The RISC-V Instruction Set Manual

Volume II: Privileged Architecture

Document Version 1.12-draft

Editors: Andrew Waterman[1], Krste Asanović[1,2], John Hauser
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu, jh.riscv@jhauser.us

August 13, 2021

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

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

Preface

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

Module	Version	Status
Machine ISA	*1.12*	*Draft*
Supervisor ISA	*1.12*	*Draft*
Hypervisor ISA	*0.6*	*Draft*

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

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

- Changed MRET and SRET to clear `mstatus`.MPRV when leaving M-mode.
- Reserved additional `satp` patterns for future use.
- Stated that the `scause` Exception Code field must implement bits 4–0 at minimum.
- Relaxed I/O regions have been specified to follow RVWMO. The previous specification implied that PPO rules other than fences and acquire/release annotations did not apply.
- Constrained the LR/SC reservation set size and shape when using page-based virtual memory.
- PMP changes require an SFENCE.VMA on any hart that implements page-based virtual memory, even if VM is not currently enabled.

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

- Removed the N extension.
- Defined the mandatory RV32-only CSR `mstatush`, which contains most of the same fields as the upper 32 bits of RV64's `mstatus`.
- Permitted the unconditional delegation of less-privileged interrupts.
- Added optional big-endian and bi-endian support.
- Made priority of load/store/AMO address-misaligned exceptions implementation-defined relative to load/store/AMO page-fault and access-fault exceptions.
- PMP reset values are now platform-defined.
- An additional 48 optional PMP registers have been defined.
- Software breakpoint exceptions are permitted to write either 0 or the PC to x`tval`.
- Clarified that bare S-mode need not support the SFENCE.VMA instruction.

Finally, the hypervisor architecture proposal has been extensively revised.

Preface to Version 1.11

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

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

Changes from version 1.10 include:

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

Preface to Version 1.10

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

- The previous version of this document was released under a Creative Commons Attribution 4.0 International License by the original authors, and this and future versions of this document will be released under the same license.
- The explicit convention on shadow CSR addresses has been removed to reclaim CSR space. Shadow CSRs can still be added as needed.
- The `mvendorid` register now contains the JEDEC code of the core provider as opposed to a code supplied by the Foundation. This avoids redundancy and offloads work from the Foundation.
- The interrupt-enable stack discipline has been simplified.
- An optional mechanism to change the base ISA used by supervisor and user modes has been added to the `mstatus` CSR, and the field previously called Base in `misa` has been renamed to `MXL` for consistency.
- Clarified expected use of XS to summarize additional extension state status fields in `mstatus`.
- Optional vectored interrupt support has been added to the `mtvec` and `stvec` CSRs.
- The SEIP and UEIP bits in the `mip` CSR have been redefined to support software injection of external interrupts.
- The `mbadaddr` register has been subsumed by a more general `mtval` register that can now capture bad instruction bits on an illegal instruction fault to speed instruction emulation.
- The machine-mode base-and-bounds translation and protection schemes have been removed from the specification as part of moving the virtual memory configuration to `sptbr` (now `satp`). Some of the motivation for the base and bound schemes are now covered by the PMP registers, but space remains available in `mstatus` to add these back at a later date if deemed useful.
- In systems with only M-mode, or with both M-mode and U-mode but without U-mode trap support, the `medeleg` and `mideleg` registers now do not exist, whereas previously they returned zero.
- Virtual-memory page faults now have `mcause` values distinct from physical-memory access faults. Page-fault exceptions can now be delegated to S-mode without delegating exceptions generated by PMA and PMP checks.
- An optional physical-memory protection (PMP) scheme has been proposed.
- The supervisor virtual memory configuration has been moved from the `mstatus` register to the `sptbr` register. Accordingly, the `sptbr` register has been renamed to `satp` (Supervisor Address Translation and Protection) to reflect its broadened role.
- The SFENCE.VM instruction has been removed in favor of the improved SFENCE.VMA instruction.
- The `mstatus` bit MXR has been exposed to S-mode via `sstatus`.
- The polarity of the PUM bit in `sstatus` has been inverted to shorten code sequences involving MXR. The bit has been renamed to SUM.
- Hardware management of page-table entry Accessed and Dirty bits has been made optional; simpler implementations may trap to software to set them.

- The counter-enable scheme has changed, so that S-mode can control availability of counters to U-mode.
- H-mode has been removed, as we are focusing on recursive virtualization support in S-mode. The encoding space has been reserved and may be repurposed at a later date.
- A mechanism to improve virtualization performance by trapping S-mode virtual-memory management operations has been added.
- The Supervisor Binary Interface (SBI) chapter has been removed, so that it can be maintained as a separate specification.

Preface to Version 1.9.1

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

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

Contents

Chapter 1

Introduction

此文档描述RISC-V特权架构,其覆盖RISC-V系统非特权ISA以外的所有方面,包括特权指令以及运行操作系统和附加外部器件所需要的额外功能.

我们设计决策的述评按照与此段相同进行排版,如果读者只对标准本身感兴趣则可以跳过此部分(译者注:我们使用楷体表示此述评)

我们简要的强调以下:此文档所描述的整个特权等级设计可以替换为完全不同的特权等级设计,而不需要更改非特权ISA,甚至可能不需要更改ABI.特别地,此特权标准旨在运行现现有的流行操作系统,因此具象化了传统的基于等级(level-based)的保护模型.其他的特权规范可能包含其他更灵活的保护域(protection-domain)模型.为简化表达,此文本表现的好像只有这一种特权架构.

1.1 RISC-V Privileged Software Stack Terminology

此节讲解用于描述RISC-V中特权软件栈(priviledge software stack terminology)的组件的术语.图1.1展示了RISC-V体系结构支持的一些可能的软件栈.左手边展示了一个简单的系统,其只支持单个应用运行在应用执行环境(application execution environment,AEE).此应用使用特定的应用二进制接口(application binary interface,ABI)编码运行.ABI包括所支持的用户级ISA加上一组与AEE交互的ABI调用.ABI对应用隐藏了AEE的细节以允许实现AEE时拥有更大的灵活性.在多个不同的宿主OS上可以原生地实现相同的ABI,或者由运行在一个拥有不同原生ISA的机器上的用户模式模拟环境实现此ABI.

我们的图形化约定使用黑框加上白色文本表示抽象接口,以将它们与实现接口的组件的具体实例分开.

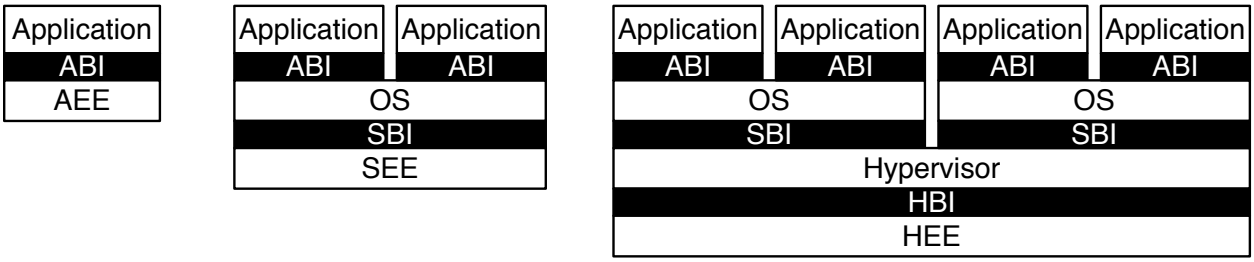


图1.1 支持不同形式的特权执行(privileged execution)的不同实现.

中间的结构展示了传统的操作系统(operating system,OS),其支持多个应用程序的执行.每个应用程序都通过ABI与OS进行通信,OS提供其AEE.就像应用程序通过ABI与AEE对接一样,RISC-V操作系统通过 supervisor binary interface(SBI) 与 supervisor execution environment(SEE) 对接. SBI包含 user-level 和 supervisor-level 指令以及一组SBI函数调用.对所有SEE的实现使用单个SBI使得单个OS二进制镜像可以运行在任意SEE上.在低端硬件平台上SEE可以是简单的boot loader和BIOS风格的IO系统,或者是在高端服务器上的hypervisor提供的虚拟机,或是在体系结构仿真环境中宿主操作系统上的一个翻译层.

大部分 supervisor-level ISA的定义都没有从执行环境和/或硬件平台中分离出SBI,使得虚拟化和新硬件平台的搭建变得复杂.

最右边的结构展示了虚拟机监视器结构,在单个 hypervisor 中支持多个多程序的OS.每个OS都通过SBI与 hypervisor 通信, hypervisor 提供 SEE. hypervisor 使用 hypervisor binary interface(HBI) 与 hypervisor execution environment(HEE)进行通信.

ABI,SBI,和HBI的工作正在进行中,我们现在优先支持第二类虚拟机,其SBI由S-mod的操作系统提供.

RISC-V的硬件实现通常需要特权ISA以外的更多特性以支持各样的执行环境(AEE,SEE,或HEE).

1.2 Privilege Levels

在任何时间,RISC-V的硬件线程(hardware thread,hart)都是运行在某个特权等级上,特权等级在一个或多个CSRs(控制和状态寄存器 control and status register,CSRs)中编码为某个模式.如Table 1.1所示,当前定义了三个RISC-V特权等级.

特权等级旨为不同软件栈的组件之间提供保护,且尝试执行当前特权模式所不被允许的操作将会引起exception.这些exception通常会导致trap到底层执行环境.

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.

在描述中,我们尝试分离代码编写的特权等级和代码所运行的特权模式,尽管它们通常是捆绑的.例如,在拥有三个特权模式的系统上,一个supervisor等级的操作系统可以运行在supervisor模式,但是也可以在经典虚拟机监视器上以用户模式运行,其虚拟机监视器所在的系统有两个或更多个特权模式.在这两种情况下,可以使用相同的supervisor等级操作系统二进制代码,其代码编码为supervisor-level SBI且期望可以使用supervisor等级的特权指令和CSRs.当以用户模式运行客操作系统时,所有的supervisor等级的行为都会trap且被运行在更高特权等级的SEE所模拟.

机器等级拥有最高特权,且是RISC-V硬件平台唯一一个被强制实现的特权等级.运行在机器模式(M-mode)的代码通常是天然可信的,因为其能访问机器的最底层.M-mode可以用来管理在RISC-V上的secure execution environment.用户模式(U-mode)和管理员模式(S-mode)分别用于传统应用程序和操作系统.

每个特权等级拥有一组核心的特权ISA拓展,带有可选的拓展和变体.例如,机器模式支持可选的内存保护标准拓展.同样,管理员模式可以拓展到支持第二类虚拟机,如第5章所述.

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.

所有的硬件实现必须提供M-mode,因为这是唯一可以无约束的访问整个机器的模式.最简单的RISC-V实现可能只提供M-mode,不过这样不能提供保护以防不正确的或有恶意的代码.

即使仅实现了M-mode,PMP拓展(可选的)的lock特性可以提供一些受限的保护.

许多RISC-V实现也会支持用户模式(U-mode)以保护系统中的其他部分不会受到应用代码的影响.添加管理员模式(S-mode)可以在管理员级别操作系统和SEE间提供隔离.

hart通常在U-mode运行应用代码,直到遇到某些trap(例如,管理员调用或计时器中断)导致强制切换到trap handler,其通常运行在特权等级更高的模式.接下来hart将执行trap handler,其最终总会恢复U-mode的执行(在遇到此trap的指令上或者之后).会提升特权等级的trap以术语vertical trap描述,而其他依然停留在相同特权级别的trap以术语horizontal trap描述.RISC-V特权体系结构为trap到不同特权层级提供了灵活的安排.

---

horizontal trap可以实现为vertical trap,再由此vertical trap将控制权返回给低特权模式下的trap handler.

1.3 Debug Mode

实现可能包含debug模式以支持片外调试和/或生产测试.debug模式(D-mode)可以认为是一个额外的特权模式,甚至比M-mode拥有更多访问权限.单独的debug规范描述了RISC-V hart在debug模式下的操作.debug模式保留了少量的只可在D-mode下访问的CSR地址,且可能也在平台上保留了部分物理地址空间.

Chapter 2

Control and Status Registers (CSRs)

在RISC-V ISA中,SYSTEM 主操作码(major opcode,指令的前7位)用来编码所有的特权指令.这些指令可以分为两个主要类别:定义在Zicsr拓展中的自动读-改-写CSRs和所有其他特权指令.特权体系结构需要Zicsr拓展;其他特权指令需要哪些CSR取决于特权体系结构特征的集合.

除了此手册卷I所描述的非特权状态,一个实现可能包含额外的CSRs,特权等级的某些子集可以通过卷I描述的CSR指令来访问这些CSRs.在此章,我们映射出CSR地址空间.接下来的章节描述每个CSRs相应于不同特权等级的功能,以及通常与特定的特权等级紧密相关的特权指令.注意,尽管CSRs和指令都与某个特权等级有关,但是它们对于高等级的特权级别(相对高级)来说都是可以访问的.

标准CSRs不会因为读取而产生副作用,但是可能会因为写而产生副作用(副作用定义在卷I).

2.1 CSR Address Mapping Conventions

标准RISC-V ISA留出了12-bit编码空间(csr[11:0])以支持最多4096个CSR.按照约定,csr地址的高4bit(csr[11:8])用来编码相应于不同特权等级的读写访问,如表 2.1所示.最高2bit(csr[11:10])用来指示此寄存器是否可读/写(00,01,或10)或只读(11).接下来的2bit(csr[9:8])编码可以访问此CSR的最低特权等级.

CSR的地址约定使用CSR地址高位编码默认访问特权.这样简化了硬件上的错误检查以及提供更大的CSR空间,但是确实限制了CSRs到此地址空间的映射.

实现可能允许一个更高特权等级trap低特权等级对CSR的访问(除非访问是被允许的)以中断此次访问.这种变化对于低特权的软件应当是透明的.

试图访问不存在的CSR将引起非法指令exception.试图访问没有适当权限等级的CSR或写只读CSR也引起非法指令exception.一个读/写寄存器可能也含有某些位是只读位,在这种情况下忽视对只读位的写入.

表 2.1指出在标准使用和自定义使用间分配CSR地址的约定.用于自定义的CSR地址在将来不会被标准拓展重新定义.

机器模式标准可读可写CSRs 0x7A0-0x7BF留作debug系统使用.其中机器模式可以访问0x7A0-0x7AF,但0x7B0-0x7BF只对debug模式可见.当机器模式访问后面这组寄存器时,实现应当产生非法指令exception.

有效的虚拟化要求尽量多的指令原生地运行在虚拟化环境中,而任何特权访问都trap到虚拟机监视器[1].对于在低特权模式下只读但是高特权模式下可读写的CSR寄存器,将会遮蔽到单独的CSR地址.这样防止了trap允许的低特权访问同时依然能trap非法访问.当前只有计数器是遮蔽的CSR.

2.2 CSR Listing

表2.2-2.6列出了当前已分配CSR地址的CSR寄存器.计数器,计时器和浮点CSRs都是标准非特权CSRs.其他寄存器都用于特权模式,在后续章节描述.注意,并不是所有的寄存器都需要实现.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000–0x0FF	Standard read/write
01	00	XXXX	0x400–0x4FF	Standard read/write
10	00	XXXX	0x800–0x8FF	Custom read/write
11	00	0XXX	0xC00–0xC7F	Standard read-only
11	00	10XX	0xC80–0xCBF	Standard read-only
11	00	11XX	0xCC0–0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100–0x1FF	Standard read/write
01	01	0XXX	0x500–0x57F	Standard read/write
01	01	10XX	0x580–0x5BF	Standard read/write
01	01	11XX	0x5C0–0x5FF	Custom read/write
10	01	0XXX	0x900–0x97F	Standard read/write
10	01	10XX	0x980–0x9BF	Standard read/write
10	01	11XX	0x9C0–0x9FF	Custom read/write
11	01	0XXX	0xD00–0xD7F	Standard read-only
11	01	10XX	0xD80–0xDBF	Standard read-only
11	01	11XX	0xDC0–0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200–0x2FF	Standard read/write
01	10	0XXX	0x600–0x67F	Standard read/write
01	10	10XX	0x680–0x6BF	Standard read/write
01	10	11XX	0x6C0–0x6FF	Custom read/write
10	10	0XXX	0xA00–0xA7F	Standard read/write
10	10	10XX	0xA80–0xABF	Standard read/write
10	10	11XX	0xAC0–0xAFF	Custom read/write
11	10	0XXX	0xE00–0xE7F	Standard read-only
11	10	10XX	0xE80–0xEBF	Standard read-only
11	10	11XX	0xEC0–0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300–0x3FF	Standard read/write
01	11	0XXX	0x700–0x77F	Standard read/write
01	11	100X	0x780–0x79F	Standard read/write
01	11	1010	0x7A0–0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0–0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0–0x7FF	Custom read/write
10	11	0XXX	0xB00–0xB7F	Standard read/write
10	11	10XX	0xB80–0xBBF	Standard read/write
10	11	11XX	0xBC0–0xBFF	Custom read/write
11	11	0XXX	0xF00–0xF7F	Standard read-only
11	11	10XX	0xF80–0xFBF	Standard read-only
11	11	11XX	0xFC0–0xFFF	Custom read-only

表 2.1:分配的RISC-V CSR地址范围

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

Table 2.2: Currently allocated RISC-V unprivileged CSR addresses.

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor 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.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.

Table 2.3: Currently allocated RISC-V supervisor-level CSR addresses.

Number	Privilege	Name	Description
Hypervisor Trap Setup			
0x600	HRW	hstatus	Hypervisor status register.
0x602	HRW	hedeleg	Hypervisor exception delegation register.
0x603	HRW	hideleg	Hypervisor interrupt delegation register.
0x604	HRW	hie	Hypervisor interrupt-enable register.
0x606	HRW	hcounteren	Hypervisor counter enable.
0x607	HRW	hgeie	Hypervisor guest external interrupt-enable register.
Hypervisor Trap Handling			
0x643	HRW	htval	Hypervisor bad guest physical address.
0x644	HRW	hip	Hypervisor interrupt pending.
0x645	HRW	hvip	Hypervisor virtual interrupt pending.
0x64A	HRW	htinst	Hypervisor trap instruction (transformed).
0xE12	HRO	hgeip	Hypervisor guest external interrupt pending.
Hypervisor Protection and Translation			
0x680	HRW	hgatp	Hypervisor guest address translation and protection.
Debug/Trace Registers			
0x6A8	HRW	hcontext	Hypervisor-mode context register.
Hypervisor Counter/Timer Virtualization Registers			
0x605	HRW	htimedelta	Delta for VS/VU-mode timer.
0x615	HRW	htimedeltah	Upper 32 bits of htimedelta, RV32 only.
Virtual Supervisor Registers			
0x200	HRW	vsstatus	Virtual supervisor status register.
0x204	HRW	vsie	Virtual supervisor interrupt-enable register.
0x205	HRW	vstvec	Virtual supervisor trap handler base address.
0x240	HRW	vsscratch	Virtual supervisor scratch register.
0x241	HRW	vsepc	Virtual supervisor exception program counter.
0x242	HRW	vscause	Virtual supervisor trap cause.
0x243	HRW	vstval	Virtual supervisor bad address or instruction.
0x244	HRW	vsip	Virtual supervisor interrupt pending.
0x280	HRW	vsatp	Virtual supervisor address translation and protection.

Table 2.4: Currently allocated RISC-V hypervisor and VS CSR addresses.

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
0x34A	MRW	mtinst	Machine trap instruction (transformed).
0x34B	MRW	mtval2	Machine bad guest physical address.
Machine Memory Protection			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
		⋮	
0x3AE	MRW	pmpcfg14	Physical memory protection configuration.
0x3AF	MRW	pmpcfg15	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3EF	MRW	pmpaddr63	Physical memory protection address register.

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

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

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

2.3 CSR Field Specifications

下面的定义和缩写都是用来指定CSRs中字段的行为.

Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

某些CSR的整个读/写字段都留作将来使用.软件应当忽视从这些字段中读出的值,当软件需要写入相同寄存器的不同字段的时候,应当保留之前字段中的值.为了前向兼容性,不提供这些字段的实现必须将这些字段硬连线到零.

为简化软件模型,在CSR中保留字段的任何向后兼容的定义都必须处理这么一种可能性,即使用非原子性的读-改-写序列对此CSR中其他字段进行更新的可能性.否则,原始CSR定义必须指定子字段只能原子性地更新,不过这可能需要一个两条指令的序列,清零和置位,且如果中间值不合法,就很可能引起问题.

Write/Read Only Legal Values (WLRL)

某些读/写CSR字段只为可能的位编码的子集指定了行为,其他位编码保留.软件不应当写合法值之外的数值到这样的字段,且除非最后一次写的是合法的值或此CSR自从被其他操作置为(例如,复位)合法值以来没再被写入过,否则软件不应当假设读取时会返回一个合法值.

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.

如果一条指令尝试写一个不支持的值到WLRL字段,允许但不规定实现需要引起非法指令exception.当对WLRL字段最后一次写入的是非法值时,实现可以在load时返回任意的位模式,不过此返回值应当取决于写入的非法值和此字段写入之前的值.

Write Any Values, Reads Legal Values (WARL)

某些读/写CSR字段只被定义了一个位编码的子集,但是允许写入任何值的同时保证load返回合法值.假想一下,若写入到此CSR不会引起任何副作用,则可以通过尝试写入一个期望的值然后查看此值是否被CSR保留以检测此CSR所支持的值的范围.在寄存器描述使用WARL标注此字段.

对WARL字段写入不支持的值,实现不会引起exception.当对此字段最后一次写入的是非法值时,实现可以返回任意的合法值,不过此返回值应当取决于写入的非法值和此字段写入之前的值.

2.4 CSR Width Modulation

如果一个CSR的宽度发生改变(例如,改变MXLEN或UXLEN,如3.1.6.2节所述),则新宽度的CSR的可写字段和位的值都由先前宽度的CSR决定,就像通过以下算法一样:

1. 先前宽度CSR的值复制到一个同宽度的临时寄存器中
2. 先前宽度CSR中的只读位清零.
3. 将临时寄存器宽度改变为新的宽度.如果新宽度W小于先前宽度,则保留临时寄存器的低W位,丢弃高位.若W大于先前宽度,则临时寄存器零拓展到W位宽.
4. 新宽度CSR中每个可写的字段从临时寄存器的相同位置取值.

改变CSR的宽度并不是读或写此CSR,因此不会引起任何副作用.

第 3 章

机器级ISA, 版本1.12

此章节描述了机器模式(M-mode)下可用的机器级操作,机器模式在RISC-V系统中的最高特权模式.M-mode用于低层次的访问硬件平台,且是复位后第一个进入的模式.M-mode还能用来实现对于硬件直接实现来说过于复杂的特性.RISC-V机器级ISA包含一个共同的核心,再根据其他支持的特权等级和硬件实现的细节进行拓展.

3.1 机器级CSRs

除了此节描述的机器级CSRs,M-mode的代码可以访问所有低特权等级的CSRs.

3.1.1 机器级ISA寄存器misa

misa CSR是一个WARL读/写寄存器,以报告hart所支持的ISA.此寄存器在所有实现中都必须是可读的.不过可以返回0值以表示未实现misa寄存器,这样就要求能通过独立的非标准机制检查CPU的功能.

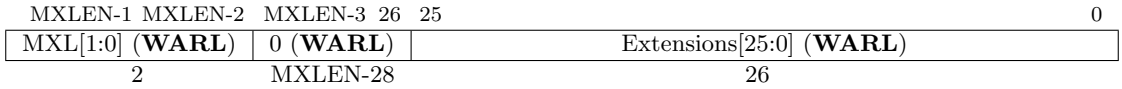


图 3.1: 机器ISA寄存器(misa).

MXL(Machine XLEN)字段编码原生基础整数ISA宽度,如表 3.1所示.MXL字段在支持多个基础ISA的实现上可能是可写的.在M-mode下有效的XLEN(MXLEN)通过设置MXL得到,如果misa是零则拥有固定值.在复位时,MXL域设置为所支持的ISA变体的最大宽度.

MXL	XLEN
1	32
2	64
3	128

表 3.1: misa中XML字段的编码

misa CSR位宽是MXLEN.如果读取misa获得非零值,则此值的MXL字段表示当前的MXLEN.如果写入misa导致MXLEN改变,则MXL移动到misa在新宽度下的最高2位.

基础宽度可以通过以下方式快速判断:对返回的misa值的符号进行判断,再左移一位和第二次符号判断.可以在不指定寄存器宽度(XLEN)的情况下使用汇编写出上述判断.因为基础位宽通过$XLEN = 2^{MXL+4}$给定.

即使misa是零也可以得到基础位宽,通过在一个寄存器中放置立即数4,再一次性左移31位,如果移位后此寄存器的结果是零,则机器是RV32,若两次移位后得到零,则是RV64,否则RV128.

在Extention字段中编码是否存在标准拓展,每个字母一个单独的位(第0位表示"A"拓展,第1位表示"B"拓展,...,第25位表示"Z"拓展).对于RV32I,RV64I,RB128I会置位低"I"位,对于RV32E会置位低"E"位.Extention字段是WARL字段,因此若实现允许修改所支持的ISA,则可以包含一些可写的位.复位后,Extention字段应当包含所支持的拓展的最大集合,如果I和E都存在则选择I.

当通过清零misa中的某一位禁用了一个标准拓展时,此拓展所定义或所修改的指令和CSRs恢复为定义的或保留的行为,就像它们并没有被实现.

RV128I基础ISA当前还没frozen,虽然此规范中大部分内容都将适用于RV128,但是当前版本文档还是只聚焦于RV32和RV64.

如果分别支持用户模式和管理员模式,则会置位"U"和"S"位.

如果存在任何非标准拓展,则置位"X"位.

---

misa CSR向机器模式代码暴露了CPU功能的基本目录.在机器模式下可以通过嗅探其他机器寄存器并在启动过程中检查系统中的ROM存储区以获取更多信息.

我们要求低特权等级执行环境调用以检查不同特权等级可用的特性而不是直接读取CPU寄存器.这样使得虚拟化层可以改变任意特权等级所观察到的ISA,且支持更丰富的命令接口而不加重硬件设计负担.

第"E"位为只读位.除非misa硬连线到零,否则"E"位一直是"I"位的补集.若一个实现同时支持RV32E和RV3II,则可以清零"I"位以选择RV32E.

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

表 3.1:misa的Extention字段编码.所有留作(reserved)将来使用的位在读取的时候必须返回0.

如果一个ISA特性x依赖于另一个ISA特性y,则试图使能特性x但不使能y会导致两个特性都未使能.例如,设置"F"=0和"D"=1会导致"F"和"D"都被清零.

实现可能对两个或多个misa字段的组合添加额外约束,在这种情况下,它们作为单个WARL字段发挥作用.若尝试写入不支持的组合,则会导致这些位被设置为某些支持的组合.

写misa可能增加IALIGN,例如取消对"C"拓展的使能.如果一条指令写misa导致IALIGN增加,且后续指令的地址非IALIGN-bit对齐,则阻止写misa,保持misa不变.

当软件使能了一个之前未使能的拓展,除非此拓展另有说明,否则所有与该拓展唯一关联的状态都是未指定的.(即标准不做规定)

3.1.2 Machine Vendor ID 寄存器 mvendorid

mvendorid CSR是32位只读寄存器,保存提供此核心的JEDEC制造商ID.在任何实现中此寄存区都必须是可读的,但是可以返回0值表示未实现此字段或者是非商业的实现.

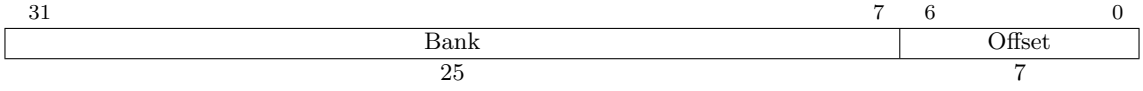


Figure 3.2: Vendor ID register (mvendorid).

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 CSR as 0x60a.

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

Previously the vendor ID was to be a number allocated by 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 寄存器 marchid

marchid CSR是MXLEN位的只读寄存器,编码了此hart的基础微架构.在所有实现中此寄存器都必须是可读的,但是可以返回0值以表示未实现此字段.通过mvendorid和marchid应当可以唯一的指明实现的hart微架构的类型.

图 3.3: Machine Architecture ID 寄存器(`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 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 寄存器 `mhartid`

mhart CSR是MXLEN位只读寄存器,其包含运行在核心上的硬件线程的整数ID.在任何实现中此寄存区都必须可读.并不要求Hart ID在多核系统中连续编号,但是最少需要一个hart的hart ID是0.在执行环境中的Hart ID必须是独一无二的.

