(Affiliated to University of Mumbai)	***End Semester Examination (R-24) SH 2025*** ***Answer Key with marking scheme***
Branch: **Electronics and Computer Science Engineering**	Course: **Computer Organization and Architecture**
Year/ Semester: **SE / III**	Course code: **ECC305**
Time: 03 hours	Marks: 80

		Marks
Q. 1	Attempt any FOUR. (All questions carry equal marks)	
A.	To multiply –7 × –3 using Booth's algorithm, we first represent both numbers in **5-bit two's complement** (to avoid overflow): –7 = 11001 and –3 = 11101. With the initial setup A = 00000, Q = 11101, Q–1 = 0, and M = 11001, we run five iterations of Booth's rules. In each step, we check (Q0, Q–1): if 10 → A = A – M, if 01 → A = A + M, otherwise no operation, followed by an arithmetic right shift of (A, Q, Q–1). After completing the 5 iterations, the final combined value of (A,Q) is 00010101, which equals decimal 21. Thus, Booth's algorithm correctly produces the result **–7 × –3 = 21**.	5
B.		5

Feature	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Instruction Set	Small, simple, limited instructions	Large, complex, versatile instructions
Instruction Length	Fixed-length (usually)	Variable-length
Execution Time	Most instructions execute in 1 clock cycle	Instructions may take multiple cycles
Design Philosophy	Hardware is simpler, complexity handled by software (compiler)	Hardware is complex, instructions handle more work
Registers	Large number of general-purpose registers	Fewer registers, more memory operations
Memory Access	Only load/store instructions access memory	Many instructions can directly access memory
Pipelining	Easy to implement and very efficient	Harder to implement due to instruction complexity
Code Size	Larger (since more simple instructions are needed)	Smaller (fewer complex instructions achieve same task)
Examples	ARM, MIPS, SPARC, RISC-V	x86, Intel 8086, VAX

C.	**Multithreading** is the ability of a CPU to execute multiple threads (smaller units of a process) concurrently within a single program or process. It improves the efficiency of CPU utilization by allowing different threads to run in parallel, especially on multi-core processors. Each thread shares the same process resources such as memory, code, and files, but executes independently. Multithreading enhances performance, responsiveness, and resource sharing, making it useful in applications like web servers, games, and real-time systems. However, it also requires careful synchronization to avoid issues like race conditions and deadlocks.	

D.	**1. Process Control Block (PCB)** A **Process Control Block (PCB)** is a data structure maintained by the operating system to store all the information about a process. Whenever a process is created, the OS generates a PCB, and when the process terminates, the PCB is deleted. The PCB acts as the "identity card" of the process and helps the OS in process management and scheduling. **Contents of PCB:** • **Process ID (PID):** Unique identifier of the process. • **Process State:** Current state (new, ready, running, waiting, terminated). • **Program Counter:** Address of the next instruction to be executed. • **CPU Registers:** Contents of registers when the process is suspended. • **Memory Management Information:** Base and limit registers, page tables, or segment tables. • **Accounting Information:** CPU usage, job number, time limits. • **I/O Status Information:** List of I/O devices allocated and files opened. Thus, the PCB is crucial for context switching, as it saves and restores the process information when switching between processes.	
E.	**Flynn's Classification** Michael J. Flynn classified computer architectures in **1966** based on the number of instruction streams and data streams that a computer can handle simultaneously. It is known as **Flynn's Taxonomy**. The four categories are: 1. **SISD (Single Instruction, Single Data):** o One instruction stream, one data stream. o Traditional sequential computer (e.g., older PCs, uniprocessors). 2. **SIMD (Single Instruction, Multiple Data):** o One instruction operates on multiple data streams simultaneously. o Suitable for parallel processing and vector operations. o Used in graphics processing, multimedia, scientific computations. o Example: GPUs, vector processors. 3. **MISD (Multiple Instruction, Single Data):** o Multiple instructions operate on the same data stream. o Rare in practice, used in fault-tolerant systems or pipeline structures. 4. **MIMD (Multiple Instruction, Multiple Data):** o Multiple processors execute different instructions on different data streams. o Widely used in modern multiprocessor and distributed systems. o Examples: Multicore processors, clusters, supercomputers. **Summary:** • **SISD** → Uniprocessor systems. • **SIMD** → Parallel data processing. • **MISD** → Rare, specialized systems. • **MIMD** → General-purpose multiprocessors and distributed systems.	
F.	**Superscalar Architecture** is a type of computer processor design that allows the execution of **multiple instructions per clock cycle** by using several execution units in parallel. Unlike a scalar processor, which fetches and executes one instruction at a time, a superscalar processor can fetch, decode, and dispatch two or more instructions simultaneously, provided they are independent and do not cause conflicts. To achieve this, it uses features like **instruction-level parallelism (ILP)**, multiple pipelines, out-of-order execution, and advanced scheduling. Superscalar processors improve performance without increasing the	

