Lab Final – Verilog

Developing code for a **D-flip flop** with an **asynchronous active low reset input**:
The design source is as follows:

```
module d_flipflop(d, clk, reset_n, q);

input d;
input clk;
input reset_n;
output reg q;

always@(posedge clk or negedge reset_n)

begin

	if(~reset_n) // If the active low signal is asserted of the reset;

		q <= 1'b0;
	else
		q <= d;

end

endmodule
```

The equivalent testbench code is the following:

```
`timescale 1ns / 1ps

module d_flipflop_tb;

   // Testbench signals (reg for inputs, wire for outputs)
   reg d;
   reg clk;
```

```
    reg reset_n;
    wire q;

    // Instantiate the DUT (Device Under Test)
    d_flipflop uut (
        .d(d),
        .clk(clk),
        .reset_n(reset_n),
        .q(q)
    );

    // Clock generation: 10 ns period
    always #5 clk = ~clk;

    initial begin
        // Initialize signals
        clk = 0;
        d = 0;
        reset_n = 1;

        // Apply asynchronous reset
        #3  reset_n = 0;   // Assert reset (active low)
        #7  reset_n = 1;   // Deassert reset

        // Apply test vectors
        #10 d = 1;
        #10 d = 0;
        #10 d = 1;

        // Apply reset again while clock is running
        #7  reset_n = 0;
        #5  reset_n = 1;

        #20 $finish;
    end

endmodule
```

Differentiation between a Moore and Mealy FSM

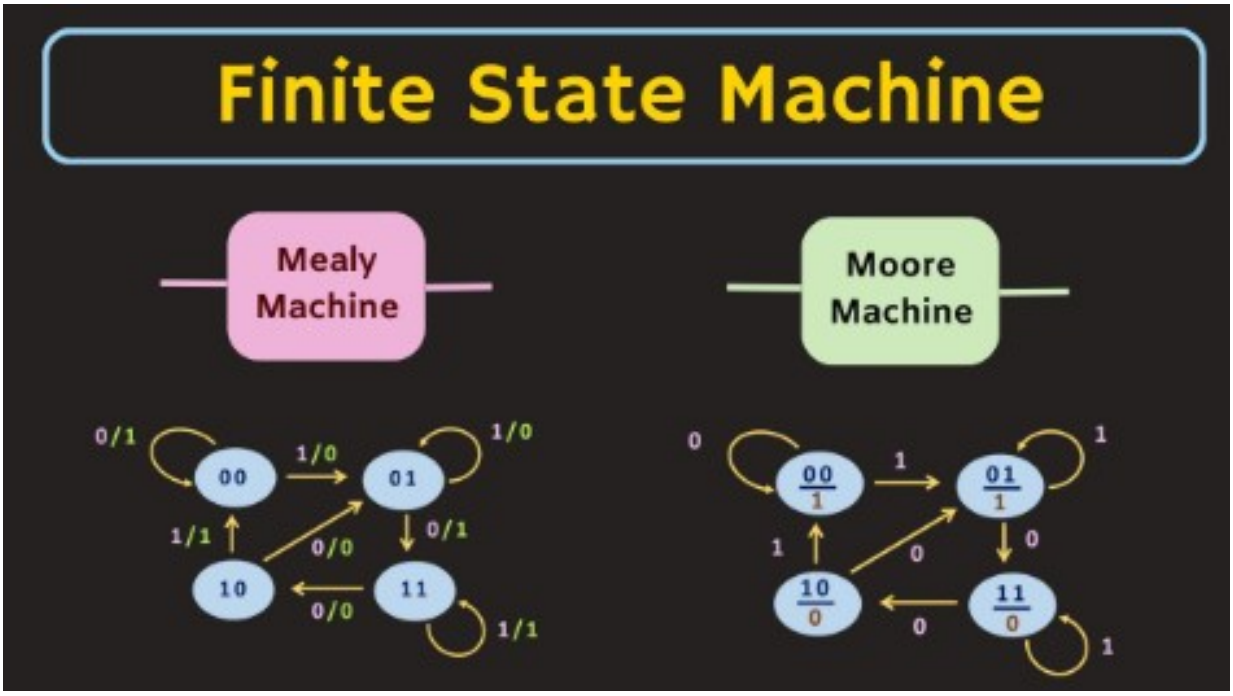
Table 1: Sequence Detector - Mealy FSM

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

Table 2: Sequence Detector - Moore FSM

In a Moore type FSM, the **number of states** in the detector is **one more (+1)** than the **number of bits in the sequence**.

