The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.7

Andrew Waterman
Yunsup Lee
Rimas Avizienis
David A. Patterson
Krste Asanović

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-49
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html

May 9, 2015

Copyright © 2015, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

The RISC-V Instruction Set Manual
Volume II: Privileged Architecture

Privileged Architecture Version 1.7:
Document Version 1.7:

Warning! This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Andrew Waterman, Yunsup Lee, Rimas Avižienis, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|rimas|pattrsn|krste}@eecs.berkeley.edu
May 9, 2015

Contents

1 Introduction **1**

1.1 RISC-V Hardware Platform Terminology . . . 1

1.2 RISC-V Privileged Software Stack Terminology . . . 2

1.3 Privilege Levels . . . 4

2 Control and Status Registers (CSRs) **7**

2.1 Instructions to access CSRs . . . 7

2.2 CSR Address Mapping Conventions . . . 9

2.3 CSR Listing . . . 9

3 Machine-Level ISA **15**

3.1 Machine-Level CSRs . . . 15

3.1.1 CPU ID Register `mcpuid` . . . 15

3.1.2 Implementation ID Register `mimpid` . . . 16

3.1.3 Hart ID Register `mhartid` . . . 17

3.1.4 Machine Status Register (`mstatus`) . . . 18

3.1.5 Privilege and Global Interrupt-Enable Stack in `mstatus` register . . . 18

3.1.6 Virtualization Management Field in `mstatus` Register . . . 19

3.1.7 Memory Privilege in `mstatus` Register . . . 20

3.1.8 Extension Context Status in `mstatus` Register . . . 20

3.1.9 Machine Trap Vector Base Address Register (`mtvec`) . . . 23

3.1.10 Machine Trap Delegation Register (`mtdeleg`) . . . 24

3.1.11 Machine Interrupt Registers (`mip` and `mie`) . . . 25

3.1.12 Machine Timer Registers (`mtime`, `mtimecmp`) . . . 26

3.1.13 Machine Scratch Register (`mscratch`) . . . 27

3.1.14 Machine Exception Program Counter (`mepc`) . . . 28

3.1.15 Machine Cause Register (`mcause`) . . . 28

3.1.16 Machine Bad Address (`mbadaddr`) Register . . . 29

3.2 Machine-Mode Privileged Instructions . . . 29

3.2.1 Instructions to Change Privilege Level . . . 29

3.2.2 Trap Redirection Instructions . . . 30

3.2.3 Wait for Interrupt . . . 31

3.3 Physical Memory Attributes . . . 32

3.4 Physical Memory Access Control . . . 32

3.5 Mbare addressing environment . . . 33

3.6 Base-and-Bound environments . . . 33

3.6.1 Mbb: Single Base-and-Bound registers (`mbase`, `mbound`) . . . 33

3.6.2 Mbbid: Separate Instruction and Data Base-and-Bound registers . . . 34

4 Supervisor-Level ISA **37**

4.1 Supervisor CSRs . . . 37

4.1.1 Supervisor Status Register (`sstatus`) . . . 38

4.1.2 Memory Privilege in `sstatus` Register . . . 38

4.1.3 Supervisor Interrupt Registers (`sip` and `sie`) . . . 38

4.1.4 Supervisor Timer Registers (`stime`, `stimecmp`) . . . 39

4.1.5 Supervisor Scratch Register (`sscratch`) . . . 39

4.1.6 Supervisor Exception Program Counter (`sepc`) . . . 40

4.1.7 Supervisor Cause Register (`scause`) . . . 40

4.1.8 Supervisor Bad Address (`sbadaddr`) Register . . . 40

4.1.9 Supervisor Page-Table Base Register (`sptbr`) . . . 41

4.1.10 Supervisor Address Space ID Register (`sasid`) . . . 42

4.2 Supervisor Instructions . . . 42

4.2.1 Supervisor Memory-Management Fence Instruction . . . 42

4.3 Supervisor Operation in Mbare Environment . . . 43

4.4 Supervisor Operation in Base and Bounds Environments . . . 43

4.5 Sv32: Page-Based 32-bit Virtual-Memory Systems . . . 43

4.5.1 Addressing and Memory Protection . . . 44

4.5.2 Virtual Address Translation Process . . . 45

4.6 Sv39: Page-Based 39-bit Virtual-Memory System . . . 46

4.6.1 Addressing and Memory Protection . . . 46

4.7 Sv48: Page-Based 48-bit Virtual-Memory System . . . 47

4.7.1 Addressing and Memory Protection . . . 47

5 Hypervisor-Level ISA **49**

6 RISC-V Privileged Instruction Set Listings **51**

7 History **53**

7.1 Funding . . . 53

Chapter 1

Introduction

This is a draft of the privileged architecture description document for RISC-V. This version does not match our existing implementations. Feedback welcome. Changes will occur before the final release.

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

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

We briefly note that the entire privileged-level design described in this document could be replaced with an entirely different privileged-level design without changing the user-level ISA, and possibly without even changing the ABI. In particular, this privileged specification was designed to run existing popular operating systems, and so embodies the conventional level-based protection model. Alternate privileged specifications could embody other more flexible protection domain models.

1.1 RISC-V Hardware Platform Terminology

A RISC-V hardware platform can contain one or more RISC-V-compatible processing cores together with other non-RISC-V-compatible cores, fixed-function accelerators, various physical memory structures, I/O devices, and an interconnect structure to allow the components to communicate.

A component is termed a *core* if it contains an independent instruction fetch unit. A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads, or *harts*, through multithreading.

A RISC-V core might have additional specialized instruction set extensions or an added *coprocessor*. We use the term *coprocessor* to refer to a unit that is attached to a RISC-V core and is mostly sequenced by a RISC-V instruction stream, but which contains additional architectural state and instruction set extensions, and possibly some limited autonomy relative to the primary RISC-V instruction stream.

We use the term *accelerator* to refer to either a non-programmable fixed-function unit or a core that can operate autonomously but is specialized for certain tasks. In RISC-V systems, we expect many programmable accelerators will be RISC-V-based cores with specialized instruction set extensions and/or customized coprocessors. An important class of RISC-V accelerators are I/O accelerators, which offload I/O processing tasks from the main application cores.

The system-level organization of a RISC-V hardware platform can range from a single-core microcontroller to a many-thousand-node cluster of shared-memory manycore server nodes. Even small systems-on-a-chip might be structured as a hierarchy of multicomputers and/or multiprocessors to modularize development effort or to provide secure isolation between subsystems.

This document focuses on the privileged architecture visible to each hart (hardware thread) running within a uniprocessor or a shared-memory multiprocessor.

1.2 RISC-V Privileged Software Stack Terminology

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

Figure 1.1 shows some of the possible software stacks that can be supported by the RISC-V architecture. The left-hand side shows a simple system that supports only a single application running on an application execution environment (AEE). The application is coded to run with a particular application binary interface (ABI). The ABI includes the supported user-level ISA plus a set of ABI calls to interact with the AEE. The ABI hides details of the AEE from the application to allow greater flexibility in implementing the AEE. The same ABI could be implemented natively on multiple different host OSs, or could be supported by a user-mode emulation environment running on a machine with a different native ISA.

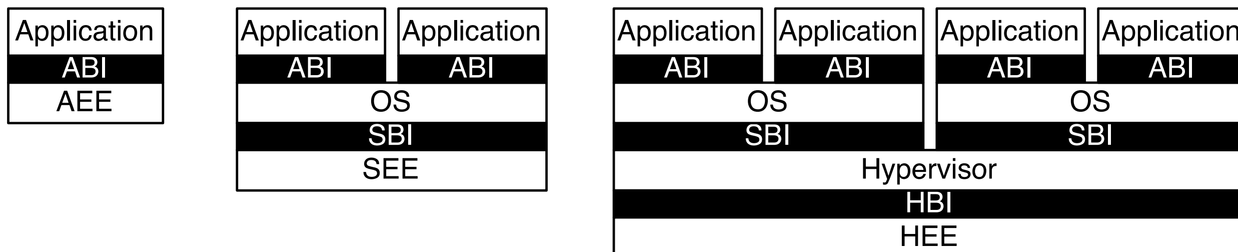


Figure 1.1: Different implementation stacks supporting various forms of privileged execution.

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

The middle configuration shows a conventional operating system (OS) that can support multiprogrammed execution of multiple applications. Each application communicates over an ABI with the OS, which provides the AEE. Just as applications interface with an AEE via an ABI, RISC-V operating systems interface with a supervisor execution environment (SEE) via a supervisor binary interface (SBI). An SBI comprises the user-level and supervisor-level ISA together with a set of SBI function calls. Using a single SBI across all SEE implementations allows a single OS binary

image to run on any SEE. The SEE can be a simple boot loader and BIOS-style IO system in a low-end hardware platform, or a hypervisor-provided virtual machine in a high-end server, or a thin translation layer over a host operating system in an architecture simulation environment.

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

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

The various ABI, SBI, and HBIs are still a work-in-progress, but we anticipate the SBI and HBI to support devices via virtualized device interfaces similar to virtio [3], and to support device discovery. In this manner, only one set of device drivers need be written that can support any OS or hypervisor, and which can also be shared with the boot environment.

Hardware implementations of the RISC-V ISA will generally require additional features beyond the privileged ISA to support the various execution environments (AEE, SEE, or HEE). We separate the features required in a hardware platform from the execution environments using a hardware abstraction layer (HAL), as shown in Figure 1.2. Note that a HAL is not necessarily present in a RISC-V software stack, as an execution environment might be provided purely via software emulation or might have been written directly to a given hardware platform without abstraction.

Later chapters provide details of proposed standard designs for RISC-V hardware platforms.

