CE221-L Digital Logic Design Lab

Lab # 11

Submitted by:

- **Shayan Rizwan** [2024585]

Submitted to: Sir Irfanullah

Semester: 3rd

Faculty of Computer Science and Engineering

GIK Institute of Engineering Sciences and Technology

Home Tasks: Finite State Machine Implementation in Verilog

Lab Activity Description

In this lab activity, you are required to design and implement Finite State Machines (FSMs) in Verilog using both Moore and Mealy models. The objective is to detect a specific binary sequence X, where X corresponds to the last non-zero digit of your registration number represented in binary (e.g., $7_{10} = 0111_2$). The complete binary representation of your registration number will serve as the input signal to the FSMs.

NOTE: Input signals are your registration number. For example, Reg # 2024123 (00100000001001000001001000011)

Required Tasks

You must perform the following tasks:

1. **Sequence Identification:**

•Identify the sequence X based on the last non-zero digit of your registration number.

2. **FSM Design:**

•Construct a state diagram and develop a state transition table for both Moore and Mealy FSMs.

3. **Verilog Implementation:**

•Write Verilog code for Moore and Mealy FSMs that detect the selected sequence X.

4. **Test Bench Development:**

•Develop appropriate test benches to verify the functionality of each FSM model.

5. **Simulation and Visualization:**

•Simulate the designs and visualize their behavior using waveform outputs.

6. **Analysis and Comparison:**

•Analyze and compare the operational differences between Moore and Mealy implementations.

Example

Registration Number: 2024123
Binary Representation: 00100000001001000001001000 11
Last Non-Zero Digit: 3
Binary Sequence X: 0011 (since $3_{10} = 0011_2$)

X = last non-zero digit of reg
X = 0 1 0 1 (== 5)

INPUT = 0010 0000 0010 0100 **0101** 1000 **0101**

Mealy FSM [Overlapping Sequence Detector]

In a Mealy type FSM, the **number of states** in the detector is the **same** as the **number of bits in the sequence**.

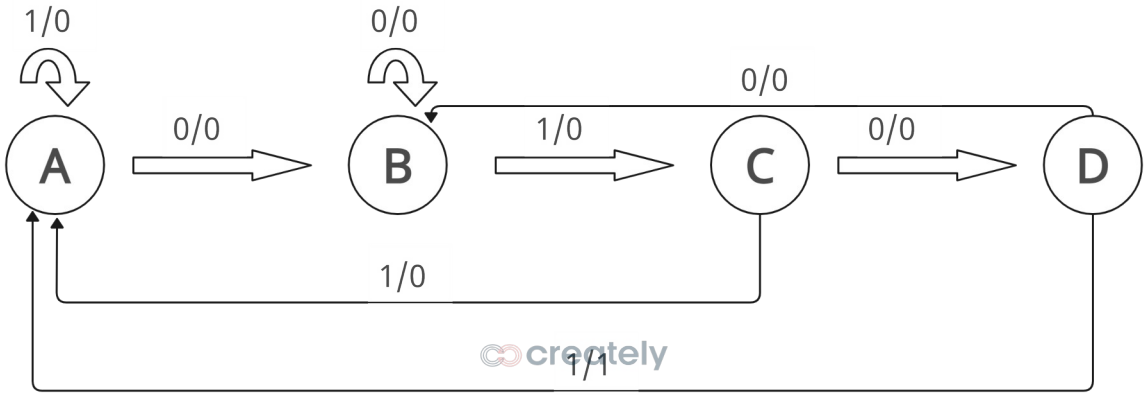
Drawing up the state transition table:

State	Received Bits	Remarks
A	-	Starting state
B	0	1-bit received correctly
C	01	2-bit received correctly
D	010	3-bit received correctly

When the 4th correct bit in the sequence is received, the output of the circuit will be 1.

Drawing up the **state transition diagram**:

[CONTINUED ON NEXT PAGE]



From the state transition diagram shown above, we make the **state table**:

Present State	Next State		Output	
	X = 0	X = 1	X = 0	X = 1
A	B	A	0	0
B	B	C	0	0
C	C	A	0	0
D	B	A	0	1

In the first and the last row, we are getting the same next states, for the X = 0, and, X = 1. But, since, the outputs are different, none of the states are identical, and therefore, there is **no redundant states** in the state diagram.

