Answers for CORE PAPER - II

Section A: Answer any 10 questions (10 x 2 = 20 Marks)

1. Convert Decimal 41 into binary number.

Decimal 41 in binary is **101001**.
Explanation: 41 ÷ 2 = 20 remainder 1, 20 ÷ 2 = 10 remainder 0, 10 ÷ 2 = 5 remainder 0, 5 ÷ 2 = 2 remainder 1, 2 ÷ 2 = 1 remainder 0, 1 ÷ 2 = 0 remainder 1 ⇒ (from bottom to top) = 101001.

---

2. Define Truth Table.

A **Truth Table** is a table that shows all possible input combinations to a logic circuit or Boolean function and the corresponding output.
It is used to represent the function of logic gates.

---

3. Write the purpose of De-multiplexer.

A **De-multiplexer (DEMUX)** is a device that takes a single input and routes it to one of several outputs based on selector lines.
It is used in data routing, communication systems, and digital signal distribution.

---

4. What is encoder? Give its uses.

An **Encoder** is a combinational circuit that converts 2^n input lines into an n-bit binary code.
Uses: Priority encoders, keyboard encoding, data compression.

---

5. Define Microcomputer.

A **Microcomputer** is a small, relatively inexpensive computer with a microprocessor as its CPU.

It includes memory, input/output ports, and is used in personal computers, embedded systems, etc.

---

6. What are the addressing modes? List the types.

Addressing modes define how the operand of an instruction is chosen.
Types: Immediate, Direct, Indirect, Register, Register Indirect, Indexed, Base Register.

---

7. What is Program Counter?

The **Program Counter (PC)** is a register in the CPU that holds the address of the next instruction to be executed.
It automatically increments after each instruction fetch.

---

8. Define Subroutine.
Answer:
A **Subroutine** is a set of instructions designed to perform a specific task, which can be called from multiple places in a program.
It helps in modular programming and code reuse.

---

9. What is an Interrupt? Mention its sources.

An **Interrupt** is a signal to the processor indicating an event that needs immediate attention.
Sources: Hardware (keyboard, timer), Software (system calls, errors).

---

10. What is memory mapped I/O?

In **Memory Mapped I/O**, I/O devices are assigned specific memory addresses. The CPU uses standard memory instructions to access them.
It simplifies hardware interface design.

---

11. What are the different types of ROMs?

Types of **ROM (Read-Only Memory)** include:

- **PROM** (Programmable ROM)
- **EPROM** (Erasable PROM)
- **EEPROM** (Electrically Erasable PROM)
- **Mask ROM** (factory programmed)

---

12. What is a bus? Define bus width.

A **Bus** is a communication pathway used to transfer data between components in a computer.
Bus width is the number of bits that can be transferred simultaneously, affecting data transfer rate.

Section B: Answer any 5 questions (5 x 6 = 30 Marks)

13. (i) Define Binary Logic.

Binary logic is the basis of all digital systems and computer operations. It is a logical system that operates using two distinct states or values, which are typically represented as **0** and **1**. These two values are known as **binary digits**, or **bits**.

Binary logic uses the principles of Boolean algebra, where variables can only have one of two values:

- **0** (False or OFF)
- **1** (True or ON)

This logic is used to implement all digital electronic circuits, especially in digital computers. Basic logical operations such as **AND, OR, NOT, NAND, NOR, XOR,** and **XNOR** are implemented using binary logic. These operations are carried out using logic gates in hardware.

For example:

- **AND gate** gives output 1 only if both inputs are 1.
- **OR gate** gives output 1 if any one of the inputs is 1.
- **NOT gate** inverts the input (1 becomes 0 and 0 becomes 1).

Importance in computers:
Binary logic is used in the design of arithmetic units, memory systems, control units, and more. It is the foundation for programming, data processing, and digital communication.

13. (ii) How negative numbers are stored in a digital computer?

Digital computers store numbers using binary representation. For negative numbers, a special method is required since binary digits (0 and 1) naturally represent only non-negative numbers. There are three common ways to store negative numbers in binary:

1. Sign-Magnitude Representation:

- The **most significant bit (MSB)** represents the sign (0 = positive, 1 = negative).
- The remaining bits represent the magnitude of the number.
- Example (using 8 bits):
    - +5 = 00000101
    - -5 = 10000101

Drawback: There are two representations for zero (positive and negative zero), and arithmetic operations are more complex.

