Some CS3410 (or equivalent) topics you might have forgotten

RVR

January 2026

Purpose and scope

This handout re-establishes core terminology and a shared mental model for an operating systems course. It deliberately presents a simplified view of a computer. The goal is conceptual alignment, not hardware fidelity.

What this model ignores (on purpose). To keep the initial picture clean, we ignore or postpone discussion of: caches, TLBs, virtual memory/page tables, memory reordering, DMA subtleties, device trees, IOMMUs, and detailed interconnect fabrics (e.g., AXI). We will revisit and refine this model as the course progresses.

1 A simplified computer model

At a high level, a computer consists of:

- one or more Central Processing Units (CPUs), each potentially with multiple cores (or "harts" in RISC-V terminology),
- memory (RAM, ROM/flash, ...),
- a collection of peripherals (devices),
- an interconnect that allows cores and devices to read and write memory and device registers.

In older textbooks this interconnect is often described as an *address bus*, a *data bus*, and a *control bus* (Figure 1). Modern systems-on-chip implement the same logical function using more complex fabrics. For OS purposes, the key idea is that cores can issue reads and writes to addresses, and those accesses are routed to either memory or devices.

2 Bits, bytes, words, and addresses

Memory is organized as *bytes*, each consisting of 8 bits. Each byte (not each bit) has an address.

If an address has x bits, then in principle the address space contains 2^x distinct byte addresses. Common cases:

- **RV32:** $x = 32$ (up to 2^{32} bytes = 4 GiB address space),
- **RV64:** $x = 64$ (architecturally 2^{64} bytes, though systems may implement fewer usable bits).

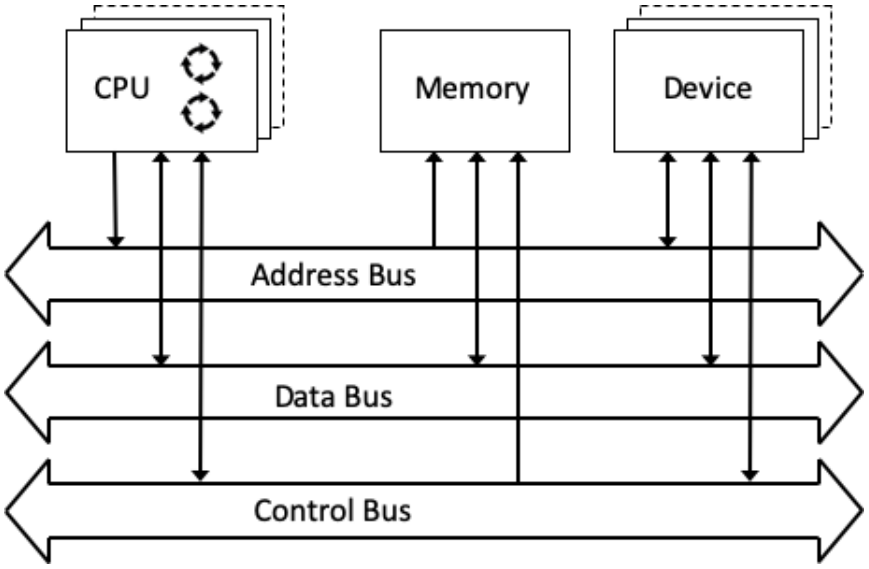

Figure 1: Insert schematic of CPU/memory/device connected by an interconnect (formerly "buses").

On a y-bit machine, a *word* is y bits. Typical cases are $y = 32$ and $y = 64$. The address of a word is the address of its first byte. Architectures differ in byte ordering (endianness); RISC-V is typically little-endian in the systems used in this course.

Hexadecimal notation

Addresses are commonly written in hexadecimal. One hex digit encodes 4 bits, so an x-bit address is written with $\lceil x/4 \rceil$ hex digits (including leading zeros). We often prepend `0x` to indicate hexadecimal.

For example, a 32-bit address ranges from `0x00000000` to `0xFFFFFFFF`. The set of all representable addresses is the *address space*.

3 Registers and core state

Each core has registers. Some hold data; others hold addresses. Two registers are especially important:

- **Program counter (PC):** the address of the next instruction to execute.
- **Stack pointer (SP):** the address of the top of the current stack.

RISC-V also has a standard calling convention. For example, the return address is held in register `ra`.

