31 27 26 25	24 20	19 15	14 12	11 7	6 0	
0001000	00010	00000	000	00000	1110011	R sret
0011000	00010	00000	000	00000	1110011	R mret
0001000	00101	00000	000	00000	1110011	R wfi
0001001	rs2	rs1	000	00000	1110011	R sfence.vma

Figure 10.2: RISC-V privileged instruction layout, opcodes, format type, and name. (Table 6.1 of [Waterman and Asanović 2017] is the basis of this figure.)

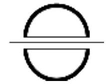
very few instructions; instead, several new control and status registers (CSRs) expose the additional functionality.

This chapter describes the RV32 and RV64 privileged architectures together. Some concepts differ only in the size of an integer register, so to keep the descriptions concise, we introduce the term XLEN to refer to the width of an integer register in bits. XLEN is 32 for RV32 or 64 for RV64.

10.2 Machine Mode for Simple Embedded Systems

Machine mode, abbreviated as M-mode, is the most privileged mode that a RISC-V *hart* (hardware thread) can execute in. Harts running in M-mode have full access to memory, I/O, and low-level system features necessary to boot and configure the system. As such, it is the only privilege mode that all standard RISC-V processors implement; indeed, simple RISC-V microcontrollers support *only* M-mode. Such systems are the focus of this section.

***Hart* is a contraction of *hardware thread*.** We use the term to distinguish them from software threads, which most programmers are familiar with. Software threads are time-multiplexed on harts. Most processor cores have only one hart.



The most important feature of machine mode is the ability to intercept and handle *exceptions*: unusual runtime events. RISC-V classifies exceptions into two categories. *Synchronous exceptions* arise as a result of instruction execution, as when accessing an invalid memory address or executing an instruction with an invalid opcode. *Interrupts* are external events that are asynchronous with the instruction stream, like a mouse button click. Exceptions in RISC-V are *precise*: all instructions prior to the exception completely execute, and none of the subsequent instructions appear to have begun execution. Figure 10.3 lists the standard exception causes.

Five kinds of synchronous exceptions can happen during M-mode execution:

- *Access fault exceptions* arise when a physical memory address doesn't support the access type—for example, attempting to store to a ROM.
- *Breakpoint exceptions* arise from executing an `ebreak` instruction, or when an address or datum matches a debug trigger.
- *Environment call exceptions* arise from executing an `ecall` instruction.
- *Illegal instruction exceptions* result from decoding an invalid opcode.
- *Misaligned address exceptions* occur when the effective address isn't divisible by the access size—for example, `amoadd.w` with an address of `0x12`.

Misaligned instruction address exceptions can't occur with the C extension because it's never possible to jump to an odd address: branches and JAL immediates are always even, and JALR masks off the least-significant bit of its effective address. Without the C extension, this exception occurs when jumping to an address that is 2 mod 4.

If you recall Chapter 2's claim that misaligned loads and stores are permitted, you might be wondering why misaligned load and store address exceptions are listed in Figure 10.3. There are two reasons. First, the atomic memory operations in Chapter 6 require naturally aligned addresses. Second, some implementors choose to omit hardware support for

Interrupt / $\overline{\text{Exception}}$ mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
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 address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

Figure 10.3: RISC-V exception and interrupt causes. The most-significant bit of `mcause` is set to 1 for interrupts or 0 for synchronous exceptions, and the least-significant bits identify the interrupt or exception. Supervisor interrupts and page-fault exceptions are only possible when supervisor mode is implemented (see Section 10.5). (Table 3.6 of [Waterman and Asanović 2017] is the basis of this figure.)

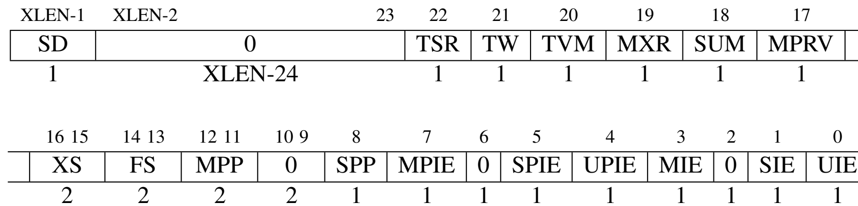


Figure 10.4: The `mstatus` CSR. The only fields present in simple processors with only Machine mode and without the F and V extensions are the global interrupt enable, MIE, and MPIE, which after an exception holds the old value of MIE. XLEN is 32 for RV32, or 64 for RV64. Figure 3.7 of [Waterman and Asanović 2017] is the basis of this figure; see Section 3.1 of that document for a description of the other fields.

misaligned regular loads and stores, because it is a difficult feature to implement and is infrequently used. Processors without this hardware rely instead upon an exception handler to trap and emulate misaligned loads and stores in software, using a sequence of smaller, aligned loads and stores. Application code is none the wiser: misaligned memory accesses work as expected, albeit slowly, while the hardware remains simple. Alternatively, more performant processors can implement misaligned loads and stores in hardware. This implementation flexibility owes to RISC-V's decision to permit misaligned loads and stores using the regular load and store opcodes, following Chapter 1's guideline to isolate architecture from implementation.

