CHAPTER TWO

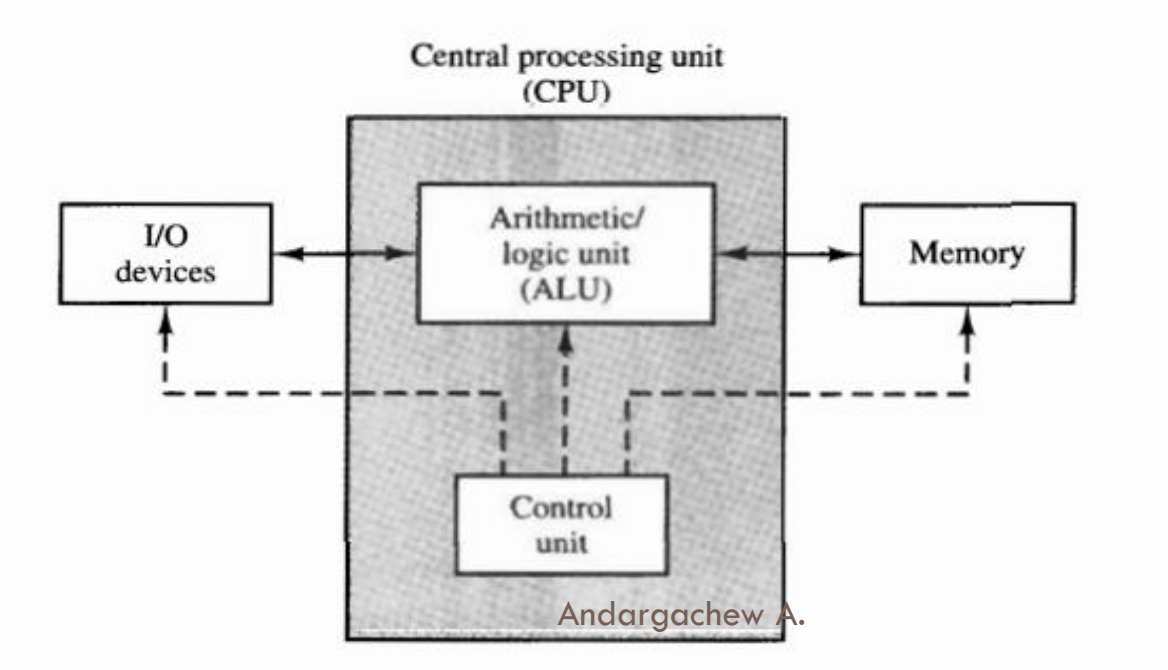
COMPUTER ORGANIZATION AND PROCESSOR ARCHITECTURE

CH-2 Contents

1. Overview of Computer System Components
2. Instruction Set Architecture (ISA)
3. CPU Datapath and Control Unit
4. Performance Concepts and Metrics
5. Performance Optimization Techniques

