Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Who Am I?

Gabriele Tolomei

Associate Professor of Computer Science, Sapienza University of Rome

- Leads the **HERCOLE Lab** (**H**uman-**E**xplainable, **R**obust, and **CO**llaborative **LE**arning)
- Co-founder & CSO at **tellmewAI** (AI Act compliance platform)

Research pillars: Explainable AI · Adversarial & Robust ML · Federated & Scalable Learning

Background: Ph.D. Ca' Foscari Venice · M.Sc. & B.Sc. University of Pisa · Previously at Yahoo Labs (London)

Useful Information: I Canale A–L

Class Schedule

When:

- Tue, 15:00–18:00; Wed, 16:00–18:00; Thu, 13:00–15:00

Where:

- Aula 4, Via De Lollis (map)

Contacts

- Lecturer: **Gabriele Tolomei** (tolomei@di.uniroma1.it)
- Office: Stanza 106, I Piano, Edificio E, Viale Regina Elena 295 (map)
- Course website: https://gtolomei.github.io/teaching/os.html
- Google Classroom (mandatory): click to join

Office Hours

Drop me a message and we'll arrange a meeting time, either in person or online! 🙂

Useful Information: Il Canale M–Z

Class Schedule

When:

- Tue, 13:00–15:00; Thu, 15:00–18:00; Fri, 13:00–15:00

Where:

- Aula 1L, Via del Castro Laurenziano, 7a (map)

Contacts

- Lecturer: **Marco Raoul Marini** (marini@di.uniroma1.it)
- Office: Stanza 314c, III Piano, Via Salaria 113 (map)
- Google Classroom (mandatory): click to join

Office Hours

Drop me a message and we'll arrange a meeting time, either in person or online! 🙂

Class Material

- Released on the course website
- Suggested books (though not mandatory):
  - ***Operating Systems: Three Easy Pieces*** — Remzi and Andrea Arpaci-Dusseau (freely available online)
  - ***Operating System Concepts, 9th Edition+*** — Silberschatz, Galvin, Gagne
  - ***Modern Operating Systems, 4th Edition+*** — Andrew S. Tanenbaum
- Any additional resource available on the Web

Assessment and Grading: Written Test

The final grade is determined through a **two-step** assessment process.

1. Written Test (Mandatory – Maximum: 24 points)

- Duration: **30 minutes**
- Consists of **48** multiple-choice questions/exercises.
- Scoring:
    - **+2** point for each correct answer
    - **0** points for unanswered questions
    - **–1** point for each incorrect answer
- A **minimum score of 18 points** is required to pass the written test and become eligible for the second (optional) step.

Assessment and Grading: Calibration

The final score S is obtained as follows:

$$S = 2C + (-1)E + 0U \quad \Longrightarrow \quad S = 2C - E$$

where:

- C = number of correct answers
- E = number of wrong answers
- U = number of unanswered questions

The theoretical score range is:

$$-48 \leq S \leq 96$$

We fix a threshold T, under which the test is considered failed (**INS**). Then, we need to map the range $[T, 96]$ into $[18, 24]$.

Assessment and Grading: Calibration

We apply a linear scaling to the score S as follows:

$$\boxed{V(S) = \begin{cases} \text{INS}, & S < T, \\ 18 + 6\dfrac{S - T}{96 - T}, & T \le S \le 96, \end{cases}}$$

where T is the minimum bar threshold, e.g., $T = 42$.

Assessment and Grading: Example

Threshold: $T = 42$

#Correct	#Wrong	#Unanswered	Score	% Correct (out of 48)
21	0	27	**42**	43.8%
22	2	24	**42**	45.8%
23	4	21	**42**	47.9%
24	6	18	**42**	50.0%
25	8	15	**42**	52.1%
26	10	12	**42**	54.2%
27	12	9	**42**	56.3%
28	14	6	**42**	58.3%
29	16	3	**42**	60.4%
30	18	0	**42**	62.5%

All combinations above correspond to the sufficiency threshold $S = 42$, which in turn will be mapped to 18.

Assessment and Grading: Oral Exam

The final grade is determined through a **two-step** assessment process.

2. Oral Examination (Optional – Up to ±8 points)

- The oral examination may be taken **only after passing the written test with a (rescaled) score of 18 points or higher**.
- Possible outcomes:
  - ***Good***, ***Very Good***, or ***Excellent*** performance: **up to +8 points**, allowing a final grade of 30 *cum laude* (30 e lode).
  - ***Average*** performance: **0 points**, i.e., no change to the written-test score.
  - ***Poor***, ***Very Poor***, or ***Extremely Poor*** performance: **up to –8 points**, potentially lowering the final grade below 18/30 and resulting in failure of the exam.

Course Organization

The course is structured into **five main modules**:

- **1. Foundations**

  Introduction to computer architecture and the C programming language.

- **2. Virtualization**

  How operating systems abstract and manage hardware resources, including the CPU, memory, and I/O devices.

- **3. Concurrency**

  Mechanisms for coordinating and synchronizing multiple processes and threads that compete for shared resources.

- **4. Persistence**

  Techniques and systems for storing data reliably across reboots, shutdowns, and power failures.

- **5. Advanced Topics**

  Selected topics that extend beyond traditional general-purpose operating systems.

Module 1: Foundations

Lecture 1.1: Fundamentals of Computer Architecture

Philosophy of Systems

> *"The purpose of computing is insight, not numbers."*
> *— Richard Hamming*

And the purpose of Operating Systems is abstraction, not hardware.
However, to build correct abstractions, we must first master the physical machinery.

Lecture Roadmap

This lecture covers the hardware foundations of computing systems, setting the stage for OS abstractions.

Specifically, we will study the pillars of a computer system:

1. **The von Neumann Architecture** — the layout
2. **The CPU** — registers, ALUs, pipelines, and cores
3. **The Memory Hierarchy** — SRAM/DRAM, locality, and caches
4. **The Input/Output (I/O) System** — controllers, interrupts, and DMA
5. **Synthesis** — the hardware/software boundary

> *Let's dive into the machine.*

Why Study Architecture in an OS Course?

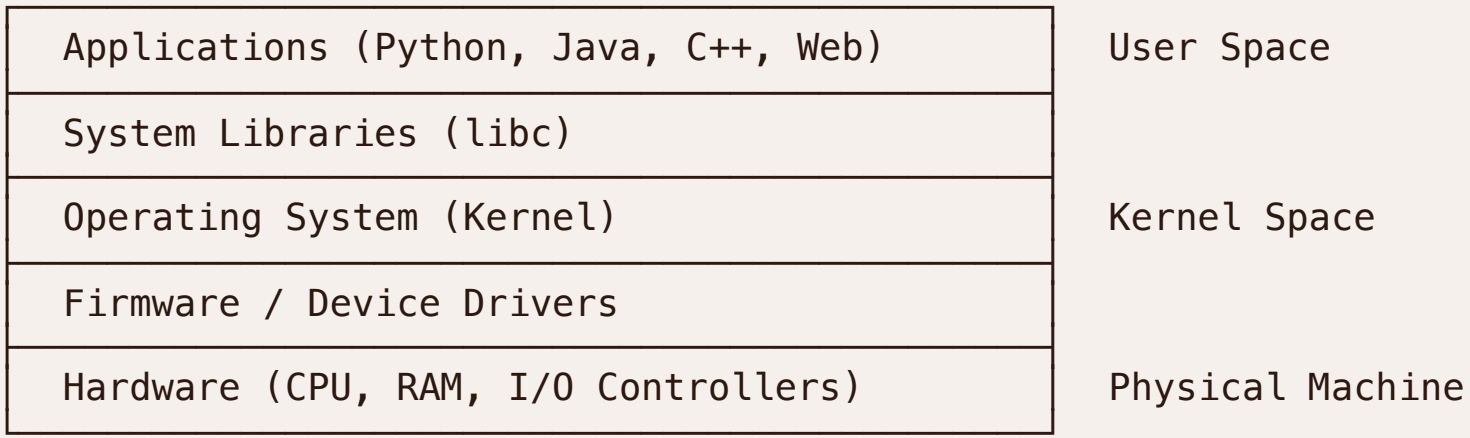
Operating Systems do not run in a vacuum; they are **managers**:

- **Hardware management:** The OS allocates CPU time, memory space, and I/O devices.
- **Abstraction provider:** The OS hides hardware complexity behind simple APIs.
- **Resource protection:** The OS relies on CPU hardware features to enforce security.

Key Lesson: You cannot write a kernel without knowing the hardware it directly controls.

The Hardware-Software Spectrum

Where does the Operating System sit?



The OS acts as the **bridge** between the logical and the physical.

Physics Rules Software

Software abstractions must respect physical limitations:

- **Speed gap:** CPUs run faster than main memory (DRAM) => fraction vs. tens of nanoseconds.
- **Storage trade-off:** Faster storage is expensive and small; cheap storage is slow and massive.
- **Concurrency:** Real multi-core systems require hardware synchronization support.

