USN | | | | | | | | | | |

Internal Assessment Test 1 – November 2024

Sub:	Digital Design and Computer Organization					Sub Code:	BCS302	Branch:	AIML	
Date:	06/11/24	Duration:	90 minutes	Max Marks:	50	Sem/Sec:	III			OBE

Answer any FIVE FULL Questions

			MARKS	CO	RBT
1	a	Identify the prime implicants and essential prime implicants of the following functions i. F(W,X,Y,Z)= ∑(0,1,2,5,7,8,10,15)	[10]	1	L3
2	a	Design and explain four-bit adder with carry look ahead A 4-bit adder with carry look-ahead is a digital circuit designed to add two 4-bit binary numbers along with a carry input. The carry look-ahead technique improves the speed of the addition process by reducing the time it takes to propagate carry bits across multiple stages in the adder. Components of a 4-Bit Adder with Carry Look-Ahead 1. Full Adders (FA): A 4-bit adder consists of 4 full adder circuits. Each full adder computes the sum of a pair of bits from the input numbers, along with the carry input from the previous stage. 2. Carry Look-Ahead Logic: This part of the circuit calculates the carries for all stages in parallel, reducing the delay associated with carry propagation. Instead of waiting for carries to propagate one stage at a time, the carry look-ahead logic computes the carry outputs quickly.	[10]	2	L2

3. Sum Generation: The sum is generated by XORing the input bits and the carry bits from the previous stage.

Full Adder

Each full adder computes the sum and carry for a pair of bits. The inputs to a full adder are:
- Two bits to be added: A_i and B_i (where $i = 0, 1, 2, 3$ for the 4-bit adder)
- A carry input C_{in} (for the first adder, $C_{in} = 0$)

The outputs from a full adder are:
- Sum (S_i) = $A_i \oplus B_i \oplus C_{in}$
- Carry (C_{out}) = $(A_i \cdot B_i) + (C_{in} \cdot (A_i \oplus B_i))$

Carry Look-Ahead Logic

The main objective of the carry look-ahead logic is to calculate the carry for each bit position without waiting for the carries to propagate through all the stages sequentially. It uses the following two signals for each bit:

- Generate (G_i): The generate signal indicates whether a carry is generated at bit position i. It is calculated as:
$$G_i = A_i \cdot B_i$$
This means a carry is generated if both bits A_i and B_i are 1.

- Propagate (P_i): The propagate signal indicates whether a carry from the previous stage will propagate through the current bit position. It is calculated as:
$$P_i = A_i \oplus B_i$$
This means a carry from the previous stage will propagate if one of A_i or B_i is 1, but not both.

The carry look-ahead logic computes the carry outputs for each bit position in parallel using the generate and propagate signals:
- Carry for bit 1 (C_1) is generated as:
$$C_1 = G_0 + (P_0 \cdot C_0)$$
- Carry for bit 2 (C_2) is generated as:
$$C_2 = G_1 + (P_1 \cdot C_1)$$
- Carry for bit 3 (C_3) is generated as:
$$C_3 = G_2 + (P_2 \cdot C_2)$$
- Carry for bit 4 (C_4) is generated as:
$$C_4 = G_3 + (P_3 \cdot C_3)$$

Where C_0 is the initial carry-in (typically 0).

4-Bit Adder with Carry Look-Ahead: Block Diagram

1. Inputs:
   - A_3, A_2, A_1, A_0 (4 bits of the first number)
   - B_3, B_2, B_1, B_0 (4 bits of the second number)
   - C_0 (carry-in)

2. Full Adders:
   - Each full adder computes the sum for each bit and generates a carry to the next bit.

3. Carry Look-Ahead Unit:
   - Computes the carry outputs C_1, C_2, C_3 using the generate and propagate signals.
   - The look-ahead logic reduces the carry propagation delay by calculating the carries in parallel.

