MAA OMWATI DEGREE COLLEGE HASSANPUR(PALWAL)

IMPORTANT NOTES

OF

COMPUTER SYSTEM ARCHITECTURE

CLASS – BCA 4TH SEM

BASIC COMPUTER CONCEPTS

Unit-1

1. Analog Computer

An **analog computer** is a type of computer that operates on **continuous physical quantities** such as temperature, pressure, voltage, speed, or current. Unlike digital computers, analog computers do not process discrete numerical values but work with continuously varying data. The output of an analog computer is generally in the form of **graphs, meter readings, or dial movements**, which provide approximate results rather than exact values. These computers are mainly used in situations where **real-time processing** is required. Analog computers are fast because they do not need conversion of data into binary form. However, their accuracy is limited and data storage is difficult. Programming an analog computer is also complex and inflexible. Common examples include thermometers, speedometers, analog voltmeters, and early flight simulators. Analog computers are mostly used in scientific research, engineering applications, and industrial process control. Due to advancements in digital technology, analog computers are now rarely used.

Characteristics

- Works with **continuous data**
- Results are **approximate**, not exact
- Output is represented by **graphs, dials, or meters**
- Less accurate compared to digital computers
- Faster for specific real-time simulations

Examples

- Speedometer
- Thermometer
- Analog voltmeter
- Aircraft flight simulators (early versions)

Applications

- Weather forecasting
- Scientific and engineering simulations
- Industrial process control
- Medical instruments (ECG machines)

Advantages

- Fast real-time processing
- Useful for complex physical simulations

Disadvantages

- Low accuracy
- Difficult to store data
- Limited programmability

---

2. Digital Computer ==

A **digital computer** is a computing device that processes data in **discrete form**, using binary digits **0 and 1**. It converts all types of input—numbers, text, images, and sound—into binary form before processing. Digital computers are highly accurate, reliable, and programmable, making them the most widely used computers today. These computers follow a sequence of instructions stored in memory to perform operations. Digital computers include desktops, laptops, smartphones, tablets, and servers. They are used in almost every field such as education, business, healthcare, banking, communication, and entertainment. Digital computers have large storage capacity and can process vast amounts of data at high speed. They are also capable of error detection and correction. The major advantage of digital computers is their flexibility and precision. However, they require conversion of real-world data into digital form using input devices and depend completely on electrical power.

Characteristics

- High accuracy
- Data is processed as numbers and symbols
- Easy storage and retrieval of data
- Programmable and versatile

Examples

- Desktop computers
- Laptops
- Smartphones
- Tablets
- Servers

Applications

- Business and banking
- Education
- Scientific research
- Communication
- Entertainment

Advantages

- High speed and accuracy
- Large storage capacity
- Easy error detection
- Flexible and reliable

Disadvantages

- Requires digital conversion of physical data
- Dependent on electricity

---

3. Functional Units of a Digital Computer ==

A digital computer consists of several **functional units** that work together to perform operations. The main functional units are the **Input Unit, Output Unit, Memory Unit, and Central Processing Unit (CPU)**. The input unit accepts data and instructions from the user and converts them into machine-readable form. The output unit displays the processed results in human-readable form. The memory unit stores data, instructions, and intermediate results. It is divided into primary memory (RAM and ROM) and secondary memory (hard disk, SSD, pen drive). The CPU is the most important unit and is known as the brain of the computer. It consists of the **Arithmetic Logic Unit (ALU)**, which performs arithmetic and logical operations, the **Control Unit (CU)**, which controls all activities of the computer, and **registers**, which provide high-speed storage. These functional units together ensure smooth and efficient computer operation.

3.1 Input Unit

- Accepts data and instructions from the user
- Converts them into machine-readable form

Examples: Keyboard, Mouse, Scanner

3.2 Output Unit

- Produces results in human-readable form

Examples: Monitor, Printer, Speaker

3.3 Memory Unit

Primary Memory

- RAM (Random Access Memory)
- ROM (Read Only Memory)

Secondary Memory

- Hard Disk
- SSD
- Pen Drive
- CD/DVD

3.4 Central Processing Unit (CPU)

The **CPU** is the brain of the computer.

Components of CPU

1. **Arithmetic Logic Unit (ALU)**
    - Performs arithmetic operations (+, −, ×, ÷)
    - Performs logical operations (AND, OR, NOT, comparisons)
2. **Control Unit (CU)**
    - Directs and coordinates operations
    - Controls instruction execution
3. **Registers**
    - Small, high-speed memory inside CPU
    - Stores instructions and data temporarily

---

4. Basic Organizational Concepts ==

Basic organizational concepts describe how a computer system operates internally to execute programs efficiently. One of the most important concepts is the **stored program concept**, where both data and instructions are stored in the main memory. The computer executes instructions sequentially unless a control instruction changes the flow. Another key concept is the **instruction cycle**, which consists of four steps: fetch, decode, execute, and store. In the fetch phase, the instruction is retrieved from memory. During decode, the instruction is interpreted by the control unit. In the execute phase, the operation is performed by the ALU or other units. Finally, the result is stored back in memory. Registers play an important role by holding

instructions and data temporarily. These organizational concepts ensure coordination between hardware components and enable efficient processing. Understanding these concepts is essential for studying computer architecture and system design.

4.1 Stored Program Concept

- Program instructions and data are stored in memory
- CPU fetches instructions sequentially

4.2 Instruction Cycle

1. Fetch
2. Decode
3. Execute
4. Store result

---

5. Von-Neumann Architecture ==

The **Von-Neumann architecture** is a fundamental computer architecture proposed by John Von Neumann. In this architecture, **data and instructions are stored in the same memory**. The main components include the Central Processing Unit (CPU), main memory, input devices, output devices, and system buses. The CPU fetches instructions from memory, decodes them, and executes them sequentially. One major advantage of Von-Neumann architecture is its **simple and cost-effective design**, which makes it easy to implement. However, it suffers from a limitation known as the **Von-Neumann bottleneck**, where data and instructions share the same communication path. This restricts the speed of data transfer between CPU and memory. Despite this limitation, Von-Neumann architecture is widely used in most general-purpose computers today. It laid the foundation for modern computer systems and influenced future architectural developments.