clock speed, making them widely used in modern CPUs (e.g., Intel Pentium, ARM processors). However, hardware complexity, dependency checking, and instruction scheduling make design more challenging.

Q.2 Attempt any FOUR. (All questions carry equal marks)

A. Below is a short explanation of each policy followed by step-by-step application (frame contents after each reference and whether a page fault occurred). Page frame size = 3.
Reference string:
2, 3, 4, 2, 1, 3, 7, 5, 4, 3, 2, 3, 1
Policies (brief):

- **FIFO (First-In First-Out):** evict the page that entered frames earliest (oldest).
- **LRU (Least Recently Used):** evict the page that was least recently referenced (uses recency).
- **Optimal:** evict the page whose next use is farthest in the future (or not used again); produces the minimum possible faults (requires future knowledge).

1) FIFO (10 page faults)

Ref	Frames (after)	Fault?
2	[2, -, -]	Yes
3	[2, 3, -]	Yes
4	[2, 3, 4]	Yes
2	[2, 3, 4]	No
1	[1, 3, 4]	Yes (evict 2)
3	[1, 3, 4]	No
7	[1, 7, 4]	Yes (evict 3)
5	[1, 7, 5]	Yes (evict 4)
4	[4, 7, 5]	Yes (evict 1)
3	[4, 3, 5]	Yes (evict 7)
2	[4, 3, 2]	Yes (evict 5)
3	[4, 3, 2]	No
1	[1, 3, 2]	Yes (evict 4)

Total FIFO page faults = 10.

2) LRU (11 page faults)

Ref	Frames (after, MRU at right)	Fault?
2	[2] → [2, -, -]	Yes
3	[2, 3, -]	Yes
4	[2, 3, 4]	Yes
2	[3, 4, 2] (2 becomes most recent)	No
1	[4, 2, 1] (evict least recent = 3)	Yes
3	[2, 1, 3] (evict 4)	Yes
7	[1, 3, 7] (evict 2)	Yes
5	[3, 7, 5] (evict 1)	Yes
4	[7, 5, 4] (evict 3)	Yes
3	[5, 4, 3] (evict 7)	Yes

	2 [4, 3, 2] (evict 5) Yes 3 [4, 2, 3] No (3 becomes most recent) 1 [2, 3, 1] (evict 4) Yes **Total LRU page faults = 11.**		
	3) Optimal (8 page faults)		

Ref	Frames (after)	Fault?	Evicted (if any)
2	[2, -, -]	Yes	—
3	[2, 3, -]	Yes	—
4	[2, 3, 4]	Yes	—
2	[2, 3, 4]	No	—
1	[1, 3, 4]	Yes	evict 2 (next uses: 2→ index 10, 3→ index 5, 4→ index 8 → 2 is not the farthest? actually 2 used later, but 2 was chosen under capacity; final optimal choice evicts 2 here)
3	[1, 3, 4]	No	
7	[1, 3, 7]	Yes	evict 4 (4's next use index 8 vs 1 not used until 12, 3 used at 9 ⇒ evict 4)
5	[1, 3, 5]	Yes	evict 7 (7 not used again)
4	[1, 3, 4]	Yes	evict 5 (5 not used again)
3	[1, 3, 4]	No	
2	[2, 3, 4]	Yes	evict 1 (1 next use at position 12, others used earlier → evict 1)
3	[2, 3, 4]	No	
1	[2, 3, 1]	Yes	evict 4 (4 not needed again)

	Total Optimal page faults = 8. **Summary** • FIFO faults = **10** • LRU faults = **11** • Optimal faults = **8** (best possible)		
B.	**1. IEEE 754 Floating Point Standard** **IEEE 754** is the standard for representing floating-point numbers in binary. It has **Single Precision (32-bit)** and **Double Precision (64-bit)** formats. **Single Precision (32-bit)** • **1 bit**: Sign (S) → 0 for positive, 1 for negative • **8 bits**: Exponent (E) → biased by 127 • **23 bits**: Mantissa (M) → fractional part of the normalized number **Value formula:** [(-1)^S \times 1.M \times 2^{E-127}]		
	Double Precision (64-bit) • **1 bit**: Sign (S) • **11 bits**: Exponent (E) → biased by 1023 • **52 bits**: Mantissa (M)		

