Module 1: Functional blocks of a computer

1. CPU

a. Central Processing Unit (CPU)

The **Central Processing Unit (CPU)** is often called the **brain of the computer**. It performs all the **processing tasks** and controls the operations of other components.

🔹 **Main Functions of CPU:**

i. **Fetch** – Takes instructions from memory.
ii. **Decode** – Understands the instruction.
iii. **Execute** – Performs the instruction (e.g., calculations, data movement).
iv. **Control** – Coordinates with input/output and memory.

🔹 **Components of CPU:**

i. **Arithmetic and Logic Unit (ALU)**
   - Performs **arithmetic operations** (like addition, subtraction) and **logical operations** (like AND, OR, NOT).

ii. **Control Unit (CU)**
   - **Directs the flow** of data between CPU, memory, and I/O.
   - Decodes instructions and tells ALU, memory, and I/O what to do.

iii. **Registers**
   - Small, fast **memory units inside CPU**.
   - Hold data, instructions, and addresses temporarily.
   - Examples: **Accumulator, Program Counter (PC), Instruction Register (IR)**.

🔹 **Working Cycle of CPU (Instruction Cycle):**

i. **Fetch** the instruction from memory.
ii. **Decode** the instruction in the CU.
iii. **Execute** the instruction in the ALU.
iv. **Store** the result (if needed).

2. Memory

Memory

Memory is a crucial functional block of a computer used to **store data and instructions** temporarily or permanently for processing.

🔹 **Types of Memory:**

a. Primary Memory (Main Memory)
- Directly accessible by the CPU.
- **Fast** but **limited in size**.
- Stores data and instructions currently in use.

✅ **Examples:**
- **RAM (Random Access Memory)**
  - **Volatile** (data lost when power off)
  - Temporary storage for running programs.
- **ROM (Read Only Memory)**
  - **Non-volatile** (data remains after power off)
  - Stores firmware or boot instructions.

b. Secondary Memory
- Used for **permanent data storage**.
- **Slower** than primary memory but **larger in size**.

✅ **Examples:**
- Hard Disk Drive (HDD)
- Solid State Drive (SSD)
- CD/DVD, Pen Drive

c. Cache Memory
- **Very fast** and located **inside or near the CPU**.
- Stores frequently used data and instructions.
- Improves CPU performance by reducing access time to main memory.

d. Registers
- Smallest and **fastest memory** inside the CPU.
- Temporarily stores **intermediate data**, instructions, and addresses.

🔹 **Key Characteristics of Memory:**

- **Speed** – Access time of memory.
- **Size** – Amount of data it can store.
- **Volatility** – Whether data is retained without power.

- **Cost** – Generally, faster memory is more expensive.

3. Input-output subsystems

Input-Output (I/O) Subsystems

The **Input-Output Subsystem** is responsible for **communication between the computer and the external world**. It includes all hardware and software used to **input data to the computer** or **output results from the computer**.

🔹 **Input Devices**

Used to **enter data and instructions** into the computer.

✅ **Examples:**
- Keyboard
- Mouse
- Scanner
- Microphone
- Touchscreen

🔹 **Output Devices**

Used to **display or deliver processed data** from the computer to the user.

✅ **Examples:**
- Monitor (Visual output)
- Printer (Hard copy output)
- Speakers (Audio output)
- Projectors

🔹 **I/O Interface**

Acts as a **bridge** between CPU/memory and I/O devices. It converts **electrical signals**, manages **data transfer**, and handles **device communication**.

✅ **Includes:**
- **Device Controller** – Manages one or more I/O devices.
- **I/O Ports** – Connect external devices (USB, HDMI, etc.)

🔹 **Types of I/O Techniques:**

i. **Programmed I/O**
   - CPU actively waits and controls the data transfer.

ii. **Interrupt-Driven I/O**
   - CPU does other tasks and is interrupted when the device is ready.

iii. **Direct Memory Access (DMA)**
   - Data is transferred directly between memory and device without CPU involvement.

🔹 **Importance of I/O Subsystem:**

- Enables interaction with users.
- Connects computer to external environment.
- Allows input of raw data and output of meaningful results.

4. Control unit

Control Unit (CU)

The **Control Unit** is a key component of the **CPU** that **directs and coordinates all operations** of the computer.

🔹 **Main Functions of Control Unit:**

i. **Fetches** instructions from memory.
ii. **Decodes** the instructions to understand what action is needed.
iii. **Sends control signals** to other parts like ALU, memory, and I/O devices to execute the instruction.
iv. **Manages data flow** between CPU and peripherals.

🔹 **Role in the Instruction Cycle:**

i. **Fetch** – CU gets the instruction from memory.
ii. **Decode** – CU interprets the instruction.
iii. **Control** – CU sends signals to execute it via ALU or other components.

🔹 **Types of Control Units:**

i. **Hardwired Control Unit**
   - Control logic is implemented with fixed electronic circuits.
   - **Faster** but **less flexible**.

ii. **Microprogrammed Control Unit**
   - Uses a control memory to store microinstructions.
   - **Easier to modify** and more **flexible**, but **slower**.

🔹 **Key Features:**

- Does **not process data**, only **controls operations**.

- Acts like a **traffic controller**, ensuring everything happens in the correct order.

5. Instruction set architecture of a CPU-register, instruction execution cycle, RTL interpretation of instructions, addressing modes, instruction set.

Instruction Set Architecture (ISA) of a CPU
Instruction Set Architecture (ISA) defines the **interface between hardware and software**. It is a part of the CPU architecture that describes:

- The **machine instructions** the CPU can execute
- The **registers**, **data types**, **addressing modes**, and **instruction formats**

🔹 A. Registers

Registers are **small, high-speed storage locations** inside the CPU used to store data temporarily during execution.
Types of Registers:

- **Program Counter (PC)**: Holds the address of the next instruction.
- **Instruction Register (IR)**: Holds the current instruction.
- **Accumulator (ACC)**: Stores intermediate results.
- **General Purpose Registers (R0, R1, etc.)**: Used for temporary storage of data.

🔹 B. Instruction Execution Cycle

Also called the **Instruction Cycle**, it involves the following steps:

i. **Fetch** – Get instruction from memory.
ii. **Decode** – Interpret what the instruction means.
iii. **Execute** – Perform the operation (e.g., arithmetic or memory access).
iv. **Store** – Save the result back to a register or memory.

➡️ These steps are **repeated for every instruction**.

🔹 C. RTL (Register Transfer Language) Interpretation

RTL (Register Transfer Language) describes how data moves between registers and memory using symbolic notation.

- RTL makes it easy to understand **how instructions are executed at the register level**.

🔹 D. Addressing Modes

Addressing modes define **how the CPU finds the operand (data)** required for an instruction.
Common Addressing Modes:

Mode	Description	Example
Immediate	Operand is part of the instruction	MOV R1, #5
Direct	Address of operand is given directly	MOV R1, [1000]
Indirect	Address of operand is in a register/memory	MOV R1, [R2]
Register	Operand is in a register	ADD R1, R2
Indexed	Uses base address + offset	MOV R1, [R2 + 4]

🔹 E. Instruction Set

An **Instruction Set** is the complete collection of instructions that a CPU can execute.
Types of Instructions:

i. **Data Transfer** – MOV, LOAD, STORE
ii. **Arithmetic** – ADD, SUB, MUL, DIV
iii. **Logical** – AND, OR, NOT
iv. **Control Transfer** – JMP, CALL, RET
v. **I/O Instructions** – IN, OUT

🧠 ISA defines the **binary encoding**, **number of operands**, and **operation type** for each instruction.

✅ **Summary:**

- **ISA** is the interface between software and hardware.
- It includes **registers, instruction formats, addressing modes**, and the **instruction set**.

- Understanding ISA is key to understanding how software communicates with CPU hardware.

6. Case study: instruction set of 8085 processor.

Case Study: Instruction Set of 8085 Processor
The **8085** is an **8-bit microprocessor** developed by Intel. It has a rich and simple **instruction set architecture** designed to support basic data processing and control operations.