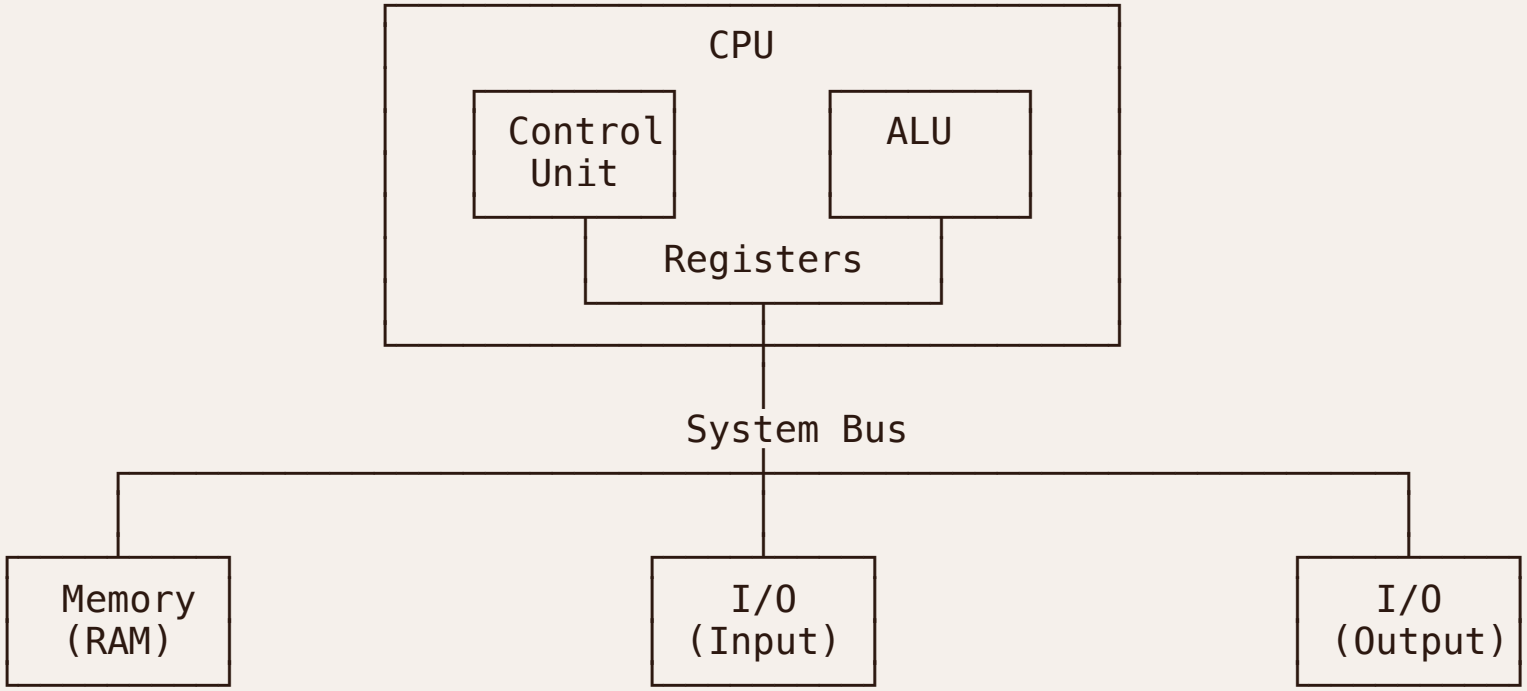
We design OS algorithms to hide these physical limitations.

The von Neumann Architecture (1945)

The theoretical foundation of almost all modern general-purpose computers.

- **Stored-Program Concept:** Instructions and data share the same physical memory space.
- **Sequential Execution:** Instructions are read one by one from memory and executed by the CPU.
- **Components:**
    i. Processing Unit (ALU + Registers)
    ii. Control Unit (PC + IR)
    iii. Memory (for data and instructions)
    iv. Mass Storage / External I/O
    v. Interconnection Bus

The von Neumann Model: Block Diagram



✅ Checkpoint 1 — Architectural Foundations

Let's test our basic understanding:

1. What is the defining characteristic of the **von Neumann Architecture**?
2. Why does an operating system designer need to understand hardware layout?
3. What are the three primary hardware components of a general-purpose computer?
4. What physical limitation does the "Memory Hierarchy" attempt to solve?

Write down your thoughts before moving to Section 1.

§ 1 — The von Neumann Model

Memory Architectures: von Neumann vs. Harvard

There are two classic ways to organize memory:

1. von Neumann Architecture

- Unified memory space for instructions (code) and data.
- Simple hardware design, but suffers from the **von Neumann Bottleneck**: memory access cannot read code and data at the same time.

2. Harvard Architecture

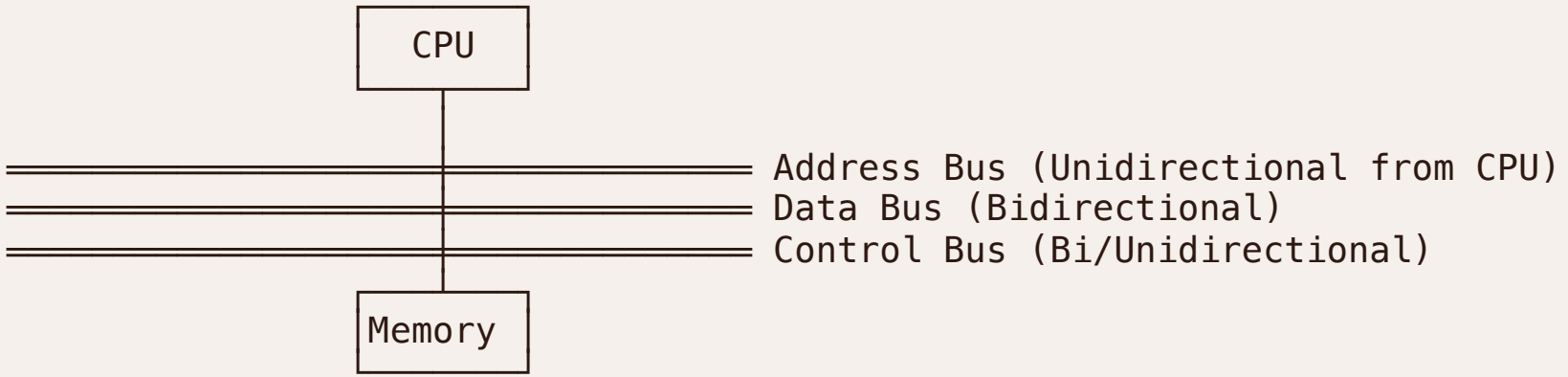
- Physically separate memory spaces and buses for instructions and data.
- Permits simultaneous code fetch and data access. Used extensively in DSPs and L1 CPU caches.

The System Bus

How do CPU, Memory, and I/O talk to each other? They use a shared set of wires called the **System Bus**.

The System Bus is divided into three functional groups:

1. **Address Bus:** Carries memory addresses from CPU to target components.
2. **Data Bus:** Transports actual data bytes between components.
3. **Control Bus:** Transports command signals (read, write, interrupt, reset).



The Address Bus

The **Address Bus** specifies the location in memory or I/O space.

- **Unidirectional:** Only the CPU (or DMA controller) drives address signals on the bus.
- **Bus Width:** The number of parallel lines in the bus determines the maximum addressable locations.
- **Formula:** A bus of N lines can address up to 2^N unique locations.

Example: A 32-bit address bus can access:

$$2^{32}\ \text{bytes} = 4,294,967,296\ \text{bytes} = 4\ \text{GiB}$$

The Data Bus

The **Data Bus** transfers actual values (data and instructions).

- **Bidirectional:** Data flows to and from the CPU (read from memory/write to memory).
- **Bus Width:** Determines how many bits can be transferred in a single clock cycle.
- **Word Size:** Typically matches the processor's word size (e.g., 32-bit or 64-bit).
- Larger data buses improve performance by reducing the number of memory accesses required to fetch wide data structures.

The Control Bus

The **Control Bus** manages bus access and synchronization.

Common Control Signals:

- **Memory Read (MEMR) / Memory Write (MEMW):** Commands memory to place data on the bus or store data from the bus.
- **I/O Read (IOR) / I/O Write (IOW):** Commands I/O devices.
- **Bus Request / Bus Grant:** Used by DMA controller to take control of the bus.
- **Interrupt Signals:** Sent by hardware devices to notify the CPU of an event.
- **Clock:** Synchronizes data transfers.