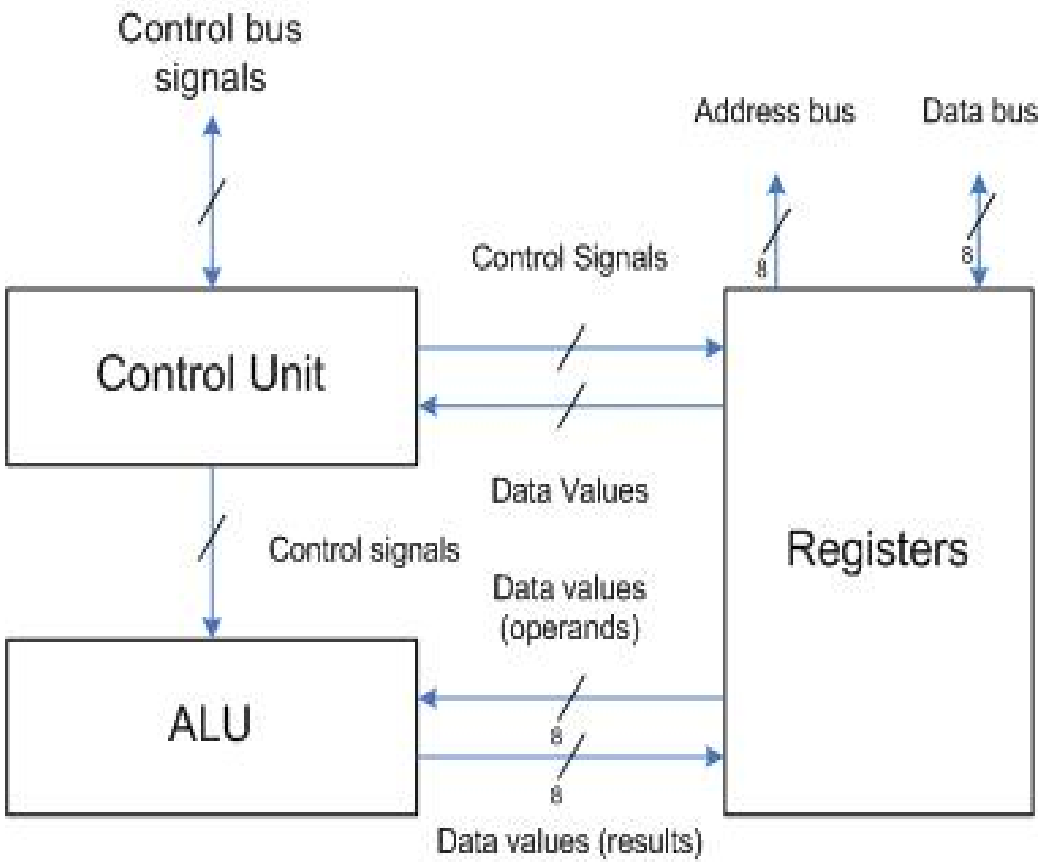
Overview of Computer System Components

- A computer system consists of interconnected components: **CPU, memory,** and **I/O subsystems -** that **communicate via the system bus** to process data and run programs efficiently.



CPU Components Overview

- The part of the computer that performs the bulk of data processing operations is called the **Central Processing Unit(CPU)**.
- It is made up of three major parts
  1. **ALU (Arithmetic Logic Unit)**: Performs calculations and logical comparisons.
  2. **CU (Control Unit)**: Directs operation of processor; interprets instructions.
  3. **Registers**: High-speed storage for temporary data and instructions.


Control bus signals
Address bus
Data bus
Control Signals
8
8
Control Unit
Data Values
Control signals
Registers
Data values (operands)
ALU
8
8
Data values (results)

Register Types and Functions

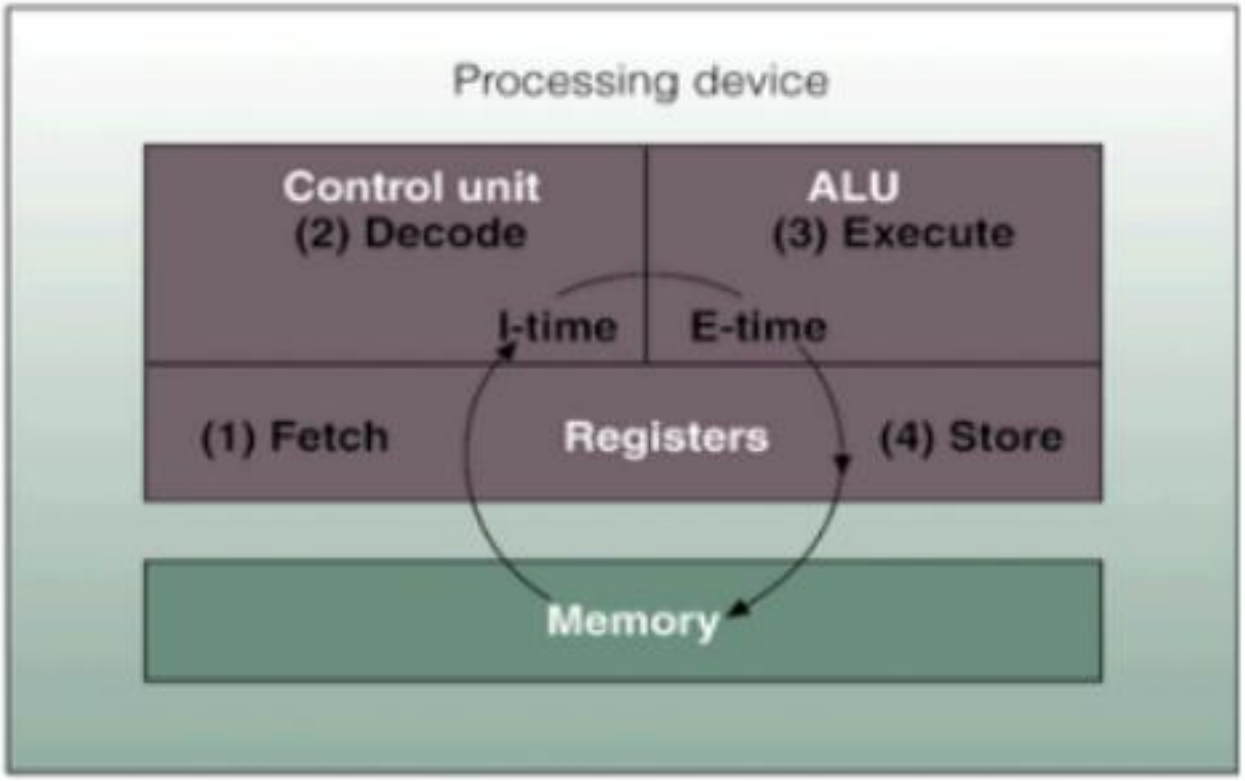
- Registers are **small, high-speed** storage elements **located inside the CPU.**

Register	Name	Function
PC	Program Counter	Holds address of next instruction
IR	Instruction Register	Stores current instruction
MAR	Memory Address Register	Holds address for memory access
MDR	Memory Data Register	Holds data read/written to memory
ACC	Accumulator	Stores intermediate arithmetic results
SP	Stack Pointer	Points to top of stack

Instruction Cycle

- Every program you run goes through a repeating sequence of steps known as the **instruction cycle.**
- Each instruction cycle in turn is subdivided into a sequence of sub-cycles or phases.