Main Components

- CPU
- Main Memory
- Input/Output Devices
- System Bus

Features

- Single memory for data and instructions
- Sequential execution of instructions

- Simple and cost-effective design

Von-Neumann Bottleneck

- CPU and memory share the same bus
- Limits performance due to data transfer speed

---

6. Bus Structure ==

A **bus** is a communication pathway that transfers data and signals between different components of a computer system. The bus structure enables interaction between the CPU, memory, and input/output devices. There are three main types of buses. The **Data Bus** carries actual data between components and is bidirectional. The **Address Bus** carries the memory address of data or instructions and is unidirectional, usually from CPU to memory. The **Control Bus** carries control signals such as read, write, interrupt, and clock signals, which coordinate system operations. The width of the bus determines how much data can be transferred at a time, affecting system performance. A well-designed bus structure ensures fast and efficient communication. Modern computers use advanced bus architectures to support high-speed data transfer and multitasking operations.

Types of Buses

1. **Data Bus**
    - Transfers actual data
    - Bidirectional
2. **Address Bus**
    - Carries memory addresses
    - Unidirectional
3. **Control Bus**
    - Carries control signals (read/write, interrupt)
    - Coordinates system operations

---

7. Number Systems ==

A **number system** defines how numbers are represented and used in a computer. Computers use different number systems depending on application requirements. The most common number systems are **Decimal (base 10), Binary (base 2), Octal (base 8), and Hexadecimal (base 16)**.

Humans commonly use the decimal system, which uses digits from 0 to 9. Computers use the binary system because electronic circuits operate in two states: ON and OFF, represented by 1 and 0. Octal and hexadecimal systems are used as shorthand representations of binary numbers to simplify programming and debugging. Each number system has positional values based on its base. Conversion between number systems is an important concept in computer science. Understanding number systems is essential for data representation, memory addressing, and low-level programming.

Types of Number Systems

Number System	Base	Digits Used
Decimal	10	0–9
Binary	2	0,1
Octal	8	0–7
Hexadecimal	16	0–9, A–F

Conversions

- Decimal to Binary
- Binary to Decimal
- Binary to Hexadecimal
- Hexadecimal to Binary

7.2 Fixed-Point Representation

Definition

Fixed-point representation stores numbers with a **fixed number of digits** after the decimal point.

Characteristics

- Simple to implement
- Fast computation
- Limited range and precision

Example

Binary fixed-point:
`101.101` → integer + fractional part

Advantages

- Simple hardware
- Faster execution

Disadvantages

- Overflow problems
- Low precision for large numbers

7.3 Floating-Point Representation

Definition

Floating-point representation stores numbers in **scientific notation form**.

General Format

Number=Sign×Mantissa×BaseExponent\text{Number} = \text{Sign} \times \text{Mantissa} \times \text{Base}^{\text{Exponent}}Number=Sign×Mantissa×BaseExponent

IEEE 754 Format (Single Precision)

- 1 bit – Sign
- 8 bits – Exponent
- 23 bits – Mantissa

Advantages

- Large range of values
- High precision

Disadvantages

- Complex hardware
- Slower than fixed-point
- Rounding errors possible

---

8. Fixed-Point Representation ==

Fixed-point representation is a method of representing real numbers where the decimal (or binary) point is fixed at a specific position. In this representation, numbers are stored as integers with an implied fractional position. Fixed-point representation is simple and requires less hardware complexity, making it faster to process. It is commonly used in systems where speed is more important than precision, such as embedded systems and digital signal processing. However, fixed-point representation has a limited range and precision. Overflow and underflow problems may occur when numbers exceed the defined limits. Also, handling very small or very large numbers becomes difficult. Due to these limitations, fixed-point representation is less flexible compared to floating-point representation. Despite this, it is still preferred in applications where memory and processing resources are limited.

---

9. Floating-Point Representation ==

Floating-point representation is a method used to represent very large or very small real numbers in a computer. It represents numbers in **scientific notation**, consisting of a sign, mantissa (significand), and exponent. The decimal point "floats" according to the exponent value, allowing a wide range of values to be stored. Most computers follow the **IEEE 754 standard** for floating-point representation. Floating-point numbers provide high precision and a large dynamic range, making them suitable for scientific calculations, engineering applications, and graphics processing. However, floating-point representation is more complex and slower than fixed-point representation. It also suffers from rounding errors due to limited precision. Despite these drawbacks, floating-point representation is widely used because of its flexibility and accuracy in handling real-world numerical data.

1. Basic Concepts of Register Transfer and Registers ==

Register Transfer refers to the movement of data between registers within the CPU. A **register** is a small, high-speed storage location inside the processor used to store data, instructions, or addresses temporarily during execution. Register transfer operations are controlled by the **control unit**, which generates timing and control signals. These transfers occur during different phases of the instruction cycle such as fetch, decode, and execute. Registers allow faster access compared to main memory, improving overall system performance. Data transfer between registers is represented using symbolic notation, which simplifies hardware description. The transfer takes place over internal CPU buses and is synchronized by clock pulses. Register transfer forms the foundation of computer organization and microprogrammed control. Understanding register transfer is essential to analyze how instructions are executed at the hardware level and how data flows inside the CPU during processing.

---

2. Types of Registers ==

Registers are classified based on their function within the CPU. **General-purpose registers** store data and intermediate results during computation. **Special-purpose registers** perform specific tasks. The **Program Counter (PC)** holds the address of the next instruction to be executed. The **Instruction Register (IR)** stores the currently executing instruction. The **Memory Address Register (MAR)** holds the address of the memory location to be accessed, while the **Memory Buffer Register (MBR)** temporarily stores data being transferred to or from memory. The **Accumulator (AC)** stores results of arithmetic and logical operations. **Index registers** support array and indexed addressing, while **status or flag registers** store condition codes such as zero, carry, overflow, and sign. Registers play a vital role in improving speed, reducing memory access time, and supporting efficient instruction execution.

