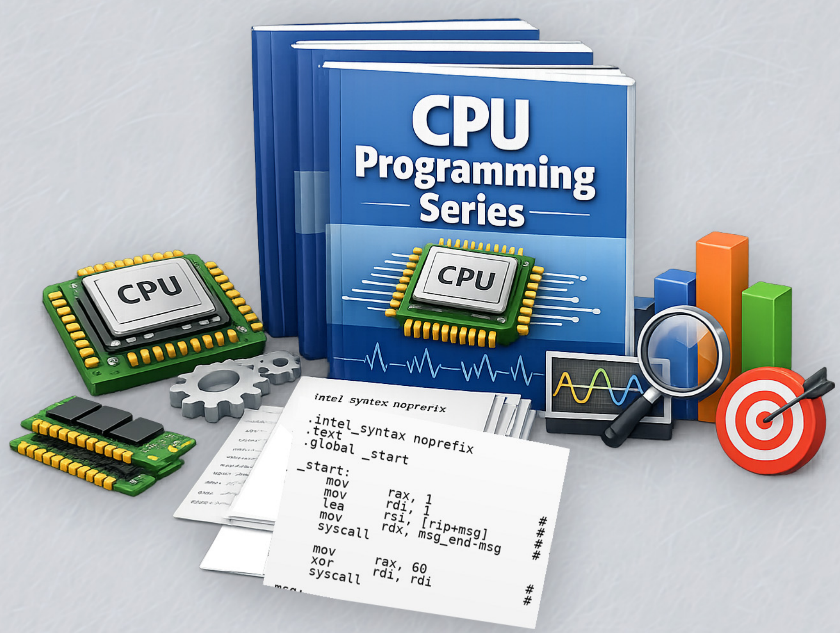
https://simplifycpp.org

CPU Programming Series

RISC-V Privilege, Traps, and Syscalls

U / S / M Modes Explained



```
intel syntex noprerix
.intel_syntax noprefix
.text
.global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [rip+msg]          #
    mov     rdx, msg_end-msg        #
    syscall                         #
                                    #
    mov     rax, 60
    xor     rdi, rdi                #
    syscall                         #
```

17

Prepared by Ayman Alheraki

CPU Programming Series

RISC-V Privilege, Traps, and Syscalls

U / S / M Modes Explained

Prepared by Ayman Alheraki

simplifycpp.org

February 2026

Contents

Preface

Why Privilege, Traps, and Syscalls Matter

Modern systems are not defined only by the instructions an application can execute, but by the **boundaries** that prevent it from executing everything. The **RISC-V privilege architecture** is the contract that separates:

- **Applications (U-mode)** that must be isolated and untrusted by default.
- **Operating system kernels (S-mode)** that manage processes, memory, files, and devices.
- **Platform firmware / machine control (M-mode)** that owns the hardware and defines the lowest-level policy.

The mechanism that makes these boundaries **enforceable** is the trap system:

- **Exceptions** (synchronous): illegal instructions, access faults, page faults, environment calls.
- **Interrupts** (asynchronous): timer, external devices, inter-processor events.

A syscall is not a normal function call. It is a **controlled privilege transition**: **User code requests a service** that only the kernel or firmware is allowed to perform, and the CPU performs a **trap** that transfers control into a higher-privilege handler.
This booklet focuses on the exact mental model you must internalize to write correct systems code:

- **Privilege is a hardware state** (current mode) with strict rules.
- **Trap entry is a hardware control transfer**, not a branch you can "undo".
- **The handler must preserve architectural state explicitly** (registers, stack discipline, return CSRs).
- **Syscalls are ABIs**, not just instructions: register conventions, return rules, and error reporting.

Why this matters for programmers:

- If you misunderstand traps, you will corrupt state (registers/stack), break re-entrancy, or return to the wrong mode.
- If you misunderstand privilege, you will accidentally design insecure or non-portable low-level code.
- If you misunderstand syscalls, your code will appear correct but fail under real kernels and real toolchains.

Example 1: A syscall is a trap, not a call

A typical OS syscall path is:

- U-mode places arguments in agreed registers.

- U-mode executes `ecall`.
- CPU traps into S-mode (or into M-mode, depending on environment and delegation).
- Handler runs with higher privilege, then returns using `sret` or `mret`.

```
/* User-mode: request a service from the kernel via ECALL.
   NOTE: exact syscall numbers and error conventions are OS-defined.
    ↪  */
.text
.global user_do_syscall
user_do_syscall:
    /* a0..a5 = up to 6 integer arguments (common OS practice) */
    li   a0, 1                  /* arg0 example */
    li   a1, 2                  /* arg1 example */
    li   a7, 64                 /* syscall number example (OS-defined) */
    ecall                       /* trap to kernel */
    /* return: a0 usually holds return value or negative error
     ↪  (OS-defined) */
    ret
```

Example 2: S-mode uses SBI via ECALL to request machine services

On many RISC-V systems, the kernel in S-mode cannot directly access certain platform features and instead calls the **Supervisor Binary Interface (SBI)** by executing `ecall` with a defined register convention (extension ID, function ID, arguments, and return pair).

```
/* Supervisor-mode: request timer programming via SBI (conceptual
 ↪  example).
   Convention: a7 = extension ID, a6 = function ID, a0..a5 = args,
    ↪  returns (a0=err, a1=value). */
```

```
.text
.global sbi_call_example
sbi_call_example:
    li   a7, 0x54494D45     /* EID example: "TIME" (platform-defined
     ↪  IDs exist; treat as interface contract) */
    li   a6, 0              /* FID example */
    li   a0, 0              /* arg0 */
    li   a1, 0              /* arg1 */
    ecall                   /* trap into M-mode firmware (SEE) */
    /* a0 = error code, a1 = return value */
    ret
```

Example 3: Why trap correctness is architectural correctness

A minimal trap handler must: (1) save caller context, (2) inspect cause, (3) dispatch, (4) restore context, (5) return with the correct `*ret` instruction.

```
/* Extremely simplified sketch: do not use as-is in production.
   Real handlers must follow ABI, stack alignment, and per-OS
    ↪  conventions. */
.text
.global supervisor_trap_entry
supervisor_trap_entry:
    addi sp, sp, -128           /* reserve trap frame (example size)
     ↪  */
    sd   ra, 0(sp)
    sd   t0, 8(sp)
    sd   t1, 16(sp)
    sd   t2, 24(sp)
    sd   a0, 32(sp)
    sd   a1, 40(sp)
    sd   a2, 48(sp)
    sd   a3, 56(sp)
    sd   a4, 64(sp)
    sd   a5, 72(sp)
    sd   a6, 80(sp)
    sd   a7, 88(sp)

    /* Read trap cause and EPC (supervisor view) */
    csrr t0, scause
    csrr t1, sepc

    /* Dispatch is OS-defined (exception/interrupt decoding) */
    /* ... */

    /* Restore and return */
    ld   ra, 0(sp)
    ld   t0, 8(sp)
    ld   t1, 16(sp)
    ld   t2, 24(sp)
    ld   a0, 32(sp)
    ld   a1, 40(sp)
    ld   a2, 48(sp)
    ld   a3, 56(sp)
    ld   a4, 64(sp)
    ld   a5, 72(sp)
    ld   a6, 80(sp)
    ld   a7, 88(sp)
    addi sp, sp, 128
    sret
```

The purpose of this booklet is to make these flows **predictable**: you will know what the CPU does automatically, what software must do, and how to design handlers and syscall paths that remain correct under interrupts, nested traps, and real toolchains.

Target Audience and Scope

This booklet is written for programmers who want a rigorous and practical understanding of RISC-V privilege and trap mechanics in real systems.

Who should read this

- Systems programmers building kernels, runtimes, monitors, hypervisors, or firmware components.
- Low-level developers writing debuggers, emulators, or ISA tooling that must model traps correctly.
- Application developers who want to understand what actually happens at the syscall boundary.
- C/C++ and Rust programmers moving from "language-level" thinking to "CPU contract" thinking.

Prerequisites

- Comfort reading RV64 assembly and basic instruction semantics.
- Basic familiarity with stacks, calling conventions, and register save/restore discipline.

- Basic OS concepts: user/kernel boundary, exceptions vs interrupts, and syscall motivation.

Scope boundaries

This booklet is intentionally focused:

- Privilege modes (U/S/M) and the **precise** control-flow rules between them.
- Trap causes, vectoring, entry/exit CSRs, and safe handler structure.
- Syscall principles: `ecall` semantics and environment-defined ABI contracts (OS ABI, SBI).

What This Booklet Covers (and What It Does Not)

Covered

- **Privilege model:** what U/S/M mean architecturally, and what each is allowed to do.
- **Trap mechanics:** how exceptions/interrupts transfer control; what CSRs are written; how EPC is chosen.
- **Delegation concept:** why some traps can be routed to S-mode vs handled in M-mode.
- **Trap vectors:** direct vs vectored mode, and the handler layout implications.
- **Return semantics:** `mret` and `sret`, and restoring privilege/interrupt state.
- **Syscalls:** `ecall` as a controlled boundary; typical register-based syscall patterns; the difference between OS syscalls and SBI calls.

Not covered (by design)

- Full virtual memory details (Sv39/Sv48/Sv57 page table formats, PBMT, invalidation extensions) beyond what is needed to understand traps.
- Hypervisor mode details (HS/VS, virtualization trap routing) except brief conceptual notes where relevant.
- Complete OS-specific syscall tables or numbers (these are environment-defined and change per OS).
- Device driver programming, PLIC/CLINT specifics, and platform integration details beyond the trap interface level.

Key discipline you will practice throughout this booklet

- Treat **syscalls and SBI calls as ABIs**: stable register contracts, defined clobbers, defined returns.
- Treat **trap handlers as critical code**: correct save/restore, correct stack alignment, re-entrancy awareness.
- Separate **architectural guarantees** from **environment policy**: the CPU defines trap entry/exit rules; the OS/firmware defines syscall meaning and error conventions.

Chapter 1

RISC-V Privilege Model Overview

1.1 Privilege Levels vs ISA Levels

RISC-V deliberately separates two ideas that are often mixed in CPU discussions:

- **ISA level (Instruction Set Architecture):** the programmer-visible instructions, registers, encodings, and base execution semantics (for example, RV64I as the integer base ISA, plus optional extensions such as M/A/F/D/C).
- **Privilege level (Execution privilege):** the *authority* of the currently running context (User / Supervisor / Machine), controlling which instructions, CSRs, and resources are legal to access.

What this separation gives you

- You can discuss "what an instruction does" (ISA semantics) without assuming a specific OS, firmware, or security policy.

- You can discuss “who is allowed to do what” (privilege semantics) without changing the base instruction behavior.
- The same RV64I instruction stream can run in multiple modes, but **with different allowed side effects**.

Practical model: one ISA, multiple authorities

Think of privilege as a hardware-enforced permission system over:

- **Privileged instructions** (e.g., returns from traps, memory-management controls).
- **CSRs** (Control and Status Registers), many of which are readable/writable only at specific privilege levels.
- **Resource ownership** (interrupt control, trap routing, timer programming, page table control, etc.).

Example 1: Same instruction, different legality depending on privilege

Some instructions are defined by the ISA but are **illegal** in lower privilege (raising an illegal-instruction exception). A key example is returning from a trap:

```
/* Conceptual: mret/sret are privileged returns. Executing them in
 ↪  the wrong mode traps. */
.text
.global bad_return_in_user
bad_return_in_user:
    /* If executed in U-mode: illegal instruction exception (cannot
     ↪  "return from trap" in U-mode). */
    mret
```

The ISA defines the encoding and architectural effect, but the privilege model defines whether the instruction is permitted.

Example 2: CSR access is privilege-checked by hardware

CSR instructions exist in the ISA, but each CSR has privilege rules. Attempting to access a CSR without sufficient privilege raises an exception.

```
/* Conceptual: reading supervisor/machine CSRs from U-mode is not
 ↪  permitted. */
.text
.global user_try_read_status
user_try_read_status:
    /* In U-mode, csrr on sstatus/mstatus typically causes an illegal
     ↪  instruction exception. */
    csrr a0, sstatus
    ret
```

Example 3: ISA extensions vs privilege extensions

ISA extensions add **instructions and behavior** (e.g., compressed instructions with C, multiply/divide with M). Privilege-related features add **execution context and control** (e.g., S-mode availability, trap delegation, virtual memory support). They are orthogonal: you can have RV64IMAC with or without S-mode support, depending on the platform.

Mental checkpoint:

- ISA answers: *What does this instruction mean?*
- Privilege answers: *Am I allowed to execute it or access this CSR/resource right now?*

1.2 Why RISC-V Uses Explicit Privilege Separation

RISC-V privilege is explicit by design: it minimizes “magic” behavior and pushes policy to software while keeping clear, enforceable hardware boundaries.

1) Security boundary you can reason about

User code must be untrusted by default. The CPU must guarantee that U-mode cannot:

- reprogram interrupt routing or masks,
- rewrite trap vectors,
- disable protections by touching control CSRs,
- access privileged memory mappings directly,
- escape into higher privilege without an approved trap path.

In RISC-V, this is achieved with:

- **mode-restricted access** to CSRs and privileged instructions,
- **trap-based control transfer** for sensitive operations (exceptions/interrupts),
- **explicit return instructions** (`sret`, `mret`) that restore privilege state only in legal contexts.

2) OS and firmware portability

By separating:

- **Machine mode** as the ultimate hardware owner, and

- **Supervisor mode** as the OS kernel mode,

RISC-V supports platforms where the kernel may not own all hardware details. This enables a clean "platform firmware" layer that exposes services to the OS via a defined interface (commonly via `ecall`).

3) Clear trap contract (precise control transfers)

A trap is not a branch and not a function call. It is a **hardware-defined** transfer that:

- records why control was transferred (`*cause`),
- records where to resume (`*epc`),
- records additional fault details when applicable (`*tval`),
- enters a handler at a privileged vector (`*tvec`).

Because this is explicit, you can build robust systems code with a predictable state machine:

- U-mode requests service → trap
- handler saves context → dispatch
- handler restores context → privileged return

Example 4: Explicit syscall boundary via ECALL

```
/* Conceptual: ECALL is the explicit boundary crossing instruction.
   The meaning (syscall number, arguments, error convention) is
    ↪  environment-defined. */
.text
.global u_syscall_example
```

```
u_syscall_example:
    li    a7, 1           /* syscall number (environment-defined) */
    li    a0, 123         /* arg0 */
    ecall                 /* trap into higher privilege handler */
    ret                   /* a0 typically returns result or error
     ↪  (environment-defined) */
```

4) Delegation is explicit, not accidental

A key idea in RISC-V is that not all traps must be handled at the highest privilege. The platform can **delegate** certain traps so that S-mode can handle them directly, while M-mode remains a minimal hardware supervisor.
This supports both styles:

- minimalist firmware + full-featured OS kernel,
- rich firmware + thinner OS kernel.

1.3 Comparison with x86 and ARM (Conceptual)

This section is conceptual: the goal is to map ideas, not to copy any one platform's details.

Privilege naming: U/S/M vs Rings vs Exception Levels

- **RISC-V:** U-mode (apps), S-mode (kernel), M-mode (firmware / machine control).
- **x86:** Rings (CPL 0–3), with OS typically using ring 0 for kernel and ring 3 for user.
- **ARM A-profile:** Exception Levels (EL0 user, EL1 kernel, EL2 hypervisor, EL3 secure monitor).

