Paper: Computer Organization and Architecture
Code: MCAN-103

Unit 1: Introduction to Digital Logic Design (8L)

This unit provides the foundational concepts of **Digital Logic Design**, covering Boolean algebra, logic gates, simplification of Boolean expressions, combinational and sequential circuits, memory devices, and flip-flops. These concepts are essential for designing digital systems and understanding how computers and other digital devices function.

1.1 Axioms and Laws of Boolean Algebra

Boolean algebra is a mathematical structure that deals with binary variables (values of 0 and 1) and logical operations. It is the foundation of digital circuit design.

- **Basic Axioms and Laws**:
    1. **Identity Law**:
        - $A+0=A$ $A + 0 = A$
        - $A \cdot 1=A$ $A \cdot 1 = A$
    2. **Null Law**:
        - $A+1=1$ $A + 1 = 1$
        - $A \cdot 0=0$ $A \cdot 0 = 0$
    3. **Idempotent Law**:
        - $A+A=A$ $A + A = A$
        - $A \cdot A=A$ $A \cdot A = A$
    4. **Complement Law**:
        - $A+\overline{A}=1$ $A + \overline{A} = 1$
        - $A \cdot \overline{A}=0$ $A \cdot \overline{A} = 0$
    5. **Distributive Law**:
        - $A \cdot (B+C)=(A \cdot B)+(A \cdot C)$ $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
        - $A+(B \cdot C)=(A+B) \cdot (A+C)$ $A + (B \cdot C) = (A + B) \cdot (A + C)$
    6. **De Morgan's Laws**:
        - $\overline{A \cdot B}=\overline{A}+\overline{B}$ $\overline{A \cdot B} = \overline{A} + \overline{B}$
        - $\overline{A+B}=\overline{A} \cdot \overline{B}$ $\overline{A + B} = \overline{A} \cdot \overline{B}$

These laws are used to simplify Boolean expressions and design efficient digital circuits.

1.2 Reduction of Boolean Expressions

Reducing a Boolean expression involves applying the axioms and laws of Boolean algebra to simplify complex expressions, reducing the number of gates and components needed in digital circuits.

- **Example**:
  Simplify the Boolean expression:
  $A \cdot (A+B)$A \cdot (A + B)

  Solution: Using the **Absorption Law**:
  $A \cdot (A+B) = A$A \cdot (A + B) = A

This simplification results in a circuit that requires fewer gates.

1.3 Conversion Between Canonical Forms

Canonical forms are standard representations of Boolean expressions. There are two canonical forms: **Sum of Products (SOP)** and **Product of Sums (POS)**.

- **Sum of Products (SOP)**:
  An expression where terms are ORed and each term is a product (AND operation) of variables or their complements.
  - Example: $(A \cdot B)+(\overline{A} \cdot C)$(A \cdot B) + (\overline{A} \cdot C)
- **Product of Sums (POS)**:
  An expression where terms are ANDed and each term is a sum (OR operation) of variables or their complements.
  - Example: $(A+B) \cdot (\overline{A}+C)$(A + B) \cdot (\overline{A} + C)

1.4 Karnaugh Map (K-map) - 4 Variables

A **Karnaugh Map (K-map)** is a graphical tool used for simplifying Boolean expressions, especially for expressions with 2 to 6 variables. It provides a systematic method to visualize the simplification of SOP and POS expressions.

- **4-variable K-map**:
  A 4-variable K-map is a 4x4 grid where each cell represents a minterm of a 4-variable Boolean expression. Variables are placed on the axes, and the cells are filled based on the truth table values of the expression.

  Steps for K-map simplification:

  1. Identify the ones in the K-map (representing minterms).

2. Group the adjacent ones into the largest possible rectangular groups (size of 1, 2, 4, 8, etc.).
3. Write the simplified Boolean expression by eliminating variables that change within each group.

- **Example**: For the Boolean expression F(A,B,C,D)=∑m(1,3,5,7,9,11,15)F(A, B, C, D) = \sum m(1, 3, 5, 7, 9, 11, 15), plot the minterms on a 4-variable K-map and simplify by grouping adjacent ones.