🔹 Key Features of 8085:

- 8-bit processor (data bus is 8 bits wide)
- 16-bit address bus (can access 64KB memory)
- Contains **5 general-purpose registers (B, C, D, E, H, L)**, **accumulator (A)**, **program counter (PC)**, and **stack pointer (SP)**
- Includes **74 instructions** and **246 opcodes**

🔹 Categories of 8085 Instructions:

Category	Description	Example
1. Data Transfer	Move data between registers, memory	MOV A, B, LDA 2050H
2. Arithmetic	Perform addition, subtraction	ADD B, SUB C
3. Logical	Perform AND, OR, NOT, comparison	ANA B, CPI 32H
4. Branching	Alter flow of execution	JMP 2050H, CALL 3000H, RET
5. Stack & I/O	Interact with I/O devices, stack	PUSH B, POP D, IN 01H, OUT 02H
6. Machine Control	Control processor operation	HLT, NOP, DI, EI

🔹 Instruction Format in 8085:

8085 instructions can be **1-byte, 2-byte, or 3-byte** based on the operands.

Type	Example	Bytes Used
1-byte	MOV A, B	1 byte
2-byte	MVI A, 32H	2 bytes
3-byte	LDA 2050H	3 bytes

🔹 Addressing Modes in 8085:

1. **Immediate** – Operand is specified in the instruction (MVI A, 05H)
2. **Register** – Operand is a register (MOV A, B)
3. **Direct** – Memory address is given directly (LDA 2050H)
4. **Register Indirect** – Memory address is in register pair (MOV A, M where HL pair points to memory)
5. **Implicit** – Operation is predefined (CMA, RRC)

🔹 Example Instruction Execution:

Instruction: LDA 2050H
➡️ Action: Load the content of memory location 2050H into the accumulator.
➡️ Bytes: 3 (3A 50 20 in hex)

✅ **Summary:**

- 8085 has a **simple and powerful instruction set**.
- Supports multiple **instruction formats** and **addressing modes**.
- Used widely for learning **ISA and CPU instruction processing**.

Module 1: Data representation

1. Signed number representation

Signed Number Representation
Signed numbers allow representation of **both positive and negative integers** in binary. Since computers use only 0s and 1s, we need special techniques to indicate the sign.

🧠 **Most Significant Bit (MSB)** is used as the **sign bit**:

- 0 → Positive number
- 1 → Negative number

🔹 **Methods of Signed Number Representation:**

1. Sign-Magnitude Representation

- MSB indicates the sign, remaining bits indicate magnitude.
- Simple but not efficient for arithmetic operations.

📌 Example (4-bit):

- +5 = 0101
- -5 = 1101

2. 1's Complement Representation

- Positive: same as sign-magnitude.
- Negative: **invert all bits** of the positive binary number.

📌 Example (4-bit):

- +5 = 0101
- -5 = 1010 (1's complement of 0101)

✏️ **Issue**: Two representations of 0:
0000 (+0) and 1111 (–0)

3. 2's Complement Representation

✅ **Most commonly used** method in modern systems.

- Positive: same as unsigned binary.
- Negative: **1's complement + 1**.

📌 Example (4-bit):

- +5 = 0101
- -5 = 1's complement of 0101 → 1010 + 1 = 1011

✏️ **Only one zero**: 0000

✏️ Easy arithmetic without extra rules

🔹 **Range of Numbers (for n-bit representation):**

Representation	**Range**
Sign-Magnitude	$-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
1's Complement	$-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
2's Complement	$-(2^{n-1})$ to $+(2^{n-1}-1)$

📌 Example (for 4 bits):

- **2's Complement** range: –8 to +7

✅ **Summary:**

- Signed numbers are needed to represent **positive and negative values**.
- **2's complement** is preferred due to **simpler arithmetic and single zero**.
- Knowing how to **convert and calculate** in all three formats is important.

2. Fixed and floating- point representations

Fixed-Point and Floating-Point Representations

Computers need ways to represent **real numbers (with fractions)**. Two main methods are:

🔹 **A. Fixed-Point Representation**

In **Fixed-Point**, the **position of the decimal (or binary point)** is fixed.

➤ **Types:**

a. **Integer Representation** – No fractional part (e.g., 00010110 = 22).

b. **Fixed Fractional Representation** – A fixed number of bits are reserved for the fraction.

📌 **Example:**

If we have 8 bits and assume **4 bits for integer** and **4 bits for fraction**:
Binary: 00011010 → Integer: 0001 (1), Fraction: 1010 (0.625)
Result = 1.625

✅ **Pros:**

- Simple hardware
- Fast computation

❌ **Cons:**

- Limited range and precision
- Not suitable for very large or very small numbers

🔹 **B. Floating-Point Representation**

In **Floating-Point**, the decimal point can **"float"** — meaning its position can change depending on the exponent.

🔸 **General Format:**

± Mantissa × Base ^ Exponent

➤ **IEEE 754 Standard (for 32-bit float):**

Field	**Bits**	**Purpose**
Sign Bit	1	0 = positive, 1 = negative
Exponent	8	Biased exponent
Mantissa	23	Significant digits (normalized form)

🔁 **Comparison Table:**

Feature	**Fixed-Point**	**Floating-Point**
Decimal position	Fixed	Can vary (floating)
Precision	Low	High
Range	Small	Large
Speed	Fast	Slower (due to complex logic)
Hardware	Simple	Complex

✅ **Summary:**

- **Fixed-point** is good for **simple, fast calculations** with limited range.
- **Floating-point** is suitable for **scientific and real-time applications** due to its **high range and precision**.
- **IEEE 754** is the standard for floating-point in modern processors.

3. Character representation

Character Representation

Computers represent characters (letters, digits, symbols) using **binary codes**. Each character is assigned a **unique binary value** through standard coding systems.

🔹 **Why Character Representation?**

Computers understand only **binary (0 and 1)**. To handle text (like 'A', '3', '@'), characters must be **encoded** into binary using a **character set**.

🔹 **Common Character Encoding Standards**

Standard	**Bits**	**No. of Characters**	**Used For**
ASCII	7/8	128 / 256	English letters, digits, symbols
EBCDIC	8	256	IBM mainframe systems
Unicode	16/32	>65,000	All world languages, emojis, symbols

🔸 **1. ASCII (American Standard Code for Information Interchange)**

- **7-bit** code (values from 0 to 127)
- Extended ASCII uses 8 bits (0 to 255)

📌 Examples:

- 'A' = 65 = 01000001
- 'a' = 97 = 01100001
- '0' = 48 = 00110000
- '@' = 64 = 01000000

🔸 **2. EBCDIC (Extended Binary Coded Decimal Interchange Code)**

- **8-bit** code developed by IBM.
- Not widely used in modern systems.
- Mainly found in **mainframe and legacy systems**.

🔸 **3. Unicode**

- Supports **global languages**, emojis, and symbols.
- Can be **16-bit (UTF-16)** or **32-bit (UTF-32)**
- First 128 characters of Unicode = same as ASCII

📌 Example:

- 'A' = U+0041
- 'अ' (Devanagari) = U+0905
- '😊' = U+1F60A

✅ **Summary:**

- Characters are stored using **binary codes**.
- **ASCII** is most common for English text.
- **Unicode** supports **multiple languages and symbols**, making it ideal for modern applications.
- **Character encoding** is essential for **data communication and storage**.

4. Computer arithmetic- integer addition and subtraction, ripple carry adder, carry look-ahead adder, etc.

Computer Arithmetic

Computer arithmetic deals with performing basic operations like **addition**, **subtraction**, **multiplication**, and **division** using binary numbers. This topic focuses on **integer addition & subtraction**, and types of **binary adders**.

🔹 **A. Integer Addition and Subtraction**

✅ **Binary Addition Rules:**

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

📌 Example:

```
 1011  (11)
+ 0101  (5)
= 10000  (16, with carry)
```

✅ **Binary Subtraction Rules:**

A	B	Diff	Borrow
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

📌 Binary subtraction is often done using **2's complement**:

A – B = A + (2's complement of B)

🔹 **B. Ripple Carry Adder (RCA)**

- A simple binary **adder circuit** built by connecting multiple **full adders** in series.
- Each full adder handles 1-bit of two binary numbers.
- The **carry output** from each adder becomes the **carry input** to the next.