How syscalls cross privilege

All major architectures provide an **explicit trap instruction** for system calls:

- **RISC-V:** `ecall`
- **x86:** `syscall` / `sysenter` (modern fast paths), historically `int 0x80`
- **ARM:** `svc` (supervisor call), with `hvc`/`smc` for hypervisor/secure monitor calls

Conceptually, they all do the same:

- validate the boundary,
- switch to a privileged handler entry point,
- preserve a minimal return context,
- require software to save/restore the rest.

Trap vectoring: explicit tables and entry rules

- **RISC-V:** `mtvec`/`stvec` select base address and mode (direct/vectored).
- **x86:** IDT-based dispatch with gate descriptors; entry rules depend on the gate type and privilege checks.
- **ARM:** vector tables with fixed offsets; EL-specific vector base registers select table base.

The shared lesson for systems programmers:

- handlers must be extremely strict about context saving and return sequencing,
- the return instruction is privilege-sensitive and restores privileged state.

Privilege surface area philosophy

A useful way to compare philosophies:

- **RISC-V:** keep the architectural mechanism small and explicit; let the environment define policy and ABIs.
- **x86:** long historical evolution; rich legacy model; multiple syscall/trap mechanisms across generations.
- **ARM:** clean EL layering for OS/hypervisor/secure worlds; strong standardization around exception levels.

Example 5: One concept, three spellings

The same conceptual operation exists across architectures:

- **RISC-V:** “enter kernel” via `ecall`; return via `sret`.
- **x86:** “enter kernel” via `syscall`; return via `sysret` (or `iretq` depending on path).
- **ARM:** “enter kernel” via `svc`; return via exception return sequence (architecture-defined).

What changes is the **mechanical contract** (registers saved, vector selection, flags/state restoration), but the design target is identical: a safe, auditable boundary between untrusted code and privileged control.

Takeaway for this series: RISC-V privilege is easiest to master if you treat it as a **state machine**:

- current mode (U/S/M),

- allowed operations (instructions, CSRs, memory access),
- trap entry rules (what hardware writes, where control jumps),
- trap return rules (what `sret`/`mret` restores).

Once this model is precise, the rest of the booklet (traps, handlers, and syscalls) becomes a disciplined engineering exercise rather than trial and error.

Chapter 2

Execution Modes: U, S, and M

2.1 Definition of Each Mode

RISC-V defines privilege as a **hardware execution state** that controls what the currently running context is allowed to do. The privilege mode is not a software convention; it is enforced by the CPU and affects instruction legality, CSR accessibility, interrupt control, and trap routing.

User Mode (U-mode)

U-mode is the **least privileged** execution mode. It is intended for:

- application code,
- language runtimes,
- untrusted components (by default),
- user-space libraries.

U-mode must be unable to directly control the machine. Its only path to privileged services is via **traps** (exceptions/interrupts), most importantly **syscalls** using `ecall`.

Supervisor Mode (S-mode)

S-mode is typically used for the **operating system kernel**. It is intended for:

- process/thread scheduling,
- virtual memory management (page tables, address spaces),
- system call handling,
- device-driver frameworks (depending on platform design).

S-mode still may not own *all* machine resources. On many platforms, some facilities remain under M-mode control, accessed indirectly through a machine-level interface (commonly via an `ecall`-based service layer).

Machine Mode (M-mode)

M-mode is the **highest privilege** mode and is always present in standard RISC-V privileged implementations. It is intended for:

- platform firmware (boot code, low-level init),
- machine resource control (interrupt routing, timers, traps at machine level),
- defining what can be delegated to S-mode.

M-mode is the final authority: it can configure trap entry, control interrupt enabling, and define which events may be handled directly by S-mode via delegation.

Core rule: privilege mode selects authority, not "what ISA you are running". The same ISA instruction stream may run in any mode, but privileged operations are restricted.

2.2 Allowed Instructions per Mode

RISC-V does not define “separate ISAs” for each mode. Instead:

- Most base instructions are executable in all modes.
- Certain instructions and CSR accesses are **privileged** and therefore require S-mode or M-mode.
- Illegal privilege usage triggers a trap (typically illegal-instruction exception).

Common instruction categories across modes

(1) Unprivileged integer/branch/load/store

In RV64I, arithmetic, branches, loads, and stores are generally usable in U/S/M (subject to memory protection).

```
/* These are generally legal in U/S/M. Actual success depends on
 ↪  memory permissions. */
.text
.global common_ops
common_ops:
    addi a0, a0, 1
    add  a1, a1, a2
    beq  a0, x0, done
    ld   t0, 0(a3)
    sd   t0, 8(a3)
done:
    ret
```

(2) Privileged returns: mret and sret

Return-from-trap instructions are privileged because they restore privilege state and interrupt enable state.

```
/* Conceptual: sret is legal only when executing with supervisor
 ↪  privilege
   and when trap-return conditions are satisfied. */
.text
.global supervisor_return
supervisor_return:
    sret
```

(3) CSR access: legality depends on CSR and current privilege

CSR access instructions exist in the ISA, but each CSR has privilege restrictions. Attempting to access restricted CSRs traps.

```
/* Conceptual: access to privileged CSRs from insufficient privilege
 ↪  traps. */
.text
.global csr_access_examples
csr_access_examples:
    csrr a0, cycle        /* typically readable in U-mode if counter
     ↪  access is allowed by platform */
    csrr a1, sstatus      /* typically requires supervisor privilege
     ↪  */
    csrr a2, mstatus      /* requires machine privilege */
    ret
```

Practical view: what is "allowed" is two-dimensional

Instruction success depends on **both**:

- **Privilege legality:** is the instruction/CSR permitted in this mode?
- **Memory/IO legality:** even if the instruction is legal, memory access can fault due to protections.

Example: A legal load that still traps

A load is legal in U-mode, but can still fault if the address is not mapped/allowed.

```
/* Legal instruction, but may trap with a page fault / access fault
 ↪  depending on mapping/permissions. */
.text
.global u_load_may_fault
u_load_may_fault:
    ld   a0, 0(a1)        /* a1 points to some address; access may
     ↪  fault */
    ret
```

2.3 Hardware-Enforced Boundaries

RISC-V enforces boundaries through a combination of:

- **privilege checks on instructions and CSR accesses**,
- **memory protection and translation** (where implemented/enabled),
- **trap routing and privileged return rules**.

1) Privilege checks: illegal instruction and illegal CSR access

If code in U-mode attempts to:

- execute privileged control instructions (e.g., `mret`, `sret`),
- write trap vector CSRs (`mtvec`, `stvec`) without privilege,
- disable/enable interrupts via restricted CSRs,

the CPU does not “silently ignore” it. It traps.

```
/* U-mode cannot just "become" the kernel by touching privileged
 ↪  CSRs. */
.text
.global u_try_priv_escalation
u_try_priv_escalation:
    csrw stvec, a0     /* attempt to rewrite supervisor trap vector:
     ↪  traps in U-mode */
    sret               /* attempt to return-as-supervisor: traps in
     ↪  U-mode */
    ret
```

2) Trap boundaries: the only sanctioned privilege entry is a trap

Privilege transitions do not occur by ordinary jumps. Entering S-mode or M-mode happens by:

- exceptions (including `ecall`),
- interrupts,
- returns from traps (`sret`/`mret`) executed by the privileged handler.

Example: U-mode requests service via ECALL (conceptual syscall path)

```
/* User-mode: use ecall to request privileged service.
   ABI details (syscall numbers, args, error conventions) are
    ↪  environment-defined. */
.text
.global u_request_service
u_request_service:
    li   a7, 10          /* service number (example) */
    li   a0, 42          /* arg0 */
    ecall                /* trap into handler */
    ret                  /* return value typically in a0 */
```

3) Privileged return is controlled restoration of state

Returning from a trap restores privileged state fields (e.g., previous privilege and interrupt-enable bits) from status CSRs. This is why `sret`/`mret` are restricted and why handler correctness matters.

Example: Supervisor trap handler must preserve user context

```
/* Minimal conceptual skeleton: save enough state, handle, restore,
 ↪  then sret.
   Real kernels must follow ABI, alignment, and possibly per-thread
    ↪  trap-frame layout. */
.text
.global s_trap_entry_skeleton
s_trap_entry_skeleton:
    addi sp, sp, -96
    sd   ra, 0(sp)
    sd   a0, 8(sp)
    sd   a1, 16(sp)
    sd   a2, 24(sp)
    sd   a3, 32(sp)
    sd   a4, 40(sp)
    sd   a5, 48(sp)
    sd   a6, 56(sp)
    sd   a7, 64(sp)

    csrr t0, scause
    csrr t1, sepc
    /* dispatch based on cause, e.g., syscall vs fault vs interrupt
     ↪  */
    /* ... */

    ld   ra, 0(sp)
    ld   a0, 8(sp)
    ld   a1, 16(sp)
    ld   a2, 24(sp)
    ld   a3, 32(sp)
    ld   a4, 40(sp)
    ld   a5, 48(sp)
    ld   a6, 56(sp)
    ld   a7, 64(sp)
    addi sp, sp, 96
    sret
```

4) Memory protection: mode does not override permissions

Even in S-mode, memory access can trap if mappings and permissions do not allow it. Privilege increases authority, but does not eliminate correctness rules.

5) Delegation: who handles a trap is policy configured by higher privilege

A common platform design is:

- M-mode firmware handles certain machine-level responsibilities.
- Many traps are delegated so S-mode can handle them directly (e.g., most user faults and syscalls).

Key takeaway

- U-mode cannot execute privileged control instructions or touch privileged CSRs.
- The only valid entry into higher privilege is through traps.
- Trap handlers must be correct and disciplined because privileged returns restore security-critical state.
- “Allowed” is not just privilege: memory and environment policy still apply.

Chapter 3

Machine Mode (M-Mode) in Practice

3.1 Reset and Boot Flow

Machine mode (M-mode) is the architectural **root of control** on RISC-V systems. After reset, each hart begins execution in M-mode at a platform-defined reset vector. The early boot sequence is therefore a firmware responsibility: it must bring the CPU and platform into a state where either an OS kernel (typically in S-mode) or a stand-alone runtime (possibly in M-mode) can execute safely.

Reset state: what matters immediately

At reset, the firmware must assume:

- privilege is M-mode (highest authority),
- stack is not set up,
- trap vectors may be undefined until configured,

- memory map, device clocks, and DRAM controllers may not be initialized,
- multiple harts may start or may be released later (platform policy).

Typical boot phases (conceptual)

1. **Minimal CPU bring-up:** establish a safe stack, disable/guard interrupts, set trap vector.
2. **Platform init:** clocks, DRAM, UART, interrupt controller, timers (platform-specific).
3. **Runtime environment:** decide whether to run a payload in M-mode, or launch an OS in S-mode.
4. **Handoff:** configure delegation, expose services via SBI, then enter the next stage.

Example 1: Minimal M-mode reset stub (conceptual)

This example shows the canonical structure: set `mtvec`, initialize a stack, then jump into firmware main. It is intentionally minimal and platform-neutral.

```
/* Minimal M-mode reset stub (conceptual).
   Platform-specific reset vector placement is handled by
    ↪  linker/script/ROM. */
.text
.globl _start
_start:
    /* Point mtvec to a machine trap handler early */
    la    t0, m_trap_entry
    csrw  mtvec, t0

    /* Establish a temporary stack (symbol provided by linker script)
     ↪  */
    la    sp, _m_stack_top

    /* Optionally mask machine interrupts until ready */
    csrw  mie, x0          /* disable machine interrupt enables */
    csrci mstatus, 0x8     /* clear MIE bit (bit position depends on
    ↪  mstatus encoding; conceptual) */

    /* Jump to firmware main */
    tail  m_firmware_main

/* Placeholder: machine trap entry */
m_trap_entry:
    /* In real firmware: save context, inspect mcause,
    ↪  service/dispatch, restore, mret */
    j     m_trap_entry     /* trap-loop placeholder */
```

Boot decision: two common environments

- **Bare-metal:** firmware remains the “OS” and runs application code directly (often still M-mode).
- **OS-based:** firmware provides SBI services and boots an S-mode kernel (common on Linux-class systems).

3.2 Firmware Responsibilities

M-mode firmware is responsible for establishing a correct platform execution environment. The exact list varies by device class, but the responsibilities are stable in principle.

Core responsibilities

- **CPU bring-up:** set trap vectors, stacks, safe interrupt state, per-hart init.
- **Memory and platform init:** DRAM controller, device discovery/config, early console/UART.
- **Interrupt/timer setup:** configure machine-level interrupt routing and timer sources.
- **Trap policy:** decide which traps/interrupts remain at M-level and which are delegated to S-level.
- **SBI services:** provide a stable interface for S-mode software to request machine operations.
- **Boot handoff:** enter the kernel entry point with defined register conventions and state.

Multi-hart responsibilities (common pattern)

- one hart becomes the boot hart (initializes shared platform state),
- other harts may wait in a low-power loop until released,
- firmware must provide a safe method for inter-hart startup and coordination.

Example 2: Releasing secondary harts (conceptual pattern)

```
/* Conceptual multi-hart pattern:
   - hart0 initializes shared resources
   - other harts wait for a "go" flag, then jump to a common entry */
.text
.globl m_firmware_main
```

```
m_firmware_main:
    csrr  a0, mhartid           /* a0 = hart id */

    bnez  a0, secondary_wait    /* hart0 continues, others wait */

    /* hart0 init: platform init, SBI init, prepare kernel handoff */
    call  platform_init
    call  sbi_init
    call  prepare_handoff

    /* signal secondaries (memory-mapped or RAM flag) */
    la    t0, secondary_go
    li    t1, 1
    sd    t1, 0(t0)

    /* continue to handoff */
    call  enter_supervisor

secondary_wait:
    la    t0, secondary_go
1:
    ld    t1, 0(t0)
    beqz  t1, 1b

    /* secondary hart per-hart init, then join */
    call  per_hart_init
    call  enter_supervisor
    j     .
```

```
.data
.align 3
secondary_go:
    .dword 0
```

3.3 Full Hardware and CSR Control

M-mode has access to the complete set of machine resources and machine-level CSRs. This includes:

- **machine trap setup:** `mtvec`, `mepc`, `mcause`, `mtval`
- **machine status and interrupt control:** `mstatus`, `mie`, `mip`
- **delegation:** `medeleg`, `mideleg` (when S-mode is implemented)
- **machine protection features:** physical memory protection (PMP) configuration (platform feature)
- **platform interfaces:** timer/interrupt controller configuration (platform-specific)

Privilege meaning in practice

In M-mode you can:

- define where traps enter (via `mtvec`),
- choose which traps S-mode can receive (via delegation),
- restrict physical memory regions even for S/U (via PMP, when used),
- configure interrupt enable/disable and routing at the machine level.