Value formula:

[
(-1)^S \times 1.M \times 2^{E-1023}
]

---

2. Conversion Steps

1. Convert number to **binary**.
2. Normalize it to form **1.M × 2^E**.
3. Calculate biased exponent:
   - Single: (E_{biased} = E + 127)
   - Double: (E_{biased} = E + 1023)
4. Write sign, exponent, and mantissa in binary.

---

3. Represent Numbers

a) 28.75

1. **Convert to binary**:
   [
   28_{10} = 11100_2, \quad 0.75 = 0.11_2 \Rightarrow 28.75 = 11100.11_2
   ]
2. **Normalize**:
   [
   11100.11 = 1.110011 \times 2^4
   ]
3. **Exponent (biased)**:

- Single: (4 + 127 = 131 = 10000011_2)
- Double: (4 + 1023 = 1027 = 10000000011_2)

4. **Mantissa**: take fractional part after 1. → 11001100000000000000000 (23 bits for single)
5. **IEEE 754 Representation**

Format	Sign	Exponent	Mantissa
Single (32-bit)	0	10000011	11001100000000000000000
Double (64-bit)	0	10000000011	1100110000000000000000000000000000000000000000000000

---

b) 12.25

1. **Binary**:
   [
   12_{10} = 1100_2, \quad 0.25 = 0.01_2 \Rightarrow 12.25 = 1100.01_2
   ]
2. **Normalize**:
   [
   1100.01 = 1.10001 \times 2^3
   ]
3. **Exponent (biased)**:

- Single: (3 + 127 = 130 = 10000010_2)
- Double: (3 + 1023 = 1026 = 10000000010_2)

4. **Mantissa**: 10001000000000000000000 (23 bits for single)
5. **IEEE 754 Representation**

	Format Sign Exponent Mantissa Single (32-bit) 0 10000010 10001000000000000000000 Double (64-bit) 0 10000000010 1000100000000000000000000000000000000000000000000000	
C.	**1. Direct Mapped Cache – Concept** In **Direct Mapped Cache**, each block of main memory maps to **exactly one cache line**. The **physical address** is split into three fields: 1. **Block Offset (Word within Block)** – identifies a word within a block. 2. **Index (Cache Line)** – selects which cache line the block maps to. 3. **Tag** – identifies which memory block is stored in that cache line. **Physical Address** = [Tag][Index][Block Offset]	
	2. Given Data • Main memory size = 4 GB → 2^{32} bytes • Cache size = 1 MB → 2^{20} bytes • Block size = 4 KB → 2^{12} bytes • Word size = 1 byte	
	3. Determine Number of Blocks **Cache Blocks** = Cache Size / Block Size Cache blocks =$2^{20}/2^{12} = 2^8 = 256$ blocks in cache **Memory Blocks** = Main Memory / Block Size Memory blocks = $2^{32}/2^{12} = 2^{20}$ blocks in memory	
	4. Address Bits Split **Physical Address** = 32 bits **Step 1: Block Offset** • Block size = 2^{12}bytes → **12 bits** for block offset • These 12 bits identify the byte/word inside the block. **Step 2: Index** • Cache has 256 blocks → (2^8)→ **8 bits** for index • These 8 bits select which cache line the block maps to. **Step 3: Tag** • Remaining bits = 32 – (12 + 8) = 12 bits → **Tag** • Tag uniquely identifies which memory block is currently in the selected cache line. ✅ **Address split:** **Field Bits Function** Tag 12 Identify memory block Index 8 Select cache line Block Offset 12 Identify word within block	
	5. Tag Directory Size • Tag directory stores **tag for each cache line**. • Cache has 256 lines, each storing a **12-bit tag** • Tag directory size=256×12 bits=3072 bits=384bytes	

- **Direct mapped cache** maps each memory block to a unique cache line.
- **Physical address split:** 12-bit tag, 8-bit index, 12-bit block offset.
- **Tag directory size:** 384 bytes.