Since we have four states, it can be represented using two bits. Therefore, let the MSB equal to Q0 and the LSB equal to Q1.

State assignment:

A – 00

B – 01

C – 10

D – 11

Based on these state assignments, the state table can be rewritten in the following manner:

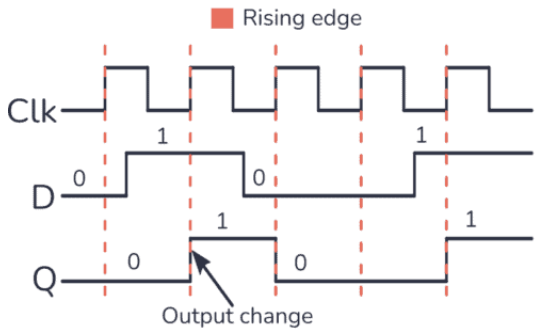
Present State	Next State		Output	
	X = 0	X = 1	X = 0	X = 1
00	01	00	0	0
01	01	10	0	0
10	10	00	0	0
11	01	00	0	1

The next task at hand is to select the appropriate flip flop, and find the excitations for each flipflop.

For the purpose of this task, we select the **D-flipflops**. Now, since there are two bits in the state assignments, we will select two (2) D-flipflops.

[**NOTE**: If we were to use any other flip-flop, then we would look at the excitation table of that particular flip-flop, and then make the final state table.]

For a **D flip-flop**, the **output at the next clock edge equals the value applied to the input D at that edge**. Therefore, whatever output we want in the next state must be applied to the D input beforehand. The timing diagram given below illustrates that:



For this, we rewrite the state table in a different way:

Input	Present State		Next State		Inputs To Flipflop		Output
X	Q1	Q0	Q1(t+1)	Q0(t+1)	D1	D0	Y
0	0	0	0	1	0	1	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	1

The next step is to find the algebraic expression of the inputs of the flipflop in terms of the input X and the present states, Q1 and Q0, because by applying these inputs to the D flipflop, we will get the required next state.

Now, given the state table, the state encoding that we set, and have decided upon the choice of flip-flop, as well as the boolean equations for next-state and output logic, we move towards the **verilog implementation**.

Design Source [Code]

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/15/2025 06:00:48 PM
// Design Name:
// Module Name: sequence_detector
// Project Name:
```

```
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
//////////////


module sequence_detector (
    input  wire clk,
    input  wire rst,
    input  wire X,
    output reg  Y
);

    // State bits
    reg Q1, Q0;
    reg D1, D0;

    // -------------------------
    // Combinational Logic
    // -------------------------
    always @(*) begin

        // Flip-flop input equations
        D0 = ~X;
        D1 = (~Q1 & Q0 & X) | (Q1 & ~Q0 & ~X);

        // Mealy output
        // Default
        Y = 1'b0;
        if (Q1 & Q0 & X)
            Y = 1'b1;
    end

    // -------------------------
    // Sequential Logic
    // -------------------------
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            Q1 <= 1'b0;
            Q0 <= 1'b0;
        end else begin
            Q1 <= D1;
            Q0 <= D0;
        end
    end

