode. The virtualization mode also indicates whether two-level 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-Level 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 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-level address translation and to control the behavior of a VS-mode guest: `hstatus`, `hedeleg`, `hideleg`, and `hgatp`.

Additionally, several *background* supervisor CSRs are copies of one of the existing *foreground* supervisor CSRs. For example, the `bsstatus` CSR is the background copy of the foreground `sstatus` CSR. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the background supervisor CSRs with their foreground counterparts. When V=0, the background supervisor CSRs contain VS-mode's version of those CSRs, and the foreground supervisor CSRs contain HS-mode's version. When V=1, the background supervisor CSRs contain HS-mode's version, and the foreground supervisor CSRs contain VS-mode's version. The background registers are accessible to HS-mode, but not to VS-mode.

The swapping of foreground and background supervisor registers can be implemented either by physically copying bits or by dynamically changing the CSR addresses of hardware registers. The CSR addresses of the background supervisor registers have been aligned with their foreground counterparts to minimize the cost of swapping registers simply by changing their addresses.

In this section, we use the term *HSXLEN* to refer to the effective XLEN when executing in HS-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. The `hstatus` register provides facilities analogous to the `mstatus` register that track and control the exception behavior of a VS-mode guest.

HSXLEN-1 23	22	21	20	19 11	10	9	8	7	6 1	0
WPRI	VTSR	VTW	VTVM	**WPRI**	SPV	STL	SP2P	SP2V	**WPRI**	SPRV
HSXLEN-23	1	1	1	9	1	1	1	1	6	1

Figure 5.1: Hypervisor-mode status register (`hstatus`).

The `hstatus` fields VTSR, VTW, and VTVM are defined analogously to the `mstatus` fields TSR, TW, and TVM, but affect the trapping behavior of the SRET, WFI, and virtual-memory management instructions only when V=1.

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 a trap is taken into HS-mode, bits SP2V and SP2P are set to the values that SPV and the HS-level SPP had before the trap. (Before the trap, the HS-level SPP is `sstatus`.SPP if V=0, or `bsstatus`.SPP if V=1.) When an SRET instruction is executed when V=0, the reverse assignments occur: after SPV and `sstatus`.SPP have supplied the new virtualization and privilege modes, they are written with SP2V and SP2P, respectively.

The STL bit (Supervisor Translation Level), which indicates which address-translation level caused an access-fault or page-fault exception, is also written by the implementation whenever a trap is taken into HS-mode. On an access or page fault due to guest physical address translation, STL is set to 1. For any other trap into HS-mode, STL is set to 0.

The SPRV bit modifies the privilege with which loads and stores execute when not in M-mode. When SPRV=0, translation and protection behave as normal. When SPRV=1, load and store memory addresses are translated and protected as though the current virtualization mode were set to `hstatus`.SPV and the current privilege mode were set to the HS-level SPP (`sstatus`.SPP when V=0, or `bsstatus`.SPP when V=1). Table 5.2 enumerates the cases.

For simplicity, SPRV is in effect even when in U-mode, VS-mode, or VU-mode, but in normal use will only be enabled for short sequences in HS-mode.

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

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`.

SPRV	SPV	SPP	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	1	0	VU-level access with two-level translation and protection. The HS-level MXR bit makes any executable page readable. The VS-level MXR makes readable those pages marked executable at the VS translation level only if readable at the guest-physical translation level.
1	1	1	VS-level access with two-level translation and protection. The HS-level MXR bit makes any executable page readable. The VS-level MXR makes readable those pages marked executable at the VS translation level only if readable at the guest-physical translation level. The VS-level SUM bit applies instead of HS-level SUM.

Table 5.2: Effect on load and store translation and protection under SPRV.

Figure 5.2: Hypervisor Exception Delegation Register `hedeleg`.

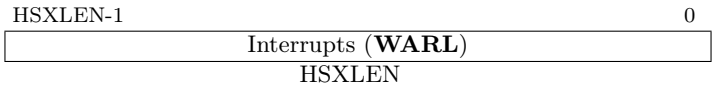


Figure 5.3: Hypervisor Interrupt Delegation Register `hideleg`.

The `hedeleg` and `hideleg` registers are only active when V=1. When V=1, a trap that has been delegated to HS-mode (using `medeleg` or `mideleg`) is further delegated to VS-mode if the corresponding `hedeleg` or `hideleg` bit is set. If the N extension for user-mode interrupts is implemented, the VS-mode guest may further delegate the interrupt to VU-mode by setting the corresponding bit in `sedeleg` or `sideleg`.

When V=0 and the N extension for user-mode interrupts is implemented, a trap that has been delegated to HS-mode can be further delegated to U-mode by setting the corresponding bit in `sedeleg` or `sideleg`.

When an access-fault or page-fault exception is caused by guest physical address translation, the trap is not delegated beyond HS-mode, regardless of the setting of `hedeleg`.

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

The `hgatp` register is an HSXLEN-bit read/write register, formatted as shown in Figure 5.4 for HSXLEN=32 and Figure 5.5 for HSXLEN=64, which controls guest physical address translation and protection. Similar to CSR `satp`, this register holds the physical page number (PPN) of the guest-physical root page table; a virtual machine identifier (VMID), which facilitates address-translation

fences on a per-virtual-machine basis; and the MODE field, which selects the address-translation scheme for guest physical addresses. When `mstatus`.TVM=1, attempts to read or write `hgatp` while executing in HS-mode will raise an illegal instruction exception.

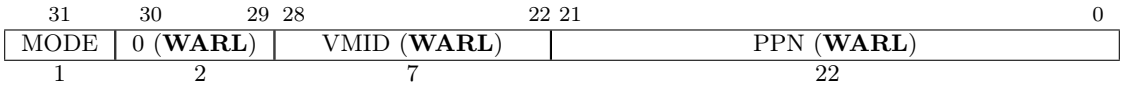


Figure 5.4: RV32 Hypervisor guest address translation and protection register `hgatp`.