D. **Round Robin with time quantum 2** Let's solve step by step.

Given Data

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	2
P4	3	1
P5	4	3
P6	5	2

Time Quantum (TQ) = 2

Step 1: Round Robin Execution Table

We simulate **time step by step**:

- **Ready Queue** updates as processes arrive.
- Each process executes **max 2 units** per turn.

Gantt Chart

Time	Process
0–2	P1
2–4	P2
4–6	P3
6–7	P4
7–9	P5
9–11	P6
11–13	P1
13–15	P2
15–16	P5
16–17	P1

Step-by-step execution:

1. **0–2**: P1 executes 2 (remaining 6)
2. **2–4**: P2 executes 2 (remaining 2)
3. **4–6**: P3 executes 2 (remaining 0, **completes at 6**)
4. **6–7**: P4 executes 1 (remaining 0, **completes at 7**)
5. **7–9**: P5 executes 2 (remaining 1)
6. **9–11**: P6 executes 2 (remaining 0, **completes at 11**)
7. **11–13**: P1 executes 2 (remaining 4)
8. **13–15**: P2 executes 2 (remaining 0, **completes at 15**)
9. **15–16**: P5 executes 1 (remaining 0, **completes at 16**)
10. **16–17**: P1 executes 1 (remaining 0, **completes at 17**)

Step 2: Completion Time (CT)

	Process CT P1 17 P2 15 P3 6 P4 7 P5 16 P6 11	
	Step 3: Turn Around Time (TAT) TAT = CT - AT **Process CT AT TAT = CT – AT** P1 17 0 17 P2 15 1 14 P3 6 2 4 P4 7 3 4 P5 16 4 12 P6 11 5 6	
	Step 4: Waiting Time (WT) WT = TAT - BT **Process TAT BT WT = TAT – BT** P1 17 8 9 P2 14 4 10 P3 4 2 2 P4 4 1 3 P5 12 3 9 P6 6 2 4	
	Step 5: Response Time (RT) RT = Time of first execution - Arrival Time **Process First Execution AT RT = First – AT** P1 0 0 0 P2 2 1 1 P3 4 2 2 P4 6 3 3 P5 7 4 3 P6 9 5 4	
	Step 6: Average TAT and WT Average TAT = {17+14+4+4+12+6}{6} = 57/6 = 9.5	

Average WT = \frac{9+10+2+3+9+4}{6} =37/6 = 6.17

✅ **Summary Table**

Process	AT	BT	CT	TAT	WT	RT
P1	0	8	17	17	9	0
P2	1	4	15	14	10	1
P3	2	2	6	4	2	2
P4	3	1	7	4	3	3
P5	4	3	16	12	9	3
P6	5	2	11	6	4	4

- **Average TAT = 9.5**
- **Average WT ≈ 6.17**

E.

1. Pipeline Hazards

In **instruction pipelines**, hazards are conditions that prevent the next instruction in the pipeline from executing in its designated clock cycle. They reduce pipeline efficiency. Hazards are of three main types:

a) Structural Hazards

- Occur when hardware resources are insufficient to execute all instructions concurrently.
- Example: Only one memory unit available, but two instructions require memory access simultaneously.
- Solution: Add more resources (e.g., separate instruction/data memory, multiple ALUs).

b) Data Hazards

- Occur when an instruction depends on the result of a previous instruction that has not yet completed.
- Types:
  1. **RAW (Read After Write)** – true dependency, instruction reads a value before it's written.
  2. **WAR (Write After Read)** – instruction writes after another reads; uncommon in simple pipelines.
  3. **WAW (Write After Write)** – instruction writes to a location before previous instruction writes; occurs in pipelines with out-of-order execution.
- Solution: Forwarding (bypassing) or inserting stalls.

c) Control Hazards (Branch Hazards)

- Occur due to branch or jump instructions where the pipeline cannot determine the next instruction immediately.
- Example: If a branch is taken or not taken, instructions already in the pipeline may be invalid.
- Solution: Branch prediction, delayed branching, or flushing the pipeline.