---

3. Register Transfer Language (RTL) ==

Register Transfer Language (RTL) is a symbolic notation used to describe **microoperations** and data transfers between registers. It provides a concise and hardware-oriented way to represent internal operations of a digital computer. In RTL, the assignment symbol (←) indicates the transfer of data from one register to another. For example, `R1 ← R2` means the contents of register R2 are transferred to register R1. Control conditions can also be specified using conditional expressions, such as `P: R1 ← R2`, meaning the transfer occurs only if condition P is true. RTL helps in understanding instruction execution, designing control units, and writing microprograms. It abstracts hardware details while clearly representing data flow. RTL is widely used in computer organization and architecture to describe register operations, memory access, and arithmetic logic operations at the microlevel.

---

4. Data Transfer between Registers ==

Data transfer between registers involves moving data from one register to another inside the CPU. This transfer is performed using internal buses and controlled by the control unit. A simple register transfer is represented as `R1 ← R2`, indicating that the content of register R2 is copied into register R1. The transfer usually occurs within one clock cycle. Multiple register transfers can be executed simultaneously if hardware resources allow. To avoid conflicts, only one register is permitted to place data on the bus at a time. The destination register receives the data when its load signal is activated. Register-to-register transfer is faster than memory access and is essential for instruction execution, temporary data storage, and intermediate computations. These transfers form the basis for arithmetic, logical, and control operations inside the processor.

---

5. Bus and Memory Transfer ==

A **bus** is a shared communication pathway used to transfer data between registers, memory, and input/output devices. In **bus transfer**, multiple registers are connected to a common bus through multiplexers or tri-state buffers. Only one source places data on the bus at a time, while one or more destination registers receive it. **Memory transfer** involves reading from or writing data to memory. During a memory read operation, the address is placed in the Memory Address Register (MAR), and data is transferred from memory to the Memory Buffer Register (MBR). In a write operation, data moves from a register to memory. Bus and memory transfers are synchronized by control signals and clock pulses. Efficient bus design reduces hardware complexity and improves system performance.

---

6. Arithmetic Microoperations ==

Arithmetic microoperations are basic arithmetic operations performed on data stored in registers. These operations are executed by the **Arithmetic Logic Unit (ALU)**. Common arithmetic microoperations include addition, subtraction, increment, decrement, and transfer. Examples include `R3 ← R1 + R2`, `R1 ← R1 − R2`, and `R2 ← R2 + 1`. These operations are fundamental to instruction execution such as arithmetic instructions, loop control, and address calculations. Arithmetic microoperations use binary arithmetic and may generate condition flags like carry, zero, and overflow. They are essential for mathematical computation and control flow in programs. The speed and efficiency of arithmetic microoperations directly impact the overall performance of the processor.

---

7. Logic Microoperations ==

Logic microoperations perform bit-wise logical operations on the contents of registers. These operations are also executed by the ALU. Common logic microoperations include AND, OR, XOR, and NOT. For example, `R3 ← R1 AND R2` performs a bit-wise AND operation. Logic microoperations are widely used in data manipulation, bit masking, setting or clearing bits, and comparison operations. They are essential in control applications and digital signal processing. Logic microoperations operate independently on each bit, making them suitable for low-level hardware control and optimization. The results may affect status flags such as zero or sign. These operations help implement decision-making and control structures within computer programs.

---

8. Shift Microoperations ==

Shift microoperations shift the bits of a register either left or right. They are used for data manipulation, arithmetic operations, and bit-level processing. There are three main types of shift operations: **logical shift**, **arithmetic shift**, and **circular (rotate) shift**. In a logical shift, zeros are inserted into vacant bit positions. An arithmetic shift preserves the sign bit while shifting,

making it useful for signed numbers. Circular shifts rotate bits without losing any data. Shift microoperations are commonly used for multiplication or division by powers of two, data encoding, and cryptographic operations. These operations are fast and efficient, making them valuable in performance-critical applications. Shift microoperations play an important role in low-level programming and processor design.

Unit-2

1. Instruction Codes ==

Instruction codes are binary patterns that specify the operations to be performed by a computer. Each instruction code consists of two main parts: the **opcode (operation code)** and the **operand**. The opcode defines the type of operation such as addition, subtraction, data transfer, or logical operation, while the operand specifies the data or memory location on which the operation is performed. Instruction codes are stored in main memory and fetched by the CPU during execution. They are decoded by the control unit to generate appropriate control signals. Instruction codes may have different formats depending on the architecture, such as zero-address, one-address, two-address, or three-address instructions. Efficient instruction coding reduces memory usage and improves execution speed. Instruction codes form the basis of program execution and play a crucial role in the overall performance of the computer system.

---

2. Common Bus System Architecture ==

A **common bus system architecture** is used to transfer data between registers, memory, and the CPU using a shared communication pathway known as a bus. Instead of having separate connections between each component, a single bus simplifies hardware design and reduces cost. The bus is controlled by the control unit, which ensures that only one data source places information on the bus at a time. Registers are connected to the bus through multiplexers or tri-state buffers. The common bus architecture supports register-to-register transfer, memory access, and input/output operations. It also improves system flexibility by allowing easy expansion of components. However, because the bus is shared, simultaneous data transfers are not possible, which may reduce performance. Despite this limitation, common bus system architecture is widely used due to its simplicity and efficiency in basic computer design.

---

3. Computer Instructions: Instruction Set ==

An **instruction set** is a collection of all instructions that a computer's processor can understand and execute. It defines the capabilities of the CPU and acts as an interface between hardware and software. The instruction set includes instructions for data transfer, arithmetic operations, logical operations, control flow, and input/output operations. Each instruction in the set has a unique

opcode. The size and complexity of the instruction set depend on the processor design. A well-designed instruction set improves program efficiency and reduces execution time. Instruction sets are classified into **CISC (Complex Instruction Set Computer)** and **RISC (Reduced Instruction Set Computer)** architectures. In CISC, instructions are complex and powerful, while RISC uses simpler and faster instructions. The instruction set plays a critical role in determining processor performance, programmability, and compatibility with software.