Example 3: Configure delegation for common OS-style trap handling

A typical OS wants S-mode to handle most user-space exceptions and interrupts, while M-mode retains platform-only traps. Firmware can delegate selected events using `medeleg`/`mideleg`.

```
/* Conceptual delegation example:
   delegate "most" synchronous exceptions and supervisor-level
    ↪  interrupts to S-mode.
   Exact bit positions are defined by the privileged spec; treat
    ↪  masks as illustrative. */
.text
.globl configure_delegation
configure_delegation:
    /* Example: delegate selected exceptions */
    li    t0, -1
    csrw  medeleg, t0

    /* Example: delegate selected interrupts */
    li    t1, -1
    csrw  mideleg, t1

    ret
```

Example 4: Establish supervisor trap vector before entering S-mode

If S-mode is used, firmware typically sets S-mode trap vector (`stvec`) before handoff, or ensures the kernel sets it very early. M-mode can write supervisor CSRs.

```
/* Conceptual: firmware writes stvec to point to the kernel's early
 ↪  trap entry,
   then prepares mstatus/mepc for an M->S transition via mret. */
.text
.globl enter_supervisor
enter_supervisor:
    /* Point stvec to supervisor trap entry (kernel-provided address)
     ↪  */
    la    t0, s_trap_entry
    csrw  stvec, t0

    /* Set mepc to supervisor entry point */
    la    t1, s_kernel_entry
    csrw  mepc, t1

    /* Set MPP field in mstatus to Supervisor so mret enters S-mode.
       Exact bitfield manipulation is defined by privileged spec;
        ↪  shown conceptually. */
    csrr  t2, mstatus
    /* clear MPP then set to S (conceptual mask ops) */
    /* ... */
    csrw  mstatus, t2

    /* Return-from-trap: transitions into S-mode at mepc */
    mret

/* Placeholders for linkage */
s_trap_entry:
    j s_trap_entry
s_kernel_entry:
    j s_kernel_entry
```

Hardware-enforced boundary creation

M-mode does not just run “more code”; it defines the protection boundary:

- S/U cannot execute machine-only instructions or access machine-only CSRs.
- Delegation determines which privilege level receives which traps.
- PMP (when configured) can prevent S/U from accessing certain physical memory regions.

3.4 SBI Relationship Overview

The Supervisor Binary Interface (SBI) is the standard contract that allows S-mode software (typically an OS kernel) to request machine-level operations from M-mode firmware in a portable way.

Why SBI exists

Even with S-mode implemented, some actions remain fundamentally machine/platform-specific, such as:

- programming timers in a platform-defined way,
- sending inter-processor interrupts (IPIs),
- managing system reset/shutdown,
- platform-level performance/PMU operations (depending on implementation),
- advanced platform resource management.

SBI turns these into a stable call interface. The common mechanism is:

- S-mode prepares registers (extension ID, function ID, arguments),
- executes `ecall`,
- firmware in M-mode handles the request and returns results in registers.

Example 5: SBI-style call wrapper (conceptual register contract)

This wrapper illustrates the idea: inputs in argument registers and IDs, `ecall`, outputs returned.

```
/* Conceptual SBI call wrapper.
   Convention: a0..a5 = args, a6 = function id, a7 = extension id.
   Returns commonly (a0=error, a1=value). Exact details are
    ↪  SBI-defined. */
.text
.globl sbi_call
sbi_call:
    ecall
    ret
```

Example 6: SBI console output (conceptual)

Some environments expose console services via SBI. The OS can print early diagnostics without owning the UART driver.

```
/* Conceptual: putchar via SBI.
   IDs here are placeholders; do not treat as numeric truth. */
.text
.globl sbi_putchar_example
sbi_putchar_example:
    li    a7, 0x01       /* extension id (placeholder) */
    li    a6, 0x00       /* function id (placeholder) */
    mv    a0, a0         /* a0 = character */
    ecall
    ret
```

Key separation of responsibilities

- **S-mode kernel:** policies, scheduling, virtual memory, syscalls, process model.
- **M-mode firmware:** platform bring-up, machine resource control, SBI services, delegation policy.

Takeaway: M-mode is not “optional kernel mode”; it is the architectural root that boots the system, creates the boundary, and (on OS platforms) exposes a stable machine-service interface to the supervisor via SBI.

Chapter 4

Supervisor Mode (S-Mode)

4.1 Operating System Role

Supervisor mode (S-mode) is the privilege level intended for the **operating system kernel**. Its job is to implement the system-wide policies that untrusted user programs must not control, while still running efficiently and predictably.

What the OS kernel owns in S-mode

- **Process and thread management:** creation, scheduling, context switching.
- **System call interface:** controlled entry from U-mode via `ecall`.
- **Virtual memory:** page tables, address spaces, memory permissions.
- **Exception and interrupt handling:** user faults, timer interrupts, external interrupts (when delegated).
- **Resource arbitration:** file systems, IPC, device access policy.

What S-mode is not

S-mode is not automatically “master of the machine.” On many systems:

- machine-level interrupt routing, timers, and platform services remain under M-mode firmware control,
- the kernel uses an M-mode interface (commonly SBI) for platform operations,
- certain traps/interrupts may be handled in M-mode unless explicitly delegated.

Example 1: User-to-kernel boundary (syscall entry)

The OS kernel provides services to user programs by exposing a syscall ABI. User code enters the kernel using `ecall`.

```
/* User-mode syscall entry (conceptual).
   Register conventions and syscall IDs are OS-defined. */
.text
.global u_make_syscall
u_make_syscall:
    li   a7, 5          /* syscall number (example) */
    li   a0, 100        /* arg0 */
    li   a1, 200        /* arg1 */
    ecall               /* trap into S-mode handler (typical OS
     ↪  design) */
    ret
```

4.2 Kernel Execution Model

A correct kernel model in S-mode is built around a small set of invariants:

Invariants a kernel must enforce

- **All privilege transitions are explicit**: U-mode cannot jump into the kernel; it must trap.
- **Trap entry is a controlled context**: handler begins with minimal CPU-saved state; software saves the rest.
- **Return to user must be intentional**: resume address and privilege state must be validated and restored.
- **Preemption is real**: asynchronous interrupts can arrive; kernel must be re-entrancy-aware.

Kernel control paths

A kernel in S-mode runs along a few primary control paths:

- **Syscall path:** U-mode `ecall` → S-mode trap → syscall dispatch → `sret`
- **Fault path:** page fault / access fault → resolve or signal/terminate
- **Interrupt path:** timer tick / external interrupt → scheduler / device ISR → return

Example 2: Minimal supervisor trap entry skeleton

This skeleton shows the essential responsibilities: save context, read `scause`/`sepc`, dispatch, restore, `sret`.

```
/* Minimal S-mode trap entry skeleton (conceptual).
   Real kernels must follow ABI requirements, alignment, and
    ↪  per-thread trap frame layout. */
.text
```

```
.global s_trap_entry
s_trap_entry:
    addi sp, sp, -176

    /* Save caller state (subset shown; real kernels save all needed
     ↪  regs) */
    sd   ra, 0(sp)
    sd   gp, 8(sp)
    sd   tp, 16(sp)

    sd   t0, 24(sp)
    sd   t1, 32(sp)
    sd   t2, 40(sp)
    sd   t3, 48(sp)
    sd   t4, 56(sp)
    sd   t5, 64(sp)
    sd   t6, 72(sp)

    sd   a0, 80(sp)
    sd   a1, 88(sp)
    sd   a2, 96(sp)
    sd   a3, 104(sp)
    sd   a4, 112(sp)
    sd   a5, 120(sp)
    sd   a6, 128(sp)
    sd   a7, 136(sp)

    /* Read trap metadata */
    csrr t0, scause
    csrr t1, sepc
    csrr t2, stval

    /* Dispatch based on cause (interrupt bit + code) */
    /* ... syscall vs fault vs interrupt ... */

    /* Restore state */
    ld   ra, 0(sp)
    ld   gp, 8(sp)
    ld   tp, 16(sp)

    ld   t0, 24(sp)
    ld   t1, 32(sp)
    ld   t2, 40(sp)
    ld   t3, 48(sp)
    ld   t4, 56(sp)
    ld   t5, 64(sp)
    ld   t6, 72(sp)

    ld   a0, 80(sp)
    ld   a1, 88(sp)
    ld   a2, 96(sp)
    ld   a3, 104(sp)
    ld   a4, 112(sp)
    ld   a5, 120(sp)
    ld   a6, 128(sp)
    ld   a7, 136(sp)

    addi sp, sp, 176
    sret
```

Example 3: Syscall dispatch core idea

Syscall identification is typically carried in a designated register (commonly `a7` in many ABIs). The kernel reads it, validates it, then dispatches to an implementation.

```
/* Conceptual syscall dispatch snippet inside a trap handler.
   Treat as idea-level; real kernels use tables, bounds checks, and
    ↪  per-ABI rules. */
    /* On syscall entry: a7 = syscall number, a0..a5 = args */
    mv   t3, a7              /* syscall id */
    /* validate t3, then dispatch */
    /* ... */
```

4.3 Delegation from M-Mode

M-mode firmware is the root authority that decides which traps and interrupts S-mode can handle directly. This is done via delegation controls (conceptually: exception delegation and interrupt delegation).

Why delegation exists

- Keep M-mode firmware small and platform-focused.
- Allow the OS kernel to handle user-level faults and syscalls without entering M-mode.
- Improve performance by avoiding unnecessary privilege transitions.
- Keep platform-specific responsibilities (timers, reset, IPIs, etc.) in firmware when desired.

Delegation effect (conceptual)

When delegation is configured:

- a user exception (e.g., page fault, `ecall` from U-mode) traps directly into S-mode,
- the supervisor handler runs, then returns to user via `sret`,
- M-mode is not involved unless the event is not delegated or requires machine services.

Example 4: What delegation changes for a syscall

Without delegation: U-mode `ecall` → M-mode handler → firmware forwards/handles.
With delegation: U-mode `ecall` → S-mode handler → kernel handles syscalls directly.

```
/* Conceptual view only: the instruction is the same, but trap
 ↪  routing differs by delegation policy. */
.text
.global u_syscall_same_instruction
u_syscall_same_instruction:
    ecall      /* routed to S-mode or M-mode depending on delegation
     ↪  */
    ret
```

S-mode and SBI: delegation does not remove the need for firmware services

Even if most traps are delegated to S-mode, the kernel may still need M-mode services (platform operations). Those calls typically happen from S-mode via `ecall` into M-mode under the SBI contract.

4.4 Interaction with Virtual Memory

Virtual memory is where S-mode becomes a true OS kernel environment: it defines and enforces process-level isolation. The CPU uses supervisor configuration to translate virtual addresses and enforce permissions.

Supervisor responsibilities for memory

- **Create address spaces:** per-process page tables.
- **Define permissions:** read/write/execute, user/supervisor accessibility.
- **Handle page faults:** demand paging, copy-on-write, lazy mapping, stack growth.
- **Switch contexts:** switch active address space during process scheduling.

Key principle: loads/stores can trap even if the instruction is legal

Memory access legality depends on mapping and permissions, not just instruction legality.

```
/* U-mode load is legal instruction-wise, but may trap as a page
 ↪  fault if unmapped or forbidden. */
.text
.global u_read_may_page_fault
u_read_may_page_fault:
    ld   a0, 0(a1)      /* if a1 points to unmapped memory, a page
     ↪  fault occurs */
    ret
```

Virtual memory faults become traps to the kernel

Typical VM-related exceptions include:

- instruction fetch page fault,
- load page fault,
- store/AMO page fault,
- access faults when permissions or physical protections are violated.

In these cases:

- `scause` identifies the fault type,
- `sepc` holds the faulting instruction address,
- `stval` provides a faulting address or additional detail (when applicable),
- control enters the kernel trap handler at `stvec`.

Example 5: Fault-driven control flow is part of normal OS operation

A minimal conceptual flow:

1. U-mode touches a virtual address not mapped.
2. CPU raises a page-fault exception and traps to S-mode.
3. Kernel inspects `scause` and `stval`.
4. Kernel maps a page (or rejects access).
5. Kernel resumes U-mode at `sepc` (or signals failure).

```
/* Conceptual: inside S-mode trap handler, read fault metadata */
    csrr t0, scause     /* what fault? */
    csrr t1, sepc       /* where did it happen? */
    csrr t2, stval      /* which address (often) triggered it? */
    /* decide: map page / kill task / signal error */
```

Kernel execution model with VM

With virtual memory enabled, the kernel typically:

- runs in S-mode with privileged mappings available,
- exposes user pages with user permissions,
- keeps kernel pages protected from user access,
- uses trap handling to enforce and manage the memory model.

Takeaway: S-mode is where OS policy lives. Delegation determines which events reach S-mode directly. Virtual memory turns those events into a daily mechanism: faults are not “rare crashes” but part of the normal kernel control flow that enforces isolation and implements features such as demand paging and safe syscalls.

Chapter 5

User Mode (U-Mode)

5.1 Application Execution Rules

User mode (U-mode) is the least privileged execution mode and is the normal environment for:

- application programs,
- language runtimes,
- user-space libraries,
- untrusted or sandboxed components.

Rule 1: U-mode executes the unprivileged ISA, not “a different CPU”

U-mode runs the same base ISA (RV64I plus optional extensions) as other modes, but:

- privileged instructions are illegal,

- privileged CSRs are inaccessible,
- memory access is filtered by the current address space and permissions.

Rule 2: U-mode has no direct access to privileged services

Operations that affect global machine state (interrupts, trap vectors, page tables, timers) are not performed directly from U-mode. Instead, U-mode must request services through controlled entry points.

Rule 3: The only sanctioned privilege crossing is a trap

U-mode enters the kernel/firmware by causing a trap:

- **syscall:** `ecall` (intentional),
- **faults:** page faults, access faults, illegal instruction (unintentional),
- **interrupts:** asynchronous events that preempt execution (timer, external).

Example 1: U-mode requests service using ECALL (syscall boundary)

```
/* U-mode syscall request (conceptual).
   Convention depends on OS ABI: syscall id often in a7, args in
    ↪  a0..a5. */
.text
.global u_syscall_request
u_syscall_request:
    li   a7, 4          /* syscall number (example, OS-defined) */
    li   a0, 1          /* arg0 */
    li   a1, 2          /* arg1 */
    ecall                   /* trap to S-mode handler (typical) */
    ret                     /* a0 holds return value/error (OS-defined)
     ↪  */
```

Example 2: U-mode cannot “call” the kernel by jumping to its address

Even if U-mode knows a kernel address, it cannot safely or legally jump into kernel text:

- the address may not be mapped in the user address space,
- execution may trap due to permissions (no user execute),
- even if mapped incorrectly, it would violate the privilege boundary and security design.

5.2 Restricted Access Model

U-mode is restricted by **hardware privilege checks** and **memory protection**.

1) Instruction legality restrictions

U-mode cannot execute instructions that:

- return from privileged traps (`sret`, `mret`),
- directly manage privileged machine state,
- depend on privileged CSRs for correctness.

Example 3: Privileged return instruction is illegal in U-mode

```
/* If executed in U-mode: traps as illegal instruction. */
.text
```

```
.global u_illegal_return
u_illegal_return:
    sret
```

2) Memory access restrictions (the dominant restriction in practice)

Even legal loads/stores can trap if permissions do not allow it.

```
/* Legal instruction, but may trap as a page fault or access fault.
 ↪  */
.text
.global u_read_may_fault
u_read_may_fault:
    ld   a0, 0(a1)
    ret
```

3) I/O and device restrictions

User software typically cannot directly access device control registers or privileged memory regions. Access attempts trap (access fault) or are blocked by page table permissions.

4) Interrupt control restrictions

User software cannot control global interrupt enable/disable state or routing. This prevents user code from masking interrupts to evade scheduling or security enforcement.