1.5 Half Adder and Full Adder

Half Adder and **Full Adder** are combinational circuits used to add binary numbers.

- **Half Adder**:
  A Half Adder adds two single-bit binary numbers and produces a sum and a carry-out.

○ **Truth Table**:

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

○ **Sum**: A⊕BA \oplus B

○ **Carry**: A⋅BA \cdot B

- **Full Adder**:
  A Full Adder adds two single-bit binary numbers along with a carry input from a previous stage. It produces a sum and a carry-out.

○ **Truth Table**:

A	B	Cin	Sum	Carry

0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

    - **Sum**: A⊕B⊕CinA \oplus B \oplus Cin
    - **Carry**: (A⋅B)+(Cin⋅(A⊕B))(A \cdot B) + (Cin \cdot (A \oplus B))

1.6 4-bit Parallel Parity Bit Generator and Checker Circuit

- **Parity Bit Generator**:
  A parity bit is added to data to ensure error detection. For **even parity**, the parity bit is chosen such that the total number of 1s is even. For **odd parity**, the total number of 1s is odd.

- **4-bit Parallel Parity Generator**: A 4-bit parallel parity generator calculates the parity bit for a 4-bit input.
    - If the input is A,B,C,DA, B, C, D, the parity bit is calculated as P=A⊕B⊕C⊕DP = A \oplus B \oplus C \oplus D.
- **Parity Checker**:
  A parity checker ensures that the data received is correct by verifying that the number of 1s (including the parity bit) is correct (even or odd).

1.7 Decoder, Encoder, Multiplexer

- **Decoder**:
  A decoder takes a binary input and activates one of several output lines

based on the input value.

    - Example: A 2-to-4 decoder takes a 2-bit input and produces 4 output lines, each corresponding to one of the 4 possible input values.
- **Encoder**:
  An encoder performs the inverse operation of a decoder, converting a set of outputs into a binary input code.

    - Example: A 4-to-2 encoder has 4 input lines and 2 output lines, where the input corresponds to the binary code on the output.
- **Multiplexer**:
  A multiplexer is a circuit that selects one of many input signals and forwards it to the output.

    - Example: A 4-to-1 multiplexer has 4 input lines and 1 output line, where the input selected is determined by 2 control (selection) lines.

1.8 IC RAM, ROM, and Memory Organization

- **RAM (Random Access Memory)**:
  RAM is volatile memory used to store data and instructions that are currently being processed. It allows read and write operations.

- **ROM (Read-Only Memory)**:
  ROM is non-volatile memory used to store firmware or permanent data. Data stored in ROM cannot be modified easily.

- **Memory Organization**:
  Memory is organized into cells, each with an address, and can be accessed randomly in both RAM and ROM. Memory organization involves understanding how bits are stored and accessed in a memory device.

1.9 Sequential Circuits, State Transitions, and Flip-Flops

- **Sequential Circuits**:
  Sequential circuits have memory and their output depends not only on the current inputs but also on the past history of inputs. They are used to implement state machines.

- **State Transitions**:
  A state transition diagram shows how a system transitions from one state to another based on inputs.

- **Flip-Flops**:
  Flip-flops are basic memory elements that store one bit of data and can change states based on clock inputs. They are used in sequential circuits.

    - **RS Flip-Flop**:
      A flip-flop with two inputs: Set (S) and Reset (R). It stores data in binary form.

    - **JK Flip-Flop**:
      A more versatile flip-flop that can toggle its output based on input values J and K.

    - **D Latch**:
      A latch with a single data input (D). It stores the value of D when the clock is high.

    - **Master-Slave Flip-Flop**:
      A combination of two flip-flops used to create edge-triggered behavior, preventing race conditions.

Summary

1. **Boolean Algebra**: Fundamental laws used for simplifying digital expressions.
2. **Karnaugh Map**: A graphical method to simplify Boolean expressions.
3. **Adders**: Circuits used to perform binary addition, such as Half Adders and Full Adders.
4. **Memory and Logic Devices**: Including parity bit generators, decoders, multiplexers, RAM, ROM, and memory organization.
5. **Sequential Circuits and Flip-Flops**: Essential for designing systems with memory, including state transitions and various types of flip-flops.