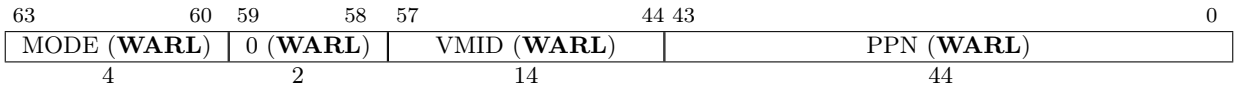


Figure 5.5: RV64 Hypervisor guest address translation and protection register `hgatp`, for MODE values Bare, Sv39x4, and Sv48x4.

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` have no effect.

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 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`.

RV64 implementations are not required to support all defined RV64 MODE settings. (However, a write to `hgatp` with an unsupported MODE value is not ignored as it is for `satp`.)

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.

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` are ignored. An implementation that supports only the defined paged virtual-memory schemes and/or Bare may hardwire PPN[1:0] to zero.

The number of supervisor physical address bits is implementation-defined; any unimplemented address bits are hardwired to zero in `hgatp`.PPN. The number of VMID bits is also implementation-defined 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 guest physical 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 5.3.1) before or after writing `hgatp`.

5.2.4 Background Supervisor Status Register (`bsstatus`)

The `bsstatus` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.6. When V=0, the `bsstatus` register holds VS-mode's version of several fields of the `sstatus` register: UXL, MXR, SUM, FS, SPP, SPIE, and SIE. When V=1, `bsstatus` holds HS-mode's version of these fields. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps these fields in `bsstatus` with their counterparts in `sstatus`. The other fields in `sstatus` are unchanged.

When V=1, both `bsstatus`.FS and `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).

When V=0, `bsstatus` does not directly affect the behavior of the machine, unless the MPRV feature in the `mstatus` register or the SPRV feature in the `hstatus` register is used to execute a load or store *as though* V=1.

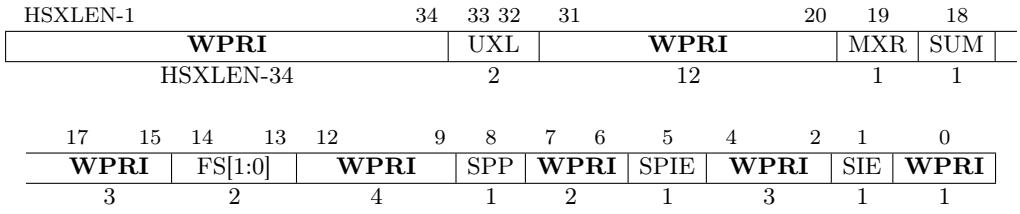


Figure 5.6: Background supervisor status register (`bsstatus`) for RV64.

5.2.5 Background Supervisor Interrupt Registers (`bsip` and `bsie`)

The `bsip` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.7. When V=0, the `bsip` register holds VS-mode's version of the `sip` register. When V=1, `bsip` holds HS-mode's version of the `sip` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the defined fields of `bsip` with their counterparts in `sip`. The other fields in `sip` are unchanged.

[NOTE: Need to describe how `bsip`.SEIP interacts with PLIC. Current thinking is that `bsip`.SEIP should purely be a read-write storage bit to emulate the PLIC for VS-mode; the PLIC should not be wired into `bsip`.SEIP.]

HSXLEN-1	10	9	8	6	5	4	2	1	0
WPRI		SEIP	**WPRI**		STIP	**WPRI**		SSIP	**WPRI**
HSXLEN-10		1	3		1	3		1	1

Figure 5.7: Background supervisor interrupt-pending register (`bsip`).

The `bsie` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.8. When V=0, the `bsie` register holds VS-mode's version of the `sie` register. When V=1, `bsie` holds HS-mode's version of the `sie` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the defined fields of `bsie` with their counterparts in `sie`. The other fields in `sie` are unchanged.

HSXLEN-1	10	9	8	6	5	4	2	1	0
WPRI		SEIE	**WPRI**		STIE	**WPRI**		SSIE	**WPRI**
HSXLEN-10		1	3		1	3		1	1

Figure 5.8: Background supervisor interrupt-enable register (`bsie`).

When V=0, `bsip` and `bsie` do not affect the behavior of the machine. When V=1, they hold the active interrupt-pending and interrupt-enable bits, respectively, for HS-mode; if any bit position holds a 1 in both registers, an interrupt will be taken.

The `bsip` and `bsie` CSRs do not hold copies of the user-mode interrupt fields. The expectation is that the context-switch code will swap the `uip` and `uie` CSRs along with the other user-mode interrupt registers (`ustatus`, `utvec`, etc.) if that feature is enabled.

5.2.6 Background Supervisor Trap Vector Base Address Register (`bstvec`)

The `bstvec` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.9. When V=0, the `bstvec` register holds VS-mode's version of the `stvec` register. When V=1, `bstvec` holds HS-mode's version of the `stvec` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the contents of `bstvec` and `stvec`.

When V=0, `bstvec` does not directly affect the behavior of the machine. When V=1, it controls the value to which the `pc` will be set upon a trap into HS-mode.

Figure 5.9: Background supervisor trap vector base address register (`bstvec`).

5.2.7 Background Supervisor Scratch Register (`bsscratch`)

The `bsscratch` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.10. When V=0, the `bsscratch` register holds VS-mode's version of the `sscratch` register. When V=1, `bsscratch` holds HS-mode's version of the `sscratch` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the contents of `bsscratch` and `sscratch`.

