COMPUTER ORGANIZATION AND ARCHITECTURE

Introduction to Computer Organization and Architecture

Computer Architecture:

Computer architecture refers to the *design* and *structure* of a computer system. It defines the system's functionality, which includes the instruction set, data types, memory addressing modes, and I/O mechanisms. In essence, it is the conceptual framework that determines how the system will operate at a high level.

Key components:

- **Instruction Set Architecture (ISA)**: Defines the machine's instructions and how they are executed.
- **Data Types**: Specifies the types of data (e.g., integers, floating-point numbers) the system can process.
- **Memory Architecture**: Defines the layout and the organization of memory (e.g., RAM, Cache).
- **I/O Architecture**: Defines how the system interacts with peripheral devices.

Examples of computer architectures include CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing).

Computer Organization:

Computer organization refers to the *physical realization* of the computer architecture. It focuses on how the system components (hardware) are implemented and connected to achieve the desired functionality specified by the architecture.

Key components:

- **Control Unit**: Directs operations within the CPU and coordinates the execution of instructions.
- **Arithmetic and Logic Unit (ALU)**: Performs arithmetic and logical operations.
- **Registers**: Small, fast storage locations that store intermediate data during processing.
- **Memory System**: Includes RAM, cache memory, and sometimes virtual memory.

In short:

- **Architecture** = What the system is capable of doing.
- **Organization** = How it does what it is capable of.

Aspect	Computer Architecture	Computer Organization
Definition	Refers to the *design and specification* of the system's functionality.	Refers to the *implementation* and arrangement of hardware components.
Focus	High-level design and capabilities (what the system can do).	Low-level design and physical realization (how the system is built).
Scope	Includes instruction set, data types, memory addressing, and I/O mechanisms.	Includes the physical components like CPU, ALU, registers, buses, and memory.
Concerned With	Defines the *interface* between hardware and software.	Deals with the *actual physical hardware* and how it implements the architecture.
Key Components	Instruction Set Architecture (ISA), Addressing Modes, Data Types.	Control Unit, ALU, Registers, Memory, Bus System.
Purpose	To specify the *functional requirements* of the system.	To determine *how the specified functionality is physically realized*.
Example	RISC or CISC architecture, 32-bit or 64-bit architecture.	How the CPU handles instructions, how many bits are processed in each operation, etc.
Changes	Changes in architecture typically require new software or programs.	Changes in organization affect the hardware and implementation details.

1. Von Neumann Architecture

🔷 **Definition:**

Von Neumann Architecture is a computer architecture model proposed by **John von Neumann in 1945**, where **data and instructions (programs)** are stored in the **same memory unit** and processed by the **same set of hardware**.

🧩 **Basic Components:**

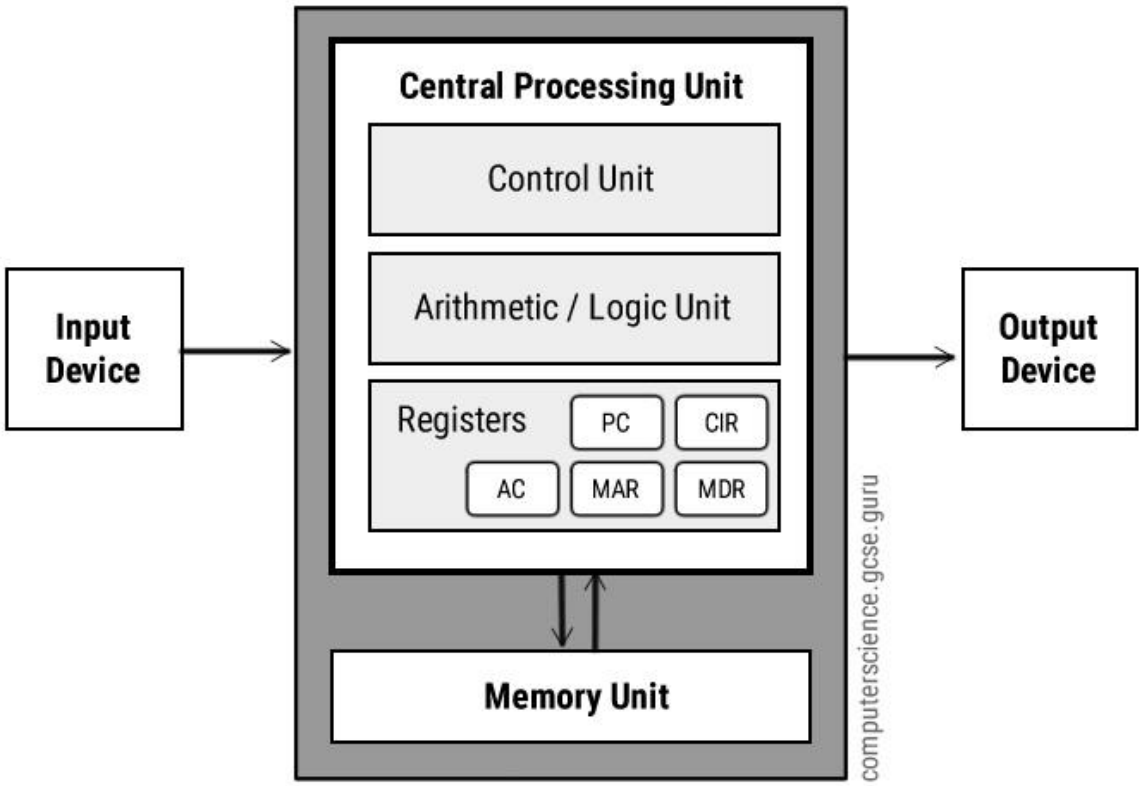
1. **Memory Unit**
    - Stores both **data** and **instructions**.
    - Example: RAM.
2. **Arithmetic and Logic Unit (ALU)**
    - Performs mathematical operations like addition, subtraction.
    - Also handles logical comparisons.