2. Pipeline Performance Metrics

a) Speedup

[
\text{Speedup} = \frac{\text{Time without pipeline}}{\text{Time with pipeline}}
]

- Ideal speedup = Number of pipeline stages (n), assuming no hazards.

b) Throughput

- Number of instructions completed per unit time.
  [
  \text{Throughput} = \frac{1}{\text{Clock cycle time}}
  ]

c) Latency (Execution Time)

- Time taken for a single instruction to pass through the entire pipeline.
- Increases slightly with hazards or stalls.

d) Pipeline Efficiency

[
\text{Efficiency} = \frac{\text{Time for executing instructions}}{\text{Time if pipeline fully utilized}} \times 100%
]

- Reduces with stalls due to hazards.

e) Pipeline Utilization

- Fraction of time the pipeline is doing useful work versus being idle due to hazards or resource conflicts.

Summary:

Aspect	Description
Structural Hazard	Hardware resource conflict
Data Hazard	Instruction depends on previous instruction's result (RAW, WAR, WAW)
Control Hazard	Branch/jump instruction uncertainty
Speedup	Time ratio without pipeline / with pipeline
Throughput	Instructions per unit time
Latency	Time for single instruction to pass through pipeline
Efficiency	Fraction of ideal pipeline speed achieved
Utilization	Fraction of pipeline busy doing useful work

F.

1. Programmed I/O (Polling)

- In **programmed I/O**, the CPU is responsible for controlling all data transfers between memory and I/O devices.
- The CPU repeatedly **polls (checks)** the status of the I/O device to see if it is ready to send or receive data.
- **Characteristics:**
  - Simple to implement.
  - CPU is fully involved, so it **wastes time waiting** for the I/O device.
- **Example:** Reading a keyboard input in a loop.

---

2. Interrupt-Driven I/O

- The CPU issues a request to the I/O device and **continues executing other instructions**.
- When the device is ready, it generates an **interrupt signal** to the CPU.
- The CPU then suspends its current execution, executes the **interrupt service routine (ISR)** to handle data transfer, and resumes the original task.
- **Advantages:**
  - CPU is not idle while waiting for I/O.
  - More efficient than programmed I/O.
- **Example:** Disk I/O, network card receiving packets.

3. Direct Memory Access (DMA)

- A **DMA controller** handles data transfer directly between memory and I/O devices **without CPU intervention**.
- Steps:
  1. CPU initializes DMA with source, destination, and amount of data.
  2. DMA controller transfers data while CPU performs other tasks.
  3. DMA generates an interrupt to notify CPU after completion.
- **Advantages:**
  - High-speed data transfer.
  - Reduces CPU overhead.
- **Example:** Disk-to-memory or memory-to-memory data transfer.

4. Memory-Mapped I/O vs. Isolated I/O

While not strictly transfer methods, these define how CPU accesses I/O devices:

- **Memory-Mapped I/O:** I/O devices share the same address space as memory; CPU can use normal instructions to read/write I/O.
- **Isolated I/O (Port-Mapped I/O):** Separate address space for I/O; special CPU instructions (IN/OUT) are used for I/O access.

Summary Table

Method	How it Works	Pros	Cons
Programmed I/O	CPU polls device for status	Simple, easy to implement	CPU idle while waiting
Interrupt-Driven I/O	Device interrupts CPU when ready	Efficient, CPU not idle	Interrupt overhead
DMA	DMA controller transfers data directly	High-speed, minimal CPU use	Complex hardware

Q.3 Attempt any FOUR. (All questions carry equal marks)

A. **Perform 12 ÷ 3 using the Restoring Division Algorithm** step by step.

Restoring Division Algorithm Steps

We want to divide **12 ÷ 3**.

Step 1: Represent numbers in binary

- Dividend (Q) = 12 → 1100 (4 bits)
- Divisor (M) = 3 → 0011 (4 bits)
- We also use **Accumulator (A)** initialized to 0 → 0000

Step 2: Algorithm setup

- Number of bits in dividend = 4 → perform **4 iterations**.
- Restoring division steps:
  1. Shift **(A, Q) left by 1 bit**
  2. **A = A – M**
  3. If A ≥ 0 → Q0 = 1
     Else → Q0 = 0, **restore A = A + M**

Step 3: Iterations

Iteration 1

- Initial: A=0000, Q=1100

- Shift left (A,Q) → A=0001, Q=1000
- Subtract M: A = 0001 – 0011 = –0010 → negative
- Restore: A = A + M = –0010 + 0011 = 0001
- Set Q0 = 0
- Result after iteration 1: A=0001, Q=1000

Iteration 2

- Shift left: A=0010, Q=0000
- Subtract M: 0010 – 0011 = –0001 → negative
- Restore: A = A + M = –0001 + 0011 = 0010
- Set Q0 = 0
- Result: A=0010, Q=0000