Unit 2: Instruction Set Architecture (8L)

The **Instruction Set Architecture (ISA)** is the interface between software and hardware, defining the set of instructions a processor can execute. It provides a critical role in understanding how a computer system operates at a low level, particularly how data is accessed and manipulated within memory and how instructions are executed by the processor.

2.1 Memory Locations and Addresses

Memory in a computer system is organized in a structured manner, where each memory cell (or byte) is assigned a unique **address**. This organization facilitates access to data during program execution.

- **Byte Addressability**:
  Memory is byte-addressable, meaning each address refers to a single byte of memory. Each byte has a unique address, and when data is stored or retrieved, it is done in units of bytes (e.g., an 8-bit value). The byte addressability is the fundamental aspect of most modern computer systems.

- **Big-Endian vs. Little-Endian Assignments**:
  The **endianness** of a system defines how multi-byte data (e.g., 16-bit, 32-bit values) is stored in memory.

    - **Big-Endian**: The most significant byte (MSB) is stored first at the lowest address, followed by the less significant bytes.
        - Example: For a 32-bit value 0x123456780x12345678, it would be stored in memory as:
            - Address 0: 0x120x12
            - Address 1: 0x340x34
            - Address 2: 0x560x56
            - Address 3: 0x780x78
    - **Little-Endian**: The least significant byte (LSB) is stored first at the lowest address, followed by the more significant bytes.
        - Example: For the same 32-bit value 0x123456780x12345678, it would be stored in memory as:
            - Address 0: 0x780x78
            - Address 1: 0x560x56
            - Address 2: 0x340x34
            - Address 3: 0x120x12
    - **Impact of Endianness**:
      The difference in storage order can affect data interpretation when transferring data between systems with different endianness.

2.2 Word Alignment

Word alignment refers to the practice of arranging data in memory so that it is stored at an address that is a multiple of its size (e.g., a 4-byte word stored at an address that is a multiple of 4). Alignment can improve memory access speed and ensure that the processor can efficiently fetch data.

- **Aligned Access**:
  On many architectures, it is preferable to align data at word boundaries to ensure optimal performance. For example, a 4-byte word should be stored at a memory address that is a multiple of 4 (e.g., 0x1000, 0x1004).

- **Unaligned Access**:
  Some architectures allow unaligned access, where data can be stored at any address, but accessing misaligned data may incur a performance penalty or even an error in some systems.

2.3 Instructions and Instruction Sequencing

Instructions are the basic operations that a CPU can perform. Instruction sequencing refers to the order in which instructions are executed.

- **Instruction Format**:
  An instruction consists of an **opcode** (the operation to be performed) and **operand(s)** (the data or addresses involved in the operation).
  Example:

  - **Opcode**: ADD, MOV, SUB
  - **Operands**: Registers, memory addresses, constants
- **Instruction Pipeline**:
  Modern processors use instruction pipelines to improve execution speed. The pipeline breaks the instruction execution process into stages such as **fetch**, **decode**, **execute**, **memory access**, and **write-back**. This allows multiple instructions to be processed simultaneously, improving throughput.

- **Control Unit**:
  The control unit manages the sequencing of instructions, ensuring that the correct operation is performed at the right time.

2.4 Addressing Modes

Addressing modes define how the operand of an instruction is specified. There are several addressing modes that a processor can support, each providing a different way of calculating the effective address of the operand.

- **Immediate Addressing Mode**:
  The operand is a constant (literal) value, specified directly in the instruction.
    - Example: `MOV R1, #5` (R1 is loaded with the constant value 5)
- **Register Addressing Mode**:
  The operand is stored in a register.
    - Example: `MOV R1, R2` (R1 is loaded with the value stored in R2)
- **Direct Addressing Mode**:
  The address of the operand is given explicitly in the instruction.
    - Example: `MOV R1, [1000]` (R1 is loaded with the value stored at memory address 1000)
- **Indirect Addressing Mode**:
  The instruction specifies a register or memory location that contains the address of the operand.
    - Example: `MOV R1, [R2]` (R1 is loaded with the value stored at the memory address contained in R2)
- **Indexed Addressing Mode**:
  The address of the operand is calculated by adding an index (offset) to a base address.
    - Example: `MOV R1, [R2 + #4]` (R1 is loaded with the value at the address R2 + 4)
- **Register Indirect Addressing Mode**:
  The operand's address is located in a register, and the instruction accesses the memory at that address.
    - Example: `MOV R1, (R2)` (R1 is loaded with the value at the address contained in R2)
- **Base Register Addressing Mode**:
  The base register holds a base address, and an offset is added to determine the final operand address.
    - Example: `MOV R1, [R2 + R3]` (R1 is loaded with the value at the address R2 + R3)