2. One's Complement Representation:

- Positive numbers are stored as-is.
- Negative numbers are stored by inverting all the bits of the positive number.
- Example (8-bit):
    - +5 = 00000101
    - -5 = 11111010

Drawback: Like sign-magnitude, it has two representations of zero and complicates arithmetic.

3. Two's Complement Representation (Most Common):

- Positive numbers remain unchanged.
- Negative numbers are obtained by taking the one's complement and then adding 1.
- Example:
    - +5 = 00000101
    - -5:
        - One's complement: 11111010
        - Add 1: 11111011

Advantages:

- Only one representation of zero.
- Easier hardware implementation for addition and subtraction.
- Widely used in modern computer systems.

Conclusion:
Among the above methods, **two's complement** is the most efficient and widely used method for storing negative numbers in digital computers due to its simplicity in arithmetic operations and unique representation of zero.

---

14. Discuss the following in detail:

(a) Shift Registers

A **Shift Register** is a sequential digital circuit used to store and transfer binary data. It is made up of a series of flip-flops connected in such a way that the output of one flip-flop becomes the input of the next. Shift registers operate under the control of clock pulses, and with each pulse, the data is shifted one position either to the left or to the right. The direction of shifting depends on the design and application. The main types of shift registers include:

- **Serial-In Serial-Out (SISO):** Data enters and leaves the register in a serial manner.
- **Serial-In Parallel-Out (SIPO):** Data enters serially and is read out in parallel.
- **Parallel-In Serial-Out (PISO):** Data enters all at once and exits one bit at a time.
- **Parallel-In Parallel-Out (PIPO):** Data enters and exits simultaneously in parallel.

There are also **Bidirectional** and **Universal Shift Registers**, which provide additional flexibility such as shifting data in both directions or combining serial and parallel operations. Shift registers are widely used in applications like data conversion (serial to parallel or vice versa), digital memory, delay circuits, and data manipulation in microprocessors.

(b) Multiplexers

A **Multiplexer (MUX)** is a combinational circuit that allows multiple input signals and selects one of them to be the output based on the value of selection inputs. It functions like a digitally controlled switch. A multiplexer with 2^n input lines requires n selection lines to choose which input is connected to the output. For example, a 4-to-1 MUX has four inputs (I_0 to I_3), two select lines (S_0, S_1), and one output.

The output Y of a 4-to-1 MUX is given by:
$\mathbf{Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3}$

Multiplexers are used in various digital systems for data routing, signal selection, resource sharing, and implementing logic functions. They are particularly useful in communication systems, microprocessors, and computer memory systems where efficient control over data flow is required.

---

15. Differentiate Between Combinational and Sequential Circuits

https://www.geeksforgeeks.org/difference-between-combinational-and-sequential-circuit/

---

16. Write a Short Note on Accumulator Register?

An accumulator register is **a special-purpose register in a CPU used to store the intermediate results of arithmetic and logical operations**. It's essentially a temporary storage location for calculations performed by the Arithmetic Logic Unit (ALU).

Here's a more detailed explanation:

Purpose:

- **Storing Intermediate Results:** The accumulator holds the output of arithmetic or logical operations, allowing the CPU to perform subsequent calculations using the result without needing to fetch it from main memory.
- **Efficiency:** This design can improve the speed of calculations, as the accumulator is a faster and more direct storage location than main memory.
- **Implicit Operand:** In some CPU architectures (like early ones), the accumulator is the implicit destination for the results of many instructions, simplifying the instruction set.

How it Works:

- When an arithmetic or logical operation is performed, the result is typically stored in the accumulator register.
- The CPU can then use the value stored in the accumulator as an operand for subsequent operations.
- The accumulator can also be used as a source operand for an operation, bringing the value into the ALU.

Examples:

- In the 8085 and 8051 microprocessors, the accumulator register (A) is a common example.
- In early CPU designs, the accumulator was a primary component for storing and manipulating data.

Advantages:

- **Faster Access:** The accumulator is typically faster to access than main memory, leading to potentially faster calculations.
- **Simplified Instructions:** In accumulator-based CPU designs, the instruction set can be simpler because the accumulator is the implicit destination for results.