1. **Fetch** an instruction from memory
2. **Decode** the instruction
3. **Execute** the instruction
4. **Store** the result


Processing device
Control unit
(2) Decode
ALU
(3) Execute
I-time
E-time
(1) Fetch
Registers
(4) Store
Memory

Example : Control Flow

- To understand how the instruction cycle works in practice, let's consider a simple example:
- **Instruction:** ADD R1, R2 → R3
- **Steps:**
  1. **Fetch** the instruction from memory.
  2. **Decode:** CU identifies the operation as ADD.
  3. **Fetch operands** (R1, R2) from registers.
  4. **ALU performs the addition** and **stores the result** in R3.
  5. **Update the PC** for the **next instruction.**

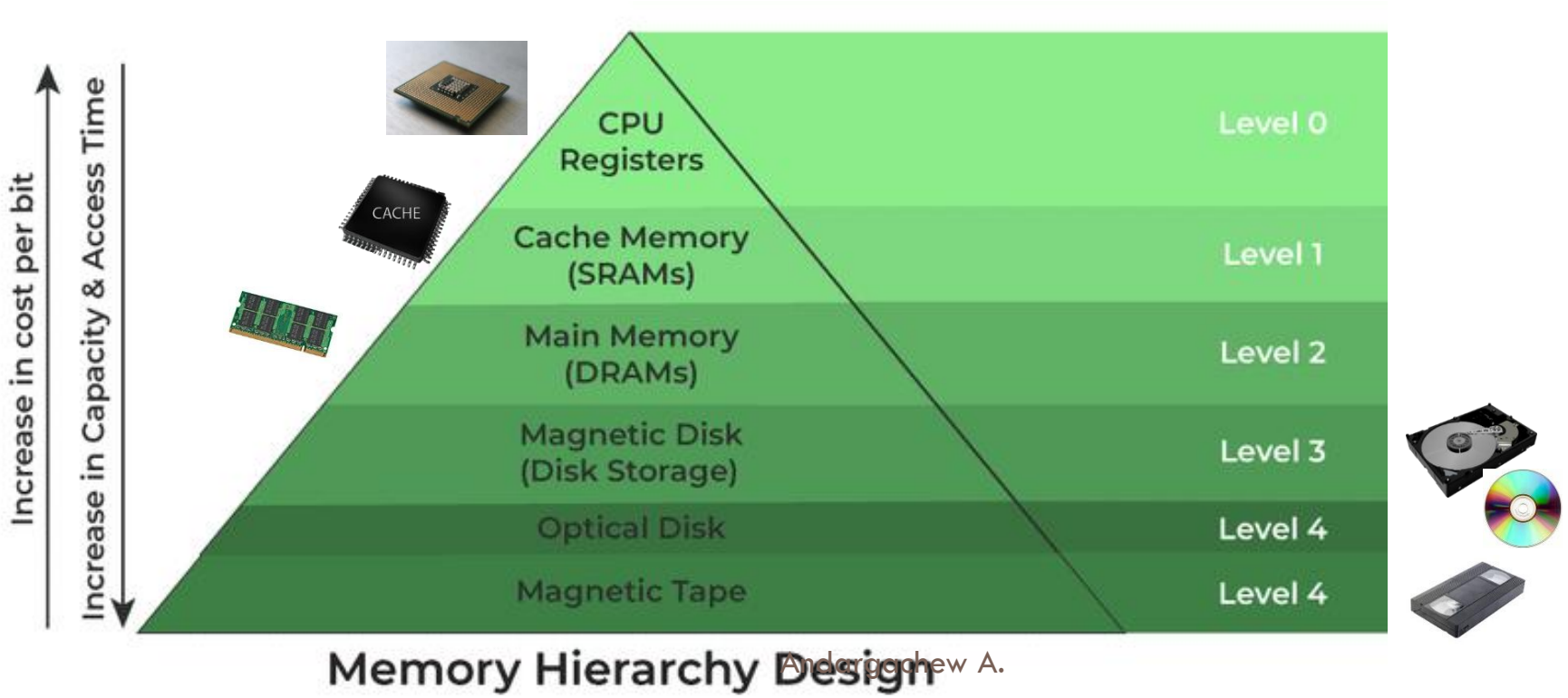
Computer Memory Overview

- Memory works like a human brain - it stores **data** and **instructions**.
  - It is the **storage space in a computer** for data processing and required instructions.
- Memory is divided into many small parts called **cells**.
  - Each cell has a **unique address,** ranging from **0** to **size – 1**.