There are three standard sources of interrupts: software, timer, and external. Software interrupts are triggered by storing to a memory-mapped register and are generally used by one hart to interrupt another hart, a mechanism other architectures refer to as an *interprocessor interrupt*. Timer interrupts are raised when a hart's time comparator, a memory-mapped register named `mtimecmp`, exceeds the real-time counter `mtime`. External interrupts are raised by a platform-level interrupt controller, to which most external devices are attached. As different hardware platforms have different memory maps and demand divergent features of their interrupt controllers, the mechanisms for raising and clearing these interrupts differ from platform to platform. What *is* constant across all RISC-V systems is how exceptions are handled and interrupts are masked, the topic of the next section.

10.3 Machine-Mode Exception Handling

Eight control and status registers (CSRs) are integral to machine-mode exception handling:

- `mtvec`, *Machine Trap Vector*, holds the address the processor jumps to when an exception occurs.
- `mepc`, *Machine Exception PC*, points to the instruction where the exception occurred.
- `mcause`, *Machine Exception Cause*, indicates which exception occurred.
- `mie`, *Machine Interrupt Enable*, lists which interrupts the processor can take and which it must ignore.
- `mip`, *Machine Interrupt Pending*, lists the interrupts currently pending.

Encoding	Name	Abbreviation
00	User	U
01	Supervisor	S
11	Machine	M

Figure 10.5: RISC-V privilege levels and their encoding.

- mtval, *Machine Trap Value*, holds additional trap information: the faulting address for address exceptions, the instruction itself for illegal instruction exceptions, and zero for other exceptions.
- mscratch, *Machine Scratch*, holds one word of data for temporary storage.
- mstatus, *Machine Status*, holds the global interrupt enable, along with a plethora of other state, as Figure 10.4 shows.

When executing in M-mode, interrupts are only taken if the global interrupt-enable bit, mstatus.MIE, is set. Furthermore, each interrupt has its own enable bit in the mie CSR. The bit positions in mie correspond to the interrupt codes in Figure 10.3: for example, mie[7] corresponds to the M-mode timer interrupt. The mip CSR has the same layout and indicates which interrupts are currently pending. Putting all three CSRs together, a machine timer interrupt can be taken if mstatus.MIE=1, mie[7]=1, and mip[7]=1.

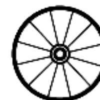
RISC-V also supports vectored interrupts, wherein the processor jumps to an interrupt-specific address, rather than a single entry point. This addressing eliminates the need to read and decode mcause, speeding up interrupt handling. Setting mtval[0] to 1 enables this feature; interrupt cause *x* then sets the PC to (mtval-1+4*x*), instead of the usual mtval.

When a hart takes an exception, the hardware atomically undergoes several state transitions:

- The PC of the exceptional instruction is preserved in mepc, and the PC is set to mtvec. (For synchronous exceptions, mepc points to the instruction that caused the exception; for interrupts, it points where execution should resume after the interrupt is handled.)
- mcause is set to the exception cause, as encoded in Figure 10.3, and mtval is set to the faulting address or some other exception-specific word of information.
- Interrupts are disabled by setting MIE=0 in the mstatus CSR, and the previous value of MIE is preserved in MPIE.
- The pre-exception privilege mode is preserved in mstatus' MPP field, and the privilege mode is changed to M. Figure 10.5 shows the encoding of the MPP field. (If the processor only implements M-mode, this step is effectively skipped.)

To avoid overwriting the contents of the integer registers, the prologue of an interrupt handler usually begins by swapping an integer register (say, a0) with the mscratch CSR. Usually, the software will have arranged for mscratch to contain a pointer to additional in-memory scratch space, which the handler uses to save as many integer registers as its body will use. After the body executes, the epilogue of an interrupt handler restores the registers it saved to memory, then again swaps a0 with mscratch, restoring both registers to their pre-exception values. Finally, the handler returns with mret, an instruction unique to M-mode. mret sets the PC to mepc, restores the previous interrupt-enable setting by copying the mstatus MPIE field to MIE, and sets the privilege mode to the value in mstatus' MPP field, essentially reversing the actions described in the preceding paragraph.

Figure 10.6 shows RISC-V assembly code for a basic timer interrupt handler following this pattern. It simply increments the time comparator then returns to the previous task, whereas a more realistic timer interrupt handler might invoke a scheduler to switch between tasks. It is not preemptible, so it keeps interrupts disabled throughout the handler. Those caveats aside, it is a complete example of a RISC-V interrupt handler on a single page!



Sometimes it's desirable to take a higher-priority interrupt while processing a lower-priority exception. Alas, there's only one copy of the `mepc`, `mcause`, `mtval`, and `mstatus` CSRs; taking a second interrupt would destroy the old values in these registers, causing data loss without some additional help from software. A preemptible interrupt handler can save these registers to an in-memory stack before enabling interrupts, then, just prior to exiting, disable interrupts and restore the registers from the stack.