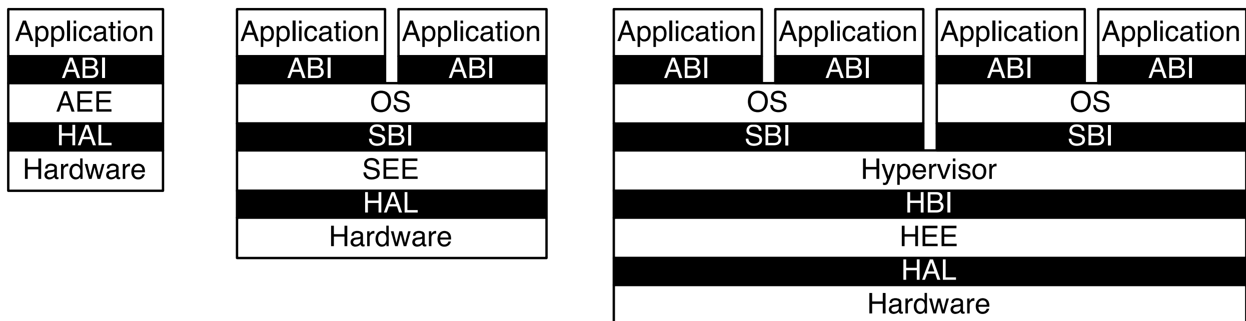


Figure 1.2: Hardware abstraction layers (HALs) abstract underlying hardware platforms from the execution environments.

1.3 Privilege Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Four RISC-V privilege levels are currently defined as shown in Table 1.1.

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

Table 1.1: RISC-V privilege levels.

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

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is inherently trusted, as it has low-level access to the machine implementation. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively, while hypervisor-mode (H-mode) is intended to support virtual machine monitors.

Each privilege level has a core set of privileged ISA extensions with optional extensions and variants. For example, machine-mode supports several optional standard variants for address translation and memory protection.

Although none are currently defined, future hypervisor-level ISA extensions will be added to improve virtualization performance. One common feature to support hypervisors is to provide a second level of translation and protection, from supervisor physical addresses *to* hypervisor physical addresses*.*

Implementations might provide anywhere from 1 to 4 privilege modes trading off reduced isolation for lower implementation cost, as shown in Table 1.2.

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

All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine. The simplest RISC-V implementations may provide only M-mode, though this will provide no protection against incorrect or malicious application code. Many RISC-V implementations will also support at least user mode (U-mode) to protect the rest of the system

Number of levels	Supported Modes
1	M
2	M, U
3	M, S, U
4	M, H, S, U

Table 1.2: Supported combinations of privilege modes.

from application code. Supervisor mode (S-mode) can be added to provide isolation between a supervisor-level operating system and the SEE and HAL code. The hypervisor mode (H-mode) is intended to provide isolation between a virtual machine monitor and a HEE and HAL running in machine mode.

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

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

Chapter 2

Control and Status Registers (CSRs)

The SYSTEM major opcode is used to encode all privileged instructions in the RISC-V ISA. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other privileged instructions. In this chapter, we describe instructions to access the CSRs, as these are common to all privilege levels. The following chapters describe the function of each of the CSRs according to privilege level, as well as the other privileged instructions which are generally closely associated with a particular privilege level. Note that although CSRs and instructions are associated with one privilege level, they are also accessible at all higher privilege levels.

Placing all privileged instructions under a common major opcode and structure simplifies hardware trap encoding and provision of virtualized execution environments.

In this draft version of the specification, many CSR registers contain fields whose value is currently not used and set to zero or whose value currently only supports a limited range of settings, but which might in future support an expanded range of settings. In general, software should only write fields with supported values, and hardware should only return specified default values. Prior to the final release of the specification, these will be appropriately marked to indicate the exact behavior required for a correct forward-compatible implementation.

2.1 Instructions to access CSRs

In addition to the user-level state described in Volume I of this manual, an implementation may contain additional CSRs, accessible by some subset of the privilege levels. The following instructions are provided to atomically read and modify CSRs. Instructions that manipulate CSRs might also have other side effects.

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	zimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRCI	dest	SYSTEM

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR.

The CSRRS (Atomic Read and Set Bit in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

The CSRRC (Atomic Read and Clear Bit in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

For both CSRRS and CSRRC, if *rs1*=`x0`, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write. Note that if *rs1* specifies a register holding a zero value other than `x0`, the instruction will still write the unmodified value back to the CSR.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit immediate (zimm[4:0]) field encoded in the *rs1* field instead of a value from an integer register. If the zimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write.

The assembler pseudo-instruction to read a CSR, CSRR *rd, csr*, is encoded as CSRRS *rd, csr, x0*. The assembler pseudo-instruction to write a CSR, CSRW *csr, rs1*, is encoded as CSRRW *x0, csr, rs1*, while CSRWI *csr, zimm*, is encoded as CSRRWI *x0, csr, zimm*.

Further assembler pseudo-instructions are defined to set and clear bits in the CSR when the old value is not required: CSRS/CSRC *csr, rs1*; CSRSI/CSRCI *csr, zimm*.

2.2 CSR Address Mapping Conventions

The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (csr[11:8]) are used to encode the read and write accessibility of the CSRs according to privilege level as shown in Table 2.1. The top two bits (csr[11:10]) indicate whether the register is read/write (`00`, `01`, or `10`) or read-only (`11`). The next two bits (csr[9:8]) indicate the lowest privilege level that can access the CSR (`00` for user, and `01` for supervisor).

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

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

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

Table 2.1 also indicates the convention to allocate CSR addresses between standard and non-standard uses. The CSR addresses reserved for non-standard uses will not be redefined by future standard extensions. The shadow addresses are reserved to provide a read-write address via which a higher privilege level can modify a register that is read-only at a lower privilege level. Note that if one privilege level has already allocated a read/write shadow address, then any higher privilege level can use the same CSR address for read/write access to the same register.

Effective virtualization requires that as many instructions run natively as possible inside a virtualized environment, while any privileged accesses trap to the virtual machine monitor [1]. CSRs that are read-only at some lower privilege level are shadowed into separate CSR addresses if they are made read-write at a higher privilege level. This avoids trapping permitted lower-privilege accesses while still causing traps on illegal accesses.

2.3 CSR Listing

Tables 2.2–2.5 lists the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are the only standard user-level CSRs currently defined. The other registers are used by privileged code, as descrbed in the following chapters. Note that not all registers are required on all implementations.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:6]		
User CSRs				
00	00	XX	0x000-0x0FF	Standard read/write
01	00	XX	0x400-0x4FF	Standard read/write
10	00	XX	0x800-0x8FF	Non-standard read/write
11	00	00-10	0xC00-0xCBF	Standard read-only
11	00	11	0xCC0-0xCFF	Non-standard read-only
Supervisor CSRs				
00	01	XX	0x100-0x1FF	Standard read/write
01	01	0X	0x500-0x57F	Standard read/write
01	01	1X	0x580-0x5FF	Non-standard read/write
10	01	00-10	0x900-0x9BF	Standard read/write shadows
10	01	11	0x9C0-0x9FF	Non-standard read/write shadows
11	01	00-10	0xD00-0xDBF	Standard read-only
11	01	11	0xDC0-0xDFF	Non-standard read-only
Hypervisor CSRs				
00	10	XX	0x200-0x2FF	Standard read/write
01	10	0X	0x600-0x67F	Standard read/write
01	10	1X	0x680-0x6FF	Non-standard read/write
10	10	00-10	0xA00-0xABF	Standard read/write shadows
10	10	11	0xAC0-0xAFF	Non-standard read/write shadows
11	10	00-10	0xE00-0xEBF	Standard read-only
11	10	11	0xEC0-0xEFF	Non-standard read-only
Machine CSRs				
00	11	XX	0x300-0x3FF	Standard read/write
01	11	0X	0x700-0x77F	Standard read/write
01	11	1X	0x780-0x7FF	Non-standard read/write
10	11	00-10	0xB00-0xBBF	Standard read/write shadows
10	11	11	0xBC0-0xBFF	Non-standard read/write shadows
11	11	00-10	0xF00-0xFBF	Standard read-only
11	11	11	0xFC0-0xFFF	Non-standard read-only

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

Number	Privilege	Name	Description
User Floating-Point CSRs			
0x001	URW	`fflags`	Floating-Point Accrued Exceptions.
0x002	URW	`frm`	Floating-Point Dynamic Rounding Mode.
0x003	URW	`fcsr`	Floating-Point Control and Status Register (`frm` + `fflags`).
User Counter/Timers			
0xC00	URO	`cycle`	Cycle counter for RDCYCLE instruction.
0xC01	URO	`time`	Timer for RDTIME instruction.
0xC02	URO	`instret`	Instructions-retired counter for RDINSTRET instruction.
0xC80	URO	`cycleh`	Upper 32 bits of `cycle`, RV32I only.
0xC81	URO	`timeh`	Upper 32 bits of `time`, RV32I only.
0xC82	URO	`instreth`	Upper 32 bits of `instret`, RV32I only.

Table 2.2: Currently allocated RISC-V user-level CSR addresses.

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	`sstatus`	Supervisor status register.
0x101	SRW	`stvec`	Supervisor trap handler base address.
0x104	SRW	`sie`	Supervisor interrupt-enable register.
0x121	SRW	`stimecmp`	Wall-clock timer compare value.
Supervisor Timer			
0xD01	SRO	`stime`	Supervisor wall-clock time register.
0xD81	SRO	`stimeh`	Upper 32 bits of `stime`, RV32I only.
Supervisor Trap Handling			
0x140	SRW	`sscratch`	Scratch register for supervisor trap handlers.
0x141	SRW	`sepc`	Supervisor exception program counter.
0xD42	SRO	`scause`	Supervisor trap cause.
0xD43	SRO	`sbadaddr`	Supervisor bad address.
0x144	SRW	`sip`	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	`sptbr`	Page-table base register.
0x181	SRW	`sasid`	Address-space ID.
Supervisor Read/Write Shadow of User Read-Only registers			
0x900	SRW	`cyclew`	Cycle counter for RDCYCLE instruction.
0x901	SRW	`timew`	Timer for RDTIME instruction.
0x902	SRW	`instretw`	Instructions-retired counter for RDINSTRET instruction.
0x980	SRW	`cyclehw`	Upper 32 bits of `cycle`, RV32I only.
0x981	SRW	`timehw`	Upper 32 bits of `time`, RV32I only.
0x982	SRW	`instrethw`	Upper 32 bits of `instret`, RV32I only.