endmodule
```

Although the Mealy output for this sequence detector is logically $Y = X * Q1 * Q0$, it is not written directly as Y = X & Q1 & Q0 in Verilog to follow good FSM and RTL design practices. Implementing the output inside a combinational always @(*) block with a clear default assignment improves readability and prevents accidental latch inference. This style also makes the FSM behavior easier to relate to the state diagram by explicitly showing that the output is asserted only for a specific state–input combination. Moreover, structured conditional logic is safer for Mealy machines, as it reduces the risk of unintended glitches and allows the design to be extended or modified more easily. Therefore, even though the Boolean equation is correct, the chosen coding approach is more robust, clear, and scalable for practical FSM implementations.

In an FSM, the **sequential logic block** and the **combinational logic block** work together to control state transitions in a clean and predictable way.

The **sequential logic block** (usually written as `always @(posedge clk or posedge reset)`) is responsible for **storing the current state** of the machine. It models the physical **D flip-flops** used in hardware. On every rising edge of the clock, this block loads the value of the *next state* into the *present state*. If a reset is asserted, it forces the FSM into a known initial state. Importantly, this block does **not decide** what the next state should be—it only updates the state when the clock arrives.

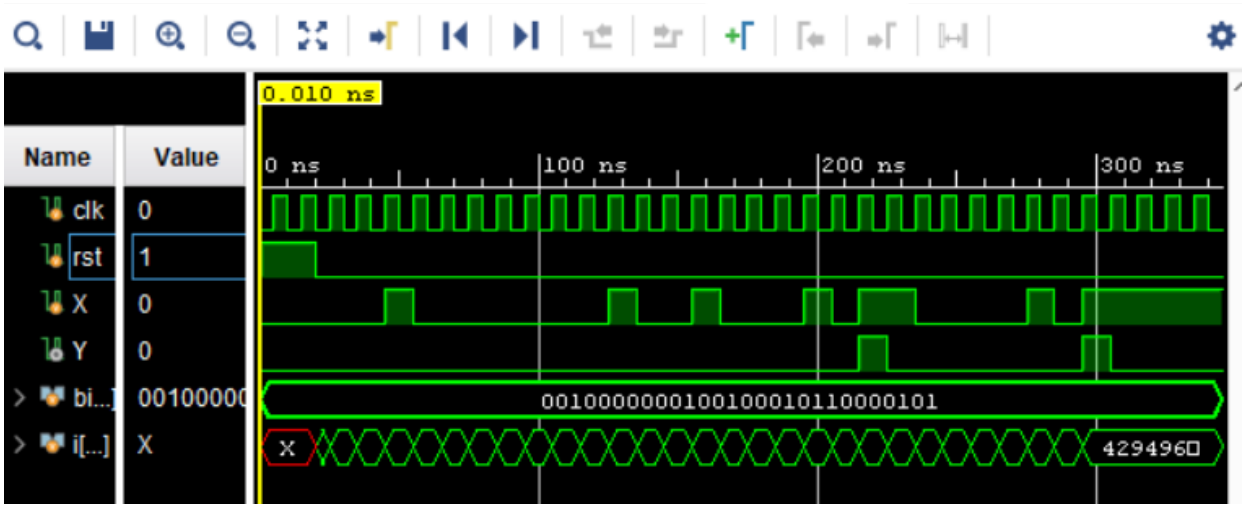
The **combinational logic block** (written as `always @(*)`) is responsible for **decision-making**. It computes the **next state** and the **output** based on the current state (from the sequential block) and the input `X`. This block directly represents the Boolean logic derived from the state table or equations (such as D0, D1, and Y). Since it is combinational, any change in the input or present state immediately updates the next-state signals.

Test Bench [Code]

```
`timescale 1ns / 1ps

module sequence_detector_tb;

    reg clk;
    reg rst;
    reg X; // Input sequence bit (register)
    wire Y; // Output detection signal (wire)

    sequence_detector dut (
        .clk(clk),
        .rst(rst),
        .X(X),
        .Y(Y)
    );

    always #5 clk = ~clk;

    reg [27:0] bitstream = 28'b0010000000100100010110000101;
    integer i;

    initial begin
        clk = 0;
        rst = 1;
        X = 0;
        #20 rst = 0;
```

```
        for (i = 27; i >= 0; i = i - 1) begin
            @(posedge clk);
            X = bitstream[i];
        end