📌 Example (4-bit RCA):
Add A3A2A1A0 and B3B2B1B0 using 4 full adders connected one after another.

❌ **Problem:**

- **Slow** due to **carry propagation delay** (carry ripples through all adders).

🔹 **C. Carry Look-Ahead Adder (CLA)**

✅ **Faster** alternative to ripple carry adder.

- Uses **Generate (G)** and **Propagate (P)** logic to predict the carry in advance.
- Eliminates the delay of carry rippling through stages.

➤ **Key Equations:**

- **Generate**: Gi = Ai · Bi
- **Propagate**: Pi = Ai + Bi
- **Carry**: Ci+1 = Gi + Pi · Ci

✅ **Advantage:**

- Much **faster addition**, especially for large bit sizes.
- Used in **modern processors** for high-speed arithmetic.

🔹 **Optional: Other Adders (for reference)**

- **Half Adder**: Adds two bits, gives Sum and Carry.
- **Full Adder**: Adds two bits plus a Carry-in.
- **Parallel Adder**: Group of full adders for multi-bit numbers.
- **Carry-Save Adder**: Used in multipliers to reduce propagation.

5. Multiplication- shift-and add, Booth multiplier, carry save multiplier, etc.

Multiplication in Computer Arithmetic

Binary multiplication is more complex than addition but follows the same concept as decimal multiplication — multiply and add partial results.

🔹 **A. Shift-and-Add Multiplication**

This is the **simplest binary multiplication** method. It mimics long-hand multiplication using shifts and additions.

🔸 **Steps:**

1. Start with Product = 0.
2. Check each bit of the **Multiplier** from LSB to MSB.
3. If bit = 1, **add the Multiplicand** to the product.
4. **Left shift** the multiplicand by one after each step.
5. Repeat for all bits.

📌 **Example:** Multiply A = 1010 (10) and B = 1100 (12)

```
   1010   (Multiplicand)
 × 1100   (Multiplier)
-------------
   0000   (0 × 1010)
+  00000    (0 × 1010 shifted)
+  101000    (1 × 1010 shifted)
+ 1010000    (1 × 1010 shifted again)
-------------
= 11000000 (120)
```

✅ Simple to implement
❌ Slow due to multiple additions

🔹 **B. Booth's Multiplication Algorithm**

Efficient technique for **signed binary multiplication** using 2's complement numbers.

🔸 **Key Idea:**

- **Reduces the number of additions** required by encoding sequences of 1s.
- Uses **Booth's Recoding** to handle both +ve and –ve multipliers.

🔸 **Booth's Recoding Rules (for 2-bit pairs):**

Bit Pair	Operation
00	No operation
01	Add multiplicand
10	Subtract multiplicand
11	No operation

Then, arithmetic **right shift** after each step.

✅ Handles **positive & negative** numbers
✅ Reduces number of steps in some cases
❌ More complex logic

🔹 **C. Carry Save Multiplier**

Used in **high-speed multiplication**, especially in **hardware circuits**.

🔸 **Idea:**

- Instead of summing partial products sequentially, it uses **carry-save adders** to add all partial products simultaneously.
- Delays carry propagation to the end — **saves time**.

🔸 **Steps:**

i. Generate all partial products.
ii. Use **carry-save adders** to add them in parallel.
iii. Final result is computed using a **carry-propagate adder**.

✅ Very **fast**, suitable for **pipelined or parallel processors**

❌ Complex hardware, used in high-performance systems

6. Division restoring and non-restoring techniques

Division Techniques: Restoring & Non-Restoring

Binary division is similar to long division in decimal. Two main methods are used in computers:

- **Restoring Division**
- **Non-Restoring Division**

🔹 **Key Terms:**

- **Dividend (Q)**: Number to be divided
- **Divisor (M)**: Number to divide by
- **Quotient**: Result of division
- **Remainder**: What's left after division

🔸 **A. Restoring Division**

This method **restores** the previous value of the remainder when a subtraction results in a negative value.

➤ **Steps:**

i. Initialize **remainder = 0**
ii. Repeat for n bits (where n is the bit-width of the dividend):
   - Left shift **[Remainder + Quotient]** by 1 bit
   - Subtract **Divisor (M)** from the remainder
   - If result is **negative**:
     - Restore remainder (i.e., **add back M**)
     - Set current Quotient bit to **0**
   - Else:
     - Keep new remainder
     - Set current Quotient bit to **1**

📌 **Example (4-bit):**
Divide Q = 1011 (11) by M = 0011 (3) → Result = 0111 (3) remainder 10

✅ **Pros:**

- Simple to understand

❌ **Cons:**

- **Restoring** step adds overhead

🔸 **B. Non-Restoring Division**
This is an **optimized** version where the restoring step is **skipped**. It uses the sign of the remainder to decide the next operation.

➤ **Steps:**

i. Initialize **remainder = 0**
ii. Repeat for n bits:
   - Left shift **[Remainder + Quotient]**
   - If Remainder is **positive**:
     - Subtract **M**
     - If result is positive → Quotient bit = 1
     - Else → Quotient bit = 0
   - If Remainder is **negative**:
     - Add **M**
     - If result is positive → Quotient bit = 0
     - Else → Quotient bit = 1
iii. Final correction: If remainder is negative, add back **M**

✅ **Pros:**

- Faster than restoring
- No extra "restore" operation

❌ **Cons:**

- Slightly more complex logic than restoring

🔄 **Comparison Table:**

Feature	**Restoring Division**	**Non-Restoring Division**
Speed	Slower	Faster
Restoration Step	Yes	No
Complexity	Lower	Slightly higher
Common Usage	Simpler processors	Optimized processors

✅ **Summary:**

- Both techniques are used for **binary division** in CPUs.
- **Restoring** is easy to understand but slower.
- **Non-Restoring** is more efficient and widely used in modern hardware.

7. Floating point arithmetic.

Floating Point Arithmetic
Floating point representation allows computers to **represent real numbers (fractions, very large or very small numbers)** efficiently, similar to scientific notation.

🔹 **A. What is Floating Point Representation?**
Just like in scientific notation (e.g., 1.23×10^4), floating point numbers are expressed as:
N = M × b^ E
Where:

- **M = Mantissa** (also called Significand)
- **b = Base** (commonly 2 in binary systems)
- **E = Exponent** (signed number)

🔹 **B. IEEE 754 Floating Point Standard**
This is the most widely used format for floating-point representation in computers.

🔸 **1. Single Precision (32-bit)**

Field	**Bits**	**Description**
Sign	1	0 = positive, 1 = negative
Exponent	8	Biased exponent (bias = 127)
Mantissa	23	Fractional part (normalized)

🔍 Example:
-5.75 in IEEE 754 (32-bit):
Sign = 1
Binary = 101.11 = 1.0111×2^2
Exponent = 2 + 127 = 129 = 10000001
Mantissa = 01110000000000000000000
So, IEEE 754 = 1 10000001 01110000000000000000000

🔸 **2. Double Precision (64-bit)**

Field	**Bits**	**Description**
Sign	1	
Exponent	11	Bias = 1023
Mantissa	52	

🔹 **C. Floating Point Operations**
Floating point arithmetic is different from integer arithmetic due to **normalization and rounding**.

🔸 **1. Addition/Subtraction**

1. **Align exponents** by shifting the smaller number.
2. Add/subtract mantissas.
3. Normalize the result.
4. Round if needed.

🔸 **2. Multiplication**

1. Multiply mantissas.
2. Add exponents.
3. Normalize and round.

🔸 **3. Division**

1. Divide mantissas.
2. Subtract exponents.
3. Normalize and round.

🔹 **D. Special Values in IEEE 754**

Value Type	**Exponent**	**Mantissa**	**Descripti on**
Zero	0	0	+0 or -0
Infinity	All 1s	0	+∞ or −∞
NaN	All 1s	≠ 0	Not a Number (e.g., 0/0)
Denormalized	0	≠ 0	Very small numbers

✅ **Summary:**

Operation	**Steps**
Addition/Sub	Align exponent → Add/Sub mantissas → Normalize → Round
Multiplication	Multiply mantissas → Add exponents → Normalize → Round
Division	Divide mantissas → Subtract exponents → Normalize → Round

Module 2: Introduction to X86 architecture/ CPU control unit design

1. Hardwired and micro-programmed design approaches.

Hardwired and Microprogrammed Control Unit
The **Control Unit (CU)** in a CPU is responsible for **generating control signals** that guide the execution of instructions.
There are **two main ways to design a Control Unit**:

i. **Hardwired Control**

ii. **Microprogrammed Control**

🔹 **A. Hardwired Control Unit**

In a **hardwired control unit**, control signals are generated by **combinational logic circuits** like gates, flip-flops, and decoders.