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

Number	Privilege	Name	Description
			Hypervisor Trap Setup
0x200	HRW	hstatus	Hypervisor status register.
0x201	HRW	htvec	Hypervisor trap handler base address.
0x202	HRW	htdeleg	Hypervisor trap delegation register.
0x221	HRW	htimecmp	Hypervisor wall-clock timer compare value.
			Hypervisor Timer
0xE01	HRO	htime	Hypervisor wall-clock time register.
0xE81	HRO	htimeh	Upper 32 bits of htime, RV32I only.
			Hypervisor Trap Handling
0x240	HRW	hscratch	Scratch register for hypervisor trap handlers.
0x241	HRW	hepc	Hypervisor exception program counter.
0x242	HRW	hcause	Hypervisor trap cause.
0x243	HRW	hbadaddr	Hypervisor bad address.
			Hypervisor Protection and Translation
0x28X	TBD	TBD	TBD.
			Hypervisor Read/Write Shadow of Supervisor Read-Only Registers
0xA01	HRW	stimew	Supervisor wall-clock timer.
0xA81	HRW	stimehw	Upper 32 bits of supervisor wall-clock timer, RV32I only.

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

Number	Privilege	Name	Description
		Machine Information Registers	
0xF00	MRO	mcpuid	CPU description.
0xF01	MRO	mimpid	Vendor ID and version number.
0xF10	MRO	mhartid	Hardware thread ID.
		Machine Trap Setup	
0x300	MRW	mstatus	Machine status register.
0x301	MRW	mtvec	Machine trap-handler base address.
0x302	MRW	mtdeleg	Machine trap delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x321	MRW	mtimecmp	Machine wall-clock timer compare value.
		Machine Timers and Counters	
0x701	MRW	mtime	Machine wall-clock time.
0x741	MRW	mtimeh	Upper 32 bits of mtime, RV32I 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	mbadaddr	Machine bad address.
0x344	MRW	mip	Machine interrupt pending.
		Machine Protection and Translation	
0x380	MRW	mbase	Base register.
0x381	MRW	mbound	Bound register.
0x382	MRW	mibase	Instruction base register.
0x383	MRW	mibound	Instruction bound register.
0x384	MRW	mdbase	Data base register.
0x385	MRW	mdbound	Data bound register.
		Machine Read-Write Shadow of Hypervisor Read-Only Registers	
0xB01	MRW	htimew	Hypervisor wall-clock timer.
0xB81	MRW	htimehw	Upper 32 bits of hypervisor wall-clock timer, RV32I only.
		Machine Host-Target Interface (Non-Standard Berkeley Extension)	
0x780	MRW	mtohost	Output register to host.
0x781	MRW	mfromhost	Input register from host.

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

Chapter 3

Machine-Level ISA

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

3.1 Machine-Level CSRs

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

3.1.1 CPU ID Register `mcpuid`

The `mcpuid` register is an XLEN-bit read-only register containing information regarding the capabilities of the CPU implementation. This register must be readable in any implementation, but a value of zero can be returned to indicate the CPU ID feature has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.

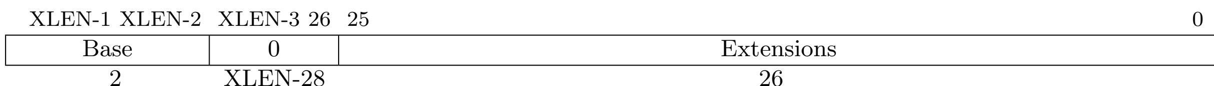


Figure 3.1: Machine CPU ID register (`mcpuid`).

The Base field encodes the native base integer ISA as shown in Table 3.1. For implementations that support multiple ISA variants, the Base field always describes the widest supported ISA variant as this is the ISA mode entered in machine-mode at reset.

The base can be quickly ascertained using branches on the sign of the returned `mcpuid` *value,*

Value	Description
0	RV32I
1	RV32E
2	RV64I
3	RV128I

Table 3.1: Encoding of Base field in `mcpuid`

and possibly a shift left by one and a second branch on the sign. These checks can be written in assembly code without knowing the register width (XLEN) of the machine.

The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet (bit 0 encodes presence of extension "A" , bit 1 encodes presence of extension "B", through to bit 25 which encodes presence of the future "Z" standard extension). The "I" bit will be set for RV32I, RV64I, RV128I base ISAs, and the "E" bit will be set for RV32E.

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

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

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

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

3.1.2 Implementation ID Register `mimpid`

The `mimpid` provides a unique encoding of the source and 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 data fields are not implemented.

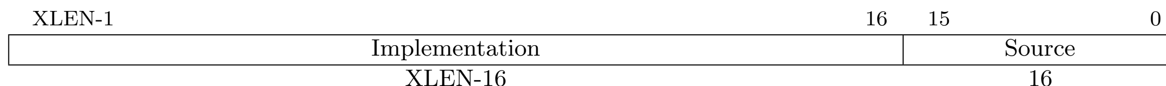


Figure 3.2: Machine Implementation ID register (`mimpid`).

The 16-bit Source field is used to describe the origin of the processor design and is divided into two categories: open-source repos and proprietary implementations. Values `0x0001`–`0x7FFE` are reserved for open-source projects, while values `0x8001`–`0xFFFE` are reserved for closed-source implementations. Values `0x7FFF` and `0xFFFF` are reserved for future expansion. Value `0x8000` is reserved to indicate an anonymous source, which can be used during development before a Source ID is allocated.

Current allocated values for Source are shown in Table 3.2.

Source value	Description
`0x0000`	CPU ID unimplemented
`0x0001`	UC Berkeley Rocket repo
`0x0002`–`0x7FFE`	Reserved for open-source repos
`0x7FFF`	Reserved for extension
`0x8000`	Reserved for anonymous source
`0x8001`–`0xFFFE`	Reserved for proprietary implementations
`0xFFFF`	Reserved for extension

Table 3.2: Encoding of Source field in `mimpid`

The remaining `XLEN`-16 bits of the `mimpid` register are available to encode the implementation details of the design, including microarchitecture type and version number. The format of this field is left to the provider of the Source, but will be printed by standard tools as a hexadecimal string without leading zeros, so the Implementation value should be right-justified with subfields aligned on nibble boundaries to ease human readability.

The `mimpid` value should reflect the design of the RISC-V processor itself and not any surrounding system. Separate mechanisms should be used to encode outer system details.

The intent is for the open-source ID to represent the repo around which development occurs rather than a particular organization. The convention adopted within the Implementation field can be used to segregate branches of the design, including by organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source IDs should only be allocated to released, functioning open-source projects, and will likely be administered by a forthcoming foundation. Commercial IDs will likely be allocated by a forthcoming trade association.

3.1.3 Hart ID Register `mhartid`

The `mhartid` register is an XLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero.

Figure 3.3: Hart ID register (`mhartid`).

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

We do not use CSRs or privileged instructions to convey other information about the organization of the underlying hardware platform, as this would require an unbounded extensible mechanism and must include non-RISC-V cores and slave devices. The system developer must provide a means for machine-mode code to interrogate the platform and discover the system structure.

3.1.4 Machine Status Register (`mstatus`)

The `mstatus` register is an XLEN-bit read/write register formatted as shown in Figure 3.4. The `mstatus` register keeps track of and controls the hart's current operating state. Restricted views of the `mstatus` register appear as the `hstatus` and `sstatus` registers in the H and S privilege-level ISAs respectively.

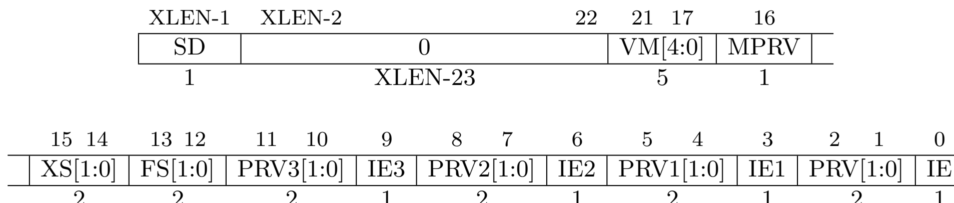


Figure 3.4: Machine-mode status register (`mstatus`).

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

The PRV[1:0] field stores the current privilege mode of the hart, encoded as shown in Table 1.1. If the implementation provides only M-mode, then these two bits are hard-wired to binary 11.

The IE bit indicates whether interrupts are enabled for the current privilege mode (1=Enabled, 0=Disabled), and is primarily used to disable interrupts to ensure atomicity with respect to interrupt handlers at the current privilege level. When a hart is running in a given privilege mode, interrupts for higher privilege modes are always enabled while interrupts for lower privilege modes are always disabled. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected interrupts before ceding control to a lower privilege level.

The active IE bit is located in bit 0 to allow it to be atomically set or cleared using a single CSR instruction.

To support nested traps, a stack of PRV and IE bits is provided, the depth of which is equal to the number of supported privilege modes, where PRV0 is the active privilege mode PRV (i.e., PRV0–PRVN for N privilege modes), except if the implementation only supports machine mode in which case the stack is two deep and all PRV fields are hardwired to 11. When a trap is taken, the stack is pushed to the left and PRV is set to the privilege mode of the activated trap handler with IE=0. On a return from the trap handler (using an ERET instruction), the stack is popped to the right and the leftmost entry (PRVN) is set to the lowest-supported privilege mode with interrupts enabled (i.e., on a machine with only M mode, PRV1=M and IE1=1, while on machines with two or more modes, PRVN=U and IEN=1 on return from a trap handler). In normal operation, the stack should contain monotonically increasing privilege modes from left to right (oldest to newest).

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

We considered adding an additional level to the privilege stack for implementations with multiple privilege levels, to allow M-mode software to generate traps without saving and restoring the `mstatus` *register. However, current M-mode software that might generate exceptions does not seem to benefit from this feature. For example, when emulating missing hardware features using M-mode software, the* `mstatus` *register is typically manipulated for other reasons (e.g., to set the* `MPRV` *bit). Saving and restoring the privilege stack can be folded into such actions at no cost.*

If M-mode software wishes to enable interrupts, then saving and restoring the privilege stack can similarly be folded into the interrupt enable/disable sequence.