        repeat (5) @(posedge clk);
        $stop;
    end

endmodule
```

Output:



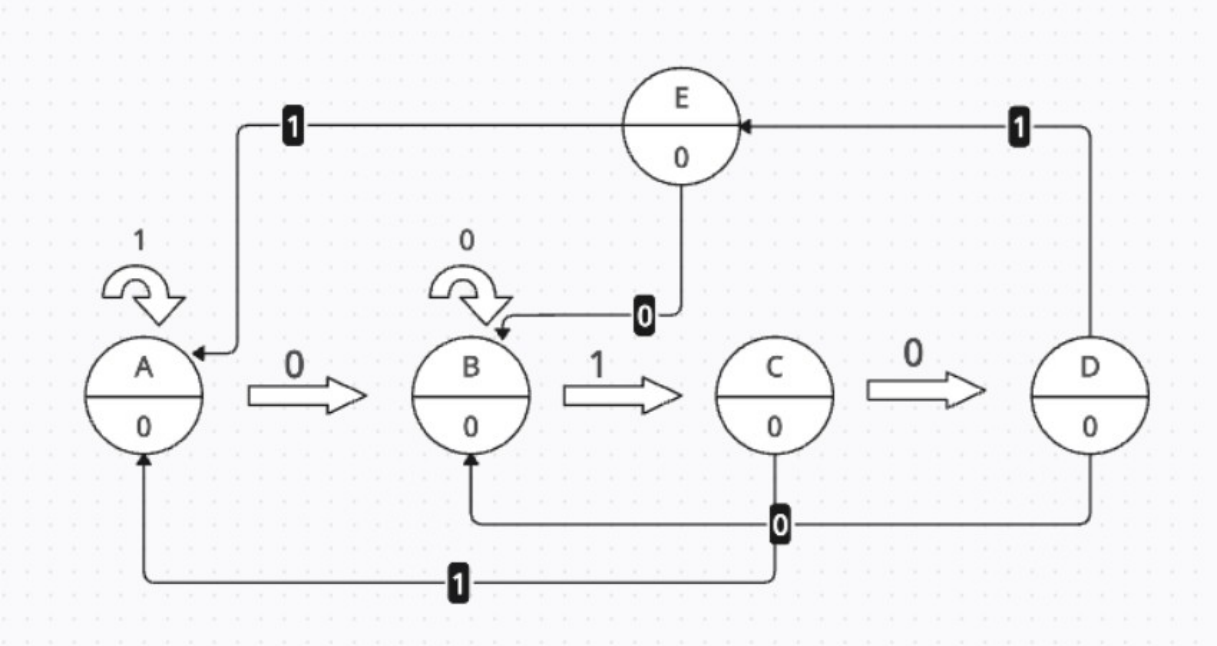
Moore FSM [Overlapping Sequence Detector]

In a Moore type FSM, the **number of states** in the detector is the one more than the number of bits in the sequence.

Drawing up the state transition table:

State	Received Bits	Remarks
A	-	Starting state
B	0	1-bit received correctly
C	01	2-bit received correctly
D	010	3-bit received correctly
E	0101	Correct sequence is received

Drawing up the **state transition diagram**:



Since we have five states, it can be represented using atleast three (3) bits.

State assignment:

A – 000

B – 001

C – 010

D – 011

E – 100

Design Source [Code]

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/16/2025 12:02:23 AM
// Design Name:
// Module Name: sequence_detector
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
//////////////////////////////////////////////////////////////////////////////
//////////////



module sequence_detector (
    input clk,
    input rst,
    input X,
    output reg Y
);

    // State bits (flip-flops)
    reg Q0, Q1, Q2;        // Q0 = MSB, Q2 = LSB
    reg D0, D1, D2;        // D inputs

    // ==========================
    // Sequential Logic (D-FFs)
    // ==========================
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            Q0 <= 1'b0;
            Q1 <= 1'b0;
            Q2 <= 1'b0;
        end else begin
            Q0 <= D0;
            Q1 <= D1;
            Q2 <= D2;
        end
```

```
    end

    // ==========================
    // Combinational Logic
    // ==========================
    always @(*) begin
        // Default hold state
        D0 = Q0;
        D1 = Q1;
        D2 = Q2;

        // State A: 000
        if (Q0==0 && Q1==0 && Q2==0) begin
            if (X==0) begin D2=1; end // B
        end

        // State B: 001
        else if (Q0==0 && Q1==0 && Q2==1) begin
            if (X==1) begin D1=1; D2=0; end // C
        end

        // State C: 010
        else if (Q0==0 && Q1==1 && Q2==0) begin
            if (X==0) begin D1=1; D2=1; end // D
            else begin D1=0; D2=0; end     // A
        end

        // State D: 011
        else if (Q0==0 && Q1==1 && Q2==1) begin
```

```
            if (X==1) begin D0=1; D1=0; D2=0; end // E
            else begin D1=0; D2=1; end            // B
        end

        // State E: 100
        else if (Q0==1 && Q1==0 && Q2==0) begin
            if (X==0) begin D0=0; D2=1; end // B
            else begin D0=0; D1=0; D2=0; end // A
        end
    end

    // ==========================
    // Moore Output Logic
    // ==========================
    always @(*) begin
        if (Q0==1 && Q1==0 && Q2==0)
            Y = 1'b1;
        else
            Y = 1'b0;
    end

endmodule
```

