2. Digital Logic and Microprocessor

2.1 Digital Logic:

- Number systems, logic levels, logic gates, Boolean algebra.
- Sum-of-products and product-of-sums methods.
- Truth tables and Karnaugh maps.

2.2 Combinational and Arithmetic Circuits:

- Multiplexers, demultiplexers, decoders, and encoders.
- Binary addition and subtraction.
- Operations on signed and unsigned binary numbers.

2.3 Sequential Logic Circuits:

- RS flip-flops, gated flip-flops, edge-triggered flip-flops, and master-slave flip-flops.
- Types of registers and applications of registers.
- Asynchronous and synchronous counters.

2.4 Microprocessor:

- Internal architecture and features.
- Assembly language programming.

2.5 Microprocessor System:

- Memory device classification and hierarchy.
- Interfacing I/O and memory parallel interfaces.
- Introduction to PPI, serial interfaces, synchronous/asynchronous transmission, and DMA controllers.

2.6 Interrupt Operations:

- Interrupts, interrupt service routines, and interrupt processing.

2.1 Digital Logic

1. Number Systems, Logic Levels, Logic Gates, Boolean Algebra

1. **Number Systems**

Number systems are fundamental to both mathematics and computing, serving as the foundation for performing calculations and representing data. Different number systems use various **bases**, and each has specific uses depending on the application.

- **Binary System**: It is the **foundation of digital electronics** and computing. Every piece of data in computers (such as images, text, and sound) is eventually broken down into **binary code**, which consists of sequences of **0s and 1s**.
- **Decimal System**: It is the standard system used in everyday life for counting and arithmetic. It's a **base-10** system, meaning it uses 10 digits (0-9).

- **Octal System**: A base-8 number system using digits `0-7`. It is often used in computing as a shorthand for binary numbers.
- **Hexadecimal System**: A base-16 number system using digits `0-9` and letters `A-F` (representing values 10-15). It is commonly used in programming to represent binary data more compactly.

---

2. **Logic Levels**

In digital electronics and computing, **logic levels** represent binary states, which are fundamental to how information is processed and stored in digital circuits.

- **High (1)**: Represents a logic high or "true" state, typically corresponding to a voltage near the supply voltage (e.g., 5V).
- **Low (0)**: Represents a logic low or "false" state, typically corresponding to a ground or low voltage (e.g., 0V).

---

3. **Logic Gates**

Logic gates are the basic building blocks of digital circuits. They perform logical operations on one or more binary inputs to produce a binary output.

- **AND Gate**: Output is `1` only if both inputs are `1`.
- **OR Gate**: Output is `1` if at least one input is `1`.
- **NOT Gate (Inverter)**: Output is the inverse of the input. If input is `1`, output is `0`, and vice versa.
- **NAND Gate**: Output is the inverse of the AND gate. Output is `1` except when both inputs are `1`.
- **NOR Gate**: Output is the inverse of the OR gate. Output is `1` only when both inputs are `0`.
- **XOR Gate**: Output is `1` if the inputs are different.
- **XNOR Gate**: Output is `1` if the inputs are the same.

---

4. **Boolean Algebra**

Boolean algebra is a mathematical framework used to handle binary variables and logic operations. It forms the foundation for designing and analyzing digital circuits, computer algorithms, and programming logic. Boolean algebra involves variables that can have only two possible states: **0** (false) and **1** (true).

- **Boolean Variables**: These variables represent two possible states, `0` and `1`.
- **Basic Operations**:
  - **AND**: `A * B` or `A AND B`
  - **OR**: `A + B` or `A OR B`
  - **NOT**: `¬A` or `NOT A`
- **Boolean Laws**:
  - **Commutative**: `A + B = B + A`, `A * B = B * A`
  - **Associative**: `(A + B) + C = A + (B + C)`, `(A * B) * C = A * (B * C)`
  - **Distributive**: `A * (B + C) = (A * B) + (A * C)`

  - **Identity**: `A + 0 = A`, `A * 1 = A`
  - **Null**: `A + 1 = 1`, `A * 0 = 0`
  - **Complement**: `A + ¬A = 1`, `A * ¬A = 0`

---

2. Sum-of-Products and Product-of-Sums Methods

1. **Sum-of-Products (SOP)**

SOP is a Boolean expression where several product terms (AND operations) are summed (OR operations).

- **Example**: The Boolean expression `A * B + C` is in SOP form. The terms `A * B` and `C` are the product terms, and they are summed with the OR operator.
- **Application**: SOP is often used in designing digital circuits with AND and OR gates.

---

2. **Product-of-Sums (POS)**

POS is a Boolean expression where several sum terms (OR operations) are multiplied (AND operations).

- **Example**: The Boolean expression `(A + B) * (C + D)` is in POS form. The terms `(A + B)` and `(C + D)` are sum terms, and they are multiplied with the AND operator.
- **Application**: POS is used in digital circuit design when the expression needs to be implemented with NAND gates.

---

3. Truth Tables and Karnaugh Maps

1. **Truth Tables**

A **truth table** is a tabular representation of all possible input combinations and their corresponding outputs for a Boolean function or logic circuit.

- **Steps to Create a Truth Table**:
  1. List all possible input combinations.
  2. Determine the output for each combination based on the Boolean expression or circuit.
  3. Present the results in a table format.
- **Example** for a 2-input AND gate:

A	B	Output(A and B)
0	0	0
0	1	0
1	0	0
1	1	1

---

2. **Karnaugh Maps (K-map)**

A **Karnaugh map** is a graphical representation used to simplify Boolean expressions. It helps identify patterns in the truth table to minimize the Boolean expression.

- **Steps to Use K-map**:
  1. Construct a K-map grid with cells representing all possible input combinations.
  2. Place the output values from the truth table into the corresponding cells.
  3. Group adjacent cells with 1s in powers of two (1, 2, 4, 8, etc.).
  4. Write the simplified Boolean expression based on the grouped cells.
- **Example** for a 2-variable K-map:

A/B	0	1
0	0	1
1	1	1

The simplified Boolean expression for this K-map is: **A + B**.

Example: 2-Variable K-map

Let's work through an example to demonstrate the steps involved in simplifying a Boolean function using a K-map.

Given Boolean Expression:

F(A,B)=A'B+AB'

We need to simplify this expression using a K-map.

Step 1: Construct the K-map Grid

For a 2-variable Boolean function, the K-map will have four cells. Here's how it looks:

A/B	0	1
0	A'B'	A'B
1	AB'	AB

Step 2: Fill in the Output Values

Now, we need to fill in the K-map with the values from the given Boolean expression.

- A'B means A is 0 and B is 1, so place a 1 in the cell corresponding to A = 0 and B = 1.
- AB' means A is 1 and B is 0, so place a 1 in the cell corresponding to A = 1 and B = 0.

The K-map looks like this:

A/B	0	1
0	0	1
1	1	0

Step 3: Group Adjacent 1s

Now, let's group the 1s:

- We have two 1s in the K-map: one at A=0,B=1 and one at A=1,B=0.
- These two 1s form a pair. This is a "2-cell" group.

Step 4: Write the Simplified Boolean Expression

For the 2-cell group, look at the variables:

- (A is 0 in one cell and 1 in the other), so we **include A**.
- (B is 0 in one cell and 1 in the another).so we **include B** too.

Thus, the simplified expression is **A + B**.