Disadvantages:

- **Limited Storage:** The accumulator is usually a fixed-size register, limiting the amount of data it can hold.
- **Single-Purpose:** In some CPU architectures, the accumulator can be dedicated to a specific purpose, limiting its versatility.

---

17. **Explain the Working of a T Flip-Flop**

https://www.geeksforgeeks.org/t-flip-flop/

---

18. Simplify the Boolean Function Using K-Map:

https://www.electrical4u.com/simplifying-boolean-expression-using-k-map/

19.Explain Full Adder in Detail

https://www.geeksforgeeks.org/full-adder-in-digital-logic/

Section C: Answer any 3 questions (3 x 10 = 30 Marks)

20. Don't Care Conditions and Don't Care's in a BCD System

Introduction

In digital logic design, **“don’t care” conditions** refer to input combinations for which the output of a circuit is **not specified** or **not relevant**. These conditions can be treated as either logic **0 or 1**, depending on which choice helps **simplify the Boolean expression** or circuit design. They are denoted by an **“X”** in truth tables or Karnaugh maps.

“Don’t cares” are particularly useful when designing logic circuits using simplification methods like **Karnaugh maps (K-maps)**. They provide flexibility because you can choose their values (0 or 1) strategically to achieve minimal logic expressions.

Why Don’t Care Conditions Occur?

Don’t care conditions occur in several scenarios:

- When certain input combinations **will never happen** in actual use.
- When the output **does not matter** for some inputs.
- In systems that work with **limited valid inputs** out of all possible binary combinations.

Don't Care Conditions in a BCD System

BCD stands for **Binary Coded Decimal**, where each decimal digit (0–9) is represented using **4 binary bits**. However, a 4-bit binary number can represent values from **0000 (0) to 1111 (15)**. In BCD, only the values **0000 to 1001 (0 to 9)** are valid. The remaining combinations from **1010 (10) to 1111 (15)** are **invalid** and **do not occur** in BCD.

These invalid combinations (**1010 to 1111**) are considered **“don’t care” conditions** in logic design using BCD. Designers use them to simplify logic expressions in circuits like **BCD to 7-segment decoders**, BCD adders, etc.

Example: Don’t Cares in BCD

Valid BCD inputs:

- 0000 (0)
- 0001 (1)
- ...
- 1001 (9)

Invalid BCD inputs (Don’t cares):

- 1010 (10)
- 1011 (11)
- 1100 (12)
- 1101 (13)
- 1110 (14)
- 1111 (15)

When plotting these values on a **K-map**, the invalid combinations (10 to 15) are marked as **X (don’t care)**, allowing us to combine adjacent groups and reduce the logic expression.

Applications of Don't Care Conditions

1. **K-map Optimization:** Enables easier grouping to minimize Boolean functions.
2. **Hardware Simplification:** Reduces gate usage in digital circuits.

3. **Logic Design in Counters & ALUs:** Makes designing memory units and arithmetic circuits more efficient.

Thus, Don't Care conditions play a vital role in simplifying circuit designs while maintaining efficiency.

Conclusion

Don’t care conditions are a powerful concept in logic design, providing opportunities for simplification and optimization. In **BCD systems**, don’t cares naturally arise due to the exclusion of binary combinations from 1010 to 1111. Recognizing and using these conditions effectively helps in designing more efficient and compact digital circuits.

---

21. NAND as a Universal Gate

Introduction

In digital electronics, a **universal gate** is a logic gate that can be used to implement **any Boolean function** without needing to use any other type of gate. The **NAND gate** and **NOR gate** are called universal gates because you can construct all the other basic logic gates (NOT, AND, OR) using just NAND or just NOR gates.

Among them, **NAND** is more commonly used in hardware design due to its **cost-effectiveness**, **ease of fabrication**, and **functional versatility**.

NAND Gate Basics

A **NAND gate** is a combination of an AND gate followed by a NOT gate. It gives an output that is the **complement of the AND operation**. Its Boolean expression is:

$$Y = \overline{A \cdot B}$$

A	B	Y (NAND)
0	0	1
0	1	1
1	0	1
1	1	0

NAND Implementations of Basic Gates

You can implement **NOT, AND, and OR** gates using only NAND gates:

1. NOT Gate Using NAND

By connecting both inputs of a NAND gate to the same variable:

$$Y = \overline{A \cdot A} = \overline{A}$$

So, a single NAND gate can act as a NOT gate.