Let's Calculate: Bus Width and Memory Capacity

Let's look at the mathematical limits of address buses:

Bus Width (Bits)	Total Addressable Locations	Human-Readable Size
16-bit	$2^{16} = 65,536$	64 KiB
20-bit	$2^{20} = 1,048,576$	1 MiB (Intel 8086)
32-bit	$2^{32} = 4,294,967,296$	4 GiB
40-bit	$2^{40} = 1,099,511,627,776$	1 TiB
64-bit	2^{64}	16 EiB (Exabytes)

Modern 64-bit CPUs often implement a 48-bit physical address space (256 TiB), saving hardware costs while providing more than enough RAM space.

The von Neumann Bottleneck

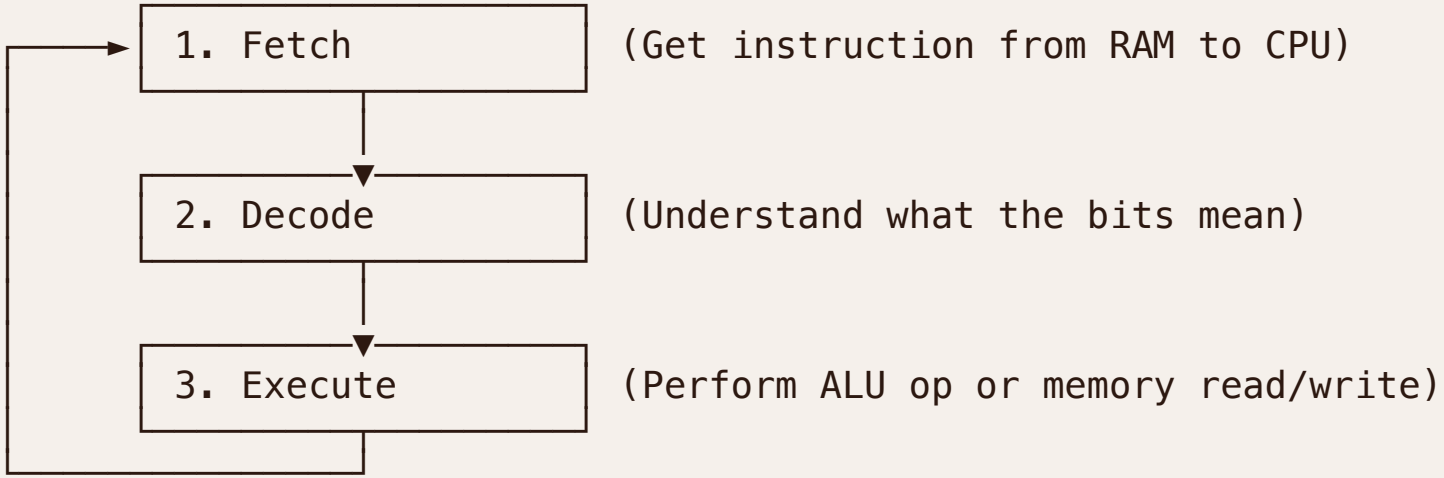
Since instructions and data share the same bus, the CPU cannot read/write both simultaneously.

```
      CPUs run at extremely high speeds (~3 – 5 GHz)
                          ↕
       Bus speeds & Memory access times are slower
```

- The CPU spend significant time waiting for memory transfers to complete.
- **OS Mitigation:** The OS relies on caches, paging, and intelligent scheduling to minimize memory stall latency.

The Instruction Cycle

The base process of computer operation. Every computer program runs this cycle continuously from boot to shutdown:



Inside the Cycle: 1. Fetch

Details of the **Fetch** operation:

1. The CPU copies the value from the **Program Counter (PC)** into the **Memory Address Register (MAR)** and places its content onto the address bus.
2. The Control Unit asserts a **Memory Read** signal on the control bus.
3. Memory reads the address in MAR from the address bus and places the instruction byte(s) on the data bus.
4. The CPU reads the instruction from the data bus into the **Instruction Register (IR)**.
5. The **Program Counter (PC)** is incremented to point to the next instruction.

Inside the Cycle: 2. Decode

Details of the **Decode** operation:

1. The **Control Unit (CU)** reads the bits stored in the **Instruction Register (IR)**.
2. It identifies the **Opcode** (Operation Code, e.g., `ADD`, `LOAD`, `JUMP`).
3. It identifies the **Operands** (registers, memory addresses, or raw values).
4. The CU configures internal data paths (e.g., setting multiplexers, routing signals) to prepare the ALU and registers for execution.

Inside the Cycle: 3. Execute

Details of the **Execute** operation:

- **ALU operations:** If the instruction is arithmetic (e.g., `add %eax, %ebx`), the ALU takes register inputs, performs the calculation, and writes the result back.
- **Memory operations:** If it is a load/store (e.g., `mov (%rax), %rcx`), the CPU reads/writes RAM using MAR and MDR.
- **Control flow:** If it is a jump/branch (e.g., `jmp 0x4010a0`), the PC is updated directly with the target address, changing execution flow.

✅ Checkpoint 2 — Bus & Instruction Cycle

Let's test our understanding:

1. If a system has a 24-bit address bus, what is the maximum amount of byte-addressable memory it can support?
2. What are the three components of the System Bus, and who drives each?
3. Detail the exact register movements that happen during the **Fetch** phase of the instruction cycle.
4. Why does a branch instruction alter the Program Counter?

Discuss with your colleague or write down your answers.

§ 2 — The Central Processing Unit (CPU)

The Central Processing Unit (CPU)

The CPU is the "brain" of the computer, executing instructions stored in memory.

It contains three main building blocks:

1. **The Datapath:** Contains the registers and the Arithmetic Logic Unit (ALU) to manipulate data.
2. **The Control Unit (CU):** The coordinator that fetches instructions, decodes them, and drives the datapath.
3. **Internal Bus:** Internal paths connecting Registers, ALU, and CU.

The Datapath: Processing Data

The Datapath performs the physical operations:

- **Arithmetic Logic Unit (ALU):** A combinational logic circuit that performs:
    - Arithmetic: Addition, subtraction, multiplication, division.
    - Bitwise: AND, OR, XOR, NOT, shifts.
- **General-Purpose Registers (GPRs):** Fast internal storage cells used to hold operands and intermediate results.
- Input ports of the ALU are fed from GPRs, and the output of the ALU is written back to a GPR.

The Control Unit: The Conductor

The Control Unit determines *how* and *when* the datapath acts:

- It is a finite state machine (FSM) synchronized by the system clock.
- **Inputs:** Instruction Register (IR), Flags (status register), Clock signal.
- **Outputs:** Control signals to ALU (select operation), Registers (enable write/read), and Memory/IO control lines.
- It acts like a routing matrix that guides the flow of data through the CPU.

CPU Registers

Registers are the fastest, smallest memory units in the system. They sit at the very top of the memory hierarchy.

They are split into two conceptual groups:

1. **User-visible registers (General-Purpose):** Directly addressable by assembly instructions (e.g., `%rax`, `%rbx` in x86-64).
2. **Control & Status registers (Special-Purpose):** Used by the CU to manage execution state (e.g., PC, IR, PSR).

General-Purpose Registers (GPRs)

GPRs hold data and addresses during execution:

- **x86-64 GPRs:** 16 registers (`%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, and `%r8` to `%r15`).
- **ARM64 GPRs:** 31 registers (`x0` to `x30`).
- GPRs allow calculations to happen inside the CPU without slow memory accesses.
- An assembly instruction like `add %rax, %rbx` runs in less than a nanosecond because it is entirely register-based.

Special-Purpose: Program Counter (PC)

The **Program Counter** (also called Instruction Pointer `IP` in x86):

- Holds the memory address of the **next** instruction to be fetched.
- **Automatic increment:** As soon as an instruction is fetched, the PC automatically increments by the size of the fetched instruction.
- **Modification:** Branches, jumps, and subroutine calls explicitly overwrite the PC, directing execution elsewhere.

Special-Purpose: Instruction Register (IR)

The **Instruction Register** holds the current instruction bits:

- When fetched from memory, the instruction's binary pattern is stored here.
- The IR's output is connected to the instruction decoder inside the Control Unit.
- The IR is **not** directly writable or readable by standard user-space instructions. It is strictly for internal CPU control.

Special-Purpose: Stack Pointer (SP)

The **Stack Pointer** (e.g., `%rsp` on x86-64) holds the address of the top of the call stack.

- Essential for implementing subroutines, function calls, local variables, and return addresses.
- Automatically modified by instruction calls:
  - `push value` → SP decrements, stores value at address in SP.
  - `pop register` → SP reads value at address in SP, increments.
- Directly managed by the compiler to create stack frames.

Special-Purpose: MAR and MDR

These registers buffer communication with the memory bus:

Memory Address Register (MAR)

- Holds the address currently being read from or written to RAM.
- Connected directly to the **Address Bus**.