---

4. Instruction Cycle ==

The **instruction cycle** refers to the sequence of steps performed by the CPU to execute an instruction. It consists of four main phases: **fetch, decode, execute, and store**. During the fetch phase, the CPU retrieves the instruction from memory using the Program Counter (PC). In the decode phase, the control unit interprets the instruction and identifies the required operation. During execution, the CPU performs the specified operation using the ALU, registers, or memory. Finally, the result is stored in a register or memory. Some instructions may also include additional steps such as interrupt checking. The instruction cycle is repeated continuously until the program ends. Efficient execution of the instruction cycle ensures high system performance. Understanding the instruction cycle helps explain how software instructions are converted into hardware operations.

---

5. Register Reference Instructions ==

Register reference instructions operate directly on the data stored in CPU registers. These instructions do not require memory access and are usually executed within a single clock cycle. Register reference instructions are identified by specific opcode patterns and are used for operations such as clearing registers, complementing data, incrementing values, and shifting bits. Examples include clearing the accumulator, complementing the accumulator, and shifting left or right. These instructions improve execution speed because registers provide fast data access compared to memory. Register reference instructions are essential for efficient data manipulation and control operations. They play a significant role in optimizing program performance and reducing instruction execution time. Since they operate entirely within the CPU, they are widely used in arithmetic processing and logical control tasks.

---

6. Memory Reference Instructions ==

Memory reference instructions are instructions that access data stored in main memory. These instructions include operations such as load, store, add, subtract, and logical operations involving memory operands. Memory reference instructions typically consist of an opcode, an address field, and sometimes an indirect addressing bit. During execution, the CPU uses the address field

to locate the operand in memory. These instructions require more execution time than register reference instructions due to memory access delays. Memory reference instructions are essential for handling large amounts of data that cannot be stored entirely in registers. They form the core of most computer programs and enable interaction between CPU and memory. Efficient handling of memory reference instructions improves overall system performance.

---

7. Input-Output Reference Instructions ==

Input-Output (I/O) reference instructions are used to transfer data between the CPU and input/output devices such as keyboards, printers, and disks. These instructions control communication between the processor and external devices. I/O reference instructions include operations such as input data transfer, output data transfer, enabling interrupts, and disabling interrupts. They allow the computer to receive input from users and provide output results. I/O instructions are executed under the control of the control unit and may involve special I/O registers. These instructions are essential for interactive computing and real-time systems. Efficient input-output operations ensure smooth communication between hardware devices and the processor. I/O reference instructions play a crucial role in system responsiveness and overall functionality.

1. Interrupts – Introduction and Concept ==

An **interrupt** is a signal that temporarily halts the normal execution of a program and transfers control to a special routine called the **Interrupt Service Routine (ISR)**. Interrupts allow the CPU to respond quickly to important events such as input/output requests, hardware failures, or program errors. Instead of continuously checking devices (polling), the CPU can perform useful work and respond only when an interrupt occurs. When an interrupt is generated, the CPU completes the current instruction, saves its state, and jumps to the ISR. After servicing the interrupt, control returns to the original program. Interrupts improve system efficiency, responsiveness, and multitasking capability. They are widely used in operating systems, real-time systems, and embedded systems. Without interrupts, the CPU would waste time monitoring devices continuously, resulting in poor performance.

---

2. Maskable and Non-Maskable Interrupts ==

Interrupts can be classified as **maskable** and **non-maskable** based on whether they can be disabled. **Maskable interrupts** are interrupts that can be enabled or disabled by the CPU using interrupt control instructions. They are generally used for low-priority tasks such as keyboard input, printer signals, or disk operations. Maskable interrupts allow the processor to ignore certain interrupts during critical sections of program execution.
Non-maskable interrupts (NMI) cannot be disabled and are used for high-priority events. These interrupts occur due to serious conditions such as hardware failures, power failure, or

memory errors. Because NMIs have the highest priority, the CPU responds to them immediately. Maskable interrupts provide flexibility, while non-maskable interrupts ensure system reliability and safety. Together, they help manage priorities and ensure efficient system operation.

---

3. Hardware and Software Interrupts ==

Interrupts are also classified as **hardware interrupts** and **software interrupts** based on their source. **Hardware interrupts** are generated by external hardware devices such as keyboards, timers, disk drives, or network cards. These interrupts signal the CPU that a device requires attention. Hardware interrupts are asynchronous, meaning they can occur at any time during program execution.
Software interrupts, on the other hand, are generated by programs using special instructions. They are synchronous and occur as a result of program execution. Software interrupts are commonly used to request operating system services, such as system calls. Examples include interrupt instructions used for input/output operations or error handling. Hardware interrupts improve responsiveness to external events, while software interrupts provide a controlled way for programs to interact with the operating system.

---

4. Interrupt Service Routine (ISR) ==

An **Interrupt Service Routine (ISR)** is a special program that executes in response to an interrupt. When an interrupt occurs, the CPU transfers control to the ISR associated with that interrupt. The ISR identifies the source of the interrupt and performs the necessary actions, such as reading input data, sending output, or handling errors. Before executing the ISR, the CPU saves the current program state, including register contents and program counter. ISRs are designed to be short and efficient to minimize delay in returning to the main program. After the ISR completes its task, it restores the saved state and resumes normal execution. Proper design of ISRs is essential to ensure fast response and system stability. ISRs play a critical role in multitasking and real-time systems.

---

5. Context Switching ==

Context switching is the process of saving the current state of a running program and restoring the state of another program or process. During an interrupt, the CPU must preserve the context of the interrupted program, including register values, program counter, stack pointer, and status flags. This saved information is stored in memory or a process control block. Once the interrupt is serviced, the CPU restores the saved context and resumes execution. Context switching enables multitasking, allowing the CPU to handle multiple tasks efficiently. Although necessary, context switching introduces some overhead because saving and restoring context takes time.

Efficient context switching is crucial for high-performance operating systems and real-time applications. It ensures smooth execution and proper coordination among multiple processes.