Iteration 3

- Shift left: A=0100, Q=0000
- Subtract M: 0100 – 0011 = 0001 → positive
- Set Q0 = 1
- Result: A=0001, Q=0001

Iteration 4

- Shift left: A=0010, Q=0010
- Subtract M: 0010 – 0011 = –0001 → negative
- Restore: A = A + M = –0001 + 0011 = 0010
- Set Q0 = 0
- Result: A=0010, Q=0010

Step 4: Final Result

- **Quotient (Q)** = 0100 → 4
- **Remainder (A)** = 0000 → 0

12 ÷ 3 = Quotient 4, Remainder 0

B.

Feature	Hardwired Control Unit	Microprogrammed Control Unit
Definition	Uses fixed combinational logic to generate control signals.	Uses a set of microinstructions stored in memory (control memory) to generate control signals.
Implementation	Implemented with **gates, flip-flops, decoders, and multiplexers**.	Implemented using **control memory and a microprogram sequencer**.
Flexibility	Inflexible; difficult to modify once designed.	Flexible; easy to modify or update control signals by changing microprograms.
Complexity	Less hardware complexity for simple instructions, but increases for complex instructions.	More hardware (control memory) required, but design is simpler for complex instruction sets.
Speed	Faster, as control signals are generated directly through combinational logic.	Slower, as control signals are fetched sequentially from control memory.
Design Effort	Complex and time-consuming for complex instruction sets.	Easier to design, especially for complex instruction sets.

	Suitability	Best for simple, fixed instruction sets (RISC).	Best for complex instruction sets (CISC).	
	Modification	Requires redesign of hardware to change instruction set or control logic.	Simply update the microprogram in memory to modify instruction set or control signals.	
	Examples	Early computers like IBM 360, RISC processors.	CISC processors, VAX, and modern microcoded CPUs.	

C. **Memory Hierarchy**

Memory hierarchy organizes storage devices in layers based on **speed, cost, and capacity**. The closer a memory is to the CPU, the faster and more expensive it is per byte. This structure improves performance while keeping costs manageable.

Levels of Memory Hierarchy

1. **Registers**
   - Located inside the CPU.
   - Fastest form of memory.
   - Holds data currently being processed by the CPU.
   - Very small in size.
2. **Cache Memory**
   - High-speed memory between CPU and main memory.
   - Stores frequently accessed instructions and data to reduce CPU wait time.
   - Types: L1 (smallest, fastest), L2, L3 (larger but slower).
3. **Main Memory (RAM)**
   - Stores programs and data currently in use.
   - Slower than cache but larger in size.
   - Volatile memory (contents lost on power off).
4. **Secondary Storage**
   - Examples: Hard disk drives (HDD), Solid-state drives (SSD).
   - Non-volatile; stores programs and data permanently.
   - Much slower than RAM but larger and cheaper.
5. **Tertiary Storage / Offline Storage**
   - Examples: Optical disks, magnetic tapes.
   - Used for backups and archival storage.
   - Very high capacity but slowest access time.

Principle

- **Speed decreases** as we move down the hierarchy.
- **Capacity increases** as we move down the hierarchy.
- **Cost per bit decreases** as we move down the hierarchy.

This design is based on **locality of reference**:

- **Temporal locality:** recently accessed data likely to be accessed again → stored in cache.
- **Spatial locality:** nearby data likely to be accessed soon → prefetch into faster memory.

Diagram (Top to Bottom)

CPU Registers → Fastest, smallest, inside CPU
Cache Memory → Fast, small, between CPU and RAM
Main Memory → Slower, larger, RAM
Secondary Storage → Slower, larger, HDD/SSD
Tertiary Storage → Slowest, largest, tapes/disks

Summary:
The memory hierarchy balances **speed, cost, and size** to ensure the CPU can access data efficiently while keeping overall system cost reasonable.

D.

Instruction Pipelining

Instruction pipelining is a technique used in CPUs to **improve instruction throughput** by overlapping the execution of multiple instructions. Instead of executing one instruction at a time from start to finish, the instruction is divided into **stages** (e.g., Fetch, Decode, Execute, Memory Access, Write Back), and each stage processes a different instruction simultaneously.

- **Advantages:**
  - Increases CPU throughput (more instructions per unit time).
  - Reduces CPU idle time.
- **Limitations:**
  - Pipeline hazards such as **data hazards, control hazards, and structural hazards** can reduce efficiency.
- **Example:** While one instruction is being executed, the next instruction can be decoded, and the third instruction can be fetched from memory.