Memory Data Register (MDR)

- Also called Memory Buffer Register (MBR).
- Holds the data byte(s) read from RAM or waiting to be written to RAM.
- Connected directly to the **Data Bus**.

Special-Purpose: Flags / Program Status Register

The **PSR** (often called `EFLAGS` or `RFLAGS` on x86) contains bit flags indicating the status of the CPU and the last ALU operation.

Key Flags:

- **Zero Flag (Z):** Set to 1 if the last ALU result was zero.
- **Sign/Negative Flag (S/N):** Set to 1 if the last result was negative.
- **Carry Flag (C):** Set if an unsigned addition overflowed.
- **Overflow Flag (V/O):** Set if a signed arithmetic overflowed.
- **Interrupt Enable Flag (I):** Enables/Disables hardware interrupts.

How the OS and Code Use Flags

Flags are essential for decision-making in code:

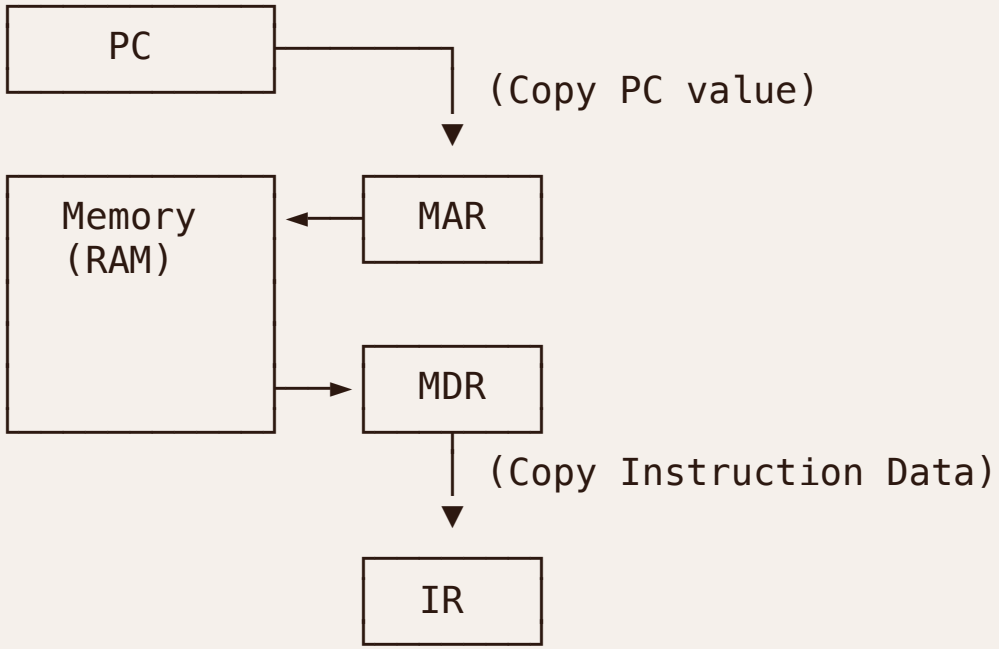
```
int x = 5;
int y = 5;
if (x == y) {
    // Branch taken!
}
```

In assembly, this maps to:

```
cmp eax, ebx   ; Performs subtraction (eax - ebx), updates flags
je  label      ; Jump if Equal (checks if Zero Flag is 1)
```

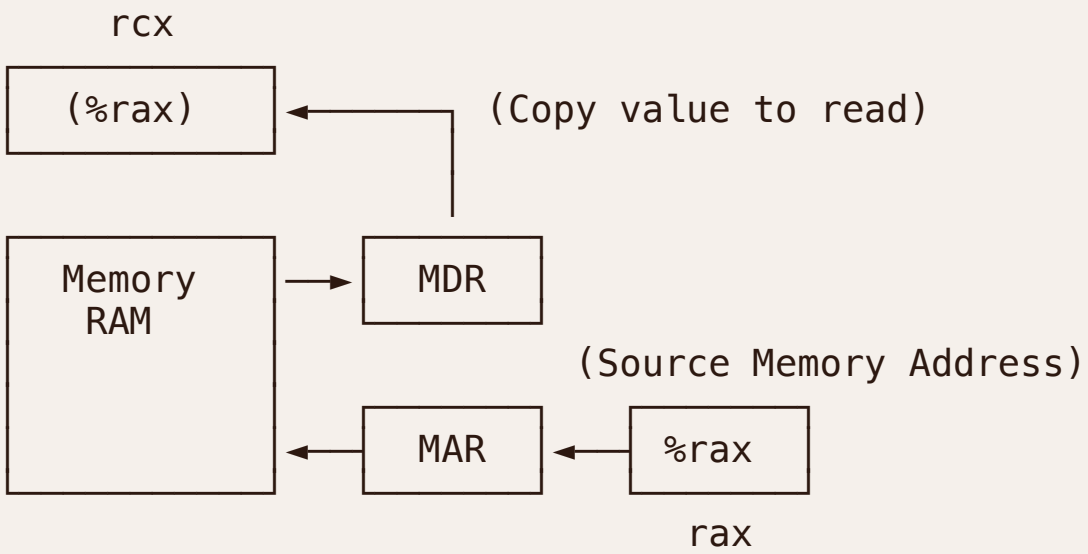
Without flags, conditional execution and loops would be impossible.

Register Interactions during Fetch Phase



Register Interactions during Memory Load

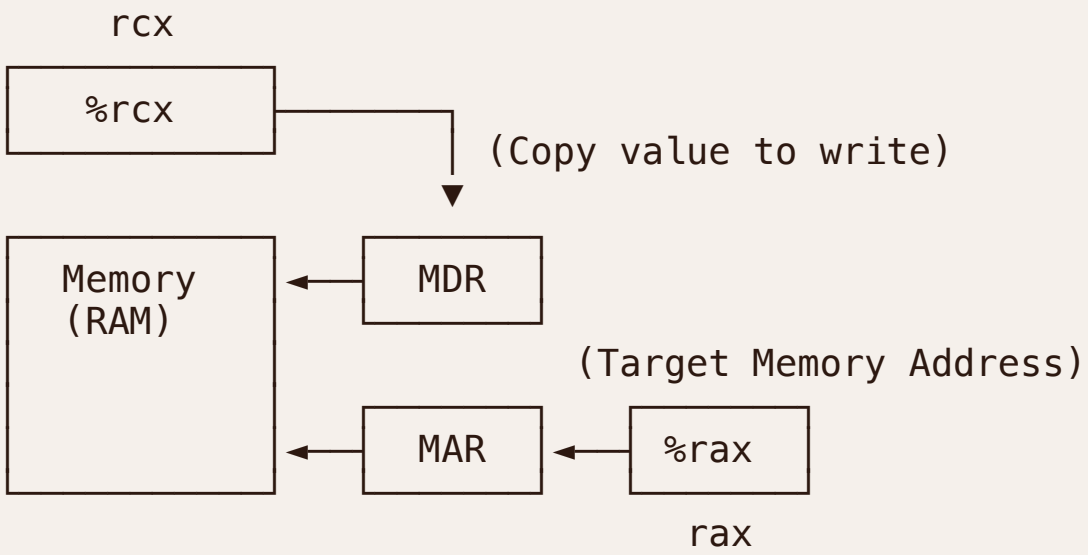
How a memory value is read into a register `mov (%rax), %rcx`:



Eventually, `%rcx = (%rax)`

Register Interactions during Memory Store

How a register value is written back to memory `mov %rcx, (%rax)`:



Eventually, `%(rax) = %rcx`

✅ Checkpoint 3 — CPU Registers & Data Paths

Let's test our understanding:

1. What is the difference between a general-purpose register and a special-purpose register?
2. Explain the relationship between the MAR, MDR, Address Bus, and Data Bus.
3. How does the **Zero Flag (ZF)** allow a program to implement an `if (x == 0)` check?
4. What register tracks the location of the next instruction, and how does it change during normal sequential execution?

Review your answers before moving on.

Instruction Pipelining

In simple processors, the CPU fetches, decodes, and executes one instruction completely before moving to the next.

Pipelining is an optimization technique where multiple instructions are overlapped in execution.

Analogy: An assembly line or laundry:

- You don't wait for your first load of laundry to dry before starting to wash the second load.
- You wash, dry, and iron in parallel.