Figure 3.5: Hart ID register (mhartid).

在特定情况下,我们必须保证只有一个hart运行某些代码(例如,复位),因此要求需要一个hart ID为0.

为了效率,系统实现者应当致力于减少系统中最大hart ID的大小.

3.1.6 Machine Status Registers (mstatus 和 mstatush)

mstatus是MXLEN位的读/写寄存器,对于RV32的格式如图 3.6所示,对于RV64的格式如图 3.7所示. mstatus寄存器持续跟踪和控制hart的当前操作状态.A restricted view of mstatus appears as the sstatus register in the S-level ISA.

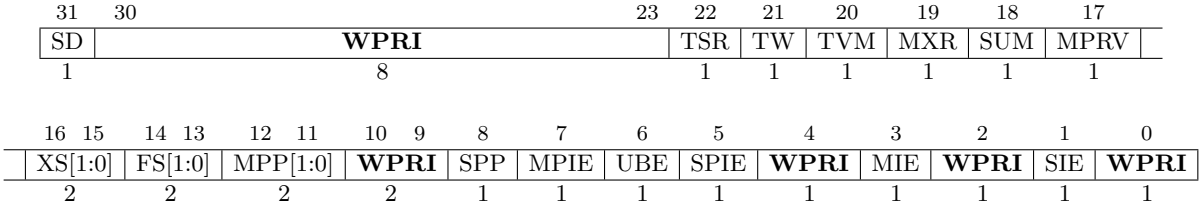


图 3.6: RV32的机器模式状态寄存器(mstatus)

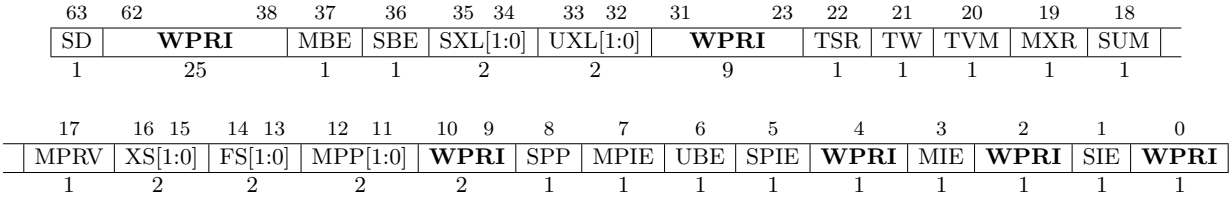


图 3.7: RV64的机器模式状态寄存器(mstatus)

mstatush寄存器是只用于RV32的32位读/写寄存器,格式如图 3.8所示.mstatush的30:4位通常包含和RV64 mstatus的62:36位相同的字段.mstatush中不包含SD,SXL和UXL字段.

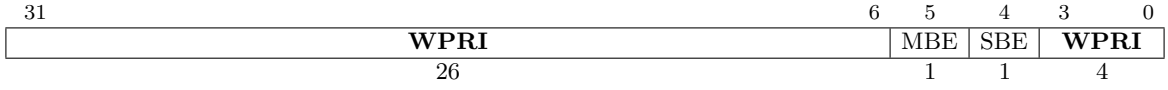


图 3.8: RV32I的额外的机器状态寄存器(mstatush)

3.1.6.1 Privilege and Global Interrupt-Enable Stack in mstatus register

对M-mode和S-mode分别提供了全局中断使能位MIE和SIE.
这些位主要用于保证在当前特权模式下的中断处理的原子性.

全局xIE都位于mstatus的低位部分,以保证它们能使用一条指令进行原子性的清零或置位.

当一个hart运行在特权模式x,则xIE=1时全局地使能了中断,xIE=0时全局地禁止了中断.无论是否为低特权模式w(w<x)设置全局wIE,低特权模式w的中断一直全局地禁止.无论是否为高特权模式y(y>x)设置全局yIE,高特权模式y的中断一直全局地使能.在移交控制权到低特权模式之前,更高特权等级的代码可以使用单独的中断控制来禁止所需的y的中断.

在移交控制权到低特权模式之前,高特权模式y可以禁止y的所有中断,不过这并不常见,因为其只留下了同步trap,不可屏蔽的中断或复位作为此hart重新获取控制权的方法.

为支持嵌套trap,每个可以响应中断的特权模式x都拥有两级中断使能位和特权模式栈.xPIE保存了trap之前活动的中断使能位的值,xPP保存先前的特权模式.xPP字段只能保存小于等于x的特权模式,所以MPP位宽为2,SPP位宽为1.当从y特权模式trap到x特权模式,xPIE设置为xIE的值;xIE置为0;xPP设置为y.

对于低特权模式来说,任何trap(同步或异步的)通常交给更高特权模式处理,且在进入高特权模式时关闭中断.高等级的trap handler要么处理此trap后使用保存在栈中的信息返回,要么没有马上返回被中断的上下文,则会在重新使能中断之前保存栈中的信息,所以每个栈只需要一条记录.

分别使用MRET或SRET指令从M-mode或S-mode的trap中返回.当执行xRET指令,假设xPP保存的值是y,xIE设置为xPIE;特权模式变为y;xPIE设置为1;xPP设置为所支持的最低特权模式(如果实现了U-mode则设置为U,否则M).如果xPP!=M,则将MPRV设置为0.

在xRET上设置xPP到所支持的最低特权模式有助于识别两级特权模式栈管理上的软件错误.

---

xPP字段是WARL字段,其只能保存特权模式x和任何实现了但等级低于x的特权等级.如果特权模式x未实现,则xPP必须硬连线到0.

M-mode的软件可以通过写某个模式到MPP然后将其读回以检测是否实现了此特权模式.
如果机器只提供了U和M模式,那么只需要单个硬件存储位就可以表示MPP中的00和11.

3.1.6.2 Base ISA Control in mstatus Register

对于RV64系统,SXL和UXL字段都是WARL字段,其控制S-mode和U-mode的XLEN的值.这两个字段的编码和misa的MXL字段相同(表 3.1).S-mode和U-mode的有效XLEN用分别术语SXLEN和UXLEN表示.

对于RV32系统,不存在SXL和UXL字段,且SXLEN=32,UXLEN=32.

对于RV64系统,如果不支持S-mode,则SXL硬连线到零,否则SXL为WARL字段,编码当前SXLEN的值.特别的,实现可能使得SXL变为只读字段,其值一直保证SXLEN=MXLEN.

对于RV64系统,如果不支持U-mode,则UXL硬连线到零,否则UXL为WARL字段,编码当前UXLEN的值.特别的,实现可能使得UXL变为只读字段,其值一直保证UXLEN=MXLEN或UXLEN=SXLEN.

不论何时在任意模式的XLEN设置到小于最大支持的XLEN时,所有的操作必须忽略源操作数寄存器中超过设置的XLEN的位,且必须符号拓展结果以填满目的寄存器最大支持的XLEN.

我们要求操作需要使用定义的值填满硬件寄存器是为了避免由实现来定义这种行为

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

如果MXLEN从32位变到更大的宽度,则mstatus字段SXL和UXL(如果没被限制到单个值)获得相应的支持的最大宽度的值(表 3.1),但对应的宽度不大于MXLEN.

3.1.6.3 Memory Privilege in mstatus Register

MPRV(Modify PRiVilege)位修改load和store执行时的特权等级.当MPRV=0,load和store保持正常,使用当前特权模式的传递和保护机制.当MPRV=1,按照MPP的特权等级对load和store的内存地址进行转换和保护,以及应用端序.指令地址转换和保护不受MPRV设置的影响.如果不支持用户模式,则MPRV应当硬连线到0.

若MRET或SRET所返回的特权等级低于M模式则要设置MPRV=0.

MXR(Make eXecutable Readable)位修改load对虚拟内存访问的特权.当MXR=0,仅当load的页被标注为可读(R=0,图4.17)时才能成功.当MXR=1,当load的页被标注为可读或可执行(R=1或X=1)时才能成功.

当虚拟内存未生效时,MXR不会造成影响.如果不支持S-mode,则MXR硬连线到0.

加入MPRV和MXR机制是为了提高M-mode的过程模拟硬件缺失的特性的效率,例如非对齐load和store.MPRV避免了在软件中执行地址转换.MXR允许load从标注为只可执行(execute-only)的页中加载指令.

当前特权模式和MPP中指定的特权模式可能设置了不同的XLEN.当MPRV=1,load和store的内存地址都视作当前的XLEN被设置为了MPP的XLEN,具体规则在3.1.6.2节.

SUM(permit Supervisor User Memory accesses,允许管理员访问用户内存)位修改了S模式的load和store访问虚拟内存的权限.当SUM=0,S模式若访问U模式可访问的页面(图4.17,U=1)将会引起错误.当SUM=1,允许上述访问.若基于页的虚拟内存未生效,则SUM不会造成影响.注意,虽然SUM在非S模式下一般不会造成影响,但是当MPRV=1且MPP=S时SUM会造成影响.如果不支持S模式或者satp.MODE硬连线到0,则SUM硬连线到0.

MXR和SUM机制只影响编码在页表项中的权限的解释.特别的,它们不影响因为PMA或PMP引起的访问错误例外(access-fault exception).

3.1.6.4 mstatus和statush寄存器中的端序控制

在mstatus和mstatush中的MBE,SBE,UBE位都是WARL字段,其控制内存访问的端序(不包括取指).取指一直是小端序.

MBE控制M-mode(假设mstatus.MPRV=0)的内存访问(非取指)是小端序(MBE=0)还是大端序(MBE=1).

如果不支持S-mode,则SBE硬连线到0.否则,SBE控制S-mode下显式的load和store内存访问是小端序(SBE=0)还是大端序(SBE=1).

如果不支持U-mode,则SBE硬连线到0.否则,UBE控制U-mode下显式的load和store内存访问是小端序(UBE=0)还是大端序(UBE=1).

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

只有在人为的情景下才有可能同时用大端序和小端序解释给定内存管理数据结构.实际上,只有在虚拟机上才会出现SBE改变,在这种情况下,新旧内存管理数据结构都不会以不同的端序重新解释.在这种情况下不需要额外的SFENCE.VMA,beyond what would ordinarily be required for a world switch.

如果支持S-mode,实现可能让SBE变为MBE的一个只读副本.如果支持U-mode,实现可能让UBE变为MBE或SBE的一个只读副本.

如果MBE,SBE,UBE字段都硬连线到0,则实现只支持小端序内存访问.如果MBE,SBE,UBE字段都硬连线到1,则实现只支持大端序内存访问(取指除外).

卷I定义hart的地址空间为连续地址空间上由2^{XLEN}字节组成的环形序列.地址和字节位置间的关系是固定的,不受端序影响的.端序影响的是多字节量和内存字节间的映射.

标准RISC-V ABI预计为纯小端序或纯大端序,而不使用混合端序.尽管如此,既然这么定义端序控制,那么就允许一个某端序的OS以相反的端序执行用户代码.还考虑了非标准使用的可能性,因此软件可以根据需要改变内存访问的端序.

RISC-V指令为统一的小端序,以从当前所设置的端序中解耦出指令编码,这对软件和硬件都有利.否则,举个例子,尽管在执行过程中端序一直在改变,但是RISC-V的汇编器或反汇编器需要一直知道.相反的,固定指令的端序,有时精心编写的软件即使在二进制的形式中也是端序不可知的,就像位置无关的代码.

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

3.1.6.5 mstatus寄存器中的虚拟化支持(Vvirtualization support)

TVM(Trap Virtual Memory)是WARL字段,其支持拦截supervisor的虚拟内存管理操作.当TVM=1,若执行在S-mode下,则读/写satp CSR或执行SFENCE.VMA指令会引起illegal instruction exception.当TVM=0,则在S-mode下允许上述操作.若不支持S-mode,则TVM硬连线到0.

TVM机制通过允许客操作系统执行在S-mode,而不是在U-mode进行虚拟化,提高了虚拟化效率.对于访问大多数SCR而言,此方法避免了trap.

trap对satp的访问和SFENCE.VMA指令为lazily populate shadow page table提供了必要的钩子.

TW(Timeout Wait)是WARL字段,其支持拦截WFI指令(3.3.3节).当TW=0,WFI指令在没被其他原因阻止的情况下可以运行在低特权模式.当TW=1,如果WFI是在地铁全模式下执行且没在实现特定的时间内完成,则WFI指令引起illegal instruction exception.时间限制可以是0,在这种情况下当TW=1时,WFI在低特权模式下总会引起illegal instruction exception.若没有比M低的特权模式,则TW硬连线到0.

---

trap WFI指令可以使得虚拟机切换到另外一个客操作系统,而不是在当前操作系统做无用的等待.

当实现了S-mode,若U-mode下执行的WFI未在实现特定的时间内完成则会引起illegal instruction exception.此标准的未来版本可能一个特性,以允许S-mode有选择的允许在U-mode下WFI.这样的特性仅当TW=0时才能激活.

TSR(Trap SRET)是WARL字段,其支持拦截管理员异常返回指令(SRET).当TSR=1,且执行在S-mode下,若尝试使用SRET则会引起illegal instruction exception.当TSR=0,则在S-mode允许上述操作.当不支持S-mode时,TSR硬连线到0.

对于不提供hypervisor拓展的实现而言,trap trap SRET可被用来模拟此拓展.

3.1.6.6 Extension Context Status in mstatus Register

RISC-V的一个主要目标就是支持大量的拓展,因此我们定义了标准接口,以在不改变特权模式代码(特别是supervisor-level OS)的情况下支持任意的用户模式状态拓展.

目前,V拓展是唯一定义了非浮点CSR和数据寄存器以外额外状态的标准拓展.

FS[1:0]WARL字段和XS[1:0]只读字段用通过分别跟踪当前浮点单元的状态和其他用户拓展,减少上下文保存和恢复的开销.FS字段编码浮点单元的状态,包括fcsr和浮点数据寄存器f0-f31,XS字段编码额外的用户模式拓展和相关状态的状态.上下文切换子程序可以检查这些字段以确定是否需要状态保存或者恢复.如果需要保存或者恢复,则通常需要额外的指令和CSRs来实现和优化流程.

预计大部分上下文切换将无需保存/恢复浮点单元和/或其他拓展,因此通过SD位提供快速的检查.

FS和XS使用如表3.3所示的相同状态编码,有四个可能的状态值:Off,Initial,Clean,Dirty

如果实现了F拓展,则FS字段不应当硬连线到0.

若F拓展和S-mode都未实现,则SF硬连线到0/若实现了S-mode但未实现F拓展,则FS可选是否硬连线到0.

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

表 3.3: FS[1:0] 和 XS[1:0] 状态字段

实现S-mode但无F拓展的情况下,可以将FS字段硬连线到0(但并不强制要求).某些实现可能选择不将FS硬连线到0,以通过不可见的trap到M-mode为S-mode和U-mode提供F拓展的模拟.

若没有用户拓展要求新的状态,XS字段硬连线到0.对于有新状态的额外拓展,其需要提供与XS状态等价的CSR字段.XS字段表示的是所有拓展状态的概括,如表3.3所示.

尽管单个的拓展可以使用与XS不同的编码,但是XS字段报告的是所有用户拓展状态字段的最大状态值.

SD为只读位,其总结了FS字段或XS字段是否存在dirty状态以要求保存拓展的用户上下文到内存.如果XS和FS都硬连线到0,则SD一直是0.

当某个拓展的状态设置为Off,任何尝试read或write相应状态的指令将会引起illegal instruction exception.当状态是Initial,对应的状态应当有一个初始的常数值.当状态是Clean,相应的状态可能与初始值不同,但是与上下文交换中存储的最后一次值匹配.

相应的特权代码在上下文保存时,其只需要将为Dirty的相应状态进行写出,然后将此拓展的状态改为Clean.在上下文恢复时,仅当状态为Clean时才需要从内存中加载上下文(在恢复时,状态永远不会是Dirty).若状态是Initial,则上下文恢复时必须将上下文设置到一个初始常数值以避免安全漏洞,不过这样可以避免访问内存.例如浮点寄存器可以初始化到立即数值0.

在保存上下文之前,FS和XS字段由特权代码读取当恢复用户上下文时,特权代码直接设置FS字段,通过写入拓展的状态寄存器间接的设置XS字段.无论是何特权模式,在指令执行的过程中状态字段也会改变.

用户模式ISA的拓展经常包含额外的用户模式状态,且该状态可以比基础整数寄存器多得多.此拓展可能仅被个别应用使用,或者是在单个应用中某个短的时间段内使用.为提高性能,用户模式拓展可以定义额外的指令以允许用户模式软件将此单元变为初始化状态或者直接关闭此单元.

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.

执行用户模式指令将一个单元禁用且将它置为Off状态后,若任何后续指令在重新将它启动之前尝试使用此单元将会引起illegal instruction exception.用户状态的指令在启动一个单元的时候必须保证合适的初始化此单元的状态,因为与此同时此单元可能被其他上下文使用.

改变FS的设置不会影响浮点寄存器状态的内容.特别是,设置FS=Off不会毁坏状态,设置FS=initial也不会清除内容.不过其他拓展在设置Off时可能不会保留状态.

即使浮点寄存器未被修改,实现也可以报告状态为dirty,以实现不精准的跟踪浮点寄存器状态.在某些实现上,一些不会改变浮点寄存器状态的指令可能会导致状态从Initial或Clean转变为Dirty.而在其他的实现中,可能根本不跟踪dirty的程度,在这种情况下,有效的FS状态只有Off和Dirty,且尝试设置FS到Initial或Clean都会导致其被设置为Dirty.

FS的定义并没有禁止由于错误的推测而导致设置FS为Dirty.某些平台可能会选择禁止推测性的写入FS,以关闭一个潜在的侧信道.

如果指令显式的或隐式的写浮点寄存器或fcsr,但是并未改变其内容,且FS=Initial或FS=Clean,则由实现定义FS是否转变为Dirty.

表3.1展示了FS或XS状态位的所有可能的状态转移.注意,标准浮点拓展不支持用户模式的取消配置或禁用/使能的指令.

初始化,保存,恢复拓展状态的标准特权指令通过将状态视为不透明的对象,以隔离特权代码与拓展状态的细节.

许多协处理器拓展只在有限的上下文中使用,以允许软件结束使用后安全的取消配置或禁用单元.这样降低了协处理器(有大量状态)的上下文开销.

XS字段概括了所有附加拓展状态,不过在拓展中可能有额外的微体系结构位以进一步的减少上下文保存和恢复的开销.

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

Table 3.4: FS and XS state transitions.

SD为只读位,当FS或XS编码为Dirty状态的时候SD置位(即SD=((FS==11)OR(XS==11))).特权代码可以快速的判断是否有整数寄存器集和PC以外的内容需要保存.

使用标准指令对浮点单元的状态进行初始化,保存和恢复.特权代码必须知道FLEN以为保存f寄存器提供合适的空间.

所有的特权模式共享FS和XS.在拥有超过一个特权模式的系统中,supervisor模式通常直接使用FS和XS来记录与supervisor级保存的上下文有关的状态.其他特权更高的活动的模式在它们相应的上下文版本中,必须更加保守的保存和恢复拓展状态.

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

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

mtvec是MXLEN位的WARL读/写寄存器,其保存trap向量的设置,由基础地址(BASE)和向量模式(MODE)组成.

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

必须实现mtvec寄存器,不过其可以包含硬连线的只读值.如果mtvec可写,由实现定义此寄存器可以保存的值的集合.BASE字段所保存的值必须四字节边界对齐,且MODE的设置可能对BASE附加额外的对齐约束.

对于trap向量基地址的实现我们给予了相当大的灵活性.一方面我们不希望对于低端的实现附加大量的状态位,但另外一方面我们希望为大的系统提供灵活性.

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.

表3.5展示了MODE字段的编码.当MODE=Direct,所有到机器模式的trap都会设置pc到BASE字段指定的地址.当MODE=Vectored,所有到机器模式的的同步exception会导致pc设置为BASE字段指定的地址,所有的中断导致pc设置为BASE字段指定的地址加上中断原因号(interrupt cause number)的四倍.例如,机器模式的定时器中断(38页的表3.6)会导致pc设置为BASE+0x1c.

当使能了vector中断,中断原因0(相应于用户模式软件中断)会将pc设置为与同步exception相同的位置.因为用户模式软件中断要么被禁用要么被委派到用户模式,所以在实践中一般不会引起此二义性.

一个实现可能对于不同的模式有着不同的对齐约束.特别是,MODE=Vectored可能相较于MODE=Direct拥有更严格的对齐约束.

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

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

3.1.8 Machine Trap Delegation Registers (medeleg and mideleg)

默认情况下,所有特权等级的trap都在机器模式处理(不过机器模式的handler可以使用MRET指令将trap重新定位到合适的特权等级,3.3.2节).为增加性能,实现可以在medeleg和mideleg中提供单独的读/写位,以指示相应的exception和interrupt直接被低特权等级处理.medeleg(machine exception delegation register)和mideleg(machine interrupt delegation register)都是MXLEN位的读/写寄存器.

拥有S-mode的系统中必须存在medeleg和mideleg寄存器,设置medeleg或mideleg中的位将会分派相应的trap(发生在S-mode或U-mode的trap)到S-mode的trap handler.对于不存在S-mode的系统,上述两个寄存器也不得存在.

在1.9.1和更早的版本中,在只有M-mode或M/U不存在N(原文写的N)的系统中,要求存在这些寄存器,不过都是硬连线到0.不过因为misa寄存器指示了是否存在这些寄存器,所以没用理由要求它们在这种情况下返回0值.

当trap已经委派给S-mode,则在scause寄存器中写入trap cause;sepc寄存器其中写入引发此trap的指令的虚拟地址.stval寄存器写入由exception特定的数据;mstatus的SPP字段写入引发trap时正在活动的特权模式;mstatus的SPIE字段写入引发trap时SIE字段的值;清零mstatus的SIE字段.不会写入mcause,mepc和mvtal以及mstatus中的MPP,MPIE字段.

实现可以选择只实现可委派的trap的子集,可以通过对medeleg或mideleg中的每一位写入1,再读回查看哪些位依然保持为1检查支持委派的位.

实现不得将medeleg中的任何位硬连线到1,即任何支持委派的同步trap也必须支持不被委派.同样的,实现不得将mideleg中相应于机器级中断的位硬连线到1(但是对于低级的中断可以这样做).

在1.11和更早的版本中禁止硬连线mideleg中的任何位到1.平台标准可能一直添加了这样的约束.

永远不会将高特权模式的trap转变为地铁全模式的trap.例如,如果M-mode委派illegal instruction exception到S-mode,且M-mode的软件执行了一条非法指令,这个trap会在M-mode处理,而不是委派到S-mode.相反的,trap可以平行的处理.使用相同的例子,如果M-mode委派illegal instruction exception到S-mode,且S-mode的软件执行了一条非法指令,则此trap在S-mode处理.

委派中断会导致在委派者特权等级上屏蔽此中断.例如,如果通过置位mideleg[5]将supervisor定时器中断(STI)委派给S-mode,则当执行在M-mode时,STI不会被处理.相反,若mideleg[5]清零,则不论当前模式,STI都会移交控制到M-mode处理.

图 3.10: Machine Exception Delegation Register `medeleg`.

medelg为38页的表3.6所示的每个同步exception分配了一个bit位置,每个bit位置的下标都等于返回到mcause寄存器中的值.(即,置位第8位以允许用户模式系统调用委派给低特权的trap handler).

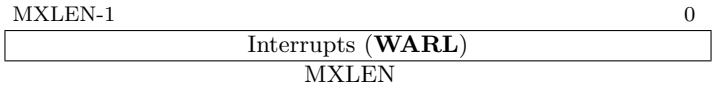


图 3.11: Machine Interrupt Delegation Register `mideleg`.

mideleg为每个中断保存trap委派位,位的布局与mip寄存器中的位相匹配(即STIP中断委派控制位于第5位).

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

mip寄存器是MXLEN位读/写寄存器,包含正在等待的中断的信息,mie寄存器是MXLEN位读/写寄存器,包含中断使能位.中断原因编号i(如3.1.15节报告到mcause的)相应于mip和mie中的第i位.15:0位仅用于标准终端原因,16位即以上的用来平台或自定义使用.

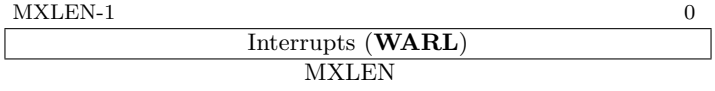


图 3.12: Machine Interrupt-Pending Register (`mip`).

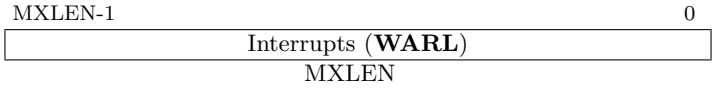


图 3.13: Machine Interrupt-Enable Register (`mie`).

如果以下全部成立,则中断i将会trap到M-mode(即导致权限模式变为M-mode):(a)要么当前特权模式是M-mode且mstatus中的MIE置位,或者当前特权模式低于M-mode;(b)mip和mie的第i位都置位;(c)如果存在mideleg寄存器,mideleg中的第i位没有置位.

从一个中断开始(或停止)在mip中等待,对上述中断条件的计算必须在限定时间内完成,对于执行xRET指令或显式的写入中断条件所依赖的CSR(包括mip,mie,mstatus和mideleg)也需要立即计算出上述中断条件.

对M-mode的中断要优先于任意到低特权模式的中断.

mip中的每个位都可能是可写的或是只读的.当mip中的第i位是可写的,一个正在等待的中断i可以通过写入0到此位以清零.如果中断i正在等待,但是mip中的第i位是只读的,则实现必须提供某些其他机制来清零此正在等待的中断.

若mie中的某一位相应的中断可以变为等待状态,则此位必须是可写的.mie中不可写的位必须硬连线到0.

寄存器mip和mie的标准部分(15:0位)的格式如图3.14和3.15所示.

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

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

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

图 3.15: Standard portion (bits 15:0) of mie.

机器级中断寄存器只处理少量根中断源,且有着固定的服务优先级,独立的外部中断控制器可以实现更复杂的优先策略,以将更大的中断集合以多路复用的形式送入机器级中断源.

在mip寄存器中不可见不可屏蔽的中断,因为当执行NMI(non-maskable interrupt) trap handler时才隐式的知道它的存在.

mip.MEIP和mip.MEIE位分别为机器等级外部中断的终端等待和中断使能位.MEIP是只读位,通过平台特定的中断控制器将其置位和清零.

mip.MTIP和mip.MTIE位分别为机器等级定时器中断的终端等待和中断使能位.MTIP是只读位,通过写内存映射的机器模式的定时器比较寄存器进行清零.

mip.MSIP和mip.MSIE位分别为机器等级定时器中断的终端等待和中断使能位.MSIP是只读位,通过访问内存映射的控制寄存器进行写入,远端hart使用此控制寄存器提供机器级处理期间中断.hart可以使用内存映射控制寄存器写自己的MSIP.

如果未实现supervisor模式,则mip的SEIP,STIP和SSIP以及mie的SSIE,SEIE,和STIE都硬连线到0.

如果实现了supervisor模式,则mip.SEIP和mie.SEIE是supervisor级外部中断的中断等待和中断使能位.SEIP是可写位,可以被M-mode的软件写入以指示S-mode有一个外部中断正在等待.此外,平台中断控制器可以产生supervisor级外部中断.基于软件可写的SEIP位和外部中断控制器信号的逻辑或的产生supervisor级外部中断等待.当通过CSR指令读取mip,返回到rd的SEIP位的值是上述逻辑或的值,来自中断控制器的信号不会被用来计算写入到软件可写的SEIP的值.只有软件可写的SEIP位参与到CSRRS或CSRRC指令的读-改-写序列.

SEIP位的行为是为了允许更高级特权层模拟外部中断且不丢失任何真实的外部中断.结果是CSR指令的此行为相较于常规的CSR访问行为稍有变化.

如果实现了supervisor模式,则mip.STIP和mie.STIE是supervisor级定时器中断的中断等待和中断使能位.STIP是可写位,可以被M-mode的软件写入以传递定时器中断给S-mode.

如果实现了supervisor模式,则mip.SSIP和mie.SSIE是supervisor级软件中断的中断等待和中断使能位.STIP是可写位.

通过实现特定的机制实现supervisor级的处理器间中断,例如,通过环境调用到SEE,最终对接收方hart的MSIP位产生一个机器模式的写入.