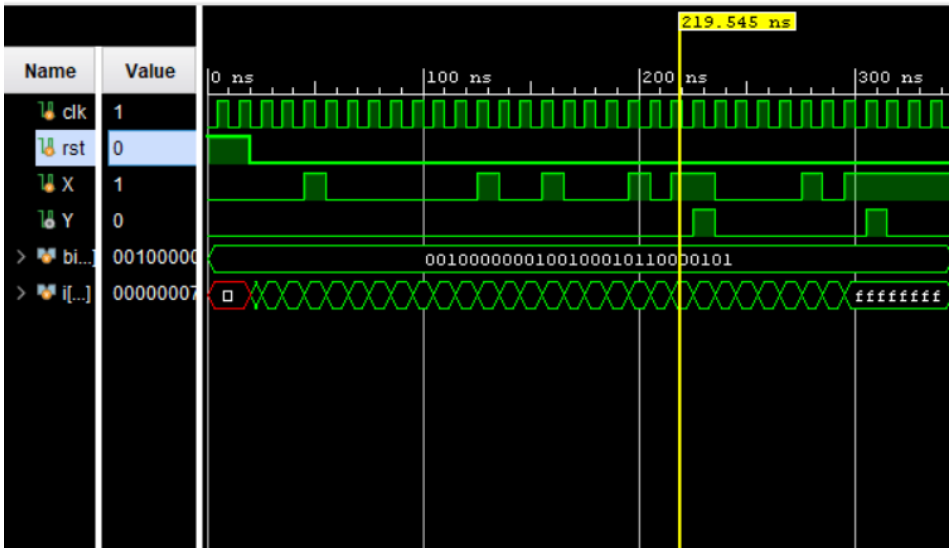
Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////
//////////////
```

```
// Company:
// Engineer:
//
// Create Date: 12/16/2025 12:03:23 AM
// Design Name:
// Module Name: sequence_detector_tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


`timescale 1ns / 1ps


module sequence_detector_tb;


    reg clk;
    reg rst;
    reg X; // Input sequence bit (register)
    wire Y; // Output detection signal (wire)
```

```
    sequence_detector dut (
        .clk(clk),
        .rst(rst),
        .X(X),
        .Y(Y)
    );

    always #5 clk = ~clk;

    reg [27:0] bitstream = 28'b0010000000100100010110000101;
    integer i;

    initial begin
        clk = 0;
        rst = 1;
        X = 0;
        #20 rst = 0;

        for (i = 27; i >= 0; i = i - 1) begin
            @(posedge clk);
            X = bitstream[i];
        end

        repeat (5) @(posedge clk);
        $stop;
    end

endmodule
```

Output



The same testbench can be used for both Mealy and Moore FSM implementations since the input stimulus, clocking, and reset behavior remain unchanged. The difference lies only in the internal FSM logic and the timing of the output response.

Moore and Mealy finite state machines differ mainly in how their outputs are generated. In a **Moore FSM**, the output depends **only on the current state**, so the output changes **only on clock edges** when the state changes. Because of this, the output appears **one clock cycle after** the input sequence is fully detected, making Moore machines more predictable and stable.

In contrast, a **Mealy FSM** generates its output based on the **current state and the current input**. This allows the output to change **immediately within the same clock cycle** when the final input of the sequence arrives. As a result, Mealy machines typically detect sequences **faster** than Moore machines.