In addition to the `mret` instruction we introduced above, M-mode provides just one other instruction: `wfi` (*Wait For Interrupt*). `wfi` informs the processor that there isn't any useful work to do, so it should enter a lower-power mode until any enabled interrupt becomes pending, i.e., (`mie` & `mip`)$\neq$0. RISC-V processors implement this instruction in a variety of ways, including stopping the clock until an interrupt becomes pending; some simply execute it as a `nop`. Hence, `wfi` is typically used inside a loop.

■ ***Elaboration: `wfi` works whether or not interrupts are globally enabled.***

If `wfi` is executed when interrupts are globally enabled (`mstatus`.MIE=1), and then an enabled interrupt becomes pending, the processor jumps to the exception handler. If, on the other hand, `wfi` is executed when interrupts are globally disabled, and then an enabled interrupt becomes pending, the processor continues executing the code following the `wfi`. This code typically examines the `mip` CSR to decide what to do next. This strategy can reduce interrupt latency as compared to jumping to the exception handler, because there's no need to save and restore integer registers.

10.4 User Mode and Process Isolation in Embedded Systems

Although Machine mode is sufficient for simple embedded systems, it is only suitable when the entire codebase is trusted, since M-mode has unfettered access to the hardware platform. More often, it is not practical to trust all of the application code, because it is not known in advance or is too vast to prove correct. So, RISC-V provides mechanisms to protect the system from the untrusted code, and to protect untrusted processes from each other.

Untrusted code must be forbidden from executing privileged instructions, like `mret`, and accessing privileged CSRs, like `mstatus`, as these would allow the program to take control of the system. This restriction is accomplished easily enough: an additional privilege mode, *User mode* (U-mode), denies access to these features, generating an illegal instruction exception when attempting to use an M-mode instruction or CSR. Otherwise, U-mode and M-mode behave very similarly. M-mode software can enter U-mode by setting `mstatus`.MPP to U (which, as Figure 10.5 shows, is encoded as 0), then executing an `mret` instruction. If an exception occurs in U-mode, control is returned to M-mode.

Untrusted code must also be restricted to access only its own memory. Processors that implement M and U modes have a feature called *Physical Memory Protection* (PMP), which allows M-mode to specify which memory addresses U-mode can access. PMP consists of several address registers (usually eight to sixteen) and corresponding configuration registers, which grant or deny read, write, and execute permissions. When a processor in U-mode

```
# save registers
csrrw a0, mscratch, a0  # save a0; set a0 = &temp storage
sw a1, 0(a0)            # save a1
sw a2, 4(a0)            # save a2
sw a3, 8(a0)            # save a3
sw a4, 12(a0)           # save a4

# decode interrupt cause
csrr a1, mcause         # read exception cause
bgez a1, exception      # branch if not an interrupt
andi a1, a1, 0x3f       # isolate interrupt cause
li a2, 7                # a2 = timer interrupt cause
bne a1, a2, otherInt    # branch if not a timer interrupt

# handle timer interrupt by incrementing time comparator
la a1, mtimecmp         # a1 = &time comparator
lw a2, 0(a1)            # load lower 32 bits of comparator
lw a3, 4(a1)            # load upper 32 bits of comparator
addi a4, a2, 1000       # increment lower bits by 1000 cycles
sltu a2, a4, a2         # generate carry-out
add a3, a3, a2          # increment upper bits
sw a3, 4(a1)            # store upper 32 bits
sw a4, 0(a1)            # store lower 32 bits

# restore registers and return
lw a4, 12(a0)           # restore a4
lw a3, 4(a0)            # restore a3
lw a2, 4(a0)            # restore a2
lw a1, 0(a0)            # restore a1
csrrw a0, mscratch, a0  # restore a0; mscratch = &temp storage
mret                    # return from handler
```

Figure 10.6: RISC-V code for a simple timer interrupt handler. The code assumes that interrupts are globally enabled by setting `mstatus`.MIE; that timer interrupts have been enabled by setting `mie`[7]; that the `mtvec` CSR has been set to the address of this handler; and that the `mscratch` CSR has been set to the address of a buffer that contains 16 bytes of temporary storage to save the registers. The prologue saves five registers, preserving `a0` in `mscratch` and `a1`–`a4` in memory. It then decodes the exception cause by examining `mcause`: interrupt if `mcause`<0, or synchronous exception if `mcause`≥0. If it is an interrupt, it checks that the lower bits of `mcause` equal 7, indicating an M-mode timer interrupt. If it is a timer interrupt, it adds 1000 cycles to the time comparator, so that the next timer interrupt will occur about 1000 timer cycles in the future. Finally, the epilogue restores the `a0`–`a4` and `mscratch`, then returns whence it came using `mret`.