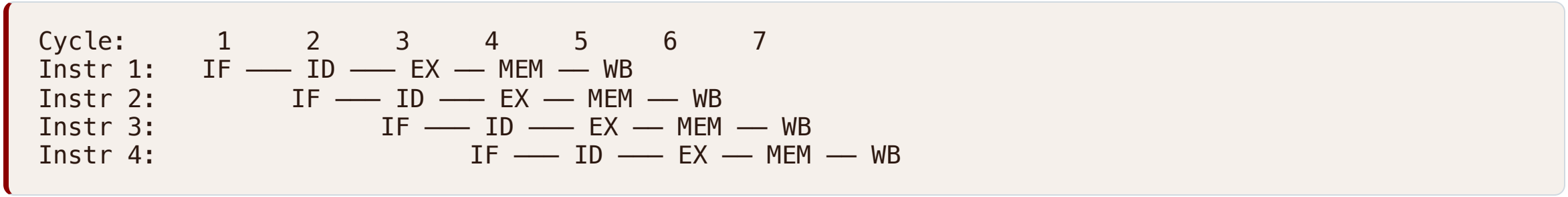
The Classic RISC 5-Stage Pipeline

A standard pipeline divides instruction execution into 5 stages:

1. **IF (Instruction Fetch):** Get instruction from L1 Cache/RAM.
2. **ID (Instruction Decode):** Decode instruction, read registers.
3. **EX (Execute):** Perform ALU calculation or compute address.
4. **MEM (Memory Access):** Read or write data from/to data cache.
5. **WB (Writeback):** Write result back to destination register.

Every stage operates on a different instruction simultaneously in each clock cycle.

Pipelining Diagram: Parallel Execution



- In Cycle 1, only Instr 1 is active (IF).
- In Cycle 5, **five instructions** are executing in parallel at different stages.
- Ideal output: 1 instruction completed per clock cycle ($IPC = 1$).

Pipeline Hazards

Under ideal conditions, pipelines achieve maximum speedup. However, **hazards** prevent the next instruction from executing in its designated clock cycle:

- **Structural Hazards:** Resource conflict where two stages need the same physical hardware (e.g., single memory unit for code fetch and data write).
- **Data Hazards:** Instruction depends on the result of a previous instruction still in the pipeline.
- **Control Hazards:** Branch instructions change the PC, rendering fetched instructions invalid.

Structural Hazards

Occur when two instructions attempt to use the same hardware resource simultaneously.

```
      IF (Fetch Instr 4 from RAM)  <── Conflict!
                  ↕
      MEM (Read Data for Instr 1 from RAM)
```

- **Solution:** Duplicate resources.
    - Modern CPUs use separate L1 instruction and L1 data caches (**Harvard Architecture** at L1 level) to avoid memory access conflicts.

Data Hazards

Occurs when an instruction needs data from a prior instruction that has not yet completed.

```
add %rax, %rbx    ; Writes result to %rbx in stage 5 (WB)
sub %rbx, %rcx    ; Reads %rbx in stage 2 (ID) -> Reads OLD value!
```

- **Solutions:**
  - **Stalling (Bubble):** Pause the dependent instruction until the data is written.
  - **Data Forwarding (Bypassing):** Add internal routing paths to send the ALU result directly to the ALU input of the next stage before it is written back to registers.

Control Hazards

Occur when the CPU fetches instructions before determining the outcome of a branch/jump instruction.

```
cmp eax, ebx
jne target_label   ; If taken, the next fetched instruction is wrong!
mov ecx, edx       ; Fetched during JNE, but must be discarded if taken.
```

- When a branch is taken, all speculative instructions behind it in the pipeline must be cleared (**Pipeline Flush**), wasting clock cycles.

Solving Control Hazards: Branch Prediction

Modern CPUs employ **Branch Predictors** to guess the branch outcome:

- **Static Prediction:** Guess based on code direction (e.g., loops branches are assumed taken, forward branches not taken).
- **Dynamic Prediction:** CPU maintains history tables (Branch Target Buffer) to track past behavior.
- **Speculative Execution:** CPU executes instructions along the predicted path. If the prediction was wrong, it rolls back the state and flushes the pipeline.

OS Impact: Branch mispredictions cause latency. High-performance code is written to avoid branchy logic.

Instruction Set Architectures (ISA)

The ISA defines the interface between hardware and software.

CISC (Complex Instruction Set Computer)

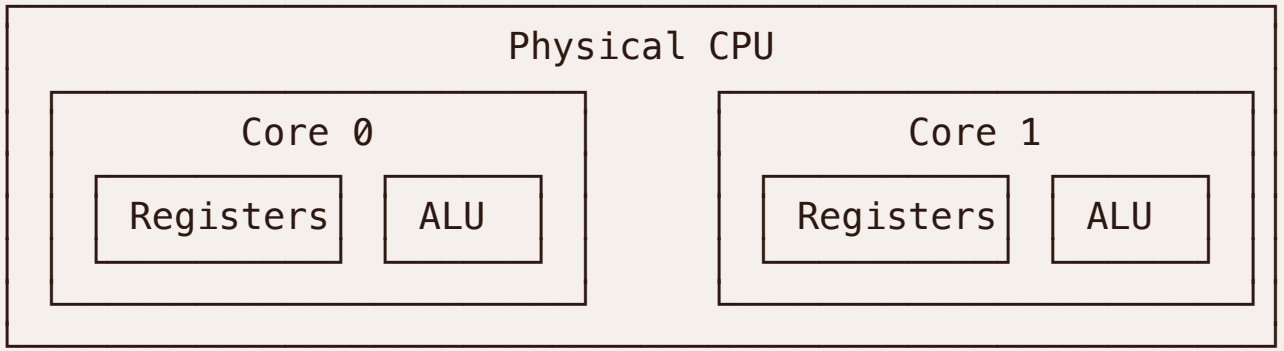
- Large instruction set, variable length.
- Rich instruction set (e.g., directly add memory value to a register).
- *Example:* Intel x86/x86-64.

RISC (Reduced Instruction Set Computer)

- Small, uniform instructions, fixed length.
- Load/Store architecture: arithmetic only operates on registers.
- *Example:* ARM (Apple M-series, mobile devices, servers), RISC-V.

Multi-Core Processor Architectures

To bypass power walls, chip designers put multiple processors (cores) on a single physical die.

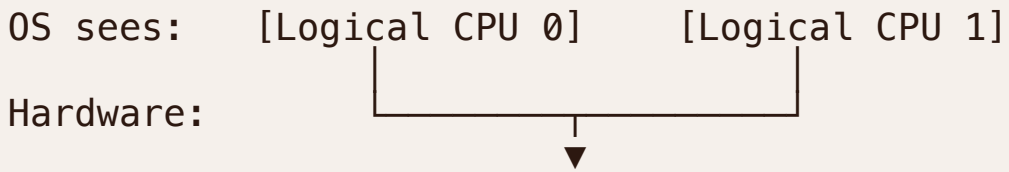


- Each core is an independent CPU capable of running its own instruction cycle.
- **OS Role:** The OS scheduler must allocate processes/threads across these cores.

Simultaneous Multithreading (SMT)

Also known as **Hyper-Threading** (Intel's terminology):

- A single physical CPU core contains duplicate state registers (PC, GPRs, Flags) but shares execution units (ALU, pipeline).
- The operating system sees **two logical CPUs** for a single core.
- If one logical thread is stalled (e.g., waiting for memory), the core executes instructions from the other thread, keeping the ALU busy.



CPU Performance: Clock Speed & Cycles

The CPU is driven by an oscillating **System Clock**:

- **Clock Period (T):** The duration of one clock cycle (e.g., 0.25 nanoseconds).
- **Clock Frequency (f):** Clock cycles per second, measured in Hertz (Hz).

$$f = \frac{1}{T}$$

- A 4.0 GHz CPU completes 4×10^9 clock cycles per second.
- Simple instructions might take 1 cycle; complex ones (like division or memory reads) take many cycles.

CPU Performance metrics: IPC and CPI

How many instructions does a CPU execute per cycle?

- **CPI (Cycles Per Instruction):** Average number of clock cycles required to execute one instruction.
- **IPC (Instructions Per Cycle):** The inverse of CPI.

$$\text{IPC} = \frac{1}{\text{CPI}}$$

- Modern superscalar processors execute multiple instructions in parallel, achieving $\text{IPC} > 1$ under ideal workloads.
- **The Execution Formula:**

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

✅ Checkpoint 4 — Pipelining & CPU Architectures

Let's test our understanding:

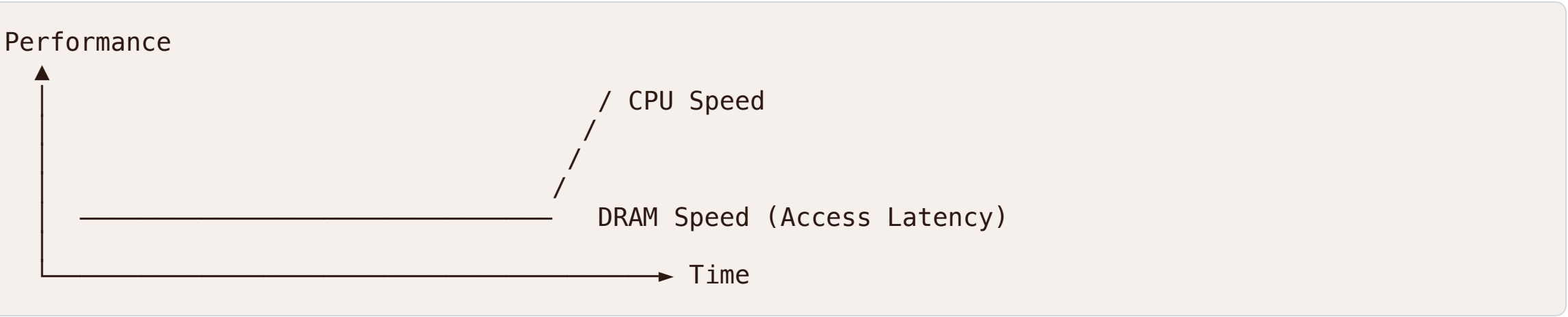
1. Sketch a diagram showing a 3-stage pipeline (Fetch, Decode, Execute) running three instructions sequentially.
2. Define structural, data, and control hazards with one example each.
3. How does hyper-threading differ from multi-core architecture?
4. If a program has $1,000$ instructions, runs with an average CPI of 1.5 on a $2\ \mathrm{GHz}$ CPU, how long does it take to execute?