Typically, `bsscratch` is used to hold a pointer to the hart-local hypervisor context (when V=1) or supervisor context (when V=0). The contents of `bsscratch` do not directly affect the behavior of the machine.

Figure 5.10: Background supervisor scratch register (`bsscratch`).

5.2.8 Background Supervisor Exception Program Counter (`bsepc`)

The `bsepc` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.11. When V=0, the `bsepc` register holds VS-mode's version of the `sepc` register. When V=1, `bsepc` holds HS-mode's version of the `sepc` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the contents of `bsepc` and `sepc`.

The contents of `bsepc` do not directly affect the behavior of the machine.

`bsepc` is a **WARL** register that must be able to hold the same set of values that `sepc` can hold.

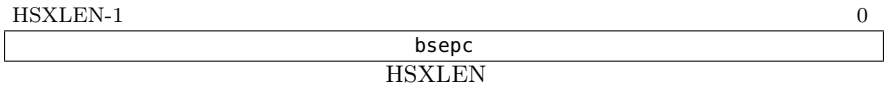


Figure 5.11: Background supervisor exception program counter (`bsepc`).

5.2.9 Background Supervisor Cause Register (`bscause`)

The `bscause` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.12. When V=0, the `bscause` register holds VS-mode's version of the `scause` register. When V=1, `bscause` holds HS-mode's version of the `scause` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the contents of `bscause` and `scause`.

The contents of `bscause` do not directly affect the behavior of the machine.

`bscause` is a **WLRL** register that must be able to hold the same set of values that `scause` can hold.

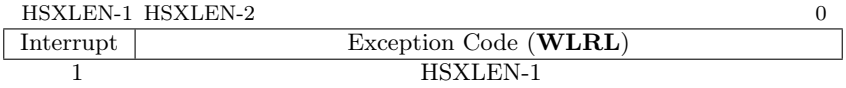


Figure 5.12: Background supervisor cause register (`bscause`).

5.2.10 Background Supervisor Trap Value Register (`bstval`)

The `bstval` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.13. When V=0, the `bstval` register holds VS-mode's version of the `stval` register. When V=1, `bstval` holds HS-mode's version of the `stval` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the contents of `bstval` and `stval`.

The contents of `bstval` do not directly affect the behavior of the machine.

`bstval` is a **WARL** register that must be able to hold the same set of values that `stval` can hold.

Figure 5.13: Background supervisor trap value register (`bstval`).

5.2.11 Background Supervisor Address Translation and Protection Register (`bsatp`)

The `bsatp` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.14 for RV32 and Figure 5.15 for RV64. When V=0, the `bsatp` register holds VS-mode's version of the `satp` register. When V=1, `bsatp` holds HS-mode's version of the `satp` register. When transitioning between virtualization modes (V=0 to V=1, or vice-versa), the implementation swaps the contents of `bsatp` and `satp`.

`bsatp` does not directly affect the behavior of the machine, unless the MPRV feature in the `mstatus` register or the SPRV feature in the `hstatus` register is used to execute a load or store *as though* V is the opposite of its actual setting. The interpretation of the MODE, ASID, and PPN fields is the same as for `satp`.

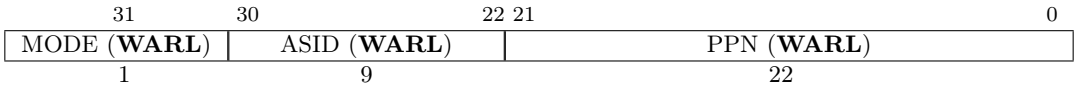


Figure 5.14: RV32 background supervisor address translation and protection register `bsatp`.

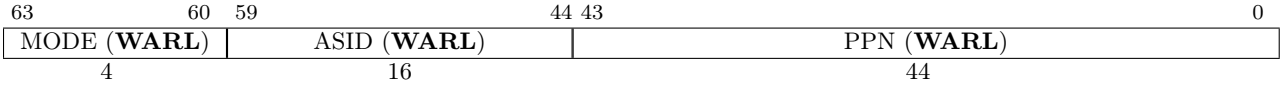


Figure 5.15: RV64 background supervisor address translation and protection register `bsatp`, for MODE values Bare, Sv39, and Sv48.

5.3 Hypervisor Instructions

The hypervisor extension adds two new privileged fence instructions.

5.3.1 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.GVMA	vmid	gaddr	PRIV	0	SYSTEM
HFENCE.BVMA	asid	vaddr	PRIV	0	SYSTEM

The hypervisor memory-management fence instructions, HFENCE.GVMA and HFENCE.BVMA, are valid only in HS-mode when `mstatus`.TVM=0, or in M-mode (irrespective of `mstatus`.TVM). These instructions perform a function similar to SFENCE.VMA (Section 4.2.1), except applying to the guest-physical memory-management data structures controlled by CSR `hgatp` (HFENCE.GVMA) or the VS-level memory-management data structures controlled by CSR `bsatp` (HFENCE.BVMA). Instruction SFENCE.VMA applies only to the memory-management data structures controlled by the foreground `satp`.

If an HFENCE.BVMA instruction executes without trapping, its effect is much the same as temporarily entering VS-mode (with the usual swapping of foreground and background supervisor registers) and executing SFENCE.VMA. Executing an HFENCE.BVMA guarantees that any stores in the instruction stream prior to the HFENCE.BVMA are ordered before implicit references to VS-level memory-management data structures when those implicit references

- are subsequent to the HFENCE.BVMA, and
- occur when `hgatp`.VMID has the same setting as it did when HFENCE.BVMA executed.