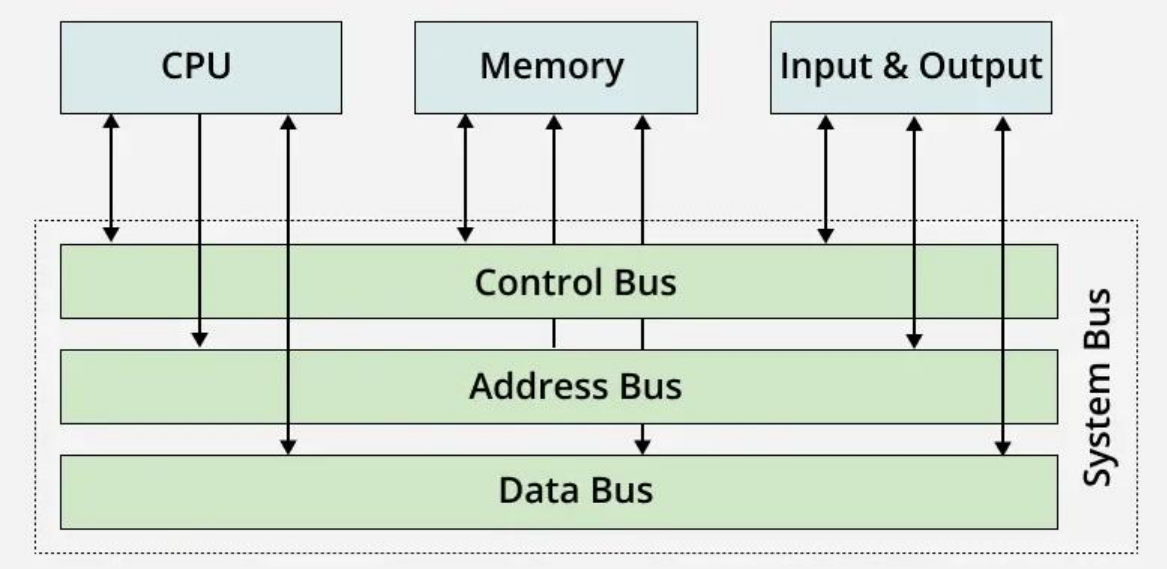
- **Example**
  - 64K-word memory = 64 × 1024 = **65,536 locations**
  - Addresses range from **0 to 65,535**

- **Addressing Bits**
  - The **number of bits required** to address all cells depends on total locations.
  - For **65,536 cells → 16 bits** are required to represent all addresses.

- **Memory is primarily of two types**
  - **Internal Memory** – registers, cache memory and primary/main memory
  - **External Memory** - magnetic disk, optical disk, magnetic tape etc.