Check your calculations before proceeding.

§ 3 — Memory Hierarchy & Cache

The Memory Wall: CPU vs RAM

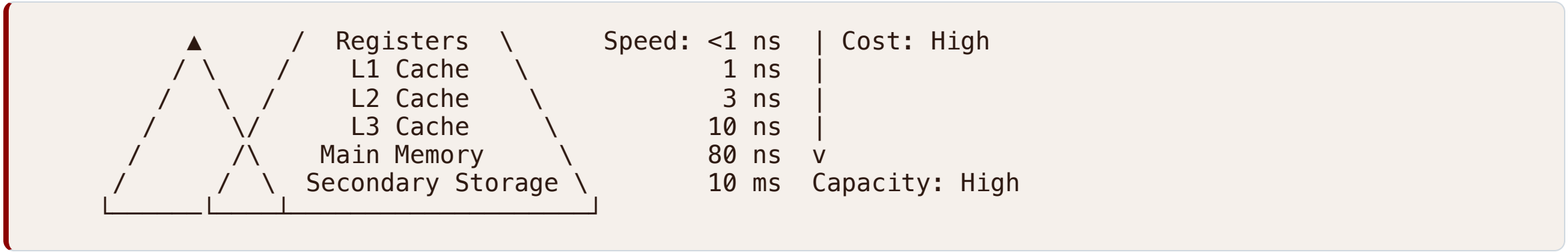
Processor speeds have grown exponentially over decades, but DRAM speeds have grown much slower.



This speed gap is called the **Memory Wall**. The CPU wastes performance if it idles waiting for memory access.

The Memory Hierarchy

To solve the memory wall, computer architects use a hierarchical structure:



- **Rule:** As we go down the hierarchy, access speed decreases, capacity increases, and cost per byte decreases.

Memory Hierarchy: Key Metrics

Comparison of typical modern memory components:

Level	Component	Typical Capacity	Access Latency
L0	CPU Registers	< 1 KiB	< 0.5 ns
L1	L1 Cache (per core)	32 - 64 KiB	1 - 2 ns
L2	L2 Cache (per core)	256 - 512 KiB	3 - 5 ns
L3	L3 Cache (shared)	8 - 64 MiB	10 - 20 ns
L4	Main Memory (DRAM)	8 - 64 GiB	60 - 100 ns
L5	SSD (NAND Flash)	256 GiB - 2 TiB	10 - 100 µs
L6	HDD (Magnetic Disk)	1 - 10 TiB	5 - 10 ms

RAM Technologies: SRAM vs. DRAM

Primary memory uses two main technologies:

SRAM (Static RAM)

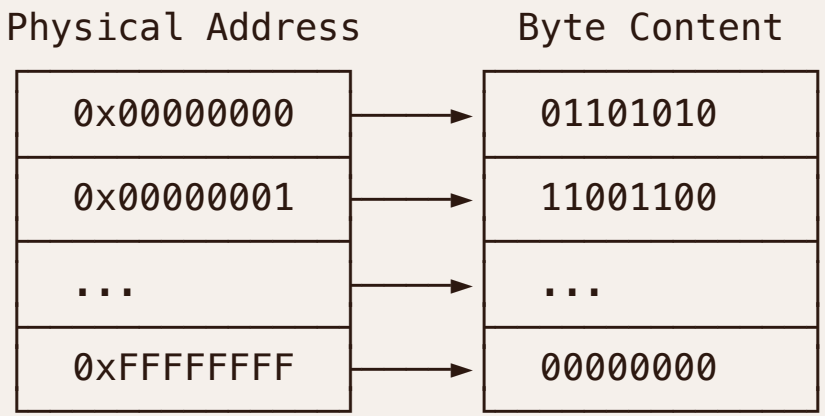
- Uses 6 transistors per cell to hold charge.
- Fast, low density, high power, expensive.
- Used for CPU caches (L1, L2, L3).

DRAM (Dynamic RAM)

- Uses 1 transistor and 1 capacitor per cell.
- Slow (capacitors need periodic refresh), high density, low power, cheap.
- Used for Main Memory (DRAM modules).

Physical Memory Organization

Main memory is structured as a 1D grid of byte addresses:



- **Byte-addressable:** Each unique address refers to exactly 1 byte (8 bits) of storage.
- **Word alignment:** Accessing multi-byte words (e.g., 4 or 8 bytes) is fastest when aligned to address boundaries that are multiples of word size.

The Principle of Locality

The memory hierarchy is highly effective because programs exhibit **Locality of Reference**:

- **Temporal Locality:** If a memory location is accessed once, it is highly likely to be accessed again in the near future.
- **Spatial Locality:** If a memory location is accessed, nearby memory locations are highly likely to be accessed in the near future.

CPUs exploit locality by copying data into high-speed caches.

Locality in Action: Temporal Locality

Consider this simple C code loop:

```
int sum = 0;
for (int i = 0; i < 1000; i++) {
    sum += array[i];
}
```

- **Temporal Locality:** The variables `sum` and `i` are accessed repeatedly in every iteration of the loop.
- **Hardware Action:** The CPU keeps `sum` and `i` in CPU registers or L1 cache, avoiding DRAM access entirely during the loop.

Locality in Action: Spatial Locality

Using the same code example:

```
int sum = 0;
for (int i = 0; i < 1000; i++) {
    sum += array[i];
}
```

- **Spatial Locality:** The elements of `array` are stored contiguously in the virtual address space. Accessing `array[i]` is immediately followed by accessing `array[i+1]`.
- **Hardware Action:** When `array[0]` is accessed and results in a cache miss, the CPU fetches an entire **cache line** (typically 64 bytes) from the memory hierarchy into the L1 cache. For example, if each element is 4 bytes (`int`), this cache line will contain up to 16 consecutive array elements (`array[0]` to `array[15]`), depending on alignment.

Cache Memory Architecture

Caches do not store individual bytes; they operate on fixed-size blocks called **Cache Lines** (typically 64 bytes).

A Cache Line entry contains:

- **Valid Bit:** Indicates if the cached data is valid.
- **Tag Bit:** Identifies the memory block's original address.
- **Data Block:** The actual 64 bytes copied from RAM.

Cache Performance: Hits & Misses

When the CPU requests memory address A:

Cache Hit

- The CPU finds address A in the cache.
- Serviced in 1–2 clock cycles.

Cache Miss

- The CPU does not find A in the cache.
- The CPU halts execution (stalls) while the memory controller fetches the cache line containing A from DRAM, incurring a ~100 cycle latency penalty.

Cache Miss Penalties

Let's calculate average memory access time (AMAT):

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

Assume:

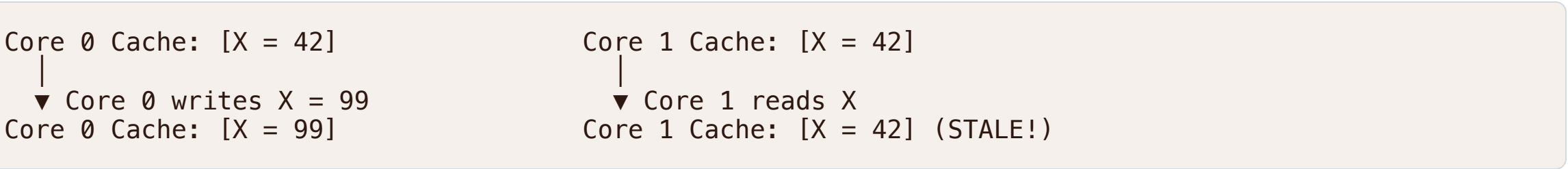
- L1 Hit Time = 1 cycle
- L1 Miss Penalty (fetch from RAM) = 100 cycles

Miss Rate	AMAT (Cycles)	Performance Impact
0% (All Hits)	1.0	Baseline
1%	2.0	2x slower!
5%	6.0	6x slower!
10%	11.0	11x slower!