5.3 Why U-Mode Cannot Access CSRs

Control and Status Registers (CSRs) include architectural control points such as:

- trap vectors (`stvec`, `mtvec`),

- exception PCs (`sepc`, `mepc`),
- cause registers (`scause`, `mcause`),
- status/interrupt control (`sstatus`, `mstatus`, `sie`, `mie`),
- address translation controls (supervisor-level translation configuration).

Allowing arbitrary U-mode CSR access would destroy the privilege model:

- user code could redirect trap entry to attacker-controlled code,
- user code could alter return state to escalate privilege,
- user code could disable enforcement mechanisms and timers,
- user code could learn privileged state that should not be observable.

CSR access is privilege-checked by hardware

CSR instructions exist in the ISA, but CSR addresses are partitioned by access rules. A CSR read/write from insufficient privilege traps (typically as illegal instruction).

Example 4: Attempt to read a supervisor CSR from U-mode

```
/* Conceptual: reading sstatus in U-mode is not permitted on standard
 ↪  OS configurations. */
.text
.global u_try_read_sstatus
u_try_read_sstatus:
    csrr a0, sstatus      /* traps in U-mode */
    ret
```

Example 5: Attempt to rewrite a trap vector CSR from U-mode

```
/* Conceptual: user cannot set supervisor trap entry. */
.text
.global u_try_rewrite_stvec
u_try_rewrite_stvec:
    la   a0, attacker_entry
    csrw stvec, a0        /* traps in U-mode */
    ret

attacker_entry:
    j attacker_entry
```

Controlled exposure: counters may be selectively visible

Some CSRs (notably performance and cycle counters) may be made accessible to U-mode by platform policy. This does not weaken control because these are observational and can be gated. However, **control CSRs** must remain protected.

5.4 Security Rationale

U-mode restrictions are not "inconveniences"; they are the minimum structure that makes multi-process systems safe.

Threat model baseline

Assume any user process may be:

- buggy (accidental memory corruption),
- malicious (attempting privilege escalation),

- compromised (executing attacker-controlled code).

Therefore, the CPU must guarantee:

- user code cannot overwrite kernel state directly,
- user code cannot rewrite security-critical control registers,
- user code cannot bypass auditing points (syscalls and faults),
- the kernel can preempt and manage user execution reliably.

Security boundary = controlled entry + controlled return

The secure pattern is:

- **Entry:** user requests a privileged action via a trap (`ecall`) or triggers a fault.
- **Verification:** kernel validates arguments, pointers, permissions, and policy.
- **Action:** kernel performs privileged work.
- **Return:** kernel returns to user via `sret` with controlled privilege state.

Example 6: Why argument validation exists (conceptual)

User pointers are untrusted. The syscall boundary is where the kernel must validate that a user-provided pointer:

- lies in user-mapped memory,
- has required permissions,
- does not alias kernel memory,

- does not violate policy.

```
/* Conceptual syscall argument scenario:
   a0 may hold a user pointer. Kernel must validate it before
    ↪  dereference. */
.text
.global u_pass_pointer
u_pass_pointer:
    /* a0 = pointer into user buffer (example) */
    /* a7 = syscall id */
    ecall
    ret
```

Example 7: Faults are a security feature

A page fault is not merely “an error”; it is the hardware enforcing the rule: **you may only access memory you are authorized to access**. The fault traps into the kernel, which then decides whether to:

- map memory (legitimate demand paging),
- deny access (security violation),
- terminate or signal the process (policy decision).

Takeaway: U-mode is designed to be powerful for computation but powerless for control. It can execute the unprivileged ISA, but must cross the privilege boundary through traps. CSR and privileged instruction restrictions are the hardware guarantees that make kernel security, process isolation, and reliable scheduling possible.

Chapter 6

Control and Status Registers (CSRs)

6.1 CSR Address Space and Privilege Encoding

Control and Status Registers (CSRs) are the architectural control plane of RISC-V. They are accessed using dedicated CSR instructions and are central to:

- privilege state control (status, interrupt enables),
- trap configuration (vectors, causes, fault metadata),
- time/counter exposure (cycle, instret, etc.),
- virtualization and memory-management control (when implemented),
- implementation-defined observation and tuning (when exposed through standard CSRs).

CSR address space (12-bit)

CSR addresses are encoded as a 12-bit immediate within CSR instructions. This has two practical consequences:

- the CSR number is part of the instruction encoding (fast and compact),
- permission checks are performed by hardware based on **current privilege** and the **CSR's class**.

CSR classes: privilege and access intent

RISC-V CSRs are organized so that privilege and access class are visible from the CSR number. Two major axes matter to programmers:

- **Minimum privilege required:** user, supervisor, or machine.
- **Access type:** read/write vs read-only vs special (e.g., counters exposed by policy).

In practice, you do not "probe" CSRs blindly. A correct low-level program:

- knows which CSRs exist in its execution environment,
- accesses only those legal for its privilege,
- treats optional CSRs as feature-dependent.

CSR access instructions

CSR access is performed by a small set of instructions that implement atomic read-modify-write behavior:

- `csrrw`: swap register with CSR (read old value, write new value)
- `csrrs`: set bits in CSR (read old, write old OR rs1)
- `csrrc`: clear bits in CSR (read old, write old AND r̃s1)
- immediate forms: `csrrwi`, `csrrsi`, `csrrci`

Example 1: Basic CSR reads and writes

```
/* CSR read: rd = CSR */
.text
.global csr_read_examples
csr_read_examples:
    csrr a0, sstatus     /* read supervisor status (requires S-mode)
     ↪  */
    csrr a1, sepc        /* read supervisor exception PC (requires
     ↪  S-mode) */
    ret
```

```
/* CSR write: CSR = rs */
.text
.global csr_write_examples
csr_write_examples:
    csrw stvec, a0       /* write supervisor trap vector (requires
     ↪  S-mode or higher) */
    csrw sie,   a1       /* write supervisor interrupt-enable bits
     ↪  (requires S-mode) */
    ret
```

Example 2: Atomic bit set/clear on a CSR

A common kernel task is to set/clear specific bits without disturbing others.

```
/* Atomically set bits in a CSR: old = CSR; CSR |= rs1 */
.text
.global csr_set_bits_example
csr_set_bits_example:
    li   t0, 0x2
    csrrs a0, sstatus, t0   /* a0 = old sstatus; set bit(s) in
     ↪  sstatus */
    ret
```

```
/* Atomically clear bits in a CSR: old = CSR; CSR &= ~rs1 */
.text
.global csr_clear_bits_example
csr_clear_bits_example:
    li   t0, 0x2
    csrrc a0, sstatus, t0   /* a0 = old sstatus; clear bit(s) in
     ↪  sstatus */
    ret
```

6.2 Read/Write Permissions

CSR permissions are enforced by hardware and depend on:

- **current privilege mode** (U/S/M),
- **CSR minimum privilege requirement**,
- **CSR access type** (read/write vs read-only),
- **platform policy** for selective exposure (notably counters).

Minimum privilege requirement

- **U-mode CSRs:** intended for unprivileged observation/control that does not compromise security.
- **S-mode CSRs:** kernel control plane (trap vectors, interrupt enables, virtual memory controls).

- **M-mode CSRs:** firmware root control (delegation, machine interrupts, machine trap vectors, PMP, etc.).

Read-only and write-ignored are not the same

Two distinct permission outcomes exist:

- **Read-only CSR:** writing it is illegal (traps) or is defined as no-op depending on CSR class.

- **WPRI/WLRL behavior:** some bits are defined as reserved; writing may be ignored or constrained.

A correct systems programmer follows one rule: **only set/clear documented bits; never assume reserved bits are writable or stable.**

Example 3: Safe bit manipulation pattern

Instead of writing an entire status CSR with a "magic constant", read-modify-write the specific bits.

```
/* Safe pattern: modify only targeted bits. */
.text
.global safe_status_update
safe_status_update:
    csrr t0, sstatus        /* read current */
    li   t1, 0x2
    or   t0, t0, t1         /* set a bit (illustrative) */
    csrw sstatus, t0        /* write back */
    ret
```

Counters and timers: policy-gated visibility

Cycle and instruction-retired counters may be readable in U-mode depending on platform configuration. This is a controlled exposure for profiling/measurement and does not grant privileged control.

```
/* Reading cycle counter may be allowed in U-mode if enabled by
 ↪  policy. */
.text
.global read_cycle_counter
read_cycle_counter:
    csrr a0, cycle
    ret
```

6.3 Illegal CSR Access Behavior

When CSR access violates privilege or permission rules, the CPU enforces the boundary by raising a trap. The most common outcome for illegal CSR access is an **illegal instruction exception**.

What makes CSR access illegal

- executing a CSR instruction in a mode lower than the CSR requires,
- writing a read-only CSR,
- accessing a CSR that is not implemented (environment-dependent),
- violating CSR-specific rules (e.g., writing invalid encodings to constrained fields).

Trap metadata produced

On illegal CSR access, trap metadata is recorded in the current trap CSRs:

- `*cause`: indicates illegal instruction,
- `*epc`: address of the faulting instruction,
- `*tval`: may record the faulting instruction or related detail (platform/privilege-defined).

Example 4: U-mode attempts to read a supervisor CSR

```
/* In U-mode, this typically traps as illegal instruction. */
.text
.global u_illegal_csr_read
u_illegal_csr_read:
    csrr a0, sstatus
    ret
```

Example 5: U-mode attempts to rewrite supervisor trap vector

```
/* User code cannot redirect kernel trap entry. This traps. */
.text
.global u_illegal_csr_write
u_illegal_csr_write:
    la   a0, fake_trap
    csrw stvec, a0
    ret

fake_trap:
    j fake_trap
```

Example 6: Supervisor attempts machine-only CSR access

Even S-mode cannot access machine-only CSRs unless the platform provides a different mechanism. Such access traps.

```
/* In S-mode, access to mstatus is typically illegal (machine-only
 ↪  CSR). */
.text
.global s_illegal_machine_csr
s_illegal_machine_csr:
    csrr a0, mstatus
    ret
```

Kernel strategy: treat illegal CSR access as a controlled fault

A robust kernel/firmware trap handler may:

- terminate the user process (for U-mode illegal CSR access),
- emulate or virtualize access (in specialized environments),
- report an error and refuse the operation.

Example 7: Trap handler reads scause/sepc on illegal instruction

```
/* Conceptual: inside S-mode trap handler, capture why we trapped. */
    csrr t0, scause      /* illegal instruction shows up here */
    csrr t1, sepc        /* faulting PC */
    csrr t2, stval       /* may hold the instruction bits or related
     ↪  detail */
```

Takeaway: CSRs are the control plane, and therefore they are tightly permissioned. Privilege encoding and access classes ensure that U-mode cannot rewrite control state, S-mode controls OS policy, and M-mode remains the root authority. Illegal CSR access is not undefined behavior; it is a deliberate trap that preserves the security boundary.

Chapter 7

Status and Trap CSRs

7.1 mstatus / sstatus

Purpose

`mstatus` and `sstatus` are the primary **status control** CSRs for machine and supervisor contexts. They capture and control:

- global interrupt enable state for the current privilege,
- the “previous interrupt enable” snapshot used by trap entry/return,
- the “previous privilege” used by privileged return (`mret`/`sret`),
- other execution-state fields that influence trap behavior and privilege transitions (implementation/extension dependent).

Conceptual fields you must understand

- **Interrupt enable (IE):** whether interrupts are globally enabled at that privilege.

- **Previous interrupt enable (PIE):** saved copy of IE on trap entry, restored on trap return.
- **Previous privilege (PP):** recorded privilege to return to.

The exact bit positions are defined by the privileged architecture, but the semantics are stable: trap entry saves state into “previous” fields and disables interrupts for the new handler context, then `*ret` restores state deterministically.

Example 1: Enable/disable supervisor interrupts (conceptual bitmask)

```
/* Conceptual: set/clear the supervisor interrupt-enable bit in
 ↪  sstatus.
   Bit masks are illustrative; real code should use the official bit
    ↪  definitions. */
.text
.global s_enable_interrupts
s_enable_interrupts:
    li    t0, 0x2
    csrrs x0, sstatus, t0   /* sstatus |= mask */
    ret

.global s_disable_interrupts
s_disable_interrupts:
    li    t0, 0x2
    csrrc x0, sstatus, t0   /* sstatus &= ~mask */
    ret
```

Example 2: Trap entry/return relies on saved previous-state

On a trap into S-mode, hardware updates:

- `sstatus.SPIE` ← `sstatus.SIE`
- `sstatus.SIE` ← 0
- `sstatus.SPP` ← previous privilege (U or S)

Then `sret` reverses the restore in a controlled way. This is why `sret` is privileged and why status must not be corrupted.

7.2 mtvec / stvec

Purpose

`mtvec` and `stvec` hold the trap-vector configuration for machine and supervisor privilege. They determine:

- the base address where trap handling begins,
- whether trap entry is **direct** or **vectored**.

Direct vs vectored (behavioral contract)

- **Direct mode:** all traps enter at the same base address.
- **Vectored mode:** interrupts enter at base + offset derived from interrupt cause; exceptions still enter at base.

Alignment requirement

Trap-vector base addresses must satisfy alignment rules. In practice, you place trap entry in a properly aligned code region and avoid "clever" low-bit usage unless you fully control the environment.

Example 3: Configure stvec to a trap entry (direct mode assumption)

```
/* Set stvec base to s_trap_entry (direct mode assumption). */
.text
.global s_set_trap_vector
s_set_trap_vector:
    la    t0, s_trap_entry
    csrw  stvec, t0
    ret

s_trap_entry:
    /* save context, handle, restore, sret */
    j s_trap_entry
```

Example 4: Vectored table layout idea (interrupts only)

In vectored mode, interrupts can jump to base + 4*cause (conceptual pattern). A common layout is:

- base handler reads `*cause` and dispatches, or
- a small jump table of stubs at fixed offsets.

```
/* Conceptual vectored layout: stubs at fixed offsets.
   The exact offset scheme is architectural; table illustrates the
    ↪  idea. */
.text
.align 6
s_vector_base:
    j s_common_trap         /* base entry */
    j s_irq_1               /* vector 1 */
    j s_irq_2                /* vector 2 */
    j s_irq_3                /* vector 3 */

s_common_trap:
    j s_common_trap

s_irq_1:
    j s_irq_1
s_irq_2:
    j s_irq_2
s_irq_3:
    j s_irq_3
```

7.3 mepc / sepc

Purpose

`mepc` and `sepc` hold the **exception program counter** for traps taken into M-mode or S-mode. They record the address of the instruction to resume (or the faulting instruction address, depending on trap type).

Core rules

- On trap entry, hardware writes `*epc`.
- On trap return (`mret`/`sret`), the CPU resumes at `*epc` (subject to architectural rules).
- Handlers often adjust `*epc` to skip an instruction (e.g., advance past `ecall`) if the handler completes it.

Example 5: Advance sepc to resume after ECALL (conceptual)

A syscall handler typically advances `sepc` so that returning to user does not re-execute `ecall`.

```
/* Conceptual syscall handling: sepc += 4 to skip the ecall
 ↪  instruction (assuming 32-bit encoding).
   If compressed instructions are in use, instruction length must be
    ↪  handled carefully. */
.text
.global s_advance_sepc_after_ecall
s_advance_sepc_after_ecall:
    csrr  t0, sepc
    addi  t0, t0, 4
    csrw  sepc, t0
    ret
```

Instruction-length note

If the C extension is enabled, instruction length may be 2 or 4 bytes. Robust kernels determine instruction length from the faulting instruction encoding (or enforce fixed-length entry stubs).