When the stack is popped, the lowest-supported privilege mode with interrupts enabled is added to the bottom of stack to help catch errors that cause invalid entries to be popped off the stack.

PRV fields need only be able to store supported privilege modes.

If the machine provides only U and M modes, then only a single hardware storage bit is required to represent either 00 or 11. However, software should only write valid values to these fields to preserve compatibility.

3.1.6 Virtualization Management Field in `mstatus` Register

The virtualization management field VM[4:0] indicates the currently active scheme for virtualization, including virtual memory translation and protection. Table 3.3 shows the currently defined virtualization schemes. Only the Mbare mode is mandatory for a RISC-V hardware implementation. The Mbare, Mbb, and Mbbid schemes are described in Sections 3.5–3.6, while the page-based virtual memory schemes are described in later chapters.

Each setting of the VM field defines operation at all supported privilege levels, and the behavior of some VM settings might differ depending on the privilege levels supported in hardware.

Value	Abbreviation	Modes Required	Description
0	Mbare	M	No translation or protection.
1	Mbb	M, U	Single base-and-bound.
2	Mbbid	M, U	Separate instruction and data base-and-bound.
3–7	*Reserved*		
8	Sv32	M, S, U	Page-based 32-bit virtual addressing.
9	Sv39	M, S, U	Page-based 39-bit virtual addressing.
10	Sv48	M, S, U	Page-based 48-bit virtual addressing.
11	Sv57	M, S, U	Reserved for page-based 57-bit virtual addressing.
12	Sv64	M, S, U	Reserved for page-based 64-bit virtual addressing.
13–31	*Reserved*		

Table 3.3: Encoding of virtualization management field VM[4:0].

Mbare corresponds to no memory management or translation, and so all effective addresses regardless of privilege mode are treated as machine physical addresses. Mbare is the mode entered at reset.

Mbb is a base-and-bounds architectures for systems with at least two privilege levels (U and M). Mbb is suited for systems that require low-overhead translation and protection for user-mode code, and that do not require demand-paged virtual memory (swapping is supported). A variant Mbbid provides separate address and data segments to allow an execute-only code segment to be shared between processes.

Sv32 is a page-based virtual-memory architecture for RV32 systems providing a 32-bit virtual address space designed to support modern supervisor-level operating systems, including Unix-based systems.

Sv39 and Sv48 are page-based virtual-memory architectures for RV64 systems providing a 39-bit or 48-bit virtual address space respectively to support modern supervisor-level operating systems, including Unix-based systems.

Sv32, Sv39, and Sv48 require implementations to support M, S, and U privilege levels. If H-mode is also present, additional operations are defined for hypervisor-level code to support multiple supervisor-level virtual machines. Hypervisor-mode support for virtual machines has not yet been defined.

The existing Sv39 and Sv48 schemes can be readily extended to Sv57 and Sv64 virtual address widths. Sv52, Sv60, Sv68, and Sv76 virtual address space widths are tentatively planned for RV128 systems, where virtual address widths under 68 bits are intended for applications requiring 128-bit integer arithmetic but not larger address spaces.

Our current definition of the virtualization management schemes only supports the same base architecture at every privilege level. Variants of the virtualization schemes can be defined to support narrow widths at lower-privilege levels, e.g., to run RV32 code on an RV64 system.

3.1.7 Memory Privilege in `mstatus` Register

The MPRV bit modifies the privilege level at which loads and stores execute. When MPRV=0, translation and protection behave as normal. When MPRV=1, data memory addresses are translated and protected as though PRV were set to the current value of the PRV1 field. Instruction address-translation and protection are unaffected.

When an exception occurs, MPRV is reset to 0.

The MPRV mechanism was conceived to improve the efficiency of M-mode routines that emulate missing hardware features, e.g., misaligned loads and stores.

3.1.8 Extension Context Status in `mstatus` Register

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

The FS[1:0] and XS[1:0] read/write fields are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extension

respectively. The FS field encodes the status of the floating-point unit, including the CSR `fcsr` and floating-point data registers `f0`–`f31`, while the XS field encodes the status of any additional user-mode extension and associated state. The SD bit is a read-only bit that summarizes whether either the FS field or XS field encodes a dirty state that will require saving extended user context to memory. In systems without a floating-point unit, the FS field is hardwired to zero, and in systems without additional user extensions requiring new state, the XS field is hardwired to zero. If both XS and FS are hardwired to zero, then SD is also always zero.

The FS and XS fields use the same status encoding as shown in Table 3.4, with the four possible status values being Off, Initial, Clean, and Dirty.

Status	Meaning
0	Off
1	Initial
2	Clean
3	Dirty

Table 3.4: Encoding of FS[1:0] and XS[1:0] status fields.

To date, there are no standard extensions that define additional state beyond the floating-point CSR and data registers.

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

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

The FS and XS fields are set by privileged code when resuming a user context, and are read by the privileged code before saving the context. The status fields will also be updated during execution of instructions, regardless of privilege mode.

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

For example, a coprocessor might require to be configured before use and can be “unconfigured” after use. The unconfigured state would be represented as the Initial state for context save. If the same application remains running between the unconfigure and the next configure (which would set status to Dirty), there is no need to actually reinitialize the state at the unconfigure instruction,

as all state is local to the user process, i.e., the Initial state may only cause the coprocessor state to be initialized to a constant value at context restore, not at every unconfigure.

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

Table 3.5 shows all the possible state transitions for the FS or XS status bits. Note that the standard floating-point extensions do not support user-mode unconfigure or disable/enable instructions.

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

Table 3.5: Encoding of FS[1:0] and XS[1:0] status fields.

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

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

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

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

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

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

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

All privileged modes share a single copy of the FS and XS bits. In a system with more than one privileged mode, supervisor mode would normally use the FS and XS bits directly to record the status with respect to the supervisor-level saved context. Other more-privileged active modes must be more conservative in saving and restoring the extension state in their corresponding version of the context, but can rely on the Off state to avoid save and restore, and the Initial state to avoid saving the state.

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.

3.1.9 Machine Trap Vector Base Address Register (`mtvec`)

The `mtvec` register is an XLEN-bit read/write register that holds the base address of the M-mode trap vector.

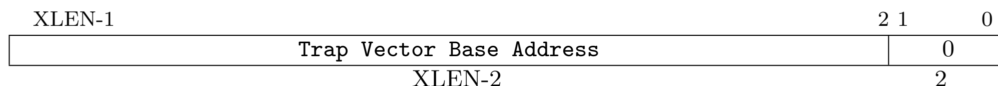


Figure 3.5: Machine trap vector base address register (`mtvec`).

The `mtvec` register must always be implemented, but can contain a hard-wired read-only value. Two standard values, `0xF...FFE00` and `0x0...00100`, are specified for high and low locations of the trap vector respectively, and one of these should be present in `mtvec` after reset. The standard reset vector is either `0xF...FFF00` or `0x0...0200` for high and low locations of the trap vector respectively.

The `mtvec` register can be implemented as a read/write register to support a variable trap vector base address. The number of writable bits in the mtvec register can vary by implementation, but

Address	Handler
High Trap Vector Addresses	
0xF...FE00	Trap from user-mode
0xF...FE40	Trap from supervisor-mode
0xF...FE80	Trap from hypervisor-mode
0xF...FEC0	Trap from machine-mode
0xF...FEFC	Non-maskable interrupt(s)
0xF...FF00	Reset vector
Low Trap Vector Addresses	
0x100	Trap from user-mode
0x140	Trap from supervisor-mode
0x180	Trap from hypervisor-mode
0x1C0	Trap from machine-mode
0x1FC	Non-maskable interrupt(s)
0x200	Reset vector

Table 3.6: Standard locations of M-mode trap vector addresses at either high or low memory locations.

the two standard values above must be supported if any bits are writable. The value in the `mtvec` register must always be aligned on a 4-byte boundary (low two bits are always zero). The sign bit should always be writable if any bits are writable, and the sign must be extended down from the sign bit to the next writable bit. The value returned by reading a variable `mtvec` register should always match the value used to generate the PC base address when handling traps.

A trap in privilege level P causes a jump to the address `mtvec` + $P\times$`0x40`. Non-maskable interrupts cause a jump to address `mtvec` + `0xFC`. Additional trap vector entry points can be defined by implementations to allow more rapid identification and service of certain trap causes.

We allow for considerable flexibility in implementation of the trap vector base address. On the one hand we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems. Different system contexts can mandate high or low locations of reset and trap handling code and so we support both as standard hard-wired vectors.

3.1.10 Machine Trap Delegation Register (`mtdeleg`)

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can quickly redirect traps back to the appropriate level using `mrts` and `mrth` instructions (Section 3.2.2). To increase performance, implementations can provide individual read/write bits within `mtdeleg` to indicate that certain traps should be processed directly by a lower privilege level.

The machine trap delegation register (`mtdeleg`) is an XLEN-bit read/write register that must be implemented, but which can contain a read-only value of zero, indicating that hardware will always direct all traps to machine mode.

If a hypervisor mode is present, a set bit in `mtdeleg` register will delegate any corresponding trap in U-mode, S-mode, or H-mode to the H-mode trap handler. H-mode may in turn set corresponding

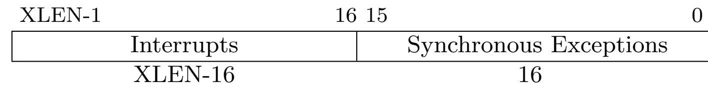


Figure 3.6: Machine Trap Delegation Register `mtdeleg`.

bits in the `htdeleg` register to delegate traps that occur in S-mode or U-mode to the S-mode trap handler.

If only a supervisor mode is present, then setting a bit in `mtdeleg` will delegate any corresponding trap in S-mode or U-mode to the S-mode trap handler.

If neither hypervisor nor supervisor modes are implemented, the `mtdeleg` register should be hardwired to zero.

An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in `mtdeleg` to see which bit positions hold a one.