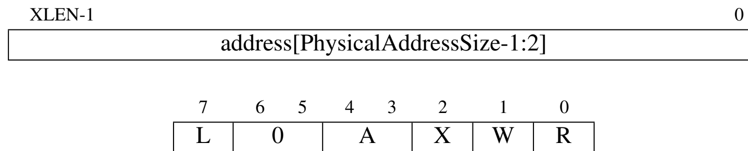


Figure 10.7: A PMP address and configuration register. The address register is right-shifted by 2, and if physical addresses are less than XLEN-2 bits wide, the upper bits are zeros. The R, W, and X fields grant read, write, and execute permissions. The A field sets the PMP mode, and the L field locks the PMP and corresponding address registers.

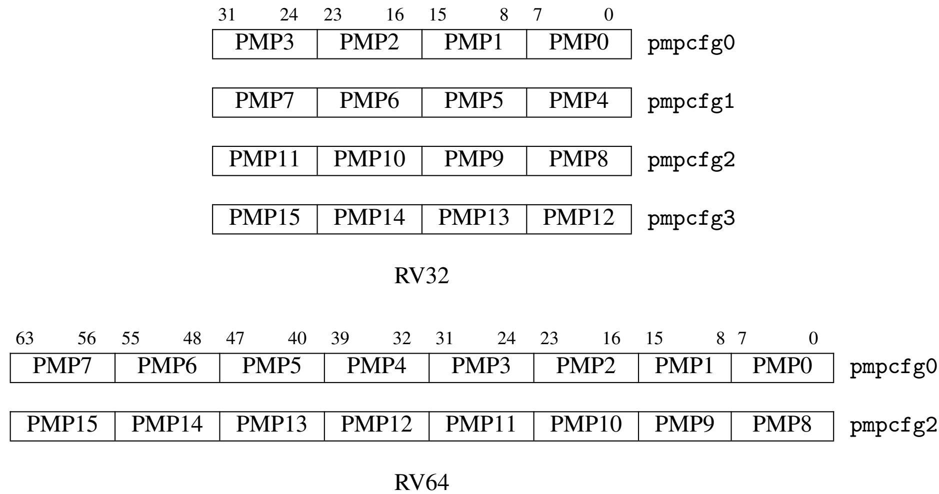


Figure 10.8: The layout of PMP configurations in the `pmpcfg` CSRs. For RV32 (above), the sixteen configuration registers are packed into four CSRs. For RV64 (below), they are packed into the two even-numbered CSRs.

attempts to fetch an instruction, or execute a load or store, the address is compared against all of the PMP address registers. If the address is greater than or equal to PMP address i, but less than PMP address i+1, then PMP i+1's configuration register decides whether that access may proceed; otherwise, it raises an access exception.

Figure 10.7 shows the layout of a PMP address and configuration register. Both are CSRs, with the address registers named `pmpaddr0` to `pmpaddr`N, where N+1 is the number of PMPs implemented. The address registers are shifted right two bits because PMPs have a four-byte granularity. The configuration registers are densely packed in the CSRs to accelerate context switching, as Figure 10.8 shows. A PMP's configuration consists of R, W, and X bits, which when set permit loads, stores, and fetches, respectively, and a mode field, A, which when 0 disables this PMP or when 1 enables it. The PMP configuration also supports other modes and can be locked, features described in [Waterman and Asanović 2017].

10.5 Supervisor Mode for Modern Operating Systems

The PMP scheme described in the previous section is attractive for embedded systems because it provides memory protection at relatively low cost, but it has several drawbacks that limit its use in general-purpose computing. Since PMP supports only a fixed number of memory regions, it doesn't scale to complex applications. And since these regions must be contiguous in physical memory, the system can suffer from memory fragmentation. Finally, PMP doesn't efficiently support paging to secondary storage.

Fragmentation occurs when memory is available, but not in large enough contiguous chunks to be useful.

More sophisticated RISC-V processors handle these problems the same way as nearly all general-purpose architectures: using page-based virtual memory. This feature forms the core of *supervisor mode* (S-mode), an optional privilege mode designed to support modern Unix-like operating systems, such as Linux, FreeBSD, and Windows. S-mode is more privileged than U-mode, but less-privileged than M-mode. Like U-mode, S-mode software can't use M-mode CSRs and instructions, and is subject to PMP restrictions. This section covers S-mode interrupts and exceptions, and the next section details the S-mode virtual-memory system.

Why not unconditionally delegate interrupts to S-mode? One reason is virtualization: if M-mode wants to virtualize a device for S-mode, its interrupts should go to M-mode, not S-mode.

By default, all exceptions, regardless of privilege mode, transfer control to the M-mode exception handler. Most exceptions in a Unix system, though, should invoke the operating system, which runs in S-mode. The M-mode exception handler could re-route exceptions to S-mode, but this extra code would slow down the handling of most exceptions. So, RISC-V provides an *exception delegation* mechanism, by which interrupts and synchronous exceptions can be delegated to S-mode selectively, bypassing M-mode software altogether.