7.4 mcause / scause

Purpose

`mcause` and `scause` describe **why** a trap occurred. They encode:

- whether the trap was an **interrupt** or an **exception**,

- the **cause code** identifying the specific reason (timer interrupt, illegal instruction, page fault, etc.).

Interrupt vs exception

A standard decoding pattern is:

- check the high bit to distinguish interrupt vs exception,
- use the remaining bits as the cause code.

Example 6: Decode scause into (is_interrupt, code)

```
/* Decode scause:
   - MSB indicates interrupt (1) vs exception (0)
   - low bits contain the cause code */
.text
.global s_decode_scause
s_decode_scause:
    csrr  t0, scause

    /* is_interrupt = scause >> (XLEN-1) */
    srli  t1, t0, 63         /* for RV64 */

    /* code = scause & ((1<<(XLEN-1))-1) */
    li    t2, -1
    srli  t2, t2, 1          /* mask with low 63 bits set */
    and   t3, t0, t2         /* t3 = cause code */

    /* t1 = 0/1, t3 = code */
    ret
```

Example 7: Dispatch pattern skeleton

```
/* Conceptual dispatch skeleton: */
    csrr t0, scause
    srli t1, t0, 63         /* interrupt? */
    beqz t1, handle_exception
handle_interrupt:
    /* dispatch interrupt code */
    j done
handle_exception:
    /* dispatch exception code */
done:
```

7.5 mtval / stval

Purpose

`mtval` and `stval` provide **trap-specific additional information**. The most common uses are:

- the faulting virtual address for page faults and access faults,
- the faulting instruction bits for illegal-instruction traps (environment-dependent),
- other trap-specific values defined by the architecture for particular causes.

How to use it safely

- Always interpret `*tval` in conjunction with `*cause`.
- Do not assume `*tval` is meaningful for every trap.

- For page faults, `*tval` is the primary input to fault resolution (map/deny).

Example 8: Page fault handling reads stval as fault address

```
/* Conceptual: inside S-mode page fault handler */
    csrr t0, scause      /* type of page fault */
    csrr t1, sepc        /* where it happened */
    csrr t2, stval       /* faulting virtual address */
    /* resolve: validate address, map page, or terminate */
```

Example 9: Illegal instruction handling may consult stval

```
/* Conceptual: illegal instruction trap handling */
    csrr t0, scause
    csrr t1, sepc
    csrr t2, stval       /* may hold instruction bits or auxiliary
    ↪   detail depending on implementation */
    /* policy: kill task, emulate, or report fault */
```

Takeaway: these status and trap CSRs form the minimal state machine for privilege control:

- `*status` controls interrupt/return state,
- `*tvec` selects trap entry,
- `*epc` selects where execution resumes,
- `*cause` tells why the trap happened,
- `*tval` provides the key extra operand for fault diagnosis.

A correct kernel/firmware treats them as a contract: save/restore discipline, strict decoding, and controlled returns.

Chapter 8

Traps: Concept and Hardware Behavior

8.1 What Is a Trap

A **trap** is a hardware-defined control transfer that interrupts the normal sequential execution flow and transfers control to a privileged handler. Traps are the architectural foundation for:

- fault handling (illegal instructions, access faults, page faults),
- asynchronous event handling (interrupts),
- controlled privilege crossings (syscalls via `ecall`),
- enforcing isolation between user code and privileged control.

Trap vs branch vs function call

A trap is not a branch and not a function call:

- a branch is requested by the current instruction stream,

- a function call is a software convention that saves a return address and transfers control,
- a trap is **forced or requested** by architectural rules and switches into a handler **at higher authority**.

Two trap families

- **Exceptions (synchronous):** caused by the currently executing instruction (or its attempt to access memory).
- **Interrupts (asynchronous):** caused by external or timing events, independent of the current instruction.

Common trap sources in practice

- **Syscall request:** `ecall`
- **Illegal instruction:** executing privileged ops in U-mode, invalid encodings, forbidden CSR access
- **Memory faults:** load/store/access faults, instruction fetch faults, page faults
- **Interrupts:** timer ticks, external device interrupts, software interrupts (IPIs)

Example 1: ECALL intentionally triggers a trap

```
/* U-mode syscall request (conceptual).
   Environment defines syscall numbers and ABI conventions. */
.text
.global u_do_ecall
u_do_ecall:
    li   a7, 1          /* syscall number (example) */
    li    a0, 42           /* argument */
    ecall                  /* requested trap into privileged handler */
    ret
```

Example 2: A fault triggers a trap unintentionally

```
/* Legal instruction, but can trap if a1 points to unmapped or
 ↪  forbidden memory. */
.text
.global u_faulting_load
u_faulting_load:
    ld    a0, 0(a1)        /* may trap as page fault or access fault */
    ret
```

8.2 Precise Trap Guarantees

RISC-V traps are designed to be **precise** in the architectural sense: when a trap is taken, the architectural state is as if the faulting instruction did not complete, and the recorded exception PC (`*epc`) precisely identifies where to resume or diagnose.

What "precise" means for systems programmers

- If an instruction traps, its architectural effects are not partially committed.
- The handler can reliably inspect `*epc` to find the trapping instruction.
- The handler can resume execution by returning to `*epc` (or to `*epc` plus instruction length if appropriate).
- For memory faults, the fault address is provided through `*tval` when defined for the cause.

Why this matters

Precise traps are what make:

- demand paging correct (fault, map, retry),
- safe syscalls correct (trap, handle, resume),
- debugging and signals correct (precise fault attribution),
- emulation and virtualization feasible (decode and emulate the trapped instruction).

Example 3: Demand paging relies on precise retry

The OS can:

1. take a page fault on a load/store,
2. map the missing page,
3. return to the same instruction address,
4. re-execute the instruction successfully.

```
/* Faulting instruction (U-mode):
   On a page fault, kernel can map memory and resume at the same PC
    ↪  (sepc). */
.text
.global u_demand_page_example
u_demand_page_example:
    ld   t0, 0(a0)      /* may fault first time, succeed after kernel
     ↪  maps page */
    addi t0, t0, 1
    ret
```

Example 4: Syscall handling typically advances EPC

A syscall handler usually advances `sepc` so the `ecall` is not executed again after return.

```
/* Conceptual: advance sepc to skip the trapping ECALL (assumes
↪  4-byte ECALL encoding). */
.text
.global s_skip_ecall
s_skip_ecall:
    csrr t0, sepc
    addi t0, t0, 4
    csrw sepc, t0
    ret
```

Instruction length note (compressed)

If the C extension is enabled, the instruction at `*epc` may be 2 or 4 bytes. Robust handlers either:

- decode the trapped instruction length, or
- ensure syscall entry uses known-length instructions (controlled stubs), or
- enforce policy that avoids ambiguity where required.

8.3 Trap Entry Flow at the CPU Level

Trap entry is a hardware state transition with a defined sequence of architectural effects. While the exact details differ between M-mode and S-mode targets, the pattern is consistent.

1) Determine trap target privilege

The CPU decides whether the trap is handled in:

- M-mode (machine trap handler), or
- S-mode (supervisor trap handler),

based on the current mode, delegation configuration, and the type of trap.

2) Record trap metadata

On trap entry into a given privilege level, hardware writes:

- `*epc` with the trapping/resume PC,
- `*cause` with the interrupt/exception classification and cause code,
- `*tval` with additional fault information when defined.

3) Update status for controlled execution

Hardware updates status to create a safe handler context:

- record previous privilege (so return knows where to go),
- snapshot interrupt-enable into previous-interrupt-enable,
- disable interrupts for the handler by clearing current interrupt-enable (at that privilege).

4) Set PC to trap vector

The new PC is selected from:

- `mtvec` for machine-level traps,
- `stvec` for supervisor-level traps,

using direct or vectored mode rules.

Example 5: Minimal supervisor trap entry reads metadata

This snippet shows the first operations a real handler performs: read `scause`, `sepc`, `stval`.

```
/* Conceptual: first steps in an S-mode trap handler. */
.text
.global s_trap_entry_min
s_trap_entry_min:
    /* Save enough registers before using them (omitted for brevity)
     ↪  */

    csrr t0, scause      /* why did we trap? interrupt or exception +
     ↪  code */
    csrr t1, sepc        /* where did it happen? */
    csrr t2, stval       /* extra info (fault address/instruction
     ↪  detail when defined) */

    /* Dispatch:
       - if interrupt: handle timer/external/software
       - if exception: handle syscall/page fault/illegal
        ↪  instruction/etc. */
    j s_trap_entry_min
```

Example 6: Common interrupt-vs-exception decode pattern

```
/* Decode scause (RV64):
   MSB = 1 => interrupt, 0 => exception
   low 63 bits = cause code */
.text
.global s_scause_decode_pattern
s_scause_decode_pattern:
    csrr t0, scause
    srli t1, t0, 63       /* is_interrupt */
    li   t2, -1
    srli t2, t2, 1        /* mask low 63 bits */
    and  t3, t0, t2       /* cause code */
    ret
```

5) Software responsibilities start immediately

Hardware does **not** save general-purpose registers for you. That is the handler's job. A correct handler begins with:

- switching to a known-good stack (if required by design),
- saving caller registers to a trap frame,
- only then performing complex dispatch or calling C code.

Example 7: Trap entry must save state before calling other code

```
/* Conceptual: save minimal state, then call a higher-level handler
 ↪  (C/Rust). */
.text
```

```
.global s_trap_entry_callout
s_trap_entry_callout:
    addi sp, sp, -64
    sd   ra, 0(sp)
    sd   a0, 8(sp)
    sd   a1, 16(sp)
    sd   a2, 24(sp)

    csrr a0, scause     /* pass scause */
    csrr a1, sepc       /* pass sepc */
    csrr a2, stval      /* pass stval */
    call s_trap_dispatch_c

    ld   ra, 0(sp)
    ld   a0, 8(sp)
    ld   a1, 16(sp)
    ld   a2, 24(sp)
    addi sp, sp, 64
    sret

/* Placeholder symbol for linkage */
s_trap_dispatch_c:
    ret
```

Takeaway: traps are the CPU's controlled transfer mechanism. They are precise, record enough metadata to diagnose and resume, and create a handler context by updating status. From the first instruction of the trap handler onward, correctness is software's responsibility: save context, decode cause, handle, restore, then return with `sret`/`mret`.

Chapter 9

Exceptions

9.1 Illegal Instructions

Definition

An **illegal instruction exception** occurs when the CPU decodes an instruction that is not permitted to execute in the current environment. This includes:

- an invalid or reserved encoding,
- a valid encoding for an instruction that is **not implemented** (missing extension),
- a privileged instruction executed at insufficient privilege (e.g., `sret` in U-mode),
- an illegal CSR access (CSR instruction is valid, but the addressed CSR is not permitted).

Architectural effect (diagnostic contract)

On trap entry, the handler receives:

- ⋆epc: address of the faulting instruction,
- ⋆cause: illegal-instruction exception code,
- ⋆tval: often contains additional information (commonly the instruction bits or related detail when defined/implemented).

Why illegal instruction exists

It is a hard boundary that prevents:

- privilege escalation by executing privileged operations,
- accidental execution of instructions not supported by the hardware,
- user-space programs from probing or configuring privileged state through forbidden CSR accesses.

Example 1: Privileged return executed in U-mode

```
/* In U-mode, sret is illegal and traps (illegal instruction
 ↪  exception). */
.text
.global u_illegal_sret
u_illegal_sret:
    sret
```

Example 2: Illegal CSR access from U-mode

```
/* In U-mode, reading sstatus is typically illegal and traps. */
.text
.global u_illegal_csr
```

```
u_illegal_csr:
    csrr a0, sstatus
    ret
```

Example 3: Trap handler pattern for illegal instruction

A kernel typically terminates the process or delivers a signal/exception object. A firmware monitor may emulate the instruction if implementing a compatibility layer.

```
/* Conceptual: inside S-mode trap handler */
    csrr t0, scause    /* illegal instruction? */
    csrr t1, sepc      /* faulting PC */
    csrr t2, stval     /* may include instruction bits */
```

9.2 Environment Calls

Definition

Environment calls are explicit exceptions used to request services from a more privileged environment. In RISC-V, this is the `ecall` instruction. Its meaning depends on the current privilege:

- `ecall` from U-mode: enter the OS kernel (typical syscall path).
- `ecall` from S-mode: enter machine firmware services (commonly SBI on many platforms).
- `ecall` from M-mode: enter an even higher environment if one exists (rare; system-specific).

Key property: ECALL does not define the ABI

`ecall` defines the **trap mechanism**, not the syscall numbering, argument registers, or error convention. Those are defined by the **environment ABI** (OS syscall ABI, SBI, or monitor ABI).

Example 4: U-mode syscall request via ECALL (conceptual)

```
/* U-mode: syscall request (IDs/register usage are OS-defined). */
.text
.global u_ecall_syscall
u_ecall_syscall:
    li    a7, 10          /* syscall number (example) */
    li    a0, 1           /* arg0 */
    li    a1, 2           /* arg1 */
    ecall
    ret
```

Example 5: S-mode uses ECALL for SBI-style firmware service (conceptual)

```
/* S-mode: request machine service (conceptual SBI calling pattern).
   Typically: a7 = extension id, a6 = function id, a0..a5 = args. */
.text
.global s_ecall_sbi
s_ecall_sbi:
    li    a7, 0x53524954   /* placeholder EID */
    li    a6, 0            /* placeholder FID */
    li    a0, 123
    ecall
    ret                        /* a0/a1 commonly return error/value by
     ↪   convention */
```

Correct return behavior

The handler usually advances `*epc` so execution resumes after the `ecall`. Otherwise, returning to `*epc` would re-trigger the same environment call.

```
/* Conceptual: advance sepc to skip ECALL (assumes 4-byte ECALL). */
.text
.global s_advance_after_ecall
s_advance_after_ecall:
    csrr t0, sepc
    addi t0, t0, 4
    csrw sepc, t0
    ret
```

9.3 Instruction, Load, and Store Faults

What these faults mean

Faults in this group occur when an instruction fetch or data access violates a physical or access rule that is not (or not only) a virtual-memory translation problem.
Typical categories:

- **Instruction access fault:** fetch not permitted or fails at the physical/access level.
- **Load access fault:** read not permitted or fails at the physical/access level.
- **Store/AMO access fault:** write/atomic not permitted or fails at the physical/access level.

- **Instruction address misaligned / load/store misaligned:** if misalignment traps are enabled/implemented.

How to distinguish from page faults

- **Page faults** arise from virtual memory translation or permission checks in the page-table layer.

- **Access faults** arise when the access is blocked or fails beyond translation (e.g., physical protection, bus error).

In both cases, `*tval` commonly carries the faulting address when defined for that cause.