The low 16 bits of `mtdeleg` has a bit position allocated for every synchronous exception shown in Table 3.7, with the index of the bit position equal to the value returned in the `mcause` register (i.e., setting bit 8 allows user-mode environment calls to be delegated to a lower-privilege trap handler).

Bits 16 and above hold trap delegation bits for individual interrupts, with the layout of bits matching those in the `mip` register shifted left by 16 bits (i.e., STIP interrupt delegation control is located in bit 21 of `mtdeleg`).

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

The `mip` register is an XLEN-bit read/write register containing information on pending interrupts, while `mie` is the corresponding XLEN-bit read/write register containing interrupt enable bits. Only the lower bits corresponding to software interrupts (SSIP, HSIP, MSIP) in `mip` are writable through this CSR address, while the remaining bits are read-only. Restricted views of the `mip` and `mie` registers appear as the `hip`/`hie` and `sip`/`sie` registers in the H and S privilege-level ISAs respectively.

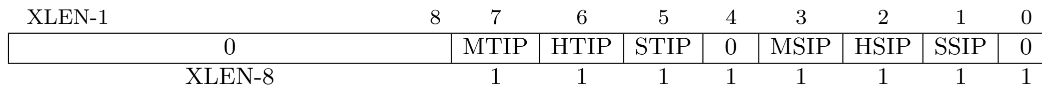


Figure 3.7: Machine interrupt-pending register (`mip`).

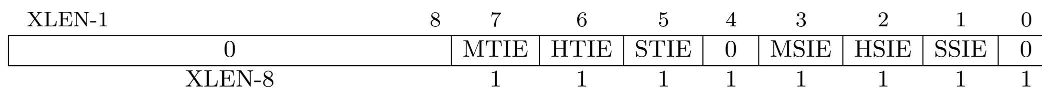


Figure 3.8: Machine interrupt-enable register (`mie`).

Space has been reserved for possibly adding user-level software interrupts in the future.

The MTIP, HTIP, STIP bits correspond to timer interrupt-pending bits for supervisor, hypervisor, and machine timer interrupts respectively, and are cleared by writing to the `mtimecmp`, `htimecmp`, or `stimecmp` register respectively.

For each supported non-user privilege mode there is a separate timer interrupt-enable bit, named MTIE, HTIE, STIE for M-mode, H-mode, and S-mode timer interrupts respectively. If a privilege mode is not supported, the associated interrupt-enable bit is hardwired to zero.

Space has been reserved for possibly adding user-level timer interrupts in the future.

Each of the supported non-user privilege levels has a separate software interrupt-pending bit (MSIP, HSIP, SSIP), which can be both read and written by CSR accesses from code running on the local hart at the associated or any higher privilege level. If a privilege level is not supported, the associated software interrupt-pending bit is hardwired to zero. The machine-level MSIP bits can also be written by accesses from remote harts to provide machine-mode interprocessor interrupts. Interprocessor interrupts for lower privilege levels are implemented through SBI or HBI calls to the SEE or HEE respectively, which might ultimately result in a machine-mode write to the receiving hart's MSIP bit.

The software interrupt for a given privilege level is disabled if the relevant SIE bit in the `mie` is clear or if the global IE bit in the `mstatus` register is clear when the hart is executing in that privilege mode, or if the hart is executing at a higher privilege mode.

We only allow a hart to directly write its own HSIP and SSIP bits when running in hypervisor or supervisor mode, as other hypervisor-level or supervisor-level harts might be virtualized and possibly descheduled by higher privilege levels. We rely on SBI and HBI calls to provide interprocessor interrupts for this reason. Machine-mode harts are not virtualized and can directly interrupt other harts by setting their MSIP bits, typically using uncached writes to memory-mapped control registers, possibly inside a global interrupt controller on the hardware platform.

Implementations might add additional machine-level interrupt sources to these registers.

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

3.1.12 Machine Timer Registers (`mtime`, `mtimecmp`)

M-mode includes a timer facility provided by the `mtimecmp` register together with the real-time counter `mtime`. The hardware platform must provide a facility for determining the timebase of `mtime`, which must run at a constant frequency.

The `mtimecmp` register has 32-bit precision on all RV32, RV64, and RV128 systems. A timer interrupt is posted when the low 32 bits of the `mtime` register match the value in the low 32 bits of the `mtimecmp` register. The interrupt remains posted until it is cleared by writing the `mtimecmp` register. The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

The timer facility is defined to use wall-clock time rather than a cycle counter to support modern

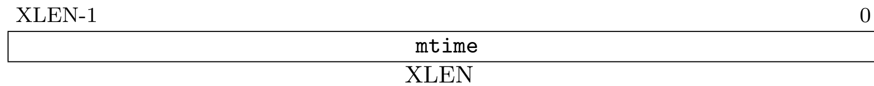


Figure 3.9: Machine time register.

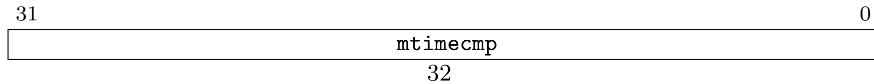


Figure 3.10: Machine time compare register.

processors that run with a highly variable clock frequency to save energy through dynamic voltage and frequency scaling. Simple fixed-frequency systems can use a single clock for both cycle counting and wall-clock time.

True real-time clocks (RTCs) are relatively expensive to provide (requiring a crystal or MEMS oscillator), and have to run even when the rest of system is powered down, so usually there is only one in a system. Given an underlying real-time clock (RTC), we can implement any of the virtual timers by storing delta values in static (i.e., non-incrementing) registers. On a store to `mtime`*, the implementation will actually read the RTC, subtract the current RTC value from the desired* `mtime` *value, and store the difference in the* `mtime` *register for each hart. When* `mtime` *is read, the underlying RTC is read again, and the stored delta in* `mtime` *is added to form the result returned by the instruction. The same approach can be used for the various wall-clock timers at each privilege level, and to calculate correct timer compare values.*

One issue in variable-frequency systems is that the real-time clock (RTC) and the compare registers will usually be held in a separate clock domain from the processor, and so accesses to the register and the interrupt signals will incur the latency of a clock-domain crossing.

3.1.13 Machine Scratch Register (`mscratch`)

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

Figure 3.11: Machine-mode scratch register.

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

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

For hard real-time systems, some systems use a more complex register banking scheme to map separate interrupt-context register banks into the architectural register namespace to provide very low latency interrupt handling. We are exploring a different approach for hard real-time systems that instead provides multiple complete register contexts, to reduce software complexity and improve performance.

3.1.14 Machine Exception Program Counter (`mepc`)

`mepc` is an XLEN-bit read/write register formatted as shown in Figure 3.12. The low bit of `mepc` (`mepc[0]`) is always zero. On implementations that do not support instruction-set extensions with 16-bit instruction alignment, the two low bits (`mepc[1:0]`) are always zero.

The `mepc` *register can never hold a PC value that would cause an instruction-address-misaligned exception.*

When a trap is taken, `mepc` is written with the virtual address of the instruction that encountered the exception.

Figure 3.12: Machine exception program counter register.

3.1.15 Machine Cause Register (`mcause`)

The `mcause` register is an XLEN-bit read-write register formatted as shown in Figure 3.13. The Interrupt bit is set if the exception was caused by an interrupt. The Exception Code field contains a code identifying the last exception. The center bits will read zero,and should be written with zero to support future expansion of the Exception Code field. Table 3.7 lists the possible machine-level exception codes.

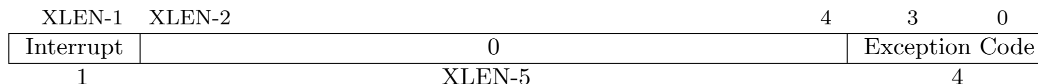


Figure 3.13: Machine Cause register `mcause`.

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

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

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Environment call from H-mode
0	11	Environment call from M-mode
0	≥12	*Reserved*
1	0	Software interrupt
1	1	Timer interrupt
1	≥2	*Reserved*

Table 3.7: Machine cause register (`mcause`) values.

3.1.16 Machine Bad Address (`mbadaddr`) Register

`mbadaddr` is an XLEN-bit read-write register formatted as shown in Figure 3.14. When an instruction-fetch address-misaligned exception, or instruction-fetch access exception, or load or store address-misaligned exception, or load or store access exception occurs, `mbadaddr` is written with the faulting address. The value in `mbadaddr` is undefined for other exceptions.

Figure 3.14: Machine bad address register.

For instruction-fetch access faults on RISC-V systems with variable-length instructions, `mbadaddr` will point to the portion of the instruction that caused the fault while `mepc` will point to the beginning of the instruction.

3.2 Machine-Mode Privileged Instructions

3.2.1 Instructions to Change Privilege Level

Instructions to change privilege level are encoded under the PRIV minor opcode. ECALL (Environment Call) and EBREAK (Environment Breakpoint) are available at all privilege levels, while ERET (Environment Return) is only available at privilege levels S, H, and M.

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
ERET	0	PRIV	0	SYSTEM

The ECALL instruction is used to make a request to a higher privilege level. The binary interface to the execution environment will define how parameters for the request are passed, but usually these will be in defined locations in the integer register file. Executing an ECALL instruction causes an Environment Call exception.

We have renamed SCALL in the user ISA to ECALL to make it more general. This renaming does not change the opcode encoding or the functionality fo the user-mode instruction, but will require a change to assembler/disassembler to support the new name.

The EBREAK instruction is used by debuggers to cause control to be transferred back to the debugging environment. Executing an EBREAK instruction causes a Breakpoint exception.

The standard does not allow unused bits in the EBREAK encoding to be used to encode debugging information as this is better kept in a hash table indexed by the appropriate `epc` *register.*

After handling a trap, the ERET instruction is used to return to the privilege level at which the trap occurred. In addition to manipulating the privilege stack as described in Section 3.1.5, ERET sets the `pc` to the value stored in the `Xepc` register, where `X` is the privilege mode (S, H, or M) in which the ERET instruction was executed.