4. Outputs:
   - Sum bits: S_3, S_2, S_1, S_0
   - Final carry-out C_4

Detailed Explanation:

1. Sum Calculation:
   For each bit i, the sum is calculated as:
   $$S_i = A_i \oplus B_i \oplus C_i$$
   Where C_i is the carry for that bit (computed using carry look-ahead).

2. Carry Calculation:
   The carry for each bit is determined using the carry look-ahead unit. The carry for each bit position i is:
   $$C_i = G_{i-1} + P_{i-1} \cdot C_{i-1}$$
   And G_i and P_i are generated as described earlier.

Advantages of Carry Look-Ahead Adder:

- Speed: The carry look-ahead technique significantly reduces the carry propagation delay, which is the main bottleneck in ripple carry adders.
- Parallelism: Carry look-ahead allows the carry for all bits to be computed in parallel, as opposed to the ripple carry adder where each carry depends on the previous one.

Example:

Consider adding two 4-bit numbers $A = 1101_2$ (13 in decimal) and $B = 1011_2$ (11 in decimal), with carry-in $C_0 = 0$.

Full Adder Calculations:

A	B	Carry-in C_0	Sum	Carry-out
1	1	0	0	1
1	0	1	0	1

		| 0 | 1 | 1 | 0 | 1 | | 1 | 1 | 1 | 1 | 1 | Carry Look-Ahead: - \(G_0 = A_0 \cdot B_0 = 1 \cdot 1 = 1 \) - \(P_0 = A_0 \oplus B_0 = 1 \oplus 1 = 0 \) So, for each carry: - \(C_1 = G_0 + P_0 \cdot C_0 = 1 + 0 \cdot 0 = 1 \) - \(C_2 = G_1 + P_1 \cdot C_1 = 0 + 1 \cdot 1 = 1 \) - \(C_3 = G_2 + P_2 \cdot C_2 = 1 + 1 \cdot 1 = 1 \) Finally, the sum is: - \(S_3 = A_3 \oplus B_3 \oplus C_3 = 1 \oplus 1 \oplus 1 = 1 \) - \(S_2 = A_2 \oplus B_2 \oplus C_2 = 0 \oplus 1 \oplus 1 = 0 \) - \(S_1 = A_1 \oplus B_1 \oplus C_1 = 1 \oplus 0 \oplus 1 = 0 \) - \(S_0 = A_0 \oplus B_0 \oplus C_0 = 1 \oplus 1 \oplus 0 = 0 \) So the result is \(S_3S_2S_1S_0 = 10000_2 \) (24 in decimal), with a final carry-out of 1. .			
3	a	Design a verilog program to implement types of De-Multiplexer. A De-Multiplexer (DeMUX) is a combinational circuit that takes a single input and routes it to one of many outputs based on a set of selection signals. Essentially, it "demuxes" a single data line into multiple output lines, allowing for one-to-many communication. The general structure of a De-MUX includes an input, several outputs, and a set of selection lines. In Verilog, a De-MUX can be implemented for different configurations based on the number of outputs and the number of selection lines. For example, a 1-to-2, 1-to-4, and 1-to-8 De-MUX are common types. Let's break down the Verilog code to implement the following types of De-Multiplexers: 1. 1-to-2 De-MUX: It routes the input to one of two outputs based on 1 selection line. 2. 1-to-4 De-MUX: It routes the input to one of four outputs based on 2 selection lines. 3. 1-to-8 De-MUX: It routes the input to one of eight outputs based on 3 selection lines. 1. 1-to-2 De-Multiplexer in Verilog ```verilog module demux_1to2(input wire data_in, // Input data input wire select, // Selection line output wire data_out0, // Output 0 output wire data_out1 // Output 1); // Routing data based on selection assign data_out0 = (select == 0) ? data_in : 0; assign data_out1 = (select == 1) ? data_in : 0;	[10]	2	L3

endmodule
```

Explanation:
- Input: `data_in` is the input signal that needs to be routed to one of the two outputs.
- Selection Line: `select` is a 1-bit control line that determines which output receives the `data_in` signal.
- Outputs: `data_out0` and `data_out1` are the two possible outputs.
- The assign statements route `data_in` to `data_out0` when `select = 0` and to `data_out1` when `select = 1`. If the selection line is not active, the output is set to 0.

2. 1-to-4 De-Multiplexer in Verilog