Example 6: Instruction fetch fault (conceptual)

```
/* Jumping to an unmapped or non-executable address can fault on
 ↪  instruction fetch. */
.text
.global u_bad_jump
u_bad_jump:
    li   t0, 0x0           /* placeholder invalid address */
    jr   t0                /* may trap as instruction fetch fault/page
     ↪  fault */
```

Example 7: Load/store access fault vs page fault (conceptual)

```
/* A legal load/store can trap due to permissions or physical access
 ↪  failure. */
.text
.global u_access_fault_examples
u_access_fault_examples:
    ld   a0, 0(a1)        /* may trap: load access fault or load page
     ↪  fault */
    sd   a2, 0(a1)        /* may trap: store access fault or store
     ↪  page fault */
    ret
```

Handler workflow

A supervisor trap handler typically:

- reads `scause` to classify fault type,
- reads `sepc` to locate the faulting instruction,
- reads `stval` to obtain the faulting address or detail,
- applies policy: kill process, signal, retry, or escalate.

```
/* Conceptual fault classification inside trap handler */
    csrr t0, scause
    csrr t1, sepc
    csrr t2, stval
```

9.4 Page Faults Overview

Definition

A **page fault** occurs when a virtual address access cannot be completed due to:

- translation failure (no valid mapping for the virtual page),
- permission failure at the page-table level (e.g., user access forbidden, write forbidden, execute forbidden),

- other translation/permission conditions defined by the address translation scheme (Sv39/Sv48/etc.).

Page faults are fundamental to operating systems because they enable:

- process isolation (each process has its own address space),
- demand paging (allocate/map pages when first used),
- copy-on-write (share pages until written),
- memory-mapped files and lazy loading.

Three common page-fault classes

- **Instruction page fault:** fetch from an unmapped/non-executable page.
- **Load page fault:** read from an unmapped/forbidden page.
- **Store/AMO page fault:** write/atomic to an unmapped/forbidden page.

Example 8: Demand paging retry pattern

The CPU traps on the first access. The kernel maps the page, then returns to retry the same instruction. This relies on precise `sepc` and meaningful `stval`.

```
/* User code: first touch triggers a page fault, then succeeds after
 ↪  kernel maps memory. */
.text
.global u_demand_paging_touch
u_demand_paging_touch:
    ld   t0, 0(a0)         /* first time may fault; after mapping,
     ↪  retry succeeds */
    addi t0, t0, 1
    ret
```

Example 9: Page fault handler reads fault address

```
/* Conceptual: S-mode page fault handling snippet */
    csrr t0, scause      /* indicates load/store/inst page fault */
    csrr t1, sepc        /* faulting instruction PC */
    csrr t2, stval       /* faulting virtual address */
    /* if valid access: map page and return to retry */
    /* else: terminate or signal */
```

Practical policy outcomes

When a page fault occurs, the kernel typically chooses one of:

- **Resolve and retry:** allocate/map, then return to `sepc`.
- **Reject:** deliver fault to process (signal/exception), terminate, or deny access.
- **Emulate:** specialized environments may emulate or virtualize access (less common for general OS).

Takeaway: exceptions are the synchronous enforcement mechanism of the privilege model. Illegal instructions protect control state, environment calls implement the syscall boundary, access faults enforce physical/protection constraints, and page faults implement the virtual-memory contract that enables isolation and modern OS memory features.

Chapter 10

Interrupts

10.1 Synchronous vs Asynchronous Events

RISC-V trap sources split into two fundamental categories:

Synchronous events (exceptions)

- Caused by the currently executing instruction or its memory access.
- Repeatable: re-executing the same instruction typically reproduces the trap until the cause is fixed.
- Examples: illegal instruction, page fault, load/store fault, `ecall`.

Asynchronous events (interrupts)

- Caused by events external to the current instruction stream.

- Not tied to a specific instruction intent; they arrive when hardware asserts an interrupt condition.
- Delivery is gated by interrupt-enable state and privilege rules.
- Examples: timer tick, external device interrupt, software interrupt (IPI).

Key property: interrupts are traps too

Interrupts use the same trap machinery:

- `*cause` marks “interrupt” and provides a cause code,
- `*epc` records where to resume,
- `*tvec` selects the entry point (direct/vectored),
- status CSRs snapshot and disable interrupts for the handler context,
- handler returns via `sret`/`mret`.

Example 1: Interrupt vs exception decode

```
/* Decode scause on RV64:
   MSB=1 => interrupt, MSB=0 => exception */
.text
.global s_is_interrupt
s_is_interrupt:
    csrr t0, scause
    srli t1, t0, 63      /* t1=1 => interrupt */
    ret
```

10.2 Machine vs Supervisor Interrupts

RISC-V distinguishes interrupts by **target privilege** and by **interrupt class**. Two supervisor-visible CSRs sets exist:

- **Supervisor-level:** `sie` (enable), `sip` (pending), `sstatus` (global SIE/SPIE).
- **Machine-level:** `mie` (enable), `mip` (pending), `mstatus` (global MIE/MPIE).

Where interrupts are handled

An interrupt can be handled in:

- **M-mode** (machine handler at `mtvec`), or
- **S-mode** (supervisor handler at `stvec`),

depending on platform configuration, delegation policy, and current execution context.

Delegation concept

Firmware (M-mode) can delegate certain interrupts to S-mode. When delegated, the kernel can receive and handle them directly without bouncing through M-mode.

Two-level gating: pending + enabled + global

An interrupt is taken only if all of these are true (conceptual):

- the corresponding pending bit is set (`mip`/`sip`),
- the corresponding enable bit is set (`mie`/`sie`),
- the global interrupt-enable for the privilege is set (`mstatus.MIE` or `sstatus.SIE`),
- privilege routing rules allow delivery to the current trap target.

Example 2: Enable a supervisor interrupt source (conceptual bitmask)

```
/* Conceptual: enable one supervisor interrupt source in sie.
   Bit masks are illustrative; real code uses official bit
    ↪  definitions. */
.text
.global s_enable_one_irq
s_enable_one_irq:
    li    t0, 0x20
    csrrs x0, sie, t0     /* sie |= mask */
    ret
```

Example 3: Global enable/disable of supervisor interrupts (conceptual)

```
/* Conceptual: toggle SIE in sstatus. */
.text
.global s_irq_on
s_irq_on:
    li    t0, 0x2
    csrrs x0, sstatus, t0
    ret

.global s_irq_off
s_irq_off:
    li    t0, 0x2
    csrrc x0, sstatus, t0
    ret
```

What the kernel must assume

- Interrupts can preempt kernel code unless masked.
- Trap entry does not save GPRs; the handler must save context immediately.
- Nested interrupts are possible if the handler re-enables interrupts.

10.3 Timer, Software, and External Interrupts

RISC-V commonly classifies interrupts into three functional groups at each privilege target:

- **Timer interrupts:** periodic or programmed time events (scheduling, timekeeping).
- **Software interrupts:** inter-processor interrupts (IPIs), rescheduling signals, cross-hart coordination.
- **External interrupts:** devices and interrupt controllers (I/O completion, network, storage, etc.).

Timer interrupts

Purpose: provide preemption and time slicing, drive scheduler ticks or high-resolution timers.

Kernel design consequences

- a timer interrupt is the standard mechanism to regain control from a running process,
- timer interrupts are performance-critical: handlers must be short and predictable,
- kernel often performs a minimal accounting step then schedules deferred work.

Software interrupts (IPIs)

Purpose: allow one hart to signal another hart. Common uses:

- reschedule another hart,
- request TLB shootdown,
- coordinate stop-the-world operations,
- wake a sleeping hart.

External interrupts

Purpose: deliver device events. External interrupts usually enter through a platform interrupt controller and are then dispatched by the kernel to the responsible driver.

Example 4: Read interrupt cause and dispatch (conceptual)

This example shows a common pattern: decode interrupt vs exception, then branch to interrupt handler path.

```
/* Conceptual trap dispatch fragment (RV64). */
.text
.global s_trap_dispatch_fragment
s_trap_dispatch_fragment:
    csrr t0, scause
    srli t1, t0, 63          /* interrupt? */
    beqz t1, handle_exception

handle_interrupt:
    /* code = low 63 bits */
    li   t2, -1
    srli t2, t2, 1
    and  t3, t0, t2

    /* dispatch based on t3: timer/software/external (codes are
     ↪  architectural) */
    /* ... */
    ret

handle_exception:
    /* syscall/page fault/illegal instruction/etc. */
    ret
```

Example 5: Minimal timer interrupt skeleton (conceptual)

A timer interrupt handler typically:

- acknowledges/clears the timer interrupt source (platform/firmware-dependent),
- updates timekeeping and scheduler state,
- decides whether to preempt the current task.

```
/* Conceptual S-mode timer interrupt skeleton: */
.text
.global s_timer_irq
s_timer_irq:
    /* save minimal context (omitted) */

    /* acknowledge timer source:
       - may require SBI call to program next timer on some platforms
       - or direct device register writes on others */
    /* ... */

    /* scheduler tick / accounting */
    /* ... */

    /* restore and return */
    sret
```

Example 6: Software interrupt (IPI) handling idea

```
/* Conceptual software interrupt handler:
   - clear pending software interrupt source (platform-specific)
   - perform requested cross-hart action */
.text
.global s_soft_irq
s_soft_irq:
    /* clear software interrupt pending */
    /* ... */
    /* perform IPI action: resched, shootdown, etc. */
    /* ... */
    sret
```

Example 7: External interrupt handling idea

```
/* Conceptual external interrupt handler:
   - query interrupt controller for IRQ id
   - dispatch to device driver */
.text
.global s_ext_irq
```

```
s_ext_irq:
    /* irq_id = read_interrupt_controller() */
    /* dispatch_driver(irq_id) */
    sret
```

Correctness checklist for interrupt handlers

- Save/restore all registers you clobber (or use a defined trap-frame convention).
- Preserve stack alignment and avoid calling conventions violations.
- Acknowledge/clear the interrupt source to avoid immediate re-entry storms.
- Keep the top-half handler minimal; defer heavy work when possible.
- Decide carefully whether/when to re-enable interrupts (nested interrupts policy).

Takeaway: interrupts are asynchronous traps. Their delivery is gated by pending bits, enable bits, and global status. M-mode vs S-mode handling depends on delegation and platform design. Timer, software, and external interrupts form the core set that drives scheduling, multi-hart coordination, and device I/O in real operating systems.

Chapter 11

Trap Handling and Return

11.1 Trap Vector Modes (Direct / Vectored)

Trap entry begins at an address chosen by the trap-vector CSRs:

- `mtvec` for traps handled in M-mode,
- `stvec` for traps handled in S-mode.

Each trap vector supports two conceptual modes:

Direct mode

- All traps (exceptions and interrupts) enter at the same base address.
- A single entry stub saves context, reads `*cause`, and dispatches.
- This is the simplest and most common design for educational kernels and firmware.

Vectored mode

- Exceptions still enter at the base address.
- Interrupts enter at base plus an implementation-defined per-cause offset (conceptual vectoring).
- This enables fast dispatch for high-frequency interrupts by using a table of stubs.

Example 1: Configure stvec for direct entry

```
/* Direct mode assumption: stvec points to a single entry stub. */
.text
.global s_set_stvec_direct
s_set_stvec_direct:
    la   t0, s_trap_entry
    csrw stvec, t0
    ret

s_trap_entry:
    j s_trap_entry
```

Example 2: Conceptual vectored table layout

In vectored mode, you typically place a small jump table at the vector base. Each entry jumps to a dedicated interrupt stub, while exceptions land at the base.

```
/* Conceptual vectored layout: not numeric-truth, but
↪   structural-truth. */
.text
.align 6
```

```
s_vector_base:
    j s_exception_entry     /* base: exceptions */
    j s_irq_stub_1          /* interrupt vector 1 */
    j s_irq_stub_2          /* interrupt vector 2 */
    j s_irq_stub_3          /* interrupt vector 3 */

s_exception_entry:
    j s_exception_entry

s_irq_stub_1:
    j s_irq_stub_1
s_irq_stub_2:
    j s_irq_stub_2
s_irq_stub_3:
    j s_irq_stub_3
```

Design rule

Even in vectored mode, most real kernels share a common core:

- minimal stub per vector,
- common prologue saves context,
- shared dispatch logic based on `*cause`,
- shared epilogue restores context and returns.

11.2 Software Context Saving

Hardware does not preserve your general-purpose registers on trap entry. The handler must save enough state to:

- resume the interrupted context exactly,
- call out to higher-level code (C/C++/Rust) safely,
- support nested traps if the design enables it.

What must be saved

At minimum, a robust trap frame typically saves:

- all caller-saved registers that the handler will clobber,
- often all integer registers for simplicity and correctness,
- special registers used by the ABI contract (`gp`, `tp`) as required by the runtime,
- trap metadata (cause, epc, tval) either in registers passed to a dispatcher or stored in the frame.

Two common trap-frame strategies

- **Full-save frame:** save almost all GPRs on every trap (simple, heavier).
- **Selective-save frame:** save only required regs, with strict rules about what the handler may clobber (fast, complex).

Example 3: Supervisor trap prologue/epilogue (full-save style, conceptual)

```
/* Conceptual full-save S-mode trap entry.
   Frame layout and saved set are OS-defined; this illustrates
    ↪  disciplined structure. */
.text
.global s_trap_entry_fullsave
s_trap_entry_fullsave:
    addi sp, sp, -256

    sd   ra,   0(sp)
    sd   sp,   8(sp)         /* optional: store original sp if using
     ↪  alternate stacks */
    sd   gp,  16(sp)
    sd   tp,  24(sp)

    sd   t0,  32(sp)
    sd   t1,  40(sp)
    sd   t2,  48(sp)
    sd   s0,  56(sp)
    sd   s1,  64(sp)
    sd   a0,  72(sp)
    sd   a1,  80(sp)
    sd   a2,  88(sp)
    sd   a3,  96(sp)
    sd   a4, 104(sp)
    sd   a5, 112(sp)
    sd   a6, 120(sp)
    sd   a7, 128(sp)
    sd   s2, 136(sp)
    sd   s3, 144(sp)
    sd   s4, 152(sp)
    sd   s5, 160(sp)
    sd   s6, 168(sp)
    sd   s7, 176(sp)
    sd   s8, 184(sp)
    sd   s9, 192(sp)
    sd   s10,200(sp)
    sd   s11,208(sp)
    sd   t3, 216(sp)
    sd   t4, 224(sp)
    sd   t5, 232(sp)
    sd   t6, 240(sp)

    /* Read trap metadata */
    csrr a0, scause
    csrr a1, sepc
    csrr a2, stval

    /* Call high-level dispatcher (optional) */
    call s_trap_dispatch

    /* Restore */
    ld   ra,   0(sp)
    ld   gp,  16(sp)
    ld   tp,  24(sp)

    ld   t0,  32(sp)
    ld   t1,  40(sp)
    ld   t2,  48(sp)
    ld   s0,  56(sp)
    ld   s1,  64(sp)
    ld   a0,  72(sp)
    ld   a1,  80(sp)
    ld   a2,  88(sp)
    ld   a3,  96(sp)
    ld   a4, 104(sp)
    ld   a5, 112(sp)
    ld   a6, 120(sp)
    ld   a7, 128(sp)
    ld   s2, 136(sp)
    ld   s3, 144(sp)
    ld   s4, 152(sp)
    ld   s5, 160(sp)
    ld   s6, 168(sp)
    ld   s7, 176(sp)
    ld   s8, 184(sp)
    ld   s9, 192(sp)
    ld   s10,200(sp)
    ld   s11,208(sp)
    ld   t3, 216(sp)
    ld   t4, 224(sp)
    ld   t5, 232(sp)
    ld   t6, 240(sp)

    addi sp, sp, 256
    sret
```

```
/* Placeholder for linkage */
s_trap_dispatch:
    ret
```

Critical rule

Do not call into C/C++/Rust before saving the registers that the calling convention assumes are preserved/clobbered. Your trap entry is not a normal function call; you must **create** a valid calling environment first.