✅ **Features:**

- Uses **finite state machines (FSM)**.
- Fast and efficient.
- Difficult to modify once designed.

📌 **Example:**

- Instructions are decoded using logic circuits.
- Control signals are generated based on instruction opcode and clock.

✅ **Advantages:**

- **High speed** (suitable for RISC).
- Efficient for **simple, fixed instruction sets**.

❌ **Disadvantages:**

- Difficult to design for **complex instruction sets**.
- **Not flexible**; changes require redesigning the whole circuit.

🔹 **B. Microprogrammed Control Unit**

In this design, control signals are generated by executing a **sequence of microinstructions** stored in a **control memory (ROM)**.

✅ **Features:**

- Each machine instruction is broken into **micro-operations**.
- Micro-operations are executed via a **microprogram**.

📌 **Components:**

- **Control Memory**: Stores microinstructions.
- **Microinstruction Register**: Holds current microinstruction.
- **Microprogram Counter**: Points to next microinstruction.

✅ **Advantages:**

- **Easy to modify** or update (especially in CISC systems).
- Easier to design and debug.

❌ **Disadvantages:**

- **Slower** than hardwired control due to memory access.
- May require more memory.

🔁 **Comparison Table:**

Feature	**Hardwired Control**	**Microprogrammed Control**
Speed	Faster	Slower
Flexibility	Low	High
Implementation	Logic circuits	Microinstructions in memory
Suitable for	RISC processors	CISC processors
Modifiability	Hard to modify	Easy to modify

✅ **Summary:**

- **Hardwired CU**: Faster but fixed logic.
- **Microprogrammed CU**: Flexible but slower.
- Choice depends on **processor design goals**.

2. Case study- design of a simple hypothetical CPU

Case Study – Design of a Simple Hypothetical CPU

This topic walks you through the design of a basic CPU with a small instruction set, simple control logic, and standard operations — a **"learning model CPU"** used to understand how instructions are executed.

We'll design a CPU step-by-step with essential components.

🔹 **A. Assumptions for the Hypothetical CPU**

To keep it simple, let's assume:

- 8-bit word size
- 4-bit opcodes (→ 16 instructions)
- 8 general-purpose registers (R0 to R7)
- Single accumulator (A)
- Simple instruction types: data transfer, arithmetic, logic, and control

🔹 **B. Basic Components of the CPU**

Component	**Function**
Registers	Store temporary data (R0 to R7, A)
ALU	Performs arithmetic and logic operations
Control Unit	Decodes instructions and generates control signals
Program Counter (PC)	Holds address of next instruction
Instruction Register (IR)	Holds current instruction
Memory	Stores instructions and data
Bus System	Transfers data among components

🔹 **C. Instruction Format**

Let's define a simple **8-bit instruction format**:

Bits	**Description**
4 bits	Opcode (operation code)
4 bits	Operand (register/memory)

📌 Example:

0001 0011 → Opcode: 0001 (ADD), Operand: 0011 (R3)

🔹 **D. Example Instruction Set**

Opcode	**Mnemonic**	**Operation**
0000	LOAD	Load from register to Accumulator
0001	ADD	Add register to Accumulator
0010	SUB	Subtract register from Acc
0011	AND	Logical AND with Acc
0100	OR	Logical OR with Acc
0101	STORE	Store Accumulator to Register
0110	JMP	Jump to address
1111	HLT	Halt

🔹 **E. Execution Cycle (Instruction Cycle)**

Each instruction is executed in the following phases:

1. **Fetch**:
   - IR ← Memory [PC]
   - PC ← PC + 1
2. **Decode**:
   - Decode opcode and operand
3. **Execute**:
   - Perform operation via ALU or memory access

🔹 **F. Sample Instruction Execution**

📌 Example Instruction: 0001 0011 (ADD R3)

1. **Fetch**:
   - Instruction fetched from memory: 0001 0011
2. **Decode**:
   - Opcode = 0001 → ADD
   - Operand = 0011 → R3
3. **Execute**:
   - A ← A + R3

🔹 **G. Block Diagram (Conceptual)**

A simple CPU would include:

- Program Counter
- Instruction Register
- Control Unit
- ALU
- Registers (R0-R7, A)
- Data & Instruction Memory
- Internal Bus

✅ **Summary**

This hypothetical CPU:

- Helps understand **control flow, instruction decoding**, and **ALU operations**

- Uses a simple architecture to demonstrate **basic CPU functionality**
- Is the foundation for understanding real-world CPUs like 8085 or ARM

Module 2: Memory system design

1. Semiconductor memory technologies

Semiconductor Memory Technologies
Semiconductor memory refers to memory devices made using semiconductor materials, typically **silicon**. These are essential components of computers and digital systems for storing data and programs.
Types of Semiconductor Memory
Semiconductor memory can be classified into two main categories:
A. Volatile Memory
Data is lost when power is turned off.
a. **RAM (Random Access Memory)**
   - **SRAM (Static RAM):**
     - Uses flip-flops.
     - Faster and more expensive.
     - Used for cache memory.
     - No need for refreshing.
   - **DRAM (Dynamic RAM):**
     - Uses capacitors.
     - Slower and cheaper than SRAM.
     - Needs periodic refreshing.
     - Used for main memory (in computers).

B. Non-Volatile Memory
Retains data even after power is turned off.
a. **ROM (Read-Only Memory):**
   - Data is written once, mainly used for firmware.
   - Types:
     - **PROM (Programmable ROM):** Can be programmed once.
     - **EPROM (Erasable PROM):** Can be erased using UV light and reprogrammed.
     - **EEPROM (Electrically Erasable PROM):** Can be erased electrically.
     - **Flash Memory:** A type of EEPROM, supports faster erase/write cycles. Widely used in USB drives, SSDs.

2. Memory organization

Memory Organization
Memory organization refers to the way memory is structured and accessed in a computer system. It affects how data is stored, retrieved, and managed efficiently.

a. Memory Hierarchy
Memory is organized in a hierarchy to balance **speed**, **cost**, and **capacity**:

Level	Type	Speed	Cost	Capacity
i.	Registers	Very High	Very High	Very Low
ii.	Cache (L1, L2)	High	High	Low
iii.	Main Memory (RAM)	Moderate	Moderate	Medium
iv.	Secondary Memory	Low	Low	High
v.	Tertiary Storage	Very Low	Very Low	Very High

- **Registers:** Located inside the CPU; store data currently in use.
- **Cache:** Small, fast memory between CPU and RAM (L1, L2, L3).
- **Main Memory (RAM):** Stores active programs and data.
- **Secondary Memory:** Hard drives, SSDs.
- **Tertiary Storage:** Optical disks, magnetic tapes.

b. Memory Structure
Memory is logically divided into:
- **Word:** Smallest unit of data that CPU can read/write in one cycle.
- **Byte:** 8 bits; basic unit for measuring memory.
- **Address:** Unique number assigned to each memory location.

c. Addressing Modes
Defines how memory locations are accessed:
- **Direct Addressing**
- **Indirect Addressing**
- **Indexed Addressing**
- **Base Register Addressing**
- **Immediate Addressing**

d. Memory Access Methods
i. **Sequential Access:** Data is accessed in order (e.g., tapes).
ii. **Direct Access:** Any location can be accessed directly (e.g., HDD).
iii. **Random Access:** All locations equally accessible (e.g., RAM).
iv. **Associative Access (Content Addressable):** Data is accessed by content, not address (e.g., cache).

---

e. Word Alignment and Memory Interleaving
- **Word Alignment:** Ensures that word-sized data is stored at memory addresses that are multiples of word size. Improves access speed.
- **Memory Interleaving:** Divides memory into modules to allow simultaneous access and increase speed.