2. AND Gate Using NAND

To get an AND function from NAND gates:

- First, perform NAND: A⋅B‾\overline{A \cdot B}A⋅B
- Then, invert the output using a NAND as NOT:

$$Y = \overline{\overline{A \cdot B}} = A \cdot B$$

This requires **two NAND gates**.

3. OR Gate Using NAND

To implement an OR gate using NAND gates, use De Morgan’s Theorem:

$$A + B = \overline{(\overline{A} \cdot \overline{B})}$$

Steps:

- Use two NAND gates to generate $\overline{A}$ and $\overline{B}$
- Use a third NAND to perform $\overline{\overline{A} \cdot \overline{B}}$

This gives the OR function using **three NAND gates**.

Importance in Digital Design

The NAND gate is called universal because **any digital circuit**—no matter how complex—can be built using only NAND gates. This simplifies manufacturing and reduces cost, as only one type of gate needs to be produced or fabricated in integrated circuits.

Conclusion

The **NAND gate** is a fundamental and powerful component in digital logic design. Its ability to implement all other gates makes it a **universal gate**. Understanding how to use NAND gates to build other logic gates is essential for efficient digital circuit design and optimization.

---

22. Flip-Flop and Its Types

A **flip-flop** is a basic **sequential logic circuit** used to **store one bit of data**. It has two stable states, making it a **bistable multivibrator**. Flip-flops are the fundamental building blocks of digital electronics used in **memory units**, **registers**, **counters**, and **control circuits**. They are edge-triggered, meaning they change states only during a transition of a clock signal (either rising or falling edge).

Each flip-flop has one or more inputs and two outputs — **Q** and **Q̄ (complement of Q)**. The state of a flip-flop changes based on its inputs and clock pulses, and it can **retain its output** even after the inputs are removed, making it ideal for data storage.

Types of Flip-Flops

There are several types of flip-flops, each with different input combinations and behavior. The main types include:

1. SR Flip-Flop (Set-Reset Flip-Flop)

- **Inputs**: S (Set), R (Reset)
- **Function**: Sets or resets the output based on input.
- If S = 1 and R = 0 → Q = 1 (Set state)
- If S = 0 and R = 1 → Q = 0 (Reset state)
- If S = 0 and R = 0 → No change (Hold state)
- If S = 1 and R = 1 → Invalid condition (undefined)

Use: Simple memory storage but not used widely due to invalid state issue.

2. D Flip-Flop (Data or Delay Flip-Flop)

- **Inputs**: D (Data)
- **Function**: Captures the value on the D input at the moment of the clock edge and stores it.

- If D = 1 → Q = 1
- If D = 0 → Q = 0

This flip-flop removes the invalid state present in SR flip-flop by combining S and R into a single input.

Use: Widely used in data storage, registers, and shift registers.

3. JK Flip-Flop

- **Inputs**: J, K
- Improved version of SR flip-flop without the invalid state.
- If J = 0, K = 0 → No change
- If J = 0, K = 1 → Reset
- If J = 1, K = 0 → Set
- If J = 1, K = 1 → Toggle (Q becomes $\overline{Q}$)

Use: Commonly used in counters due to toggle function.

4. T Flip-Flop (Toggle Flip-Flop)

- **Input**: T (Toggle)
- If T = 0 → No change
- If T = 1 → Toggle the output (Q becomes $\overline{Q}$)

It is derived from the JK flip-flop by connecting both J and K inputs together.

Use: Used in frequency dividers and binary counters.

Conclusion

Flip-flops are essential components in digital circuits used to store and control data. Depending on the logic required — such as hold, toggle, set/reset, or data storage —

different types of flip-flops (SR, D, JK, T) are used. Their applications span across digital memory, timing circuits, and sequential logic systems.

---

23. Time Delay Using Register Pairs with Example

Introduction to Time Delay in Microprocessors

In microprocessor programming (especially in 8085), **time delay routines** are often required to control the speed of operations, synchronize hardware components, or introduce a pause between operations. Since early microprocessors do not have built-in timers, **software delays** are created using **registers and loops**.