2.5 Assembly Language

Assembly language is a low-level programming language that is closely tied to the machine architecture. It uses mnemonics to represent machine-level instructions and is specific to a processor's ISA.

- **Syntax**:
  Assembly instructions have a specific format:
  - **Mnemonic**: The operation to be performed (e.g., `MOV`, `ADD`).
  - **Operand(s)**: Data or memory addresses involved in the operation.
  - Example: `MOV R1, #10` (Load the value 10 into register R1)
- **Assembler**:
  An assembler translates assembly language code into machine code. It converts human-readable instructions into the binary form that the CPU can execute.

2.6 Subroutines

Subroutines (also called **functions** or **procedures**) are blocks of code designed to perform a specific task. They are used to avoid code duplication and enhance modularity.

- **Calling a Subroutine**:
  To call a subroutine, the program must pass control to the subroutine. This typically involves pushing the return address onto the stack.

- **Return from Subroutine**:
  After completing the task, the subroutine returns control to the calling program. This is done by popping the return address from the stack and jumping to that address.

- **Stack Usage**:
  The stack is used to store return addresses, local variables, and arguments when calling and returning from subroutines.

2.7 Additional Instructions

Some processors provide additional instructions to enhance functionality and performance. These might include:

- **Shift Instructions**:
  Shift operations move bits in a register to the left or right. For example, `SHL` (Shift Left) and `SHR` (Shift Right).

- **Rotate Instructions**:
  Rotate operations move bits around the edges of a register. For example, `ROL` (Rotate Left) and `ROR` (Rotate Right).

- **Jump Instructions**:
  Jump instructions control the flow of execution by altering the program counter (PC). For example, unconditional jumps (`JMP`) or conditional jumps (`JE` for "jump if equal").

2.8 Dealing with 32-Bit Immediate Values

Immediate values are constants directly included in the instruction. A **32-bit immediate value** refers to a 32-bit constant operand that can be used in an instruction.

- **Handling Large Constants**:
  Some architectures use special instructions or encoding schemes to allow the use of large immediate values, as they cannot always fit directly within the instruction format.

- **Sign Extension**:
  When a smaller immediate value is used in an instruction, it may need to be sign-extended to match the size of the destination operand. This ensures the value is correctly represented in the larger format.

- **Example**:
  In an instruction like `MOV R1, #0x12345678`, the immediate value `0x12345678` is a 32-bit constant that will be loaded into register R1.

Summary

- **Memory Locations & Addressing**: Understanding byte addressability, endianness, and memory alignment is crucial for managing data in memory.
- **Instructions & Addressing Modes**: The processor can execute various instructions using different addressing modes, allowing flexibility in how data is accessed.
- **Assembly Language**: Assembly provides a low-level view of the instructions that a processor can execute, making it crucial for understanding hardware behavior.
- **Subroutines**: Subroutines help organize and modularize code, making it reusable and easier to manage.
- **32-Bit Immediate Values**: Special handling is required for large constant values within instructions to ensure they are correctly interpreted and used.

Unit 3: Basic Processing Unit & Pipelining (8L)

The **Basic Processing Unit** (CPU) is the heart of a computer system, executing instructions and performing calculations. This unit plays a crucial role in determining the overall performance of a computer system. **Pipelining** is a technique used to improve the throughput and efficiency of the CPU by allowing multiple instruction stages to overlap.

3.1 Basic Processing Unit

The **Basic Processing Unit** refers to the central component of a computer responsible for fetching, decoding, and executing instructions. It comprises several key components that work together to process data efficiently.