Optimizing code for high cache hits is a primary goal of system software.

Cache Coherency

In multi-core systems, each core has its own private L1 cache. This introduces the **Cache Coherence Problem**:



- If Core 1 reads X from its cache, it gets the outdated value.
- **Hardware Solution:** Coherence protocols (e.g., MESI protocol) invalidate or update copy in other caches whenever a write occurs.

Cache Write Policies: Write-Through

When the CPU writes data to a cache block, how is RAM updated?

Write-Through Policy

- The CPU writes to both the Cache Line **and** Main Memory simultaneously.
- **Advantage:** Main memory is always up to date. Safe and simple.
- **Disadvantage:** Every write operation is limited by slow DRAM write speeds, negating cache benefits for writes.

Cache Write Policies: Write-Back

Write-Back Policy

- The CPU updates only the cache line.
- The cache line is marked as **Dirty** (modified).
- The actual write to Main Memory is deferred until the line is evicted from the cache to make room for new data.
- **Advantage:** Write operations complete at cache speed.
- **Disadvantage:** Memory can contain stale values, and eviction requires extra cycles to write dirty blocks to RAM.

Cache Line Mapping Policies

How does memory map to a specific location in cache? Three primary schemes:

1. **Direct Mapped:** Each memory block maps to exactly one cache slot.
2. **Fully Associative:** Any memory block can be stored in any cache slot.
3. **Set Associative:** Cache is divided into sets. A memory block maps to a set, but can reside in any slot within that set. (Used by most modern CPUs, e.g., 8-way set associative).

Cache Mapping: Direct Mapped

- Simplest design: cache index calculated using modulo arithmetic.

$$\text{Cache Slot} = \text{Block Address} \pmod{\text{Number of Cache Slots}}$$

- Conflict prone: if two memory blocks needed by a loop map to the same slot, they continuously evict each other (**Cache Thrashing**).

```
RAM Blocks:    0, 4, 8 ⟶ Maps to Cache Slot 0
RAM Blocks:    1, 5, 9 ⟶ Maps to Cache Slot 1
```

Cache Mapping: Fully Associative

- Ultimate flexibility: a block can go anywhere.
- No cache conflict thrashing.
- Requires parallel search hardware to check tags of every single cache line simultaneously.
- Very expensive and slow for large caches; used only in very small structures (like TLBs).

Cache Mapping: Set Associative

A compromise between Direct Mapped and Fully Associative.

- Cache is organized into N sets.
- A block maps to a set based on index bits.
- Inside the set, it can place in any of K ways (e.g., 4-way set associative).
- Reduces conflicts while maintaining reasonable search speeds.

✅ Checkpoint 5 — Memory Hierarchy & Cache

Let's test our understanding:

1. Explain the difference between SRAM and DRAM in terms of speed, density, and cost.
2. Why is L1 cache faster than L3 cache?
3. Define **Temporal Locality** and **Spatial Locality** and write a code snippet showing both.
4. If a CPU core writes to variable `x` in its private cache, how do other cores learn of this write?
5. Compare the performance trade-offs of Write-Through vs. Write-Back policies.

Consolidate your answers before proceeding.

§ 4 — Input/Output (I/O) & Bus Architecture

The Input/Output (I/O) System

A computer must communicate with external devices to be useful:

- **Storage Devices:** SSDs, HDDs, USB Flash.
- **Network Devices:** NICs (Ethernet, Wi-Fi).
- **User Interfaces:** Keyboard, Mouse, Display.
- **Sensors & Actuators:** Embedded hardware controllers.

All I/O transfers are managed by specialized controller hardware.

Device Controllers and Drivers

I/O devices are managed by two layers:

Device Controller (Hardware)

- A microchip on the motherboard or expansion card that directly controls the physical device (e.g., spin disk, adjust laser).
- It exposes a set of control registers to the CPU.

Device Driver (Software)

- A kernel-space OS module that understands how to talk to a specific device controller.
- Standardizes the interface so the OS kernel sees all disks or networks similarly.

Port-Mapped vs. Memory-Mapped I/O

How does the CPU read and write to device controller registers?

1. Port-Mapped I/O (PMIO)

- Dedicated CPU instructions (e.g., `in` and `out` on x86) and a separate I/O address space.

2. Memory-Mapped I/O (MMIO)

- Device registers are mapped directly into the physical memory address space.
- The CPU reads/writes device registers using normal pointer instructions (e.g., `*device_reg = value`). Most modern architectures use MMIO.

Inside a Device Controller: Registers

A controller typically exposes three classes of registers:

Register Type	Role	Example
Status Register	Read-only. Reports current state of the device.	"Device Busy", "Data Ready", "Error Detected"
Command Register	Write-only. Orders the device to do work.	"Start Read", "Send Packet", "Reset Device"
Data Register	Read/Write. Holds data being sent or received.	Keyboard key code, network packet byte

I/O Transfer Protocols

How does the CPU coordinate data transfer with an I/O device?

Three primary protocols:

1. **Programmed I/O (Polling)**
2. **Interrupt-Driven I/O**
3. **Direct Memory Access (DMA)**

Let's study the mechanics, advantages, and drawbacks of each.

1. Programmed I/O (Polling)

The simplest method: the CPU manually drives the transfer.

```
1. CPU sets command register to "READ"
2. Loop:
     Read Status Register
     If status is "BUSY" -> Go to 2 (Busy-Wait Loop)
3. Read data from Data Register
```

- The CPU actively checks the status register in a loop until the device completes.
- **Drawback:** The CPU is 100% occupied doing nothing during the wait, wasting cycles.

Polling in Pseudo-C

What polling looks like in code:

```
void write_byte_polling(char data) {
    // Wait until controller is ready to receive
    while (*DEVICE_STATUS_REG & STATUS_BUSY) {
        // Busy-waiting: wastes CPU cycles
    }
    // Write data to data register
    *DEVICE_DATA_REG = data;
    // Command the controller to start transmission
    *DEVICE_CMD_REG = CMD_TRANSMIT;
}
```

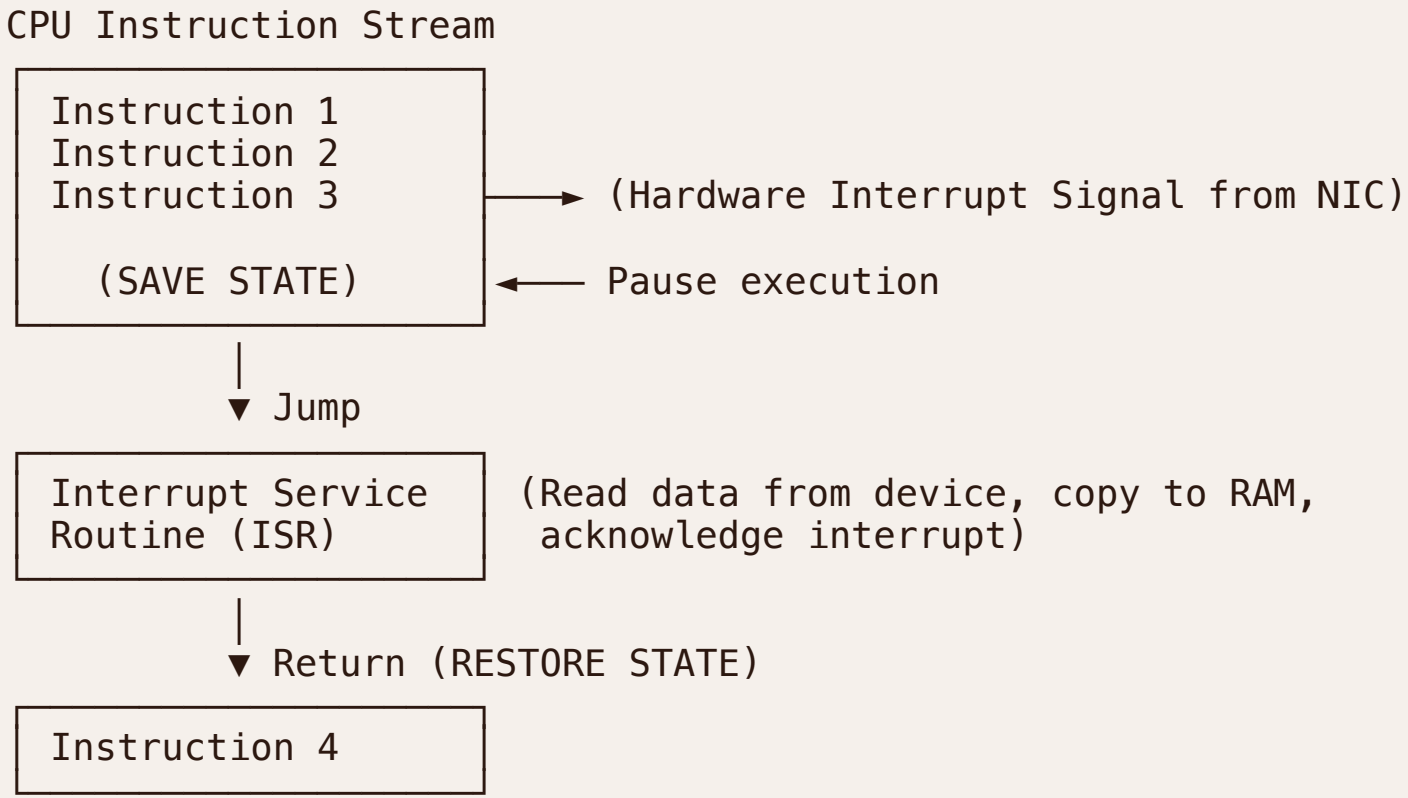
- Useful only in simple microcontrollers or when the device response time is extremely short.