Benefits of Using K-maps:

- **Simplification**: K-maps provide a straightforward method for minimizing Boolean expressions, especially when dealing with 2-4 variables.
- **Reduction in Circuit Complexity**: The simplified Boolean expressions result in fewer logic gates, making digital circuits more efficient and cost-effective.
- **Visualization**: K-maps provide a visual way to group terms and easily spot patterns that lead to simplifications.

Generalization for More Variables:

K-maps can be expanded to more variables, such as 3-variable (8 cells) and 4-variable (16 cells). K-maps, which work in the same way but involve larger grids and more complex groupings. The process remains the same: fill in the K-map, group adjacent 1s, and write the simplified Boolean expression based on those groups. Simplified Boolean expression means, minimum gates involved to design.

Conclusion

Understanding number systems, logic levels, gates, and Boolean algebra is fundamental to digital electronics. Sum-of-Products (SOP) and Product-of-Sums (POS) methods simplify logic expressions for circuit design. Truth tables outline all input-output possibilities, while Karnaugh maps minimize Boolean expressions, optimizing circuit efficiency. These concepts enable the design of reliable and efficient digital systems.

2.2 Combinational & Arithmetic Circuit

1. Multiplexers, Demultiplexers, Decoders, and Encoders

1. **Multiplexers (MUX)**

A **multiplexer** is a combinational circuit that selects one of many inputs and forwards it to a single output line.

- It has **n** data inputs, 1 output, and $\log_2(n)$ selection lines.

- **Example**: A 4-to-1 multiplexer has 4 data inputs, 1 output, and 2 selection lines (S_1, S_0).

Truth Table for 4-to-1 MUX:

S_1	S_0	Output
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

---

2. **Demultiplexers (DEMUX)**

A **demultiplexer** takes a single input and routes it to one of many output lines.

- It has 1 input, **n** outputs, and $\log_2(n)$ selection lines.
- **Example**: A 1-to-4 demultiplexer takes 1 input and routes it to one of the 4 output lines based on the selection lines (S_1, S_0).

Truth Table for 1-to-4 DEMUX:

S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

---

3. **Decoders**

A **decoder** is a combinational circuit that decodes an n-bit input to a corresponding 2^n output.

- It converts binary information into a specific code.
- **Example**: A 2-to-4 decoder takes a 2-bit input and produces 4 outputs, with one of the outputs being 1 based on the input.

Truth Table for 2-to-4 Decoder:

A_1	A_0	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

---

4. **Encoders**

An **encoder** is the reverse of a decoder. It encodes an **n**-input into a

$$\log_2(n)$$

-bit output.

- **Example**: A 4-to-2 encoder takes 4 inputs and produces a 2-bit binary code corresponding to the active input.

Truth Table for 4-to-2 Encoder:

D_3	D_2	D_1	D_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

---

2. Binary Addition and Subtraction

1. **Binary Addition**

Binary addition is similar to decimal addition but operates with base 2 (digits `0` and `1`).

- **Rules for binary addition**:
  - `0 + 0 = 0`
  - `0 + 1 = 1`
  - `1 + 0 = 1`
  - `1 + 1 = 10` (carry `1` to the next higher bit)

2. **Binary Subtraction**

Binary subtraction follows the same concept as decimal subtraction.

- **Rules for binary subtraction**:
  - `0 - 0 = 0`
  - `1 - 0 = 1`
  - `10 - 1 = 1` (borrow `1` from the next higher bit)
  - `1 - 1 = 0`

---

3. Operations on Signed and Unsigned Binary Numbers

1. **Unsigned Binary Numbers**

In unsigned binary, all bits are used to represent the value. There is no sign bit, so the number can only represent non-negative values.

- **Example**: `1011` (binary) = `11` (decimal)

2. **Signed Binary Numbers**

In signed binary, one bit is used as the sign bit, where `0` represents positive and `1` represents negative.

- **Two's Complement Representation**: Common method to represent signed binary numbers.
- To find the two's complement of a binary number, invert all bits and add `1` to the least significant bit (LSB).

Example: Representing `-5` in 4-bit two's complement:

- Step 1: Convert `5` to binary: `0101`
- Step 2: Invert all bits: `1010`
- Step 3: Add `1`: `1011` (This is `-5` in two's complement).

Complements in Binary and Decimal Systems

Complements are useful in binary and decimal systems for performing arithmetic operations like subtraction using addition. Below are the explanations for **One's Complement**, **Two's Complement**, **Nine's Complement**, and **Ten's Complement**.

---

1. **Subtraction: (5 - 3) using Two's Complement**

We know that subtraction can be represented as:

- $$5 - 3 = 5 + (-3)$$

Step-by-step:

Step	Action	Explanation	Binary
1.	Represent 5 in binary	Convert 5 to binary (4 bits)	(0101)
2.	Convert 3 to binary	Convert 3 to binary (4 bits)	(0011)
3.	Two's complement of 3	Invert bits of 3 (0011) and add 1	Invert: (0011 to 1100),then add 1: (1100 + 1 = 1101)
4.	Add 5 and -3	Perform binary addition of (0101 + 1101)	(10010)
5.	Discard the carry	Ignore the carry-over (MSB is discarded)	The result after discarding the carry: (0010)
6.	Final result	Interpret the result	(0010) is the binary representation of (2).Thus, (5 - 3 = 2).

---

2. **Subtraction: (3 - 5) using Two's Complement**

We know that subtraction can be represented as:

- $$3 - 5 = 3 + (-5)$$

Step-by-step:

Step	Action	Explanation	Binary
1.	Represent 3 in binary	Convert 3 to binary (4 bits)	(0011)
2.	Convert 5 to binary	Convert 5 to binary (4 bits)	(0101)
3.	Two's complement of 5	Invert bits of 5 (0101) and add 1	Invert: (0101 to 1010),then add 1: (1010 + 1 = 1011)
4.	Add 3 and -5	Perform binary addition of (0011 + 1011)	(1110)
5.	Interpret the result	Convert the result from two's complement to decimal	Since the MSB is 1, the number is negative.Invert (1110 to 0001), add 1: (0001 + 1 = 0010), which is 2.The result is negative, so it's (-2).Thus, (3 - 5 = -2).

Summary Table:

Subtraction	Action	Binary Result	Decimal Result
(5 - 3)	(5 + (-3))	(0010)	(2)
(3 - 5)	(3 + (-5))	(1110)	(-2)

Key Difference Between 1's & 2's Complement

In 1's complement, if there is a **carry-out** (i.e., an extra 1 in the 5th bit position), we discard the carry and add it back to the result (this is called **end-around carry**).

For 1's Complement:

- When the MSB is **1**, the number is negative.
- To convert it to a positive value, you **invert all the bits**.

For 2's Complement:

- When the MSB is **1**, the number is negative.

- To find the positive equivalent, you **invert all the bits** and **add 1** to the result.

Key takeaway:

- **MSB = 1** always indicates a negative number in both 1's complement and 2's complement.
- The difference is in how you process the negative number (invert bits and add 1 for 2's complement).