3. **Control Unit (CU)**
    - Directs the flow of data and instructions.
    - Fetches instructions from memory and decodes them.
4. **Input/Output Devices (I/O)**
    - Used to communicate with the computer.
    - Example: Keyboard (input), Monitor (output).
5. **Registers**
    - Small, fast memory units inside the CPU.
    - Temporarily hold data, instructions, addresses, etc.

📋 **Working Cycle: The Von Neumann Cycle**

1. **Fetch**: CU fetches instruction from memory.
2. **Decode**: CU decodes the instruction.
3. **Execute**: ALU or another part executes the instruction.
4. **Store**: Result may be written back to memory or a register.

This is often referred to as the **Fetch-Decode-Execute Cycle**.



✅ **Advantages:**

- **Simpler Design**: One memory for both data and instructions.
- **Low Cost**: Fewer components.

❌ **Disadvantages (Von Neumann Bottleneck):**

- Only **one operation at a time** can be performed (either fetch instruction or fetch data).
- Limits speed → known as the **Von Neumann Bottleneck**.

Bus Structures

A **bus** is a communication pathway that transfers data, instructions, and control signals between various components of a computer system. It connects the CPU, memory, and peripheral devices, allowing data to be transferred across different parts of the system.

Types of Buses:

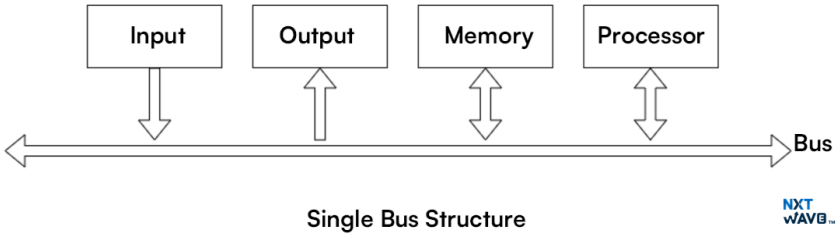
1. **Data Bus**:
    - **Purpose**: Transfers the actual data between the CPU, memory, and I/O devices.
    - **Function**: Data buses are bidirectional (data can be transferred in both directions), and they carry data to and from memory or I/O devices.
2. **Address Bus**:
    - **Purpose**: Carries the address of the memory location or I/O port where data needs to be read or written.
    - **Function**: The address bus is **unidirectional**, meaning it only carries addresses from the CPU to memory or I/O devices.
3. **Control Bus**:
    - **Purpose**: Carries control signals that coordinate the actions of various components of the computer system.
    - **Function**: The control bus is used to manage the timing and operation of the data and address buses and to ensure that the correct operations are performed at the correct time.

🔹 **Single-Bus Organization**

🧠 **Definition:**

All major components (CPU, memory, I/O devices) **share a single common bus** for data transfer.