```verilog
module demux_1to4(
    input wire data_in,  // Input data
    input wire [1:0] select, // 2-bit selection signal
    output wire data_out0, // Output 0
    output wire data_out1, // Output 1
    output wire data_out2, // Output 2
    output wire data_out3  // Output 3
);

    // Routing data based on the selection signal
    assign data_out0 = (select == 2'b00) ? data_in : 0;
    assign data_out1 = (select == 2'b01) ? data_in : 0;
    assign data_out2 = (select == 2'b10) ? data_in : 0;
    assign data_out3 = (select == 2'b11) ? data_in : 0;

endmodule
```

Explanation:
- Input: `data_in` is the input signal.
- Selection Line: `select` is a 2-bit signal, which gives us 4 possible combinations (00, 01, 10, 11) to choose one of the four outputs.
- Outputs: `data_out0`, `data_out1`, `data_out2`, and `data_out3` are the four possible outputs.
- The `assign` statements check the selection signal and route `data_in` to the appropriate output.

3. 1-to-8 De-Multiplexer in Verilog

```verilog
module demux_1to8(
    input wire data_in,  // Input data
    input wire [2:0] select, // 3-bit selection signal
    output wire data_out0, // Output 0
    output wire data_out1, // Output 1
    output wire data_out2, // Output 2
    output wire data_out3, // Output 3
    output wire data_out4, // Output 4
    output wire data_out5, // Output 5
    output wire data_out6, // Output 6
    output wire data_out7  // Output 7
);
```
```

```
    // Routing data based on the selection signal
    assign data_out0 = (select == 3'b000) ? data_in : 0;
    assign data_out1 = (select == 3'b001) ? data_in : 0;
    assign data_out2 = (select == 3'b010) ? data_in : 0;
    assign data_out3 = (select == 3'b011) ? data_in : 0;
    assign data_out4 = (select == 3'b100) ? data_in : 0;
    assign data_out5 = (select == 3'b101) ? data_in : 0;
    assign data_out6 = (select == 3'b110) ? data_in : 0;
    assign data_out7 = (select == 3'b111) ? data_in : 0;

endmodule
```

Explanation:
- Input: `data_in` is the input signal.
- Selection Line: `select` is a 3-bit signal, which gives us 8 possible combinations (000 to 111) to choose one of the eight outputs.
- Outputs: `data_out0` to `data_out7` are the eight possible outputs.
- The `assign` statements check the selection signal and route `data_in` to the appropriate output.

Simulation Testbench for 1-to-2 De-MUX:

To verify the functionality of these De-MUX modules, we can write a simple testbench. Here's an example testbench for the 1-to-2 De-MUX:

```verilog
module tb_demux_1to2();

    reg data_in;      // Input data
    reg select;       // Selection line
    wire data_out0;   // Output 0
    wire data_out1;   // Output 1

    // Instantiate the 1-to-2 De-MUX
    demux_1to2 uut (
        .data_in(data_in),
        .select(select),
        .data_out0(data_out0),
        .data_out1(data_out1)
    );

    // Test procedure
    initial begin
        // Test case 1: data_in = 1, select = 0
        data_in = 1;
        select = 0;
        #10;

        // Test case 2: data_in = 1, select = 1
        data_in = 1;
        select = 1;
        #10;

        // Test case 3: data_in = 0, select = 0
```

```
        data_in = 0;
        select = 0;
        #10;

        // Test case 4: data_in = 0, select = 1
        data_in = 0;
        select = 1;
        #10;

        // End the simulation
        $finish;
    end

    // Monitor the outputs
    initial begin
        $monitor("Time = %0t | data_in = %b | select = %b | data_out0 = %b | data_out1
= %b",
                 $time, data_in, select, data_out0, data_out1);
    end

endmodule
```