---

3. **Subtraction: (5 - 3) using 10's Complement**

We know that subtraction can be represented as:

- $$5 - 3 = 5 + (-3)$$

Step-by-step:

Step	Action	Explanation	Decimal	Binary
1.	Represent 5 in binary	Convert 5 to binary (4 digits)	(5)	(0101)
2.	Represent 3 in binary	Convert 3 to binary (4 digits)	(3)	(0011)
3.	10's complement of 3	Find 9's complement and add 1: (9 - 3 = 6); then add 1: (6 + 1 = 7)	(-3)	(0111)
4.	Add 5 and -3	Add the binary of 5 and the 10's complement of 3: (0101 + 0111)	(5 + (-3))	(1100)
5.	Interpret the result	Since there's no carry, the result is the answer	(2)	(0010)

Final result:

- $$5 - 3 = 2 \quad \text{(in decimal, binary 0010)}$$

---

4. **Subtraction: (3 - 5) using 10's Complement**

We know that subtraction can be represented as:

- $$3 - 5 = 3 + (-5)$$

Step-by-step:

Step	Action	Explanation	Decimal	Binary
1.	Represent 3 in binary	Convert 3 to binary (4 digits)	(3)	(0011)
2.	Represent 5 in binary	Convert 5 to binary (4 digits)	(5)	(0101)
3.	10's complement of 5	Find 9's complement and add 1: (9 - 5 = 4);then add 1: (4 + 1 = 5)	(-5)	(0101)
4.	Add 3 and -5	Add the binary of 3 and the 10's complement of 5: (0011 + 0101)	(3 + (-5))	(1000)
5.	Interpret the result	Since the result is negative, invert (1000) (1's complement) to get (0111),then add 1: (0111 + 1 = 1000), so the result is (-2)	(-2)	(1110)

Final result:

- $$3 - 5 = -2 \quad \text{(in decimal, binary 1110)}$$

Summary Table:

Subtraction	Action	Binary Result	Decimal Result
(5 - 3)	(5 + (-3))	(0010)	(2)
(3 - 5)	(3 + (-5))	(1110)	(-2)

Key Difference Between 9's & 10's Complement

In 9's complement, if there is a carry-out (i.e., an extra 1 in the 5th bit position), we discard the carry and add it back to the result (this is called end-around carry).

For 9's Complement:

- When the **MSB** is **1**, the number is **negative**.
- To convert it to a positive value, you **subtract each bit from 9**. This means inverting each bit (for a 4-bit number, subtracting each bit from 9).

For 10's Complement:

- When the **MSB** is **1**, the number is **negative**.
- To find the positive equivalent, you **subtract each bit from 9**, and then **add 1** to the result. This is similar to how 2's complement works but with 9's complement instead.

Key takeaway:

- **MSB = 1** always indicates a **negative** number in both **9's complement** and **10's complement**.
- The difference is in how you process the negative number:
    - **9's complement**: Subtract each bit from 9.
    - **10's complement**: Subtract each bit from 9 and add 1 to the result.

Conclusion

- Multiplexers (MUX), Demultiplexers (DEMUX), Decoders, and Encoders are essential components in digital circuits for managing data flow.
- A multiplexer selects **one** input from multiple inputs and forwards it to a **single output** based on the control signals (selection lines). A demultiplexer takes **one** input and routes it to **one of several outputs** based on the control signals (selection lines). It essentially performs the inverse operation of a multiplexer.
- Decoders convert binary inputs to specific outputs, and Encoders perform the reverse.
- Binary addition and subtraction follow base-2 rules with carry and borrow operations, similar to decimal.
- Complements (1's, 2's, 9's, and 10's) are techniques used to simplify subtraction in both binary and decimal systems by converting subtraction into addition.

2.3 Sequential Logic Circuits

Introduction

There are two different types of digital circuits, combinational and sequential circuits. The combinational circuit generates an output signal based on its current input state. It does not need any kind of triggering pulse called a clock pulse. Whereas the sequential circuit changes output in the presence of the clock pulse.

A **sequential circuit** generates output based on its previous output state and current input state. It consists of a combinational circuit with a memory unit such as a flip-flop or latch. Flip-flops are edge-sensitive and latches are level-sensitive. The memory unit is used to provide feedback.

Edge triggering and level triggering are two different types of triggering methods used in digital circuits. It enables the circuit to initiate the output signal transition from one state to another. These both kinds of triggering are equally important and used to date.

1. RS Flip-Flops, Gated Flip-Flops, Edge-Triggered Flip-Flops, and Master-Slave Flip-Flops

1. **RS Flip-Flop**

An **RS (Reset-Set) flip-flop** is a basic bistable multivibrator that stores a single bit of data. It has two inputs:

- **Set (S):** Used to set the output to 1
- **Reset (R):** Used to reset the output to 0