---

6. Interrupt Identification: Daisy Chaining ==

Daisy chaining is a method used to identify the source of an interrupt when multiple devices are connected to the CPU. In this technique, devices are connected in a serial chain. When an interrupt occurs, the CPU sends an interrupt acknowledge signal that passes through the chain. The first device that has requested an interrupt captures the signal and responds by placing its interrupt vector on the bus. Devices closer to the CPU have higher priority than those further away. Daisy chaining is simple and requires minimal hardware. However, it has limitations such as fixed priority and delay for lower-priority devices. Despite these drawbacks, daisy chaining is commonly used in small systems due to its simplicity and low cost.

---

7. Interrupt Identification: Polling and Vectored Interrupt ==

Polling is a method where the CPU checks each device in sequence to determine which one has generated an interrupt. The CPU queries devices one by one until the interrupting device is found. Polling is simple to implement but inefficient because it wastes CPU time, especially when many devices are present.
In contrast, a **vectored interrupt** allows the interrupting device to supply the address of its ISR directly to the CPU. This eliminates the need for polling and speeds up interrupt handling. Each device has a unique interrupt vector. Vectored interrupts provide fast response and efficient identification of interrupt sources. Modern systems prefer vectored interrupts due to their speed and scalability, while polling is mainly used in simple or low-cost systems.

---

8. Interrupt Cycle ==

The **interrupt cycle** is a sequence of steps performed by the CPU to handle an interrupt. It begins when an interrupt request is detected. The CPU completes the current instruction and checks whether interrupts are enabled. If accepted, the CPU saves the current context, including the program counter and status register. Next, it determines the source of the interrupt and transfers control to the appropriate ISR. The ISR executes and services the interrupt. After completion, a return-from-interrupt instruction restores the saved context. Finally, the CPU resumes execution of the interrupted program. The interrupt cycle ensures proper handling of external and internal events without losing program state. Efficient interrupt cycles are essential for responsive and reliable computer systems.

Unit-3

1. Central Processing Unit (CPU): Introduction ==

The **Central Processing Unit (CPU)** is the most important component of a computer system and is often called the *brain of the computer*. It is responsible for executing instructions, performing calculations, and controlling the operations of all other components. The CPU interprets program instructions stored in memory and processes data according to those instructions. It performs arithmetic, logical, control, and input/output operations. The CPU mainly consists of three parts: the **Arithmetic Logic Unit (ALU)**, which performs arithmetic and logical operations; the **Control Unit (CU)**, which directs and coordinates the activities of the computer; and **registers**, which provide fast temporary storage. The CPU works in synchronization with the system clock and follows the instruction cycle to execute programs. The overall performance of a computer largely depends on the speed and efficiency of its CPU.

---

2. General Register Organization ==

General register organization refers to the arrangement and use of registers inside the CPU to store data and instructions during processing. Registers are small, high-speed storage units that allow faster access compared to main memory. In a general register organization, multiple registers are connected through a common bus system and controlled by the control unit. These registers can store operands, intermediate results, addresses, and instructions. General-purpose registers can be used for different tasks, making the processor more flexible and efficient. The ALU receives data from registers, performs operations, and stores results back into registers. Register selection and data transfer are managed using control signals. This organization reduces memory access, improves execution speed, and supports complex instruction execution. General register organization is widely used in modern processors to enhance performance and flexibility.

---

3. Stack Organization: Register Stack and Memory Stack ==

A **stack organization** is a method of organizing data in a **Last In First Out (LIFO)** manner. In a **register stack**, the stack is implemented using a set of CPU registers. A stack pointer register keeps track of the top of the stack. Register stacks are fast but limited in size. In contrast, a **memory stack** is implemented in main memory, allowing larger storage but slower access compared to register stacks. Stack organization is commonly used for expression evaluation, subroutine calls, parameter passing, and interrupt handling. Operations on a stack include **push** (insert data) and **pop** (remove data). Stack-based systems do not require explicit operand addresses, simplifying instruction formats. Stack organization plays an important role in program execution and control flow management.

4. Reverse Polish Notation (RPN) ==

Reverse Polish Notation (RPN) is a mathematical notation in which operators follow their operands. Unlike infix notation, RPN does not require parentheses to define operation order. For example, the infix expression `(A + B)` is written as `A B +` in RPN. RPN is widely used in stack-based computers because it simplifies expression evaluation. Operands are pushed onto the stack, and when an operator is encountered, the required operands are popped, the operation is performed, and the result is pushed back onto the stack. This reduces the need for complex instruction decoding and addressing. RPN improves computational efficiency and reduces hardware complexity. It is commonly used in calculators, compilers, and stack-oriented architectures.

---

5. Three-Address Instruction Format ==

A **three-address instruction format** contains three operand addresses: two source operands and one destination operand. The general form is:
`R1 ← R2 op R3`
where R2 and R3 are source registers, and R1 stores the result. This format allows operations to be performed in a single instruction, reducing the total number of instructions required. Three-address instructions are easy to understand and support efficient programming. However, they require more bits to represent operands, increasing instruction length. Due to larger instruction size, they are less commonly used in modern architectures. This format is mainly found in high-level conceptual designs and some early computer systems. Despite higher memory usage, three-address instructions simplify compiler design and program logic.

---

6. Two-Address Instruction Format ==

In a **two-address instruction format**, one operand acts as both a source and a destination. The general form is:
`R1 ← R1 op R2`
Here, R1 contains one operand and also stores the result. This format reduces instruction length compared to three-address instructions but may require additional instructions to preserve data. Two-address instructions are commonly used in many processors due to a good balance between instruction size and efficiency. They reduce hardware complexity and memory usage. However, programmers and compilers must handle data carefully to avoid overwriting important values. This instruction format is widely adopted in general-purpose processors.

---

7. One-Address Instruction Format ==

A **one-address instruction format** uses a single explicit address and an implicit accumulator register. The general form is:
`AC ← AC op M`
where AC is the accumulator and M is a memory location. One-address instructions reduce instruction length and simplify hardware design. However, they require frequent use of the accumulator, which can lead to more instructions and reduced flexibility. This format was commonly used in early computers. One-address instructions rely heavily on load and store operations, increasing memory access. Despite these limitations, the simplicity of this format makes it easy to implement and understand.