我们只允许hart对自己的SSIP进行直接写入,而不能是其他hart,因为其他hart可能被虚拟化了,或者被更高特权等级的取消调度了.因此我们依赖于调用SEE以提供处理器间中断.机器模式的hart不会被虚拟化,且能通过设置其他hart的MSIP位直接中断其他hart,通常对内存映射的控制寄存器(取决于平台标准)使用非缓存的I/O写.

多个同时发生且目标为M-mode的中断按照如下降序优先级处理:MEI,MSI,MTI,SEI,SSI,STI.

我们基于以下理由选择机器级中断的固定优先级排序规则:

对高特权模式发起的中断必须优先于低特权模式的中断以支持抢占.

在16位及以上的平台特定的机器级中断源按平台特定的有限期,不过通常会选择拥有最高的服务优先级以支持快速本地的向量中断.

因为外部中断通常由device产生,可能只需要很少的中断服务事件,所以外部中断优先内部中断(timer/software)进行处理.

软件中断先于内部定时器中断处理是因为内部定时器中断通常用于时间切片,在这种情况下时间精度不是很重要,而软件中断通常用于处理期间通信.当需要高精度定时时,可以避开软件中断,或者高精度定时器中断可以安排到不同的中断路径.软件中断位于mip的最低4位,因为这些位经常被软件改写,且可以使用只有5位立即数的单个CSR指令完成修改.

Restricted views of the mip and mie registers appear as the sip and sie registers for supervisor level. 如果通过设置mideleg寄存器中的位将中断委派给S-mode,则中断在sip寄存器中可见且可以使用sie寄存器屏蔽.否则sip和sie中相应的位表现为硬连线到0.

3.1.10 Hardware Performance Monitor

M-mode包含基础硬件性能监视工具.mcycle CSR计数处理器核(hart在其上运行)执行的时钟周期. minstret CSR记录此hart所退休的指令.在所有RV32和RV64系统上mcycle和minstret寄存器都是64-bit精度.

计数器寄存器在hart复位后可以是任意值,且可以写入给定值.任意的写CSR指令在其完成之后造成影响. 在同一个核心上的hart可能共享mcycle CSR,在这种情况下写入mcycle将会被其他hart所见. 平台应当提供一种机制指示哪些hart共享一个mcycle CSR.

硬件性能监视器包含29个额外的64-bit event计数器,mhpmcounter3-mhpmcounter31.事件选择器CSR(mhpmevent3-mhpmevent31)都是MXLEN-bit的WARL寄存器,其控制这哪些事件触发相应的计数器进行累加.这些事件的意义由平台定义,但是事件0定义为"无事件".所有计数器都应当实现,但是硬连线所有的计数器和相应的事件选择器也是合法的实现.

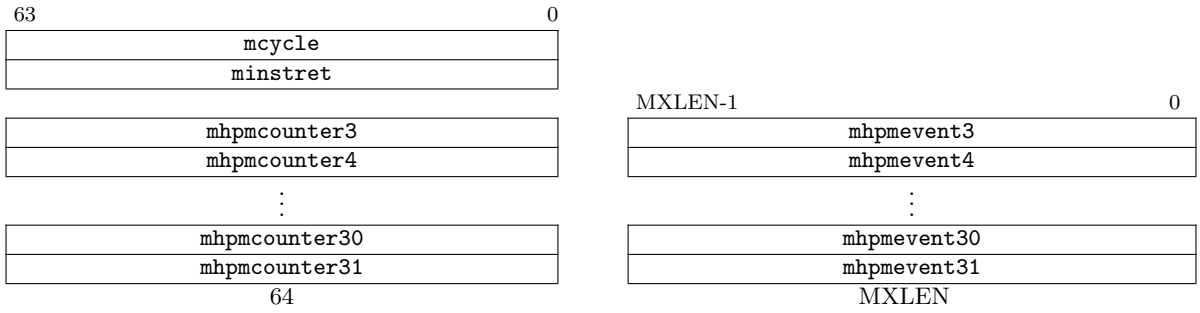


图 3.16: Hardware performance monitor counters.

mhpmcounter都是WARL寄存器,在RV32和RV64上都是64-bit精度.

此标准在未来会定义一种机制,在硬件性能监视计数器溢出后产生中断

在RV32中读取mcycle,minstret和mhpmcountern CSR将会返回CSR的低32位,读取mcycleh,minstreth和mhpmcounternh CSR返回相应计数器的高32位.

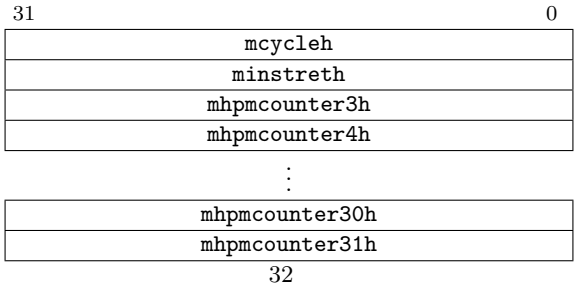


图 3.17: 硬件性能计数器的高32位, RV32 only.

3.1.11 Machine Counter-Enable Register (mcounteren)

计数器使能寄存器mcounteren是32位寄存器,其控制低一级的特权模式是否可用硬件性能监视计数器.

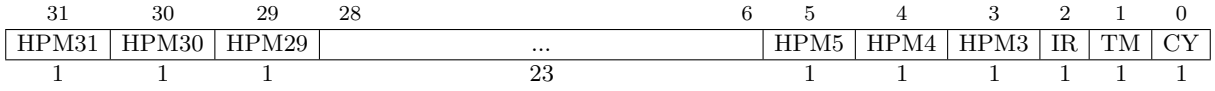


Figure 3.18: Counter-enable register (mcounteren).

此寄存器只控制是否能访问.对此寄存器的读写都不会影响底层的计数器,底层计数器在不可访问的时候依然保持累加.

当mcounteren寄存器中的CY,TM,IR或HPMn位被清零,执行在S-mode或U-mode下尝试读取相应的计数器(cycle,time,instrest或hpmcountern)会引起illegal instruction exception.当其中一些位被置位,则在下一个实现了的特权模式中允许访问相应的寄存器(如果实现S-mode则是S-mode,否则U-mode).

计数器使能位用最少的硬件支持两种常见的使用场景.对于不需要高性能定时器和计数器的系统,机器模式软件可以trap访问且在软件中实现所有的特性.对于需要高性能定时器和计数器但并不担心弄乱底层硬件计数器的系统,可以将计数器直接暴露给低级特权模式.

cycle,instret和hpmcountern CSRs分别是mcycle,minstret和mhpmcountern的只读影子寄存器.相似的,在RV32I中,cycleh,instreth和hpmcounternh CSRs分别是mcycleh,minstreth和mhpmcounternh的只读影子寄存器.在RV32I中,timeh CSR是内存映射的mtime寄存器的高32位的投影,time是mtime低32位的投影.

实现可以将读取time和timeh变为load内存映射的mtime寄存器,或者在M-mode软件中模拟此功能.

对于有U-mode的系统,必须实现mcounteren,不过所有的字段都是WARL的且可以硬连线到0,表明执行在低特权模式下读取相应的计数器会引起illegal instruction exception.对于不存在U-mode的系统,也不得存在mcounteren.

3.1.12 Machine Counter-Inhibit CSR (mcountinhibit)

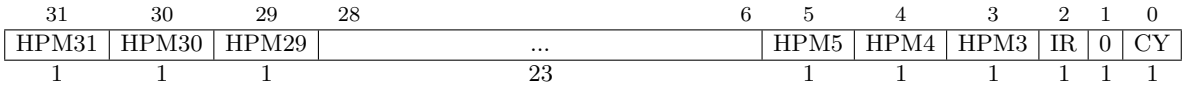


Figure 3.19: Counter-inhibit register mcountinhibit.

计数器禁止寄存器mcountinhibit是32位的WARL寄存器,其控制硬件性能监视计数器是否增长.设置此寄存器只控制相应的硬件寄存器是否增长;对计数器的访问不受此寄存器的影响.

当mcountinhibit寄存器中的CY,IR,或HPMn位被清零,则相应的计数器照常增长.当CY,IR或HPMn位被置位,则相应的计数器不再增长.

mcycle CSR可能被同一个核心中的不同hart所共享,在这种情况下,mcountinhibit.CY字段也被这些hart共享,因此对mcountinhibit.CY进行写入将对其他hart可见.

如果未实现mcountinhibit,则实现的行为就像此寄存器所有位都设置为0.

当不再需要cycle和instret计数器时,可以有选择的禁止它们以降低能量消耗.提供可以禁止所有计数器的CSR还可以进行原子性地采样(即在read之前先禁止增长).

因为time计数器可以在多核之间共享,所以不能使用mcountinhibit机制禁用它

3.1.13 Machine Scratch Register (mscratch)

mscratch寄存器是MXLEN位的读/写寄存器,专用于机器模式.通常其用来保存指向机器模式hart局部上下文空间的指针,且在进入M-mode的trap handler时与一个用户寄存器进行交换.

图 3.20: Machine-mode scratch register.

MIPS ISA分配了两个用户寄存器(k0/k1)供操作系统使用.尽管MIPS的策略提供了一个快速且简单的实现,不过其也减少了可用的用户寄存器,且没有拓展到更多的特权等级或嵌套trap.其要求两个寄存器在返回用户等级前要清零以避免潜在的安全漏洞以及提供确定的debug行为.

RISC-V ISA的用户ISA指在提供多种可能的特权系统环境,因此我们不想在用户级ISA中注入任何OS所依赖的特性.RISC-V的CSR交换指令可以快速的保存/恢复值到mscratch寄存器.与MIPS的设计不同,在用户上下文运行时,OS可以依赖于mscratch寄存器保存数值.

3.1.14 Machine Exception Program Counter (mepc)

mepc是MXLEN为的读/写寄存器,格式如图3.21所示.mepc的最低位(mepc[0]一直是零).对于只支持IALIGN=32的系统,最低两位(mepc[1:0])一直是零.

如果实现允许IALIGN可以为16或32(例如,通过改变misa CSR),那么不论何时IALIGN=32,mepc[1]位在读取时一直是屏蔽的,从而看上去是0.对于MRET指令,此位也是屏蔽的.虽然屏蔽了mepc[1],但其在IALIGN=32时依然是可写的.

mepc是WARL寄存器,因此必须能够保存所有有效的虚拟地址.其无需能够保存所有可能的无效地址.在写mepc之前,实现可能需要转换一个无效的地址到一些mepc能够保存的无效地址.

当地址转换没有生效时,虚拟地址和物理地址是一样的.因此mepc必须能表示的地址的集合包括物理地址的集合(可以用作合法pc或有效地址的物理地址).

当trap到M-mode时,mepc写入触发中断或产生exception的指令的虚拟地址.否则,实现永远不会写mepc,尽管软件可以显式地写mepc.

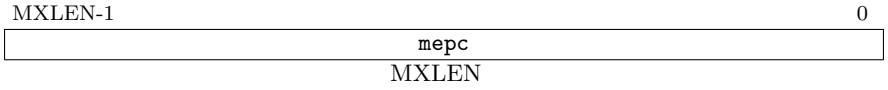


图 3.21: Machine exception program counter register.

3.1.15 Machine Cause Register (mcause)

mcause寄存器是MXLEN位的读-写寄存器,格式如图3.22所示.当trap到M-mode,mcause写入指示引起此trap的编码.否则,实现永远不会写mcause,尽管软件可以显式地写mcause.

如果是因中断引起此trap,则mcause的Interrupt位置位.Exception Code字段保存的编码标识最后的异常或中断.Table 3.6列出了可能的机器级异常编码.Exception Code是WLRL字段,因此其只保证能够保存所支持的例外编码.

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

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

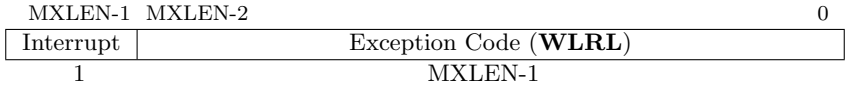


图 3.22: Machine Cause register `mcause`.

注意,load和load-reserved指令引起load exception,store,store conditional和AMO指令引起store/AMO exception.

可以对mcause寄存器值的符号使用单个分支指令分辨出中断和其他trap.左移指令可以移除中断位且放大例外编码,可以用来索引中断向量表.

我们并不从非法操作码例外中区分出特权指令例外.这样简化了体系结构且隐藏了实现所支持的高等级特权指令的细节.服务于此trap的特权等级可以实现一种策略,是否区分这些例外;如果需要区分,那么对于给定的opcode是否视为非法指令或是特权指令例外.

如果一条指令引起了多个同步exception,表3.7的降序优先排序支持了会采用与报告到mcause的例外.

Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12	Instruction page fault
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8, 9, 11	Environment call
	3	Environment break
	3	Load/Store/AMO address breakpoint
可选地,这些可能拥有最低的优先级	6	Store/AMO address misaligned
	4	Load address misaligned
	15	Store/AMO page fault
	13	Load page fault
	7	Store/AMO access fault
	5	Load access fault

表 3.7: 按降序优先排序的同步例外优先级.

注意load/store/AMO地址非对齐例外可能相较于load/store/AMO页错误和地址错误例外拥有高的或低的优先级

---

load/store/AMO地址非对齐和页错误例外的相对优先级由实现定义,这样可以灵活地适配两个设计点.对于永远不支持非对齐访问的实现,可以无条件的引起非对齐地址例外,而无需执行地址转换或保护检查.对于支持部分物理地址非对齐访问的实现,则在确认此非对齐访问是否进行之前必须执行地址转换和检查,在这种情况下引起页错误例外或访问错误例外更合适.

指令地址断点例外与数据地址断点(又名watchpoint)例外和environment break(因EBREAK指令引起)例外有相同的原因编码但是有不同的优先级.

指令地址非对齐例外都是因为有着非对齐的目标的控制流指令引起的,而不是被取指的行为引起的.因此,这些例外相较于其他指令地址例外拥有更低的优先级.

3.1.16 Machine Trap Value Register (mtval)

mtval寄存器是MXLEN位的读-写寄存器,格式如图3.23所示.当trap到M-mode,mtval要么设置到零要么写入例外特定的信息,以辅助软件处理trap.否则实现永远不会写mtval,尽管软件可以显式地写mtval.硬件平台指定哪个例外必须设置mtval以提供信息以及哪些例外会无条件的设置mtval为零.如果硬件平台指定没有例外会写入mtval,则mtval硬连线到0.

如果mtval没有硬连线到0,那么当取指,load或store引起地址非对齐,访问错误,或页错误例外时,mtval写入引发此错误的指令的虚拟地址.On an illegal instruction trap, mtval may be written with the first XLEN or ILEN bits of the faulting instruction as described below. 对于其他trap,mtval设置为零,不过将来的标准可能会为这些trap重新定义mtval的设置.

---

mtval寄存器取代了之前标准的mbadaddr除了提供bad address,此寄存器现在可以提供引发了非法指令trap的bad instruction(且在将来可能用作返回其他信息).返回指令位可以加速指令模拟以及移除了模拟非法指令时可能出现的竞争.

当基于分页的虚拟内存使能时,mtval写入出错的虚拟地址(即使是物理内存访问错误例外也写入虚拟地址).对大部分实现而言,这样的设计减少了数据路径开销,尤其是对于有硬件page-table walker的实现而言.

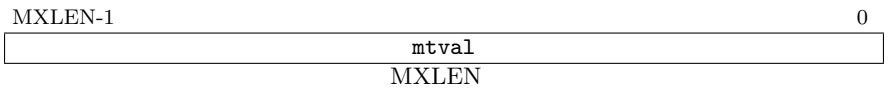


图 3.23: Machine Trap Value register.

对于引起访问错误或页错误的非对齐load和store,mtval将会保存引起此错误的访问部分的虚拟地址(除非mtval硬连线到0).

在拥有可变长指令的系统上,对于指令访问错误或页错误例外,mtval将会保存引起此错误的指令部分的虚拟地址(除非mtval硬连线到0).mepc将会将会指向此指令的开头地址.

对于非法指令例外,mtval可以可选地用于返回错误的指令位(mepc指向此错误指令在内存中的地址).

如果不提供此特性,则在非法指令错误时将mvtal设置为0.

如果提供此特性,则在非法指令trap后,mtval将包含以下最短的那个:

1. 发生错误的指令
2. 发生错误的指令的前ILEN位
3 发生错误的指令的前XLEN位

加载到mtval的值向右对齐,未用到的高位清零.

在mtval中捕获错误指令有以下的好处:降低了指令模拟的开销,如果是非对齐的指令则避免了潜在的多次部分指令加载,以及当加载指令到数据寄存器时所产生的的缓存缺失或低速的不可缓存的访问.以及如果其他代理操作了指令内存则存在原子性的问题,这可能发生在动态翻译系统中.

一个要求是在trap之前将整个指令(或至少是前XLEN位)写入到mtval中.因为实现在解码之前通常会取出整条指令,所以此要求应当不会对实现有过多的约束,而且这样可以避免复杂的软件handler.

mtval中的零值表示要么不支持此特性,要么是取到了非法的零指令.可以从指令内存中加载由mepc所指向的数据分辨以上两种情况(或者在runtime之前查询系统配置信息以安装合适的trap handler)

如果mtval未硬连线到零,则它是WARL寄存器,必须能够保存所有合法的虚拟地址和零值.其无需有能力保存所有可能的非法地址.在写mtval之前,实现可能将一个非法地址转变为mtval有能力保存的非法地址.如果实现了返回错误指令位的特性,mtval必须有能力保存小于2^N的所有数值(N是XLEN和ILEN中较小的那个).

3.2 Machine-Level Memory-Mapped Registers

3.2.1 Machine Timer Registers (mtime and mtimecmp)

平台提供实时计数器,以内存映射的机器模式读-写寄存器mtime暴露出来.mtime必须按照常数速率增加,且平台必须提供一种机制检测time的时间基数.mtime寄存器在溢出时会绕回.

对于所有RV32和RV64系统,mtime寄存器都拥有64位精度.平台提供一个64位的内存映射的机器模式定时器比较寄存器(mtimecmp).当mtime包含的值大于等于(按无符号数)mtimecmp的值将会产生一个机器定时器中断等待.直到mtimecmp大于(通常是因为写mtimecmp)mtime时中断才会消失.如果中断有效且mie寄存器的MTIE位置位,中断才会被处理.

becomes pending whenever `mtime` contains a value greater than or equal to `mtimecmp`, treating the values as unsigned integers. The interrupt remains posted until `mtimecmp` becomes greater than `mtime` (typically as a result of writing `mtimecmp`). The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

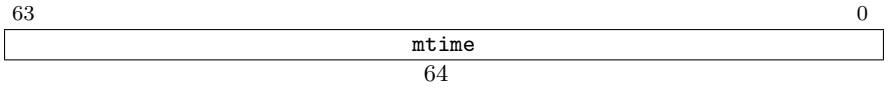


图 3.24: Machine time register (memory-mapped control register).

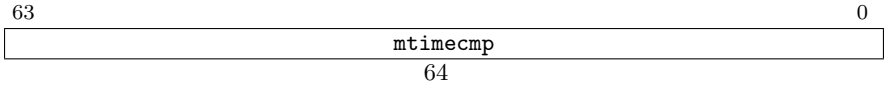


图 3.25: Machine time compare register (memory-mapped control register).

将定时器定义为挂钟时间而不是周期计数器以支持可变时钟频率的现代处理器通过动态电压和频率变化节约能量.

准确的实时时钟(real-time clock)相对较昂贵,且在系统中其他部分都下电后可能依然要允许,所以在系统中通常只有一个RTC,位于处理器的一个不同的频率/电压域.因此RTC必须被系统中的所有hart共享,而且访问RTC将会潜在地导致电平转换和跨时钟域.因此将mtime作为内存映射的寄存器暴露出来更为自然(相较于作为CSR).

低特权等级没有自己的timecmp寄存器.取而代之的是机器模式的软件通过复用定时器中断为hart提供任意数量的虚拟定时器.

简单的固定频率的系统可以将单个时钟用作周期计数器和挂钟时间.

保证对mtime和mtimecmp的写入最终会反映到MTIP,但是并不一定立即发生.

如果中断handler在增加了mtimecmp后立即返回,则可能产生假定时器中断,因为MTIP在此器件可能还未下降.所有软件应当假设可能会发生这种事情,不过也应当假设这种事件很难发生.与轮询MTIP直到其下降为止,接受偶尔发生的虚假定时器中断往往更为高效.

在RV32中,对mtimecmp的内存映射写只会修改此寄存器的32位.下述代码序列使用比较数的中间值避免了写入64位的mtimecmp时所产生的的虚假定时器中断.

对于RV64,支持对mtime和mtimecmp寄存器的自然对齐的64位内存访问,且访问是原子的.

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

图 3.26: 在RV32中设置64位的定时器比较数的示例代码.假设是小端内存系统,且此寄存器位于强排序I/O域.保存-1到mtimecmp的低有效位防止了mtimecmp临时小于mtime.

3.3 Machine-Mode Privileged Instructions

3.3.1 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

ECALL指令用于对支持执行环境产生请求.在U-mode,S-mode或M-mode下执行ECALL分别产生environment-call-from-U-mode exception, environment-call-from-S-mode exception或environment-call-from-M-mode exception,除此之外没有其他操作.

因为ECALL对每个源特权模式产生不同的exception,所以环境调用例外可以有选择性的委派出去.对于Unix-like操作系统,一个常见的例子是将environment-call-from-U-mode exception委派给S-mode,而不是委派其他的exception.

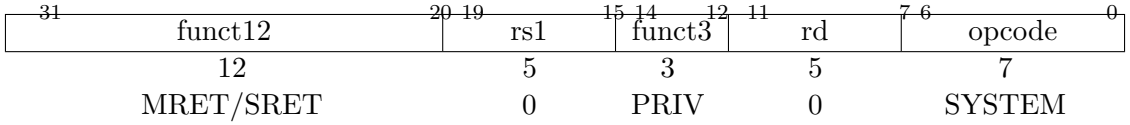
调试器使用EBRAK指令将控制权移交回debug环境.其产生breakpoint exception,除此之外没有其他操作.

如此标准卷I的用于压缩指令的"C"标准拓展所述,C.BREAK指令与BREAK指令执行相同的操作

ECALL和EBREAK使得相应特权模式x的xepc寄存器设置到ECALL或EBREAK指令所在的地址.因为ECALL和EBREAK产生的是同步例外,所以不应当视作它们退休,因此也不应当增加minstret CSR.

3.3.2 Trap-Return Instructions

在PRIV minor opcode中编码了从trap中返回的指令.



在处理完trap后需要返回,所以为每个特权等级提供了独立的trap返回指令,MRET和SRET.MRET如果支持supervisor模式,则必须提供SRET,若不支持supervisor模式,则对SRET指令引起非法指令异常.当mstatus中的TSR=1时,SRET应当如3.1.6.5节所述引起非法指令异常.特权模式x或更高特权模式可以执行xRET指令,执行一个低特权模式的xRET会pop此相关的低特权模式的中断使能和特权模式栈.除了操作3.1.6.1节所描述的特权栈,xRET还将pc设置为保存在xepc寄存器中的值.

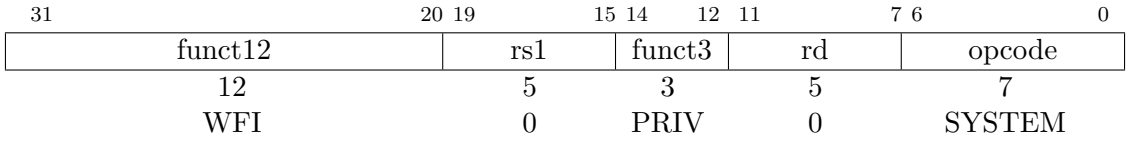
早先只有单个ERET指令(早先也称为SRET).为支持用户级中断的加入,我们需要添加一个单独的URET指令以继续允许使用ERET指令实现操作系统代码的经典虚拟化.It then became more orthogonal to support a different xRET instruction per privilege level.

如果支持A拓展,xRET指令可以但不强制清空任何在外LR address reservation.如果有需要,在执行xRET之前,trap handler应当显式的清空(例如,使用一个dummy SC)reservation.

如果xRET指令总会清空LR reservation,那么就不可使用调试器在LR/SC序列之间进行单步调试.

3.3.3 Wait for Interrupt

等待中断指令(Wait for Interrupt instruction ,WFI)示意实现可以阻塞当前hart直到出现需要服务的中断.执行WFI指令也可以用来示意硬件平台将合适的中断优先分配到此hart.所有特权的模式都可用WFI,可选地对U-mode可用.当mstatus的TW=1(如3.1.6.5节所述)此指令可能引起非法指令异常.



当hart阻塞时出现了可用的中断,此中断例外发生在接下来的那条指令上,即在trap handler中恢复执行且mepc = pc +4.

因为中断例外和trap发生在后续那条指令上,所以从trap handler只需要执行一个简单的返回就可以从WFI指令后的代码继续执行.

WFI指令的目的是给实现提供暗示,因此一个合法的实现可以简单地将WFI实现为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.

当禁用中断时也能执行WFI指令.WFI的操作必须不受mstatus中的全局中断位(MIE和SIE)以及委派寄存器mideleg的影响(即如果一个局部有效的中断正在等待,则hart必须恢复执行,即使此中断已经被委派给低特权的模式),不过WFI的操作需要遵守独立的中断使能(例如MTIE)(即如果中断正在等待但是没有单独地使能,实现应当避免恢复hart的执行).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.

如果导致hart继续执行的事件不会引起中断处理,则hart从pc+4继续执行,软件必须决定需要采取哪些措施,包括在没有可执行的事件的情况下循环回去重复执行WFI.

通过在禁用中断时允许唤醒操作,因为当前上下文在WFI执行之前已被保存或者丢弃,所以可以调用不需要保存当前上下文的中断处理程序的其他入口点.

因为实现可以自由的将WFI实现为NOP,所以WFI之后的软件代码必须显式的检查任何相关的正在等待但是被禁用的中断,且应当在没有检查到合适的中断的时候回头重新执行WFI.可以查询mip或sip寄存器以检查机器模式或管理模式的中断是否存在.

WFI的操作不受委派寄存器的设置的影响.

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.

未来的拓展可能会有"wait-for-event",用来等待内存发生改变或者等待信息到达.

3.4 Reset

复位后,hart的特权模式设置为M.mstatud的MIE和MPRV字段复位到0.如果支持小端内存访问,则mstatus/mstatush的MBE字段复位到0.misa寄存器复位到所支持拓展的最大集合以及所支持的最宽的MXLEN,如3.1.1节所述.pc设置到实现所定义的复位向量.mcause寄存器设置为某个值,此值指示此次复位的原因.可写的PMP寄存器的A和L字段福我到0,除非平台为某些PMP寄存器的A和L字段指示了不同的复位值.hart的其他状态都是UNSPECIFIED.

复位后的mcause的值是由特定实现解释的,不过如果不区分不同的复位调剂则应当返回0到mcause.对于区分不同复位条件的实现,应当只使用0来表示最完全的复位.

某些设计可能有多种复位原因(例如,上电复位,外部硬复位,欠压检查,看门口定时器过期,睡眠模式唤醒),机器模式软件和调试器可能希望可以区分这些原因.

mcause的复位值和mcause的同步例外值可能混淆.不过应当不会出现二义性,因为复位后的pc通常与trap之后的pc不同.

3.5 Non-Maskable Interrupts

不可屏蔽中断(Non-maskable interrupts,NMIs)仅用于硬件错误的情况,不论hart的中断使能位是何状态,立即引起跳转,跳转运行在M-mode下由实现定义的NMI向量.mepc寄存器写入被中断的指令的虚拟地址,mcause设置为指示此次的源头是NMI的值.The NMI can thus overwrite state in an active machine-mode interrupt handler.

写入到mcause的值由实现定义.mcause的Interrupt位应当置位以指示此次是中断引起的.Exception Code 0留作表示"未知原因",实现若不通过mcause寄存器区分NMIs的源头,则应当在Exception Code中返回0.

不像复位,NMIs不会复位处理器状态,从而实现诊断,报告和可能的硬件错误控制.

3.6 Physical Memory Attributes

完整系统的物理内存映射包括各种地址范围,某些相应于内存区域,某些相应于内存映射的控制寄存器,以及某些地址空间上的空洞.某些内存区域可能不支持读,写,或执行,而某些可能不支持subword或subblock访问,某些可能不支持原子操作,某些可能不支持缓存一致性或可能拥有不同的内存模型.相似地,内存映射的控制寄存器对其所支持的访问宽度,对原子操作的支持,以及对读写访问是否有相关的副作用都有所不同.在RISC-V系统中,机器的物理地址空间的每个区域的性质和能力被称为物理内存属性(physical memory attitudes,PMAs).此节描述RISC-V PMA术语以及RISC-V系统如何实现和检查PMAs.