The `mideleg` (*Machine Interrupt Delegation*) CSR controls which interrupts are delegated to S-mode. Like `mip` and `mie`, each bit in `mideleg` corresponds to the exception code of the same number in Figure 10.3. For example, `mideleg`[5] corresponds to the S-mode timer interrupt; if set, S-mode timer interrupts will transfer control to the S-mode exception handler, rather than the M-mode exception handler.

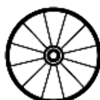
S-mode doesn't directly control timer and software interrupts but instead uses the `ecall` instruction to request M-mode to set up timers or send interprocessor interrupts on its behalf. This software convention is part of the *Supervisor Binary Interface*.

Any interrupt delegated to S-mode can be masked by S-mode software. The `sie` (*Supervisor Interrupt Enable*) and `sip` (*Supervisor Interrupt Pending*) CSRs are S-mode CSRs that are subsets of the `mie` and `mip` CSRs. They have the same layout as their M-mode counterparts, but only the bits corresponding to interrupts that have been delegated in `mideleg` are readable and writable through `sie` and `sip`. The bits corresponding to interrupts that haven't been delegated are always zero.

M-mode can also delegate synchronous exceptions to S-mode using the `medeleg` (*Machine Exception Delegation*) CSR. The mechanism is analogous to interrupt delegation, but the bits in `medeleg` correspond instead to the synchronous exception codes in Figure 10.3. For example, setting `medeleg`[15] will delegate store page faults to S-mode.

Note that exceptions will never transfer control to a less-privileged mode, no matter the delegation settings. An exception that occurs in M-mode is always handled in M-mode. An exception that occurs in S-mode might be handled by either M-mode or S-mode, depending on the delegation configuration, but never U-mode.

S-mode has several exception-handling CSRs, `sepc`, `stvec`, `scause`, `sscratch`, `stval`, and `sstatus`, which perform the same function as their M-mode counterparts described in Section 10.2. Figure 10.9 shows the layout of the `sstatus` register. The supervisor exception return instruction, `sret`, behaves the same as `mret`, but it acts on the S-mode exception-handling CSRs instead of the M-mode ones.



The act of taking an exception is also very similar to M-mode. If a hart takes an exception and it is delegated to S-mode, the hardware atomically undergoes several similar state

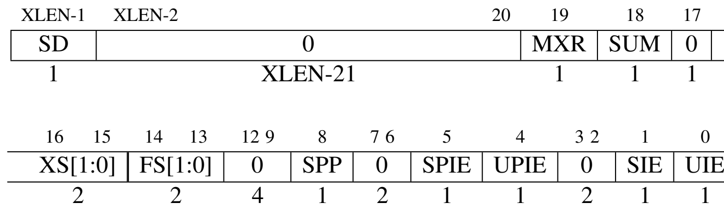


Figure 10.9: The `sstatus` CSR. `sstatus` is a subset of `mstatus` (Figure 10.4), hence the similar layout. SIE and SPIE hold the current and pre-exception interrupt enables, analogous to MIE and MPIE in `mstatus`. XLEN is 32 for RV32, or 64 for RV64. Figure 4.2 of [Waterman and Asanović 2017] is the basis of this figure; see Section 4.1 of that document for a description of the other fields.

transitions, using S-mode CSRs instead of M-mode ones:

- The PC of the exceptional instruction is preserved in `sepc`, and the PC is set to `stvec`.
- `scause` is set to the exception cause, as encoded in Figure 10.3, and `stval` is set to the faulting address or some other exception-specific word of information.
- Interrupts are disabled by setting SIE=0 in the `sstatus` CSR, and the previous value of SIE is preserved in SPIE.
- The pre-exception privilege mode is preserved in `sstatus`' SPP field, and the privilege mode is changed to S.

10.6 Page-Based Virtual Memory

S-mode provides a conventional virtual memory system that divides memory into fixed-size *pages* for the purposes of address translation and memory protection. When paging is enabled, most addresses (including load and store effective addresses and the PC) are *virtual addresses* that must be translated into *physical addresses* in order to access physical memory. Virtual addresses are translated to physical addresses by traversing a high-radix tree known as the *page table*. A leaf node in the page table indicates whether the virtual address maps to a physical page, and, if so, which privilege modes and access types have permission to access the page. Accessing a page that is unmapped or grants insufficient permissions results in a *page fault exception*.

4 KiB pages have been popular for five decades starting with the IBM 360 model 67. Atlas, the first computer with paging, had 3 KiB pages (it had 6-byte words). We find it remarkable that, after a half-century of exponential growth in computer performance and memory capacity, the page size remains virtually unchanged.