3.2.2 Trap Redirection Instructions

31–20	19–15	14–12	11–7	6–0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRTS	0	PRIV	0	SYSTEM
MRTH	0	PRIV	0	SYSTEM

The MRTS (Machine Redirect Trap to Supervisor) instruction delegates the handling of a trap from M-mode to S-mode. MRTS changes the privilege mode to S and sets the `pc` to the supervisor's trap handler, which is stored in the `stvec` register. Additionally, the values in the `mepc`, `mcause`, and `mbadaddr` registers are copied to the `sepc`, `scause`, and `sbadaddr` registers, respectively.

The MRTH (Machine Redirect Trap to Hypervisor) instruction is defined analogously, but transfers control to `htvec` in H-mode. `mepc`, `mcause`, and `mbadaddr` are copied to `hepc`, `hcause`, and `hbadaddr`, respectively.

Simple implementations may direct all traps to an M-mode trap handler, even those destined for

a lower-privilege mode. The trap-redirection instructions allow the M-mode handler to quickly transfer control to the lower-privilege mode's trap handler.

Opcode space has been reserved for the HRTS instruction, which would redirect a trap from H-mode to S-mode. To facilitate horizontal user-mode traps, we have also reserved space for MRTU, HRTU, and SRTU instructions.

3.2.3 Wait for Interrupt

The Wait for Interrupt instruction (WFI) provides a hint to the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should be routed to this hart. WFI is available at the S, H, and M privilege levels.

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

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

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

The WFI instruction is just a hint, and a legal implementation is to implement WFI as a NOP.

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

Interrupts can be disabled when the WFI instruction is executed, but the hart must resume execution if any interrupts (enabled or not) are pending, or become pending while the hart is stalled. If any masked interrupt is or becomes pending, execution will resume at `pc` + 4, and software must determine what action to take for any pending interrupt.

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

The `mip`, `hip`, `sip` *registers can be interrogated to determine the presence of any interrupt in machine, hypervisor, or supervisor mode respectively.*

As implementations are free to implement WFI as a NOP, software must explicitly check for any relevant pending but disabled interrupts in the code following an WFI, and should loop back to the WFI if no suitable interrupt was detected.

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

3.3 Physical Memory Attributes

Access to system physical memory is mediated by machine mode. Because the layout of the physical memory space is highly system dependent, this section describes the overall approach of specifying attributes for each physical address range rather than specific details for a given hardware platform.

The physical memory map for a complete system includes various memory regions and various memory-mapped control registers. Some memory regions might not exist, some might be read only, some might not support subword or even subblock accesses, some might not support atomic operations, and some might not support cache coherence. Similarly, memory-mapped control registers vary in their supported access widths, and whether read and write accesses have associated side effects.

While many systems specify such attributes in the virtual memory page tables, this injects platform-specific information into a virtualized layer and can cause system errors unless attributes are correctly initialized in each page-table entry for each platform physical memory region. In addition, the available page sizes might not be optimal for specifying attributes in the physical memory space.

For RISC-V, we separate out specification of machine physical memory attributes into a separate custom hardware structure. In many cases, the attributes are known at system design time for each physical address region, and can be hard-wired into the memory datapath of each RISC-V processor of the system. Alternatively, machine-level control registers can be provided to specify these attributes at a granularity appropriate to each region on the platform (for example, if an on-chip SRAM can be flexibly divided between cacheable and uncacheable uses). These attributes will be applied to any access to the physical memory region, including accesses that have undergone virtual to physical memory translation.

To aid in system debugging, we strongly recommend that RISC-V processors trap illegal physical memory accesses precisely at the core, instead of reporting them as imprecise machine check errors from the memory subsystem.

3.4 Physical Memory Access Control

To contain faults and support secure processing, it is desirable to limit the physical addresses accessible by a lower-privilege context running on a hart. Similar to the physical memory attributes described in the previous section, a RISC-V system should provide per-hart control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory space. The granularity of the access control settings can be varied for different physical memory address spaces on the hardware platform, and certain region's privileges can be hardwired.

These machine-mode physical memory access controls are applied for all accesses when the hart is running in U, S, or H modes, and for load and stores when the hart is running in M mode with the MPRV bit set in the `mstatus` register. As with the physical address attributes described in the previous section, illegal physical memory accesses should be trapped precisely.

3.5 Mbare addressing environment

The Mbare environment is selected at reset or can be entered at any time by writing the VM field in the `mstatus` register.

In the Mbare environment all virtual addresses are converted with no translation into physical addresses, with truncation of any excess high order bits. The physical memory attributes and access controls described in the previous sections can be used to constrain accesses.

3.6 Base-and-Bound environments

This section describes the Mbb virtualization environment, which provides a base-and-bound translation and protection scheme. There are two variants of base-and-bound, Mbb and Mbbid, depending on whether there is a single base-and-bound (Mbb) or separate base-and-bounds for instruction fetches and data accesses (Mbbid). This simple translation and protection scheme has the advantage of low complexity and deterministic high performance, as there are never any TLB misses during operation.

3.6.1 Mbb: Single Base-and-Bound registers (`mbase`, `mbound`)

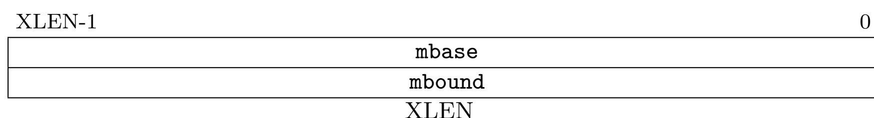


Figure 3.15: Single Base-and-Bound Registers.

The simpler Mbb system has a single base `mbase` and single bound `mbound` register. Mbb is enabled by writing the value 1 to the VM field in the `mstatus` register.

The base-and-bound registers define a contiguous virtual-address segment beginning at virtual address 0 with a length given in bytes by the value in `mbound`. This virtual address segment is mapped to a contiguous physical address segment starting at the physical address given in the `mbase` register.

When Mbb is in operation, all lower-privilege mode (U, S, H) instruction-fetch addresses and data addresses are translated by adding the value of `mbase` to the virtual address to obtain the physical address. Simultaneously, the virtual address is compared against the value in the bound register. An address fault exception is generated if the virtual address is equal to or greater than the virtual address limit held in the `mbound` register.

Machine-mode instruction fetch and data accesses are not translated or checked in Mbb (except for loads and stores when the MPRV bit is set in `mstatus`), so machine-mode effective addresses are treated as physical addresses.

3.6.2 Mbbid: Separate Instruction and Data Base-and-Bound registers

Figure 3.16: Separate instruction and data base-and-bound registers.

The Mbbid scheme separates the virtual address segments for instruction fetches and data accesses to allow a single physical instruction segment to be shared by two or more user-level virtual address spaces while a separate data segment is allocated to each. Mbbid is enabled by writing 2 to the VM field of `mstatus` register.

The split instruction and data base-and-bounds scheme was famously used on Cray supercomputers, where it avoids most runtime overheads related to translation and protection provided the segments fit in physical memory.

The `mibase` and `mibound` registers define the physical start address and length of the instruction segment respectively, while `mdbase` and `mdbound` specify the physical start address and length of the data segment respectively.

The data virtual address segment begins at address 0, while the instruction virtual address segment begins half way through the virtual address space, at an address given by a leading 1 following by XLEN-1 trailing zeros (e.g., `0x8000_0000` for 32-bit address space systems). The virtual addresses of lower privilege-mode instruction fetches are first checked to ensure their high bit is set; if not, an exception is generated. The high bit is subsequently treated as zero when adding the base to the virtual address and when checking the bound.

The data and instruction virtual address segments should not overlap, and we felt it more important to preserve the potential of zero page data accesses (using a 12-bit offset from register `x0`*) than to support instruction entry points using JALR with* `x0`*. In particular, a single JAL can directly access all of a 2 MiB code segment.*

To simplify linking, the instruction virtual address segment start address should be constant independent of the length of the complete binary. Placing at the midpoint of virtual memory minimizes the circuitry needed to separate the two segments.

Systems that provide Mbbid must also provide Mbb. Writes to the CSR addresses corresponding to `mbase` should write the same value to `mibase` & `mdbase`, and writes to `mbound` should write the same value to `mibound` & `mdbound` to provide compatible behavior. Reads of `mbase` should return the value in `mdbase` and reads of `mbound` should return the value in `mdbound`. When VM is set to Mbb, instruction fetches no longer check the high bit of the virtual address, and no longer reset the high bit to zero before adding base and checking bound.

While the split scheme allows a single physical instruction segment to be shared across multiple user process instances, it also effectively prevents the instruction segment from being written by the user program (data stores are translated separately) and prevents execution of instructions from the data segment (instruction fetches are translated separately). These restrictions can prevent some forms of security attack.

On the other hand, many modern programming systems require, or benefit from, some form of runtime-generated code, and so these should use the simpler Mbb mode with a single segment, which is partly why supporting this mode is required if providing Mbbid.

Chapter 4

Supervisor-Level ISA

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 always operates inside a virtual memory scheme defined by the VM field in the machine-mode `mstatus` register. Supervisor-level code is written to a given VM scheme, and cannot change the VM scheme in use.

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

Supervisor mode is deliberately restricted in terms of interactions with underlying physical hardware, such as physical memory and device interrupts, to support clean virtualization. A more conventional virtualization-unfriendly operating system can be ported by using M-mode to initially map physical memory into the supervisor virtual memory address space, and by delegating device interrupts to S-mode.

4.1 Supervisor CSRs

A number of CSRs are provided for the supervisor.

The supervisor should only view CSR state that should be visible to a supervisor-level operating system. In particular, there is no information about the existence (or non-existence) of higher privilege levels (hypervisor or machine) visible in the CSRs accessible by the supervisor. Additional CSRs, visible only to the higher-privilege levels, will encode if a processor is currently executing in a privilege level greater than supervisor level.

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 XLEN-bit read/write register formatted as shown in Figure 4.1. The `sstatus` register keeps track of the processor's current operating state.

XLEN-1	XLEN-2 … 17	16	15 … 14	13 … 12	11 … 5	4	3	2 … 1	0
SD	0	MPRV	XS[1:0]	FS[1:0]	0	PS	PIE	0	IE
1	XLEN-18	1	2	2	7	1	1	2	1

Figure 4.1: Supervisor-mode status Register.