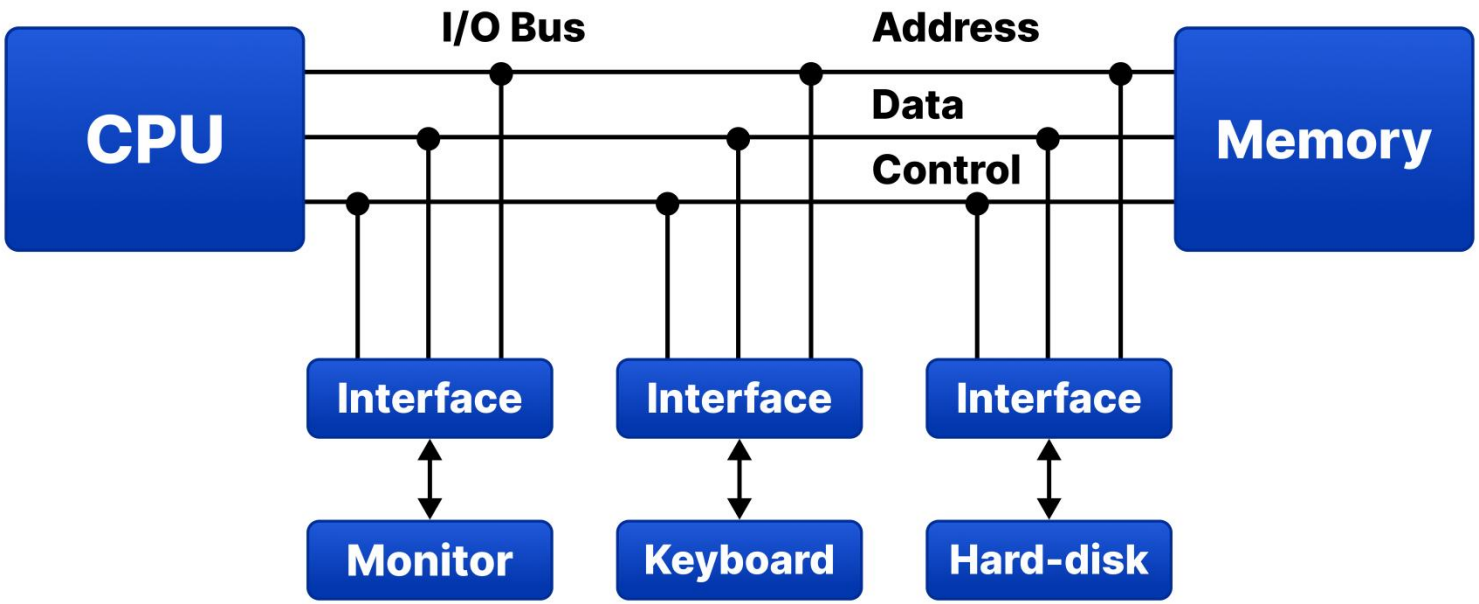


- **CPU–Memory Interaction**
  - CPU constantly exchanges **data & instructions** with **main memory.**
  - Communication happens via **system buses:**
    - **Address Bus** – carries memory addresses (via **MAR**)
    - **Data Bus** – transfers data (via **MDR**)
    - **Control Bus** – handles **Read/Write** signals



- **Peripheral Devices & I/O System**
  - **I/O Subsystem:** enables communication between the CPU and external environment.
  - **Peripheral Devices:** input/output units like:
    - **Input**: Keyboard
    - **Output**: Monitor (CRT, LCD)
      - **Hardcopy**: Printer (Dot Matrix, Inkjet, Laser)
  - ASCII standard code for alphanumeric data; **commonly used in I/O transfers.**

- **I/O Interface**
  - **Special communication links** for interfacing peripherals to CPU
  - The purpose of the communication link is to **resolve the differences that exist between the central computer and each peripheral.**
    1. Signal type
    2. Data transfer rate
    3. Data format
    4. Operating mode


I/O Bus
Address
Data
Control
CPU
Memory
Interface
Interface
Interface
Monitor
Keyboard
Hard-disk

Instruction Set Architecture (ISA)

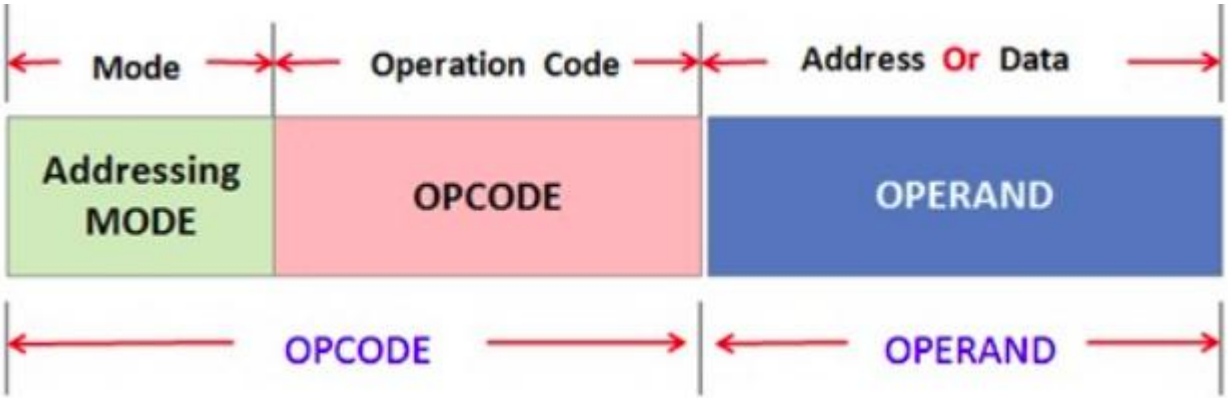
- ISA defines the **boundary** between **hardware** and **software.**
  - It serves as a **contract,** allowing **software to communicate with the CPU** solely through **predefined instructions.**
    - These instructions**, Instruction Code** (binary codes), tell the processor **what operations to perform and on which data.**
- By specifying **supported instructions, data types,** and **memory access methods:**
  - ISA **ensures compatibility between programs and processors.**
- **Understanding ISA** reveals **how high-level code** is **translated** into **machine-level execution.**

ISA Components: Instruction Code Fields

- An ISA defines how instructions are **structured** and **executed.**
  - **Operation (Opcode):** the action to perform
    - Example: ADD, LOAD, JUMP
  - **Operands:** Data or addresses used in operations
    - **Types**: Immediate, Register, Memory
  - **Addressing Modes:** How CPU finds operands
    - **Examples**: Immediate, Direct, Indirect, Indexed
- The overall layout of **opcode** + **addressing mode** + **operands: Instruction Format**
  - **Data Types:** size/format of data (byte, word, float, etc)