RISC-V paging schemes are named Sv*X*, where *X* is the size of a virtual address in bits. RV32's paging scheme, Sv32, supports a 4 GiB virtual-address space, which is divided into 2^{10} *megapages* of size 4 MiB. Each megapage is subdivided into 2^{10} *base pages*—the fundamental unit of paging—each 4 KiB. Hence, Sv32's page table is a two-level tree of radix 2^{10}. Each entry in the page table is four bytes, so a page table is itself 4 KiB. It's no coincidence that a page table is exactly the size of a page: this design simplifies operating-system memory allocation.

Figure 10.10 shows the layout of an Sv32 page-table entry (PTE), which has the following fields, explained from right to left:

- The V bit indicates whether the rest of this PTE is valid (V=1). If V=0, any virtual-address translation that traverses this PTE results in a page fault.

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

Figure 10.10: An RV32 Sv32 page-table entry (PTE).

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

Figure 10.11: An RV64 Sv39 page-table entry (PTE).

- The R, W, and X bits indicate whether the page has read, write, and execute permissions, respectively. If all three bits are 0, this PTE is a pointer to the next level of the page table; otherwise, it's a leaf of the tree.
- The U bit indicates whether this page is a user page. If U=0, U-mode cannot access this page, but S-mode can. If U=1, U-mode can access this page, but S-mode cannot.

The OS relies on the A and D bits to decide which pages to swap to secondary storage. Periodically clearing the A bits helps the OS approximate which pages have been least-recently used. The D bit indicates a page is even more expensive to swap out, because it must be written back to secondary storage.

- The G bit indicates this mapping exists in all virtual-address spaces, information the hardware can use to improve address-translation performance. It is typically only used for pages that belong to the operating system.
- The A bit indicates whether the page has been accessed since the last time the A bit was cleared.
- The D bit indicates whether the page has been dirtied (i.e., written) since the last time the D bit was cleared.
- The RSW field is reserved for the operating system's use; the hardware ignores it.
- The PPN field holds a physical page number, which is part of a physical address. If this PTE is a leaf, the PPN is part of the translated physical address. Otherwise, the PPN gives the address of the next level of the page table. (Figure 10.10 divides the PPN into two subfields to simplify the description of the address-translation algorithm.)

The other RV64 paging schemes simply add more levels to the page table. Sv48 is nearly identical to Sv39, but its virtual-address space is 2^9 times bigger and its page table is one level deeper.

RV64 supports multiple paging schemes, but we describe only the most popular one, Sv39. Sv39 uses the same 4 KiB base page as Sv32. The page-table entries double in size to eight bytes so they can hold bigger physical addresses. To maintain the invariant that a page table is exactly the size of a page, the radix of the tree correspondingly falls to 2^9. The tree is three levels deep. Sv39's 512 GiB address space is divided into 2^9 *gigapages*, each 1 GiB. Each gigapage is subdivided into 2^9 megapages, which in Sv39 are slightly smaller than in Sv32: 2 MiB. Each megapage is subdivided into 2^9 4 KiB base pages.

Figure 10.11 shows the layout of an Sv39 PTE. It's identical to an Sv32 PTE, except the PPN field has been widened to 44 bits to support 56-bit physical addresses, or 2^{26} GiB of physical-address space.

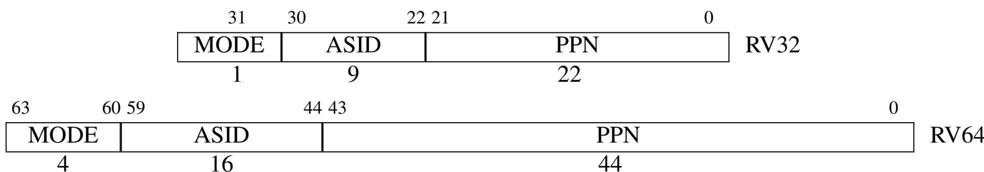


Figure 10.12: The `satp` CSR. Figures 4.11 and 4.12 of [Waterman and Asanović 2017] are the bases for this figure.

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

RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

Figure 10.13: The encoding of the MODE field in the `satp` CSR. Table 4.3 of [Waterman and Asanović 2017] is the basis for this figure.

■ ***Elaboration: Unused address bits***

Since Sv39's virtual addresses are narrower than an RV64 integer register, you might wonder what becomes of the remaining 25 bits. Sv39 mandates that address bits 63–39 be copies of bit 38. Thus, the valid virtual addresses are `0000_0000_0000_0000`$_{hex}$–`0000_003f_ffff_ffff`$_{hex}$ and `ffff_ffc0_0000_0000`$_{hex}$–`ffff_ffff_ffff_ffff`$_{hex}$. The gap between these two ranges is, of course, 2^{25} times bigger than the size of the two ranges combined, seemingly wasting 99.999997% of the values a 64-bit register can represent. Why not make better use of those extra 25 bits? The answer is that, as programs grow to require more than 512 GiB of virtual-address space, architects want to increase the address space without breaking backwards compatibility. If we allowed programs to store extra data in the upper 25 bits, it would be impossible to later reclaim those bits to hold bigger addresses. Allowing data storage in unused address bits is a grievous error, but one that has recurred many times in computing history.