PMAs是底层硬件的内在属性,而且在系统运行期间很少改变.不像3.7节所描述的物理内存保护值,PMAs不因执行上下文而变化.某些内存区域的PMAs在芯片设计时便已确定-例如,一个on-chip ROM.其他的则在板子设计时确定,例如取决于有哪些芯片被连接到了片外总线.片外总线可能支持器件在每个上电周期(冷拔插)有所改变或者在系统运行时动态的改变(热拔插).某些器件可能支持运行时配置以支持不同的用途,这就意味着PMA有所不同-例如,一个片内scrachpad RAM可以一个终端应用内的一个核心私有地缓存,或者在另一个终端应用中作为共享的非缓存内存访问.

大部分系统会要求在硬件中至少会动态地检查一部分PMAs,此检查位于物理地址已知后的执行流水线中,因为某些操作并不是所有物理内存地址都支持的,而且某些操作需要知道可配置的PMA属性的当前设置.虽然许多其他体系结构在虚拟内存页表中指定某些PMAs且使用TLB将这些属性告知流水线,但是这种方法在虚拟化层注入了平台特定的信息,而且除非为每个物理内存区域的每个页表项进行了正确的初始化,否则可能引起系统错误.此外,对于为物理地址空间指定属性来说,页面大小可能并不合适,因此导致地址空间碎片化以及对昂贵的TLB项低效地使用.

对于RISC-V,我们将标准和PMAs的检查拆分到一个单独的硬件结构,PMA checker.在大部分情况下,每个物理地址区域的属性在系统设计时都是已知的,并且可以硬连线到PMA checker.对于运行时可配置的属性,则可以提供平台特定的内存映射的控制寄存器,以便为平台上的每个区域使用合适的粒度指定这些属性(例如一个片内SRAM可以灵活地变为可缓存的或不可缓存的).对于任何对物理内存的访问,都需要检查PMAs,包括已经经过了虚拟地址转物理地址的内存访问.为帮助系统调试,我们强烈建议,如果可能的话RISC-V处理器要精确地捕获未通过PMA检查的物理内存访问.被精确捕获的PMA违例表现为指令,load或store访问错误例外,有别于虚拟内存页错误例外.不过精确的PMA trap并不总是可能的,例如在使用访问失败作为设备发现机制一部分的传统总线中进行设备探测时.在这种情况下,从器件返回的错误响应将会报告为非精确的总线错误中断.

对软件来说,PMAs必须是可读的,以便正确地访问器件,或者正确的配置其他需要访问内存的硬件,比如DMA引擎. 因为PMAs与给定的物理平台布局紧密相关,所以许多细节都是平台特定的,软件可以通过PMAs了解硬件的物理内存属性.某些器件,特别是传统总线,不支持发现PMAs,因此如果尝试不支持的访问会给出错误响应或者超时.通常,平台特定的代码会提取出PMAs然后通过某些标准表示将这些信息告知高层级低特权的软件.

对于支持动态重配置PMAs的平台,会提供一个接口,通过此接口对可以正确重配置平台的机器模式代码传递请求以设置这些属性.例如,切换某些内存区域的可缓存性可能需要平台特定的操作参与,比如只在机器模式可用的cache flush.

3.6.1 Main Memory versus I/O versus Vacant Regions

对于一个给定的内存地址范围,其最重要的属性是是否为常规主存或为I/O器件或为空洞.常规主存需要具有以下指定的许多属性,而I/O设备可以具有更广泛的属性范围.不适用归类到常规主存的内存区域都归类为I/O区域,例如device scratchpad RAMs.空洞区域也归类为I/O区域,不过其属性指定此区域不支持任何访问.

3.6.2 Supported Access Type PMAs

访问类型指定了支持何种访问宽度(从8-bit字节到长的多字突发传输)以及对每个访问宽度是否支持非对齐访问.

尽管运行在RISC-V hart上的软件不能直接地产生产生到内存的突然传输,但是软件可能不得不编程DMA引擎以访问I/O器件,因此可能需要了解所支持的访问大小.

主内存区域一直支持附属器件发起的所有访问宽度的读和写,而且可以指定是否支持取指.

某些平台可能指定所有的主内存都支持取指.其他一些平台可能禁止从某些主内存区域中取指.

在一些情况下,访问主内存的处理器或器件可能支持其他类型宽度,不过必须能与主内存所支持的类型一起工作.

I/O区域可以指定支持哪些数据宽度的读,写或执行访问的组合.

对于基于分页的虚拟内存的系统,I/O和内存区域可以指定支持哪些硬件页表读和硬件页表写的组合.

---

*Unix-like*操作系统通常要求所有可缓存的主内存都得支持pagetable walk.

3.6.3 Atomicity PMAs

原子性PMAs描述在此地址区域中支持哪些原子指令.对原子性指令的支持分类两类:LR/SC和AMOs.

某些平台可能要求所有可缓存的主内存需要支持处理器请求的所有原子操作.

3.6.3.1 AMO PMA

在AMO PMA内有四个支持等级:AMONone,AMOSwap,AMOLogical和AMOArithmetic. AMONone表示不支持任何AMO操作. AMOSwap表示在此地址范围只支持amoswap指令.AMOLogical表示支持amoswap指令和所有的逻辑AMO指令(amoand, amoor,amoxor).AMOArithmetic表示支持所有的RISC-V AMO指令.对于每个支持等级,对于给定宽度的自然对齐的AMO,如果底层内存区域支持此宽度,则支持此AMO.主内存和I/O区域可能只支持处理器所支持的原子操作的一个子集(可能是空集).

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

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

如果可能的话,我们建议对于I/O区域至少支持AMOLogical.

3.6.3.2 Reservability PMA

对于LR/SC,有三个支持等级,表示reservability和eventuality属性的组合:RsrvNone, RsrvNonEventual,和RsrvEventual.RsvrNone表示不支持任何LR/SC操作(此区域non-reservable). RsrvNoneEventual表示支持此操作(此区域reservable),不过没有最终成功保证(非特权ISA标准中有所描述).RsrvEventual表示支持此操作且提供最终成功保证.

如果可能的话,我们建议对主内存区域提供RsrvEventual支持.大部分I/O区域都不支持LR/SC访问,因为它们大都建立在缓存一致性方案之上,不过某些可能支持RsrvNonEventual或RsrvEventual.

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

3.6.3.3 Alignment

支持对齐LR/SC或对齐AMOs的内存区域也可能对某些地址和访问宽度支持非对齐的LR/SC或非对齐的AMOs.如果对于一个给定的地址和访问宽度,一个非对齐的LR/SC或AMO产生了address-misaligned exception,那么所有使用此地址和访问宽度的load,store,LRs/SCs和AMOs必须产生address-misaligned exception.

标准"A"拓展不支持非对齐AMOs或非对齐LR/SC pairs."Zam"拓展提供了对非对齐AMOs的支持.当前并未标准化对非对齐LR/SC序列的支持,因此对非对齐地址的LR和SC必须引起例外.

如果非对齐的AMOs会引起address-misaligned exception,则强制要求非对齐的load和store也需要引起address-misaligned exception,从而允许在M-mode的trap handler中模拟非对齐AMOs.此handler通过获取一个全局互斥锁以及在临界区内模拟这次访问以保证原子性.如果非对齐load和store的handler使用相同的互斥锁,则所有使用相同字大小到给定地址的访问都是互相原子的.

实现对于某些非对齐访问可能引起access-fault exception而不是address-misaligned exception,以指示此条指令不应当被trap handler所模拟.对于一个给定的地址和访问宽度,如果所有非对齐的LRs/SCs和AMOs都产生access-fault exception,则使用此地址和访问宽度的常规的非对齐的load和store无需原子性地执行.

3.6.4 Memory-Ordering PMAs

为了按FENCE指令和原子指令的排序位进行排序,将地址空间区域分类为主存储器区域和I/O区域.

一个hart对主存区域的访问不仅能被其他hart所观察到,而且能被其他有能力在主存系统中发起请求的其他器件所观察到(例如DMA引起).一致性的主存区域要么有RVWMO内存模型要么有RVTSO内存模型.非一致性的主存区域拥有实现定义的内存模型.

一个hart对I/O区域的访问不仅能被其他hart和总线总控设备所观察到,而且能被目标从I/O器件所观察到,且对I/O区域的访问要么是relaxed ordering要么是strong ordering.如此标准的卷I A.4.2节所述,其他hart和总线总控器件所观察到的对relaxed ordering I/O区域的访问类似于访问RVWMO内存区域的顺序.相反,其他hart和总线总控器件以程序序观察到对strong ordering I/O区域的访问.

每个强排序的I/O区域指定一个编号了的排序channel,通过这种方法可以在不同的I/O区域提供排序保证.channel 0只用来表示点对点强排序,即hart对单个相关的I/O区域的访问是强排序的.

channel 1用与跨所有I/O区域提供全局强排序.一个hart对任意关联channel 1的I/O区域的任意访问只会以程序序被所有其他hart和I/O器件所观察到,包括including relative to accesses made by that hart to relaxed I/O regions or strongly ordered I/O regions with different channel numbers.换句话说,任何对channel 1区域的访问都等价于在此指令前后执行了fence io,io指令.

其他更大的channel号用于排序某hart对相同channel号的不同区域之间的访问.

系统可以支持对每个内存区域的排序属性进行动态配置.

强排序可以用来提高对传统器件驱动代码的兼容性,或者在已知实现不会重排序访问时,与插入显式的排序指令相比,强排序可以提高性能.

局部强排序(channel 0)是强排序的默认形式,因为如果在hart和I/O器件之间只有单个有序通信路径,则可以直接提供channel 0.

通常,不同的强排序I/O区域如果共享相同的互联路径而且此路径不会重排序请求,则可以共享相同的排序channel而不需要额外的硬件.

3.6.5 Coherence and Cacheability PMAs

Coherence是为单个物理地址所定义的属性,其表示某个代理对此地址的写最终都会被在系统中的其他代理所见.Coherence不应该与系统的内存一致性模型弄混,后者定义了在给定的整个内存系统的读写历史下,内存读取可以返回哪些值.在RISC-V平台中,考虑到软件复杂性,性能,和功耗的影响,不鼓励使用硬件非一致性区域,

一个内存区域的可缓存性不应当此区域的软件视角,影响软件视角的是PMAs中的其他差异,比如主存与I/O分类,内存序,支持的访问和原子操作以及coherence.出于此考虑,我们将可缓存性视作平台级设置,且只被机器模式软件管理.

如果平台对一个内存区域支持可配置的可缓存性设置,平台特定的机器模式子程序将会改变此设置,而且在有必要的情况下冲刷cache,所以只在可缓存性设置改变期间,此系统是非一致性的.上述短暂的状态不应对低特权等级可见.

我们将RISC-V cache分为三类:master-private,shared,slave-private.master-private cache依附于单个master代理,即对内存系统发起read/write请求的agent.shared cache介于master和slave之间,而且可以按层次组织.slave-private cache不会影响coherence,因为它们都位于单个slave中,不会影响master的其他PMAs,因此在此不再考虑.

对于不被任何代理所缓存的共享内存区域,可以简单的提供coherence PMA.此类区域的PMA将简单的表示它不应被私有或共享cache所缓存.

对于只读的区域,coherence也是简单的,此区域可以安全的被多个代理缓存,而不需要缓存一致性策略.

一些读-写区域可能只被单个代理所访问,在这种情况下它们可以被此代理的私有地缓存,而不需要缓存一致性策略.此类区域的PMA将表示它们可以被缓存.这些数据也可以缓存到共享cache中,因为其他dialing不会访问此区域.

如果一个代理可以缓存一个读-写区域,且此区域会被其他代理访问,不论是否缓存,都要求有缓存一致性策略来避免使用过时了的值.在缺少硬件缓存一致性(硬件非一致性区域)的区域中,缓存一致性可以完全在软件中实现,不过软件一致性策略很难正确的实现,而且经常因为需要保守的软件指导的缓存冲刷而有着严重的性能影响.硬件缓存一致性策略需要更复杂的硬件以及因为缓存一致性探测可能会影响性能,不过对硬件都是不可见的.but are otherwise invisible to software.

对每个硬件缓存一致性区域,PMA会指出此区域是一致性的,而且如果此系统包含多个一致性控制器,还会指出使用了哪个硬件一致性控制器.

对于一些系统,一致性控制器可能是有一个外部的共享缓存,此共享缓存本身可能按层级的访问其他外部缓存一致性控制器.

平台中的大部分内存区域对软件都是一致性的,因为它们都会固定为要么不可缓存,要么只读,要么硬件一致性,或者只会被一个代理访问.

3.6.6 Idempotency PMAs

Idempotency PMAs描述对一个地址区域的读和写是否具有幂等性的(idempotent).主存区域都假设为幂等性的.对于I/O区域,读和写的幂等性可以单独的指定(例如读是幂等的但是写不是).如果访问是非幂等的,即对任何的读或写访问都有潜在的副作用,则应当避免推测性的或冗余的访问.

为了定义幂等性PMAs,不认为由冗余访问引起的内存序(观察到的内存序)的改变是副作用

尽管设计的硬件应当避免推测性的或冗余性的访问标记为非幂等性的内存区域,但是依然有必要确保软件或者编译器优化不会对非幂等性 内存区域产生虚假的访问.

非幂等性的区域可能不支持非对齐访问.对这种区域的非对齐访问应当引起access-fault exception而不是address-misaligned exception,这表示软件不应当使用多个小的访问来模拟此非对齐访问,因为这样可能导致无法预料的副作用.

3.7 Physical Memory Protection

为支持安全处理和抑制错误,有必要限制运行在hart的软件对物理地址的访问性.可选的物理内存保护(physical memory protection,PMP)单元为每个hart提供机器模式控制寄存器以允许对每个物理内存区域指定物理内存访问权限(read,write,execute).PMP值和3.6节所描述的PMA一起并行的检查.

PMP访问控制设置的粒度由平台特定,不过标准PMP编码最小支持4字节的区域.特定区域的权限可以是硬连线的--举个例子,某些区域可能永远只是机器模式可见的,而低特权层级不可见.

平台对物理内存保护的需求千差万别,一些平台可能提供其他PMP结构,以补充或替代本节所描述的方案.

PMP检查应用于所有有效特权模式是S或U的访问,包括S和U模式下的取指,以及当mstatus寄存器中的MPRV位清零时S和U模式的数据访问,还有当mstatus的MPRV位置位以及MPP字段是S或U情况下的所有特权模式的数据访问.PMP检查还应用于有效特权模式是S情况下的虚拟地址转换的页表访问.可选地,PMP检查可能额外地应用到M-mode的访问,在这种情况下,PMP寄存器本身是锁定的,即使是M-mode的软件都不能改变这些寄存器,直到此hart复位.实际上,PMP可以为S和U模式授予许可,其默认是没有许可的,也可以从M-mode撤销许可,其默认拥有所有许可.

处理器上总是能精准的trap PMP违例.

3.7.1 Physical Memory Protection CSRs

通过一个8位的配置寄存器和一个MXLEN位的地址寄存器描述PMP表项.一些PMP设置额外地使用了与先前PMP表项相关联的地址寄存器.实现可以实现0,16,或64个PMP CSRs.所有的PMP CSR字段都是WARL的,且可以硬连线到0. PMP CSRs只可被M-mode访问.

为了最小化上下文切换事件,PMP配置寄存器都打包进CSRs.对于RV32,16个CSRs,pmpcfg0-pmpcfg15位64个PMP表项保存着配置pmp0cfg-pmp3cfg,如图3.27所示.对于RV64,8个偶数编号的CSRs,pmpcfg0,pmpcfg2,...,pmpcfg14为64个PMP表项保存着配置,如图3.28所示.对于RV64,奇数编号的配置寄存器pmpcfg1,pmpcfg3,...,pmpcfg15都是非法的.

RV64系统使用pmpcfg2保存PMP表项8-15的配置,而不是pmpcfg1.此设计降低了支持多MXLEN值的开销,因为在RV32和RV64的pmpcfg2[31:0]中都出现了PMP表项9-11.

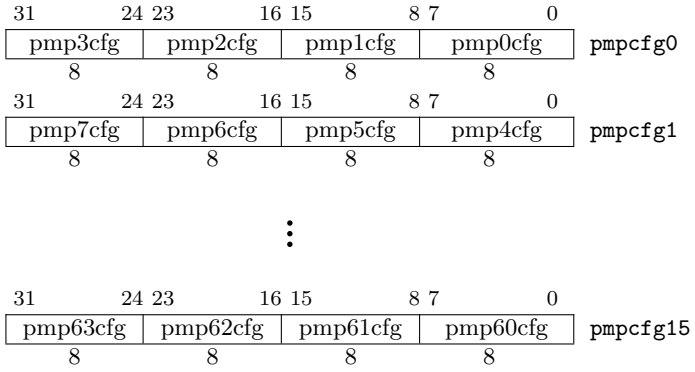


图 3.27: RV32 PMP 配置CSR 布局.

PMP地址寄存器为命名pmpadr0-pmpaddr64的CSR.对于RV32,每个PMP地址寄存器编码34位物理地址的33-2位,如图3.29所示.对于RV64,每个PMP地址寄存器编码56位物理地址的55-2位,如图3.30所示.不是所有的物理地址位都被实现了,所有pmpaddr寄存器是WARL的.

对于RV32,4.3节描述的Sv32基于分页的虚拟内存方案支持34位的物理地址,所以PMP方案必须支持比XLEN更宽的地址.对于RV64,4.4节和4.5节描述的Sv39和Sv48基于分页的虚拟内存方案支持56位的物理地址空间,所以RV64的PMP地址寄存器加上了相同的限制.

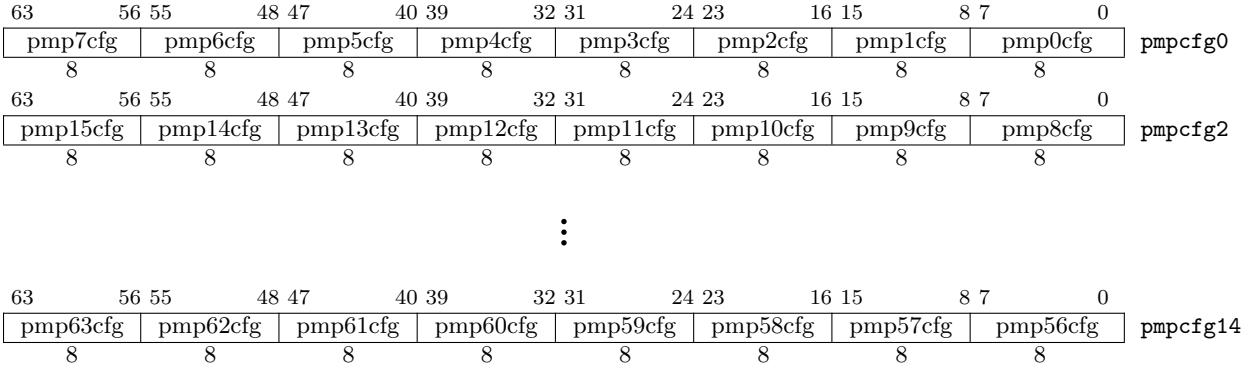


图 3.28: RV64 PMP 配置CSR 布局.

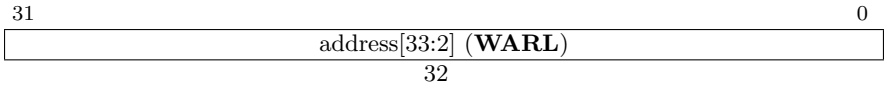


Figure 3.29: PMP 地址寄存器格式, RV32.

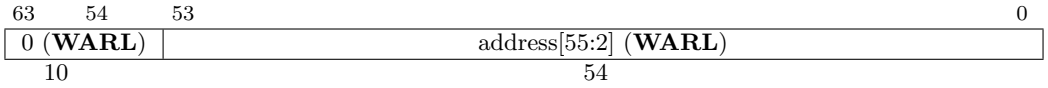


图 3.30: PMP 地址寄存器格式, RV64.

图3.31展示了PMP配置寄存器的布局.当置位R,W,和X位时,分别表示此PMP表项允许读,写和执行指令.当其中某位清零时,则禁止相应的访问.R,W,X字段构成了一个统一的WARL字段,组合R=0和W=1保留.剩余两个字段A和L在后续小节描述.

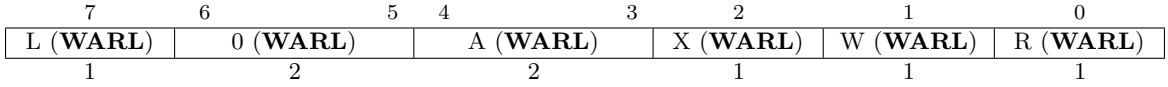


Figure 3.31: PMP configuration register format.

尝试从没有执行许可的PMP区域进行取将引起instruction access-fault exception.尝试对没有读许可的PMP区域内的地址执行load或load-reserved指令将引起load access-fault exception.尝试对没有写许可的PMP区域内的地址执行store,store-conditional或AMO指令将引起store access-fault exception.

如果MXLEN改变,pmpxcfg字段的内容保留,不过出现在由新MXLEN规定的pmpcfgy CSR中.例如,当MXLEN从64改变到32,pmp4cfg从pmpcfg0[39:32]移动到pmpcfg1[7:0]. pmpaddr CSR遵循常规CSR位宽调整规则,如2.4节所述.

Address Matching

PMP表项的配置寄存器的A字段编码了相关的PMP地址寄存器的地址匹配模式.表3.9展示了此字段的编码. 当A=0时,此PMP表项禁用了,且不匹配任何地址.其他两种支持的地址匹配模式:按2的幂自然对齐的区域(naturally aligned power-of-2 regions,NAPOT),包括按4字节自然对齐区域(naturally aligned four-byte region,NA4)的特殊情况;以及任意范围的顶部边界(Top of range,TOR).这些模式支持四字节粒度.

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

表 3.9: PMP配置寄存器中的A字段编码

NAPOT利用相关的地址寄存器的低位进行范围编码,如表3.10所示

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	$2^{\text{XLEN}+1}$-byte NAPOT range
0111...1111	NAPOT	$2^{\text{XLEN}+2}$-byte NAPOT range
1111...1111	NAPOT	$2^{\text{XLEN}+3}$-byte NAPOT range

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

如果选择了TOR,则相关的地址寄存器构成地址范围的顶部,前一个PMP地址寄存器构成地址范围的底部.如果PMP表项i的A字段设置为TOR,则此表现匹配的地址y是$\texttt{pmpaddr}_{i-1} <= y < \texttt{pmpaddr}_i$(与$\texttt{pmpcfg}_{i-1}$的值无关).如果PMP表项0的A字段设置为TOR,则下边界为0,因此其匹配的地址是$y < \texttt{ppaddr}_0$.

如果$\texttt{pmpaddr}_{i-1} \geq \texttt{pmpaddr}_i$ 且 $\texttt{pmpcfg}_i\texttt{.A}=TOR$, 则 PMP 表项i 不匹配任何地址.

尽管PMP 机制支持的最小区域是四字节,但是平台可能指定更大粒度的PMP区域.通常PMP区域粒度是2^{G+2} 字节,而且对整个PMP区域内都必须相同. 当 $G \geq 1$, 则不可选择NA4模式. 当 $G \geq 2$ 且置位 $\texttt{pmpcfg}_i$.A[1] ,即模式是NAPOT, 则 $\texttt{pmpaddr}_i$[G-2:0] 位读都为1. 当 $G \geq 1$ 且 $\texttt{pmpcfg}_i$.A[1]清零, 即模式是OFF或TOR, 则 $\texttt{pmpaddr}_i$[G-1:0] 位读都为0. 位$\texttt{pmpaddr}_i$[G-1:0] 不影响TOR的地址匹配逻辑.尽管改变$\texttt{pmpcfg}_i$.A[1] 影响从$\texttt{pmpaddr}_i$读取的数值,不过它并不影响存储在寄存器中的底层数值--当$\texttt{pmpcfg}_i$从NAPOT改变为TOR/OFF再变回NAPOT,$\texttt{pmpaddr}_i$[G-1] 保持其原本的值.

软件可能通过向pmp0cfg写入0然后对pmpaddr0全写入1,再读回pmpaddr0的数值来探测PMP粒度.如果把存在1的最低位的位下标作为G,则PMP粒度是2^{G+2}字节.

如果当前的XLEN大于MXLEN,则PMP地址寄存器从MXLEN零拓展到XLEN位宽,以用于地址匹配.

Locking and Privilege Mode

L位指示此PMP表项已经锁定,即对此配置寄存器和相关的地址寄存器的写都会被忽视.锁定的PMP表项持续锁定,直到此hart复位.如果PMP表项i被锁定,则对pmpicfg和pmpaddri的写都被忽略.除此之外,如果PMP表项i被锁定且pmpicfg.A设置为TOR,则对pmpaddri-1的写也被忽略.

即使A字段设置为OFF,置位L位依然会锁定PMP表项.

除了锁定PMP表项,L位还指示是否对M-mode的访问施加R/W/X许可限制.当L位置位,这些许可应用于所有特权模式.当L位清零,任何只要匹配PMP表项的M-mode的访问都会成功;R/W/X许可只应用于S和U模式.

Priority and Matching Logic

PMP表项按静态优先级排列.最低编号的PMP表项匹配一次访问的所有字节,以判断此次访问是否成功.必须匹配一次访问的所有字节才算成功匹配了PMP表项,否则匹配失败,这点与L,R,W和X位无关.例如,如果一个PMP表项被配置为匹配4字节范围0xC-0xF,则对范围0x8-0xF的8字节放回将匹配失败,假设用于匹配此地址的表型是最高优先级的PMP表项.

如果一个PMP表项匹配一次访问的所有字节,则L,R,W和X位判断此次访问是成功还是失败.如果L位清零且此次访问的特权模式是M,此次访问成功.另外,如果L位置位,或此次访问的特权莫是S或U,则只有对应于此次访问类型的R,W,或X位置位时,此次访问才能成功.

如果没有PMP表项匹配上M-mode的访问,此次访问还是成功.如果没有PMP表项匹配上S或U-mode的访问,且至少实现了一个PMP表项,则此次访问失败.

如果至少实现了一个PMP表项,但是所有的PMP表项的A字段都设置为OFF,那么所有的S-mode和U-mode的内存访问都会失败.

失败的访问将产生instruction,load或store access-fault exception.注意,单个指令可能产生多个访问,且这些访问可能不是互相原子的.如果指令所产生的访问至少有一个失败了,则产生access-fault exception,尽管此指令产生的其他访问可能成功了并且引起了可见的副作用.值得注意的是,引用虚拟内存的指令被分解为多次访问.

在某些实现上 ,非对齐的load,store和取指都可能分解为多个访问,在access-fault exception发生之前可能已经有部分访问成功了.尤其是非对齐的store,其中有一部分通过了PMP检查且可能变得可见了,即使另外一部分未通过PMP检查.即使store的地址是自然对齐的,相似的行为也可能会出现在宽度大于XLEN位的浮点store上(例如,在RV32D中的FSD指令).

3.7.2 Physical Memory Protection and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in Chapter 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, memory accesses check the PMP settings synchronously, so no fence is needed.

Chapter 4

Supervisor-Level ISA, Version 1.12

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

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

4.1 Supervisor CSRs

A number of CSRs are provided for the supervisor.

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

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

4.1.1 Supervisor Status Register (`sstatus`)

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

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

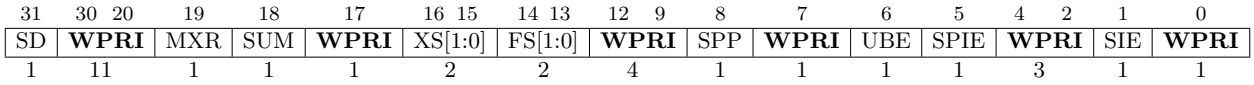


Figure 4.1: Supervisor-mode status register (`sstatus`) for RV32.