📋 **Example Structure:**



Single Bus Structure

✅ **Advantages:**

- **Simple design**
- **Lower cost**
- Easy to implement

❌ **Disadvantages:**

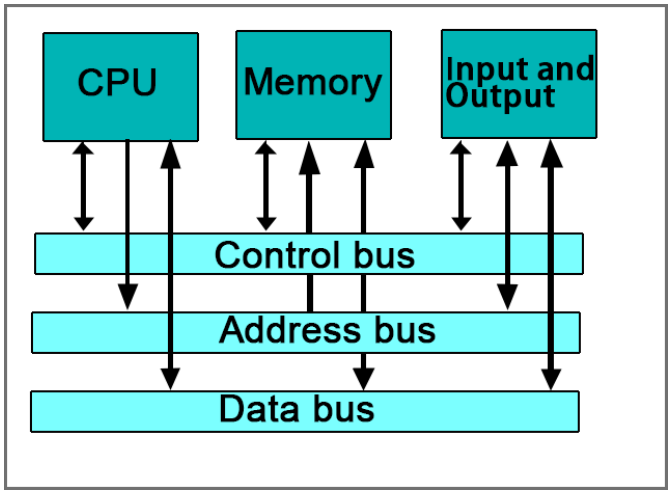
- Only **one operation at a time** can occur on the bus.
- Causes **bottleneck** (traffic jam on the bus).
- Slower performance as system scales.

🔹 **Multi-Bus Organization**

🧠 **Definition:**

System uses **multiple buses** to allow **parallel data transfers** between components, improving speed and efficiency.

📋 **Example Structure:**



✅ **Advantages:**

- **Faster**: Multiple data transfers can happen simultaneously.
- Reduces bottlenecks.
- Better **system performance**.

❌ **Disadvantages:**

- **Complex design**
- **Higher cost**
- Requires more control logic

Aspect	Single Bus System	Multi-Bus System
Cost	Lower cost due to fewer buses.	Higher cost due to additional buses.
Design Complexity	Simpler design with fewer components.	More complex design and control logic.
Performance	Lower performance due to shared bandwidth.	Higher performance with parallel data transfer.
Scalability	Less scalable, performance degrades with more components.	More scalable, performance is maintained even with more devices.
Data Transfer Speed	Slower data transfer due to bus contention.	Faster data transfer with no contention.
Bandwidth	Limited bandwidth shared by all components.	Higher bandwidth with dedicated buses for different tasks.
Bus Contention	High bus contention, leading to delays.	Low bus contention, reducing delays.
Efficiency	Less efficient due to shared resources.	More efficient with parallel communication.
Use Cases	Suitable for simpler, cost-sensitive systems.	Suitable for high-performance and complex systems.

Issues in Computer Design

When designing computers, engineers face several important challenges:

1. Performance

- Aim: Fast execution.
- Depends on processor speed, memory, and parallelism.

2. Cost

- Goal: Keep it affordable.
- Higher performance = Higher cost.

3. Power Consumption

- Important for mobiles/laptops.
- More power = more heat → needs better cooling.

4. Size

- Depends on the device (big for servers, small for phones).

- Affects layout and components.

5. Compatibility

- Should work with older software and hardware.
- Ensures smooth upgrades.

6. Scalability

- Easy to upgrade (RAM, storage, etc.).
- Extends device life.

7. Reliability

- Should run without crashing.
- Use error detection and backups.

8. Security: Protect data and programs from attacks.

9. Ease of Design: Simpler designs are easier to test and debug.

10. Time to Market: Faster development gives an edge in competition.

Computer Arithmetic: Number Systems

📌 What is a Number System?

A number system defines how numbers are represented and used in a computer.

🔹 1. Binary Number System (Base 2)

- Uses: 0 and 1 only.
- Used internally by all digital computers.
- Example:
  Decimal 5 = Binary 101
  $(1\times2^2 + 0\times2^1 + 1\times2^0 = 4 + 0 + 1 = 5)$

🔹 2. Octal Number System (Base 8)

- Uses digits: 0 to 7.
- Easier to read than binary, sometimes used in programming.
- Example:
  Decimal 9 = Octal 11

🔹 3. Hexadecimal Number System (Base 16)

- Uses: 0–9 and A–F (where A=10, B=11, ..., F=15).
- Compact representation of binary.
- Example:
  Decimal 255 = Hex FF
  $(15\times16^1 + 15\times16^0 = 240 + 15 = 255)$

➕ 4. Signed and Unsigned Numbers

🔸 Unsigned Numbers

- Only positive numbers (including zero).
- Example (4-bit):
  0000 = 0
  1111 = 15

🔸 Signed Numbers

- Can represent both positive and negative numbers.
- Common method: **2's Complement**
- In 4-bit:
  - 0111 = +7
  - 1000 = –8
  - 1111 = –1

🔁 Number System Conversions

1. Decimal to Binary

Steps:

- Divide the decimal number by 2
- Write down the remainder
- Repeat until quotient is 0
- Reverse the remainders

26 ÷ 2 = 13 → R: 0

13 ÷ 2 = 6 → R: 1

6 ÷ 2 = 3 → R: 0

3 ÷ 2 = 1 → R: 1

1 ÷ 2 = 0 → R: 1

→ Binary = 11010

---

2. Decimal to Octal

Steps:

- Divide the number by 8, record remainders
- Reverse the remainders

26 ÷ 8 = 3 → R: 2

3 ÷ 8 = 0 → R: 3

→ Octal = 32

---

3. Decimal to Hexadecimal

Steps:

- Divide the number by 16
- Remainders above 9 are replaced with letters (10 = A, ..., 15 = F)

26 ÷ 16 = 1 → R: 10 → A

1 ÷ 16 = 0 → R: 1

→ Hex = 1A

---

4. Binary to Decimal

Steps:

- Multiply each bit by 2^n (right to left) and add.