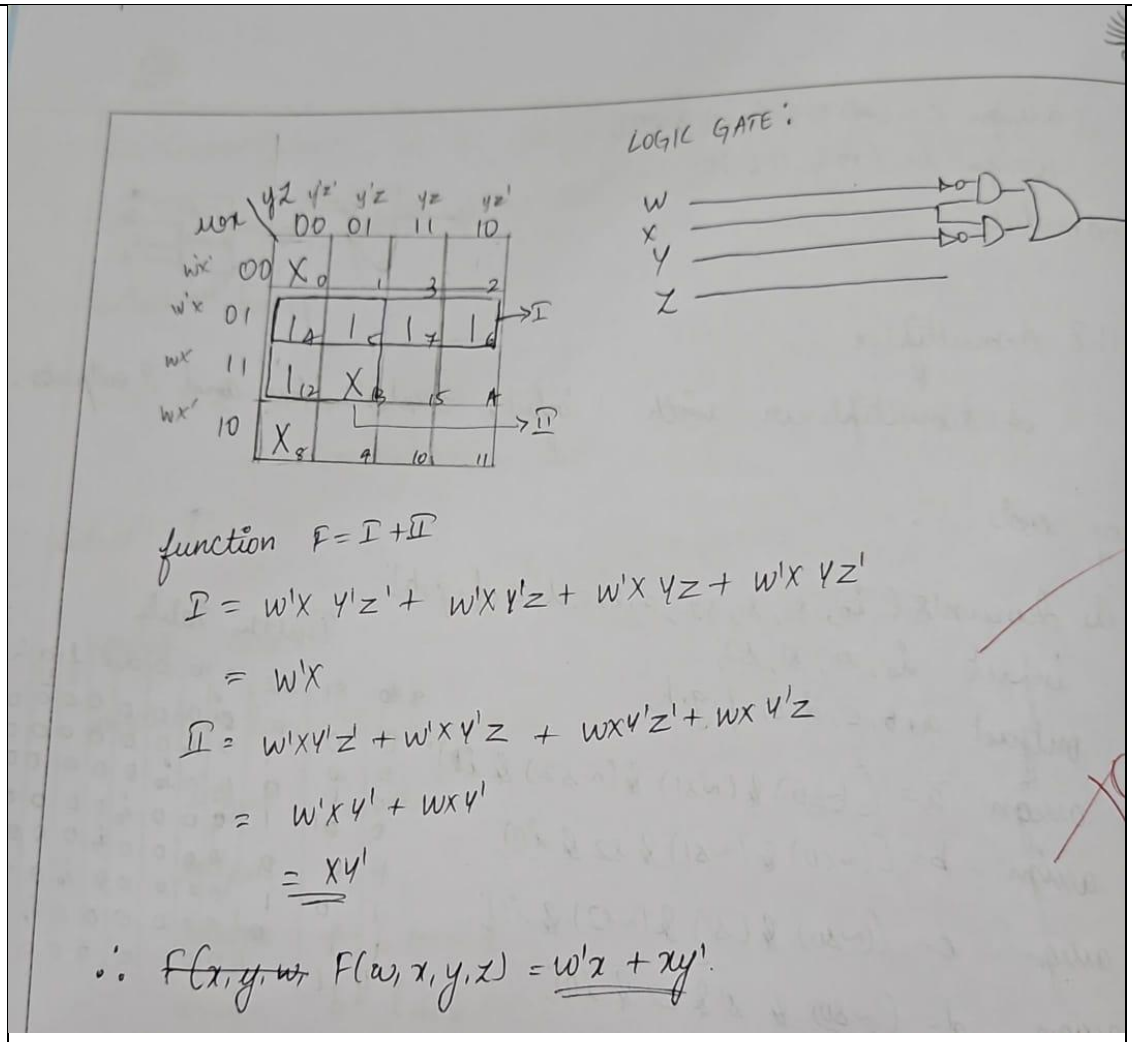
Explanation:
- Test Procedure: The testbench applies different values of `data_in` and `select` and monitors the outputs (`data_out0` and `data_out1`).
- The `$monitor` statement prints the values of the inputs and outputs at each time step, allowing you to verify if the correct output is activated based on the selection line.