Using Register Pairs for Time Delay

In 8085 microprocessor, delay can be generated by **decreasing the contents of registers** repeatedly in a loop. The **register pair** (like **D-E** or **B-C**) is used to store a **16-bit count**. The program enters a loop where the register pair is decremented continuously. Once the register pair becomes zero, the loop ends, thereby creating a time delay.

The time delay generated depends on:

- The number of machine cycles the instructions take.
- The frequency of the system clock.
- The value loaded into the register pair.

Delay Loop Concept

Let’s say we use **D-E register pair** for generating a delay. The logic is:

1. Load a 16-bit value into D-E.

2. Decrement the D-E pair until it reaches 0000H.
3. Exit the loop.

This introduces a delay, as the processor spends time in decrementing and checking the loop.

Example Program: Delay Using D-E Pair

assembly

CopyEdit

```
MVI D, FFH       ; Load D with FFH (high byte)

MVI E, FFH       ; Load E with FFH (low byte)

DELAY: DCX D     ; Decrement register pair DE

       MOV A, D

       ORA E     ; Check if D and E are zero

       JNZ DELAY ; If not zero, repeat loop

       HLT       ; Halt the program
```

Explanation of the Program

- `MVI D, FFH` and `MVI E, FFH` together initialize the DE pair with **FFFFH**.
- `DCX D` (Decrement D-E): reduces the count by 1.
- `MOV A, D` and `ORA E`: combines D and E using OR to check if both are zero.

- `JNZ DELAY`: If the result is not zero, it jumps back to `DELAY`.
- `HLT`: Ends the program after delay loop completes.

This loop runs **65,535 times** (FFFFH in decimal), and since each iteration takes multiple machine cycles, it creates a significant time delay.

Conclusion

Using register pairs like D-E or B-C in 8085 microprocessor is a simple and effective method to generate time delays. It is particularly useful in applications like LED blinking, sensor polling, or any timing-sensitive operation in embedded systems. The delay duration can be controlled by adjusting the initial values in the register pair.

---

24. Explain Bus Arbitration

Bus arbitration is a process used in computer systems where **multiple devices or processors share a common communication bus**, and only **one device can use the bus at a time**. It determines **which device gets control of the bus** when more than one requests it simultaneously. Bus arbitration ensures **orderly and conflict-free communication** between devices in a system.

Need for Bus Arbitration

In multiprocessor systems or systems with multiple peripherals (like DMA controllers, CPUs, I/O devices), many components may want to access memory or other shared resources at the same time. If more than one device tries to access the bus simultaneously, it can cause **data corruption or system malfunction**. Therefore, **bus arbitration is necessary to manage access** in a fair and efficient manner.

How Bus Arbitration Works

The process of bus arbitration involves:

1. **Requesting the Bus**: Devices that need access to the bus send a request signal to the arbiter (the controller that manages arbitration).

2. **Arbitration Decision**: The arbiter decides which device gets the bus based on a specific protocol or priority scheme.

3. **Granting the Bus**: The arbiter sends a grant signal to the selected device.

4. **Using the Bus**: The selected device uses the bus to complete its data transfer.

5. **Releasing the Bus**: After the operation, the device releases the bus so that others can use it.

Types of Bus Arbitration

1. **Centralized Arbitration**:
   A single bus arbiter (usually part of the CPU or a controller) controls access to the bus. All devices send their requests to this central controller.

   - Example: Daisy chaining.

2. **Distributed Arbitration**:
   No central controller is used. Each device participates in the arbitration process and decides collectively who gets the bus.

3. **Priority-based Arbitration**:
   Devices are assigned priorities. The arbiter always grants access to the

highest-priority device.

Example: Daisy Chain Arbitration

In this method, devices are connected in series. A request passes through each device in the chain, and the device closest to the controller with a request gets access first. It's **simple** but **not fair**, as lower-priority devices (farther down the chain) may have to wait longer.

Conclusion

Bus arbitration is a **crucial mechanism** in digital systems that share a common bus among multiple devices. It ensures that only **one device communicates at a time**, thereby preventing conflicts and ensuring **efficient and fair resource sharing**. Whether using centralized or distributed methods, effective arbitration enhances system performance and stability.