Module 2: peripheral devices and their characteristics

1. Input-output subsystems

Input-Output (I/O) Subsystems
The **I/O subsystem** is a critical part of an operating system that manages all input and output operations. It acts as a bridge between the **hardware (I/O devices)** and the **system software**, ensuring smooth and efficient communication.

Key Components of I/O Subsystem
a. I/O Devices
These are hardware components used for input (e.g., keyboard, mouse) and output (e.g., monitor, printer).
b. Device Drivers
- A software module that acts as an interface between the OS and a specific device.
- Translates generic I/O commands into device-specific commands.
- Example: A printer driver tells the OS how to communicate with a specific model of printer.

c. I/O Controllers
- Hardware that controls one or more I/O devices.
- Handles data transfer between the CPU and devices.
- Includes registers for data, control, and status.

d. Interrupts
- Signals sent to the CPU when an I/O operation is complete or needs attention.
- Helps avoid busy waiting by letting CPU perform other tasks while waiting for I/O completion.

I/O Techniques
a. Programmed I/O
- CPU actively waits and checks device status.
- Simple but inefficient (CPU is busy-waiting).

b. Interrupt-Driven I/O
- CPU issues a command and continues executing.
- Device sends an interrupt when it's ready.
- More efficient than programmed I/O.

c. Direct Memory Access (DMA)
- Allows devices to transfer data directly to/from memory without CPU intervention.
- CPU is only involved at the beginning and end of the transfer.
- Best for large data transfers (like from hard drives or network cards).

I/O Scheduling

- Determines the order in which I/O requests are processed.
- Aims to reduce waiting time and improve system performance.
- Common disk scheduling algorithms: FCFS, SSTF, SCAN, LOOK.

Functions of I/O Subsystem in OS

- Abstracts device details for user programs.
- Manages buffering, caching, and spooling.
- Handles errors and recoveries in I/O operations.
- Manages sharing of I/O devices among multiple processes.

2. I/O device interface

I/O Device Interface

An **I/O device interface** is the **communication link** between the **CPU (or memory)** and **I/O devices**. It allows the operating system and applications to communicate with hardware devices using standardized methods.

Purpose of I/O Interface

- To provide a **consistent way** to connect and control various I/O devices.
- To **handle the differences** between the CPU and I/O devices (speed, data format, control methods).

Main Components of I/O Interface

Component	Description
Data Register	Temporarily holds data to be transferred between device and system.
Control Register	Contains control signals (e.g., start, stop, reset) sent to the device.
Status Register	Indicates device status (ready, busy, error).
Address Decoder	Identifies the correct I/O device using a unique address.

Types of I/O Device Interfaces

a. **Memory-Mapped I/O**

- I/O devices are assigned **memory addresses**.
- CPU uses normal memory instructions (like MOV) to communicate.
- Pros: Simple programming model.
- Cons: Consumes part of memory address space.

b. **I/O-Mapped (Port-Mapped) I/O**

- I/O devices have **separate I/O addresses**, distinct from memory.
- Special instructions (like IN and OUT) are used for I/O.
- Pros: Full memory space is available for RAM.
- Cons: Needs special I/O instructions.

Synchronous vs Asynchronous I/O

Type	Description
Synchronous	CPU waits until the I/O operation is completed.
Asynchronous	CPU continues execution and is notified via interrupt when I/O completes.

Functionality of I/O Interface

- Synchronizes data transfer between devices and CPU.
- Manages device control and monitoring (start, stop, reset).
- Converts data formats (e.g., serial to parallel).
- Handles interrupts and error detection.

3. I/O transfers- program controlled, interrupt driven and DMA, privileged and non-privileged instructions, software interrupts and exceptions.

I/O Transfers & Related Concepts

This topic deals with the **methods of data transfer** between the CPU and I/O devices, and the classification of instructions and interrupts in an operating system.

A. I/O Transfer Methods

1. Program Controlled I/O (Polling)

- The **CPU continuously checks** the device status until it is ready for data transfer.
- **Simple** but **inefficient**, as CPU wastes time in busy-waiting.

Example: Reading a key from the keyboard by repeatedly checking if a key is pressed.

2. Interrupt-Driven I/O

- The CPU issues an I/O command and **continues executing other tasks**.
- The device **sends an interrupt** when it's ready for data transfer.
- More **efficient** than polling as it saves CPU time.

Example: Printer signals CPU when it's ready to accept more data.

3. Direct Memory Access (DMA)

- A **DMA controller** handles the data transfer directly between the I/O device and **main memory**, without CPU intervention.
- CPU is only involved to **initiate** and **complete** the transfer.
- Best suited for **large blocks of data** (e.g., hard disk transfers).

Advantages:

- Faster transfers.
- Frees up CPU for other tasks.

B. Privileged and Non-Privileged Instructions

1. Privileged Instructions

- Only **executed in kernel mode** (supervisor mode).
- Used for **critical operations** like I/O control, memory management, etc.
- Prevents unauthorized access to hardware.

Examples: IN, OUT, HLT, ENABLE INTERRUPTS

2. Non-Privileged Instructions

- Executed in **user mode**.
- Cannot directly access hardware or OS services.
- Used in normal user applications.

Examples: Arithmetic operations, logical operations, data movement.

C. Software Interrupts and Exceptions

1. Software Interrupts

- Generated by **software (program)** to request services from the operating system.
- Usually triggered by **system calls**.
- Allows user programs to access OS services safely.

Example: INT 80h in x86 assembly to call Linux OS functions.

2. Exceptions

- Events that **disrupt normal program flow** due to errors or special conditions.
- Handled by the OS through an **exception handler**.

Types of Exceptions:

- **Divide by zero**
- **Invalid memory access**
- **Arithmetic overflow**

4. Program and processes- role of interrupts in process state transitions

Program and Processes – Role of Interrupts in Process State Transitions

A. Program vs Process

Program	Process
A **static set of instructions** stored on disk.	A **dynamic execution** of a program.
Passive entity.	Active entity (with state and resources).
No CPU or memory usage.	Requires CPU, memory, I/O, etc.
Example: A .exe file.	A running instance of the .exe file.

B. Process States

A process goes through various states during its execution:

1. **New** – Process is being created.
2. **Ready** – Process is ready to run, waiting for CPU.
3. **Running** – CPU is executing the process.
4. **Waiting (Blocked)** – Process is waiting for some event (like I/O completion).

5. **Terminated** – Process has finished execution.

C. Process State Transitions
Here's how a process transitions between states:

- **New → Ready**: OS admits the process.
- **Ready → Running**: Scheduler assigns CPU.
- **Running → Waiting**: Process requests I/O or some event.
- **Waiting → Ready**: I/O completed or event occurred.
- **Running → Ready**: Preemption (higher priority process or time slice ends).
- **Running → Terminated**: Process completes or is killed.

D. Role of Interrupts in Process State Transitions
Interrupts are **signals to the CPU** that something needs immediate attention. They play a crucial role in **managing processes** efficiently.
Types of Interrupts and Their Roles:

1. **Timer Interrupt**
    - Triggered after a set time slice.
    - Causes **pre-emption**:
        - Running → Ready
        - Allows multitasking by giving CPU to another process.
2. **I/O Interrupt**
    - Triggered when I/O operation is complete.
    - Changes process state:
        - Waiting → Ready
3. **Hardware Interrupt**
    - Comes from external devices (keyboard, mouse).
    - May wake up waiting processes or create new ones.
4. **Software Interrupt (System Call)**
    - A process requests OS services (like file access).
    - Can cause:
        - Running → Waiting (if waiting for I/O or resource)
5. **Exception (Trap)**
    - Caused by errors (e.g., divide by zero).
    - OS may terminate the process (Running → Terminated) or handle it differently.

5. I/O device interfaces- SCII, USB.

I/O Device Interfaces – SCSI and USB
I/O device interfaces are **standardized communication systems** used to connect peripheral devices (like printers, hard drives, etc.) to a computer system. Two commonly used interfaces are:

A. SCSI (Small Computer System Interface)
Overview:

- Pronounced **"scuzzy"**.
- A **parallel interface** standard used to connect and transfer data between computers and peripheral devices.
- Originally developed in the 1980s, still used in high-performance and server environments.