Instruction Example: Operation Code and Operand

- The **opcode** tells the CPU ***what* to do,** while the **operand** field tells it *on* ***what data* to perform the operation.**
- **Example:**
  - **Instruction: 1100 0101**
    - → Opcode = ADD
    - → Operand = Register *5*
  - **Opcode length (n bits) supports up to 2^n operations.**



- **Number of Operands and Instruction Format**
  - Operands represent the **data, addresses,** or **registers** involved in an operation.
    - The number of operands determines **the structure of the instruction** and **how the CPU fetches data.**
  - Different architectures support different operand counts, which influences **instruction length, complexity,** and **flexibility.**
    - **0, 1, 2, or 3-operand architectures.**
    - Example
      - **0: PUSH, POP (stack)**
      - **1: LOAD A (accumulator)**
      - **2: ADD A, B**
      - **3: ADD R1, R2, R3 (RISC style)**

Instruction Code Formats

- Now that we understand instruction operands, let's explore **how they're arranged with opcodes in the instruction word.**
  - Different architectures organize instructions in distinct ways, influencing performance and complexity.
- **Common formats include:**
  - **Register–Memory** (e.g., x86): one operand in a register, the other in memory
  - **Register–Register** (e.g., ARM, RISC-V): both operands in registers
  - **Immediate Addressing:** operand value is embedded in the instruction
- These formats impact **instruction length, decoding difficulty,** and **execution speed.**

Instruction Decoding: From Instruction Code to Micro-operations

- When the CPU decodes an instruction, it interpret ***the instruction code (binary form)*** according to the ISA's specification.
- **Example: 8 bits CPU**
  - Instruction code: **0110 01 10**
    - **Opcode (4 bits) | Operand-1 (2 bits) | Operand-2 (2 bits)**
  - **According to the ISA definition:**
    - 0: **Register** Addressing mode 110 = **ADD** operation
    - 01 = **Register 1** and 10 = **Register 2**

      → So this instruction means **ADD contents of R1 and R2**

- **Each field** (opcode, operand, addressing bits) **triggers specific micro-operations:**
  - **Opcode bits → Select ALU function** (add, subtract, etc.)
  - **Operand bits → Select source and destination registers**
  - **Addressing bits → Determine how to fetch data**
- So, the **instruction code** tells the hardware *exactly which* ***micro-operations to execute*** and in what order.

Example: ADD Instruction in x86 vs. ARM

- Different processor architectures (x86, ARM, RISC-V) implement similar operations using unique instruction formats.

Architecture	Assembly Instruction	Description
x86	ADD AX, BX	Add contents of BX to AX (**CISC - complex**)
ARM	ADD R1, R2, R3	Add R2 and R3 → store in R1 (**RISC - simple**)

- **Instruction Set Design Philosophies**
  - Two major approaches to ISA design, **RISC** and **CISC**, differ in instruction complexity and execution strategy
    - **RISC (Reduced Instruction Set Computer)**
      - Small, simple instruction set
      - Each instruction executes quickly
      - Emphasizes hardware simplicity
    - **CISC (Complex Instruction Set Computer)**
      - Large, versatile instruction set
      - Single instructions can perform complex tasks
      - Emphasizes software compactness
  - **Design Goal:**
    - Balance simplicity (RISC) vs. flexibility (CISC)

Aspect	RISC	CISC
Instruction Length	Fixed	Variable
Instruction Count	Few (≈ 30–100)	Many (hundreds)
Execution Speed	1 cycle/instruction	Multiple cycles/instruction
Addressing Modes	Few	Many
Hardware Complexity	Simple	Complex
Code Size	Larger	Smaller
Control Unit	Hardwired	Microprogrammed
Example CPU	ARM, RISC-V	Intel x86

- Modern CPUs (like ARM and x86) **combine both philosophies in practice.**

- **Types of Instructions**
  - **Data Movement Instructions:** transfer data between memory and registers
    - Examples: MOV, LOAD, STORE
  - **Arithmetic/Logic Instructions:** perform mathematical or logical operations
    - Examples: ADD, SUB, AND, OR
  - **Control Instructions:** alter the flow of execution
    - Examples: JMP, CALL, RET, BEQ (branch if equal)
  - **Input/Output Instructions:** handle communication with devices
    - Examples: IN, OUT

Exercise 1

- **Decoding Instruction**
  - **Define a Simple CPU Model**
    - Let's imagine a **4-bit CPU** with:
      - **4 general-purpose registers (R0–R3)**
      - **4 memory locations (M0–M4)**
      - **Addressing Modes (2 bits):**
    - **8-bit instruction codes (2 bytes = 8 bits per instruction)**
  - We'll use an **8-bit instruction format** for simplicity:
    - **| Mode (2 bit) - Opcode (2 bits) | Operand (4 bits) |**