Conclusion:

- The above Verilog code demonstrates the implementation of 1-to-2, 1-to-4, and 1-to-8 De-Multiplexers.
- The `assign` statements are used to route the input to the appropriate output based on the selection signals.
- The testbench verifies the functionality of the De-MUX by applying different combinations of inputs and selection lines and monitoring the outputs.

This can be easily extended for larger De-MUX configurations or for more complex routing logic.

4	a	Simplify the Boolean function F(w, x, y, z) = ∑ (4, 5, 6, 7, 12) with don't care function d(w, x, y, z) = ∑ (0,8,13)	[10]	1	L3



5	a	Explain the single bus structure with a neat labeled diagram. Single Bus Structure The single bus structure is a design used in digital systems (particularly in computer architecture) where multiple components share a common communication line (or bus) to transfer data. In this structure, various functional units (such as registers, ALU, memory) are connected to a single data bus, and data is transferred between them through the bus. The central bus acts as a medium for data exchange, with the components communicating via this shared bus. Key Components of the Single Bus Structure: 1. Registers: These are temporary storage locations within the system where data can be loaded or read. Registers may include special-purpose registers like the accumulator, program counter, etc.	[05]	3	L2

2. ALU (Arithmetic Logic Unit): The ALU performs arithmetic and logical operations on the data. It can operate on data stored in registers and can send results back to registers.

3. Memory: Memory units store data that can be fetched, written, or read during the execution of instructions. In this structure, the memory is usually connected to the bus so that data can be written to or read from memory.

4. Control Unit (CU): The control unit coordinates and controls the flow of data between registers, the ALU, and memory. It generates control signals that determine which component (register, memory, or ALU) is active during a given cycle.

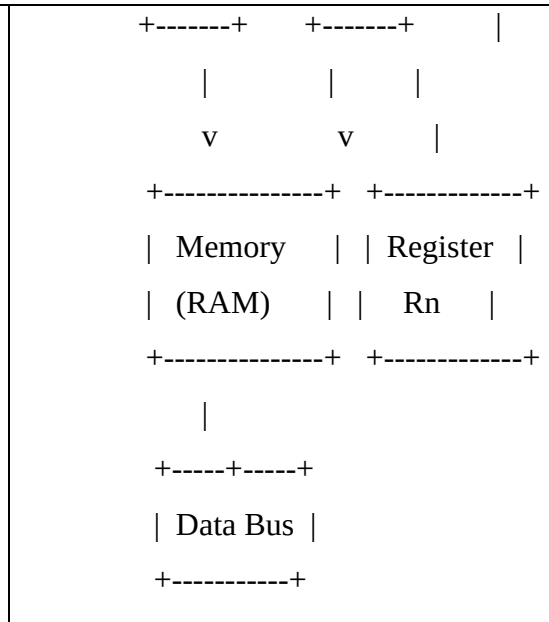
5. Bus: A set of shared lines (data bus, control bus, and address bus) used to transfer data, addresses, and control signals between the components.

The Single Bus Structure: Diagram and Explanation

Below is a simplified block diagram of a typical single-bus structure, which is commonly used in computers for data transfer.