$= 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

= 16 + 8 + 0 + 2 + 0 = **26**

---

5. Binary to Octal

Steps:

- Group 3 bits from right
- Convert each group to octal digit

011 = 3

010 = 2

→ Octal = 32

---

6. Binary to Hexadecimal

Steps:

- Group 4 bits from right
- Convert each group to hex digit

0001 = 1

1010 = A

→ Hex = 1A

---

7. Octal to Decimal

Steps:

- Multiply each digit by 8^n (right to left)

Example: Octal 32

$= 3\times8^1 + 2\times8^0 = 24 + 2 = 26$

---

8. Hex to Decimal

Steps:

- Multiply each digit by 16^n

Hex 1A

$= 1\times16^1 + 10\times16^0 = 16 + 10 = 26$

---

9. Octal to Binary

Steps:

- Convert each octal digit to 3-bit binary

Example: 3 → 011, 2 → 010 → Binary = 11010

---

10. Hex to Binary

Steps:

- Convert each hex digit to 4-bit binary

Example: 1 → 0001, A → 1010 → Binary = 11010

🔁 Number System Conversion Guide

From	To	Method
Decimal	Binary	Divide by 2, collect remainders in reverse order
Decimal	Octal	Divide by 8, collect remainders in reverse order
Decimal	Hexadecimal	Divide by 16, convert remainders ≥10 to A–F, reverse the result
Binary	Decimal	Multiply each bit by 2^n (right to left), then sum
Binary	Octal	Group bits in 3s from right, convert each group to octal digit
Binary	Hexadecimal	Group bits in 4s from right, convert each group to hex digit
Octal	Binary	Convert each octal digit to its 3-bit binary equivalent
Hexadecimal	Binary	Convert each hex digit to its 4-bit binary equivalent
Octal	Decimal	Multiply each digit by 8^n (right to left), then sum
Hexadecimal	Decimal	Multiply each digit by 16^n (right to left), using A=10 to F=15

Shift and Add Multiplication in Binary

The **Shift and Add** method is a technique used in **binary multiplication**. It is an efficient way of multiplying binary numbers, similar to the long multiplication method in decimal.

✅ **Steps of the Shift and Add Method:**

1. **Start with two binary numbers**, say **A** and **B**.
2. **Multiply** the bits of **B** one by one starting from the least significant bit (rightmost).
3. If a bit in **B** is 1, add the shifted version of **A**.
4. If the bit in **B** is 0, add 0 (no change).
5. Each successive multiplication result is **shifted left** by 1 position.
6. **Repeat** until all bits of **B** are processed.

Booth's Algorithm for Binary Multiplication

Booth's Algorithm is a well-known algorithm for **binary multiplication** that handles both positive and negative numbers. It's an efficient method used in computers to multiply binary numbers, especially in cases involving signed numbers.

✅ **Overview of Booth's Algorithm:**

- It reduces the number of operations needed for multiplication.
- It works by examining pairs of bits in the multiplier, instead of working with each bit individually.
- The algorithm is particularly useful for **signed binary numbers** (i.e., numbers in **two's complement** representation).

➡️ **Steps of Booth's Algorithm:**

1. **Initialize**:
    - Start with two numbers: **Multiplicand (M)** and **Multiplier (Q)**.
    - Also, initialize a **Q-1** bit (it's set to 0 initially).
    - We also need a **Product Register** initialized to zero. The size of the register is the sum of the number of bits in the multiplicand and multiplier.
2. **Perform the operation**:
    - Look at the current and previous bits of the multiplier (Q_0 and Q_{-1}).
    - Depending on the pair of bits, perform one of the following actions:
        - **Q_0 Q_{-1} = 01**: Add the multiplicand to the product and shift right.
        - **Q_0 Q_{-1} = 10**: Subtract the multiplicand from the product and shift right.
        - **Q_0 Q_{-1} = 00** or **Q_0 Q_{-1} = 11**: Just shift right.
3. **Repeat the steps** until all bits of the multiplier have been processed.
4. **Final result** will be in the product register.