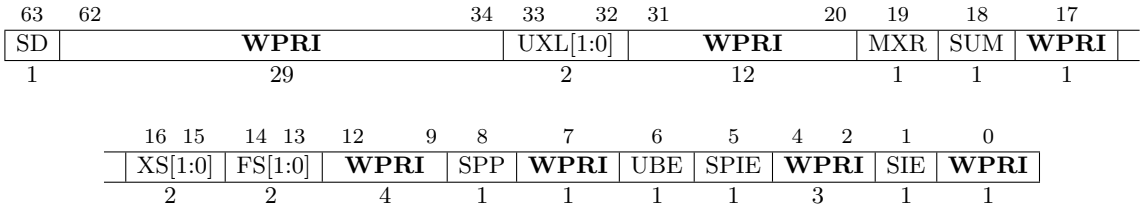


Figure 4.2: Supervisor-mode status register (`sstatus`) for RV64.

privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

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

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

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

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

4.1.1.1 Base ISA Control in `sstatus` Register

The UXL field controls the value of XLEN for U-mode, termed *UXLEN*, which may differ from the value of XLEN for S-mode, termed *SXLEN*. The encoding of UXL is the same as that of the MXL field of `misa`, shown in Table 3.1.

For RV32 systems, the UXL field does not exist, and UXLEN=32. For RV64 systems, it is a **WARL** field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=SXLEN.

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

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

4.1.1.2 Memory Privilege in `sstatus` Register

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in Figure 4.17) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect.

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Figure 4.17) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect, nor when executing in U-mode. Note that S-mode can never execute instructions from user pages, regardless of the state of SUM.

SUM is hardwired to 0 if `satp`.MODE is hardwired to 0.

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

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

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

4.1.1.3 Endianness Control in `sstatus` Register

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

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

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

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

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

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

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

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

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

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

Table 4.1: Encoding of `stvec` MODE field.

The encoding of the MODE field is shown in Table 4.1. When MODE=Direct, all traps into supervisor mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into supervisor mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number. For example, a supervisor-mode timer interrupt (see Table 4.2) causes the `pc` to be set to BASE+`0x14`. Setting MODE=Vectored may impose a stricter alignment constraint on BASE.

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

The `sip` register is an SXLEN-bit read/write register containing information on pending interrupts, while `sie` is the corresponding SXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number i (as reported in CSR `scause`, Section 4.1.8) corresponds with bit i in both `sip` and `sie`. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform or custom use.

An interrupt i will trap to S-mode if both of the following are true: (a) either the current privilege mode is S and the SIE bit in the `sstatus` register is set, or the current privilege mode has less privilege than S-mode; and (b) bit i is set in both `sip` and `sie`.

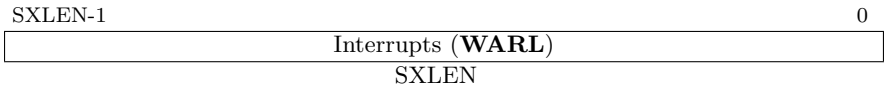


Figure 4.4: Supervisor interrupt-pending register (`sip`).

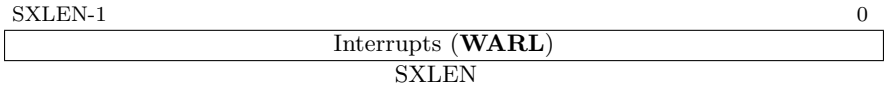


Figure 4.5: Supervisor interrupt-enable register (`sie`).

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

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

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

A bit in `sie` must be writable if the corresponding interrupt can ever become pending. Bits of `sie` that are not writable must be hardwired to zero.

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

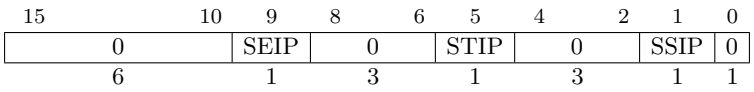


Figure 4.6: Standard portion (bits 15:0) of `sip`.

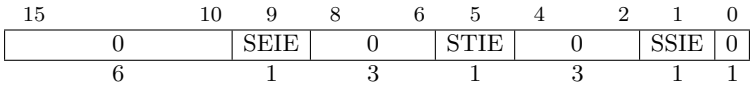


Figure 4.7: Standard portion (bits 15:0) of `sie`.

Bits `sip`.SEIP and `sie`.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. If implemented, SEIP is read-only in `sip`, and is set and cleared by the execution environment, typically through a platform-specific interrupt controller.

Bits `sip`.STIP and `sie`.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If implemented, STIP is read-only in `sip`, and is set and cleared by the execution environment.

Bits `sip`.SSIP and `sie`.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. If implemented, SSIP is writable in `sip`. A supervisor-level software interrupt

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

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

Each standard interrupt type (SEI, STI, or SSI) may not be implemented, in which case the corresponding interrupt-pending and interrupt-enable bits are hardwired to zeros. All bits in `sip` and `sie` are **WARL** fields. The implemented interrupts may be found by writing one to every bit location in `sie`, then reading back to see which bit positions hold a one.

The `sip` *and* `sie` *registers are subsets of the* `mip` *and* `mie` *registers. Reading any implemented field, or writing any writable field, of* `sip`*/*`sie` *effects a read or write of the homonymous field of* `mip`*/*`mie`*.*

Bits 3, 7, and 11 of `sip` *and* `sie` *correspond to the machine-mode software, timer, and external interrupts, respectively. Since most platforms will choose not to make these interrupts delegatable from M-mode to S-mode, they are shown as hardwired to 0 in Figures 4.6 and 4.7.*

Multiple simultaneous interrupts destined for supervisor mode are handled in the following decreasing priority order: SEI, SSI, STI.

4.1.4 Supervisor Timers and Performance Counters

Supervisor software uses the same hardware performance monitoring facility as user-mode software, including the `time`, `cycle`, and `instret` CSRs. The implementation should provide a mechanism to modify the counter values.

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

4.1.5 Counter-Enable Register (`scounteren`)

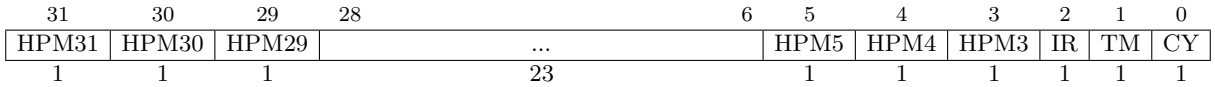


Figure 4.8: Counter-enable register (`scounteren`).

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

When the CY, TM, IR, or 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.

The setting of a bit in `mcounteren` *does not affect whether the corresponding bit in* `scounteren` *is writable. However, U-mode may only access a counter if the corresponding bits in* `scounteren` *and* `mcounteren` *are both set.*

4.1.6 Supervisor Scratch Register (`sscratch`)

The `sscratch` register is an SXLEN-bit read/write register, dedicated for use by the supervisor. Typically, `sscratch` is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, `sscratch` is swapped with a user register to provide an initial working register.

Figure 4.9: Supervisor Scratch Register.

4.1.7 Supervisor Exception Program Counter (`sepc`)

`sepc` is an SXLEN-bit read/write register formatted as shown in Figure 4.10. The low bit of `sepc` (`sepc[0]`) is always zero. On implementations that support only IALIGN=32, the two low bits (`sepc[1:0]`) are always zero.

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

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

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

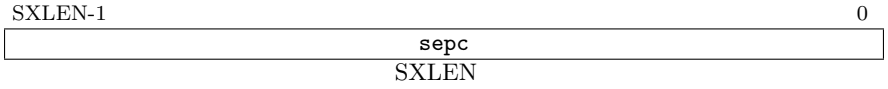


Figure 4.10: Supervisor exception program counter register.

4.1.8 Supervisor Cause Register (`scause`)

The `scause` register is an SXLEN-bit read-write register formatted as shown in Figure 4.11. When a trap is taken into S-mode, `scause` is written with a code indicating the event that caused the trap.

Otherwise, `scause` is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the `scause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Table 4.2 lists the possible exception codes for the current supervisor ISAs. The Exception Code is a **WLRL** field. It is required to hold the values 0–31 (i.e., bits 4–0 must be implemented), but otherwise it is only guaranteed to hold supported exception codes.

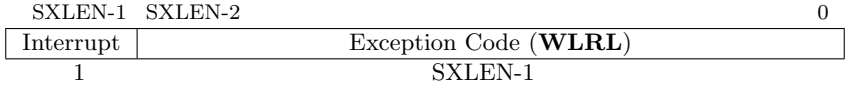


Figure 4.11: Supervisor Cause register `scause`.

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

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

4.1.9 Supervisor Trap Value (`stval`) Register

The `stval` register is an SXLEN-bit read-write register formatted as shown in Figure 4.12. When a trap is taken into S-mode, `stval` is written with exception-specific information to assist software in handling the trap. Otherwise, `stval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `stval` informatively and which may unconditionally set it to zero.

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

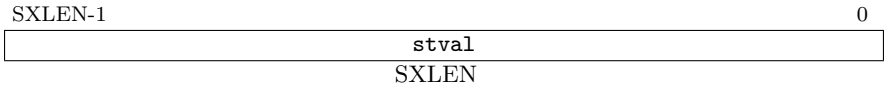


Figure 4.12: Supervisor Trap Value register.

For misaligned loads and stores that cause access-fault or page-fault exceptions, `stval` will contain the virtual address of the portion of the access that caused the fault. For instruction access-fault or page-fault exceptions on systems with variable-length instructions, `stval` will contain the virtual address of the portion of the instruction that caused the fault while `sepc` will point to the beginning of the instruction.

The `stval` register can optionally also be used to return the faulting instruction bits on an illegal instruction exception (`sepc` points to the faulting instruction in memory).

If this feature is not provided, then `stval` is set to zero on an illegal instruction fault.

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

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

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

`stval` is a **WARL** register that must be able to hold all valid virtual addresses and the value 0. It need not be capable of holding all possible invalid addresses. Prior to writing `stval`, implementations may convert an invalid address into some other invalid address that `stval` is capable of holding. If the feature to return the faulting instruction bits is implemented, `stval` must also be able to hold all values less than 2^N, where N is the smaller of XLEN and ILEN.

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

The `satp` register is an SXLEN-bit read/write register, formatted as shown in Figure 4.13 for SXLEN=32 and Figure 4.14 for SXLEN=64, which controls supervisor-mode address translation

and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the MODE field, which selects the current address-translation scheme. Further details on the access to this register are described in Section 3.1.6.5.

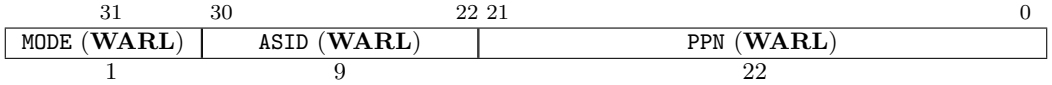


Figure 4.13: RV32 Supervisor address translation and protection register `satp`.

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

The `satp`*.PPN field might not be capable of holding all physical page numbers. Some platform standards might place constraints on the values* `satp`*.PPN may assume, e.g., by requiring that all physical page numbers corresponding to main memory be representable.*

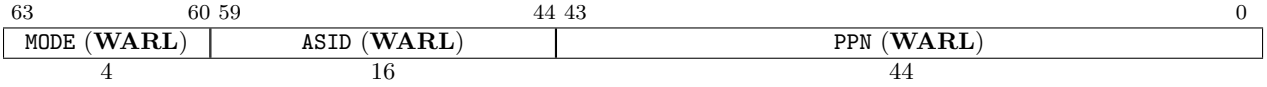


Figure 4.14: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

We store the ASID and the page table base address in the same CSR to allow the pair to be changed atomically on a context switch. Swapping them non-atomically could pollute the old virtual address space with new translations, or vice-versa. This approach also slightly reduces the cost of a context switch.

Table 4.3 shows the encodings of the MODE field for RV32 and RV64. When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 3.7. To select MODE=Bare, software must write zero to the remaining fields of `satp` (bits 30–0 for RV32, or bits 59–0 for RV64). Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an UNSPECIFIED effect on the value that the remaining fields assume and an UNSPECIFIED effect on address translation and protection behavior.

For RV32, the `satp` encodings corresponding to MODE=Bare and ASID[8:7]=3 are designated for custom use, whereas the encodings corresponding to MODE=Bare and ASID[8:7]≠3 are reserved for future standard use. For RV64, all `satp` encodings corresponding to MODE=Bare are reserved for future standard use.

Version 1.11 of this standard stated that the remaining fields in `satp` *had no effect when MODE=Bare. Making these fields reserved facilitates future definition of additional translation and protection modes, particularly in RV32, for which all patterns of the existing MODE field have already been allocated.*

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 for standard use*
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.*
12–13	—	*Reserved for standard use*
14–15	—	*Designated for custom use*

Table 4.3: Encoding of `satp` MODE field.

The number of ASID bits is UNSPECIFIED and may be zero. The number of implemented ASID bits, termed *ASIDLEN*, may be determined by writing one to every bit position in the ASID field, then reading back the value in `satp` to see which bit positions in the ASID field hold a one. The least-significant bits of ASID are implemented first: that is, if ASIDLEN > 0, ASID[ASIDLEN-1:0] is writable. The maximal value of ASIDLEN, termed ASIDMAX, is 9 for Sv32 or 16 for Sv39 and Sv48.

For many applications, the choice of page size has a substantial performance impact. A large page size increases TLB reach and loosens the associativity constraints on virtually indexed, physically tagged caches. At the same time, large pages exacerbate internal fragmentation, wasting physical memory and possibly cache capacity.

After much deliberation, we have settled on a conventional page size of 4 KiB for both RV32 and RV64. We expect this decision to ease the porting of low-level runtime software and device drivers. The TLB reach problem is ameliorated by transparent superpage support in modern operating systems [2]. Additionally, multi-level TLB hierarchies are quite inexpensive relative to the multi-level cache hierarchies whose address space they map.

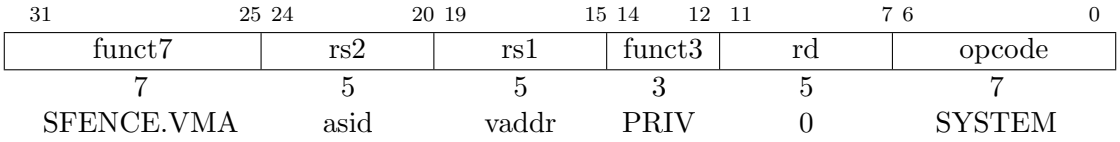
Note that writing `satp` does not imply any ordering constraints between page-table updates and subsequent address translations. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an SFENCE.VMA instruction (see Section 4.2.1) after writing `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 3.3.2, 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 explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before all subsequent implicit references from that hart to the memory-management data structures. Further details on the behavior of this instruction are described in Section 3.1.6.5 and Section 3.7.2.

The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VMA is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

SFENCE.VMA orders only the local hart's implicit references to the memory-management data structures.

Consequently, other harts must be notified separately when the memory-management data structures have been modified. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VMA in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shootdown.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, *rs2* can specify the address space. The behavior of SFENCE.VMA depends on *rs1* and *rs2* as follows:

- If *rs1*=`x0` and *rs2*=`x0`, the fence orders all reads and writes made to any level of the page tables, for all address spaces.
- If *rs1*=`x0` and *rs2*≠`x0`, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register *rs2*. Accesses to *global* mappings (see Section 4.3.1) are not ordered.
- If *rs1*≠`x0` and *rs2*=`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces.
- If *rs1*≠`x0` and *rs2*≠`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for the address space identified by integer register *rs2*. Accesses to global mappings are not ordered.

When *rs2*≠`x0`, bits SXLEN-1:ASIDMAX of the value held in *rs2* are reserved for future use and should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLEN < ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN of the value held in *rs2*.

Simpler implementations can ignore the virtual address in rs1 *and the ASID value in* rs2 *and always perform a global fence.*

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

A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. In particular, if a leaf PTE is modified but a subsuming SFENCE.VMA is not executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.

In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with rs1=`x0`*. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.*

This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

Implementations must only perform implicit reads of the translation data structures pointed to by the current contents of the `satp` register or a subsequent valid (V=1) translation data structure entry, and must only raise exceptions for implicit accesses that are generated as a result of instruction execution, not those that are performed speculatively.

Changes to the `sstatus` fields SUM and MXR take effect immediately, without the need to execute an SFENCE.VMA instruction. Changing `satp`.MODE from Bare to other modes and vice versa also takes effect immediately, without the need to execute an SFENCE.VMA instruction. Likewise, changes to `satp`.ASID take effect immediately.

The following common situations typically require executing an SFENCE.VMA instruction:

- *When software recycles an ASID (i.e., reassociates it with a different page table), it should* first *change* `satp` *to point to the new page table using the recycled ASID,* then *execute SFENCE.VMA with* rs1=`x0` *and* rs2 *set to the recycled ASID. Alternatively, software can execute the same SFENCE.VMA instruction while a different ASID is loaded into* `satp`*, provided the next time* `satp` *is loaded with the recycled ASID, it is simultaneously loaded with the new page table.*
- *If the implementation does not provide ASIDs, or software chooses to always use ASID 0, then after every* `satp` *write, software should execute SFENCE.VMA with* rs1=`x0`*. In the common case that no global translations have been modified,* rs2 *should be set to a register other than* `x0` *but which contains the value zero, so that global translations are not flushed.*
- *If software modifies a non-leaf PTE, it should execute SFENCE.VMA with* rs1=`x0`*. If any PTE along the traversal path had its G bit set,* rs2 *must be* `x0`*; otherwise,* rs2 *should be set to the ASID for which the translation is being modified.*
- *If software modifies a leaf PTE, it should execute SFENCE.VMA with* rs1 *set to a virtual address within the page. If any PTE along the traversal path had its G bit set,* rs2 *must be* `x0`*; otherwise,* rs2 *should be set to the ASID for which the translation is being modified.*
- *For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the SFENCE.VMA lazily. After modifying the PTE but before executing SFENCE.VMA, either the new or old permissions will be used. In the latter case, a page-fault exception might occur, at which point software should execute SFENCE.VMA in accordance with the previous bullet point.*

If a hart employs an address-translation cache, that cache must appear to be private to that hart. In particular, the meaning of an ASID is local to a hart; software may choose to use the same ASID to refer to different address spaces on different harts.

A future extension could redefine ASIDs to be global across the SEE, enabling such options as shared translation caches and hardware support for broadcast TLB shootdown. However, as OSes have evolved to significantly reduce the scope of TLB shootdowns using novel ASID-management techniques, we expect the local-ASID scheme to remain attractive for its simplicity and possibly better scalability.

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

When Sv32 is written to the MODE field in the `satp` register (see Section 4.1.10), the supervisor operates in a 32-bit paged virtual-memory system. In this mode, supervisor and user virtual addresses are translated into supervisor physical addresses by traversing a radix-tree page table. Sv32 is supported on RV32 systems and is designed to include mechanisms sufficient for supporting modern Unix-based operating systems.

The initial RISC-V paged virtual-memory architectures have been designed as straightforward implementations to support existing operating systems. We have architected page table layouts

to support a hardware page-table walker. Software TLB refills are a performance bottleneck on high-performance systems, and are especially troublesome with decoupled specialized coprocessors. An implementation can choose to implement software TLB refills using a machine-mode trap handler as an extension to M-mode.

Some ISAs architecturally expose virtually indexed, physically tagged *caches, in that accesses to the same physical address via different virtual addresses might not be coherent unless the virtual addresses lie within the same cache set. Implicitly, this specification does not permit such behavior to be architecturally exposed.*

For implementations that hardwire `satp`.MODE to Bare, attempts to execute an SFENCE.VMA instruction might raise an illegal instruction exception.

4.3.1 Addressing and Memory Protection

Sv32 implementations support a 32-bit virtual address space, divided into 4 KiB pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown in Figure 4.15. When Sv32 virtual memory mode is selected in the MODE field of the `satp` register, supervisor virtual addresses are translated into supervisor physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12-bit page offset is untranslated. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures (Sections 3.7), before being directly converted to machine-level physical addresses.

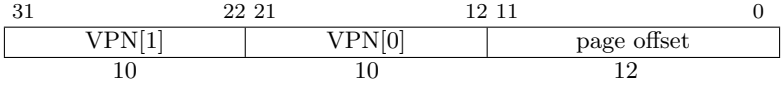


Figure 4.15: Sv32 virtual address.

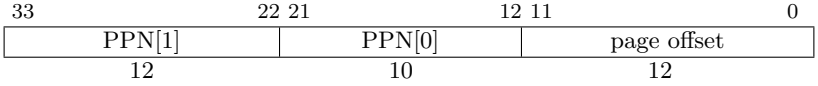


Figure 4.16: Sv32 physical address.

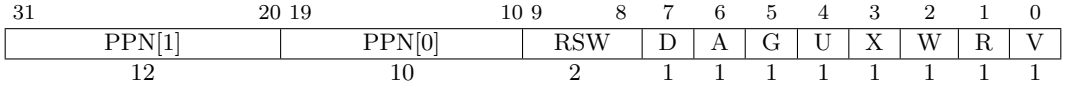


Figure 4.17: Sv32 page table entry.

Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register.

The PTE format for Sv32 is shown in Figures 4.17. The V bit indicates whether the PTE is valid; if it is 0, all other bits in the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is

a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use. Table 4.4 summarizes the encoding of the permission bits.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

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

Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load or load-reserved instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional, or AMO instruction whose effective address lies within a page without write permissions raises a store page-fault exception.

AMOs never raise load page-fault exceptions. Since any unreadable page is also unwritable, attempting to perform an AMO on an unreadable page always raises a store page-fault exception.

The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the SUM bit in the `sstatus` register is set, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the SUM bit clear, in which case, supervisor code will fault on accesses to user-mode pages. Irrespective of SUM, the supervisor may not execute code on pages with U=1.

An alternative PTE format would support different permissions for supervisor and user. We omitted this feature because it would be largely redundant with the SUM mechanism (see Section 4.1.1.2) and would require more encoding space in the PTE.

The G bit designates a *global* mapping. Global mappings are those that exist in all address spaces. For non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global. Note that failing to mark a global mapping as global merely reduces performance, whereas marking a non-global mapping as global is a software bug that, after switching to an address space with a different non-global mapping for that address range, can unpredictably result in either mapping being used.

Global mappings need not be stored redundantly in address-translation caches for multiple ASIDs. Additionally, they need not be flushed from local address-translation caches when an SFENCE.VMA instruction is executed with rs2≠`x0`.

The RSW field is reserved for use by supervisor software; the implementation shall ignore this field.

Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared.

Two schemes to manage the A and D bits are permitted:

- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, the implementation sets the corresponding bit(s) in the PTE. The PTE update must be atomic with respect to other accesses to the PTE, and must atomically check that the PTE is valid and grants sufficient permissions. The PTE update must be exact (i.e., not speculative), and observed in program order by the local hart. Furthermore, the PTE update must appear in the global memory order no later than the explicit memory access, or any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

  The PTE update is not required to be atomic with respect to the explicit memory access that caused the update, and the sequence is interruptible. However, the hart must not perform the explicit memory access before the PTE update is globally visible.

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

Mandating that the PTE updates to be exact, atomic, and in program order simplifies the specification, and makes the feature more useful for system software. Simple implementations may instead generate page-fault exceptions.

The A and D bits are never cleared by the implementation. If the supervisor software does not rely on accessed and/or dirty bits, e.g. if it does not swap memory pages to secondary storage or if the pages are being used to map I/O space, it should always set them to 1 in the PTE to improve performance.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv32 supports 4 MiB *megapages*. A megapage must be virtually and physically aligned to a 4 MiB boundary; a page-fault exception is raised if the physical address is insufficiently aligned.

For non-leaf PTEs, the D, A, and U bits are reserved for future standard use and must be cleared by software for forward compatibility.

For implementations with both page-based virtual memory and the "A" standard extension, the LR/SC reservation set must lie completely within a single base page (i.e., a naturally aligned 4 KiB region).

4.3.2 Virtual Address Translation Process

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

1. Let a be `satp`.$ppn \times$ 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-fault 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-fault 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 $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn$[LEVELS $- 1 : i] = pte.ppn$[LEVELS $- 1 : i]$.

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

This section describes a simple paged virtual-memory system designed for RV64 systems, which supports 39-bit virtual address spaces. The design of Sv39 follows the overall scheme of Sv32, and this section details only the differences between the schemes.

We specified multiple virtual memory systems for RV64 to relieve the tension between providing a large address space and minimizing address-translation cost. For many systems, 512 GiB of virtual-address space is ample, and so Sv39 suffices. Sv48 increases the virtual address space to 256 TiB, but increases the physical memory capacity dedicated to page tables, the latency of page-table traversals, and the size of hardware structures that store virtual addresses.

4.4.1 Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into 4 KiB pages. An Sv39 address is partitioned as shown in Figure 4.18. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else a page-fault exception will occur. The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

When mapping between narrower and wider addresses, RISC-V usually zero-extends a narrower address to a wider size. The mapping between 64-bit virtual addresses and the 39-bit usable address space of Sv39 is not based on zero-extension but instead follows an entrenched convention that allows an OS to use one or a few of the most-significant bits of a full-size (64-bit) virtual address to quickly distinguish user and supervisor address regions.

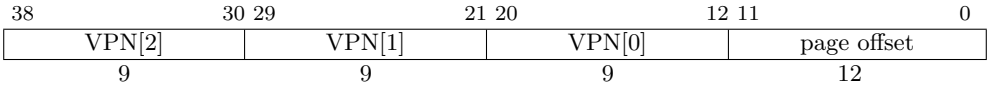


Figure 4.18: Sv39 virtual address.

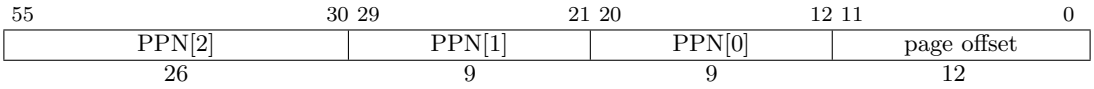


Figure 4.19: Sv39 physical address.

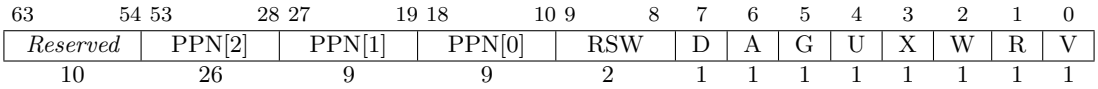


Figure 4.20: Sv39 page table entry.

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register's PPN field.

The PTE format for Sv39 is shown in Figure 4.20. Bits 9–0 have the same meaning as for Sv32. Bits 63–54 are reserved for future standard use and must be zeroed by software for forward compatibility.

We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency. These reserved bits may also be used to facilitate research experimentation. The cost is reducing the physical address space, but 64 PiB is presently ample. When it no longer suffices, the reserved bits that remain unallocated could be used to expand the physical address space.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB *megapages* and 1 GiB *gigapages*, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 4.3.2, except 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 4.21. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else a page-fault exception will occur. The 36-bit VPN is translated into a 44-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

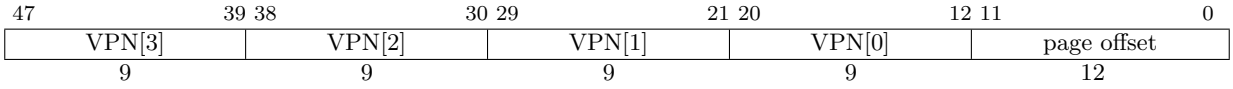


Figure 4.21: Sv48 virtual address.

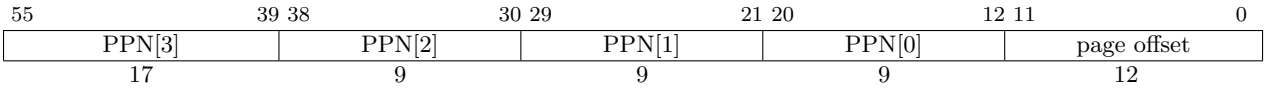


Figure 4.22: Sv48 physical address.

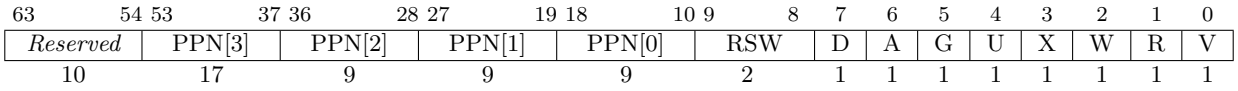


Figure 4.23: Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 4.23. Bits 9–0 have the same meaning as for Sv32. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv48 supports 2 MiB *megapages*, 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 0.6.2

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

This chapter describes the RISC-V hypervisor extension, which virtualizes the supervisor-level architecture to support the efficient hosting of guest operating systems atop a type-1 or type-2 hypervisor. The hypervisor extension changes supervisor mode into *hypervisor-extended supervisor mode* (HS-mode, or *hypervisor mode* for short), where a hypervisor or a hosting-capable operating system runs. The hypervisor extension also adds another stage of address translation, from *guest physical addresses* to supervisor physical addresses, to virtualize the memory and memory-mapped I/O subsystems for a guest operating system. HS-mode acts the same as S-mode, but with additional instructions and CSRs that control the new stage of address translation and support hosting a guest OS in virtual S-mode (VS-mode). Regular S-mode operating systems can execute without modification either in HS-mode or as VS-mode guests.