The PS bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, PS is set to 0 if the trap originated from user mode, or 1 otherwise. When an ERET instruction (see Section 3.2.1) is executed to return from the trap handler, the privilege level is set to user mode if the PS bit is 0, or supervisor mode if the PS bit is 1.

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

The PIE bit indicates whether interrupts were enabled before entering supervisor mode. When a trap is taken, PIE is set to IE and IE is set to 0. When an ERET instruction is executed, IE is set to PIE.

4.1.2 Memory Privilege in `sstatus` Register

The MPRV bit modifies the privilege level at which loads and stores execute. When MPRV=0, memory is protected as normal. When MPRV=1, data memory is protected as though the current privilege level were given by the PS bit (i.e., user level when PS=0, or supervisor level when PS=1). Instruction memory protection is unaffected by the setting of the MPRV bit.

When an exception occurs, MPRV is reset to 0.

The MPRV mechanism allows supervisor software to reference memory on behalf of the user without inadvertently accessing memory protected from the user.

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

The `sip` register is an XLEN-bit read/write register containing information on pending interrupts, while `sie` is the corresponding XLEN-bit read/write register containing interrupt enable bits.

Two types of interrupts are defined: software interrupts and timer interrupts. A software interrupt is triggered on the current hart by writing 1 to its software interrupt-pending (SSIP) bit in the `sip` register. A pending software interrupt can be cleared by writing 0 to the SSIP bit in `sip`. All other bits in the `sip` register are read-only. Software interrupts are disabled when the software interrupt enable (SSIE) bit in the `sie` register is clear.

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

A timer interrupt is pending if the STIP bit in the `sip` register is set. Timer interrupts are disabled when the STIE bit in the `sie` register is clear. A pending timer interrupt is cleared by writing the `stimecmp` register.

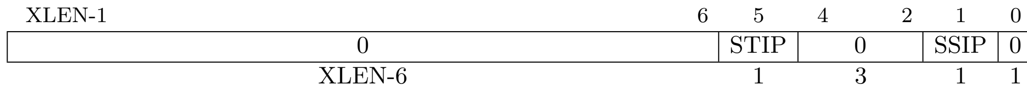


Figure 4.2: Supervisor interrupt-pending register (`sip`).

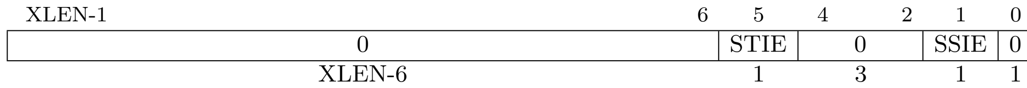


Figure 4.3: Supervisor interrupt-enable register (`sie`).

4.1.4 Supervisor Timer Registers (`stime`, `stimecmp`)

Supervisor mode includes a timer facility provided by the `stimecmp` register together with the real-time counter `stime` (`stime` is the supervisor read/write version of the user read-only `time` register). The SBI must provide a facility for determining the timebase of `stime`, which must run at a constant frequency.

The `stimecmp` register has 32-bit precision on all RV32, RV64, and RV128 systems. A timer interrupt is posted when the low 32 bits of the `stime` register match the value in the low 32 bits of the `stimecmp` register. The interrupt remains posted until it is cleared by writing the `stimecmp` register. The interrupt will only be taken if the interrupts are enabled and the STIE bit is set in the `sie` register.

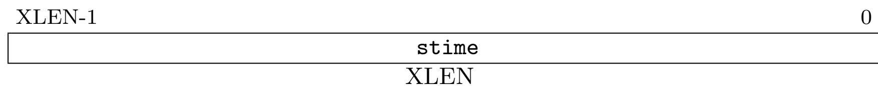


Figure 4.4: Supervisor time register.

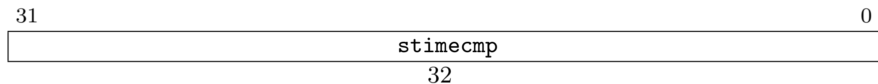


Figure 4.5: Supervisor time compare register.

4.1.5 Supervisor Scratch Register (`sscratch`)

The `sscratch` register is an XLEN-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 4.6: Supervisor Scratch Register.

4.1.6 Supervisor Exception Program Counter (`sepc`)

`sepc` is an XLEN-bit read/write register formatted as shown in Figure 4.7. The low bit of `sepc` (`sepc[0]`) is always zero. On implementations that do not support instruction-set extensions with 16-bit instruction alignment, the two low bits (`sepc[1:0]`) are always zero.

When a trap is taken, `sepc` is written with the virtual address of the instruction that encountered the exception.

Figure 4.7: Supervisor exception program counter register.

4.1.7 Supervisor Cause Register (`scause`)

The `scause` register is an XLEN-bit read-only register formatted as shown in Figure 4.8. The Interrupt bit is set if the exception was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table 4.1 lists the possible exception codes for the current supervisor ISAs.

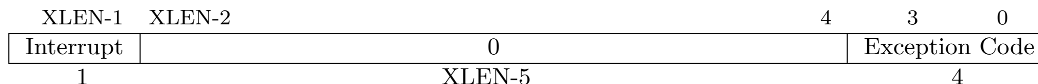


Figure 4.8: Supervisor Cause register.

4.1.8 Supervisor Bad Address (`sbadaddr`) Register

`sbadaddr` is an XLEN-bit read-only register formatted as shown in Figure 4.9. When an instruction fetch address-misaligned exception, or instruction fetch access exception, or AMO address-misaligned exception, or load or store access exception occurs, `sbadaddr` is written with the faulting address. The value in `sbadaddr` is undefined for other exceptions.

For instruction fetch access faults on RISC-V systems with variable-length instructions, `sbadaddr` will point to the portion of the instruction that caused the fault while `sepc` will point to the beginning of the instruction.

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	≥9	*Reserved*
1	0	Software interrupt
1	1	Timer interrupt
1	≥2	*Reserved*

Table 4.1: Supervisor cause register (`scause`) values.

Figure 4.9: Supervisor bad address register.

4.1.9 Supervisor Page-Table Base Register (`sptbr`)

The `sptbr` register is an XLEN-bit read/write register formatted as shown in Figure 4.10. The `sptbr` is only present on system supporting paged virtual-memory systems. This register holds the supervisor physical address of the current root page table, which must be aligned to a 4 KiB boundary.

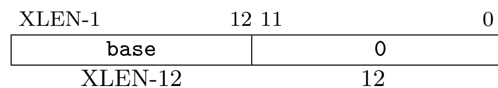


Figure 4.10: Supervisor Page-Table Base Register `sptbr`.

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.

4.1.10 Supervisor Address Space ID Register (sasid)

The sasid register is an ASIDLEN-bit read/write register formatted as shown in Figure 4.11, and is only present on systems supporting paged virtual memory. This register specifies the current address space to facilitate address-translation fences on a per-address-space basis. The SBI should provide a way to obtain ASIDLEN, which is implementation-defined and may be zero if ASIDs are not supported.

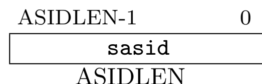
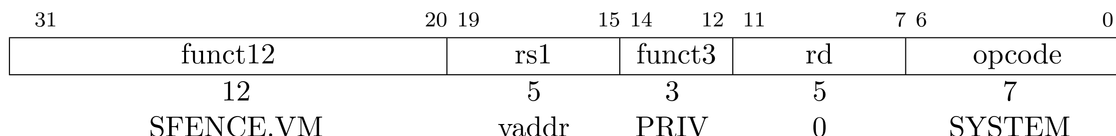


Figure 4.11: Supervisor Address-Space ID Register.

4.2 Supervisor Instructions

In addition to the ECALL, ERET, and EBREAK instructions defined in Section 3.2.1, one new supervisor-level instruction is provided.

4.2.1 Supervisor Memory-Management Fence Instruction



The supervisor memory-management fence instruction SFENCE.VM is used to synchronize updates to in-memory memory-management data structures with current execution. The SFENCE.VM instruction guarantees that any updates to in-memory memory-management data structures (e.g., page tables) will take effect for future instructions executed by this RISC-V thread.

The SFENCE.VM 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.VM is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

Note the instruction has no effect on the translations of other RISC-V threads, which must be notified separately. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VM in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shootdown. Alternatively, implementations might provide direct hardware support for remote TLB invalidation. TLB shootdowns are handled by an SBI call to hide implementation details.

The behavior of SFENCE.VM depends on the current value of the `sasid` register. If `sasid` is nonzero, SFENCE.VM takes effect only for address translations in the current address space. If `sasid` is zero, SFENCE.VM affects address translations for all address spaces. In this case, it also affects *global* mappings, which are described in Section 4.5.1.

The register operand *rs1* contains an optional virtual address argument. If *rs1*=`x0`, the fence affects all virtual address translations. For the common case that the translation data structures have only been modified for a single address mapping (e.g., one page), *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only.

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

4.3 Supervisor Operation in Mbare Environment

When the Mbare environment is selected in the VM field of `mstatus` (Section 3.1.6), supervisor-mode virtual addresses are truncated and mapped directly to supervisor-mode physical addresses. Supervisor physical addresses are then checked using any physical memory protection structures (Sections 3.3–3.4), before being directly converted to machine-level physical addresses.

4.4 Supervisor Operation in Base and Bounds Environments

When Mbb or Mbbid are selected in the VM field of `mstatus` (Section 3.1.6), supervisor-mode virtual addresses are translated and checked according to the appropriate machine-level base and bound registers. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures (Sections 3.3–3.4), before being directly converted to machine-level physical addresses.

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

When Sv32 is written to the VM field in the `mstatus` register, the supervisor operates in a 32-bit paged virtual-memory system. 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 system, and are especially troublesome with decoupled specialized coprocessors. An implementation can choose to implement software TLB refills using a machine-mode trap handler as an extension to M-mode.

4.5.1 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 4.12. When Sv32 virtual memory mode is selected in the VM field of the `mstatus` 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 3.3–3.4), before being directly converted to machine-level physical addresses.