Implicit references need not be ordered when `hgatp`.VMID is different than at the time HFENCE.BVMA 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.BVMA instruction applies only to a single virtual machine, identified by the setting of `hgatp`*.VMID when HFENCE.BVMA 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.BVMA 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.*

Executing an HFENCE.GVMA instruction guarantees that any stores in the instruction stream prior to the HFENCE.GVMA are ordered before all implicit references to guest-physical memory-management data structures subsequent to 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).

For HFENCE.GVMA, a guest physical address specified in rs1 *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* rs1 *specifies bits 33:2 of such an 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).*

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.*

5.4 Machine-Level CSRs

The hypervisor extension augments the `mstatus` CSR.

5.4.1 Machine Status Register (`mstatus`)

The hypervisor extension adds two fields to the machine-mode `mstatus` CSR, MPV and MTL, and modifies the behavior of several existing fields. Figure 5.16 shows the `mstatus` register when the hypervisor extension is provided.

MXLEN-1	MXLEN-2 36	35 34	33 32	31 23	22	21	20	19	18	17	
SD	**WPRI**	SXL[1:0]	UXL[1:0]	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV	
1	MXLEN-37	2	2	9	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]	MPV	MTL	SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2	2	2	1	1	1	1	1	1	1	1	1	1	1

Figure 5.16: Machine-mode status register (`mstatus`) for RV64.

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.

The MTL bit (Machine Translation Level), which indicates which address-translation level caused an access-fault or page-fault exception, is also written by the implementation whenever a trap is taken into M-mode. On an access or page fault due to guest physical address translation, MTL is set to 1. For any other trap into M-mode, MTL is set to 0.

The SXL field controls the value of XLEN for HS-mode. The UXL field controls the value of XLEN for VS-mode or U-mode when V=0, or for VU-mode when V=1.

The TSR and TVM fields only affect execution in HS-mode.

The TW field affects execution in all modes except M-mode.

The hypervisor extension changes the behavior of the the Modify Privilege field, MPRV. When MPRV=0, translation and protection behave as normal. When MPRV=1, loads and stores are translated and protected as though the current privilege mode were set to MPP and the current virtualization mode were set to MPV. 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-level translation and protection. The HS-level MXR bit makes any executable page readable. The VS-level MXR makes readable those pages marked executable at the VS translation level only if readable at the guest-physical translation level.
1	1	1	VS-level access with two-level translation and protection. The HS-level MXR bit makes any executable page readable. The VS-level MXR makes readable those pages marked executable at the VS translation level only if readable at the guest-physical translation level. The VS-level SUM bit applies instead of HS-level SUM.

Table 5.4: Effect on load and store translation and protection under MPRV. When MPRV=1, MPP≠3, and `hstatus`.SPRV=1, the effective privilege is further modified: `hstatus`.SPV applies instead of MPV, and the HS-level SPP applies instead of MPP.

The `mstatus` register is a superset of the foreground `sstatus` register; modifying a field in `sstatus` modifies the homonymous field in `mstatus`, and vice-versa.

5.5 Two-Level Address Translation

Whenever the current virtualization mode V is 1 (and assuming both `mstatus`.MPRV=0 and `hstatus`.SPRV=0), two-level address translation and protection is in effect. For any virtual memory access, the original virtual address is first converted by VS-level address translation, as controlled by the VS-level `satp` register, into a *guest physical address*. The guest physical address is then converted by guest physical address translation, as controlled by the `hgatp` register, into a supervisor physical address. Although there is no option to disable two-level address translation when V=1, either level of translation can be effectively disabled by zeroing the corresponding `satp` or `hgatp` register.

The VS-level MXR setting, which makes execute-only pages readable, only overrides VS-level page protection. Setting MXR at VS-level does not override guest-physical page protections. Setting MXR at HS-level, however, overrides both VS-level and guest-physical execute-only permissions.

When V=1, memory accesses that would normally bypass address translation are subject to guest physical address translation alone. This includes memory accesses made in support of VS-level 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 5.2.3).

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, guest physical 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 guest physical address 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 5.17. This partitioning is identical to that for an Sv32 virtual address as depicted in Figure 4.13 (page 63), 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 4.14.)

For Sv39x4, an incoming guest physical address is partitioned as shown in Figure 5.18. This parti-

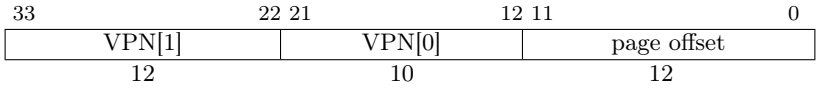


Figure 5.17: Sv32x4 virtual address (guest physical address).

tioning is identical to that for an Sv39 virtual address as depicted in Figure 4.16 (page 66), except with 2 more bits at the high end in VPN[2]. Address bits 63:41 must all be zeros, or else a page-fault exception occurs, attributed to guest physical address translation.

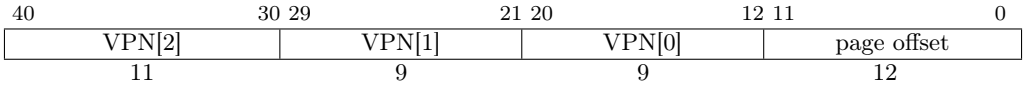


Figure 5.18: Sv39x4 virtual address (guest physical address).

For Sv48x4, an incoming guest physical address is partitioned as shown in Figure 5.19. This partitioning is identical to that for an Sv48 virtual address as depicted in Figure 4.19 (page 67), except with 2 more bits at the high end in VPN[3]. Address bits 63:50 must all be zeros, or else a page-fault exception occurs, attributed to guest physical address translation.