In HS-mode, an OS or hypervisor interacts with the machine through the same SBI as an OS normally does from S-mode. An HS-mode hypervisor is expected to implement the SBI for its VS-mode guest.

The hypervisor extension is enabled by setting bit 7 in the `misa` CSR, which corresponds to the letter H. RISC-V harts that implement the hypervisor extension are encouraged not to hardwire `misa`[7], so that the extension may be disabled.

The baseline privileged architecture is designed to simplify the use of classic virtualization techniques, where a guest OS is run at user-level, as the few privileged instructions can be easily detected and trapped. The hypervisor extension improves virtualization performance by reducing the frequency of these traps.

The hypervisor extension has been designed to be efficiently emulable on platforms that do not implement the extension, by running the hypervisor in S-mode and trapping into M-mode for hypervisor CSR accesses and to maintain shadow page tables. The majority of CSR accesses for type-2 hypervisors are valid S-mode accesses so need not be trapped. Hypervisors can support nested virtualization analogously.

5.1 Privilege Modes

The current *virtualization mode*, denoted V, indicates whether the hart is currently executing in a guest. When V=1, the hart is either in virtual S-mode (VS-mode), or in virtual U-mode (VU-mode) atop a guest OS running in VS-mode. When V=0, the hart is either in M-mode, in HS-mode, or in U-mode atop an OS running in HS-mode. The virtualization mode also indicates whether two-stage address translation is active (V=1) or inactive (V=0). Table 5.1 lists the possible privilege modes of a RISC-V hart with the hypervisor extension.

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

Table 5.1: Privilege modes with the hypervisor extension.

For privilege modes U and VU, the *nominal privilege mode* is U, and for privilege modes HS and VS, the nominal privilege mode is S.

HS-mode is more privileged than VS-mode, and VS-mode is more privileged than VU-mode. VS-mode interrupts are globally disabled when executing in U-mode.

This description does not consider the possibility of U-mode or VU-mode interrupts and will be revised if an extension for user-level interrupts is adopted.

5.2 Hypervisor and Virtual Supervisor CSRs

An OS or hypervisor running in HS-mode uses the supervisor CSRs to interact with the exception, interrupt, and address-translation subsystems. Additional CSRs are provided to HS-mode, but not to VS-mode, to manage two-stage address translation and to control the behavior of a VS-mode guest: `hstatus`, `hedeleg`, `hideleg`, `hvip`, `hip`, `hie`, `hgeip`, `hgeie`, `hcounteren`, `htimedelta`, `htimedeltah`, `htval`, `htinst`, and `hgatp`.

Furthermore, several *virtual supervisor* CSRs (VS CSRs) are replicas of the normal supervisor CSRs. For example, `vsstatus` is the VS CSR that duplicates the usual `sstatus` CSR.

When V=1, the VS CSRs substitute for the corresponding supervisor CSRs, taking over all functions of the usual supervisor CSRs except as specified otherwise. Instructions that normally read or modify a supervisor CSR shall instead access the corresponding VS CSR. When V=1, an attempt to read or write a VS CSR directly by its own separate CSR address causes a virtual instruction exception. (Attempts from U-mode cause an illegal instruction exception as usual.) The VS CSRs can be accessed as themselves only from M-mode or HS-mode.

While V=1, the normal HS-level supervisor CSRs that are replaced by VS CSRs retain their values but do not affect the behavior of the machine unless specifically documented to do so. Conversely, when V=0, the VS CSRs do not ordinarily affect the behavior of the machine other than being readable and writable by CSR instructions.

Some standard supervisor CSRs (`scounteren` and `scontext`, possibly others) have no matching VS CSR. These supervisor CSRs continue to have their usual function and accessibility even when V=1, except with VS-mode and VU-mode substituting for HS-mode and U-mode. Hypervisor software is expected to manually swap the contents of these registers as needed.

Matching VS CSRs exist only for the supervisor CSRs that must be duplicated, which are mainly those that get automatically written by traps or that impact instruction execution immediately after trap entry and/or right before SRET, when software alone is unable to swap a CSR at exactly the right moment. Currently, most supervisor CSRs fall into this category, but future ones might not.

In this chapter, we use the term *HSXLEN* to refer to the effective XLEN when executing in HS-mode, and *VSXLEN* to refer to the effective XLEN when executing in VS-mode.

5.2.1 Hypervisor Status Register (`hstatus`)

The `hstatus` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.1 when HSXLEN=32 and Figure 5.2 when HSXLEN=64. The `hstatus` register provides facilities analogous to the `mstatus` register for tracking and controlling the exception behavior of a VS-mode guest.

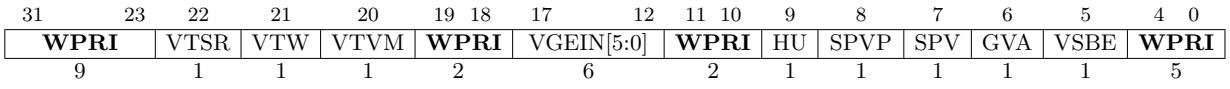


Figure 5.1: Hypervisor status register (`hstatus`) for RV32.

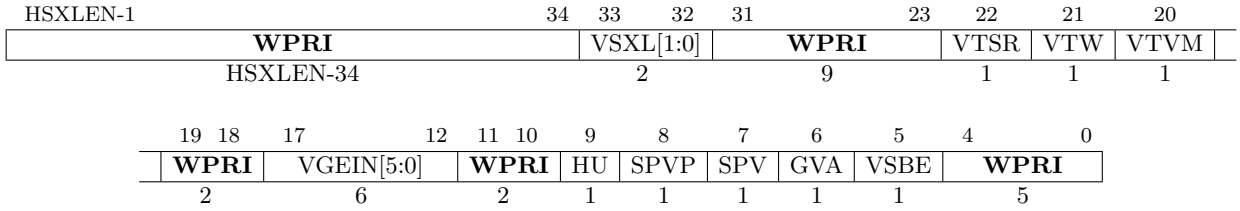


Figure 5.2: Hypervisor status register (`hstatus`) for RV64.

The VSXL field controls the effective XLEN for VS-mode (known as VSXLEN), which may differ from the XLEN for HS-mode (HSXLEN). When HSXLEN=32, the VSXL field does not exist, and VSXLEN=32. When HSXLEN=64, VSXL is a **WARL** field that is encoded the same as the MXL field of `misa`, shown in Table 3.1 on page 16. In particular, an implementation may make VSXL be a read-only field whose value always ensures that VSXLEN=HSXLEN.

If HSXLEN is changed from 32 to a wider width, and if field VSXL is not restricted to a single value, it gets the value corresponding to the widest supported width not wider than the new HSXLEN.

The `hstatus` fields VTSR, VTW, and VTVM are defined analogously to the `mstatus` fields TSR, TW, and TVM, but affect execution only in VS-mode, and cause virtual instruction exceptions instead of illegal instruction exceptions. When VTSR=1, an attempt in VS-mode to execute SRET raises a virtual instruction exception. When VTW=1 (and assuming `mstatus`.TW=0), an attempt in VS-mode to execute WFI raises a virtual instruction exception if the WFI does not complete within an implementation-specific, bounded time limit. When VTVM=1, an attempt in VS-mode to execute SFENCE.VMA or to access CSR `satp` raises a virtual instruction exception.

The VGEIN (Virtual Guest External Interrupt Number) field selects a guest external interrupt source for VS-level external interrupts. VGEIN is a **WLRL** field that must be able to hold values between zero and the maximum guest external interrupt number (known as GEILEN), inclusive. When VGEIN=0, no guest external interrupt source is selected for VS-level external interrupts. GEILEN may be zero, in which case VGEIN may be hardwired to zero. Guest external interrupts are explained in Section 5.2.4, and the use of VGEIN is covered further in Section 5.2.3.

Field HU (Hypervisor User mode) controls whether the virtual-machine load/store instructions, HLV, HLVX, and HSV, can be used also in U-mode. When HU=1, these instructions can be executed in U-mode the same as in HS-mode. When HU=0, all hypervisor instructions cause an illegal instruction trap in U-mode.

The HU bit allows a portion of a hypervisor to be run in U-mode for greater protection against software bugs, while still retaining access to a virtual machine's memory.

The SPV bit (Supervisor Previous Virtualization mode) is written by the implementation whenever a trap is taken into HS-mode. Just as the SPP bit in `sstatus` is set to the (nominal) privilege mode at the time of the trap, the SPV bit in `hstatus` is set to the value of the virtualization mode V at the time of the trap. When an SRET instruction is executed when V=0, V is set to SPV.

When V=1 and a trap is taken into HS-mode, bit SPVP (Supervisor Previous Virtual Privilege) is set to the nominal privilege mode at the time of the trap, the same as `sstatus`.SPP. But if V=0 before a trap, SPVP is left unchanged on trap entry. SPVP controls the effective privilege of explicit memory accesses made by the virtual-machine load/store instructions, HLV, HLVX, and HSV.

Without SPVP, if instructions HLV, HLVX, and HSV looked instead to `sstatus`*.SPP for the effective privilege of their memory accesses, then, even with HU=1, U-mode could not access virtual machine memory at VS-level, because to enter U-mode using SRET always leaves SPP=0. Unlike SPP, field SPVP is untouched by transitions back-and-forth between HS-mode and U-mode.*

Field GVA (Guest Virtual Address) is written by the implementation whenever a trap is taken into HS-mode. For any trap (breakpoint, address misaligned, access fault, page fault, or guest-page fault) that writes a guest virtual address to `stval`, GVA is set to 1. For any other trap into HS-mode, GVA is set to 0.

For memory access traps, GVA is redundant with field SPV (the two bits are set the same) except when the explicit memory access of an HLV, HLVX, or HSV instruction causes a fault. In that case, SPV=0 but GVA=1.

If a breakpoint trap or instruction address misaligned trap writes zero to `stval` *instead of the faulting virtual address, then GVA=0 even if SPV=1.*

The VSBE bit is a **WARL** field that controls the endianness of explicit memory accesses made from VS-mode. If VSBE=0, explicit load and store memory accesses made from VS-mode are little-endian, and if VSBE=1, they are big-endian. VSBE also controls the endianness of all implicit accesses to VS-level memory management data structures, such as page tables. An implementation may make VSBE a read-only field that always specifies the same endianness as HS-mode.

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

Registers `hedeleg` and `hideleg` are HSXLEN-bit read/write registers, formatted as shown in Figures 5.3 and 5.4 respectively. By default, all traps at any privilege level are handled in M-mode, though M-mode usually uses the `medeleg` and `mideleg` CSRs to delegate some traps to HS-mode. The `hedeleg` and `hideleg` CSRs allow these traps to be further delegated to a VS-mode guest; their layout is the same as `medeleg` and `mideleg`.

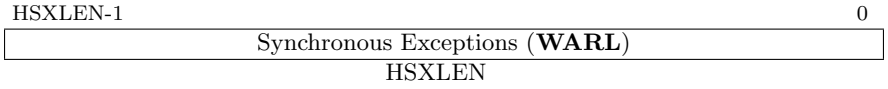


Figure 5.3: Hypervisor exception delegation register (`hedeleg`).

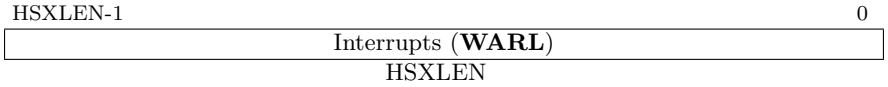


Figure 5.4: Hypervisor interrupt delegation register (`hideleg`).

A synchronous trap that has been delegated to HS-mode (using `medeleg`) is further delegated to VS-mode if V=1 before the trap and the corresponding `hedeleg` bit is set. Each bit of `hedeleg` shall be either writable or hardwired to zero. Many bits of `hedeleg` are required specifically to be writable or zero, as enumerated in Table 5.2. Bit 0, corresponding to instruction address misaligned exceptions, must be writable if IALIGN=32.

Requiring that certain bits of `hedeleg` *be writable reduces some of the burden on a hypervisor to handle variations of implementation.*

An interrupt that has been delegated to HS-mode (using `mideleg`) is further delegated to VS-mode if the corresponding `hideleg` bit is set. Among bits 15:0 of `hideleg`, only bits 10, 6, and 2 (corresponding to the standard VS-level interrupts) shall be writable, and the others shall be hardwired to zero.

When a virtual supervisor external interrupt (code 10) is delegated to VS-mode, it is automatically translated by the machine into a supervisor external interrupt (code 9) for VS-mode, including the

Bit	Attribute	Corresponding Exception
0	(See text)	Instruction address misaligned
1	Writable	Instruction access fault
2	Writable	Illegal instruction
3	Writable	Breakpoint
4	Writable	Load address misaligned
5	Writable	Load access fault
6	Writable	Store/AMO address misaligned
7	Writable	Store/AMO access fault
8	Writable	Environment call from U-mode or VU-mode
9	Read-only 0	Environment call from HS-mode
11	Read-only 0	Environment call from M-mode
12	Writable	Instruction page fault
13	Writable	Load page fault
15	Writable	Store/AMO page fault
20	Read-only 0	Instruction guest-page fault
21	Read-only 0	Load guest-page fault
22	Read-only 0	Virtual instruction
23	Read-only 0	Store/AMO guest-page fault

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

value written to `vscause` on an interrupt trap. Likewise, a virtual supervisor timer interrupt (6) is translated into a supervisor timer interrupt (5) for VS-mode, and a virtual supervisor software interrupt (2) is translated into a supervisor software interrupt (1) for VS-mode. Similar translations may or may not be done for platform or custom interrupt causes (codes 16 and above).

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

Register `hvip` is an HSXLEN-bit read/write register that a hypervisor can write to indicate virtual interrupts intended for VS-mode. Bits of `hvip` that are not writable are hardwired to zeros.

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

The standard portion (bits 15:0) of `hvip` is formatted as shown in Figure 5.6. Bits VSEIP, VSTIP, and VSSIP of `hvip` are writable. Setting VSEIP=1 in `hvip` asserts a VS-level external interrupt; setting VSTIP asserts a VS-level timer interrupt; and setting VSSIP asserts a VS-level software interrupt.

Registers `hip` and `hie` are HSXLEN-bit read/write registers that supplement HS-level's `sip` and `sie` respectively. The `hip` register indicates pending VS-level and hypervisor-specific interrupts, while `hie` contains enable bits for the same interrupts.

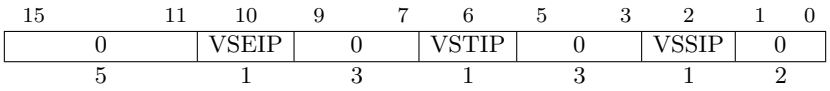


Figure 5.6: Standard portion (bits 15:0) of `hvip`.

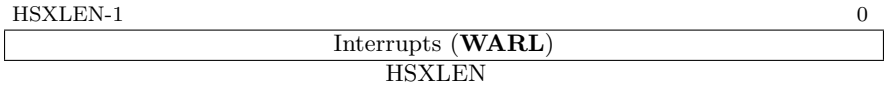


Figure 5.7: Hypervisor interrupt-pending register (`hip`).

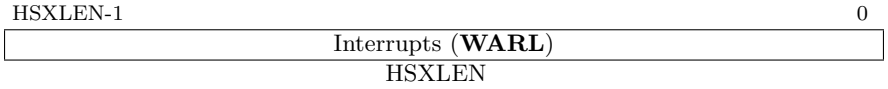


Figure 5.8: Hypervisor interrupt-enable register (`hie`).

For each writable bit in `sie`, the corresponding bit shall be hardwired to zero in both `hip` and `hie`. Hence, the nonzero bits in `sie` and `hie` are always mutually exclusive, and likewise for `sip` and `hip`.

The active bits of `hip` *and* `hie` *cannot be placed in HS-level's* `sip` *and* `sie` *because doing so would make it impossible for software to emulate the hypervisor extension on platforms that do not implement it in hardware.*

An interrupt i will trap to HS-mode whenever all of the following are true: (a) either the current operating mode is HS-mode and the SIE bit in the `sstatus` register is set, or the current operating mode has less privilege than HS-mode; (b) bit i is set in both `sip` and `sie`, or in both `hip` and `hie`; and (c) bit i is not set in `hideleg`.

If bit i of `sie` is hardwired to zero, the same bit in register `hip` may be writable or may be read-only. When bit i in `hip` is writable, a pending interrupt i can be cleared by writing 0 to this bit. If interrupt i can become pending in `hip` but bit i in `hip` is read-only, then either the interrupt can be cleared by clearing bit i of `hvip`, or the implementation must provide some other mechanism for clearing the pending interrupt (which may involve a call to the execution environment).

A bit in `hie` shall be writable if the corresponding interrupt can ever become pending in `hip`. Bits of `hie` that are not writable shall be hardwired to zero.

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

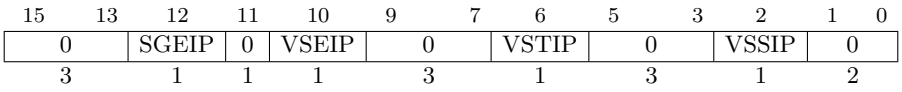


Figure 5.9: Standard portion (bits 15:0) of `hip`.

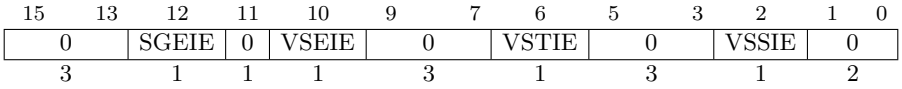


Figure 5.10: Standard portion (bits 15:0) of `hie`.

Bits `hip`.SGEIP and `hie`.SGEIE are the interrupt-pending and interrupt-enable bits for guest external interrupts at supervisor level (HS-level). SGEIP is read-only in `hip`, and is 1 if and only if the bitwise logical-AND of CSRs `hgeip` and `hgeie` is nonzero in any bit. (See Section 5.2.4.)

Bits `hip`.VSEIP and `hie`.VSEIE are the interrupt-pending and interrupt-enable bits for VS-level external interrupts. VSEIP is read-only in `hip`, and is the logical-OR of these interrupt sources:

- bit VSEIP of `hvip`;
- the bit of `hgeip` selected by `hstatus`.VGEIN; and
- any other platform-specific external interrupt signal directed to VS-level.

Bits `hip`.VSTIP and `hie`.VSTIE are the interrupt-pending and interrupt-enable bits for VS-level timer interrupts. VSTIP is read-only in `hip`, and is the logical-OR of `hvip`.VSTIP and any other platform-specific timer interrupt signal directed to VS-level.

Bits `hip`.VSSIP and `hie`.VSSIE are the interrupt-pending and interrupt-enable bits for VS-level software interrupts. VSSIP in `hip` is an alias (writable) of the same bit in `hvip`.

Multiple simultaneous interrupts destined for HS-mode are handled in the following decreasing priority order: SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI.

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

The `hgeip` register is an HSXLEN-bit read-only register, formatted as shown in Figure 5.11, that indicates pending guest external interrupts for this hart. The `hgeie` register is an HSXLEN-bit read/write register, formatted as shown in Figure 5.12, that contains enable bits for the guest external interrupts at this hart. Guest external interrupt number i corresponds with bit i in both `hgeip` and `hgeie`.

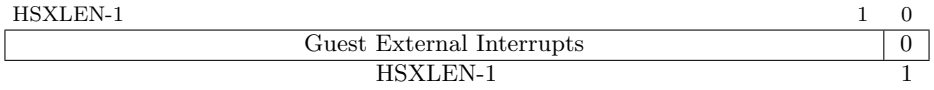


Figure 5.11: Hypervisor guest external interrupt-pending register (`hgeip`).

Figure 5.12: Hypervisor guest external interrupt-enable register (`hgeie`).

Guest external interrupts represent interrupts directed to individual virtual machines at VS-level. If a RISC-V platform supports placing a physical device under the direct control of a guest OS with

minimal hypervisor intervention (known as *pass-through* or *direct assignment* between a virtual machine and the physical device), then, in such circumstance, interrupts from the device are intended for a specific virtual machine. Each bit of `hgeip` summarizes *all* pending interrupts directed to one virtual hart, as collected and reported by an interrupt controller. To distinguish specific pending interrupts from multiple devices, software must query the interrupt controller.

Support for guest external interrupts requires an interrupt controller that can collect virtual-machine-directed interrupts separately from other interrupts.

The number of bits implemented in `hgeip` and `hgeie` for guest external interrupts is UNSPECIFIED and may be zero. This number is known as *GEILEN*. The least-significant bits are implemented first, apart from bit 0. Hence, if GEILEN is nonzero, bits GEILEN:1 shall be writable in `hgeie`, and all other bit positions shall be hardwired to zeros in both `hgeip` and `hgeie`.

The set of guest external interrupts received and handled at one physical hart may differ from those received at other harts. Guest external interrupt number i at one physical hart is typically expected not to be the same as guest external interrupt i at any other hart. For any one physical hart, the maximum number of virtual harts that may directly receive guest external interrupts is limited by GEILEN. The maximum this number can be for any implementation is 31 for RV32 and 63 for RV64, per physical hart.

A hypervisor is always free to emulate *devices for any number of virtual harts without being limited by GEILEN. Only direct pass-through (direct assignment) of interrupts is affected by the GEILEN limit, and the limit is on the number of virtual harts receiving such interrupts, not the number of distinct interrupts received. The number of distinct interrupts a single virtual hart may receive is determined by the interrupt controller.*

Register `hgeie` selects the subset of guest external interrupts that cause a supervisor-level (HS-level) guest external interrupt. The enable bits in `hgeie` do not affect the VS-level external interrupt signal selected from `hgeip` by `hstatus`.VGEIN.

5.2.5 Hypervisor Counter-Enable Register (`hcounteren`)

The counter-enable register `hcounteren` is a 32-bit register that controls the availability of the hardware performance monitoring counters to the guest virtual machine.

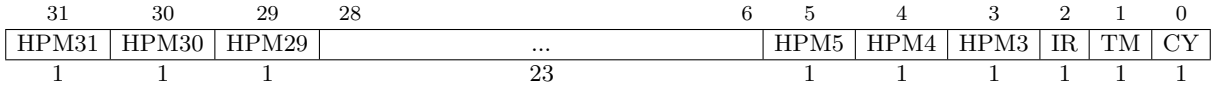


Figure 5.13: Hypervisor counter-enable register (`hcounteren`).

When the CY, TM, IR, or HPMn bit in the `hcounteren` register is clear, attempts to read the `cycle`, `time`, `instret`, or `hpmcounter`n register while V=1 will cause a virtual instruction exception if the same bit in `mcounteren` is 1. When one of these bits is set, access to the corresponding register is permitted when V=1, unless prevented for some other reason. In VU-mode, a counter is not readable unless the applicable bits are set in both `hcounteren` and `scounteren`.

`hcounteren` must be implemented. However, any of the bits may contain a hardwired value of zero, indicating reads to the corresponding counter will cause an exception when V=1. Hence, they are effectively **WARL** fields.

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

The `htimedelta` CSR is a read/write register that contains the delta between the value of the `time` CSR and the value returned in VS-mode or VU-mode. That is, reading the `time` CSR in VS or VU mode returns the sum of the contents of `htimedelta` and the actual value of `time`.

Because overflow is ignored when summing `htimedelta` *and* `time`*, large values of* `htimedelta` *may be used to represent negative time offsets.*

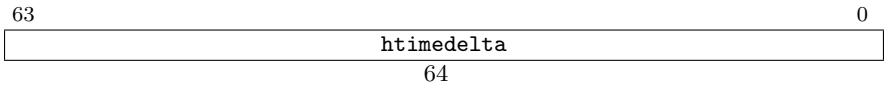


Figure 5.14: Hypervisor time delta register, HSXLEN=64.

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

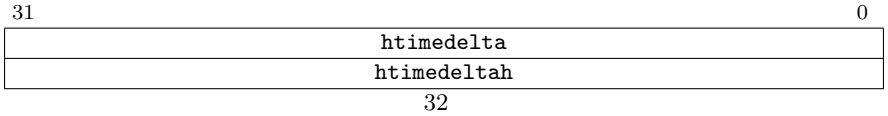


Figure 5.15: Hypervisor time delta registers, HSXLEN=32.

5.2.7 Hypervisor Trap Value Register (`htval`)

The `htval` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.16. When a trap is taken into HS-mode, `htval` is written with additional exception-specific information, alongside `stval`, to assist software in handling the trap.

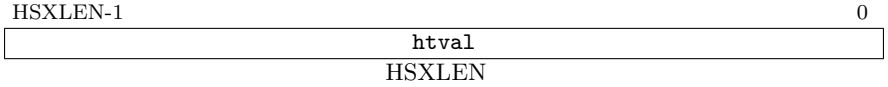


Figure 5.16: Hypervisor trap value register (`htval`).

When a guest-page-fault trap is taken into HS-mode, `htval` is written with either zero or the guest physical address that faulted, shifted right by 2 bits. For other traps, `htval` is set to zero, but a future standard or extension may redefine `htval`'s setting for other traps.

A guest-page fault may arise due to an implicit memory access during first-stage (VS-stage) address translation, in which case a guest physical address written to `htval` is that of the implicit memory access that faulted—for example, the address of a VS-level page table entry that could not be read.

(The guest physical address corresponding to the original virtual address is unknown when VS-stage translation fails to complete.) Additional information is provided in CSR `htinst` to disambiguate such situations.

Otherwise, for misaligned loads and stores that cause guest-page faults, a nonzero guest physical address in `htval` corresponds to the faulting portion of the access as indicated by the virtual address in `stval`. For instruction guest-page faults on systems with variable-length instructions, a nonzero `htval` corresponds to the faulting portion of the instruction as indicated by the virtual address in `stval`.

A guest physical address written to `htval` *is shifted right by 2 bits to accommodate addresses wider than the current XLEN. For RV32, the hypervisor extension permits guest physical addresses as wide as 34 bits, and* `htval` *reports bits 33:2 of the address. This shift-by-2 encoding of guest physical addresses matches the encoding of physical addresses in PMP address registers (Section 3.7) and in page table entries (Sections 4.3, 4.4, and 4.5).*

If the least-significant two bits of a faulting guest physical address are needed, these bits are ordinarily the same as the least-significant two bits of the faulting virtual address in `stval`*. For faults due to implicit memory accesses for VS-stage address translation, the least-significant two bits are instead zeros. These cases can be distinguished using the value provided in register* `htinst`*.*

`htval` is a **WARL** register that must be able to hold zero and may be capable of holding only an arbitrary subset of other 2-bit-shifted guest physical addresses, if any.

Unless it has reason to assume otherwise (such as a platform standard), software that writes a value to `htval` *should read back from* `htval` *to confirm the stored value.*

5.2.8 Hypervisor Trap Instruction Register (`htinst`)

The `htinst` register is an HSXLEN-bit read/write register formatted as shown in Figure 5.17. When a trap is taken into HS-mode, `htinst` is written with a value that, if nonzero, provides information about the instruction that trapped, to assist software in handling the trap. The values that may be written to `htinst` on a trap are documented in Section 5.6.3.

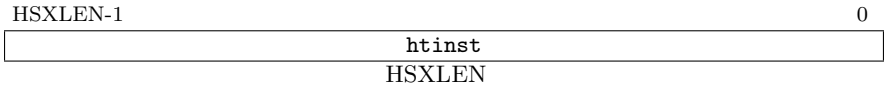


Figure 5.17: Hypervisor trap instruction register (`htinst`).

`htinst` is a **WARL** register that need only be able to hold the values that the implementation may automatically write to it on a trap.

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

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

Similar to CSR `satp`, this register holds the physical page number (PPN) of the guest-physical root page table; a virtual machine identifier (VMID), which facilitates address-translation fences on a per-virtual-machine basis; and the MODE field, which selects the address-translation scheme for guest physical addresses. When `mstatus`.TVM=1, attempts to read or write `hgatp` while executing in HS-mode will raise an illegal instruction exception.

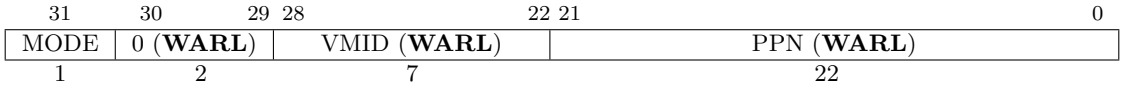


Figure 5.18: RV32 Hypervisor guest address translation and protection register `hgatp`.