Key Features:

- Supports connection of **multiple devices** (up to 7 or 15) on a single bus.
- **Faster** than traditional IDE.
- Devices can be **daisy-chained**.
- Supports a wide range of devices: hard disks, scanners, CD-ROMs, etc.

Advantages:

- High-speed data transfer.
- Multi-device support.
- Can perform **multiple I/O operations simultaneously** (supports command queuing).

Disadvantages:

- More **expensive** than simpler interfaces.
- Complex configuration (device ID, termination, etc.).

B. USB (Universal Serial Bus)
Overview:

- A **serial bus standard** for connecting peripherals to computers.
- Developed in the mid-1990s, now the most common I/O interface for consumer devices.

Key Features:

- **Plug and play** & **hot-swappable**.
- Supports up to **127 devices** via hubs.
- Different versions: **USB 1.1, 2.0, 3.0, 3.1, 3.2, USB4** with increasing data transfer speeds.

Advantages:

- Easy to use, no manual configuration.
- Widely supported and very flexible.
- Supports power delivery to charge devices.

Disadvantages:

- Lower performance compared to SCSI for enterprise-level data transfer (though USB 3.2 and USB4 are very fast).

Comparison Table: SCSI vs USB

Feature	**SCSI**	**USB**
Interface Type	Parallel	Serial
Devices Supported	7 to 15	Up to 127 (via hubs)
Speed	High (depends on version)	Moderate to High (USB 4 is fast)
Configuration	Manual (ID, termination)	Automatic (Plug & Play)
Hot Swapping	Limited	Supported
Common Usage	Servers, Workstations	Consumer devices (keyboard, USB drive)

Module 3: pipelining

1. Basic concepts of pipelining

Basic Concepts of Pipelining
Pipelining is a technique used in **CPU design** to improve performance by **executing multiple instructions simultaneously** in different stages.

What is Pipelining?

- Like an **assembly line**, pipelining breaks instruction execution into **stages**, where each stage performs a part of the task.
- While one instruction is being executed, the next can be decoded, and another can be fetched — all at the same time.

Typical Stages in Instruction Pipeline

1. **IF – Instruction Fetch**
2. **ID – Instruction Decode**
3. **EX – Execute**
4. **MEM – Memory Access**
5. **WB – Write Back**

Each stage takes one clock cycle. So ideally, after the first few cycles, one instruction completes **every cycle**.

Advantages of Pipelining

- Increases **CPU throughput** (number of instructions executed per unit time).
- Makes better use of CPU components.
- Speeds up instruction execution **without increasing clock speed**.

Ideal Speedup
If there are **n stages**, then ideally:
Speedup = n
But practically, due to hazards and delays, actual speedup is less than n.

Types of Pipeline Hazards

1. **Structural Hazard**
    - Occurs when hardware resources are shared.
    - Example: One memory used for both instructions and data.
2. **Data Hazard**
    - Occurs when an instruction depends on the result of a previous one.
    - Solved using **data forwarding** or **stalling**.
3. **Control Hazard**
    - Caused by **branching** (e.g., if-else, loops).
    - Solved using **branch prediction**.

Pipelining vs Non-Pipelining

Feature	**Pipelined CPU**	**Non-Pipelined CPU**
Execution Style	Overlapped	Sequential
Speed	Faster	Slower

Feature	Pipelined CPU	Non-Pipelined CPU
Complexity	Higher	Lower
Efficiency	Better	Less efficient

Real-Life Analogy
Like a car wash with 5 stations: soap, scrub, rinse, dry, wax.
Each car enters the next stage every few minutes — multiple cars are processed **simultaneously**.

2. Throughput and speedup

Throughput and Speedup
Both terms are used to **measure system performance**, especially in the context of **process execution** and **algorithm efficiency**.

A. Throughput

- **Definition:**
  The **number of processes (or tasks)** completed per unit of time.
- **Unit:** Tasks per second, instructions per cycle (IPC), etc.
- **High throughput** implies the system is handling more work efficiently.

B. Speedup

- **Definition:**
  A measure of **how much faster** a system or algorithm performs after **improvement or parallelization**.
- **Ideal Speedup:** Equal to the number of processors used (in parallel systems).
  But due to overhead and dependency, real speedup is often less.

Key Differences

Feature	Throughput	Speedup
Measures	Work done per time	Improvement in execution speed
Focus	Quantity of work	Performance boost
Related to	System efficiency	Algorithm or system optimization
Higher is	Better	Better

3. Pipeline hazards

Pipeline Hazards
In pipelined processors, multiple instructions are overlapped during execution. However, this parallelism can sometimes cause **conflicts**, which are called **pipeline hazards**.

✅ **What is a Pipeline Hazard?**
A **pipeline hazard** occurs when the next instruction **cannot execute in the next clock cycle**, due to some issue or dependency, thus **reducing performance** or **stalling the pipeline**.

🧩 **Types of Pipeline Hazards**

1. Structural Hazards

- Occur when **hardware resources** are not sufficient to support all simultaneous operations.
- Example: If both instruction fetch and data access need memory at the same time, but only one memory unit exists.

2. Data Hazards

- Occur when **instructions depend on the results** of previous instructions.

✅ **Types of Data Hazards:**

Type	Description	Example
RAW (Read After Write)	Instruction reads a value before it is written.	ADD R1, R2, R3 followed by SUB R4, R1, R5
WAR (Write After Read)	Instruction writes a value before it is read.	SUB R4, R1, R5 followed by MOV R1, R6
WAW (Write After Write)	Two instructions write to the same register in wrong order.	MOV R1, R2 followed by ADD R1, R3, R4

📌 **RAW** is the most common hazard in normal pipelines.

3. Control Hazards (Branch Hazards)

- Occur due to **branching or jumping instructions** (like if, goto, loop).
- The next instruction is not known until the branch decision is made.

Example:
assembly

🛠️ **How to Handle Hazards**

Hazard Type	Handling Techniques
Structural	Hardware duplication, pipeline scheduling
Data	Forwarding (bypassing), pipeline stalling
Control	Branch prediction, delay slots, speculative execution

⚠️ **Pipeline Stall (Bubble)**

- A **stall** or **bubble** is a delay inserted into the pipeline to resolve a hazard.
- Reduces performance, so pipelines try to avoid them where possible.

Module 3: Parallel processors

1. Introduction to parallel-processors

Introduction to Parallel Processors

✅ **Definition:**
A **parallel processor** is a system that **uses multiple processing elements (CPUs or cores)** to perform **multiple tasks simultaneously**, with the goal of **improving performance**, **speed**, and **efficiency**.

✅ **Why Parallel Processing?**

- Traditional (sequential) processing executes one instruction at a time.
- Parallel processing:
  - Increases **speed** of execution.
  - Handles **large datasets** efficiently.
  - Is essential for **scientific computing**, **AI**, **graphics**, and **real-time systems**.

✅ **Types of Parallelism**

1. **Data Parallelism**
   - Same operation is performed on different pieces of data.
   - Example: Adding two large arrays element-wise using multiple processors.
2. **Task Parallelism**
   - Different processors perform **different tasks** on the same or different data.
   - Example: One processor handles input, another processes data, a third displays output.

✅ **Classification of Parallel Processors (Flynn's Taxonomy)**

Type	Description
SISD	Single Instruction, Single Data – Traditional sequential processing
SIMD	Single Instruction, Multiple Data – Same instruction on multiple data (e.g., GPUs)
MISD	Multiple Instruction, Single Data – Rare, mostly theoretical
MIMD	Multiple Instruction, Multiple Data – Most common in modern multicore CPUs

✅ **Components of Parallel Processing Systems**

- **Multiple CPUs or Cores**
- **Shared or Distributed Memory**

- **Interconnection Network** (for communication)
- **Parallel Software/OS Support**

✅ **Benefits**
- High performance
- Better resource utilization
- Scalability for complex problems

✅ **Challenges**
- Synchronization of tasks
- Data sharing and consistency
- Load balancing
- Writing parallel algorithms

2. Concurrent access to memory and cache coherency

Concurrent Access to Memory and Cache Coherency

✅ **A. Concurrent Access to Memory**

In **multiprocessor or multicore systems**, multiple processors or threads may **access and modify shared memory** simultaneously. This leads to **concurrency issues**.