An S-mode CSR, `satp` (*Supervisor Address Translation and Protection*), controls the paging system. As Figure 10.12 shows, `satp` has three fields. The MODE field enables paging and selects the page-table depth; Figure 10.13 shows its encoding. The ASID (*Address Space Identifier*) field is optional and can be used to reduce the cost of context switches. Finally, the PPN field holds the physical address of the root page table, divided by the 4 KiB page size. Typically, M-mode software will write zero to `satp` before entering S-mode for the first time, disabling paging, then S-mode software will write it again after setting up the page tables.

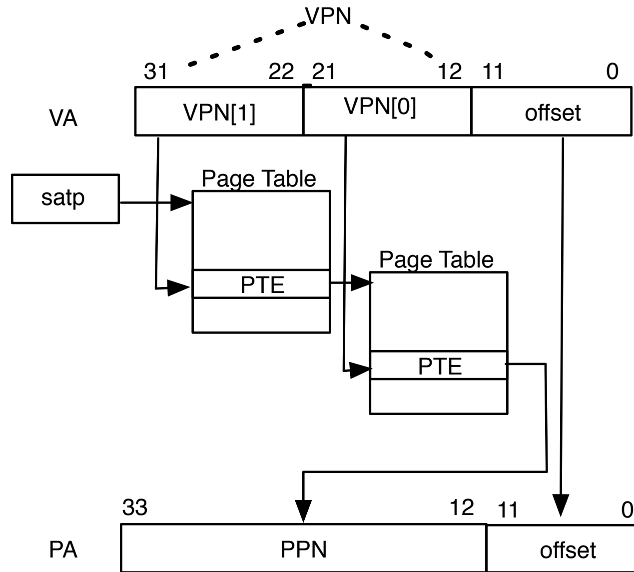


Figure 10.14: Diagram of the Sv32 address-translation process.

When paging is enabled in the `satp` register, S-mode and U-mode virtual addresses are translated into physical addresses by traversing the page table, starting at the root. Figure 10.14 depicts this process:

1. `satp`.PPN gives the base address of the first-level page table, and VA[31:22] gives the first-level index, so the processor reads the PTE located at address (`satp`.PPN×4096 + VA[31:22]×4).
2. That PTE contains the base address of the second-level page table and VA[21:12] gives the second-level index, so the processor reads the leaf PTE located at (PTE.PPN×4096 + VA[21:12]×4).
3. The leaf PTE's PPN field and the *page offset* (the twelve least-significant bits of the original virtual address) form the final result: the physical address is (LeafPTE.PPN×4096 + VA[11:0]).

The processor then performs the physical memory access. The translation process is almost the same for Sv39 as for Sv32, but with larger PTEs and one more level of indirection. Figure 10.19, at the end of this chapter, gives a complete description of the page-table traversal algorithm, detailing the exceptional conditions and the special case of superpage translations.

That's almost all there is to the RISC-V paging system, save for one wrinkle. If all instruction fetches, loads, and stores resulted in several page-table accesses, then paging would reduce performance substantially! All modern processors reduce this overhead with an address-translation cache (often called a *TLB*, for Translation Lookaside Buffer). To reduce the cost of this cache, most processors don't automatically keep it coherent with the page table—if the operating system modifies the page table, the cache becomes stale. S-mode adds one more instruction to solve this problem: `sfence.vma` informs the processor

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	WIRI	SEIP	UEIP	MTIP	WIRI	STIP	UTIP	MSIP	WIRI	SSIP	USIP
WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 10.15: Machine interrupt registers. They are XLEN-bit read/write registers that hold pending interrupts (`mip`) and the interrupt enable bits (`mie`) CSRs. Only the bits corresponding to lower-privilege software interrupts (USIP, SSIP), timer interrupts (UTIP, STIP), and external interrupts (UEIP, SEIP) in `mip` are writable through this CSR address; the remaining bits are read-only.

XLEN-1 10	9	8	7 6	5	4	3 2	1	0
WIRI	SEIP	UEIP	WIRI	STIP	UTIP	WIRI	SSIP	USIP
WPRI	SEIE	UEIE	WPRI	STIE	UTIE	WPRI	SSIE	USIE
XLEN-10	1	1	2	1	1	2	1	1

Figure 10.16: Supervisor interrupt registers. They are XLEN-bit read/write registers that hold pending interrupts (`sip`) and the interrupt enable bits (`sie`) CSRs.

that software may have modified the page tables, so the processor can flush the translation caches accordingly. It takes two optional arguments, which narrow the scope of the cache flush: rs1 indicates which virtual address' translation has been modified in the page table, and rs2 gives the address-space identifier of the process whose page table has been modified. If x0 is given for both, the entire translation cache is flushed.

■ ***Elaboration: Address-translation cache coherence in multiprocessors***

`sfence.vma` only affects the address-translation hardware for the hart that executed the instruction. When a hart changes a page table that another hart is using, the first hart must use an interprocessor interrupt to inform the second hart that it should execute an `sfence.vma` instruction. This procedure is often referred to as a *TLB shootdown*.