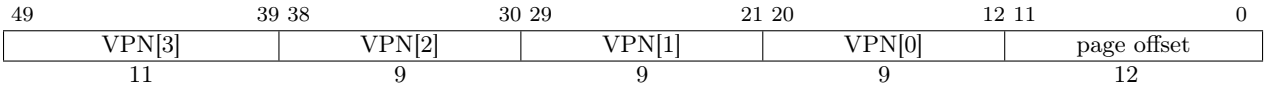


Figure 5.19: Sv48x4 virtual address (guest physical address).

The page-based guest physical 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 supporting 39-bit virtual addresses (Sv39), for example, this allows the hypervisor extension to support up to a 41-bit physical address space without either necessitating hardware support for 48-bit virtual addresses (Sv48) or falling back to emulating the larger address space with 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 × PAGESIZE;
- the current privilege mode is always taken to be U-mode; and
- instead of `mstatus`.MXR, the HS-level MXR applies (usually in `bsstatus`).

For guest physical address translation, all memory accesses (including those made to access data structures for VS-level 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

guest physical address translation the same as for VS-level address translation. For a memory access made to support VS-level 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).

Access faults and page faults raised by guest physical address translation are treated as HS-level exceptions for the purpose of exception delegation, so are not delegated to VS-mode, regardless of the setting of the `hedeleg` register.

5.5.2 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-level 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.GVMA and HFENCE.BVMA provide additional memory-management fences to complement SFENCE.VMA. These instructions are described in Section 5.3.1.

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 guest physical address translation is in use, 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.BVMA instruction is not required.

5.6 Base ISA Control

The `mstatus` field SXL determines XLEN for HS-mode.

When executing in VS-mode, XLEN is determined by the the UXL field of the background register `bsstatus`. Because `bsstatus` is swapped with `sstatus` when transitioning from VS-mode into HS-mode or M-mode, HS-mode and M-mode can modify VS-mode's XLEN via the UXL field of the foreground register `sstatus`.

When executing in U-mode or VU-mode, XLEN is determined by the UXL field of the foreground register `sstatus`.

HS-mode controls U-mode's XLEN the same way it controls VS-mode's XLEN, via `sstatus`*.UXL.*

5.7 Traps

The hypervisor extension augments the environment-call exception cause encoding. Environment calls from HS-mode use cause 9, whereas environment calls from VS-mode use cause 10. Table 5.5 lists the possible M-mode and HS-mode exception codes when the hypervisor extension is present.

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

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. If the N extension for user-mode interrupts is implemented, then U-mode traps destined for HS-mode may be further delegated to U-mode using the `sedeleg` and `sideleg` CSRs.

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. If the N extension for user-mode interrupts is implemented, then VU-mode traps destined for VS-mode may be further delegated to VU-mode using the `sedeleg` and `sideleg` CSRs.

When a trap is taken into M-mode, the following occurs: first, if the virtualization mode V was 1, the contents of the background supervisor registers are swapped with their foreground counterparts. Then, `mstatus`.MPV and `mstatus`.MPP are set according to Table 5.6.

When a trap is taken into HS-mode, the following occurs: first, if the virtualization mode V was 1, the contents of the background supervisor registers are swapped with their foreground counterparts. Then, `hstatus`.SP2V is set to `hstatus`.SPV, `hstatus`.SP2P is set to `sstatus`.SPP, and lastly `hstatus`.SPV and `sstatus`.SPP are set according to Table 5.7.

When a trap is taken into VS-mode, `sstatus`.SPP is set according to Table 5.8. Bits SP2V, SP2P, and SPV of `hstatus` are not modified, and the current virtualization state V remains 1.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	≥12	*Reserved*
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	*Reserved*

Table 5.5: Supervisor and machine cause register (`scause` and `mcause`) values when the hypervisor extension is enabled.

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` fields MPV and MPP after a trap into M-mode. Upon trap return, MPV is ignored when MPP=3.

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.

Previous Mode	SPP
VU-mode	0
VS-mode	1

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

5.8 Trap Return

The MRET instruction is used to return from a trap taken into M-mode. MRET sets the privilege mode according to the values of MPP and MPV in `mstatus`, as encoded in Table 5.6. MRET then in `mstatus` sets MPV=0, MPP=0, MIE=MPIE, and MPIE=1, and also sets `pc`=`mepc`. Finally, if the new virtualization mode V=1, the contents of the background supervisor registers are swapped with their foreground counterparts.

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 sets the virtualization and privilege modes according to the values in `hstatus`.SPV and `sstatus`.SPP, as encoded in Table 5.7. SRET then sets `hstatus`.SPV=`hstatus`.SP2V, `sstatus`.SPP=`hstatus`.SP2P, `hstatus`.SP2V=0, `hstatus`.SP2P=0, `sstatus`.SIE=`sstatus`.SPIE, `sstatus`.SPIE=1, and `pc`=`sepc`. Finally, if the new virtualization mode V=1, the contents of the background supervisor registers are swapped with their foreground counterparts.

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

Chapter 6

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
Memory-Management Instructions							
0001001		rs2	rs1	000	00000	1110011	SFENCE.VMA

Table 6.1: RISC-V Privileged Instructions

Chapter 7

History

7.1 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 35. `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)
}
```

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

```
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() == true
                 then Some(base + (EXTZ(c.Cause()) << 0b10))
                 else Some(base),
    TV_Reserved => None()
  }
}
```

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

```
function checked_mem_read forall 'n, 'n > 0. (t : ReadType, addr : xlenbits, width :
    ↪ atom('n), aq : bool, rl : bool, res: bool) -> MemoryOpResult(bits(8 * 'n)) =
  /* treat MMIO regions as not executable for now. TODO: this should actually come
      ↪ from PMP/PMA. */
  if t == Data & within_mmio_readable(addr, width)
  then mmio_read(addr, width)
  else if within_phys_mem(addr, width)
  then phys_mem_read(t, addr, width, aq, rl, res)
  else MemException(E_Load_Access_Fault)
```

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

A.2 Privilege Transition Instructions

The instructions that generate privilege transitions are implemented by generating the corresponding values of type `ctl_result`. The handling of these values is described below in Section A.5.