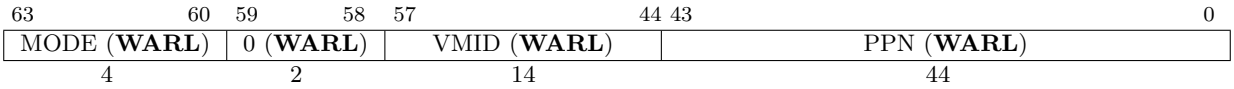


Figure 5.19: RV64 Hypervisor guest address translation and protection register `hgatp`, for MODE values Bare, Sv39x4, and Sv48x4.

Table 5.3 shows the encodings of the MODE field for RV32 and RV64. When MODE=Bare, guest physical addresses are equal to supervisor physical addresses, and there is no further memory protection for a guest virtual machine beyond the physical memory protection scheme described in Section 3.7. In this case, the remaining fields in `hgatp` must be set to zeros.

For RV32, the only other valid setting for MODE is Sv32x4, which is a modification of the usual Sv32 paged virtual-memory scheme, extended to support 34-bit guest physical addresses. For RV64, modes Sv39x4 and Sv48x4 are defined as modifications of the Sv39 and Sv48 paged virtual-memory schemes. All of these paged virtual-memory schemes are described in Section 5.5.1. An additional RV64 scheme, Sv57x4, may be defined in a later version of this specification.

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

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

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39x4	Page-based 41-bit virtual addressing (2-bit extension of Sv39).
9	Sv48x4	Page-based 50-bit virtual addressing (2-bit extension of Sv48).
10	*Sv57x4*	*Reserved for page-based 59-bit virtual addressing.*
11–15	—	*Reserved*

Table 5.3: Encoding of `hgatp` MODE field.

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

A write to `hgatp` with an unsupported MODE value is not ignored as it is for `satp`. Instead, the fields of `hgatp` are **WARL** in the normal way, when so indicated.

As explained in Section 5.5.1, for the paged virtual-memory schemes (Sv32x4, Sv39x4, and Sv48x4), the root page table is 16 KiB and must be aligned to a 16-KiB boundary. In these modes, the lowest two bits of the physical page number (PPN) in `hgatp` always read as zeros. An implementation that supports only the defined paged virtual-memory schemes and/or Bare may hardwire PPN[1:0] to zero.

The number of VMID bits is UNSPECIFIED and may be zero. The number of implemented VMID bits, termed *VMIDLEN*, may be determined by writing one to every bit position in the VMID field, then reading back the value in `hgatp` to see which bit positions in the VMID field hold a one. The least-significant bits of VMID are implemented first: that is, if VMIDLEN > 0, VMID[VMIDLEN-1:0] is writable. The maximal value of VMIDLEN, termed VMIDMAX, is 7 for Sv32x4 or 14 for Sv39x4 and Sv48x4.

Note that writing `hgatp` does not imply any ordering constraints between page-table updates and subsequent G-stage address translations. If the new virtual machine's guest physical page tables have been modified, it may be necessary to execute an HFENCE.GVMA instruction (see Section 5.3.2) before or after writing `hgatp`.

5.2.10 Virtual Supervisor Status Register (`vsstatus`)

The `vsstatus` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `sstatus`, formatted as shown in Figure 5.20 when VSXLEN=32 and Figure 5.21 when VSXLEN=64. When V=1, `vsstatus` substitutes for the usual `sstatus`, so instructions that normally read or modify `sstatus` actually access `vsstatus` instead.

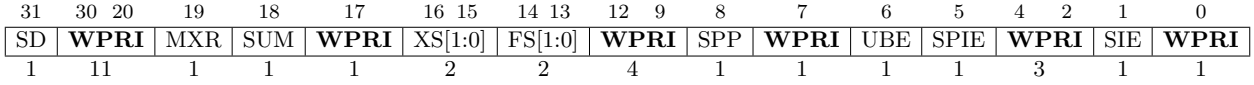


Figure 5.20: Virtual supervisor status register (`vsstatus`) for RV32.

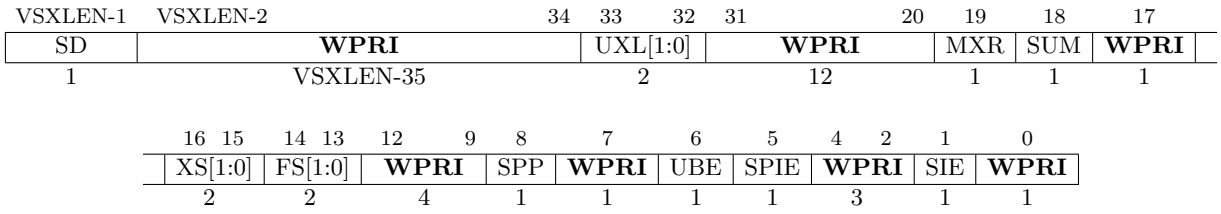


Figure 5.21: Virtual supervisor status register (`vsstatus`) for RV64.

The UXL field controls the effective XLEN for VU-mode, which may differ from the XLEN for VS-mode (VSXLEN). When VSXLEN=32, the UXL field does not exist, and VU-mode XLEN=32. When VSXLEN=64, UXL is a **WARL** field that is encoded the same as the MXL field of `misa`,

shown in Table 3.1 on page 16. In particular, an implementation may make UXL be a read-only copy of field VSXL of `hstatus`, forcing VU-mode XLEN=VSXLEN.

If VSXLEN is changed from 32 to a wider width, and if field UXL is not restricted to a single value, it gets the value corresponding to the widest supported width not wider than the new VSXLEN.

When V=1, both `vsstatus`.FS and the HS-level `sstatus`.FS are in effect. Attempts to execute a floating-point instruction when either field is 0 (Off) raise an illegal-instruction exception. Modifying the floating-point state when V=1 causes both fields to be set to 3 (Dirty).

For a hypervisor to benefit from the extension context status, it must have its own copy in the HS-level `sstatus`*, maintained independently of a guest OS running in VS-mode. While a version of the extension context status obviously must exist in* `vsstatus` *for VS-mode, a hypervisor cannot rely on this version being maintained correctly, given that VS-level software can change* `vsstatus`*.FS arbitrarily. If the HS-level* `sstatus`*.FS were not independently active and maintained by the hardware in parallel with* `vsstatus`*.FS while V=1, hypervisors would always be forced to conservatively swap all floating-point state when context-switching between virtual machines.*

Read-only fields SD and XS summarize the extension context status as it is visible to VS-mode only. For example, the value of the HS-level `sstatus`.FS does not affect `vsstatus`.SD.

An implementation may make field UBE be a read-only copy of `hstatus`.VSBE.

When V=0, `vsstatus` does not directly affect the behavior of the machine, unless a virtual-machine load/store (HLV, HLVX, or HSV) or the MPRV feature in the `mstatus` register is used to execute a load or store *as though* V=1.

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

The `vsip` and `vsie` registers are VSXLEN-bit read/write registers that are VS-mode's versions of supervisor CSRs `sip` and `sie`, formatted as shown in Figures 5.22 and 5.23 respectively. When V=1, `vsip` and `vsie` substitute for the usual `sip` and `sie`, so instructions that normally read or modify `sip`/`sie` actually access `vsip`/`vsie` instead. However, interrupts directed to HS-level continue to be indicated in the HS-level `sip` register, not in `vsip`, when V=1.

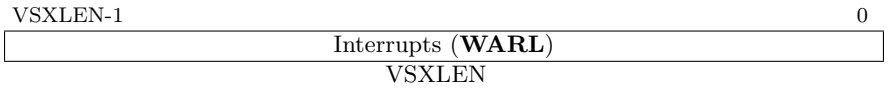


Figure 5.22: Virtual supervisor interrupt-pending register (`vsip`).

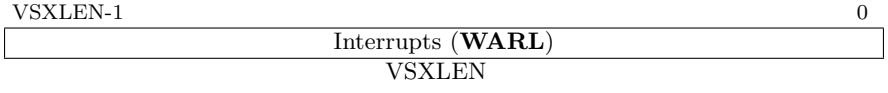


Figure 5.23: Virtual supervisor interrupt-enable register (`vsie`).

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

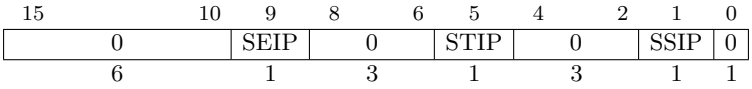


Figure 5.24: Standard portion (bits 15:0) of `vsip`.

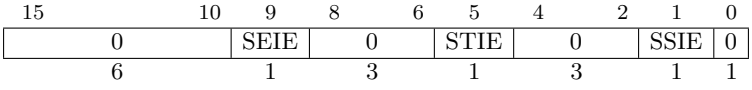


Figure 5.25: Standard portion (bits 15:0) of `vsie`.

When bit 10 of `hideleg` is zero, `vsip`.SEIP and `vsie`.SEIE are read-only zeros. Else, `vsip`.SEIP and `vsie`.SEIE are aliases of `hip`.VSEIP and `hie`.VSEIE.

When bit 6 of `hideleg` is zero, `vsip`.STIP and `vsie`.STIE are read-only zeros. Else, `vsip`.STIP and `vsie`.STIE are aliases of `hip`.VSTIP and `hie`.VSTIE.

When bit 2 of `hideleg` is zero, `vsip`.SSIP and `vsie`.SSIE are read-only zeros. Else, `vsip`.SSIP and `vsie`.SSIE are aliases of `hip`.VSSIP and `hie`.VSSIE.

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

The `vstvec` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `stvec`, formatted as shown in Figure 5.26. When V=1, `vstvec` substitutes for the usual `stvec`, so instructions that normally read or modify `stvec` actually access `vstvec` instead. When V=0, `vstvec` does not directly affect the behavior of the machine.

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

5.2.13 Virtual Supervisor Scratch Register (`vsscratch`)

The `vsscratch` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `sscratch`, formatted as shown in Figure 5.27. When V=1, `vsscratch` substitutes for the usual `sscratch`, so instructions that normally read or modify `sscratch` actually access `vsscratch` instead. The contents of `vsscratch` never directly affect the behavior of the machine.

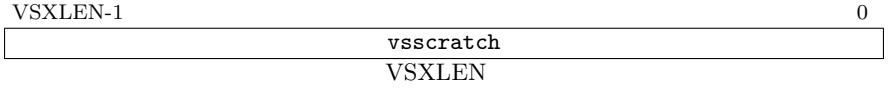


Figure 5.27: Virtual supervisor scratch register (`vsscratch`).

5.2.14 Virtual Supervisor Exception Program Counter (vsepc)

The vsepc register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register sepc, formatted as shown in Figure 5.28. When V=1, vsepc substitutes for the usual sepc, so instructions that normally read or modify sepc actually access vsepc instead. When V=0, vsepc does not directly affect the behavior of the machine.

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

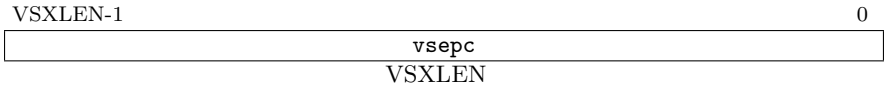


Figure 5.28: Virtual supervisor exception program counter (vsepc).

5.2.15 Virtual Supervisor Cause Register (vscause)

The vscause register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register scause, formatted as shown in Figure 5.29. When V=1, vscause substitutes for the usual scause, so instructions that normally read or modify scause actually access vscause instead. When V=0, vscause does not directly affect the behavior of the machine.

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

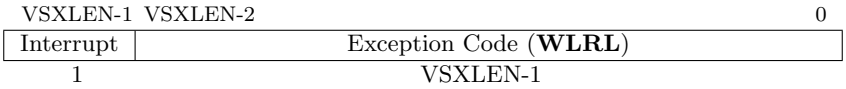


Figure 5.29: Virtual supervisor cause register (vscause).

5.2.16 Virtual Supervisor Trap Value Register (vstval)

The vstval register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register stval, formatted as shown in Figure 5.30. When V=1, vstval substitutes for the usual stval, so instructions that normally read or modify stval actually access vstval instead. When V=0, vstval does not directly affect the behavior of the machine.

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

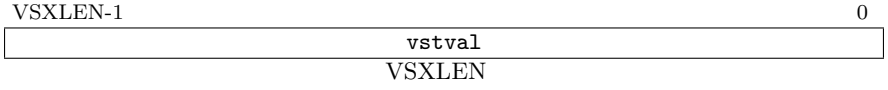


Figure 5.30: Virtual supervisor trap value register (vstval).

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

The `vsatp` register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register `satp`, formatted as shown in Figure 5.31 for VSXLEN=32 and Figure 5.32 for VSXLEN=64. When V=1, `vsatp` substitutes for the usual `satp`, so instructions that normally read or modify `satp` actually access `vsatp` instead. `vsatp` controls VS-stage address translation, the first stage of two-stage translation for guest virtual addresses (see Section 5.5).

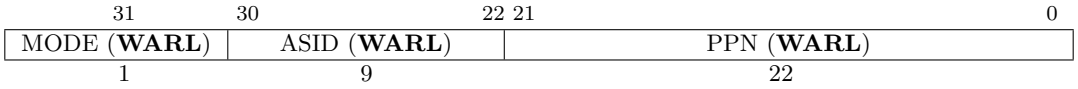


Figure 5.31: RV32 virtual supervisor address translation and protection register `vsatp`.

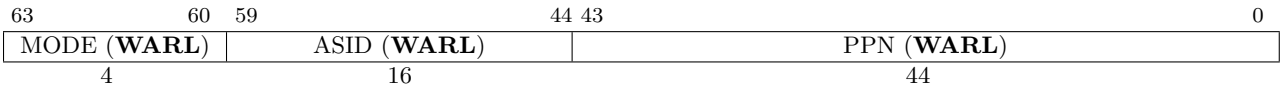


Figure 5.32: RV64 virtual supervisor address translation and protection register `vsatp`, for MODE values Bare, Sv39, and Sv48.

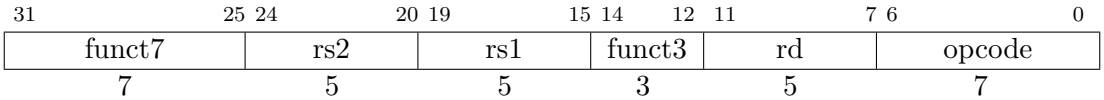
When V=0, a write to `vsatp` with an unsupported MODE value is not ignored as it is for `satp`. Instead, the fields of `vsatp` are **WARL** in the normal way.

When V=0, `vsatp` does not directly affect the behavior of the machine, unless a virtual-machine load/store (HLV, HLVX, or HSV) or the MPRV feature in the `mstatus` register is used to execute a load or store *as though* V=1.

5.3 Hypervisor Instructions

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

5.3.1 Hypervisor Virtual-Machine Load and Store Instructions



The hypervisor virtual-machine load and store instructions are valid only in M-mode or HS-mode, or in U-mode when `hstatus`.HU=1. Each instruction performs an explicit memory access as though V=1; i.e., with the address translation and protection, and the endianness, that apply to memory accesses in either VS-mode or VU-mode. Field SPVP of `hstatus` controls the privilege level of the

access. The explicit memory access is done as though in VU-mode when SPVP=0, and as though in VS-mode when SPVP=1. As usual when V=1, two-stage address translation is applied, and the HS-level `sstatus`.SUM is ignored. HS-level `sstatus`.MXR makes execute-only pages readable for both stages of address translation (VS-stage and G-stage), whereas `vsstatus`.MXR affects only the first translation stage (VS-stage).

For every RV32I or RV64I load instruction, LB, LBU, LH, LHU, LW, LWU, and LD, there is a corresponding virtual-machine load instruction: HLV.B, HLV.BU, HLV.H, HLV.HU, HLV.W, HLV.WU, and HLV.D. For every RV32I or RV64I store instruction, SB, SH, SW, and SD, there is a corresponding virtual-machine store instruction: HSV.B, HSV.H, HSV.W, and HSV.D. Instructions HLV.WU, HLV.D, and HSV.D are not valid for RV32, of course.

Instructions HLVX.HU and HLVX.WU are the same as HLV.HU and HLV.WU, except that *execute* permission takes the place of *read* permission during address translation. That is, the memory being read must be executable in both stages of address translation, but read permission is not required. For the supervisor physical address that results from address translation, the supervisor physical memory attributes must grant both *execute* and *read* permissions. (The *supervisor physical memory attributes* are the machine's physical memory attributes as modified by physical memory protection, Section 3.7, for supervisor level.)

HLVX cannot override machine-level physical memory protection (PMP), so attempting to read memory that PMP designates as execute-only still results in an access-fault exception.

HLVX.WU is valid for RV32, even though LWU and HLV.WU are not. (For RV32, HLVX.WU can be considered a variant of HLV.W, as sign extension is irrelevant for 32-bit values.)

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

5.3.2 Hypervisor Memory-Management Fence Instructions

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

The hypervisor memory-management fence instructions, HFENCE.VVMA and HFENCE.GVMA, perform a function similar to SFENCE.VMA (Section 4.2.1), except applying to the VS-level memory-management data structures controlled by CSR `vsatp` (HFENCE.VVMA) or the guest-physical memory-management data structures controlled by CSR `hgatp` (HFENCE.GVMA). Instruction SFENCE.VMA applies only to the memory-management data structures controlled by the current `satp` (either the HS-level `satp` when V=0 or `vsatp` when V=1).

HFENCE.VVMA is valid only in M-mode or HS-mode. Its effect is much the same as temporarily entering VS-mode and executing SFENCE.VMA. Executing an HFENCE.VVMA guarantees that any previous stores already visible to the current hart are ordered before all subsequent implicit

reads by that hart of the VS-level memory-management data structures, when those implicit reads are for instructions that

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

Implicit reads need not be ordered when `hgatp`.VMID is different than at the time HFENCE.VVMA executed. If operand *rs1*≠`x0`, it specifies a single guest virtual address, and if operand *rs2*≠`x0`, it specifies a single guest address-space identifier (ASID).

An HFENCE.VVMA instruction applies only to a single virtual machine, identified by the setting of `hgatp`*.VMID when HFENCE.VVMA executes.*

When *rs2*≠`x0`, bits XLEN-1:ASIDMAX of the value held in *rs2* are reserved for future use and should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLEN < ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN of the value held in *rs2*.

Simpler implementations of HFENCE.VVMA can ignore the guest virtual address in rs1 *and the guest ASID value in* rs2*, as well as* `hgatp`*.VMID, and always perform a global fence for the VS-level memory management of all virtual machines, or even a global fence for all memory-management data structures.*

Neither `mstatus`.TVM nor `hstatus`.VTVM causes HFENCE.VVMA to trap.

HFENCE.GVMA is valid only in HS-mode when `mstatus`.TVM=0, or in M-mode (irrespective of `mstatus`.TVM). Executing an HFENCE.GVMA instruction guarantees that any previous stores already visible to the current hart are ordered before all subsequent implicit reads by that hart of guest-physical memory-management data structures done for instructions that follow the HFENCE.GVMA. If operand *rs1*≠`x0`, it specifies a single guest physical address, shifted right by 2 bits, and if operand *rs2*≠`x0`, it specifies a single virtual machine identifier (VMID).

Like for a guest physical address written to `htval` *on a trap, a guest physical address specified in* rs1 *is shifted right by 2 bits to accommodate addresses wider than the current XLEN.*

When *rs2*≠`x0`, bits XLEN-1:VMIDMAX of the value held in *rs2* are reserved for future use and should be zeroed by software and ignored by current implementations. Furthermore, if VMIDLEN < VMIDMAX, the implementation shall ignore bits VMIDMAX-1:VMIDLEN of the value held in *rs2*.

Simpler implementations of HFENCE.GVMA can ignore the guest physical address in rs1 *and the VMID value in* rs2 *and always perform a global fence for the guest-physical memory management of all virtual machines, or even a global fence for all memory-management data structures.*

If `hgatp`.MODE is changed for a given VMID, an HFENCE.GVMA with *rs1*=`x0` (and *rs2* set to either `x0` or the VMID) must be executed to order subsequent guest translations with the MODE change—even if the old MODE or new MODE is Bare.

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

execute HFENCE.GVMA in HS-mode when `mstatus`.TVM=1 also causes an illegal instruction trap.

5.4 Machine-Level CSRs

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

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

The hypervisor extension adds two fields, MPV and GVA, to the machine-level `mstatus` or `mstatush` CSR, and modifies the behavior of several existing `mstatus` fields. Figure 5.33 shows the modified `mstatus` register when the hypervisor extension is implemented and MXLEN=64. When MXLEN=32, the hypervisor extension adds MPV and GVA not to `mstatus` but to `mstatush`, which must exist. Figure 5.34 shows the `mstatush` register when the hypervisor extension is implemented and MXLEN=32.

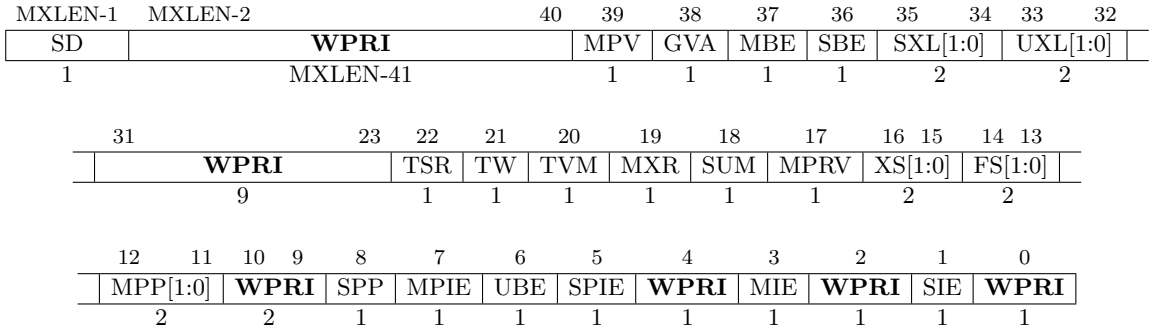


Figure 5.33: Machine status register (`mstatus`) for RV64 when the hypervisor extension is implemented.

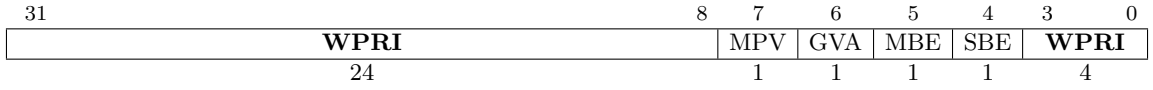


Figure 5.34: Additional machine status register (`mstatush`) for RV32 when the hypervisor extension is implemented. The format of `mstatus` is unchanged for RV32.

The MPV bit (Machine Previous Virtualization Mode) is written by the implementation whenever a trap is taken into M-mode. Just as the MPP bit is set to the (nominal) privilege mode at the time of the trap, the MPV bit is set to the value of the virtualization mode V at the time of the trap. When an MRET instruction is executed, the virtualization mode V is set to MPV, unless MPP=3, in which case V remains 0.

Field GVA (Guest Virtual Address) is written by the implementation whenever a trap is taken into M-mode. For any trap (breakpoint, address misaligned, access fault, page fault, or guest-page fault)

that writes a guest virtual address to `mtval`, GVA is set to 1. For any other trap into M-mode, GVA is set to 0.

The TSR and TVM fields of `mstatus` affect execution only in HS-mode, not in VS-mode. The TW field affects execution in all modes except M-mode.

Setting TVM=1 prevents HS-mode from accessing `hgatp` or executing HFENCE.GVMA, but has no effect on accesses to `vsatp` or instruction HFENCE.VVMA.

The hypervisor extension changes the behavior of the the Modify Privilege field, MPRV, of `mstatus`. When MPRV=0, translation and protection behave as normal. When MPRV=1, explicit memory accesses are translated and protected, and endianness is applied, as though the current virtualization mode were set to MPV and the current nominal privilege mode were set to MPP. Table 5.4 enumerates the cases.

MPRV	MPV	MPP	Effect
0	–	–	Normal access; current privilege mode applies.
1	0	0	U-level access with HS-level translation and protection only.
1	0	1	HS-level access with HS-level translation and protection only.
1	–	3	M-level access with no translation.
1	1	0	VU-level access with two-stage translation and protection. The HS-level MXR bit makes any executable page readable. `vsstatus`.MXR makes readable those pages marked executable at the VS translation stage, but only if readable at the guest-physical translation stage.
1	1	1	VS-level access with two-stage translation and protection. The HS-level MXR bit makes any executable page readable. `vsstatus`.MXR makes readable those pages marked executable at the VS translation stage, but only if readable at the guest-physical translation stage. `vsstatus`.SUM applies instead of the HS-level SUM bit.

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

MPRV does not affect the virtual-machine load/store instructions, HLV, HLVX, and HSV. The explicit loads and stores of these instructions always act as though V=1 and the nominal privilege mode were `hstatus`.SPVP, overriding MPRV.

The `mstatus` register is a superset of the HS-level `sstatus` register but is not a superset of `vsstatus`.

5.4.2 Machine Interrupt Delegation Register (`mideleg`)

When the hypervisor extension is implemented, bits 10, 6, and 2 of `mideleg` (corresponding to the standard VS-level interrupts) are each hardwired to one. Furthermore, if any guest external interrupts are implemented (GEILEN is nonzero), bit 12 of `mideleg` (corresponding to supervisor-level guest external interrupts) is also hardwired to one. VS-level interrupts and guest external interrupts are always delegated past M-mode to HS-mode.

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

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

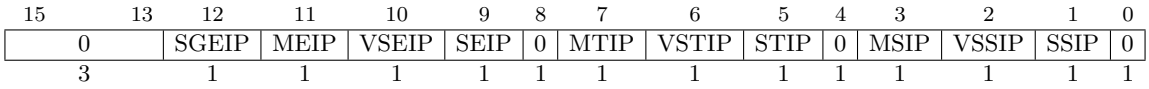


Figure 5.35: Standard portion (bits 15:0) of `mip`.

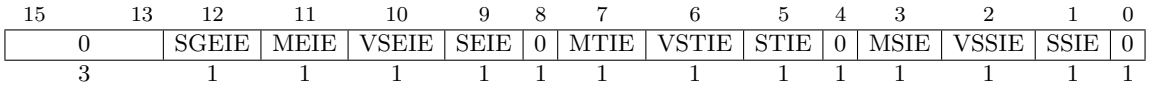


Figure 5.36: Standard portion (bits 15:0) of `mie`.

Bits SGEIP, VSEIP, VSTIP, and VSSIP in `mip` are aliases for the same bits in hypervisor CSR `hip`, while SGEIE, VSEIE, VSTIE, and VSSIE in `mie` are aliases for the same bits in `hie`.

5.4.4 Machine Second Trap Value Register (`mtval2`)

The `mtval2` register is an MXLEN-bit read/write register formatted as shown in Figure 5.37. When a trap is taken into M-mode, `mtval2` is written with additional exception-specific information, alongside `mtval`, to assist software in handling the trap.

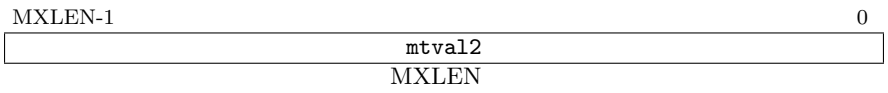


Figure 5.37: Machine second trap value register (`mtval2`).

When a guest-page-fault trap is taken into M-mode, `mtval2` is written with either zero or the guest physical address that faulted, shifted right by 2 bits. For other traps, `mtval2` is set to zero, but a future standard or extension may redefine `mtval2`'s setting for other traps.

If a guest-page fault is due to an implicit memory access during first-stage (VS-stage) address translation, a guest physical address written to `mtval2` is that of the implicit memory access that faulted. Additional information is provided in CSR `mtinst` to disambiguate such situations.

Otherwise, for misaligned loads and stores that cause guest-page faults, a nonzero guest physical address in `mtval2` corresponds to the faulting portion of the access as indicated by the virtual address in `mtval`. For instruction guest-page faults on systems with variable-length instructions, a nonzero `mtval2` corresponds to the faulting portion of the instruction as indicated by the virtual address in `mtval`.

`mtval2` is a **WARL** register that must be able to hold zero and may be capable of holding only an arbitrary subset of other 2-bit-shifted guest physical addresses, if any.