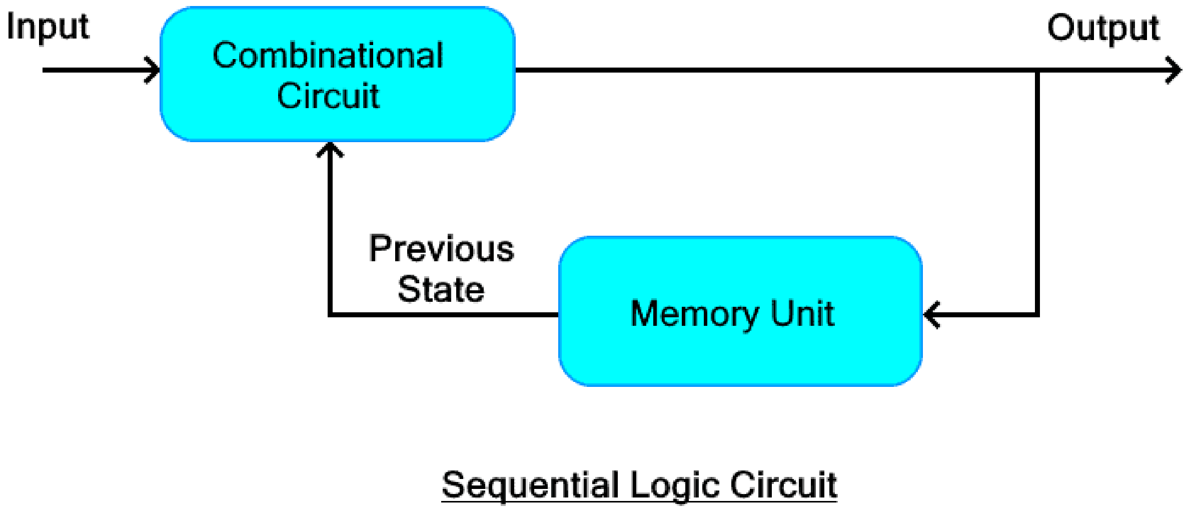

Figure 1: Sequential Logic Circuit

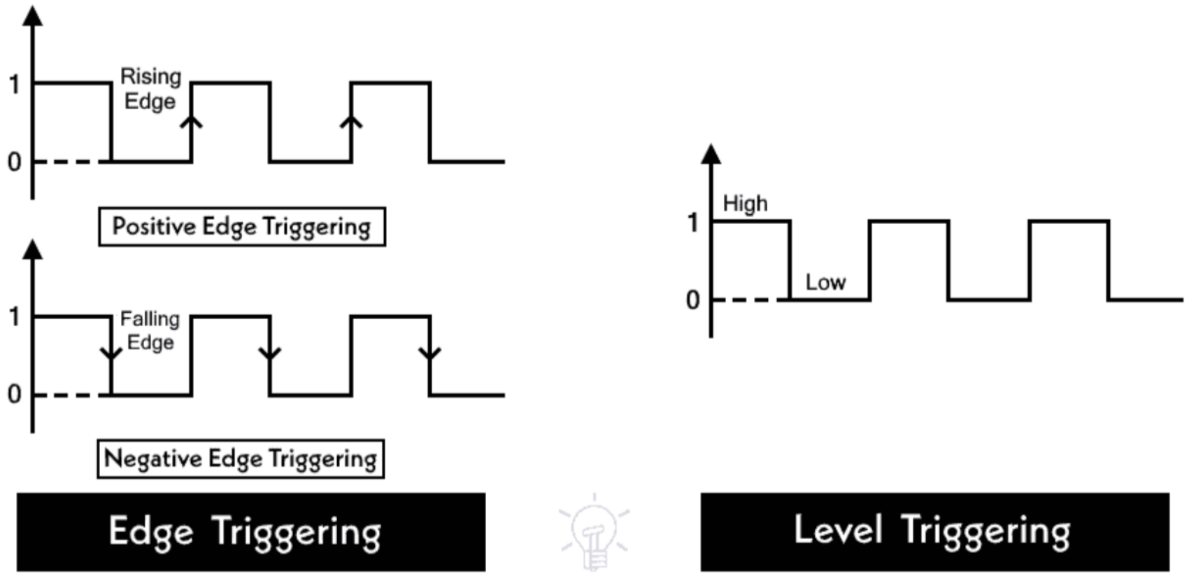

Figure 2: Edge and Level Triggering

The two outputs are:

- **Q:** Normal output
- **Q' (Q bar):** Inverted output

Working Principle:

- When **S = 1 and R = 0**, Q becomes 1 (set state).
- When **S = 0 and R = 1**, Q becomes 0 (reset state).
- When **S = 0 and R = 0**, the output remains unchanged.
- When **S = 1 and R = 1**, this state is invalid and should be avoided.

Truth Table:

S	R	Q (Next State)	Q'
0	0	No Change	No Change
0	1	0 (Reset)	1
1	0	1 (Set)	0
1	1	Invalid	Invalid

Implementation: An RS flip-flop can be built using either **NOR gates** or **NAND gates**.

---

2. **Gated Flip-Flops**

A **gated flip-flop** is an RS flip-flop with an additional **Enable (G) input**. This input acts as a control signal, ensuring that the flip-flop only operates when enabled.

Working Principle:

- When **G = 1**, the flip-flop functions as a normal RS flip-flop.
- When **G = 0**, the flip-flop holds its previous state, regardless of S and R inputs.

Truth Table:

G	S	R	Q (Next State)	Q'
0	X	X	No Change	No Change
1	0	0	No Change	No Change
1	0	1	0 (Reset)	1
1	1	0	1 (Set)	0
1	1	1	Invalid	Invalid

Circuit Implementation: Gated flip-flops are implemented by adding **AND gates** before the inputs of an RS flip-flop.

---

3. **Edge-Triggered Flip-Flops**

An **edge-triggered flip-flop** changes state only on a **clock signal transition**, either rising edge ($0 \rightarrow 1$) or falling edge ($1 \rightarrow 0$). This ensures precise timing in synchronous circuits.

Types:

1. **Positive Edge-Triggered Flip-Flop:** Activates on the rising edge.
2. **Negative Edge-Triggered Flip-Flop:** Activates on the falling edge.

Truth Table (Positive Edge-Triggered D Flip-Flop):

Clock Edge	D	Q (Next State)	Q’
↑	0	0	1
↑	1	1	0

Working Principle:

- Data is captured only at the exact moment of a **clock edge**.
- This prevents unwanted changes between clock pulses.

Usage: Edge-triggered flip-flops are widely used in **processors, registers, and counters** to ensure accurate data storage and transfer.

---

4. **Master-Slave Flip-Flops**

A **master-slave flip-flop** consists of two flip-flops connected in series:

- **Master flip-flop:** Captures input when the clock is HIGH.
- **Slave flip-flop:** Updates output when the clock goes LOW.

This design helps to **avoid timing issues and race conditions** in sequential circuits.

Working Principle:

- The **master** stores data during the clock pulse (HIGH state).
- The **slave** transfers data to the output on the next clock pulse (LOW state).

Truth Table (Master-Slave D Flip-Flop):

Clock	D	Master Q	Slave Q (Final Output)
0	X	No Change	No Change
↑	0	0	Previous State
↓	0	Previous State	0
↑	1	1	Previous State
↓	1	Previous State	1

Advantages:

- **Prevents glitches** by separating input capture and output changes.
- **Ensures stable and synchronized outputs**.
- Used in **shift registers and counters**.

These flip-flops form the foundation of digital circuit design, ensuring proper data flow and synchronization in modern computing systems.

---

2. Types of Registers and Applications

Registers are crucial in digital circuits as temporary storage elements. They hold binary data for a short time, often used for fast data manipulation and communication between different components in a system.

The key types of registers are:

1. **Shift Registers:**

Shift registers move binary data either to the left or right by a certain number of positions. Each shift operation moves data by one bit at a time.

- **How it works:** The bits of data in a shift register move through a series of flip-flops or memory cells. The shift can occur in two main directions:
  - **Left Shift**: Data is moved towards the most significant bit (MSB) end, and new data enters from the least significant bit (LSB) end.
  - **Right Shift**: Data is moved towards the least significant bit (LSB) end, and new data enters from the MSB end.
- **Use Cases:** Shift registers are used in applications that involve serial-to-parallel or parallel-to-serial data conversion.
- **Applications:**
  - Data transfer (serial to parallel or vice versa).
  - Temporary data storage.
  - Used in ADC/DAC for converting data.
  - Pulse shaping in signal processing.

---

2. **Parallel Registers:**

These registers store multiple bits of data at the same time. Each bit of data is stored in a separate memory cell (flip-flop).

- **How it works:** Data is written to or read from the entire register simultaneously. A parallel register can hold n bits of data, with each bit in a different position.
- **Use Cases:** Parallel registers are useful in applications where you need to access multiple bits of data simultaneously, such as in multi-bit digital systems.
- **Applications**:
  - Fast data storage and transfer.
  - Simultaneous data writing/reading.
  - Input/output operations in digital systems.

---

3. **Serial Registers:**

Serial registers store data one bit at a time in sequence, unlike parallel registers, which store multiple bits.

- **How it works:** In serial registers, data is shifted into or out of the register one bit at a time, and each bit is processed in sequence.
- **Use Cases:** Serial registers are commonly used in systems where data transfer occurs bit by bit over a single data line, such as in shift registers or serial communication protocols (e.g., UART, SPI).
- **Applications**:
  - Data transfer in serial communication (e.g., UART, SPI).
  - Sequential data storage.
  - Signal processing in systems with limited I/O.