Moore Machine	Mealy Machine
Output depends only upon the present state.	Output depends on the present state as well as present input.
Moore machine also places its output in the state.	**Mealy Machine** places its output on the transition.



Task: Design a Moore Finite State Machine (FSM) for Sequence Detection

Problem Statement:
Design a **non-overlapping** sequence detector that detects the binary sequence `1011`. The detector should have:

►**Input:** `din` (1-bit serial input, sampled on positive clock edge)
►**Output:** `detect` (1-bit, goes high for one clock cycle when the sequence is detected)
►**Reset:** Asynchronous active-low reset (`reset_n`)

Specifications:

1. Use **Moore machine** style (output depends only on current state).
2. When the **exact sequence** `1011` is received, `detect` should be `1` **in the same clock cycle** when the last bit (`1`) is received.
3. After detection, the FSM should restart looking for a new sequence from the next bit (non-overlapping detection).
4. Assume sequence bits arrive one per clock cycle.

Writing the **design source code**:

```
module moore_fsm(
        input clk;
        input reset_n;
        input x; // Input bit
        output y // there is no comma at the end here
        );

        reg[2:0] state_reg, state_next;
        localparam A = 3'b000;
        localparam B = 3'b001;
        localparam C = 3'b010;
        localparam D = 3'b011;
        localparam E = 3'b100;

        // state register
        always @ (posedge clk, negedge reset_n)
```

```
	begin
		if(~reset_n)
			state_reg <= A;
		else
			state_reg <= state_next;
	end

	// Next state logic
	always @ (*) // start the always block by defining your begin and end statements in the beginning
	begin
		case(state_reg) // make sure to include the endcase (IMPORTANT)
			A: if(x)
				state_next = B
			else
				state_next = A
			B: if(x)
				state_next = B
			else
				state_next = C
			C: if(x)
				state_next = D
			else
				state_next = A
			D: if(x)
				state_next = E
			else
				state_next = A
			E: if(x)
				state_next = A
			else
				state_next = A
			default: state_next = state_reg;
		endcase
	end

	// Output logic
	assign y = (state_reg == E);

endmodule
```

Developing the **test bench code** for this design source:

```
module moore_fsm_tb;

    // Outputs: wire
    // Inputs: reg

    reg clk;
    reg reset_n;
    reg x;
    wire y;

    // Instantiate the Unit Under Test
    moore_sequence_detector dut(
        .clk(clk),
        .reset_n(reset_n),
        .x(x),
        .y(y)
    );

    // Generate a clock signal
    always #5 clk = ~clk;

    reg [7:0] simple_test = 8'b10111011;
    // Should detect ONLY once (non-overlapping)
    integer i; // iterative variable to loop through all the input bit stream and assign to the
input var;

    initial
    begin
        clk = 0;
        reset_n = 0; // Assert reset (active low)
        x = 0;
        #20 reset_n = 1; // Release reset after 20ns

        for (i = 7; i >= 0; i = i-1) // Count DOWN from 7 to 0
        begin
            @(posedge clk);
```

```
        x = simple_test[i];
    end

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

For an active low asynchronous reset; the following code block is crucial in the testbench:

```
reset_n = 0; // Assert reset (active low)

x = 0; // hardcode the default input bit

#20 reset_n = 1; // Release reset after 20ns
```

🔷 FSM PRACTICE QUESTION (LAB FINAL LEVEL)

Problem Title:

A university has installed a **smart entry gate** that controls access using an **ID card and a security PIN**. The system operates synchronously on a clock and has an **active-low asynchronous reset**.

System Inputs

- `card` :
  `1` → Valid ID card detected
  `0` → No card / invalid card
- `pin` :
  `1` → Correct PIN entered
  `0` → Incorrect PIN or no PIN
- `clk` : System clock
- `reset_n` : Active-low asynchronous reset

System Outputs

- `gate_open`:
  `1` → Gate opens
  `0` → Gate remains closed
- `alarm`:
  `1` → Alarm triggered
  `0` → Alarm off

🔷 Functional Requirements

1. **Idle State**
   - System starts in the **IDLE** state.
   - Gate is closed and alarm is off.
2. **Card Verification**
   - If a **valid card** is detected (`card = 1`), the system moves to a **PIN_WAIT** state.
   - If `card = 0`, the system remains in IDLE.
3. **PIN Verification**
   - In the **PIN_WAIT** state:
     - If the correct PIN is entered (`pin = 1`), the gate opens for **one clock cycle**.
     - After opening, the system returns to IDLE.
     - If an incorrect PIN is entered (`pin = 0`), the system moves to an **ALARM** state.
4. **Alarm State**
   - In the **ALARM** state:
     - The alarm is activated.
     - The system remains in this state until reset is asserted.
     - The gate must remain closed.

5. **Reset Behavior**
    - When `reset_n = 0`, the system must **immediately return to IDLE**, regardless of the clock.

🔷 Tasks (Exactly Exam Style)

You are required to:

1. **Identify and list all states** of the FSM.
2. **Draw a complete state transition diagram** (clearly labeled with inputs and outputs).
3. **Specify whether the FSM is Moore or Mealy**, with justification.
4. **Write synthesizable Verilog code** to implement the FSM in Vivado.
    - Use **behavioral modeling**
    - Use **non-blocking assignments**
    - Use **asynchronous active-low reset**
5. Assume **one-hot or binary encoding** (your choice, but be consistent).

🔷 Important Exam Hints (Read Carefully)

- Outputs must be **deterministic for each state**
- Avoid latches
- Reset must override all logic
- Follow the **three-block FSM style** if possible:
    - State register
    - Next-state logic
    - Output logic

Any time an output is asserted for “one clock cycle”, Moore FSMs are usually preferred.

- If an input has *no meaning* in a state, **ignore it**.
- Any output that must last exactly one clock cycle is best implemented as a separate state.
- If outputs describe system behavior over a clocked interval, use Moore; if outputs must react instantly to inputs, use Mealy.
- If a state exists *only* to assert an output for one cycle, its exit must be unconditional.
  That means that, the input to transition into another state should simply be xx (don’t cares).

- In **both Moore and Mealy FSMs**, the **output logic is written inside an `always @(*)` block**.
  What changes is **what the output depends on**, **not how it is coded structurally.**
- If an output is assigned inside an `always` block, it must be declared as `reg`.

The **design source code** is as follows:

```
module fsm_practice(
    input clk,
    input reset_n,
    input [1:0] i,
    output reg [1:0] O
);

    reg [1:0] state_reg, state_next;

    localparam I  = 2'b00;
    localparam PW = 2'b01;
    localparam GO = 2'b10;
    localparam A  = 2'b11;

    // 1 State register
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n)
```

```
        state_reg <= I;
    else
        state_reg <= state_next;
end

// 2 Next-state logic
always @(*) begin
    state_next = state_reg; // default

    case (state_reg)
        I: begin
            if (i[1] == 1'b1)
                state_next = PW;
            else
                state_next = I;
        end

        PW: begin
            if (i[0] == 1'b1)
                state_next = GO;
            else
                state_next = A;
        end

        GO: begin
            state_next = I; // single-cycle pulse
        end

        A: begin
            state_next = A; // absorb state
        end
    endcase
end

// 3 Output logic (Moore)
always @(*) begin
    case (state_reg)
        I:  O = 2'b00;
        PW: O = 2'b00;
        GO: O = 2'b10;
        A:  O = 2'b01;
```

```
            default: O = 2'b00;
        endcase
    end

endmodule
```

The **test bench code** for the above task is as follows:

```
module fsm_practice_tb;

    reg clk;
    reg reset_n;
    reg [1:0] i;
    wire [1:0] O;

    // Instantiate DUT
    fsm_practice dut (
        .clk(clk),
        .reset_n(reset_n),
        .i(i),
        .O(O)
    );