Fundamental Concepts

The basic processing unit is designed to handle the following tasks:

- **Instruction Fetch**: The process of retrieving the next instruction from memory to be executed.
- **Instruction Decode**: The process of decoding the fetched instruction to understand what operation it represents and what operands it needs.
- **Instruction Execution**: The actual operation where the CPU performs calculations or manipulates data based on the decoded instruction.

Instruction Execution

The general flow of instruction execution in a CPU can be broken down into the following stages:

1. **Fetch**: The CPU retrieves the next instruction from memory.
2. **Decode**: The fetched instruction is decoded into its operational components (opcode, operand).
3. **Execute**: The instruction is executed. For example, an arithmetic instruction will perform the calculation, or a data transfer instruction will move data between memory and registers.
4. **Write-back**: The result of the execution is written back to a register or memory, depending on the operation.

These stages form the basic cycle known as the **fetch-decode-execute cycle**.

Hardware Components

A basic processing unit consists of several key hardware components:

- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logical operations (e.g., addition, subtraction, AND, OR).
- **Control Unit (CU)**: Directs the operations of the CPU by generating control signals that orchestrate the actions of the other components.
- **Registers**: Small, fast storage locations within the CPU used to store operands, results, and control information.
- **Memory Unit**: Stores data and instructions. The memory unit communicates with the CPU to provide data for computation.
- **Bus System**: A communication system that transfers data between the various components of the CPU and between the CPU and memory.

Instruction Fetch and Execution Steps

The process of executing an instruction involves several key steps:

1. **Instruction Fetch**:
   The Program Counter (PC) holds the address of the next instruction. The CPU fetches the instruction from this address and increments the PC to the next instruction's address.

2. **Instruction Decode**:
   The fetched instruction is decoded by the **control unit** to determine the operation to perform and the operands involved (whether they are in registers or memory).

3. **Operand Fetch**:
   If the instruction requires data from memory or registers, the operands are fetched and made ready for the execution phase.

4. **Execution**:
   The ALU performs the operation specified by the instruction. For arithmetic instructions, it might add or subtract values. For data transfer instructions, it moves data between memory and registers.

5. **Write-back**:
   The result of the execution is written back into the appropriate register or memory location.

Control Signals

Control signals are electrical signals sent from the **control unit** to the other components of the CPU to direct the execution of instructions. These signals help control which operations the ALU should perform, which registers should be accessed, and when data should be moved in and out of memory.

- **Hardwired Control**:
  In this approach, the control signals are generated by a fixed logic circuit. Hardwired control units are faster and simpler but less flexible.

- **Microprogrammed Control**:
  This approach uses a set of instructions (micro-operations) stored in memory to generate control signals. It's more flexible but slower compared to hardwired control.

CISC-Style Processors

- **CISC (Complex Instruction Set Computing)** processors have a large set of instructions, including multi-step operations (e.g., `ADD`, `SUB`, `MOV`, etc.). Each

instruction may perform multiple tasks, reducing the number of instructions needed for a program.
- **CISC processors** allow a high-level abstraction of operations but are often more complex and slower due to the intricate logic needed to implement complex instructions.

3.2 Pipelining

Pipelining is a technique that allows multiple instruction stages to be overlapped. It is similar to an assembly line, where each part of the process works simultaneously on different instructions, leading to improved performance and throughput.

Basic Concept of Pipelining

The concept of pipelining is based on dividing the execution of instructions into several stages:

1. **Fetch Stage**: Fetch the instruction from memory.
2. **Decode Stage**: Decode the instruction to understand the operation and operands.
3. **Execute Stage**: Perform the operation, such as an arithmetic calculation or data transfer.
4. **Memory Access Stage**: If required, access data from memory.
5. **Write-back Stage**: Write the result back to a register or memory.

These stages run in parallel for different instructions, meaning while one instruction is in the execute stage, another can be in the decode stage, and a third can be in the fetch stage, improving the throughput of the CPU.