4 Loads, stores, and memory operations

A core reads memory by executing a *load* instruction and writes memory by executing a *store* instruction. Conceptually:

- A load reads bytes from a memory address and places the result into a register.
- A store writes bytes from a register to a memory address.

At the hardware level, these operations travel over the interconnect to either memory or a device.

Atomic operations

In addition to ordinary loads and stores, most architectures provide operations that are *atomic* (indivisible) with respect to other cores. These matter whenever multiple cores share memory concurrently.

RISC-V supports atomics via two families of mechanisms:

- **Load-reserved/store-conditional (LR/SC):** useful for building locks and lock-free algorithms.
- **AMOs (atomic memory operations):** read-modify-write operations such as atomic swap or add, when the extension is available.

A classic example of an atomic primitive is *test-and-set*, which reads a memory location and sets it to a specified value, returning the old value. We will see how such primitives underpin synchronization.

5 Memory organization for a running program

A running program typically uses several logical regions of memory (see Figure 2):

- **Code (text) segment:** machine instructions and read-only constants.
- **Data segment:** global variables and static storage.
- **Heap:** dynamically allocated memory (e.g., via `malloc`); grows "up".
- **Stack:** activation records for function calls; grows "down".

The PC should point into the code segment. The SP points near the top of the stack region. The heap and stack grow toward each other; ideally they do not meet.

When multiple cores are present, each core typically has its own stack, while code and (some) data may be shared.

6 Function calls and saving context

When a core executes a function call, it must preserve enough state to return. This saved state is called the *call context*. At minimum it includes the return address; often it includes additional registers.

On RISC-V:

- The call instruction sets `ra` (the return address register).
- The callee saves any registers it must preserve (according to the ABI) by storing them on the stack.

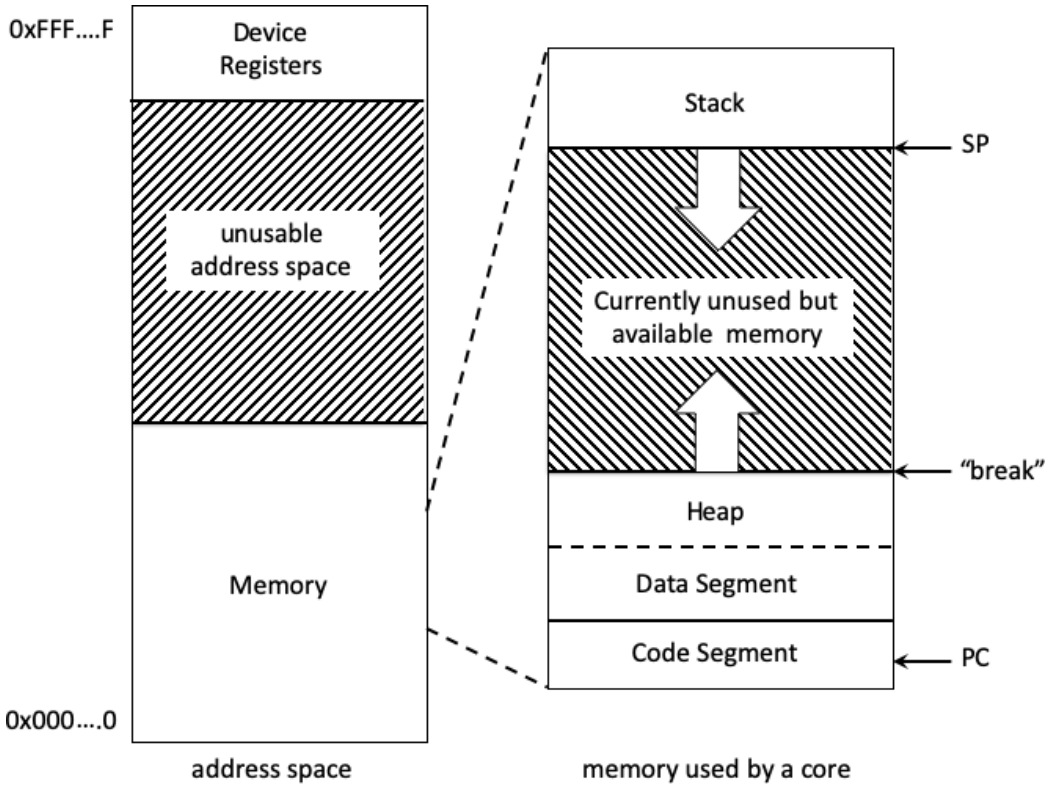

Figure 2: Insert an address-space diagram showing code/data/heap/stack and the idea of device registers mapped near the top.