---

3. Asynchronous and Synchronous Counters

1. **Asynchronous Counters**

Asynchronous counters, often called **ripple counters**, work in a way where each flip-flop (FF) in the counter is triggered by the output of the previous flip-flop. Here's a more detailed explanation:

- **Sequential Triggering**: In an asynchronous counter, the flip-flops are not triggered by the same clock signal at the same time. Instead, each flip-flop is triggered by the **output** of the preceding flip-flop. The first flip-flop receives the clock signal directly, and each subsequent flip-flop gets triggered by the output of the previous one.
- **Ripple Effect**: This sequential triggering creates what is called the **ripple effect**. The ripple effect occurs because the change in the state of each flip-flop "ripples" through the counter, triggering the next flip-flop. This leads to a slight delay in the operation because each flip-flop waits for the previous one to update before changing its state.
- **Slower Operation**: Since the flip-flops are not all triggered simultaneously, the counter is slower in operation, especially as the number of bits in the counter increases. Each flip-flop has to wait for the previous one to change before it can update, leading to propagation delay across the flip-flops.
- **Example (4-bit Asynchronous Binary Counter)**:
  - For a 4-bit binary counter, it counts from `0000` to `1111`.
  - The first flip-flop (representing the least significant bit) toggles on each clock pulse, and each subsequent flip-flop toggles based on the output of the flip-flop before it.
  - This sequential triggering leads to delays, making it slower than synchronous counters.

---

2. **Synchronous Counters**

Synchronous counters, on the other hand, are faster and more efficient because all the flip-flops are triggered simultaneously by the same clock signal. Here's how they work:

- **Simultaneous Triggering**: In a synchronous counter, all flip-flops are connected to the same clock pulse. This means that they all toggle (change state) at the same time, and there is no waiting for one flip-flop to trigger the next.
- **No Ripple Effect**: Since all flip-flops receive the clock pulse at the same time, there is no ripple effect. The counter operates in a more synchronized manner, which speeds up the overall counting process.

- **Faster Operation**: Because there is no delay between the flip-flops, synchronous counters are much faster than asynchronous counters. The entire counter changes its state in one clock cycle, making it more suitable for high-speed applications.
- **Example (4-bit Synchronous Binary Counter)**:
  - For a 4-bit binary counter, all flip-flops toggle on the same clock signal, so the counter counts from `0000` to `1111` without the delays caused by the ripple effect.
  - Each flip-flop is controlled by the same clock, and the counting happens in a synchronous manner, making the operation faster than in an asynchronous counter.

Conclusion

Flip-flops store and change data based on inputs, used in sequential logic circuits. RS, gated, edge-triggered, and master-slave flip-flops serve different purposes in timing and data synchronization. Registers, including shift registers, store and transfer data, while counters (asynchronous and synchronous) count in binary, with synchronous ones being faster.

2.4 Microprocessor

1. Internal Architecture and Features

8085 microprocessor

It is an 8-bit microprocessor designed by **Intel** in 1977 using **NMOS technology**.

Configuration:

- **8-bit data bus**
- **16-bit address bus**, which can address up to **64KB**
- **16-bit program counter**
- **16-bit stack pointer**
- **Six 8-bit registers** arranged in pairs: **BC**, **DE**, **HL**
- Requires **+5V** supply to operate at **3.2 MHz** single-phase clock
- Used in applications like **washing machines**, **microwave ovens**, **mobile phones**, etc.

Microprocessor Architecture

Understanding the architecture flow of 8085 microprocessor

The architecture of the 8085 microprocessor is designed to execute instructions in a sequence of operations, including fetching, decoding, and executing, while handling external events such as interrupts. Here's a step-by-step explanation of the flow and components involved:

---

1. Fetch Phase

- **Program Counter (PC):**
  - The PC holds the address of the next instruction to be executed.

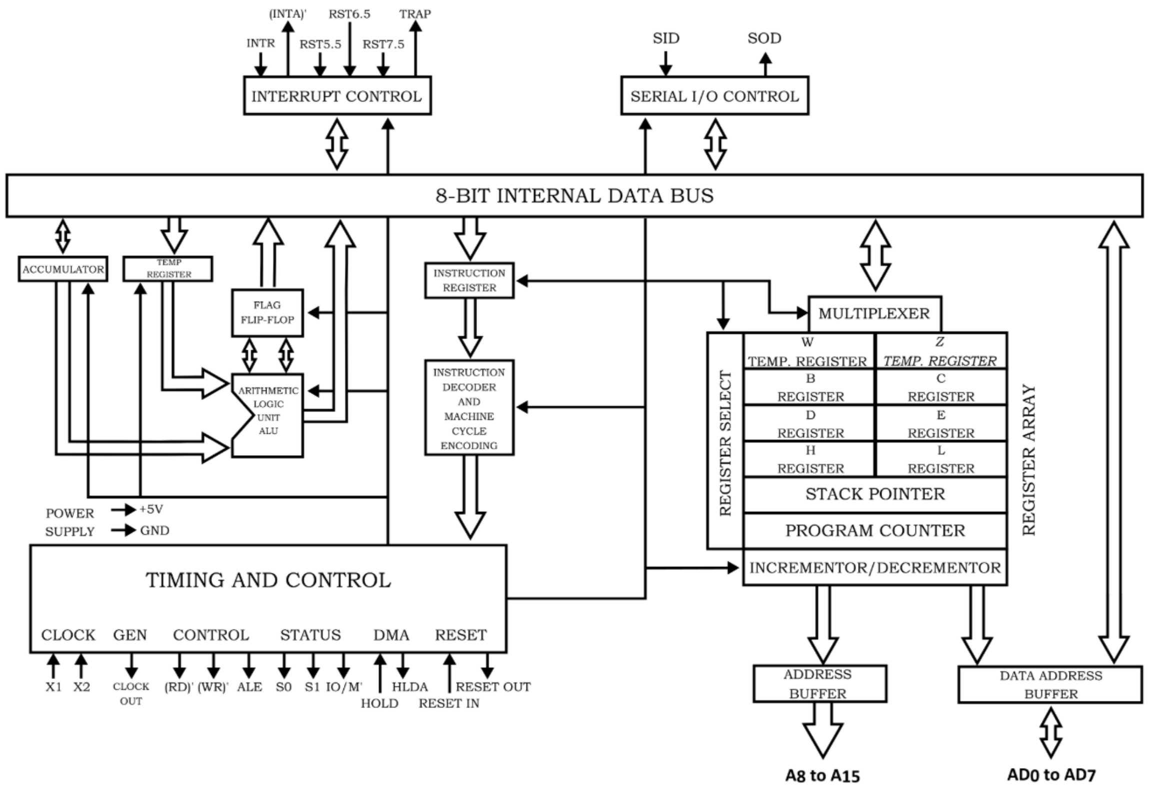

Figure 3: Architecture Diagram of 8085 microprocessor

  - It sends this address via the **Address Bus** to the memory.
- **Memory Unit:**
  - The memory receives the address and places the corresponding instruction on the **Data Bus**.
  - This instruction is then sent to the **Instruction Register (IR)** in the microprocessor.
- **Instruction Register (IR):**
  - The instruction fetched from memory is stored in the IR.
  - This marks the end of the fetching phase.

---

2. **Decode Phase**

- **Instruction Decoder:**
  - The instruction stored in the IR is decoded by the **Instruction Decoder**.
  - The decoder interprets the opcode and determines the operations to be performed.
- **Timing and Control Circuit:**
  - Based on the decoded instruction, the **Timing and Control Circuit** generates appropriate control signals.
  - These signals direct the flow of data between registers, memory, and the **Arithmetic and Logic Unit (ALU)**.