Block Diagram:

```
          +------------------+
          |  Control Unit    |
          +------------------+
              | | |
              v v v
           +---------+--------------------+
           |         |                    |
           |         |                    |
       +-----------+ +-----------+ +-------------------+
       | Register | | Register | | ALU (Arithmetic   |
       |   R1     | |   R2     | |   Logic Unit)     |
       +-----------+ +-----------+ +-------------------+
           |           |             |
           |           |             |
```



Explanation of the Components and Connections:

1. Control Unit (CU):
- The Control Unit (CU) generates the necessary control signals (like read, write, and select) for coordinating the operations of other components (like registers, ALU, and memory). It tells the bus which component should be active during each cycle.
- It controls when data should be read from or written to memory, when data should be sent to or received from the ALU, and which register should store or retrieve data.

2. Registers:
- The registers are used to temporarily store data during processing. For instance, in this diagram, we have registers like R1, R2, and Rn.
- A register can send or receive data over the bus. The data bus is used to transfer data from one register to another or between a register and memory or the ALU.

3. ALU (Arithmetic and Logic Unit):
- The ALU performs arithmetic (e.g., addition, subtraction) and logical operations (e.g., AND, OR) on the data received from the registers or memory.
- The ALU can send its result back to a register or memory via the bus. It operates on data placed on the bus, which may come from different components in the system.

4. Memory (RAM):

- Memory stores data and instructions that are used during computation. It is connected to the bus so that the Control Unit can read from or write to memory when needed.
- The memory can send data to the ALU or registers, and it can receive data from registers or the ALU.

5. Data Bus:
- The data bus is the shared communication line that carries the actual data between the various components (registers, ALU, and memory).
- The size of the data bus (i.e., the number of lines) determines how much data can be transferred at once. Common sizes are 8, 16, or 32 bits.

6. Control Bus:
- The control bus consists of signals that dictate the operations of the components. These control signals tell which operation should take place (e.g., "read" or "write" to memory or "add" in the ALU).
- The control unit generates these signals, and they direct the operation of the other components (registers, memory, ALU).

7. Address Bus:
- The address bus carries the addresses from the control unit to memory, indicating where data should be read from or written to in memory.
- The size of the address bus determines the addressing capability (i.e., how much memory can be addressed).

Working of the Single Bus Structure:

In a single-bus architecture, data is transferred sequentially over the bus. Here is how it works step by step:

1. Fetch:
- The control unit generates an address for memory and fetches an instruction or data from memory.
- The address is sent over the address bus, and the data is placed on the data bus.

2. Decode:

- The instruction fetched from memory is sent to the decoder, and the control unit decodes the instruction to generate the necessary control signals.

3. Execution:

- Depending on the decoded instruction, the data is routed through the bus to the appropriate register or ALU.
- For example, if an addition operation is required, the two operands are sent to the ALU over the bus, and the result is stored in a register or memory.

4. Write-back:

- If the result needs to be written back to memory or a register, the control unit generates a signal to write the data from the bus to the destination (e.g., memory or a register).

Advantages of Single Bus Structure:

- Simplified Design: Since there is only one bus for communication, the design is simpler and requires fewer connections between components.
- Reduced Hardware Complexity: Fewer lines are needed for interconnecting components, which can reduce the overall hardware complexity and cost.
- Flexibility: Multiple components can share the same bus, allowing dynamic data transfer.

Disadvantages of Single Bus Structure:

- Bus Contention: Only one component can use the bus at a time, leading to possible delays when multiple components need to access the bus simultaneously.
- Performance Limitation: Since all components share the same bus, performance may degrade as the system scales up (e.g., with more components or higher data throughput).
- Increased Latency: The sequential nature of the bus can lead to increased latency when transferring data, particularly when multiple operations need to be performed.