- **Instruction Set Definition**

Mnemonic	Opcode (binary)	Operation Description
MOV	00	Move data
ADD	01	Add
SUB	10	Subtract
AND	11	Logical AND

- **Addressing Modes**

Mode Bits	Type	Example	Description
00	Reg–Reg	ADD R1, R2	Add two registers
01	Reg–Mem	ADD R1, M3	Add memory to register
10	Reg–Imm	ADD R1, #2	Add constant to register
11	Mem–Reg	ADD M2, R3	Add register to memory

Example-1

Instruction: 1001 1010

Field	Bits	Meaning
Mode	10	Register - Immediate
Opcode	01	ADD
Destination	10	R2
Source	10	Immediate value #2

Decode Instruction: ADD R2, #2

- → *Add constant 2 to R2.*

Example-2

Instruction: 0111 0011

Field	Bits	Meaning
Mode	01	Register - Memory
Opcode	11	AND
Destination	00	R0
Source	11	Memory address M3

Decode Instruction: AND R0, M3

- → *ANDing the data at memory location M3 to R0.*

- **Decode the following Machine Instruction**
  1. **0110 0011**
  2. **1100 0101**
  3. **0001 1111**

CPU Datapath and Control Unit

- The **datapath** and **control unit** are the two core subsystems of a CPU.
  - The datapath performs the actual operations on data, while the control unit directs when and how those operations occur.
  - Together, they make the processor function like a coordinated machine.
    1. **Datapath:** The hardware that moves and processes data (ALU, registers, buses).
    2. **Control Unit:** The "brain" that manages timing and coordination.
    - The CPU works by fetching, decoding, and executing instructions — all controlled here.

- **Components of the Datapath**
  - The datapath includes all hardware elements that perform computation and data transfer. It supports the execution of micro-operations defined by the control signals.
  - **Main Components:**
    - **Registers:** Temporary, fast data storage for operands and results.
    - **ALU (Arithmetic Logic Unit):** Performs arithmetic and logic operations.
    - **Buses:** Transfer data among registers, memory, and ALU.
    - **Multiplexers:** Select which data paths are active at a given time.

The Control Unit: Hardwired and Microprogrammed

- The **control unit (CU)** generates control signals that tell each part of the datapath what to do at every clock cycle.
- There are two main types:
  1. **Hardwired Control Unit:**
     - Uses logic circuits to generate control signals.
     - Fast but less flexible.
  2. **Microprogrammed Control Unit:**
     - Uses a small memory (control store) with microinstructions.
     - Easier to modify but slightly slower.
- **Function:**
  - Decode instruction → Generate control signals → Sequence micro-operations.

Example: Datapath and Control in Action

- Let's trace how the **datapath** and **control unit** cooperate during a simple instruction: **ADD R1, R2 → R3.**
- **Sequence of Micro-operations:**
  1. **Fetch:**
     - PC → MAR; Memory → MDR → IR
  2. **Decode:**
     - CU interprets opcode (ADD) and operands (R1, R2, R3).
  3. **Execute:**
     - CU activates control signals:
       - R1 → ALU Input A
       - R2 → ALU Input B
       - ALU → R3 (store result)

Performance Concepts and Metrics

- In computing, **performance** refers to **how fast and efficiently a computer system executes tasks** and **processes data.**
  - A high-performing system **completes tasks quickly, using minimal time** and **resources.**
- **Factors affecting performance include:**
  - **Processor speed** (Clock rate, CPI)
  - **Memory system** (latency, cache)
  - **Storage and I/O devices**
  - **Software optimization** (compiler and algorithm efficiency)

- To measure performance, we use several key metrics (**High-Level Performance Metrics**):
  1. **Response Time**: Duration from task start to completion, including OS overhead, I/O wait, disk/memory access, and CPU execution.
  2. **Throughput**: Total work completed per unit time.
  3. **CPU Time**: Time CPU spends computing a task, excluding I/O and other program delays.
     - **Response time** and **throughput** describe **system-level performance**, while **CPU execution** time focuses on the **processor's own efficiency**.

- **Low-Level Performance Metrics**
  - **Clock Cycle (T):**
    - Time for one CPU clock tick (seconds per cycle).
    - Example: 2 GHz → 0.5 ns per cycle.
  - **Clock Rate (f):**
    - Number of cycles per second (Hz).
  - **Instruction Count (IC):**
    - Total number of instructions executed by a program.
  - **Cycles Per Instruction(CPI):**
    - Average number of cycles per executed instruction.