---

8. Zero-Address Instruction Format ==

A **zero-address instruction format** does not specify any operand addresses explicitly. It is used in **stack-based architectures**, where operands are implicitly taken from the stack. Instructions operate on the top elements of the stack. For example, an ADD instruction pops two operands from the stack, adds them, and pushes the result back. Zero-address instructions result in very compact instruction codes and simplified hardware. However, program execution may require more instructions due to frequent stack operations. This format is commonly used in stack machines and virtual machines. Zero-address instruction formats reduce memory requirements and support efficient expression evaluation.

1. Addressing Modes – Introduction ==

Addressing modes define how the operand (data) of an instruction is specified or accessed in memory or registers. They provide flexibility in instruction execution and help optimize program performance. By using different addressing modes, the same instruction can operate on different data locations. Addressing modes reduce instruction size, improve efficiency, and simplify programming. The CPU uses addressing modes to calculate the **effective address** of the operand. Different modes are used depending on whether the data is part of the instruction, stored in a register, or located in memory. Common addressing modes include implied, immediate, register direct, register indirect, and direct addressing modes. Understanding addressing modes is essential for low-level programming, instruction decoding, and efficient CPU design.

---

2. Implied Addressing Mode ==

In the **implied addressing mode**, the operand is **implicitly specified** by the instruction itself. No explicit address or operand field is required in the instruction. The operation is performed on a predefined register or location, usually the accumulator or stack top. For example, instructions like **CLEAR ACCUMULATOR** or **INCREMENT ACCUMULATOR** use implied addressing

because the operand is already known. This addressing mode results in very short instruction length and fast execution. It is commonly used in stack-based and accumulator-based architectures. However, implied addressing lacks flexibility because it operates only on specific registers. Despite this limitation, it simplifies instruction formats and reduces memory usage. Implied addressing mode is useful for frequently used operations that do not require operand specification.

---

3. Immediate Addressing Mode ==

In the **immediate addressing mode**, the operand value is directly included in the instruction itself. The data does not need to be fetched from memory or registers, making execution fast. For example, the instruction `ADD #5` adds the value 5 directly to a register or accumulator. Immediate addressing mode is useful for constants and initialization operations. Since the operand is part of the instruction, no additional memory access is required. However, the size of the operand is limited by the instruction format. Immediate addressing is commonly used for arithmetic operations, loop counters, and fixed values. This mode improves speed but reduces flexibility because the operand value cannot be changed without modifying the instruction.

---

4. Register Direct and Register Indirect Addressing Mode ==

In **register direct addressing mode**, the operand is stored in a CPU register, and the instruction specifies the register name. For example, `ADD R1` means the data in register R1 is used. This mode provides very fast execution because registers are high-speed storage units.
In **register indirect addressing mode**, the register contains the **address of the operand** rather than the operand itself. For example, `ADD (R1)` means the CPU uses the memory location whose address is stored in R1. Register indirect addressing allows access to memory with fewer instruction bits and supports dynamic data structures like arrays and pointers. While slightly slower than register direct mode, it is more flexible. Both modes reduce instruction size and improve efficiency.

---

5. Direct Addressing Mode ==

In the **direct addressing mode**, the instruction contains the **actual memory address** of the operand. The CPU directly accesses the specified memory location to fetch the data. For example, `LOAD 2000` means the data stored at memory location 2000 is loaded into a register. This addressing mode is simple and easy to understand. However, it requires more bits in the instruction to specify the address, increasing instruction length. Direct addressing provides limited memory access range depending on address size. It is slower than register addressing

because it involves memory access. Despite this, direct addressing is commonly used for fixed memory locations and simple programs.

---

6. Data Transfer Instructions ==

Data transfer instructions are used to move data between registers, memory, and input/output devices. These instructions do not modify the data; they only transfer it from one location to another. Common data transfer instructions include **LOAD**, **STORE**, **MOVE**, **PUSH**, and **POP**. For example, a LOAD instruction transfers data from memory to a register, while STORE transfers data from a register to memory. PUSH and POP are used in stack operations. Data transfer instructions are essential for program execution because all operations require data movement. Efficient data transfer reduces execution time and improves overall system performance. These instructions form the backbone of instruction execution in a computer system.

---

7. Data Manipulation Instructions ==

Data manipulation instructions perform operations on data stored in registers or memory. These include **arithmetic instructions** (ADD, SUBTRACT, MULTIPLY, DIVIDE) and **logical instructions** (AND, OR, NOT, XOR). Shift and rotate instructions are also part of data manipulation. These instructions modify the data and may affect status flags such as zero, carry, and overflow. Data manipulation instructions are executed by the **Arithmetic Logic Unit (ALU)**. They are essential for calculations, decision making, and data processing in programs. Efficient manipulation instructions improve processing speed and support complex computations in applications such as scientific calculations and data analysis.

---

8. Program Control Instructions ==

Program control instructions control the flow of program execution. These instructions determine the sequence in which instructions are executed. Common program control instructions include **JUMP**, **BRANCH**, **CALL**, **RETURN**, and **HALT**. Conditional branch instructions alter program flow based on status flags, enabling decision-making and looping. Subroutine calls allow reuse of code and modular programming. Interrupt and return instructions manage external events and multitasking. Program control instructions are essential for implementing loops, conditional statements, and function calls. They help manage execution order efficiently and support structured programming. Without program control instructions, complex programs could not be executed properly.

---

9. Reduced Instruction Set Computer (RISC) ==

A **Reduced Instruction Set Computer (RISC)** architecture uses a small, simple set of instructions designed for fast execution. Each instruction typically executes in a single clock cycle. RISC emphasizes register-based operations and uses simple addressing modes. The architecture supports pipelining, which improves instruction throughput. RISC processors require more instructions to perform complex tasks, but each instruction is fast and efficient. Examples of RISC processors include ARM and MIPS. RISC architecture reduces hardware complexity, power consumption, and execution time. It is widely used in mobile devices and embedded systems.