11.3 mret and sret Semantics

`mret` and `sret` are privileged return instructions used to exit a trap handler and resume execution.

What they do conceptually

- set the next PC from `mepc` or `sepc`,
- restore privilege mode from a saved "previous privilege" field in status,
- restore interrupt-enable state from a saved "previous interrupt enable" field,
- complete the architectural trap-return transition atomically.

Why they are privileged

If untrusted code could execute `sret`/`mret`, it could:

- forge a privileged return path,
- restore privileged interrupt state incorrectly,

- attempt privilege escalation by manipulating saved fields.

Example 4: M-mode handoff to S-mode using mret (conceptual)

Firmware commonly sets `mepc` to the supervisor entry point and configures the "previous privilege" field so that `mret` transitions into S-mode.

```
/* Conceptual M->S handoff:
   - set mepc to kernel entry
   - set status so mret drops to S-mode */
.text
.global m_enter_supervisor
m_enter_supervisor:
    la   t0, s_kernel_entry
    csrw mepc, t0

    /* configure mstatus to return to supervisor (field ops omitted;
     ↪  conceptual) */
    /* ... */

    mret

s_kernel_entry:
    j s_kernel_entry
```

11.4 Restoring Privilege and Interrupt State

Trap return is not merely "jump back"; it restores a security-critical state machine.

State restored on return

On `sret` (conceptual):

- PC ← `sepc`
- privilege ← `sstatus.SPP` (U or S)
- interrupt-enable ← `sstatus.SPIE`
- `sstatus.SPIE` ← 1 (implementation-defined rule to re-arm for next trap cycle)

On `mret` (conceptual):

- PC ← `mepc`
- privilege ← `mstatus.MPP`
- interrupt-enable ← `mstatus.MPIE`
- `mstatus.MPIE` ← 1 (re-arm rule)

Why handlers must be disciplined

A trap handler that corrupts:

- `*epc` returns to the wrong address,
- `*status` fields returns to the wrong privilege or wrong interrupt-enable state,
- saved GPR context returns with corrupted computation state.

These are not “bugs like normal bugs”; they break isolation and can become security vulnerabilities.

Example 5: Safe syscall return requires advancing EPC and preserving state

A syscall handler typically:

- reads syscall number/args,
- writes return value in `a0` (and possibly `a1`),
- advances `sepc` past `ecall`,
- restores registers,
- executes `sret`.

```
/* Conceptual: adjust sepc in a syscall handler to avoid re-executing
 ↪  ECALL. */
.text
.global s_syscall_finish
s_syscall_finish:
    csrr t0, sepc
    addi t0, t0, 4
    csrw sepc, t0
    /* return value already placed in a0 */
    ret
```

Example 6: Nested interrupts policy (conceptual)

Many kernels keep interrupts disabled during early trap handling, then selectively re-enable them after saving context. This avoids re-entrancy before the trap frame exists.

```
/* Conceptual: after saving context, optionally re-enable interrupts
 ↪  for long handlers. */
.text
.global s_maybe_enable_nested
s_maybe_enable_nested:
    /* context is already saved */
    li    t0, 0x2
    csrrs x0, sstatus, t0   /* enable supervisor interrupts (mask is
     ↪  illustrative) */
    ret
```

Correctness checklist for trap return

- **Save first:** establish a trap frame before calling other code.
- **Decode cause:** handle interrupt vs exception correctly.
- **Maintain EPC:** advance for syscalls; keep for faults meant to retry.
- **Acknowledge sources:** clear interrupt sources before return to avoid storms.
- **Restore exactly:** registers and status fields must match the expected ABI and trap-frame layout.
- **Return with *ret:** use `sret` or `mret` as appropriate, never a normal `ret`.

Takeaway: trap handling is a disciplined pipeline: vector entry selects the handler, software saves context, the handler decides whether to resume, advance, or terminate, and `sret`/`mret` restore privilege and interrupt state as an architectural state machine.

Chapter 12

Syscalls in RISC-V

12.1 ECALL Instruction Semantics

What ECALL is

`ecall` is an unprivileged instruction whose defined architectural effect is to raise an **environment call exception**. It is the canonical mechanism for requesting a privileged service through a controlled trap.

What ECALL is not

`ecall` does **not** define:

- syscall numbers,
- argument registers,
- error conventions,
- which privilege level handles the call (S-mode vs M-mode),

- what services exist.

Those are defined by the **environment ABI** (OS syscall ABI, SBI for firmware services, or a monitor ABI).

Privilege-relative meaning

The same `ecall` instruction has different **target meaning** depending on the caller privilege:

- **U-mode `ecall`:** intended to enter the OS kernel (syscall) on most OS platforms.
- **S-mode `ecall`:** commonly used to request machine firmware services (SBI) on many platforms.
- **M-mode `ecall`:** request to an even higher environment if present (rare and system-specific).

Trap metadata produced

When `ecall` traps:

- `*epc` records the address of the `ecall`,
- `*cause` indicates an environment call exception (with a code that reflects the originating privilege),
- `*tval` is typically not used for syscall semantics.

Example 1: A raw `ecall` boundary from U-mode (conceptual)

```
/* User-mode: request a service. ABI decides how to interpret
 ↪  registers. */
.text
```

```
.global u_raw_ecall
u_raw_ecall:
    ecall
    ret
```

12.2 Syscall Flow: User → Kernel

A syscall is a controlled privilege crossing implemented as:

1. user code places syscall ID and arguments in registers (ABI-defined),
2. user executes `ecall`,
3. CPU traps to the privileged handler (typically S-mode in an OS),
4. handler saves context, identifies syscall, validates inputs,
5. kernel performs the service,
6. kernel places return values in registers,
7. kernel advances `sepc` past the `ecall`,
8. kernel returns to user via `sret`.

Example 2: User stub sets registers then traps

```
/* Conceptual syscall ABI:
   - a7 = syscall number
   - a0..a5 = args
   - return in a0 (and a1 if needed) */
.text
```

```
.global u_syscall_stub
u_syscall_stub:
    li   a7, 64          /* syscall id (example) */
    mv   a0, a0          /* arg0 */
    mv   a1, a1          /* arg1 */
    ecall
    ret
```

What makes syscalls safe

Syscalls are safe only if the kernel enforces these invariants:

- user cannot enter kernel except through trap entry,
- kernel validates all user-provided data (especially pointers),
- kernel returns only through `sret` with controlled privilege state,
- kernel keeps kernel memory unmapped or inaccessible to user mappings by policy.

12.3 ABI View of Syscalls

Syscall ABI is an OS contract

A syscall ABI specifies:

- which register holds the syscall number,
- which registers hold arguments,
- which registers are clobbered vs preserved across the call,
- return value and error signaling,

- how large values (64-bit, pointers) are passed and returned.

Three layers you must not confuse

- **RISC-V ISA:** defines `ecall` as an exception trigger.
- **RISC-V psABI:** defines the *function-call ABI* (register roles, stack rules) for normal calls.
- **OS syscall ABI:** defines the *syscall convention* (numbering and register usage for syscalls).

A kernel trap handler must obey both:

- architectural trap rules (CSRs, `sret`),
- and the chosen syscall ABI (so user stubs and libc work correctly).

Example 3: ABI-style wrapper that behaves like a function

```
/* A syscall wrapper can be used like a normal function from
 ↪  assembly.
   Caller places args in a0..a2, wrapper sets a7 and traps. */
.text
.global sys_write_like
sys_write_like:
    li   a7, 1            /* syscall id (example) */
    ecall
    ret                   /* returns result in a0 */
```

Return values and errors

Many syscall ABIs use:

- `a0` as primary return value,
- sometimes a separate error indicator or a convention where negative values indicate an error.

The specific rule is OS-defined; the kernel must implement that contract consistently.

12.4 Minimal Syscall Handler Design

A minimal syscall handler design is the smallest correct core that:

- saves user context into a trap frame,
- recognizes `ecall` cause,
- reads syscall number and arguments from the saved frame (or live registers),
- validates inputs,
- dispatches to a syscall table,
- writes return values,
- advances `sepc`,
- returns via `sret`.

Cause discrimination

Syscalls are handled only when:

- `scause` indicates an exception (not interrupt),
- cause code corresponds to environment call from U-mode (in an OS syscall design).

Example 4: Trap handler detects ECALL and dispatches (conceptual skeleton)

```
/* Minimal conceptual syscall path inside an S-mode trap handler.
   Real kernels must also handle interrupts and other exceptions. */
.text
.global s_trap_entry_syscall_only
s_trap_entry_syscall_only:
    /* Save minimal state needed (omitted here for brevity) */

    csrr t0, scause
    csrr t1, sepc

    /* If not an exception, or not an ECALL-from-U, branch elsewhere
     ↪  (omitted) */
    /* ... */

    /* Syscall number in a7 (by syscall ABI) */
    mv   t2, a7

    /* Dispatch: syscall_table[t2](a0..a5) conceptually */
    /* ... produce return value in a0 ... */
    /* Advance sepc to skip ECALL (assumes 4-byte ECALL encoding) */
    addi t1, t1, 4
    csrw sepc, t1

    /* Restore state (omitted) */
    sret
```

Pointer validation is mandatory

If a syscall takes a pointer, the kernel must treat it as untrusted:

- validate it lies in user address space,
- validate mapping and permissions for the requested access,
- copy data safely between user and kernel buffers.

Example 5: Syscall with pointer argument (conceptual)

```
/* User passes a pointer in a0. Kernel must validate before
 ↪  dereference. */
.text
.global u_syscall_with_ptr
u_syscall_with_ptr:
    la   a0, user_buffer      /* user pointer */
    li   a1, 64               /* length */
    li   a7, 2                /* syscall id (example) */
    ecall
    ret
```

```
.data
.align 3
user_buffer:
    .dword 0
```

EPC management rule

- **Syscall:** advance `sepc` to resume after `ecall`.
- **Fault to retry (e.g., demand paging):** keep `sepc` unchanged so the faulting instruction is retried.

Example 6: Shared trap handler must separate syscall vs retry-fault

```
/* Conceptual: syscall advances sepc; page fault typically does not.
↪  */
    csrr t0, scause
    /* if syscall: sepc += 4 */
    /* if page fault resolved: keep sepc */
```

Takeaway: `ecall` is the architectural trap mechanism for syscalls. The syscall ABI is an environment contract layered on top of it. A minimal correct handler is a disciplined pipeline: save context, verify cause, dispatch by ABI, validate inputs, advance `sepc` for syscalls, then return with `sret` restoring privilege and interrupt state.

Appendices

Appendix A — Trap Cause Codes Reference

This appendix provides a precise, implementation-independent reference for **trap cause encoding** in RISC-V systems. Trap causes are reported through `mcause` and `scause` and form the primary contract between hardware and trap-handling software.

Exception Codes

Exceptions are **synchronous** events caused by the currently executing instruction. In `*cause`, exceptions are indicated when the most significant bit is **0**, and the remaining bits encode the exception code.

Common architectural exception codes (conceptual reference)

- Instruction address misaligned
- Instruction access fault
- Illegal instruction
- Breakpoint
- Load address misaligned

- Load access fault
- Store/AMO address misaligned
- Store/AMO access fault
- Environment call from U-mode
- Environment call from S-mode
- Instruction page fault
- Load page fault
- Store/AMO page fault

Interpretation rules

- The numeric values are architecturally defined and stable across implementations.
- Not all codes must be implemented by all systems; unsupported causes will not be raised.
- Exception codes are mutually exclusive for a given trap.

Typical software reactions

- Illegal instruction: terminate process or emulate.
- Page fault: map memory and retry, or deny access.
- Environment call: dispatch syscall or firmware service.
- Access fault: enforce protection or terminate execution.

Interrupt Codes

Interrupts are **asynchronous** events. They are identified in `*cause` when the most significant bit is **1**, with the remaining bits encoding the interrupt type.

Standard interrupt classes

- Software interrupt
- Timer interrupt
- External interrupt

Each class exists independently at different privilege targets:

- Machine-level interrupts
- Supervisor-level interrupts

Conceptual mapping

- Software interrupts: inter-hart signaling and coordination.
- Timer interrupts: scheduling and timekeeping.
- External interrupts: device and I/O events.

Key properties

- Interrupts are taken only when pending, enabled, and globally unmasked.
- Interrupt cause codes identify the source class, not the device ID.
- Device-specific identification is performed by platform interrupt controllers.

Decoding `mcause` / `scause`

The `mcause` and `scause` registers share a common encoding model:

- Most significant bit (MSB): interrupt flag
- Remaining bits: cause code

Decoding algorithm (RV64 conceptual)

- If MSB = 0: exception
- If MSB = 1: interrupt
- Cause code = low 63 bits

Example 1: Decode interrupt vs exception

```
/* Decode scause on RV64:
   MSB = 1 => interrupt
   MSB = 0 => exception */
.text
.global decode_scause_type
decode_scause_type:
    csrr t0, scause
    srli t1, t0, 63        /* t1 = 1 => interrupt, 0 => exception */
    ret
```

Example 2: Extract the cause code

```
/* Extract cause code (low 63 bits on RV64) */
.text
.global decode_scause_code
```

```
decode_scause_code:
    csrr t0, scause
    li   t1, -1
    srli t1, t1, 1          /* mask with low 63 bits set */
    and  t2, t0, t1         /* t2 = cause code */
    ret
```

Example 3: Minimal trap classification skeleton

```
/* Conceptual trap classification fragment */
.text
.global trap_classify
trap_classify:
    csrr t0, scause
    srli t1, t0, 63         /* interrupt? */
    beqz t1, handle_exception

handle_interrupt:
    /* dispatch timer / software / external */
    ret

handle_exception:
    /* dispatch syscall / page fault / illegal instruction */
    ret
```

Design rules for correct decoding

- Always decode interrupt vs exception first.
- Never assume numeric values without masking MSB.

- Use `*tval` only when meaningful for the decoded cause.
- Do not conflate access faults with page faults; treat them separately.

Takeaway: `mcause` and `scause` provide a compact, uniform encoding for all trap reasons. Correct trap handling begins with disciplined decoding: first classify interrupt vs exception, then dispatch based on the architectural cause code, using `*epc` and `*tval` to complete diagnosis and recovery.

Appendix B — Privilege Transition Rules

This appendix summarizes the **architectural rules that govern privilege transitions** in RISC-V systems. Correct handling of these rules is mandatory for security, correctness, and predictable trap behavior.

Legal and Illegal Mode Transitions

RISC-V defines three primary privilege modes in common OS designs: **U-mode**, **S-mode**, and **M-mode**. Transitions between them are **not arbitrary**; they occur only through well-defined architectural mechanisms.

Legal transitions (architecturally allowed)

- **U $\rightarrow$ S**: via a trap (e.g., `ecall`, page fault, interrupt) handled in S-mode.
- **U $\rightarrow$ M**: via a trap handled in M-mode (if not delegated).
- **S $\rightarrow$ M**: via a trap handled in M-mode (if not delegated).
- **M $\rightarrow$ S**: via `mret` with `mstatus.MPP` configured to supervisor.