---

3. **Execute Phase**

- **Control Signals:**
  - The control signals guide the flow of data:
    * **Registers:** Data required for the operation is fetched from the appropriate register.
    * **ALU:** The ALU performs the necessary calculation or logical operation. ALU role is to generate the result and also the status. Result stored on the accumulator while status if exist in the flag.
- **Accumulator (A):**
  - The result of the operation is stored in the **Accumulator**, which serves as a primary data register for the microprocessor.
- **Flags Register:**
  - The **Flags Register** is updated based on the result in the accumulator.
  - It holds information about the status of the result, such as:
    * Zero (Z), Sign (S), Carry (CY), Parity (P), and Auxiliary Carry (AC) flags.

---

4. **Program Counter Increment**

- **Increment/Decrement Circuit (ICR/DCR):**
  - After fetching an instruction, the **Increment/Decrement Circuit** increments the PC to point to the next instruction in sequence.
  - Similarly, it also manages the stack pointer during stack operations by incrementing/decrementing the address.

---

5. **Interrupts**

- **Interrupt Handling:**
  - If an interrupt is triggered, the current operation is paused, and the processor executes the **Interrupt Service Routine (ISR)**.
  - After handling the interrupt, the processor resumes the normal execution flow.

---

6. Temporary Register (WZ)

- The **WZ Register Pair** is used internally to hold temporary data during multi-step operations.
  - For example, in memory-related operations, WZ temporarily holds intermediate addresses or values.

8086 Microprocessor

It is an enhanced version of 8085 Microprocessor that was designed by Intel in 1976. It is a 16-bit microprocessor designed by Intel in 1978 using HMOS technology.

Configuration:

- 16-bit data bus
- 20-bit address bus, which can address up to 1MB
- 16-bit instruction pointer (IP) and stack pointer (SP)
- Four 16-bit general-purpose registers: AX, BX, CX, DX
- Four segment registers: CS, DS, SS, ES (64KB each)
- 6-byte instruction queue for pipelined execution
- Requires +5V supply to operate at 5-10 MHz
- Used in applications like personal computers, robotics, industrial control systems, and embedded systems.

Understanding the architecture flow of 8086 microprocessor

Here's a step-by-step explanation of the flow and components involved:

1. Fetch Phase

- **Program Counter (IP - Instruction Pointer):**
  - Holds the **offset address** of the next instruction within the **Code Segment (CS)**.
  - The **CS:IP pair** forms the full 20-bit physical address of the instruction in memory.
- **Bus Interface Unit (BIU):**
  - Sends this address to the memory via the **Address Bus**.
  - Fetches the instruction and stores it in the **Instruction Queue (6-byte FIFO buffer)**.
- **Instruction Queue:**
  - While the **Execution Unit (EU)** is decoding and executing the current instruction, the BIU fetches the next instruction to optimize processing time.

---

2. Decode Phase

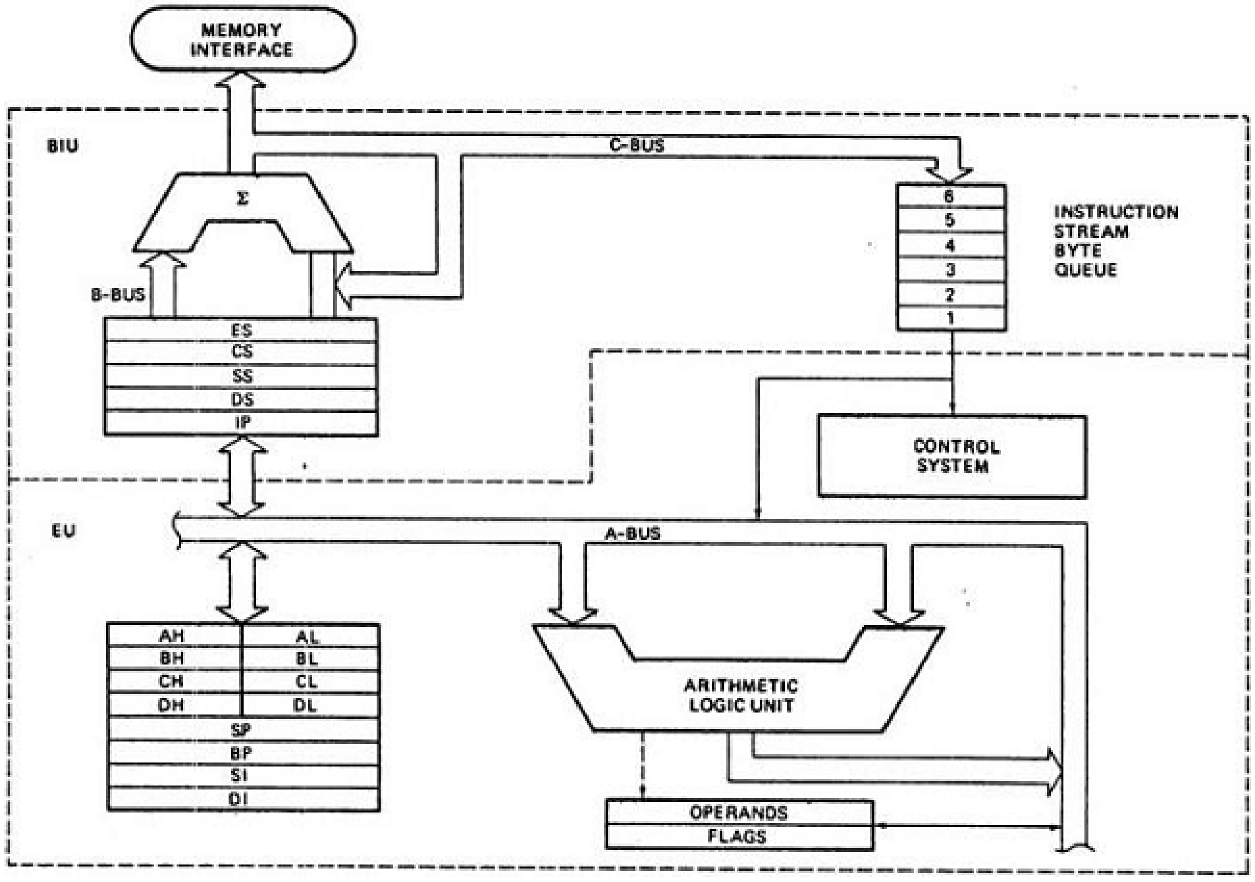

Figure 4: Architecture of 8086 Microprocessor

- **Instruction Decoder (Part of EU):**
  - Fetches the instruction from the **Instruction Queue**.
  - Decodes the operation (opcode) and determines the required operands, addressing modes, and control signals.
  - Prepares the **Timing and Control Unit** to generate signals for data transfer and ALU operations.

---

3. Execute Phase

- **Execution Unit (EU):**
  - The decoded instruction guides the EU to perform the operation:
    * Data is fetched from the **General Registers**, **Memory**, or **I/O Ports** as needed.
    * The **Arithmetic and Logic Unit (ALU)** performs calculations or logical operations.
    * Results are stored in the appropriate register or memory.
- **Flags Register:**
  - The **Flags Register** is updated based on the result of the operation, reflecting the status (e.g., Zero, Carry, Overflow).
- **Stack Pointer (SP):**
  - If the operation involves function calls or interrupts, the SP manages **push/pop operations** to save or restore return addresses and data.
- **Interrupts (if any):**
  - If an interrupt occurs, the current instruction is paused, the context is saved on the **Stack**, and the **Interrupt Service Routine (ISR)** executes.

---

4. Fetch-Execute Overlap (Pipelining)

- While the EU decodes and executes the current instruction, the **BIU fetches the next instruction**, maintaining an overlap that speeds up processing.