---

10. Complex Instruction Set Computer (CISC) ==

A **Complex Instruction Set Computer (CISC)** architecture uses a large set of powerful instructions. Each instruction can perform complex operations, reducing the number of instructions per program. CISC supports multiple addressing modes and variable instruction lengths. While CISC instructions may take multiple clock cycles to execute, they reduce program size. Examples include Intel x86 processors. CISC architecture simplifies programming and compiler design. However, it increases hardware complexity and power consumption. Despite this, CISC processors are widely used in personal computers and servers due to backward compatibility and versatility.

Unit-4

1. Memory Hierarchy ==

Memory hierarchy is the structured arrangement of different types of memory in a computer system based on speed, cost, capacity, and access time. It is designed to provide fast access to data while keeping the overall system cost low. At the top of the hierarchy are **registers**, which are the fastest but smallest and most expensive memory. Below registers is **cache memory**, followed by **main memory (RAM)**, and finally **auxiliary or secondary memory** such as hard disks and SSDs. As we move down the hierarchy, memory becomes larger and cheaper but slower. The principle of **locality of reference** (temporal and spatial locality) justifies memory hierarchy design. Frequently used data is kept in faster memory, while less-used data is stored in slower memory. Memory hierarchy improves system performance by minimizing average memory access time.

2. Main Memory Technologies ==

Main memory is the primary storage area where programs and data currently in use are stored. It is directly accessible by the CPU. The two main types of main memory technologies are **RAM (Random Access Memory)** and **ROM (Read Only Memory)**. RAM is volatile, meaning data is lost when power is turned off. It includes **Static RAM (SRAM)**, which is fast and used for cache, and **Dynamic RAM (DRAM)**, which is slower but cheaper and widely used as main memory. ROM is non-volatile and stores permanent instructions such as firmware and boot programs. Variants of ROM include PROM, EPROM, and EEPROM. Advances in semiconductor technology have improved memory density, speed, and reliability. Main memory plays a critical role in program execution speed and overall system performance.

---

3. Auxiliary Memory (Secondary Memory) ==

Auxiliary memory, also known as **secondary memory**, is used for long-term storage of data and programs. Unlike main memory, it is **non-volatile**, meaning data is retained even when power is off. Auxiliary memory has large storage capacity and low cost per bit but slower access speed. Common examples include **hard disk drives (HDDs)**, **solid-state drives (SSDs)**, **optical disks**, and **magnetic tapes**. Auxiliary memory stores operating systems, application software, user data, and backup files. Data must be transferred to main memory before being processed by the CPU. Although slower, auxiliary memory is essential for permanent storage and data recovery. It supports file systems and large databases. The performance gap between auxiliary and main memory is a key reason for cache and memory hierarchy design.

---

4. Associative Memory – Hardware Requisites ==

Associative memory, also known as **Content Addressable Memory (CAM)**, is a special type of memory that retrieves data based on content rather than address. Unlike conventional memory, CAM compares the search key with all stored entries simultaneously. The hardware requirements include **parallel comparison circuits**, **match logic**, and **associative registers**. Each memory word has a comparator that checks whether stored data matches the input key. A **match line** indicates whether a match is found. Priority encoders are used when multiple matches occur. Associative memory requires more hardware and is costlier than conventional memory. However, its ability to perform fast searches makes it valuable in cache memory, virtual memory systems, and networking devices. The hardware complexity is justified where high-speed searching is critical.

---

5. Associative Memory – Working Principle and Operations ==

The **working principle of associative memory** is based on parallel searching. When a search key is applied, all memory entries are compared simultaneously. If a match occurs, the corresponding data is retrieved instantly. This parallel operation makes associative memory extremely fast. Common operations include **read**, **write**, and **search**. During a search operation, the key is compared with stored contents, and matching entries activate match lines. In a write operation, new data is stored along with its associated key. Associative memory is often used in cache systems to quickly check whether required data is present. It is also used in translation lookaside buffers (TLB) for fast address translation. Despite its speed advantage, associative memory is limited in size due to high hardware cost.

---

6. Cache Memory – Characteristics ==

Cache memory is a small, high-speed memory placed between the CPU and main memory. Its purpose is to reduce memory access time by storing frequently used data and instructions. Cache memory is usually built using **SRAM**, which is faster than DRAM. Key characteristics of cache memory include **small size**, **very fast access time**, **high cost**, and **transparency to the programmer**. Cache works on the principle of locality of reference. It significantly improves CPU performance by reducing the need to access slower main memory. Cache memory is managed by hardware and may be organized into multiple levels such as **L1, L2, and L3 cache**. A well-designed cache system greatly enhances overall system efficiency.

---

7. Types of Cache Mapping ==

Cache mapping determines how main memory blocks are placed into cache. The three main types are **Direct Mapping**, **Associative Mapping**, and **Set-Associative Mapping**. In direct mapping, each memory block maps to a fixed cache location. It is simple and fast but suffers from frequent conflicts. Associative mapping allows a memory block to be placed anywhere in cache, reducing conflicts but increasing hardware complexity. Set-associative mapping is a compromise between the two. Cache is divided into sets, and a block can be placed in any line within a set. This reduces conflict misses and balances speed and complexity. Cache mapping techniques directly affect hit ratio, performance, and hardware cost.

---

8. Writing into Cache ==

Writing into cache refers to how data updates are handled when a write operation occurs. The two main write policies are **Write-Through** and **Write-Back**. In write-through, data is written simultaneously to cache and main memory. This ensures data consistency but increases memory traffic. In write-back, data is written only to cache and updated to main memory later, reducing memory access but requiring a dirty bit to track changes. Another consideration is **write allocation**, which determines whether data is loaded into cache on a write miss. Proper write policies improve performance and maintain data correctness. The choice of write strategy depends on system design, performance needs, and complexity.

---

9. Cache Coherence ==

Cache coherence ensures data consistency among multiple caches in a multiprocessor system. When multiple processors have their own caches, a change in one cache must be reflected in others. Without coherence, processors may use outdated data. Cache coherence is maintained using protocols such as **MSI, MESI, and MOESI**. These protocols track the state of cache blocks and ensure correct updates. Techniques like **snooping** and **directory-based coherence** are commonly used. Cache coherence is essential for correct execution of parallel programs. Although it adds hardware complexity, it ensures reliability and correctness in modern multi-core systems.