5.4.5 Machine Trap Instruction Register (`mtinst`)

The `mtinst` register is an MXLEN-bit read/write register formatted as shown in Figure 5.38. When a trap is taken into M-mode, `mtinst` is written with a value that, if nonzero, provides information about the instruction that trapped, to assist software in handling the trap. The values that may be written to `mtinst` on a trap are documented in Section 5.6.3.

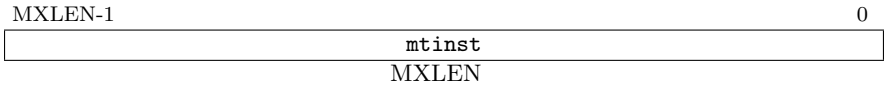


Figure 5.38: Machine trap instruction register (`mtinst`).

`mtinst` is a **WARL** register that need only be able to hold the values that the implementation may automatically write to it on a trap.

5.5 Two-Stage Address Translation

Whenever the current virtualization mode V is 1, two-stage address translation and protection is in effect. For any virtual memory access, the original virtual address is converted in the first stage by VS-level address translation, as controlled by the `vsatp` register, into a *guest physical address*. The guest physical address is then converted in the second stage by guest physical address translation, as controlled by the `hgatp` register, into a supervisor physical address. The two stages are known also as VS-stage and G-stage translation. Although there is no option to disable two-stage address translation when V=1, either stage of translation can be effectively disabled by zeroing the corresponding `vsatp` or `hgatp` register.

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

When V=1, memory accesses that would normally bypass address translation are subject to G-stage address translation alone. This includes memory accesses made in support of VS-stage address translation, such as reads and writes of VS-level page tables.

Machine-level physical memory protection applies to supervisor physical addresses and is in effect regardless of virtualization mode.

5.5.1 Guest Physical Address Translation

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

When the address translation scheme selected by the MODE field of `hgatp` is Bare, guest physical addresses are equal to supervisor physical addresses without modification, and no memory protection applies in the trivial translation of guest physical addresses to supervisor physical addresses.

When `hgatp`.MODE specifies a translation scheme of Sv32x4, Sv39x4, or Sv48x4, G-stage address translation is a variation on the usual page-based virtual address translation scheme of Sv32, Sv39, or Sv48, respectively. In each case, the size of the incoming address is widened by 2 bits (to 34, 41, or 50 bits). To accommodate the 2 extra bits, the root page table (only) is expanded by a factor of four to be 16 KiB instead of the usual 4 KiB. Matching its larger size, the root page table also must be aligned to a 16 KiB boundary instead of the usual 4 KiB page boundary. Except as noted, all other aspects of Sv32, Sv39, or Sv48 are adopted unchanged for G-stage translation. Non-root page tables and all page table entries (PTEs) have the same formats as documented in Sections 4.3, 4.4, and 4.5.

For Sv32x4, an incoming guest physical address is partitioned into a virtual page number (VPN) and page offset as shown in Figure 5.39. This partitioning is identical to that for an Sv32 virtual address as depicted in Figure 4.15 (page 73), except with 2 more bits at the high end in VPN[1]. (Note that the fields of a partitioned guest physical address also correspond one-for-one with the structure that Sv32 assigns to a physical address, depicted in Figure 4.16.)

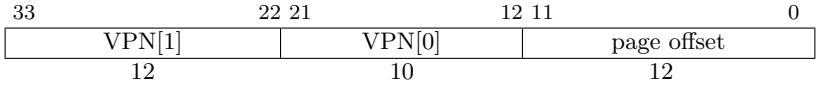


Figure 5.39: Sv32x4 virtual address (guest physical address).

For Sv39x4, an incoming guest physical address is partitioned as shown in Figure 5.40. This partitioning is identical to that for an Sv39 virtual address as depicted in Figure 4.18 (page 77), except with 2 more bits at the high end in VPN[2]. Address bits 63:41 must all be zeros, or else a guest-page-fault exception occurs.

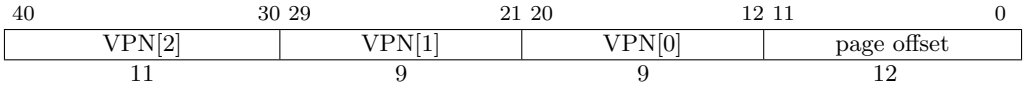


Figure 5.40: Sv39x4 virtual address (guest physical address).

For Sv48x4, an incoming guest physical address is partitioned as shown in Figure 5.41. This partitioning is identical to that for an Sv48 virtual address as depicted in Figure 4.21 (page 78), except with 2 more bits at the high end in VPN[3]. Address bits 63:50 must all be zeros, or else a guest-page-fault exception occurs.

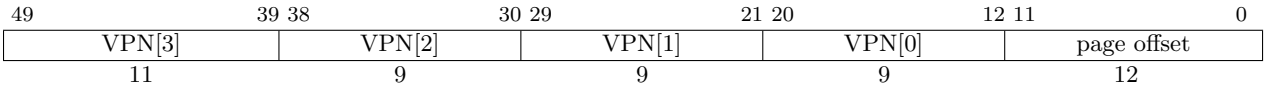


Figure 5.41: Sv48x4 virtual address (guest physical address).

The page-based G-stage address translation scheme for RV32, Sv32x4, is defined to support a 34-bit guest physical address so that an RV32 hypervisor need not be limited in its ability to virtualize real 32-bit RISC-V machines, even those with 33-bit or 34-bit physical addresses. This may include the possibility of a machine virtualizing itself, if it happens to use 33-bit or 34-bit physical addresses. Multiplying the size and alignment of the root page table by a factor of four is the cheapest way to extend Sv32 to cover a 34-bit address. The possible wastage of 12 KiB for

an unnecessarily large root page table is expected to be of negligible consequence for most (maybe all) real uses.

A consistent ability to virtualize machines having as much as four times the physical address space as virtual address space is believed to be of some utility also for RV64. For a machine implementing 39-bit virtual addresses (Sv39), for example, this allows the hypervisor extension to support up to a 41-bit guest physical address space without either necessitating hardware support for 48-bit virtual addresses (Sv48) or falling back to emulating the larger address space using shadow page tables.

The conversion of an Sv32x4, Sv39x4, or Sv48x4 guest physical address is accomplished with the same algorithm used for Sv32, Sv39, or Sv48, as presented in Section 4.3.2, except that:

- in step 1, $a =$ `hgatp`.PPN $\times$ PAGESIZE;
- the current privilege mode is always taken to be U-mode; and
- guest-page-fault exceptions are raised instead of regular page-fault exceptions.

For G-stage address translation, all memory accesses (including those made to access data structures for VS-stage address translation) are considered to be user-level accesses, as though executed in U-mode. Access type permissions—readable, writable, or executable—are checked during G-stage translation the same as for VS-stage translation. For a memory access made to support VS-stage address translation (such as to read/write a VS-level page table), permissions are checked as though for a load or store, not for the original access type. However, any exception is always reported for the original access type (instruction, load, or store/AMO).

The G bit in all G-stage PTEs is reserved for future standard use, should be cleared by software for forward compatibility, and must be ignored by hardware.

G-stage address translation uses the identical format for PTEs as regular address translation, even including the U bit, due to the possibility of sharing some (or all) page tables between G-stage translation and regular HS-level address translation. Regardless of whether this usage will ever become common, we chose not to preclude it.

5.5.2 Guest-Page Faults

Guest-page-fault traps may be delegated from M-mode to HS-mode under the control of CSR `medeleg`, but cannot be delegated to other privilege modes. On a guest-page fault, CSR `mtval` or `stval` is written with the faulting guest virtual address as usual, and `mtval2` or `htval` is written either with zero or with the faulting guest physical address, shifted right by 2 bits. CSR `mtinst` or `htinst` may also be written with information about the faulting instruction or other reason for the access, as explained in Section 5.6.3.

When an instruction fetch or a misaligned memory access straddles a page boundary, two different address translations are involved. When a guest-page fault occurs in such a circumstance, the faulting virtual address written to `mtval`/`stval` is the same as would be required for a regular page fault. Thus, the faulting virtual address may be a page-boundary address that is higher than the instruction's original virtual address, if the byte at that page boundary is among the accessed bytes.

When a guest-page fault is not due to an implicit memory access for VS-stage address translation, a nonzero guest physical address written to `mtval2`/`htval` shall correspond to the exact virtual address written to `mtval`/`stval`.

5.5.3 Memory-Management Fences

The behavior of the SFENCE.VMA instruction is affected by the current virtualization mode V. When V=0, the virtual-address argument is an HS-level virtual address, and the ASID argument is an HS-level ASID. The instruction orders stores only to HS-level address-translation structures with subsequent HS-level address translations.

When V=1, the virtual-address argument to SFENCE.VMA is a guest virtual address within the current virtual machine, and the ASID argument is a VS-level ASID within the current virtual machine. The current virtual machine is identified by the VMID field of CSR `hgatp`, and the effective ASID can be considered to be the combination of this VMID with the VS-level ASID. The SFENCE.VMA instruction orders stores only to the VS-level address-translation structures with subsequent VS-stage address translations for the same virtual machine, i.e., only when `hgatp`.VMID is the same as when the SFENCE.VMA executed.

Hypervisor instructions HFENCE.VVMA and HFENCE.GVMA provide additional memory-management fences to complement SFENCE.VMA. These instructions are described in Section 5.3.2.

Section 3.7.2 discusses the intersection between physical memory protection (PMP) and page-based address translation. It is noted there that, when PMP settings are modified in a manner that affects either the physical memory that holds page tables or the physical memory to which page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. For HS-level address translation, this is accomplished by executing in M-mode an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0`, after the PMP CSRs are written. If G-stage address translation is in use and is not Bare, synchronization with its data structures is also needed. When PMP settings are modified in a manner that affects either the physical memory that holds guest-physical page tables or the physical memory to which guest-physical page tables point, an HFENCE.GVMA instruction with *rs1*=`x0` and *rs2*=`x0` must be executed in M-mode after the PMP CSRs are written. An HFENCE.VVMA instruction is not required.

5.6 Traps

5.6.1 Trap Cause Codes

The hypervisor extension augments the trap cause encoding. Table 5.5 lists the possible M-mode and HS-mode trap cause codes when the hypervisor extension is implemented. Codes are added for VS-level interrupts (interrupts 2, 6, 10), for supervisor-level guest external interrupts (interrupt 12), for virtual instruction exceptions (exception 22), and for guest-page faults (exceptions 20, 21, 23). Furthermore, environment calls from VS-mode are assigned cause 10, whereas those from HS-mode or S-mode use cause 9 as usual.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	Virtual supervisor software interrupt
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	Virtual supervisor timer interrupt
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	Virtual supervisor external interrupt
1	11	Machine external interrupt
1	12	Supervisor guest external interrupt
1	13–15	*Reserved*
1	≥16	*Designated for platform or custom use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode or VU-mode
0	9	Environment call from HS-mode
0	10	Environment call from VS-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–19	*Reserved*
0	20	Instruction guest-page fault
0	21	Load guest-page fault
0	22	Virtual instruction
0	23	Store/AMO guest-page fault
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

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

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

When V=1, a virtual instruction exception (code 22) is normally raised instead of an illegal instruction exception if the attempted instruction is *HS-qualified* but is prevented when V=1 due to insufficient privilege or because the instruction is expressly disabled by a hypervisor CSR such as `hcounteren`. An instruction is *HS-qualified* if it would be valid to execute in HS-mode (for some values of the instruction's register operands), assuming fields TSR and TVM of CSR `mstatus` are both zero.

Special rules apply for CSR instructions that access 32-bit high-half CSRs such as `cycleh` and `htimedeltah`. When V=1 and XLEN>32, an attempt to access a high-half supervisor-level CSR, high-half hypervisor CSR, high-half VS CSR, or high-half unprivileged CSR always raises an illegal instruction exception. And in VS-mode, if the XLEN for VU-mode is greater than 32, an attempt to access a high-half user-level CSR (distinct from an unprivileged CSR) always raises an illegal instruction exception. On the other hand, when V=1 and XLEN=32, an invalid attempt to access a high-half S-level, hypervisor, VS, or unprivileged CSR raises a virtual instruction exception instead of an illegal instruction exception if the same CSR instruction for the partner *low-half* CSR (e.g. `cycle` or `htimedelta`) is HS-qualified. Likewise, in VS-mode, if the XLEN for VU-mode is 32, an invalid attempt to access a high-half user-level CSR raises a virtual instruction exception instead of an illegal instruction exception if the same CSR instruction for the partner low-half CSR is HS-qualified.

The RISC-V Privileged Architecture currently defines no user-level CSRs, but they might be added by a future version of this standard or by an extension.

Specifically, a virtual instruction exception is raised for the following cases, not necessarily a complete list:

- in VS-mode or VU-mode, attempts to access a non-high-half counter CSR when the corresponding bit in `hcounteren` is 0 and the same bit in `mcounteren` is 1;
- in VS-mode or VU-mode, if XLEN=32, attempts to access a high-half counter CSR when the corresponding bit in `hcounteren` is 0 and the same bit in `mcounteren` is 1;
- in VS-mode or VU-mode, attempts to execute a hypervisor instruction (HLV, HLVX, HSV, or HFENCE);
- in VS-mode or VU-mode, attempts to access an implemented non-high-half hypervisor CSR or VS CSR when the same access (read/write) would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VS-mode or VU-mode, if XLEN=32, attempts to access an implemented high-half hypervisor CSR or high-half VS CSR when the same access (read/write) to the CSR's low-half partner would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VU-mode, attempts to execute WFI when `mstatus`.TW=0, or to execute a supervisor instruction (SRET or SFENCE);
- in VU-mode, attempts to access an implemented non-high-half supervisor CSR when the same access (read/write) would be allowed in HS-mode, assuming `mstatus`.TVM=0;

- in VU-mode, if XLEN=32, attempts to access an implemented high-half supervisor CSR when the same access to the CSR's low-half partner would be allowed in HS-mode, assuming `mstatus`.TVM=0;
- in VS-mode, attempts to execute WFI when `hstatus`.VTW=1 and `mstatus`.TW=0, unless the instruction completes within an implementation-specific, bounded time;
- in VS-mode, attempts to execute SRET when `hstatus`.VTSR=1; and
- in VS-mode, attempts to execute an SFENCE instruction or to access `satp`, when `hstatus`.VTVM=1.

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

It is not unusual that hypervisors must emulate the instructions that raise virtual instruction exceptions, to support nested hypervisors or for other reasons. Machine level is expected ordinarily to delegate virtual instruction traps directly to HS-level, whereas illegal instruction traps are likely to be processed first in M-mode before being conditionally delegated (by software) to HS-level. Consequently, virtual instruction traps are expected typically to be handled faster than illegal instruction traps.

When not emulating the trapping instruction, a hypervisor should convert a virtual instruction trap into an illegal instruction exception for the guest virtual machine.

Because TSR and TVM in `mstatus` *are intended to impact only S-mode (HS-mode), they are ignored for determining exceptions in VS-mode.*

5.6.2 Trap Entry

When a trap occurs in HS-mode or U-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode. When a trap occurs in VS-mode or VU-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode, unless further delegated by `hedeleg` or `hideleg`, in which case it goes to VS-mode.

When a trap is taken into M-mode, virtualization mode V gets set to 0, and fields MPV and MPP in `mstatus` (or `mstatush`) are set according to Table 5.6. A trap into M-mode also writes fields GVA, MPIE, and MIE in `mstatus`/`mstatush` and writes CSRs `mepc`, `mcause`, `mtval`, `mtval2`, and `mtinst`.

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

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

When a trap is taken into HS-mode, virtualization mode V is set to 0, and `hstatus`.SPV and `sstatus`.SPP are set according to Table 5.7. If V was 1 before the trap, field SPVP in `hstatus` is set the same as `sstatus`.SPP; otherwise, SPVP is left unchanged. A trap into HS-mode also writes field GVA in `hstatus`, fields SPIE and SIE in `sstatus`, and CSRs `sepc`, `scause`, `stval`, `htval`, and `htinst`.

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

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

When a trap is taken into VS-mode, `vsstatus`.SPP is set according to Table 5.8. Register `hstatus` and the HS-level `sstatus` are not modified, and the virtualization mode V remains 1. A trap into VS-mode also writes fields SPIE and SIE in `vsstatus` and writes CSRs `vsepc`, `vscause`, and `vstval`.

Previous Mode	SPP
VU-mode	0
VS-mode	1

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

5.6.3 Transformed Instruction or Pseudoinstruction for `mtinst` or `htinst`

On any trap into M-mode or HS-mode, one of these values is written automatically into the appropriate trap instruction CSR, `mtinst` or `htinst`:

- zero;
- a transformation of the trapping instruction;
- a custom value (allowed only if the trapping instruction is nonstandard); or
- a special pseudoinstruction.

Except when a pseudoinstruction value is required (described later), the value written to `mtinst` or `htinst` may always be zero, indicating that the hardware is providing no information in the register for this particular trap.

The value written to the trap instruction CSR serves two purposes. The first is to improve the speed of instruction emulation in a trap handler, partly by allowing the handler to skip loading the trapping instruction from memory, and partly by obviating some of the work of decoding and executing the instruction. The second purpose is to supply, via pseudoinstructions, additional information about guest-page-fault exceptions caused by implicit memory accesses done for VS-stage address translation.

A transformation *of the trapping instruction is written instead of simply a copy of the original instruction in order to minimize the burden for hardware yet still provide to a trap handler the*

information needed to emulate the instruction. An implementation may at any time reduce its effort by substituting zero in place of the transformed instruction.

On an interrupt, the value written to the trap instruction register is always zero. On a synchronous exception, if a nonzero value is written, one of the following shall be true about the value:

- Bit 0 is `1`, and replacing bit 1 with `1` makes the value into a valid encoding of a standard instruction.

  In this case, the instruction that trapped is the same kind as indicated by the register value, and the register value is the transformation of the trapping instruction, as defined later. For example, if bits 1:0 are binary `11` and the register value is the encoding of a standard LW (load word) instruction, then the trapping instruction is LW, and the register value is the transformation of the trapping LW instruction.

- Bit 0 is `1`, and replacing bit 1 with `1` makes the value into an instruction encoding that is explicitly designated for a custom instruction (*not* an unused reserved encoding).

  This is a *custom value*. The instruction that trapped is a nonstandard instruction. The interpretation of a custom value is not otherwise specified by this standard.

- The value is one of the special pseudoinstructions defined later, all of which have bits 1:0 equal to `00`.

These three cases exclude a large number of other possible values, such as all those having bits 1:0 equal to binary `10`. A future standard or extension may define additional cases, thus allowing values that are currently excluded. Software may safely treat an unrecognized value in a trap instruction register the same as zero.

To be forward-compatible with future revisions of this standard, software that interprets a nonzero value from `mtinst` *or* `htinst` *must fully verify that the value conforms to one of the cases listed above. For instance, for RV64, discovering that bits 6:0 of* `mtinst` *are* `0000011` *and bits 14:12 are* `010` *is not sufficient to establish that the first case applies and the trapping instruction is a standard LW instruction; rather, software must also confirm that bits 63:32 of* `mtinst` *are all zeros. A future standard might define new values for 64-bit* `mtinst` *that are nonzero in bits 63:32 yet may coincidentally have in bits 31:0 the same bit patterns as standard RV64 instructions.*

Unlike for standard instructions, there is no requirement that the instruction encoding of a custom value be of the same "kind" as the instruction that trapped (or even have any correlation with the trapping instruction).

Table 5.9 shows the values that may be automatically written to the trap instruction register for each standard exception cause. For exceptions that prevent the fetching of an instruction, only zero or a pseudoinstruction value may be written. A custom value may be automatically written only if the instruction that traps is nonstandard. A future standard or extension may permit other values to be written, chosen from the set of allowed values established earlier.

As enumerated in the table, a synchronous exception may write to the trap instruction register a standard transformation of the trapping instruction only for exceptions that arise from explicit memory accesses (from loads, stores, and AMO instructions). Accordingly, standard transformations are currently defined only for these memory-access instructions. If a synchronous trap occurs

Exception	Zero	Transformed Standard Instruction	Custom Value	Pseudo-instruction Value
Instruction address misaligned	Yes	No	Yes	No
Instruction access fault	Yes	No	No	No
Illegal instruction	Yes	No	No	No
Breakpoint	Yes	No	Yes	No
Virtual instruction	Yes	No	Yes	No
Load address misaligned	Yes	Yes	Yes	No
Load access fault	Yes	Yes	Yes	No
Store/AMO address misaligned	Yes	Yes	Yes	No
Store/AMO access fault	Yes	Yes	Yes	No
Environment call	Yes	No	Yes	No
Instruction page fault	Yes	No	No	No
Load page fault	Yes	Yes	Yes	No
Store/AMO page fault	Yes	Yes	Yes	No
Instruction guest-page fault	Yes	No	No	Yes
Load guest-page fault	Yes	Yes	Yes	Yes
Store/AMO guest-page fault	Yes	Yes	Yes	Yes

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

for a standard instruction for which no transformation has been defined, the trap instruction register shall be written with zero (or, under certain circumstances, with a special pseudoinstruction value).

For a standard load instruction that is not a compressed instruction and is one of LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, or FLQ, the transformed instruction has the format shown in Figure 5.42.

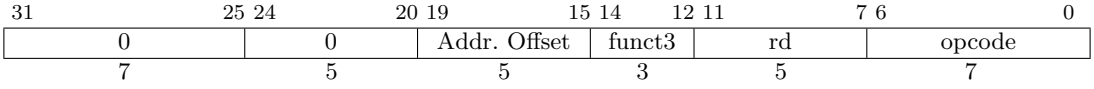


Figure 5.42: Transformed noncompressed load instruction (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, or FLQ). Fields funct3, rd, and opcode are the same as the trapping load instruction.

For a standard store instruction that is not a compressed instruction and is one of SB, SH, SW, SD, FSW, FSD, or FSQ, the transformed instruction has the format shown in Figure 5.43.

For a standard atomic instruction (load-reserved, store-conditional, or AMO instruction), the transformed instruction has the format shown in Figure 5.44.

For a standard virtual-machine load/store instruction (HLV, HLVX, or HSV), the transformed instruction has the format shown in Figure 5.45.

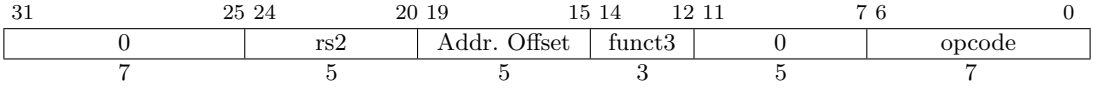


Figure 5.43: Transformed noncompressed store instruction (SB, SH, SW, SD, FSW, FSD, or FSQ). Fields rs2, funct3, and opcode are the same as the trapping store instruction.

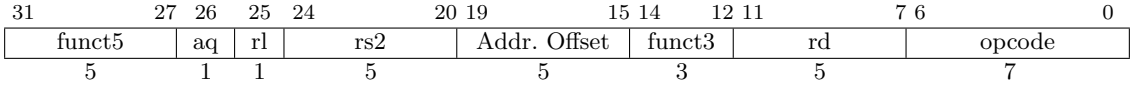


Figure 5.44: Transformed atomic instruction (load-reserved, store-conditional, or AMO instruction). All fields are the same as the trapping instruction except bits 19:15, Addr. Offset.

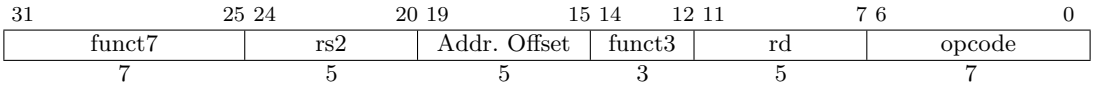


Figure 5.45: Transformed virtual-machine load/store instruction (HLV, HLVX, HSV). All fields are the same as the trapping instruction except bits 19:15, Addr. Offset.

In all the transformed instructions above, the Addr. Offset field that replaces the instruction's rs1 field in bits 19:15 is the positive difference between the faulting virtual address (written to `mtval` or `stval`) and the original virtual address. This difference can be nonzero only for a misaligned memory access. Note also that, for basic loads and stores, the transformations replace the instruction's immediate offset fields with zero.

For a standard compressed instruction (16-bit size), the transformed instruction is found as follows:

1. Expand the compressed instruction to its 32-bit equivalent.
2. Transform the 32-bit equivalent instruction.
3. Replace bit 1 with a `0`.

Bits 1:0 of a transformed standard instruction will be binary `01` if the trapping instruction is compressed and `11` if not.

In decoding the contents of `mtinst` *or* `htinst`*, once software has determined that the register contains the encoding of a standard basic load (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, or FLQ) or basic store (SB, SH, SW, SD, FSW, FSD, or FSQ), it is not necessary to confirm also that the immediate offset fields (31:25, and 24:20 or 11:7) are zeros. The knowledge that the register's value is the encoding of a basic load/store is sufficient to prove that the trapping instruction is of the same kind.*

A future version of this standard may add information to the fields that are currently zeros. However, for backwards compatiblity, any such information will be for performance purposes only and can safely be ignored.

For guest-page faults, the trap instruction register is written with a special pseudoinstruction value if: (a) the fault is caused by an implicit memory access for VS-stage address translation, and (b) a nonzero value (the faulting guest physical address) is written to `mtval2` or `htval`. If both conditions are met, the value written to `mtinst` or `htinst` must be taken from Table 5.10; zero is not allowed.

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

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

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

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

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

A *write* pseudoinstruction (0x00002020 or 0x00003020) is used for the case that the machine is attempting automatically to update bits A and/or D in VS-level page tables. All other implicit memory accesses for VS-stage address translation will be reads. If a machine never automatically updates bits A or D in VS-level page tables (leaving this to software), the *write* case will never arise. The fact that such a page table update must actually be atomic, not just a simple write, is ignored for the pseudoinstruction.

If the conditions that necessitate a pseudoinstruction value can ever occur for M-mode, then `mtinst` *cannot be hardwired entirely to zero; and likewise for HS-mode and* `htinst`*. However, in that case, the trap instruction registers may minimally support only values 0 and* 0x00002000 *or* 0x00003000*, and possibly* 0x00002020 *or* 0x00003020*, requiring as few as one or two flip-flops in hardware, per register.*

There is no harm here in ignoring the atomicity requirement for page table updates, because a hypervisor is not expected in these circumstances to emulate an implicit memory access that fails. Rather, the hypervisor is given enough information about the faulting access to be able to make the memory accessible (e.g. by restoring a missing page of virtual memory) before resuming execution by retrying the faulting instruction.

5.6.4 Trap Return

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

The SRET instruction is used to return from a trap taken into HS-mode or VS-mode. Its behavior depends on the current virtualization mode.

When executed in M-mode or HS-mode (i.e., V=0), SRET first determines what the new privilege mode will be according to the values in `hstatus`.SPV and `sstatus`.SPP, as encoded in Table 5.7. SRET then sets `hstatus`.SPV=0, and in `sstatus` sets SPP=0, SIE=SPIE, and SPIE=1. Lastly, SRET sets the privilege mode as previously determined, and sets `pc`=`sepc`.

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

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							
0001000		00010	00000	000	00000	1110011	SRET
0011000		00010	00000	000	00000	1110011	MRET
Interrupt-Management Instructions							
0001000		00101	00000	000	00000	1110011	WFI
Supervisor Memory-Management Instructions							
0001001		rs2	rs1	000	00000	1110011	SFENCE.VMA
Hypervisor Memory-Management Instructions							
0010001		rs2	rs1	000	00000	1110011	HFENCE.VVMA
0110001		rs2	rs1	000	00000	1110011	HFENCE.GVMA
Hypervisor Virtual-Machine Load and Store Instructions							
0110000		00000	rs1	100	rd	1110011	HLV.B
0110000		00001	rs1	100	rd	1110011	HLV.BU
0110010		00000	rs1	100	rd	1110011	HLV.H
0110010		00001	rs1	100	rd	1110011	HLV.HU
0110010		00011	rs1	100	rd	1110011	HLVX.HU
0110100		00000	rs1	100	rd	1110011	HLV.W
0110100		00011	rs1	100	rd	1110011	HLVX.WU
0110001		rs2	rs1	100	00000	1110011	HSV.B
0110011		rs2	rs1	100	00000	1110011	HSV.H
0110101		rs2	rs1	100	00000	1110011	HSV.W
Hypervisor Virtual-Machine Load and Store Instructions, RV64 only							
0110100		00001	rs1	100	rd	1110011	HLV.WU
0110110		00000	rs1	100	rd	1110011	HLV.D
0110111		rs2	rs1	100	00000	1110011	HSV.D

Table 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.

Bibliography

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

[2] 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.