    // Clock generation (10ns period)
    always #5 clk = ~clk;

    initial begin
        // Initial conditions
        clk = 0;
        reset_n = 0;
        i = 2'b00;

        // Hold reset
        #20;
        reset_n = 1;

        // ---- FSM stimulus ----

        // IDLE -> PIN_WAIT (card inserted)
        @(posedge clk);
```

```
        i = 2'b10;

        // PIN_WAIT -> GO (pin correct)
        @(posedge clk);
        i = 2'b11;

        // GO -> IDLE (automatic)
        @(posedge clk);
        i = 2'b00;

        // IDLE -> PIN_WAIT
        @(posedge clk);
        i = 2'b10;

        // PIN_WAIT -> ALARM (wrong pin)
        @(posedge clk);
        i = 2'b10;

        // Stay in ALARM
        repeat (2) @(posedge clk);

        // Reset from ALARM
        reset_n = 0;
        #10;
        reset_n = 1;

        repeat (3) @(posedge clk);
        $stop;
    end

endmodule
```

In Verilog behavioral modelling, there are three (3) core modules that we studied and implemented:

1. ALU
2. MUX
3. Decoder

ALU

Design source:

```
module alu( // Four (4) inputs and one (1) output
            input [7:0] operand_1,
            input [7:0] operand_2,
            input [2:0] command,
            input enable,
            output reg [15:0] out // declared as a reg since it will be used inside the always block
        );


        // Declare parameters for the different operations
        parameter ADD = 3'b000;
        parameter INC = 3'b001;
        parameter SUB = 3'b010;
        parameter MUL = 3'b011;
        parameter DIV = 3'b100;
        parameter SHL = 3'b101;
        parameter SHR = 3'b110;
        parameter AND = 3'b111;


        // A procedural block using behavioral modeling for a combinational circuit (ALU):
        always @ (operand_1, operand_2, command)
        begin
                case(command)
                        ADD : out = operand_1 + operand_2;
                        INC : out = operand_1 + 1;
                        SUB : out = operand_1 - operand_2;
                        MUL : out = operand_1 * operand_2;
```

```
                    DIV : out = operand_1 / operand_2;
                    SHL : out = operand_1 << 1'b1;
                    SHR : out = operand_1 >> 1'b1;
                    AND : out = operand_1 & operand_2;
                    default: result = 8'b0;  // Default case: Prevents latch formation
                endcase
        end

endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module alu_tb;

    // Inputs to DUT → reg
    reg [7:0] operand_1;
    reg [7:0] operand_2;
    reg [2:0] command;
    reg enable;

    // Output from DUT → wire
    wire [15:0] out;

    // Instantiate the DUT
    alu dut (
        .operand_1(operand_1),
        .operand_2(operand_2),
        .command(command),
        .enable(enable),
        .out(out)
    );

    initial begin
        // Initial values
        operand_1 = 8'd10;
        operand_2 = 8'd5;
        command   = 3'b000;
        enable    = 0;
```

```
        #10 enable = 1;   // Enable output

        // ADD
        #10 command = 3'b000; // ADD → 10 + 5 = 15

        // INC
        #10 command = 3'b001; // INC → 10 + 1 = 11

        // SUB
        #10 command = 3'b010; // SUB → 10 - 5 = 5

        // MUL
        #10 command = 3'b011; // MUL → 10 * 5 = 50

        // DIV
        #10 command = 3'b100; // DIV → 10 / 5 = 2

        // SHL
        #10 command = 3'b101; // SHL → 10 << 1 = 20

        // SHR
        #10 command = 3'b110; // SHR → 10 >> 1 = 5

        // AND
        #10 command = 3'b111; // AND → 10 & 5 = 0

        // Disable output (tri-state)
        #10 enable = 0;

        #10 $stop;
    end

endmodule
```

MUX

Design source:

```
module mux_8x1(
        input [7:0]i,
        input [2:0]s,
        output reg y
);