Pipeline Organization

A typical pipeline consists of several stages, each stage handling a specific part of the instruction cycle. The processor works on multiple instructions at once by overlapping their execution. For example:

- Instruction 1 enters the **fetch stage**.
- Instruction 2 enters the **decode stage** while Instruction 1 is in the **fetch stage**.
- Instruction 3 enters the **execute stage** while Instruction 1 is in the **decode stage** and Instruction 2 is in the **fetch stage**.

This overlap leads to faster execution of multiple instructions.

Pipelining Issues

While pipelining improves throughput, it also introduces several challenges that must be addressed:

1. **Data Dependencies**:
   Pipelining can cause issues when one instruction depends on the result of a previous instruction. For example, if Instruction 2 needs the result of Instruction 1, but Instruction 1 is still in the pipeline, it creates a **data hazard**. These dependencies must be managed using techniques like forwarding or stalls.

2. **Memory Delays**:
   If an instruction needs to access memory, the memory latency can delay the pipeline stages that require the data. **Cache memory** is often used to reduce these delays by storing frequently accessed data close to the CPU.

3. **Branch Delays**:
   Branch instructions (e.g., `if` statements or loops) introduce uncertainty in the pipeline because the next instruction depends on the result of the branch decision. To deal with this, modern CPUs use techniques like **branch prediction** to predict the direction of the branch and minimize pipeline stalls.

Pipeline Performance Evaluation

The performance of a pipelined processor is evaluated by measuring the **throughput** and **latency**:

- **Throughput**: The number of instructions completed per unit of time. Pipelining improves throughput by processing multiple instructions simultaneously.

- **Latency**: The time taken to complete a single instruction. Although pipelining improves throughput, the latency for a single instruction may remain the same or even increase slightly due to pipeline stalls.

Performance is also evaluated based on the **pipeline depth** (the number of stages) and **pipeline efficiency** (how effectively the stages overlap).

Summary

1. **Basic Processing Unit**:
   The basic processing unit is responsible for executing instructions in a sequence of fetch, decode, execute, and write-back. Key components include the ALU, control unit, and registers.

2. **Pipelining**:
   Pipelining is a technique used to overlap the execution of multiple instructions to improve CPU throughput. It is achieved by dividing the instruction cycle into stages, allowing different instructions to be processed simultaneously.

3. **Challenges in Pipelining**:
   Issues like data dependencies, memory delays, and branch delays need to be addressed for effective pipelining. Techniques like forwarding, branch prediction, and cache memory help mitigate these issues.

Unit 4: Memory Organization (8L)

Memory organization refers to the structure and methods used to manage memory in computer systems. Effective memory organization is critical for performance optimization and resource utilization in computer systems.

4.1 Basic Concepts

Memory is the storage area where data and instructions are stored temporarily or permanently. It plays a key role in computer operations. Memory is categorized into **primary memory**, **secondary storage**, and **cache memory** based on their function and speed.

- **Primary Memory**: Includes RAM (Random Access Memory) and ROM (Read-Only Memory). It is directly accessible by the CPU.
- **Secondary Storage**: Non-volatile storage like HDDs, SSDs, and optical drives.
- **Cache Memory**: High-speed memory located close to the CPU to store frequently accessed data and instructions.

4.2 Semiconductor RAM Memories

RAM (Random Access Memory)

RAM is a type of volatile memory, meaning it loses its contents when the power is turned off. It is used as temporary storage for data and instructions that the CPU actively uses.

- **Dynamic RAM (DRAM)**:
  - Requires periodic refreshing to retain data.
  - Slower but cheaper and has higher density.
  - Used in main memory.
- **Static RAM (SRAM)**:
  - Does not require refreshing.
  - Faster but more expensive.
  - Used in cache memory.

4.3 Read-Only Memories

ROM (Read-Only Memory) is a non-volatile memory used to store firmware or permanent instructions for booting the computer.

- **Types of ROM**:
  - **PROM (Programmable ROM)**: Can be written once.
  - **EPROM (Erasable Programmable ROM)**: Can be erased with ultraviolet light and reprogrammed.
  - **EEPROM (Electrically Erasable Programmable ROM)**: Can be erased and rewritten electrically, used in BIOS.

4.4 Direct Memory Access (DMA)

Direct Memory Access allows peripherals to directly transfer data to/from memory without CPU intervention.