Comparison between 8085 & 8086 Microprocessor

- Size – 8085 is 8-bitmicroprocessor, whereas 8086 is 16-bit microprocessor.
- Address Bus – 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- Memory – 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- Instruction – 8085 doesn't have an instruction queue, whereas 8086 has an instruction queue.
- Pipelining – 8085 doesn't support a pipelined architecture while 8086 supports a pipelined architecture.
- I/O – 8085 can address

  $$2^8$$

  = 256 I/O's, whereas 8086 can access

  $$2^{16}$$

  = 65,536 I/O's.
- Cost – The cost of 8085 is low whereas that of 8086 is high.

Features of a Microprocessor

- **Clock Speed**: Determines the speed at which the microprocessor can execute instructions. Measured in Hertz (Hz).
- **Instruction Set**: A set of instructions that the microprocessor can understand and execute.
- **Address Bus**: Carries addresses from the microprocessor to memory or peripherals.
- **Data Bus**: Transfers data between the microprocessor, memory, and I/O devices.
- **Control Bus**: Manages the control signals for read/write operations, memory access, etc.
- **Interrupt Handling**: Microprocessors support interrupts, allowing them to respond to external events or devices asynchronously.

---

2. Assembly Language Programming

Assembly language is a low-level programming language that provides a direct correspondence between the instructions written by the programmer and the machine code understood by the microprocessor.

- Each instruction in assembly language corresponds to a single machine code instruction, making it more efficient than high-level programming languages.
- Assembly language uses **mnemonics** (symbolic representations) to represent machine instructions, making it easier for humans to write and understand.

Basic Assembly Language Instructions

- **Data Movement**: Moves data between registers or memory.
  - Example: `MOV A, B` (Move the value in register B to register A).
- **Arithmetic Operations**: Performs arithmetic calculations.
  - Example: `ADD A, B` (Add the value in register B to register A).
- **Logic Operations**: Performs logical operations.
  - Example: `AND A, B` (Perform a bitwise AND operation between registers A and B).
- **Branching**: Changes the flow of program execution.
  - Example: `JMP address` (Jump to the specified memory address).
- **Control Instructions**: Used to control the execution flow.
  - Example: `HALT` (Stops program execution).

Example of Assembly Language Program

A simple program to add two numbers:

```
MOV A, 5     ; Move 5 into register A
MOV B, 3     ; Move 3 into register B
ADD A, B     ; Add the contents of register B to A
```

Conclusion

A microprocessor is a small, integrated circuit (IC) that performs the functions of a computer's central processing unit (CPU):

Microprocessors perform arithmetic, logic, and control functions on digital signals. They accept binary data as input, process it according to instructions stored in its memory, and provide results in binary form as output.

Microprocessors are clock-driven, register-based, and operate on numbers and symbols represented in the binary number system.

2.5 Microprocessor System

1. Memory Device Classification and Hierarchy

Memory devices are essential for storing and retrieving data in a microprocessor system. They are classified into different types based on their characteristics.

Memory Classification:

- **Primary Memory (Volatile)**:
  - **RAM (Random Access Memory)**: Temporary storage, used to store data and instructions currently in use.
  - **Cache Memory**: A small, high-speed memory used to store frequently accessed data for faster retrieval.
- **Secondary Memory (Non-Volatile)**:
  - **ROM (Read-Only Memory)**: Permanent storage used for storing firmware and boot-up instructions.
  - **EEPROM (Electrically Erasable Programmable ROM)**: A non-volatile memory that can be erased and rewritten electronically.

Memory Hierarchy:

Memory hierarchy refers to the way different types of memory are arranged in a computer system, organized by speed and size.

1. **Registers**: These are the fastest form of memory, located directly inside the CPU (Central Processing Unit). Registers hold data that the CPU is currently processing. Since they're part of the CPU, they can be accessed almost instantly. However, they are very limited in size, typically storing only a small amount of data (e.g., a few bytes).
2. **Cache Memory**: This is faster than RAM but smaller in size. Cache memory stores frequently accessed data to reduce the CPU's need to fetch it from the slower RAM. Modern CPUs have multiple levels of cache (L1, L2, L3) with varying sizes and speeds, where L1 is the fastest but smallest, and L3 is larger but slower than L1 and L2.
3. **RAM (Random Access Memory)**: RAM is much larger than cache and provides temporary storage for data and programs that are actively used by the CPU. While RAM is significantly faster than secondary storage (like hard drives), it is slower than cache memory. When the CPU needs data not in the cache, it accesses RAM, which is still much faster than pulling from secondary storage.

4. **Secondary Storage**: This refers to non-volatile storage devices like hard drives (HDDs), solid-state drives (SSDs), and optical disks. Secondary storage holds data permanently and is much slower compared to RAM and cache. It's much larger in capacity and is used to store operating systems, applications, and other data that aren't in immediate use.

The memory hierarchy allows a computer system to balance the need for speed with the need for large storage capacities. The faster the memory (like registers or cache), the more expensive and smaller it is. Conversely, slower memory types (like secondary storage) are cheaper and have much larger capacities.

---

2. Interfacing I/O and Memory Interfaces

In computing, **I/O (Input/Output) interfaces** and **memory interfaces** are the pathways through which data is transferred between the **microprocessor** and **external devices** (like keyboards, sensors, and displays), or between the microprocessor and memory (RAM, ROM, etc.).

These interfaces can use two primary methods of communication:

1. **Parallel Communication**

In **parallel communication**, multiple bits are transferred at once, each over its own line, allowing for faster data transfer.

Advantages:

- **Speed**: Since multiple bits are transferred simultaneously, the transfer rate is much higher compared to serial communication.
- **Simplicity in Design (for Short Distances)**: For small systems or short distances, parallel communication can be straightforward to implement.

Disadvantages:

- **Signal Integrity Issues**: At higher speeds, signals across multiple lines can interfere with each other, causing data corruption.
- **More Wiring**: Parallel communication requires many signal lines. For example, a 32-bit parallel connection needs 32 separate lines. This increases the complexity of the circuit and the design.
- **Costly for Long Distances**: As the distance increases, the likelihood of signal degradation grows, which makes parallel communication impractical for long-distance communication.

Use Cases:

- Memory (RAM), printers, displays, and devices requiring fast data transfers.

---

2. **Serial Communication**

In **serial communication**, data is sent bit-by-bit over a single line, making it simpler and cheaper.

Advantages:

- **Fewer Wires**: Only one signal line is needed for data transmission, making it cheaper and simpler to implement, especially for long distances.

- **Reduced Crosstalk**: With only one wire for data transfer, there's less chance of signal interference, especially over long distances.
- **Reliable Over Long Distances**: Serial communication works better over longer distances because it is less susceptible to signal degradation compared to parallel communication.

Disadvantages:

- **Slower Data Transfer**: Only one bit is sent at a time, making serial communication slower than parallel communication.
- **Latency**: The time taken to transfer data can increase as the system scales up, making it unsuitable for high-speed, real-time applications where parallel communication would be better.

Use Cases:

- USB, networking, and devices requiring simpler connections.

When to Use Parallel Communication Over Serial for Memory and I/O?