On return, the callee restores those registers from the stack and returns to the address in `ra`. Not all registers are saved on every call; which ones must be preserved depends on the calling convention.

7 Devices and memory-mapped I/O

Computers include many peripherals: storage, network interfaces, clocks, serial ports, displays, keyboards, and more. The OS interacts with devices through *device interfaces*.

A common technique is *memory-mapped I/O (MMIO)*: device control/status registers are assigned addresses within the processor's address space. Reads and writes to those addresses are routed to the device rather than RAM.

7.1 Device registers example: block device

Many devices expose a small set of registers. A block device might provide:

- **Block number register:** which block on the device to access.
- **Memory address register:** where in memory to transfer data.
- **Command register:** operation to perform (read, write, ...).
- **Status register:** completion/error status.

To read a block, the core writes the block number, writes the destination memory address, then writes a "read" command. The device performs the operation asynchronously. When it finishes (or errors), it typically raises an interrupt.

8 Signals, exceptions, and interrupts

A core's execution can be disrupted by *signals* that transfer control to a handler. There are two broad kinds:

8.1 Synchronous signals (exceptions/faults)

These occur deterministically as a direct result of executing an instruction. Examples include:

- divide-by-zero,
- accessing an invalid or privileged address,
- executing an illegal instruction,
- system calls (e.g., `ecall`),
- breakpoints (for debugging).

8.2 Asynchronous signals (interrupts)

These originate externally to the current instruction stream, usually from devices. Examples include:

- timer interrupt,
- disk completion interrupt,
- network packet arrival interrupt.

Interrupts are typically *maskable*: a core can enable/disable them (globally or by source). If interrupts are disabled, an interrupt is usually delayed (pending) rather than lost, though the details depend on the platform and interrupt controller.

9 Traps and trap handling

A *trap* is a transfer of control to privileged code in response to an exception or interrupt. On trap entry, hardware performs some standard actions:

- record the cause of the trap (exception code or interrupt source),
- save the current PC (the point to resume later, if applicable),
- set the PC to the trap handler entry point.

On RISC-V, trap state is recorded in control and status registers (CSRs), such as:

- `sepc` (saved exception PC) or `mepc` in machine mode,
- `scause`/`mcause` (trap cause),
- `stval`/`mtval` (trap value, when relevant),
- `sstatus`/`mstatus` (status bits, including interrupt enable).

9.1 Saving and restoring full context

The trap handler typically begins by saving the full register context (general registers and any other necessary state) to a trap frame on the stack. It then services the trap (e.g., handles the device interrupt or performs a system call).

If it is appropriate to resume the interrupted code, the handler restores the saved registers and executes a *return-from-trap* instruction. On RISC-V this is typically `sret` (or `mret` in machine mode), which restores the PC and returns to the prior privilege level, re-enabling interrupts as dictated by status bits.

Why handlers should be short. Trap handlers often run with interrupts disabled (at least for some classes of interrupts). Long handlers increase interrupt latency and can harm system responsiveness. A common design is to do minimal work in the handler and defer longer processing to kernel threads or bottom halves.

10 Critical sections and race conditions

If normal code and a trap handler both access the same data structure, concurrent access can create a *race condition*. For example, a keyboard interrupt handler might push characters onto a queue while normal code pops them. Without proper synchronization, the queue can be corrupted.

A basic strategy is:

1. disable interrupts (or otherwise prevent concurrent access),
2. perform the critical operation,
3. re-enable interrupts.

On multicore systems, disabling interrupts on one core does not prevent other cores from accessing shared memory; in that case, locks or other synchronization mechanisms are required in addition to (or instead of) interrupt masking.

11 Common units of memory

Memory sizes are expressed using both decimal and binary-ish prefixes. In OS work, the binary powers of two are especially common:

Common name	**Abbr.**	**Binary name**	**Size**
kilobyte	KB	kibibyte (KiB)	$2^{10} \approx 10^3$ bytes
megabyte	MB	mebibyte (MiB)	$2^{20} \approx 10^6$ bytes
gigabyte	GB	gibibyte (GiB)	$2^{30} \approx 10^9$ bytes
terabyte	TB	tebibyte (TiB)	$2^{40} \approx 10^{12}$ bytes
petabyte	PB	pebibyte (PiB)	$2^{50} \approx 10^{15}$ bytes