        // Procedural block: Behavioral modelling of a combinational circuit (MUX)
        always @ (*)
        begin
                case(s)
                        // y is getting assigned 1-bit inputs
                        3'b000 : y = i[0];
                        3'b001 : y = i[1];
                        3'b010 : y = i[2];
                        3'b011 : y = i[3];
                        3'b100 : y = i[4];
                        3'b101 : y = i[5];
                        3'b110 : y = i[6];
                        3'b111 : y = i[7];
                        default : y = 1'b0;
                endcase

        end
endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module mux_8x1_tb;

   // Inputs → reg
   reg [7:0] i;
   reg [2:0] s;
```

```
    // Output → wire
    wire y;

    // Instantiate the DUT
    mux_8x1 dut (
        .y(y),
        .i(i),
        .s(s)
    );

    initial begin
        // Apply input pattern
        i = 8'b1010_1101;  // Example data inputs

        // Select each input line one by one
        s = 3'b000; #10;   // y = i[0]
        s = 3'b001; #10;   // y = i[1]
        s = 3'b010; #10;   // y = i[2]
        s = 3'b011; #10;   // y = i[3]
        s = 3'b100; #10;   // y = i[4]
        s = 3'b101; #10;   // y = i[5]
        s = 3'b110; #10;   // y = i[6]
        s = 3'b111; #10;   // y = i[7]

        $stop;
    end

endmodule
```

Decoder

Design source:

```
module decoder_3x8(
    input [2:0] in,      // 3-bit input
    output reg [7:0] out  // 8-bit output (active high)
);

    always @(*) begin
        case(in)
            3'b000: out = 8'b00000001;
            3'b001: out = 8'b00000010;
            3'b010: out = 8'b00000100;
            3'b011: out = 8'b00001000;
            3'b100: out = 8'b00010000;
            3'b101: out = 8'b00100000;
            3'b110: out = 8'b01000000;
            3'b111: out = 8'b10000000;
            default: out = 8'b00000000; // Safety default
        endcase
    end

endmodule
```

Testbench:

```
module decoder_tb;

    // 1. Make test inputs
    reg [2:0] in;

    // 2. Get output from decoder
    wire [7:0] out;

    // 3. Connect to our decoder
    decoder_3x8 my_decoder (
```

```
    .in(in),
    .out(out)
);

// 4. Start testing
initial begin
    // Test input 000
    in = 3'b000;
    #10;

    // Test input 001
    in = 3'b001;
    #10;

    // Test input 010
    in = 3'b010;
    #10;

    // Test input 011
    in = 3'b011;
    #10;

    // Test input 100
    in = 3'b100;
    #10;

    // Test input 101
    in = 3'b101;
    #10;

    // Test input 110
    in = 3'b110;
    #10;

    // Test input 111
    in = 3'b111;
    #10;

    // End test
    $finish;
end
```

```
endmodule
```

Shift Registers

SISO (Serial In-Serial Out)

```
module SISO(
	input clk,
	input reset_n,
	input din,
	output dout
);

reg [3:0] shift;

always @ (posedge clk or posedge reset_n)
begin
	if(reset_n)
		q <= 4'b0000;
	else
		q[0] <= q[1];
		q[1] <= q[2];
		q[2] <= q[3];
		q[3] <= din;
end

assign dout = q[3];

endmodule
```

SIPO (Serial In-Parallel Out)

```
module SIPO(
        input clk,
        input reset_n,
        input din,
        output reg [3:0] dout // Defining dout as reg, since it will be used inside the always block.
);

// Since its a sequential circuit element, it will be defined in an always block
always @ (posedge clk or posedge reset_n)
begin
        if (reset_n) // if reset is asserted
                dout <= 4'b0000;
        else
                dout[3] <= dout[2];
                dout[2] <= dout[1];
                dout[1] <= dout[0];
                dout[0] <= din;
end

endmodule
```

The **difference** between the design source codes of SISO and SIPO are the following three (3):

- A seperate reg is defined in SISO, to act as the internal 4-bit shift register.
- The movement of the bits in SISO and SIPO are different.
    - The output in the case of SISO is assigned as q[3].
    - The output in the case of SIPO is assigned as q[0].
- There is no assign statement in the SIPO design source code.
  There is an assign statement in the SISO design source code.