- **Serial communication** is generally **not ideal for memory and high-performance I/O systems** that require **short-distance, high-speed communication** because **parallel** is better suited for these purposes.
- **For memory (like RAM)**: The speed and bandwidth required for handling large amounts of data quickly are much higher, and **parallel communication** is more suitable due to its ability to transfer multiple bits simultaneously over multiple lines. This is ideal for **short-distance** communication, like between the processor and RAM, where the devices are physically close.
- **For I/O systems** that demand fast data transfer (such as printers or sensors): **Parallel communication** would generally still be more effective over short distances, as it allows multiple bits to be sent at once, achieving faster communication.

---

3. Introduction to PPI, Synchronous / Asynchronous Transmission & DMA Controllers

1. **PPI (Programmable Peripheral Interface):**

A **Programmable Peripheral Interface (PPI)** is a hardware interface used in microprocessor systems to connect various peripheral devices, such as sensors, keyboards, displays, and other I/O devices, to the microprocessor.

- **Functionality**: The PPI provides the flexibility to configure the input/output operations of the microprocessor. It acts as an intermediary between the microprocessor and the connected peripherals. By using a PPI, the system can easily communicate with different types of devices in an efficient manner.
- **Input/Output Modes**: PPI typically supports both input and output modes, meaning it can send data from the processor to peripherals (output) and receive data from peripherals to the processor (input). It enables flexible data transfer and allows the microprocessor to interact with a variety of external devices.

2. **Synchronous vs. Asynchronous Transmission:**

These are two methods for transmitting data between devices, each with its own advantages and trade-offs. They differ mainly in how the data is synchronized between the sender and receiver.

- **Synchronous Transmission:** In synchronous transmission, data is sent as a continuous stream of bits, and both the sender and receiver are synchronized with a clock signal. The clock signal dictates the timing of data transfer.
  - **Characteristics**:
    - **Synchronization**: Both the transmitter and receiver share the same clock signal, ensuring that data is sent and received at the same time intervals.
    - **Speed and Reliability**: Since both sides are synchronized, synchronous transmission is generally faster and more reliable. This eliminates the need for start and stop bits, making it more efficient.
    - **Example**: Data transfer between processors in high-speed communication, such as in a computer network using protocols like Ethernet.
- **Asynchronous Transmission:** In asynchronous transmission, data is sent without synchronization to a clock signal. Instead, the data is transmitted in packets or chunks, each with start and stop markers to define the boundaries of each packet.
  - **Characteristics**:
    - **Flexibility**: Asynchronous transmission allows for the transfer of data without requiring the sender and receiver to operate in sync with a common clock signal, offering more flexibility in communication.
    - **Start and Stop Bits**: Since there is no clock signal to keep the data flow continuous, each packet is marked with a start bit (indicating the beginning of transmission) and stop bits (indicating the end of transmission). This ensures data integrity and allows the receiver to know when a new packet begins and when the previous one ends.
    - **Use Cases**: Typically used in slower-speed communications, such as serial communication, where the data transfer rate is not as high and the transmission is intermittent (e.g., communication with a keyboard or mouse).

---

3. **DMA Controllers (Direct Memory Access):**

DMA (Direct Memory Access) is a method that allows peripherals (such as hard drives, audio devices, network cards) to access the system's memory directly, bypassing the CPU. This improves performance by allowing data transfers to happen without involving the CPU in every transfer.

- **Functionality**: DMA provides a mechanism where peripherals can transfer data directly to/from memory, without CPU intervention. This means that while data is being transferred between a peripheral and memory, the CPU is free to perform other tasks.
- **Benefits**:
  - **Faster Data Transfer**: By enabling peripherals to access memory directly, DMA significantly increases the speed of data transfers. It is particularly beneficial for high-speed devices like hard drives or network interfaces where large volumes of data need to be transferred quickly.
  - **Reduced CPU Workload**: Without DMA, the CPU would have to manage every byte of data transfer between peripherals and memory, which would consume a significant amount of CPU time and resources. With DMA, the CPU can delegate this responsibility to the DMA controller, freeing it up to handle other tasks.
  - **Efficiency**: DMA increases the efficiency of the system, as it reduces the overhead

required for data transfer operations. The CPU is not burdened with managing memory transfers, allowing it to perform other critical tasks concurrently, improving the overall system's performance.

Conclusion

Memory devices are categorized into primary (volatile) and secondary (non-volatile) storage, forming a hierarchy from fast registers to slower secondary storage. I/O and memory interfaces enable communication between the microprocessor and external devices, with parallel interfaces offering faster data transfer. PPI, serial interfaces, and DMA controllers enhance system efficiency by optimizing data transmission and peripheral connectivity.

2.6 Interrupt Operations

1. Interrupts

Interrupts are signals that alert the microprocessor to stop its current operations and attend to more urgent tasks which is typically triggered by hardware devices or software.

There are basically two types of interrupts:

- **Hardware Interrupts**: Generated by external devices, such as input from a keyboard, mouse, or sensor.
- **Software Interrupts**: Triggered by software instructions (e.g., a system call or software exception).

2. Interrupt Service Routines (ISRs)

Interrupt Service Routine (ISR) is a special function or set of instructions executed when a specific interrupt occurs. Each interrupt is assigned an ISR, which is responsible for handling the interrupt and then returning control back to the main program.

The different steps involved in the ISRs:

1. **Interrupt Request (IRQ)**: An interrupt is raised by the hardware or software.
2. **Interrupt Acknowledgment**: The microprocessor acknowledges the interrupt.
3. **Execution of ISR**: The ISR is executed to handle the interrupt.
4. **Return to Normal Operation**: Once the interrupt is processed, the CPU resumes the previously interrupted task.

3. Interrupt Processing

The **interrupt processing cycle** involves the following steps:

1. **Interrupt Occurrence**: An interrupt is triggered.
2. **Interrupt Detection**: The processor detects the interrupt signal.
3. **Interrupt Acknowledgment**: The processor acknowledges the interrupt and saves the current execution state (e.g., registers, program counter) to preserve the ongoing task.
4. **ISR Execution**: The processor jumps to the corresponding ISR address to execute the interrupt-handling code.

5. **Context Restoration**: After the ISR is completed, the processor restores the saved execution state and resumes normal program execution.

4. Types of Interrupts:

There are usually two types of interrupts:

- **Maskable Interrupts**: Can be disabled (masked) by the CPU to avoid interruption during critical processes.
- **Non-Maskable Interrupts (NMI)**: Cannot be disabled and have higher priority, typically used for critical events like hardware failure.

5. Interrupt Priority

When multiple interrupts occur simultaneously, the processor must determine which one to address first. This is often done through an **interrupt priority scheme**.

There are usually two types of interrupts priorities:

- **Fixed Priority**: Each interrupt source has a fixed priority level, and the processor serves the highest-priority interrupt.
- **Dynamic Priority**: The priority may be changed depending on the circumstances or the type of interrupt.

6. Interrupt Vector Table

The **interrupt vector table** is a table in memory that stores the addresses of ISRs for various interrupt sources. When an interrupt occurs, the processor looks up the ISR address in the interrupt vector table to know where to jump for processing.

Conclusion

Interrupts are signals that temporarily pause the CPU's tasks to handle urgent events, triggered by hardware (e.g., keyboard) or software (e.g., exceptions). An ISR manages each interrupt, ensuring tasks resume after processing. Interrupts can be maskable (disable-able) or non-maskable (critical, high-priority). The CPU uses an interrupt vector table to locate ISR addresses.