- **Execution Time and Performance Relationship**
  - Performance is inversely proportional to execution time:
    - **Performance = (1 / Execution time)**
  - **Example:**
    - Machine A runs a program in 100 seconds, Machine B runs it in 125 seconds
      - (Performance of A / Performance of B) = (Execution Time of B / Execution Time of A)
      - = 125 / 100 = 1.25
    - That means **machine A is 1.25 times faster than Machine B.**

- And, the time to execute a given program can be computed as:
  - **Execution time = CPU clock cycles x clock cycle time**
    - **CPU clock cycles**
      - (No. of instructions / Program) x (Clock cycles / Instruction)
      - Instruction Count x CPI (Clock per Instruction)
  - **Execution time = IC x CPI x T_{cycle}**
    = (IC x CPI) / f

Performance Optimization Techniques

- Modern processors achieve high performance not just by **increasing clock speed,** but through **architectural techniques** that allow multiple operations to occur simultaneously.
- **Main strategies:**
  - Pipelining
  - Instruction-Level Parallelism (ILP)
  - Multithreading & Multicore
  - Cache Memory Hierarchy
  - GPU-based Parallelism

- **Pipelining**
  - Improves CPU throughput by overlapping instruction execution (like an assembly line).
    - **Pipeline stages**: Fetch → Decode → Execute → Memory → Write Back
    - Ideal goal: one instruction per clock cycle.
  - **Hazards & Solutions**
    - **Structural**: Resource conflict → use hardware duplication or scheduling.
    - **Data**: Instruction depends on another → apply forwarding/bypassing or insert stalls.
    - **Control**: Branch changes flow → use branch prediction or flush pipeline.

- **Instruction-Level Parallelism (ILP)**
  - ILP enables parallel execution of independent instructions.
    - It identifies instructions that don't depend on each other and schedules them to run simultaneously.
  - **Superscalar Architecture** is a hardware implementation of ILP
    - it uses multiple execution units.
    - Can issue 2–4 instructions per clock cycle.
    - Supports:
      - Dynamic scheduling
      - Out-of-order execution
  - **Goal**: Increase throughput beyond one instruction per cycle.
  - **Limitation**: Data and control dependencies restrict how much parallelism is achievable.

- **Multithreading and Multicore Processors**
  - As single-core performance hit physical limits, CPUs began executing **multiple threads** or **multiple cores** in parallel.
  - This allows tasks to run concurrently, improving total system throughput.
    - **Multithreading:**
      - Multiple threads share one CPU core.
      - Example: Simultaneous Multithreading (SMT) in Intel Hyper-Threading.
    - **Multicore:**
      - Multiple processing cores on one chip.
      - Each core executes separate threads.
  - **Advantages:**
    - Better parallelism for multitasking and modern applications.
    - Energy-efficient performance scaling.

- **Cache Memory Hierarchy**
  - A small, fast memory located close to the CPU.
  - Stores frequently accessed data to reduce latency.
  - **Cache Levels**
    - **L1 Cache:**
      - Smallest and fastest
      - Split into instruction and data caches
      - Closest to CPU core
    - **L2 Cache:**
      - Larger than L1
      - Slower but still fast
      - Shared by one or few cores
    - **L3 Cache:**
      - Largest and slowest among caches
      - Shared across all cores
        - Reduces access to main memory

- **GPU-based Parallelism**
  - A Graphics Processing Unit (GPU) designed for **parallel processing**.
    - Originally built for rendering graphics, now widely used in AI, scientific computing, and more.
  - **Key Features**
    - **Thousands of cores**: Handle many tasks simultaneously.
    - **SIMD model:** Executes the same instruction across multiple data points
    - **High memory bandwidth:** Optimized for large data throughput.
    - **Specialized units:** Texture mapping, shading, matrix operations.

- **CPU vs GPU**
  - Both CPUs and GPUs are processors, but they are optimized for different performance goals.
    - **CPU** → Designed for low latency and general-purpose sequential processing.
    - **GPU** → Designed for high throughput and massive parallel data processing.

Aspect	CPU (Central Processing Unit)	GPU (Graphics Processing Unit)
Design Goal	Minimize latency — fast single-task completion	Maximize throughput — many tasks in parallel
Cores	Few, complex, high clock speed	Thousands, simple, lower clock speed
Execution	Sequential / few threads	Massively parallel (SIMD/SIMT)
Control Logic	Large, handles complex instructions	Minimal, focuses on repetitive workloads
Memory Hierarchy	Large multi-level cache (L1–L3)	High-bandwidth shared memory
Best For	OS, logic-heavy tasks, control flow	AI, graphics, simulations, data-parallel jobs
Performance Focus	Low latency	High throughput

Performance Techniques and Their Impact on Metrics

Technique	Optimizes	Effect on Performance Metrics
Pipelining	Instruction Overlap	↓ CPI, ↑ Throughput
ILP / Superscalar	Multiple instruction issue	↓ CPI, ↑ Throughput
Multithreading	Thread-level parallelism	↑ Throughput, hides latency
Multicore	Parallel execution	↑ Throughput
Cache Hierarchy	Fast data access	↓ Memory latency, ↓ CPI
GPU	Massive data parallelism	↑ Throughput (high-latency tolerance)

Questions?

Thank You