🔸 **Problems with Concurrent Access:**
- **Race Conditions:** Multiple processors try to read/write shared data at the same time, causing unpredictable results.
- **Inconsistent Data:** One processor may not see the latest value updated by another processor.

🔸 **Solutions:**
- **Locks / Semaphores / Mutexes**: Used to ensure **mutual exclusion**, allowing only one processor to access a memory region at a time.
- **Atomic Operations**: Ensure operations like increment or compare-and-swap happen **without interruption**.
- **Memory Barriers/Fences**: Prevent the CPU from **reordering** memory operations to ensure consistency.

B. Cache Coherency

✅ **What is Cache?**

Each processor/core typically has its own **local cache** (L1, L2) to reduce memory access time.

✅ **What is Cache Coherency?**

When multiple caches store copies of the **same memory location**, **coherency** ensures that **all caches reflect the most recent write**.

Cache Coherency Problem

Suppose:
- Core 1 reads variable x = 10 and stores it in its cache.
- Core 2 updates x = 20 in memory.
- Now Core 1's cache has **stale data** (x = 10), causing **inconsistency**.

Coherency Protocols

To solve this, systems use **cache coherence protocols**, such as:

🔹 **MESI Protocol (Modified, Exclusive, Shared, Invalid):**
- Ensures consistency by keeping track of the **state of each cache line**.
- Transitions between states when data is read, written, or shared between cores.

State	**Meaning**
Modified	Cache has updated data, not yet written to memory.
Exclusive	Only one cache has the data, same as main memory.
Shared	Multiple caches may have the same data.
Invalid	Data in cache is not valid anymore.

Hardware vs Software Coherency

Type	**Description**
Hardware Coherency	Managed by the hardware (cache controllers), automatic and fast.
Software Coherency	Managed by OS or compiler, more flexible but slower.

✅ **Summary**

Concept	**Description**
Concurrent Access	Multiple processors accessing shared memory at the same time
Problem	Race conditions, inconsistency
Solution	Locks, atomic ops, memory fences
Cache Coherency	Keeping cache data consistent across processors
Solution	Coherence protocols like MESI

Module 4: Memory organization

1. Memory interleaving

Memory Interleaving

✅ **Definition:**

Memory interleaving is a technique used to **increase the speed of memory access** by **dividing memory into multiple modules (banks)** and accessing them in parallel.
Instead of waiting for one memory access to complete before starting the next, interleaving allows **simultaneous or overlapped access** to different memory modules, improving performance.

✅ **How It Works:**
- Main memory is divided into **multiple banks** (e.g., Bank 0, Bank 1, Bank 2, ...).
- Consecutive memory addresses are **distributed across banks**.
- While one bank is busy handling a memory request, another bank can begin processing the next.

✅ **Example (4-way Interleaving):**

Address	**Memory Bank**
100	Bank 0
101	Bank 1
102	Bank 2
103	Bank 3
104	Bank 0
105	Bank 1
...	...

- Accessing 100 to 103 can be done **in parallel**, one per bank.

✅ **Types of Interleaving**
1. **Low-order Interleaving**
   - Uses **lower bits** of the address to select the memory bank.
   - Good for sequential access (e.g., arrays, loops).
2. **High-order Interleaving**
   - Uses **higher bits** to choose the memory module.
   - Better for random access or multiprocessor systems.

✅ **Advantages of Memory Interleaving**
- Increases memory **throughput**.
- Reduces CPU **waiting time** for memory.
- Improves performance in **multi-core** and **pipelined** systems.

✅ **Drawbacks**
- Adds complexity to memory management.
- Needs **synchronization** if multiple processors access shared memory.
- Hardware cost increases with more memory banks.

✅ **Summary Table**

Feature	**Description**
Purpose	Faster memory access
Method	Divide memory into multiple banks

Feature	Description
Benefit	Parallel access → increased throughput
Example	4-way interleaving → 4 addresses at once
Used In	High-performance CPUs, multicore systems

2. Concept of hierarchical memory organization

Concept of Hierarchical Memory Organization

✅ **Definition:**

Hierarchical memory organization is a system design approach in which **memory is structured in multiple levels** based on **speed, size, and cost**. This structure helps balance **performance** and **cost-efficiency** in a computer system.

✅ **Need for Hierarchical Memory:**

- Fast memory (like cache) is expensive and small.
- Large memory (like HDD) is cheap but slow.
- A hierarchy allows **frequent data** to be accessed faster from **upper levels**, while **infrequent data** is stored in **lower levels**.

✅ **Levels of Memory Hierarchy**

Level	Type	Speed	Size	Cost/Bit	Example
L0	Registers	Fastest	Smallest	Highest	CPU registers
L1	Cache (L1, L2, L3)	Very Fast	Small	High	Intel L1 cache
L2	Main Memory (RAM)	Medium	Medium	Medium	DDR4/ DDR5 RAM
L3	Secondary Storage	Slow	Large	Low	HDD, SSD
L4	Tertiary Storage	Slowest	Largest	Cheapest	Optical discs, Tape

✅ **Working Principle:**

- When the CPU needs data:
  1. It first checks **registers**.
  2. If not found, it checks **cache memory**.
  3. If not in cache, it goes to **main memory**.
  4. If still not found, it retrieves data from **secondary/tertiary storage**.

This process is called the **memory access hierarchy**, and each step down the hierarchy adds more **latency**.

✅ **Key Concepts:**

- **Locality of Reference:**
  - **Temporal locality**: Recently accessed data is likely to be accessed again soon.
  - **Spatial locality**: Nearby memory locations are likely to be accessed soon.
- **Caching:** Frequently used data from lower levels are stored temporarily in faster upper levels.
- **Hit**: Data found in the current level.
- **Miss**: Data not found and must be fetched from a lower level.

✅ **Advantages:**

- **High speed access** for frequently used data.
- **Lower overall cost** due to limited use of expensive memory.
- Efficient memory utilization.

3. Cache memory

Cache Memory

✅ **Definition:**

Cache memory is a small, fast memory unit placed between the **CPU and main memory (RAM)**. It stores **frequently accessed data and instructions** to speed up processing.

✅ **Purpose of Cache:**

- Reduces the **average time to access memory**.
- Improves CPU efficiency by minimizing delays in data retrieval from RAM.

✅ **Why Cache is Needed:**

- **CPU speed > RAM speed**.
- Without cache, the CPU would often be idle, waiting for data from the slower main memory.

✅ **Characteristics of Cache Memory:**

Feature	Description
Speed	Very fast (next to CPU registers)
Size	Small (KBs to a few MBs)
Cost	High per bit (SRAM-based)
Location	Inside or very close to the CPU
Access Time	Much shorter than RAM

✅ **Types of Cache:**

1. **L1 (Level 1) Cache:**
   - Closest to CPU core.
   - Smallest (~32KB to 128KB) and **fastest**.
   - Usually split into **instruction** and **data** caches.
2. **L2 (Level 2) Cache:**
   - Larger than L1 (256KB to several MBs).
   - Slower than L1 but **still faster than RAM**.
   - Can be shared among cores or dedicated per core.
3. **L3 (Level 3) Cache:**
   - Shared by all cores on the processor.
   - Largest and slowest among internal caches.
   - Improves communication among cores.

✅ **Cache Operations:**

- **Hit**: Data is found in cache → Fast access.
- **Miss**: Data not in cache → Fetched from RAM → Slower.

✅ **Cache Mapping Techniques:**

1. **Direct Mapping**:
   - Each memory block maps to **only one cache line**.
   - Simple but can cause frequent replacements.
2. **Associative Mapping**:
   - Memory block can be placed in **any cache line**.
   - Flexible but **complex and slower** to search.
3. **Set-Associative Mapping**:
   - Cache is divided into **sets**, and each block maps to a set.
   - Compromise between direct and associative mapping.