```
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)) };
```

```
  nextPC = handle_exception(cur_privilege, CTL_TRAP(t), PC);
  false
}
```

```
function clause execute SRET() = {
  match cur_privilege {
    User => handle_illegal(),
    Supervisor => if mstatus.TSR() == true
                  then handle_illegal()
                  else nextPC = handle_exception(cur_privilege, CTL_SRET(), PC),
    Machine => nextPC = handle_exception(cur_privilege, CTL_SRET(), PC)
  };
  false
}
```

```
function clause execute MRET() = {
  if cur_privilege == Machine
  then nextPC = handle_exception(cur_privilege, CTL_MRET(), PC)
  else handle_illegal();
  false
}
```

A.3 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 : bits(12), 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
    0x3B0 => false, // (Disabled for Spike compatibility)
// 0x3B0 => p == Machine, // pmpaddr0

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

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

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

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

    _ => false
  }
```

```
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() == true)
```

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

    (0xC00, User) => scounteren.CY() == true,
    (0xC01, User) => scounteren.TM() == true,
    (0xC02, User) => scounteren.IR() == true,

    (_, _) => /* 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.4 Interrupt Delegation and Dispatch

curInterrupt 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() == true then Some(I_M_External)
 else if ip.MSI() == true then Some(I_M_Software)
 else if ip.MTI() == true then Some(I_M_Timer)
 else if ip.SEI() == true then Some(I_S_External)
 else if ip.SSI() == true then Some(I_S_Software)
 else if ip.STI() == true then Some(I_S_Timer)
 else if ip.UEI() == true then Some(I_U_External)
 else if ip.USI() == true then Some(I_U_Software)
 else if ip.UTI() == true then Some(I_U_Timer)
 else None()
}
```

```
function curInterrupt(priv : Privilege, pend : Minterrupts, enbl : Minterrupts, delg
    ↪ : Minterrupts)
              -> option((InterruptType, Privilege)) = {
 let en_mip : xlenbits = pend.bits() & enbl.bits();
 if en_mip == EXTZ(0b0) then None() /* fast path */
 else {
  /* check implicit enabling when in lower privileges */
  let eff_mie = priv != Machine | (priv == Machine & mstatus.MIE() == true);
  let eff_sie = priv == User | (priv == Supervisor & mstatus.SIE() == true);
  /* handle delegation */
  let eff_mip = en_mip & (~ (delg.bits())); /* retained at M-mode */
  let eff_sip = en_mip & delg.bits(); /* delegated to S-mode */

  if eff_mie & eff_mip != EXTZ(0b0)
  then match findPendingInterrupt(eff_mip) {
       Some(i) => let r = (i, Machine) in Some(r),
       None() => { internal_error("non-zero eff_mip=" ^ BitStr(eff_mip) ^ ", but
           ↪ nothing pending") }
     }
  else if eff_sie & eff_sip != EXTZ(0b0)
  then match findPendingInterrupt(eff_sip) {
       Some(i) => let r = (i, Supervisor) in Some(r),
       None() => { internal_error("non-zero eff_sip=" ^ BitStr(eff_sip) ^ ", but
           ↪ nothing pending") }
     }
  else {
    let p = if pend.MTI() == true then "1" else "0";
    let e = if enbl.MTI() == true then "1" else "0";
    let d = if delg.MTI() == true then "1" else "0";
    print_platform(" MTI: pend=" ^ p ^ " enbl=" ^ e ^ " delg=" ^ d);
    let eff_mip = en_mip & (~ (delg.bits())); /* retained at M-mode */
    let eff_sip = en_mip & delg.bits(); /* delegated to S-mode */
    print_platform("mstatus=" ^ BitStr(mstatus.bits())
              ^ " mie,sie=" ^ BitStr(mstatus.MIE()) ^ "," ^ BitStr(mstatus.SIE())
              ^ " en_mip=" ^ BitStr(en_mip)
```

```
                ^ " eff_mip=" ^ BitStr(eff_mip)
                ^ " eff_sip=" ^ BitStr(eff_sip));
    None()
   }
  }
}
```

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. *handle_exception* processes traps and privilege transition instructions, and delegates trap handling to *handle_trap*. *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 = medeleg.bits()[idx];
  let user = sedeleg.bits()[idx];
  let deleg = /* if misa.N() == true & user then User
               else */
              if misa.S() == true & super then Supervisor
              else Machine;
  /* Ensure there is no transition to a less-privileged mode. */
  if privLevel_to_bits(deleg) <_u privLevel_to_bits(p)
  then p else deleg
}
```

```
function handle_trap(del_priv : Privilege, intr : bool, c : exc_code, pc : xlenbits,
    ↪ info : option(xlenbits))
                -> xlenbits = {
  rvfi_trap();
  print_platform("handling " ^ (if intr then "int" else "exc") ^ BitStr(c) ^ " at
      ↪ priv " ^ del_priv ^ " with tval " ^ BitStr(tval(info)));

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

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

      cur_privilege = del_priv;

      print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()) ^ " (input: " ^ BitStr(
          ↪ mstatus.bits()) ^ ")"); // Spike compatible log

      cancel_reservation();