- **S** → **U**: via `sret` with `sstatus.SPP` configured to user.
- **M** → **U**: via `mret` with `mstatus.MPP` configured to user (rare in OS designs).

Illegal transitions (architecturally forbidden)

- Direct jumps between modes without a trap or privileged return.
- Executing `sret` or `mret` from insufficient privilege.
- Manually changing privilege by writing status fields from lower privilege.
- Returning to a higher privilege than recorded in the saved status field.

Example 1: Illegal attempt to return to supervisor from U-mode

```
/* Executed in U-mode: illegal instruction exception */
.text
.global u_try_sret
u_try_sret:
    sret
```

Example 2: Legal M-mode handoff to S-mode using mret (conceptual)

```
/* M-mode sets return state, then mret enters S-mode */
.text
.global m_to_s_entry
m_to_s_entry:
    la   t0, s_kernel_entry
    csrw mepc, t0

    /* mstatus.MPP must be set to supervisor (field ops omitted) */
    /* ... */

    mret

s_kernel_entry:
    j s_kernel_entry
```

Delegation Rules Summary

Delegation controls whether a trap originating at lower privilege is handled in S-mode or escalated to M-mode.

Delegation concept

- M-mode is the root authority.
- M-mode may delegate specific exceptions and interrupts to S-mode.
- Delegated traps are handled entirely by S-mode using `stvec`, `scause`, `sepc`, and `stval`.
- Non-delegated traps always enter M-mode.

Two delegation control planes

- **Exception delegation**: controls which synchronous exceptions S-mode may handle.
- **Interrupt delegation**: controls which asynchronous interrupts S-mode may handle.

Practical OS design pattern

- Delegate most page faults, syscalls, and timer interrupts to S-mode.

- Keep machine-check, platform-critical, and firmware-specific events in M-mode.
- Avoid unnecessary M-mode involvement in normal process execution.

Example 3: Effect of delegation on syscall handling

- If environment-call-from-U is delegated: `ecall` enters S-mode directly.
- If not delegated: `ecall` enters M-mode first, which must forward or emulate.

```
/* Conceptual: S-mode trap handler sees delegated ECALL */
    csrr t0, scause      /* indicates environment call from U-mode */
    csrr t1, sepc
    /* handle syscall directly in kernel */
```

Delegation rule you must not violate

If a trap is delegated, M-mode must not assume it will see it. If a trap is not delegated, S-mode must not assume it can handle it.

Common Design Mistakes

This section lists frequent and serious errors encountered in early kernel, firmware, and educational OS implementations.

Mistake 1: Assuming direct privilege jumps are possible

Attempting to jump directly from U-mode into kernel code without a trap breaks the privilege model and will either fault or create a security hole if mappings are incorrect.

Mistake 2: Misusing status fields

- Writing entire `*status` registers instead of modifying specific bits.
- Failing to preserve previous interrupt-enable fields.
- Corrupting saved privilege fields before `sret`/`mret`.

Mistake 3: Forgetting to advance EPC on syscalls

If `sepc` is not advanced after handling `ecall`, the same syscall will re-execute indefinitely.

```
/* Correct syscall completion pattern */
    csrr t0, sepc
    addi t0, t0, 4
    csrw sepc, t0
```

Mistake 4: Confusing delegation with permission

Delegation decides *where* a trap is handled, not *what* S-mode is allowed to do. Even delegated traps do not grant S-mode access to M-mode CSRs or machine-only state.

Mistake 5: Enabling interrupts too early

Re-enabling interrupts before saving context can cause nested traps that overwrite an incomplete trap frame.

Mistake 6: Treating M-mode as a fast-path kernel

M-mode is not designed for frequent entry in OS execution. Using M-mode for normal syscalls increases latency, complexity, and attack surface.

Mistake 7: Assuming one-size-fits-all privilege flows

Bare-metal firmware, embedded RTOS, and full OS kernels use different privilege flows. Design must reflect the target environment, not generic assumptions.

Takeaway: Privilege transitions in RISC-V form a strict state machine. Legal transitions occur only through traps and privileged returns, delegation controls where traps are handled, and most security failures stem from violating these rules or assuming shortcuts. A correct design is explicit, disciplined, and aligned with the architectural contract.

Appendix C — Minimal Trap & Syscall Flow

This appendix gives a compact, end-to-end view of the **minimal correct execution path** for a user-mode syscall on a typical OS-style RISC-V system where traps from U-mode are handled in S-mode. The purpose is to make the full pipeline concrete: **User** → **Trap** → **Kernel** → **Return**.

End-to-End Execution Path

Assumptions (minimal OS model)

- U-mode executes applications.
- S-mode executes the kernel.
- The kernel trap vector (`stvec`) points to an entry stub.
- The syscall ABI uses `a7` for syscall number and `a0..a5` for arguments (conceptual).
- Syscalls return a result in `a0`.

Pipeline overview (what hardware does vs what software does)

Hardware (on trap entry):

- writes `sepc` (resume PC),
- writes `scause` (trap reason),
- writes `stval` (extra detail when defined),
- updates `sstatus` previous-state fields and disables S-mode interrupts,
- sets PC to `stvec` (direct or vectored rules).

Software (kernel trap entry):

- saves enough GPR state into a trap frame,
- decodes `scause` to classify syscall vs fault vs interrupt,
- for syscall: reads syscall number and args, validates, dispatches,
- writes return value(s),
- advances `sepc` to skip `ecall`,
- restores context and returns via `sret`.

User → Trap → Kernel → Return

Step 1: User-mode issues a syscall request

```
/* U-mode: syscall stub (conceptual)
   a7 = syscall number
   a0..a2 = arguments
   a0 = return value */
.text
.global u_syscall_example
u_syscall_example:
    li    a7, 5            /* syscall id (example) */
    li    a0, 100          /* arg0 */
    li    a1, 200          /* arg1 */
    li    a2, 300          /* arg2 */
    ecall                  /* trap to S-mode kernel */
    ret                    /* resumes after kernel returns via sret */
```

Step 2: CPU trap entry records metadata and jumps to stvec

The CPU takes the trap and records:

- `sepc` = address of the trapping `ecall`
- `scause` = environment call from U-mode (exception)
- `stval` = 0 or unused (for this cause, typically)

Then it transfers control to `stvec`.

Step 3: Kernel trap entry saves context and reads trap CSRs

```
/* S-mode: minimal trap entry skeleton (conceptual) */
.text
.global s_trap_entry_minimal
s_trap_entry_minimal:
    addi sp, sp, -128

    /* Save essential state (subset for illustration) */
    sd   ra, 0(sp)
    sd   a0, 8(sp)
    sd   a1, 16(sp)
    sd   a2, 24(sp)
    sd   a7, 32(sp)

    /* Read trap metadata */
    csrr t0, scause
    csrr t1, sepc
    csrr t2, stval

    /* If this is the syscall exception, dispatch */
    /* Otherwise, handle fault/interrupt elsewhere */
    call s_syscall_dispatch_minimal

    /* Restore */
    ld   ra, 0(sp)
    ld   a0, 8(sp)
    ld   a1, 16(sp)
    ld   a2, 24(sp)
    ld   a7, 32(sp)

    addi sp, sp, 128
    sret

/* Placeholder for linkage */
s_syscall_dispatch_minimal:
    ret
```

Step 4: Minimal syscall dispatch reads a7 and returns a0

A minimal dispatcher:

- reads syscall number (in `a7` by convention),
- reads args (in `a0..a5`),
- performs the service,
- returns result in `a0`.

```
/* Minimal syscall dispatch (conceptual).
   Here: syscall id 5 returns a0 = a0 + a1 + a2 (example service). */
.text
.global s_syscall_dispatch_minimal
s_syscall_dispatch_minimal:
    /* Check syscall number */
    li   t0, 5
    bne  a7, t0, unknown_syscall

    /* Example service: sum arguments */
    add  a0, a0, a1
    add  a0, a0, a2

    /* Advance sepc to skip ECALL (assumes 4-byte ECALL encoding) */
    csrr t1, sepc
    addi t1, t1, 4
    csrw sepc, t1
    ret

unknown_syscall:
    /* Example error convention: return -1 */
    li   a0, -1

    /* Still advance sepc to avoid re-executing ECALL */
    csrr t1, sepc
    addi t1, t1, 4
    csrw sepc, t1
    ret
```

Step 5: Return to user via sret restores privilege and interrupt state

When the trap entry stub executes `sret`:

- PC resumes from `sepc` (which we advanced past `ecall`),
- privilege returns to U-mode (as recorded in `sstatus` previous-privilege field),
- interrupt-enable state is restored from the saved previous state.

Step 6: User resumes as if syscall were a function call

Execution continues after the `ecall` instruction. The return value is in `a0`.

```
/* After return:
   - a0 contains result
   - PC is after ecall
   - still in U-mode */
.text
.global u_after_syscall
u_after_syscall:
    /* a0 already has syscall result */
    ret
```

Key correctness rules (minimal but non-negotiable)

- Save registers before calling other code.
- Decode `scause` to ensure you are handling the intended trap.
- Advance `sepc` for syscalls; do not advance for retry-faults (e.g., demand paging).
- Return via `sret`; never use a normal `ret` to exit a trap.
- Keep the handler deterministic and bounded; defer heavy work when needed.

Takeaway: A syscall is a disciplined, end-to-end state machine. User code requests a service with `ecall`, hardware records trap metadata and transfers control, the kernel saves context and dispatches by ABI, then returns using `sret` after advancing `sepc`. This pipeline is the foundation for safe services, isolation, and reliable OS control.

References

RISC-V Privileged Architecture (Conceptual)

Primary specifications (authoritative)

- **The RISC-V Instruction Set Manual, Volume II: Privileged Architecture**
  Core authority for: privilege levels (U/S/M), trap model, delegation, interrupt architecture, and CSRs such as `mstatus/sstatus`, `mtvec/stvec`, `mepc/sepc`, `mcause/scause`, `mtval/stval`.

- **The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA**
  Defines `ecall`, `ebreak`, CSR instruction semantics (`csrrw/csrrs/csrrc`), and instruction encoding rules needed to reason about trap precision and EPC advancement.

Platform interface (common in real systems)

- **RISC-V Supervisor Binary Interface (SBI) Specification**
  Defines the conceptual contract for S-mode to request machine services (timers, IPIs, platform services) via `ecall`. This matters because many systems route timer programming through SBI even when the OS handles the interrupt in S-mode.

How to read these specs for this booklet

Focus on these conceptual clusters:

- **Trap**
  entry/return state machine: what gets written on entry (`*epc/*cause/*tval`) and what `sret/mret` restore.
- **Delegation mechanism:** which traps can be handled in S-mode and which must go to M-mode.
- **Interrupt gating:** pending vs enable vs global enable, and how status fields snapshot/restore.

Spec-driven micro-examples (what each document must answer)

```
/* Privileged Architecture must define:
   - what scause encodes (interrupt bit + code)
   - what sepc means (resume PC)
   - what sret restores (privilege + interrupt enable) */
.text
.global ref_privileged_min
ref_privileged_min:
    csrr t0, scause
    csrr t1, sepc
    csrr t2, stval
    sret
```

```
/* Unprivileged ISA must define:
   - ecall semantics as an environment-call exception
   - CSR instruction semantics as atomic read/modify/write */
```

```
.text
.global ref_unpriv_min
ref_unpriv_min:
    csrr a0, cycle
    ecall
    ret
```

RISC-V psABI (Conceptual)

Primary ABI document (authoritative)

- **RISC-V ELF psABI Specification**
  Core authority for: calling convention, register roles (`a0..a7`, `t*`, `s*`, `sp`, `ra`), stack alignment rules, function prologues/epilogues, ELF object conventions, and how C/C++ code expects registers to behave.

Why psABI matters for traps and syscalls

Trap entry is not a normal call; you must **manufacture** an ABI-valid state before calling higher-level code. The psABI is the document that defines:

- which registers are caller-saved vs callee-saved,
- required stack alignment (especially critical before calling C/C++),
- how arguments and return values are represented (scalars, pointers, 64-bit values).

ABI-driven trap-frame example

This is the conceptual bridge between privileged traps and language-level handlers.

```
/* psABI must answer:
   - which registers must be preserved across a C call
   - what stack alignment is required before 'call' */
.text
.global ref_psabi_bridge
ref_psabi_bridge:
    addi sp, sp, -64

    /* Save registers that the dispatcher may clobber (illustrative
     ↪  subset) */
    sd   ra, 0(sp)
    sd   s0, 8(sp)
    sd   s1, 16(sp)

    /* Pass trap metadata as normal ABI arguments */
    csrr a0, scause
    csrr a1, sepc
    csrr a2, stval
    call trap_dispatch_c

    ld   ra, 0(sp)
    ld   s0, 8(sp)
    ld   s1, 16(sp)
    addi sp, sp, 64
    sret

trap_dispatch_c:
    ret
```

Syscall ABI vs psABI

- **psABI** defines function calls.
- **Syscall ABI** is OS-defined. Many OSes choose a convention that resembles psABI register usage (e.g., syscall number in `a7`, args in `a0..a5`, return in `a0`), but that convention is **not** mandated by the psABI itself.

OS and Systems Programming Foundations

Operating systems (core concepts)

Use these as conceptual foundations for why traps/interrupts/syscalls exist and how kernels are structured:

- **Modern Operating Systems** (textbook)
  Process model, syscalls, interrupts, scheduling, virtual memory, protection domains.
- **Operating Systems: Three Easy Pieces (OSTEP)**
  High-signal conceptual grounding: CPU virtualization, memory virtualization, and concurrency.
- **Computer Systems: A Programmer's Perspective (CS:APP)**
  Bridges ISA-level behavior with process, exceptions, linking, and system interfaces.

Kernel implementation references

These provide practice-oriented grounding for trap frames, syscall dispatch, and interrupt handling structure:

- **Linux kernel documentation and source (RISC-V architecture port)**

Shows real trap entry stubs, register saving strategies, and syscall/interrupt wiring choices.

- **xv6-style teaching kernels (RISC-V editions)**
  Minimal, readable trap/syscall pipeline: ideal for validating your mental model end-to-end.

Foundational cross-check examples

These examples describe what a systems reference must clarify at a conceptual level.

```
/* OS foundations must explain:
   - why timer interrupts drive scheduling
   - why syscalls require argument validation
   - why page faults can be used for demand paging */
.text
.global ref_os_triangle
ref_os_triangle:
    /* timer interrupt -> scheduler tick */
    /* syscall -> privilege boundary */
    /* page fault -> map and retry */
    ret
```

```
/* Minimal end-to-end syscall (conceptual):
   user sets a7/a0.., ecall, kernel advances sepc, sret returns */
.text
.global ref_end_to_end
ref_end_to_end:
    li   a7, 1
    li   a0, 42
    ecall
    ret
```

How to use these references for maximum correctness

- Use the **Privileged Architecture** to define trap legality, CSR semantics, delegation, and `sret`/`mret`.
- Use the **Unprivileged ISA** to define instruction semantics (`ecall`, CSR ops) and precision guarantees.
- Use the **psABI** to define register discipline and stack rules when calling into higher-level code.
- Use OS foundations to validate **design intent**: why interrupts preempt, why syscalls validate, why faults retry.