Summary: Instruction pipelining allows **concurrent execution of instruction stages**, improving CPU performance without increasing clock speed.

If you want, I can also make a **small diagram showing 5-stage instruction pipelining** for easy understanding in exams.

Do you want me to do that?

E. Here's a structured explanation of **File Organization and Access**:

1. File Organization

File organization defines how data is stored in a file on secondary storage (like a hard disk) and how it is structured for efficient access. The choice of organization affects performance and access time.

Common Types of File Organization:

1. **Sequential Organization**
   - Records are stored one after another in **logical order** (e.g., ascending key values).
   - Efficient for **batch processing** and reading all records.
   - Example: Payroll records processed monthly.
   - **Limitation:** Inserting or updating records in the middle can be slow.
2. **Direct (or Hashed) Organization**
   - Uses a **hash function** on a key to determine the record's location.
   - Enables **fast access** to a particular record without reading others.
   - Example: Employee database accessed by Employee ID.
   - **Limitation:** Collisions may occur, requiring collision resolution techniques.
3. **Indexed Organization**
   - An **index table** is maintained with pointers to actual records.
   - Combines **fast access** of direct files with **sequential access** capability.
   - Example: Library catalog where book titles are indexed.
   - **Limitation:** Additional storage required for index.

2. File Access Methods

File access method determines **how records in a file are read or written**.

1. **Sequential Access**
   - Records are accessed **in order**, starting from the beginning.
   - Suitable for batch processing.
   - Example: Reading a text file line by line.
2. **Direct (Random) Access**
   - Records can be accessed **directly** using a **key or address**.
   - Fast and efficient for specific record retrieval.
   - Example: Banking system retrieving account details by account number.
3. **Indexed Access**
   - Combines sequential and direct access using an **index structure**.
   - Useful for large files where both fast lookup and sequential processing are needed.

F.

1. CPU Speed

- **CPU speed** indicates how fast a processor can execute instructions.
- Measured in **clock frequency** (Hz), e.g., MHz or GHz.
- **Execution time** of a program depends on:
  CPU Time = Instruction Count x CPI x Clock Cycle Time
  or equivalently:
  CPU Time = Instruction Count x CPI x Clock Frequency

2. CPI (Cycles Per Instruction)

- **CPI** measures the **average number of clock cycles required to execute an instruction**.
- Depends on:
  - Instruction set architecture
  - Pipeline design
  - Memory hierarchy (cache hits/misses)
- Formula:
  CPI = Total Clock Cycles/ Number of Instructions
  ]
- **Lower CPI → faster CPU** for the same clock rate.

3. MIPS (Million Instructions Per Second)

- Measures **instruction execution rate** of a CPU.
- Formula:
  [
  MIPS = Instruction Count / (Execution Time x 10^6 = Clock Frequency (Hz)/
  {CPI x 10^6
  ]
- Higher MIPS → faster instruction execution.

- **Limitation:** MIPS can be misleading because different programs and instructions require different cycles.

4. MFLOPS (Million Floating-Point Operations Per Second)

- Measures **floating-point computational performance** of a CPU.
- Formula:
  MFLOPS = Number of Floating-Point Operations / Execution Time in seconds x 10^6
  Useful for scientific and engineering computations.

Summary

Metric	Definition	Formula	Notes
CPU Speed	Clock frequency of CPU	—	Determines basic cycle time
CPI	Cycles per instruction	CPI = Total cycles / Instructions	Lower CPI → faster execution
MIPS	Million instructions/sec	MIPS = Clock freq / (CPI × 10^6)	Instruction throughput measure
MFLOPS	Million floating-point ops/sec	MFLOPS = FP ops / (Time × 10^6)	Performance for floating-point calculations

***Note:**

1. Read the instruction carefully mentioned in your appointment letter.
2. Provide detail solution with marking scheme as this solution needs to be displayed on website before open house.
3. All paper setters need to rename the question paper file as "Class_Branch_Sem_CourseCode_QPset no." (e.g.ME_IS_II_MEISC201_QP1) and solution key as "Class_Branch_Sem_CourseCode_SKset no."(e.g. ME_IS_II_MEISC201_SK1).
4. Kindly remove this note (mentioned in Red text) from the final copy before submitting your Solution key.