```

```
      match tvec_addr(mtvec, mcause) {
        Some(epc) => epc,
        None() => internal_error("Invalid mtvec mode")
      }
    },
    Supervisor => {
      scause->IsInterrupt() = intr;
      scause->Cause() = EXTZ(c);

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

      cur_privilege = del_priv;

      print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()) ^ " (input: " ^ BitStr(
          ↪ mstatus.bits()) ^ ")"); // Spike compatible log

      cancel_reservation();

      match tvec_addr(stvec, scause) {
        Some(epc) => epc,
        None() => internal_error("Invalid stvec mode")
      }

    },
    User => internal_error("the N extension is currently unsupported")
  };
}
```

```
function handle_exception(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);
      print_platform("trapping from " ^ cur_priv ^ " to " ^ del_priv
                     ^ " to handle " ^ e.trap);
      handle_trap(del_priv, false, e.trap, pc, e.excinfo)
    },
    (_, CTL_MRET()) => {
      let prev_priv = cur_privilege;
      mstatus->MIE() = mstatus.MPIE();
      mstatus->MPIE() = true;
      cur_privilege = privLevel_of_bits(mstatus.MPP());
      mstatus->MPP() = privLevel_to_bits(User);

      print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()) ^ " (input: " ^ BitStr(
          ↪ mstatus.bits()) ^ ")"); // Spike compatible log
      print_platform("ret-ing from " ^ prev_priv ^ " to " ^ cur_privilege);

      cancel_reservation();
      mepc & pc_alignment_mask()
    },
    (_, CTL_SRET()) => {
```

```
    let prev_priv = cur_privilege;
    mstatus->SIE() = mstatus.SPIE();
    mstatus->SPIE() = true;
    cur_privilege = if mstatus.SPP() == true then Supervisor else User;
    mstatus->SPP() = false;

    print_reg("CSR mstatus <- " ^ BitStr(mstatus.bits()) ^ " (input: " ^ BitStr(
        ↪ mstatus.bits()) ^ ")"); // Spike compatible log
    print_platform("ret-ing from " ^ prev_priv ^ " to " ^ cur_privilege);

    cancel_reservation();
    sepc & pc_alignment_mask()
   }
  }
}
```

```
function handle_mem_exception(addr : xlenbits, e : ExceptionType) -> unit = {
 let t : sync_exception = struct { trap = e,
                           excinfo = Some(addr) } in
 nextPC = handle_exception(cur_privilege, CTL_TRAP(t), PC)
}
```

```
function handle_interrupt(i : InterruptType, del_priv : Privilege) -> unit =
 nextPC = handle_trap(del_priv, true, i, PC, 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 };
 nextPC = handle_exception(cur_privilege, CTL_TRAP(t), PC)
}
```

A.6 Virtual Memory

This section covers the support of virtual memory in supervisor mode.

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)
```

```
function isPTEPtr(p : pteAttribs) -> bool = {
 let a = Mk_PTE_Bits(p);
 a.R() == false & a.W() == false & a.X() == false
}
```

```
function isInvalidPTE(p : pteAttribs) -> bool = {
  let a = Mk_PTE_Bits(p);
  a.V() == false | (a.W() == true & a.R() == false)
}
```

```
function checkPTEPermission(ac : AccessType, priv : Privilege, mxr : bool, do_sum :
    ↪ bool, p : PTE_Bits) -> bool = {
  match (ac, priv) {
    (Read, User) => p.U() == true & (p.R() == true | (p.X() == true & mxr)),
    (Write, User) => p.U() == true & p.W() == true,
    (ReadWrite, User) => p.U() == true & p.W() == true & (p.R() == true | (p.X() ==
        ↪ true & mxr)),
    (Execute, User) => p.U() == true & p.X() == true,

    (Read, Supervisor) => (p.U() == false | do_sum) & (p.R() == true | (p.X() == true
        ↪  & mxr)),
    (Write, Supervisor) => (p.U() == false | do_sum) & p.W() == true,
    (ReadWrite, Supervisor) => (p.U() == false | do_sum) & p.W() == true & (p.R() ==
        ↪ true | (p.X() == true & mxr)),
    (Execute, Supervisor) => p.U() == false & p.X() == true,

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

```
function update_PTE_Bits(p : PTE_Bits, a : AccessType) -> option(PTE_Bits) = {
  let update_d = (a == Write | a == ReadWrite) & p.D() == false; // dirty-bit
  let update_a = p.A() == false; // accessed-bit
  if update_d | update_a then {
    let np = update_A(p, true);
    let np = if update_d then update_D(np, true) else np;
    Some(np)
  } 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 *walk39*, 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. sailfnametranslateAddr 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.

```
enum PTW_Error = {
  PTW_Access, /* physical memory access error for a PTE */
  PTW_Invalid_PTE,
  PTW_No_Permission,
```

```
  PTW_Misaligned, /* misaligned superpage */
  PTW_PTE_Update /* PTE update needed but not enabled */
}

union PTW_Result = {
  PTW_Success: (paddr39, SV39_PTE, paddr39, nat, bool),
  PTW_Failure: PTW_Error
}

function translationException(a : AccessType, f : PTW_Error) -> ExceptionType = {
  let e : ExceptionType =
  match (a, f) {
    (ReadWrite, PTW_Access) => E_SAMO_Access_Fault,
    (ReadWrite, _) => E_SAMO_Page_Fault,
    (Read, PTW_Access) => E_Load_Access_Fault,
    (Read, _) => E_Load_Page_Fault,
    (Write, PTW_Access) => E_SAMO_Access_Fault,
    (Write, _) => E_SAMO_Page_Fault,
    (Fetch, PTW_Access) => E_Fetch_Access_Fault,
    (Fetch, _) => E_Fetch_Page_Fault
  } in {
/* print("translationException(" ^ a ^ ", " ^ f ^ ") -> " ^ e); */
    e
  }
}