Conclusion:

The single bus structure is a simple and effective way to interconnect different components of a computer system. It allows data to flow between registers, memory, and the ALU through a common bus. However, while it simplifies hardware design, it also introduces potential bottlenecks due to bus contention, limiting performance in more complex systems.

	b	Explain branching, in brief, using an example Branching in Computing Branching refers to the mechanism in a computer program where the control flow of the program can change direction based on certain conditions, typically using conditional statements (like `if`, `else`, `switch`) or loops (like `for`, `while`). This allows the program to make decisions and execute different instructions based on certain criteria or inputs. How Branching Works: When a program encounters a branch instruction, it decides which sequence of instructions to execute next based on the result of a condition. If the condition evaluates to true, the program follows one path (a "branch"); otherwise, it follows a different path. Branching can be either: - Conditional Branching: Executes one set of instructions if a condition is true, and another set if it's false. - Unconditional Branching: Jump to another part of the program without any condition (e.g., goto statements or function calls). Example of Conditional Branching Consider the following simple example in C programming language: ```c #include <stdio.h> int main() { int number; // Input a number printf("Enter a number: "); scanf("%d", &number); // Conditional branching if (number > 0) { printf("The number is positive.\n"); } else if (number < 0) { printf("The number is negative.\n"); } else { printf("The number is zero.\n"); } return 0; } ``` Explanation of the Example: 1. Input: The program asks the user to enter a number. 2. Condition: - The first condition `if (number > 0)` checks if the number is positive. - If the condition is true, it prints `"The number is positive"`.	[05]	3	L2

		- If the number is not positive, it checks the next condition `else if (number < 0)`, which checks if the number is negative. - If neither condition is true (i.e., the number is zero), it goes to the final `else` block, printing `"The number is zero"`. Flow of Execution (Branching Decision): - If the user enters `5`, the first condition `number > 0` is true, so the program prints `"The number is positive"`. - If the user enters `-3`, the second condition `number < 0` is true, so the program prints `"The number is negative"`. - If the user enters `0`, none of the previous conditions are true, and the program prints `"The number is zero"`. Branching in Low-Level Instructions In assembly or machine-level programming, branching typically involves jump instructions, where the program counter (PC) is changed based on conditions: - Conditional Jump: For example, an instruction like `JZ` (jump if zero) will move the program counter to a new address if a previous result was zero. - Unconditional Jump: A `JMP` instruction will unconditionally change the program counter, essentially redirecting the program's execution to a new address. Branching with Loops In addition to simple `if-else` branching, loops also involve branching. For example, consider a `while` loop: ```c int i = 0; while (i < 5) { printf("%d\n", i); i++; } ``` In this case: - The condition `i < 5` is evaluated before each iteration. If true, the body of the loop executes, and if false, the loop terminates. Conclusion: Branching allows programs to make decisions and alter their behavior based on conditions, enabling dynamic and flexible execution. It is a fundamental concept in programming that helps implement complex logic, such as error handling, conditional operations, and loops.			
6	a	What is a multiplexer? Design 3:1 multiplexer using 2:1 multiplexers What is a Multiplexer? A Multiplexer (MUX) is a combinational circuit that selects one of many inputs and forwards it to a single output line. It is often referred to as a "data selector" because it selects one of several data sources to send to the output, based on a selection signal. - Basic components of a multiplexer:	[10]	3	L3

- Inputs (Data lines): Multiple data lines are connected to the MUX, from which one is selected.
- Selection lines (Control lines): A set of binary control lines that determine which input is routed to the output.
- Output line: A single output line that carries the data from one of the selected inputs.

For example, a 2-to-1 multiplexer has 2 data inputs, 1 selection line, and 1 output. A 4-to-1 multiplexer has 4 data inputs, 2 selection lines, and 1 output.

3:1 Multiplexer

A 3-to-1 multiplexer (3:1 MUX) has 3 data inputs (`D0`, `D1`, `D2`), 2 selection lines (`S0`, `S1`), and 1 output (`Y`). The 2 selection lines can represent 4 possible combinations (00, 01, 10, 11), but since we have only 3 inputs, the 4th combination is unused. The output will be determined based on the value of the selection lines as follows:

S1	S0	Output (Y)
0	0	D0
0	1	D1
1	0	D2
1	1	(unused)

3:1 MUX Using 2:1 MUXes

To design a 3:1 multiplexer using 2:1 multiplexers, we can use the fact that a 2:1 multiplexer can be used to select between two inputs. The general idea is to create a tree of 2:1 multiplexers that can handle the 3 inputs and 2 selection lines.