- **How DMA Works**:
  - The CPU initializes the transfer by providing the necessary parameters (source, destination, and size).
  - The DMA controller takes over and handles the data transfer while the CPU performs other tasks.
  - Once the transfer is complete, the DMA controller signals the CPU with an interrupt.

Benefits:

- Reduces CPU load.
- Improves data transfer speed for high-bandwidth devices like disk drives and network cards.

4.5 Memory Hierarchy

The **memory hierarchy** organizes memory into levels based on speed, cost, and capacity.

- **Registers**: Fastest and smallest storage within the CPU.
- **Cache Memory**: Close to the CPU; stores frequently used data and instructions.
- **Main Memory (RAM)**: Larger than cache but slower; stores active processes and data.
- **Secondary Storage**: Non-volatile storage for permanent data (e.g., HDD, SSD).

4.6 Cache Memories

Cache memory is a small, high-speed memory between the CPU and main memory.

- **Levels of Cache**:
    - **L1 Cache**: Closest to the CPU, smallest, and fastest.
    - **L2 Cache**: Larger but slightly slower.
    - **L3 Cache**: Shared by CPU cores, largest, and slower than L1 and L2.
- **Performance Improvement**:
  Cache reduces memory access time by storing copies of frequently accessed data.

4.7 Performance Considerations

Several factors affect memory performance:

1. **Access Time**: The time required to access data from memory.
2. **Memory Bandwidth**: The amount of data that can be transferred in a given time.
3. **Latency**: The delay between a request and the delivery of data.

4. **Hit Ratio**: The ratio of cache hits (data found in the cache) to total memory accesses. Higher hit ratios improve performance.

4.8 Virtual Memory

Virtual memory allows systems to use more memory than physically available by temporarily transferring data from RAM to a space on the disk (called the **page file**).

- **How It Works**:
    - Divides memory into fixed-sized pages.
    - Swaps pages between RAM and disk as needed.
- **Benefits**:
    - Provides an illusion of larger memory.
    - Enables multitasking by isolating processes.
- **Challenges**:
    - Increases disk I/O operations, which can slow down performance.

4.9 Memory Management Requirements

Efficient memory management ensures optimal resource utilization. Key functions include:

1. **Allocation and Deallocation**: Assigning and freeing memory for processes.
2. **Protection**: Preventing processes from accessing unauthorized memory.
3. **Swapping**: Moving processes between main memory and disk.
4. **Fragmentation Management**: Reducing memory wastage due to small, unusable memory chunks.

4.10 Secondary Storage

Secondary storage is non-volatile, used for permanent data storage.

- **Hard Disk Drives (HDDs)**: Magnetic storage devices with large capacity and slower access times.
- **Solid-State Drives (SSDs)**: Faster and more reliable than HDDs, with no moving parts.

- **Optical Storage**: CDs, DVDs, and Blu-ray discs for data distribution and backups.

Performance Factors:

- **Access Time**: Time to read/write data.
- **Transfer Rate**: Speed of data transfer.

Summary

1. **Memory Organization** includes multiple layers like cache, main memory, and secondary storage, each serving a specific purpose in the system.
2. **RAM and ROM** are primary memory types with distinct characteristics.
3. **Cache Memory** and **Virtual Memory** improve performance by minimizing access delays.
4. **DMA** enhances data transfer efficiency by bypassing the CPU.
5. **Memory Management** ensures system stability and performance by effectively allocating and protecting memory resources.

Unit 5: Input Output & Parallel Processing (8L)

5.1 Basic Input-Output (I/O)

Accessing I/O Devices

I/O devices interact with the CPU and memory to facilitate data transfer between the computer system and external peripherals.

- **Programmed I/O**: CPU actively monitors the device until the operation completes (polling).
- **Interrupt-Driven I/O**: Device notifies the CPU upon operation completion using interrupts.
- **Direct Memory Access (DMA)**: DMA controller handles data transfer directly between memory and the device, bypassing the CPU for efficiency.

Interrupts

Interrupts are signals from I/O devices that temporarily halt the CPU's execution to address the device's needs.