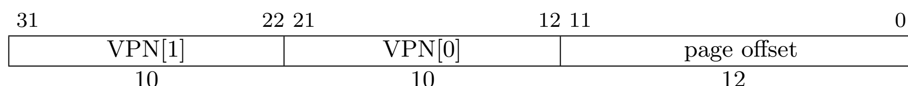


Figure 4.12: Sv32 virtual address.

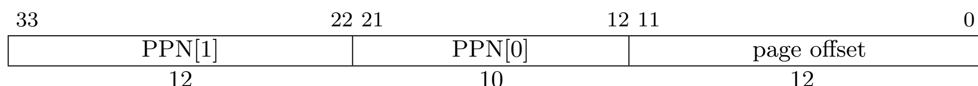


Figure 4.13: Sv32 physical address.

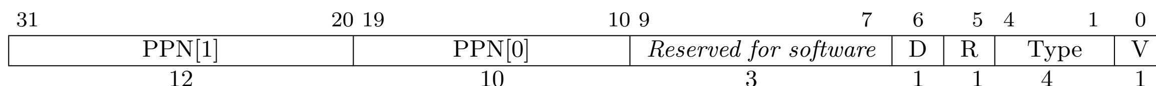


Figure 4.14: Sv32 page table entry.

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 address of the root page table is stored in the `sptbr` register.

The PTE format for Sv32 is shown in Figures 4.14. The V bit indicates whether the PTE is valid; if it is 0, bits 31–1 of the PTE are don't-cares and may be used freely by software. Otherwise, the Type field indicates whether the PTE is a pointer to the next level of the page table or a leaf PTE. If it is the latter, the Type field also encodes the access permissions. Table 4.2 details the Type field encodings.

An alternative PTE format that orthogonalizes supervisor and user permissions would be easier to explain but would require more bits to encode. This would reduce the amount of physical memory that can be addressed with a 32-bit PTE.

Supervisor page mappings may be marked *global* in the Type field. Global mappings are those that exist in all address spaces. For non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global. Note that failing to mark a global mapping as global merely reduces performance, whereas marking a non-global mapping as global is an error.

Global mappings were devised to reduce the cost of context switches. They need not be flushed from an implementation's address translation caches when an SFENCE.VM instruction is executed with a nonzero `sasid` *value.*

Copyright (c) 2010–2015, The Regents of the University of California. All rights reserved.

Type	Meaning	Global	Supervisor R	Supervisor W	Supervisor X	User R	User W	User X
0	Pointer to next level of page table.				—			
1	Pointer to next level of page table—global mapping.	•						
2	Supervisor read-only, user read-execute page.		•			•		•
3	Supervisor read-write, user read-write-execute page.		•	•		•	•	•
4	Supervisor and user read-only page.		•			•		
5	Supervisor and user read-write page.		•	•		•	•	
6	Supervisor and user read-execute page.		•		•	•		•
7	Supervisor and user read-write-execute page.		•	•	•	•	•	•
8	Supervisor read-only page.		•					
9	Supervisor read-write page.		•	•				
10	Supervisor read-execute page.		•		•			
11	Supervisor read-write-execute page.		•	•	•			
12	Supervisor read-only page—global mapping.	•	•					
13	Supervisor read-write page—global mapping.	•	•	•				
14	Supervisor read-execute page—global mapping.	•	•		•			
15	Supervisor read-write-execute page—global mapping.	•	•	•	•			

Table 4.2: Encoding of PTE Type field.

Each leaf PTE maintains a referenced (R) and dirty (D) bit. When a virtual page is read, written, or fetched from, the implementation sets the R bit in the corresponding PTE. When a virtual page is written, the implementation additionally sets the D bit in the corresponding PTE. The access that causes the R and/or D bit to be set must not appear to precede the update of the PTE. Furthermore, the PTE must be updated atomically with respect to other writes to the PTE.

The R and D bits are never cleared by the implementation. If the supervisor software does not rely on referenced and/or dirty bits, e.g. if it does not swap pages to secondary storage, it should always set them to 1 in the PTE. The implementation can then avoid issuing memory accesses to set the bits.

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.

4.5.2 Virtual Address Translation Process

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

1. Let a be the value of the `sptbr` register, and let $i = \text{LEVELS} - 1$. (For Sv32, LEVELS equals 2.)
2. Let *pte* be the value of the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$. (For Sv32, PTESIZE equals 4.)
3. If $pte.v = 0$, stop and signal an address error.

4. Otherwise, $pte.v = 1$. If $pte.type \geq 2$, continue 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 signal an address error. Otherwise, let $a = pte.ppn \times \text{PAGESIZE}$ and go to step 2. (For Sv32, PAGESIZE equals 2^{12}.)

5. A leaf PTE has been found. Determine if the requested memory access is allowed by the *pte.type* field. If not, stop and signal an address error. Otherwise, the translation is successful. Set *pte.r* to 1, and, if the memory access is a store, set *pte.d* to 1. The translated physical address is given as follows:

   - $pa.pgoff = va.pgoff$.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

4.6 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.

4.6.1 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 4.15. Load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else an address exception will occur. The 27-bit VPN is translated into a 38-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

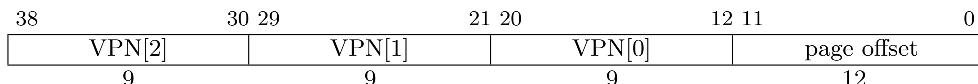


Figure 4.15: Sv39 virtual address.

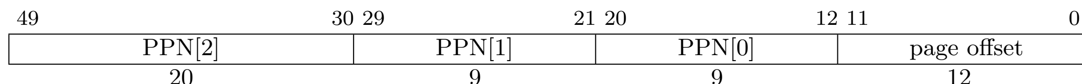


Figure 4.16: Sv39 physical address.

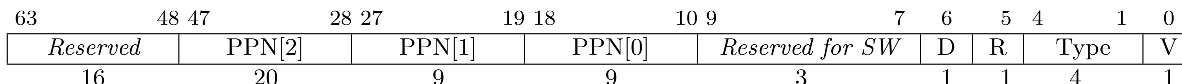


Figure 4.17: Sv39 page table entry.

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 address of the root page table is stored in the `sptbr` register.

The PTE format for Sv39 is shown in Figure 4.17. Bits 9–0 have the same meaning as for Sv32. Bits 63–48 are reserved for future use and must be zeroed by software for forward compatibility.

We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency. These reserved bits may also be used to facilitate research experimentation. The cost is reducing the physical address space, but 1 PiB is presently ample. If at some point 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.

The algorithm for virtual-to-physical address translation is the same as in Section 4.5.2, except LEVELS equals 3 and PTESIZE equals 8.

4.7 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 should also support Sv39.

We specified two 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.

Systems that support Sv48 can also support Sv39 at essentially no cost, and so should do so to support supervisor software that assumes Sv39.

4.7.1 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 4.18. Load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else an address exception will occur. The 36-bit VPN is translated into a 38-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

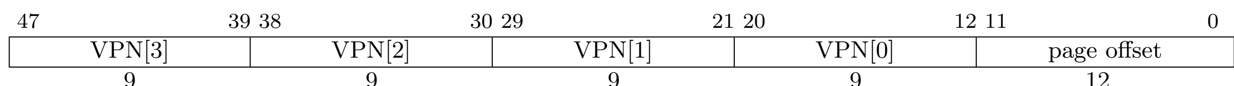


Figure 4.18: Sv48 virtual address.

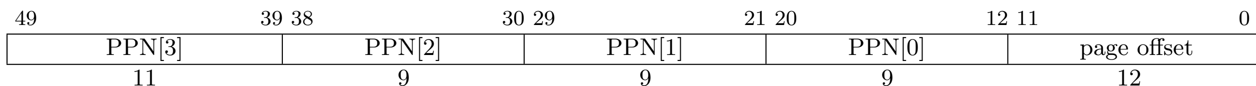


Figure 4.19: Sv48 physical address.

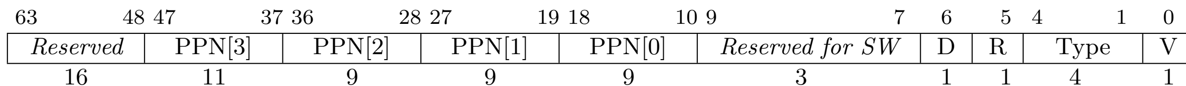


Figure 4.20: Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 4.20. 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.

The algorithm for virtual-to-physical address translation is the same as in Section 4.5.2, except LEVELS equals 4 and PTESIZE equals 8.

Chapter 5

Hypervisor-Level ISA

This chapter is a placeholder for a future RISC-V hypervisor-level common core specification.

The 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.

Chapter 6

RISC-V Privileged Instruction Set Listings

This chapter presents instruction set listings for all instructions defined in the RISC-V Privileged Architecture.

31 27 26 25 24 20	19 15	14 12	11 7	6 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type
Instructions to Access CSRs					
csr	rs1	001	rd	1110011	CSRRW rd,csr,rs1
csr	rs1	010	rd	1110011	CSRRS rd,csr,rs1
csr	rs1	011	rd	1110011	CSRRC rd,csr,rs1
csr	zimm	101	rd	1110011	CSRRWI rd,csr,imm
csr	zimm	110	rd	1110011	CSRRSI rd,csr,imm
csr	zimm	111	rd	1110011	CSRRCI rd,csr,imm
Instructions to Change Privilege Level					
000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK
000100000000	00000	000	00000	1110011	ERET
Trap-Redirection Instructions					
001100000101	00000	000	00000	1110011	MRTS
001100000110	00000	000	00000	1110011	MRTH
001000000101	00000	000	00000	1110011	HRTS
Interrupt-Management Instructions					
000100000010	00000	000	00000	1110011	WFI
Memory-Management Instructions					
000100000001	rs1	000	00000	1110011	SFENCE.VM rs1

Table 6.1: RISC-V Privileged Instructions

Chapter 7

History

Acknowledgements

Thanks to Christopher Celio, David Chisnall, Palmer Dabbelt, Matt Thomas, and Albert Ou for feedback on the privileged specification.

7.1 Funding

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

[3] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.