Design Steps:

1. Step 1: Select between D0 and D1:
   - We use a 2:1 multiplexer to choose between `D0` and `D1` based on selection line `S0`.
   - The selection line `S0` will determine which of `D0` or `D1` is passed to the next stage.

2. Step 2: Select between D2 and the output from Step 1:
   - Another 2:1 multiplexer is used to choose between `D2` and the output from Step 1 based on selection line `S1`.

In essence, the first stage of multiplexing selects between `D0` and `D1`, and the second stage of multiplexing selects between the result of that stage and `D2`.

3:1 MUX Design using 2:1 MUXes (Verilog)

```verilog
module mux2to1 (
    input wire a,     // First input
    input wire b,     // Second input
    input wire sel,   // Selection input
    output wire out   // Output
);
    assign out = (sel) ? b : a;  // Output is 'b' if sel is 1, else 'a'
endmodule
```

```
module mux3to1 (
    input wire D0,     // First input
    input wire D1,     // Second input
    input wire D2,     // Third input
    input wire S0,     // First selection line
    input wire S1,     // Second selection line
    output wire Y      // Output
);
    wire mux0_out, mux1_out;  // Intermediate outputs for 2:1 multiplexers

    // First stage: MUX between D0 and D1, controlled by S0
    mux2to1 mux0 (
        .a(D0),
        .b(D1),
        .sel(S0),
        .out(mux0_out)
    );

    // Second stage: MUX between mux0_out and D2, controlled by S1
    mux2to1 mux1 (
        .a(mux0_out),
        .b(D2),
        .sel(S1),
        .out(Y)
    );

endmodule
```

Explanation of the Verilog Code:

1. 2:1 Multiplexer (`mux2to1`):
   - The `mux2to1` module is a simple 2:1 multiplexer that selects between two inputs (`a` and `b`) based on the selection signal `sel`.
   - If `sel` is `0`, the output is `a`, and if `sel` is `1`, the output is `b`.

2. 3:1 Multiplexer (`mux3to1`):
   - The `mux3to1` module uses two `mux2to1` modules to implement the 3:1 MUX.
   - In the first stage, a 2:1 MUX selects between `D0` and `D1` based on selection line `S0`, and the output is stored in `mux0_out`.
   - In the second stage, another 2:1 MUX selects between `mux0_out` (the result of the first MUX) and `D2` based on selection line `S1`, producing the final output `Y`.

Truth Table:

The 3:1 multiplexer operates according to the following truth table:

S1	S0	Output (Y)
0	0	D0
0	1	D1
1	0	D2

How the 2:1 MUXes Work Together:

		- First Stage: The 2:1 MUX with inputs `D0` and `D1` uses the selection line `S0` to choose between `D0` and `D1`. - If `S0` is `0`, the output of this MUX will be `D0`. - If `S0` is `1`, the output will be `D1`. - Second Stage: The second 2:1 MUX uses the result from the first stage (let's call it `mux0_out`) and `D2` as inputs, and the selection line `S1` determines the output: - If `S1` is `0`, the output will be the value of `mux0_out` (which is either `D0` or `D1` depending on `S0`). - If `S1` is `1`, the output will be `D2`. Thus, the final output `Y` is selected correctly based on the values of `S0` and `S1`. Conclusion: In summary, a 3:1 multiplexer can be implemented using two 2:1 multiplexers by structuring the design as a two-stage process. The first 2:1 MUX selects between two inputs, and the second 2:1 MUX selects between the result of the first and a third input. This design efficiently reuses the 2:1 MUX block to build a more complex 3:1 multiplexer.			

CI **CCI** **HoD**