1. Peripheral Devices ==

Peripheral devices are external hardware components connected to a computer system to provide input, output, or storage functions. They extend the functionality of the CPU and main memory by enabling interaction with the external world. Peripheral devices are broadly classified into **input devices**, **output devices**, and **storage devices**. Input devices such as keyboards, mice, scanners, and microphones allow users to enter data and instructions into the computer. Output devices like monitors, printers, and speakers present processed results in human-readable form. Storage peripherals include hard disks, SSDs, and optical drives used for long-term data storage. Peripheral devices operate at different speeds and use different data formats, making direct communication with the CPU difficult. Therefore, special control mechanisms and interfaces are required. Peripheral devices play a vital role in making computers usable, interactive, and capable of handling real-world applications.

---

2. Input-Output Interface ==

An **Input-Output (I/O) interface** is a hardware component that connects peripheral devices to the CPU and main memory. Its primary function is to resolve differences in speed, data format, and control signals between the CPU and peripheral devices. The I/O interface contains data registers, status registers, control registers, and logic circuits. It manages data transfer, device selection, and error detection. The interface also generates control signals and receives status

information from devices. By using an I/O interface, the CPU can communicate with peripherals in a standardized manner without needing to know device-specific details. I/O interfaces support different modes of data transfer such as programmed I/O, interrupt-driven I/O, and DMA. They play a crucial role in ensuring reliable and efficient communication between the computer system and external devices.

---

3. Asynchronous Data Transfer ==

Asynchronous data transfer is a method of transferring data between devices that do not share a common clock. Since the CPU and peripheral devices operate at different speeds, synchronization is achieved using control signals rather than timing signals. Common asynchronous transfer techniques include **strobe control** and **handshaking**. In strobe control, a control signal indicates when data is valid. In handshaking, both sender and receiver exchange signals to coordinate data transfer. Asynchronous transfer is flexible and widely used in I/O systems because it allows devices to operate independently. However, it introduces additional overhead due to control signals and delays. Asynchronous data transfer ensures reliable communication between slow peripherals and fast processors, making it essential in input-output organization.

---

4. Programmed I/O ==

Programmed I/O is a data transfer mode where the CPU is fully responsible for managing I/O operations. In this mode, the CPU continuously checks the status of the peripheral device using polling to determine whether it is ready for data transfer. When the device is ready, the CPU transfers data between the device and memory using I/O instructions. Programmed I/O is simple to implement and requires minimal hardware. However, it is inefficient because the CPU remains busy waiting for the device, wasting processing time. This mode is suitable only for low-speed devices or simple systems. Due to CPU involvement in every transfer, programmed I/O reduces overall system performance. Despite its limitations, it helps in understanding basic I/O concepts.

---

5. Interrupt-Initiated I/O ==

Interrupt-initiated I/O improves efficiency by allowing the CPU to perform other tasks while waiting for an I/O device. In this mode, the CPU initiates the I/O operation and continues executing other programs. When the device becomes ready, it sends an interrupt signal to the CPU. The CPU temporarily suspends the current program and executes an **Interrupt Service Routine (ISR)** to handle the I/O operation. After servicing the interrupt, the CPU resumes the interrupted program. Interrupt-initiated I/O eliminates continuous polling and reduces CPU idle

time. It is more efficient than programmed I/O and widely used in modern systems. However, it requires additional hardware and software support for interrupt handling. This mode significantly improves system responsiveness and multitasking capability.

---

6. Priority Interrupt – Daisy Chaining Priority ==

Daisy chaining priority is a method used to assign priority among multiple interrupting devices. In this technique, devices are connected in a serial chain. When an interrupt occurs, the CPU sends an interrupt acknowledge signal that passes through the chain. The first device in the chain with a pending interrupt captures the signal and responds, while devices further down the chain are blocked. Devices closer to the CPU have higher priority. Daisy chaining is simple and requires less hardware. However, it has fixed priority and slower response for low-priority devices. Failure of a single device can disrupt the chain. Despite its limitations, daisy chaining is used in small and low-cost systems due to its simplicity.

---

7. Priority Interrupt – Parallel Priority Interrupt ==

In **parallel priority interrupt**, all interrupting devices are connected to the CPU through separate interrupt request lines. A **priority encoder** determines the highest-priority interrupt among simultaneous requests. The CPU then services the interrupt with the highest priority. This method provides faster interrupt recognition and flexible priority assignment. Unlike daisy chaining, priority levels can be changed easily. Parallel priority interrupt systems are more reliable and efficient but require more hardware, making them costlier. They are commonly used in high-performance and real-time systems where fast and accurate interrupt handling is critical. Parallel priority interrupts ensure minimal delay for high-priority devices and improve overall system reliability.

---

8. Direct Memory Access (DMA) – Controller ==

Direct Memory Access (DMA) allows data transfer between I/O devices and main memory without continuous CPU involvement. A **DMA controller** is a special hardware unit that manages this process. When a DMA transfer is requested, the CPU initializes the DMA controller with the starting address, transfer count, and direction of transfer. The DMA controller then takes control of the system bus and transfers data directly between memory and the device. During DMA operation, the CPU is temporarily halted or operates in parallel depending on the DMA mode. The DMA controller reduces CPU overhead and significantly improves data transfer speed, especially for large data blocks. DMA is widely used for high-speed devices such as disks and network interfaces.

9. DMA Transfer ==

A **DMA transfer** is the actual process of moving data directly between memory and an I/O device under the control of the DMA controller. DMA transfers can occur in different modes such as **burst mode**, **cycle stealing**, and **transparent mode**. In burst mode, the DMA controller transfers a block of data continuously, temporarily stopping the CPU. In cycle stealing, the DMA controller transfers one word at a time, sharing the bus with the CPU. Transparent mode allows DMA transfers only when the CPU is idle. DMA transfers are fast and efficient, reducing CPU workload and improving system performance. After completing the transfer, the DMA controller sends an interrupt to notify the CPU. DMA is essential for modern high-speed computing systems.