function walk39(vaddr, ac, priv, mxr, do_sum, ptb, level, global) -> PTW_Result = {
  let va = Mk_SV39_Vaddr(vaddr);
  let pt_ofs : paddr39 = shiftl(EXTZ(shiftr(va.VPNi(), (level * SV39_LEVEL_BITS))[(
      ↪ SV39_LEVEL_BITS - 1) .. 0]),
                        PTE39_LOG_SIZE);
  let pte_addr = ptb + pt_ofs;
  /* FIXME: we assume here that walks only access physical-memory-backed addresses,
      ↪ and not MMIO regions. */
  match (phys_mem_read(Data, 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)
    },
    MemValue(v) => {
      let pte = Mk_SV39_PTE(v);
      let pbits = pte.BITS();
      let pattr = Mk_PTE_Bits(pbits);
      let is_global = global | (pattr.G() == true);
/* 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) then {
/* print("walk39: invalid pte"); */
        PTW_Failure(PTW_Invalid_PTE)
      } else {
        if isPTEPtr(pbits) then {
          if level == 0 then {
            /* last-level PTE contains a pointer instead of a leaf */
```

```
/* print("walk39: last-level pte contains a ptr"); */
         PTW_Failure(PTW_Invalid_PTE)
       } else {
         /* walk down the pointer to the next level */
         walk39(vaddr, ac, priv, mxr, do_sum, EXTZ(shiftl(pte.PPNi(), PAGESIZE_BITS)
             ↪ ), level - 1, is_global)
       }
     } else { /* leaf PTE */
       if ~ (checkPTEPermission(ac, priv, mxr, do_sum, pattr)) then {
/* print("walk39: pte permission check failure"); */
         PTW_Failure(PTW_No_Permission)
       } else {
         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)
           } 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)
           }
         } 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)
         }
       }
     }
   }
  }
}
```

```
function translate39(vAddr, ac, priv, mxr, do_sum, level) = {
 let asid = curAsid64();
 match lookupTLB39(asid, vAddr) {
  Some(idx, ent) => {
    let pteBits = Mk_PTE_Bits(ent.pte.BITS());
    if ~ (checkPTEPermission(ac, priv, mxr, do_sum, pteBits))
    then TR39_Failure(PTW_No_Permission)
    else {
      match update_PTE_Bits(pteBits, ac) {
       None() => TR39_Address(ent.pAddr | EXTZ(vAddr & ent.vAddrMask)),
       Some(pbits) => {
         if ~ (plat_enable_dirty_update ())
         then {
           /* pte needs dirty/accessed update but that is not enabled */
           TR39_Failure(PTW_PTE_Update)
         } else {
           /* update PTE entry and TLB */
```

```
          n_ent : TLB39_Entry = ent;
          n_ent.pte = update_BITS(ent.pte, pbits.bits());
          writeTLB39(idx, n_ent);
          /* update page table */
          match checked_mem_write(EXTZ(ent.pteAddr), 8, ent.pte.bits()) {
           MemValue(_) => (),
           MemException(e) => internal_error("invalid physical address in TLB")
          };
          TR39_Address(ent.pAddr | EXTZ(vAddr & ent.vAddrMask))
         }
        }
       }
      }
    },
    None() => {
     match walk39(vAddr, ac, priv, mxr, do_sum, curPTB39(), level, false) {
      PTW_Failure(f) => TR39_Failure(f),
      PTW_Success(pAddr, pte, pteAddr, level, global) => {
       match update_PTE_Bits(Mk_PTE_Bits(pte.BITS()), ac) {
        None() => {
         addToTLB39(asid, vAddr, pAddr, pte, pteAddr, level, global);
         TR39_Address(pAddr)
        },
        Some(pbits) =>
         if ~ (plat_enable_dirty_update ())
         then {
          /* pte needs dirty/accessed update but that is not enabled */
          TR39_Failure(PTW_PTE_Update)
         } else {
          w_pte : SV39_PTE = update_BITS(pte, pbits.bits());
          match checked_mem_write(EXTZ(pteAddr), 8, w_pte.bits()) {
           MemValue(_) => {
            addToTLB39(asid, vAddr, pAddr, w_pte, pteAddr, level, global);
            TR39_Address(pAddr)
           },
           MemException(e) => {
            /* pte is not in valid memory */
            TR39_Failure(PTW_Access)
           }
          }
         }
       }
      }
     }
    }
   }
  }
}
```

```
function translationMode(priv) = {
 if priv == Machine then Sbare
 else {
  let arch = architecture(mstatus.SXL());
  match arch {
   Some(RV64) => {
    let mbits : satp_mode = Mk_Satp64(satp).Mode();
    match satpMode_of_bits(RV64, mbits) {
     Some(m) => m,
     None() => internal_error("invalid RV64 translation mode in satp")
    }
   },
```

```
    _ => internal_error("unsupported address translation arch")
   }
  }
}
```

```
union TR_Result = {
  TR_Address : xlenbits,
  TR_Failure : ExceptionType
}
```

```
function translateAddr(vAddr, ac, rt) = {
  let effPriv : Privilege = match rt {
    Instruction => cur_privilege,
    Data => if mstatus.MPRV() == true
                then privLevel_of_bits(mstatus.MPP())
                else cur_privilege
  };
  let mxr : bool = mstatus.MXR() == true;
  let do_sum : bool = mstatus.SUM() == true;
  let mode : SATPMode = translationMode(effPriv);
  match mode {
    Sbare => TR_Address(vAddr),
    SV39 => match translate39(vAddr[38 .. 0], ac, effPriv, mxr, do_sum, SV39_LEVELS -
        ↪ 1) {
              TR39_Address(pa) => TR_Address(EXTZ(pa)),
              TR39_Failure(f) => TR_Failure(translationException(ac, f))
            },
    _ => 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.