✅ **Write Policies:**

- **Write-through**: Data written to **both cache and main memory**.
- **Write-back**: Data written **only to cache**, and main memory is updated **later**.

✅ **Summary Table**

Feature	Description
Role	Stores frequently accessed data
Location	Between CPU and RAM
Speed	Faster than RAM, slower than registers
Types	L1, L2, L3
Mapping	Direct, Associative, Set-Associative
Miss/Hit	Determines data access efficiency

4. Cache size vs. block size

Cache Size vs. Block Size

This topic explores the relationship between the **total size of the cache** and the **size of each cache block (or line)**, and how these affect **performance**.

✅ **Definitions**

- **Cache Size:**
  Total amount of data the cache can hold.
  Example: 64 KB, 256 KB, etc.
- **Block Size (or Line Size):**
  The unit of data transferred between the main memory and the cache.
  Example: 16 bytes, 32 bytes, 64 bytes, etc.

✅ **Impact of Cache Size**

- **Larger Cache Size** means:
  - More data can be stored.
  - **Lower miss rate** (better performance).
  - Increased hardware cost and power consumption.
- But beyond a point, larger size may cause **slower access time** due to longer lookup time.

✅ **Impact of Block Size**

- **Smaller Block Size:**
  - More blocks → better cache utilization.
  - Lower miss penalty.
  - May lead to **more frequent misses** if spatial locality is high.
- **Larger Block Size:**
  - Fewer blocks in the same cache size → higher chance of **conflict misses**.
  - Good for programs with **high spatial locality** (access nearby data).
  - Wastes space (called **internal fragmentation**) if data isn't fully used.

✅ **Trade-offs**

Factor	**Small Block Size**	**Large Block Size**
Spatial Locality	Less utilized	Better utilized
Miss Rate	Moderate	May decrease (up to a point)
Miss Penalty	Lower	Higher
Cache Pollution	Less	More (if unused data fills cache)
Conflict Misses	Less	More (fewer blocks total)

✅ **Ideal Block Size?**

- Depends on workload and system.
- Typical range: **32 to 128 bytes**.
- Often chosen experimentally to balance **miss rate and transfer overhead**.

✅ **Summary Points:**

- Increasing **cache size** generally improves performance but at higher cost.
- Increasing **block size** helps only **up to a certain point**.
- There's a **sweet spot** where cache performs best with balanced cache and block sizes.

5. Mapping functions

Mapping Functions

✅ **Definition:**

Mapping functions are methods used to determine **where a block of main memory will be placed in the cache**.

Since cache is much smaller than main memory, mapping functions help decide:

- **Which cache line** will hold a memory block
- **How to find** a memory block during access

✅ **Types of Mapping Functions**

🔹 **1. Direct Mapping**

- **Simplest and fastest** method.
- Each block from main memory maps to **exactly one** location in the cache.

📌 **Formula:**

Cache Line Number = (Main Memory Block Number) %(Number of Cache Lines)

✅ **Pros:**

- Simple, fast to compute
- Low hardware cost

❌ **Cons:**

- High **conflict miss rate**
- Two frequently-used blocks may **keep replacing each other**

🔹 **2. Fully Associative Mapping**

- Any block of main memory can go into **any cache line**.
- Cache controller searches all lines (using tags) to find a match.

✅ **Pros:**

- **Lowest conflict misses**
- Best for small caches or critical data

❌ **Cons:**

- Expensive hardware (needs **content-addressable memory**)
- Slower access time due to full search

🔹 **3. Set-Associative Mapping**

- A **hybrid** of direct and fully associative.
- Cache is divided into **sets** (e.g., 2-way, 4-way).
- A block maps to a specific **set**, but can go in **any line** within that set.

📌 **Example:**

- In 4-way set-associative cache:
  - Each set has 4 lines
  - Block maps to one set using:

Set Number = (Block Number) mod (Number of Sets)

✅ **Pros:**

- **Lower conflict misses** than direct mapping
- **Less hardware cost** than fully associative

❌ **Cons:**

- Slightly more complex than direct mapping

✅ **Comparison Table:**

Mapping Type	**Placement Flexibility**	**Speed**	**Conflict Misses**	**Hardware Complexity**
Direct Mapping	One fixed cache line	Fast	High	Low
Fully Associative	Any line in the cache	Slow	Low	High
Set-Associative	Any line in a set	Moderate	Medium	Medium

6. Replacement algorithms

Replacement Algorithms

✅ **What are Replacement Algorithms?**

When the **cache** (or **main memory** in case of virtual memory) is full and a new block/page needs to be loaded, the system must **replace** an existing one.

Replacement algorithms decide **which block to remove** to make space for the new one.

✅ **Goals of Replacement Algorithms**

- Minimize **cache misses** or **page faults**
- Improve **system performance**
- Preserve **important/frequently used data**

✅ **Common Cache/Page Replacement Algorithms**

1. FIFO (First-In First-Out)

- Removes the block that entered the cache **first**.
- Simple, but may replace a frequently used block.

Pros: Easy to implement

Cons: Poor performance in some cases (e.g., Belady's anomaly)

2. LRU (Least Recently Used)

- Replaces the block that was **least recently used**.

- Assumes recently used data will likely be used again.

Pros: Good performance
Cons: Needs tracking of usage history (hardware/software support)

3. LFU (Least Frequently Used)

- Removes the block that has been used **least frequently**.
- Assumes data not used often will not be needed.

Pros: Good for repetitive access patterns
Cons: Hard to manage exact counts, may keep outdated blocks

4. Random Replacement

- Replaces a **randomly selected block**.
- Simple and fast, but not optimal.

Pros: Easy to implement
Cons: Unpredictable performance

5. Optimal (Theoretical)

- Replaces the block that will **not be used for the longest time in the future**.
- Not implementable in real systems, used for comparison.

✅ **Comparison Table**

Algorithm	**Strategy**	**Performance**	**Practical Use**
FIFO	Oldest block removed	Average	Yes
LRU	Least recently used block	Good	Yes (used in real systems)
LFU	Least frequently used	Good	Sometimes
Random	Random block removed	Unpredictable	Rare
Optimal	Future knowledge required	Best (ideal)	No (theoretical only)

✅ **Where Used?**

- **Cache replacement** (CPU cache)
- **Page replacement** (Virtual memory systems)
- **Disk cache**, **buffer management** in DBMS

7. Write policies

Write Policies

✅ **What are Write Policies?**

Write policies define how the system handles **write operations** to **cache and main memory** when the CPU updates data. These policies determine **when and where** the updated data is written.

✅ **Types of Write Policies**

🔹 **1. Write-Through**

- **Definition:**
  Data is **written to both cache and main memory** simultaneously.
- **Pros:**
  - Simple and ensures **data consistency**.
  - Main memory always has the **latest data**.
- **Cons:**
  - **Slower writes** due to double writing.
  - Higher memory traffic.
- **Use Case:**
  Systems where data consistency between cache and memory is critical.

🔹 **2. Write-Back (Copy-Back)**

- **Definition:**
  Data is **only written to the cache**.
  The modified block is written to main memory **only when it is replaced** (evicted).
- **Pros:**
  - **Faster write speed**.
  - Reduces memory write operations.
- **Cons:**
  - Data in memory may become **stale**.
  - Requires a **dirty bit** to track modified blocks.
- **Use Case:**
  Performance-critical systems with good cache management.

✅ **Write Miss Policies**

When the data to be written is **not found in the cache**, we have two options:

🔹 **a) Write Allocate (Fetch on Write):**

- Block is loaded into cache on a write miss.
- Followed by a write to the cache.
- Often used with **write-back**.

🔹 **b) No Write Allocate (Write No-Allocate):**

- Data is **written directly to main memory**, not loaded into cache.
- Often used with **write-through**.

✅ **Summary Table**

Policy Type	**Description**	**Cache Updated?**	**Memory Updated?**	**Use With**
Write-Through	Write to both cache and memory	Yes	Yes	Write-No-Allocate
Write-Back	Write only to cache, update memory later	Yes	Delayed (on eviction)	Write-Allocate

✅ **Real-World Relevance**

Modern CPUs may use **a combination** of these policies depending on the cache level (L1, L2, etc.) and system requirements (performance vs. consistency).