- **Types of Interrupts**:
    - **Hardware Interrupts**: Generated by external hardware (e.g., keyboard or printer).
    - **Software Interrupts**: Triggered by software instructions.
- **Interrupt Handling**:
    - CPU saves its current state.
    - Executes an Interrupt Service Routine (ISR) to process the interrupt.
    - Restores the state and resumes the previous task.

Input-Output Organization

The organization of I/O systems determines how devices interact with the CPU and memory.

1. **Bus Structure**
    - A communication system that transfers data between CPU, memory, and I/O devices.
    - **Types of Buses**:
        - **Data Bus**: Transfers data.
        - **Address Bus**: Carries the memory or device address.
        - **Control Bus**: Carries control signals like read/write.
2. **Bus Operation**
    - Involves communication between CPU and devices via data transfer, address decoding, and control signals.
3. **Bus Arbitration**
    - Resolves conflicts when multiple devices request bus access simultaneously.
    - **Arbitration Methods**:
        - **Centralized Arbitration**: Controlled by a central bus arbiter.
        - **Decentralized Arbitration**: Devices communicate directly to resolve conflicts.
4. **Interface**
    - Acts as a bridge between the CPU and I/O devices to match speed and operational differences.
    - Examples: PCI, USB, SATA interfaces.
5. **Interconnection Standards**
    - Standards define how devices are interconnected in a system.
    - Examples: USB, HDMI, Ethernet, PCIe.

5.2 Parallel Processing

Parallel processing enables the simultaneous execution of multiple instructions to improve performance and efficiency.

Hardware Multithreading

Multithreading involves multiple threads executing on a single CPU core.

- **Simultaneous Multithreading (SMT)**: Threads execute simultaneously within a core.
- **Hyper-Threading**: Intel's technology for logical core creation, enabling better utilization of CPU resources.

Vector (SIMD) Processing

- **SIMD (Single Instruction, Multiple Data)**: Executes the same instruction on multiple data simultaneously.
- Common in applications like image processing, scientific simulations, and graphics.
- **Vector Processors**: Designed for handling array-oriented operations efficiently.

Shared-Memory Multiprocessors

- Multiple processors share the same memory space and communicate via shared memory.
- **Challenges**:
    - Memory access contention.
    - Cache coherence (ensuring consistency of cached data).

Cache Coherence

Cache coherence ensures that multiple caches in a multiprocessor system have consistent data.

- **Protocols**:

    - **Write-Through**: Updates both cache and main memory simultaneously.
    - **Write-Back**: Updates cache first and main memory later.
    - **MESI Protocol**: Maintains cache consistency using states (Modified, Exclusive, Shared, Invalid).

Message-Passing Multicomputers

- Processors communicate through message-passing mechanisms rather than shared memory.
- Messages are exchanged via interconnection networks or buses.
- Common in distributed systems and clusters.

Parallel Programming for Multiprocessors

- Writing software to leverage the power of parallel processors.
- **Techniques**:
    - **Thread-Based Parallelism**: Threads share data and memory.
    - **Message Passing Interface (MPI)**: Communication between processors using messages.
    - **OpenMP**: Framework for shared-memory parallel programming.

Performance Modeling

Performance modeling evaluates the efficiency of parallel processing systems.

1. **Amdahl's Law**
    - Describes the maximum speedup achievable in a parallel system, considering the fraction of the program that cannot be parallelized.
2. **Gustafson's Law**
    - Explains how speedup increases with the size of the problem in parallel systems.
3. **Metrics**:
    - **Speedup**: Ratio of sequential execution time to parallel execution time.
    - **Efficiency**: Ratio of speedup to the number of processors.

Summary

1. **I/O Systems**: Include bus structures, arbitration, and interrupt handling to manage device interactions.
2. **Parallel Processing**: Involves multithreading, SIMD processing, and advanced processor designs for faster and efficient computation.
3. **Cache Coherence and Memory Models**: Are critical for ensuring data consistency in multiprocessor systems.
4. **Parallel Programming**: Leverages frameworks like MPI and OpenMP to utilize parallel hardware effectively.
5. **Performance Modeling**: Helps measure and optimize parallel system efficiency using speedup and efficiency metrics.