2. Interrupt-Driven I/O

To avoid busy-waiting, devices can signal the CPU when they are ready.

- **Process:**
    i. The CPU initiates the device operation (e.g., read disk sector).
    ii. The CPU continues executing other programs.
    iii. When the device finishes, it pulls a hardware **Interrupt Line** active.
    iv. The CPU pauses current execution, jumps to the **Interrupt Service Routine (ISR)** in the kernel to handle the data, and then resumes the original program.

Hardware Interrupt Flow



The Interrupt Vector Table (IVT)

How does the CPU know where the ISR code is?

- The **Interrupt Vector Table (IVT)** or Interrupt Descriptor Table (IDT) is an array of function pointers stored in RAM.
- Each index corresponds to an interrupt line (Interrupt Vector).
- The table address is stored in a special CPU register (e.g., `IDTR` on x86).
- When interrupt N occurs, the CPU reads slot N of the IVT and jumps to the address stored there.

The Interrupt Service Routine (ISR)

An ISR (also called an Interrupt Handler) is a special function written by kernel developers.

- Must be extremely fast to prevent system lag.
- **Context Saving:** The CPU hardware saves status flags and registers before executing the ISR, and restores them on return.
- **Instruction:** Returns using a special instruction (e.g., `iret` on x86) which restores saved CPU flags and program counter simultaneously.

Overhead of Interrupts

While interrupts save CPU cycles compared to polling, they are not free:

- **Context Switch Cost:** Saving/restoring registers, flushing instruction caches.
- **Interrupt Storms:** If a high-speed device (like a 10Gbps Network card) interrupts the CPU for every incoming packet, the CPU will spend 100% of its time switching contexts, locking up the system.

Solution: High-speed devices combine interrupts with polling under high load, or use DMA.

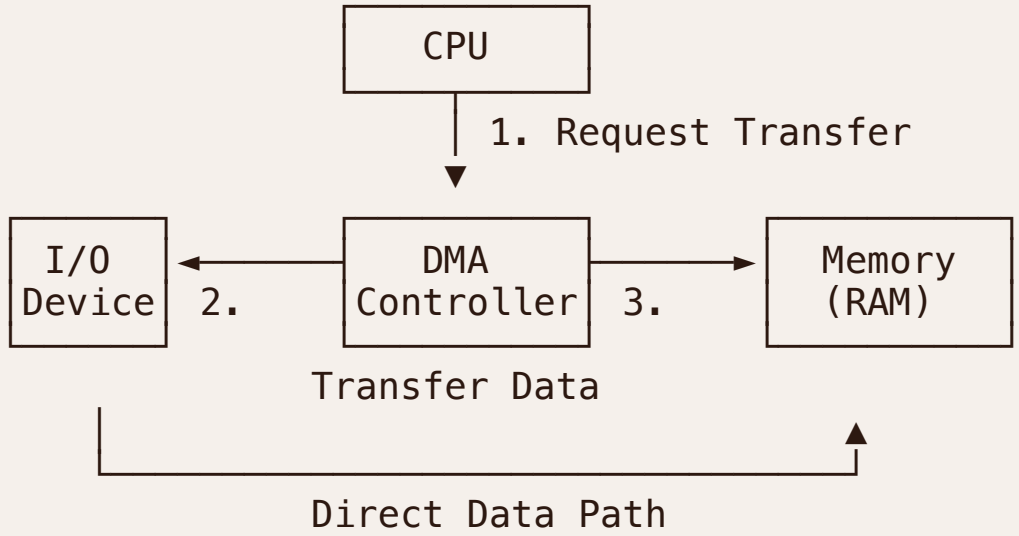
3. Direct Memory Access (DMA)

For high-speed, large-block transfers (e.g., disk reads, network packets), interrupts are too slow.

Direct Memory Access (DMA) delegates transfer to a dedicated hardware controller:

- The CPU requests the DMA controller: "Move 4 KiB from Disk to RAM address `0x7fff000`."
- The DMA controller drives the system bus, moving data directly from the device to RAM without CPU involvement.
- The CPU is completely free to execute other processes.
- The DMA controller generates a **single interrupt** when the entire 4 KiB block has been transferred.

DMA Transfer Flow Diagram



- When the transfer completes, the DMA Controller interrupts the CPU to notify completion.

Comparison of I/O Methods

Method	CPU Involvement	Overhead	Best for
Programmed I/O (Polling)	High (100% active wait)	Low entry, high execution	Very fast, simple devices
Interrupt-Driven I/O	Medium (handles context switch)	High entry overhead	Low-speed devices (keyboard)
Direct Memory Access (DMA)	Low (init & end only)	Medium setup overhead	High-speed block devices (disks, NICs)

Secondary Storage: Hard Disk Drives

The physical principles of HDDs introduce latency:

- **Components:** Platter, Spindle, Read/Write Head, Actuator Arm.
- **Latency Factors:**
    - **Seek Time:** Time to move the head to the target cylinder (5–10 ms).
    - **Rotational Latency:** Time for target sector to rotate under head.
    - **Transfer Time:** Time to read/write the bits from platter.
- **OS Role:** The OS scheduling algorithms optimize request queues to minimize disk head movement.

Secondary Storage: Solid State Drives

SSDs have no moving parts, using NAND Flash memory:

- Organised into **Pages** (e.g., 4 KiB) and **Blocks** (e.g., 128 pages).
- **Read/Write:** Can read and write at page granularity.
- **Erase:** Cannot overwrite in place. Must erase an entire Block before writing pages. (Write amplification).
- Much faster seek times than HDDs (< 0.1 ms), but wear out over time.
- **OS Role:** The OS supports TRIM commands to help the SSD controller manage empty blocks.

✅ Checkpoint 6 — I/O & Storage Systems

Let's test our understanding:

1. Contrast Memory-Mapped I/O (MMIO) with Port-Mapped I/O (PMIO).
2. Why is polling inefficient for slow I/O devices?
3. Walk through the step-by-step process of a keyboard key press reaching user software via interrupts.
4. What role does the Interrupt Vector Table play?
5. Why is DMA preferred over interrupt-driven I/O for reading files from a high-speed NVMe SSD?

Make sure you understand these mechanisms before wrapping up.

§ 5 — Summary & HW/SW Boundary

Summary: The Three Pillars

We have explored the physical foundation of computing:

1. **CPU:** The execution engine. Driven by the FDE instruction cycle, accelerated by pipelining, and structured with registers.
2. **Memory:** The data store. Organized hierarchically to balance cost and speed, relying on locality to keep the CPU fed.
3. **I/O:** The interface to reality. Managed by controllers and driven via polling, interrupts, or DMA.

How the OS Abstracts the Hardware

In the coming lectures, we will see how the OS virtualizes these physical resources:

Physical Resource	OS Abstraction	Primary Goal
CPU	Process / Thread	Virtualize execution time
Main Memory (DRAM)	Virtual Address Space	Provide private, safe memory
I/O Storage (HDD/SSD)	File System	Hide sector complexity behind files
Network Controllers	Sockets	Standardize data communication

Preview: Lecture 1.2 - Introduction to OS & Support

In the next lecture, we build the software interface on top of this architecture:

- **What is an Operating System?** Kernel, User space, System calls.
- **Dual-Mode Execution:** User Mode vs. Kernel Mode.
- **Context Switches:** Transitioning execution.
- **Interrupt Handling:** How the kernel processes hardware events.

> *Understanding the hardware is the key to understanding the OS. See you in Lecture 1.2!*

Lecture 1.1 Complete

Recommended Reading

- OSTEP: Chapter 1 (Introduction) and Chapter 2 (OS Overview)
- Reference materials on Computer Architecture

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it

- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!