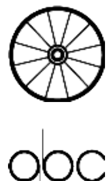
10.7 Concluding Remarks

Study after study shows that the very best designers produce structures that are faster, smaller, simpler, clearer, and produced with less effort. The differences between the great and the average approach an order of magnitude.

—Fred Brooks, Jr., 1986.

Brooks is a Turing Award laureate and an architect of the IBM System/360 family of computers, which demonstrated the importance of differentiating architecture from implementation. Descendants of that 1964 architecture are still selling today.

The modularity of the RISC-V privileged architectures caters to the needs of a variety of systems. The minimalist Machine mode supports bare-metal embedded applications at low cost. The additional User mode and Physical Memory Protection together enable multitasking in more sophisticated embedded systems. Finally, Supervisor mode and page-based virtual memory provide the flexibility needed to host modern operating systems.



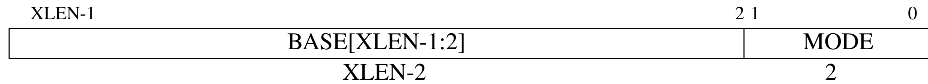


Figure 10.17: Machine and supervisor trap-vector base-address register (`mtvec` and `stvec`) CSRs. They are XLEN-bit read/write registers that hold trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE). The value in the BASE field must always be aligned on a 4-byte boundary. MODE = 0 means all exceptions set the PC to BASE. MODE = 1 sets the PC to $(BASE + (4 \times cause))$ on asynchronous interrupts.

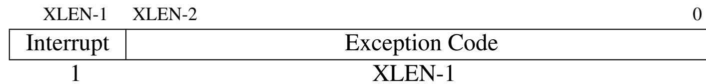


Figure 10.18: Machine and supervisor cause (`mcause` and `scause`) CSRs. When a trap is taken, the CSR is written with a code indicating the event that caused the trap. The Interrupt bit is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Tables 3.6 and 4.2 of [Waterman and Asanović 2017] map the code values to the reason for the traps.

10.8 To Learn More

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017. URL `https://riscv.org/specifications/privileged-isa/`.

Notes

[1] `http://parlab.eecs.berkeley.edu`

1. Let a be `satp`.ppn $\times$ PAGESIZE, and let i = LEVELS $-$ 1.
2. Let pte be the value of the PTE at address $a + va.vpn[i] \times$ PTESIZE.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, $pte.x$, and $pte.u$ bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception.
6. If $i > 0$ and $pa.ppn[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception.
7. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, then either:
   - Raise a page-fault exception, or:
   - Set $pte.a$ to 1 and, if the memory access is a store, also set $pte.d$ to 1.
8. The translation is successful. The translated physical address is given as follows:
   - *pa.pgoff* = *va.pgoff*.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn$[LEVELS $- 1 : i$] $= pte.ppn$[LEVELS $- 1 : i$].

Figure 10.19: The complete algorithm for virtual-to-physical address translation. va is the virtual address input and pa is the physical address output. The PAGESIZE constant is 2^{12}. For Sv32, LEVELS=2 and PTESIZE=4, whereas for Sv39, LEVELS=3 and PTESIZE=8. Section 4.3.2 of [Waterman and Asanović 2017] is the basis for this figure.

11 Future RISC-V Optional Extensions

Alan Perlis (1922–1990) was the first recipient of the Turing Award (1966), conferred for his influence on advanced programming languages and compilers. In 1958 he helped design ALGOL, which has influenced virtually every imperative programming language including C and Java.

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

—Alan Perlis, 1982

The RISC-V Foundation will develop at least eight optional extensions.

11.1 "B" Standard Extension for Bit Manipulation

The B extension offers bit manipulation, including insert, extract, and test bit fields; rotations; funnel shifts; bit and byte permutations; count leading and trailing zeros; and count bits set.

11.2 "E" Standard Extension for Embedded

To reduce the cost of low-end cores, it has 16 fewer registers. RV32E is why the saved and temporary registers are split between the registers 0-15 and 16-31 (Figure 3.2).

11.3 "H" Privileged Architecture Extension for Hypervisor Support

The H extension to the privileged architecture adds a new *hypervisor* mode and a second level of page-based address translation to improve the efficiency of running multiple operating systems on the same machine.

11.4 "J" Standard Extension for Dynamically Translated Languages

Many popular languages are usually implemented via dynamic translation, including Java and Javascript. These languages can benefit from additional ISA support for dynamic checks and garbage collection. (J stands for *Just-In-Time* compiler.)

11.5 "L" Standard Extension for Decimal Floating-Point

The L extension is intended to support decimal floating-point arithmetic as defined in the IEEE 754-2008 standard. The problem with binary numbers is that they cannot represent some common decimal fractions, such as 0.1. The motivation for RV32L is that the computation radix can be identical to the radix of the input and output.