Read-me

Akshay Joshi

akjoshi@ucsd.edu

Part – 1: Linear Feedback Shift Register (LFSR)

1. Code
    a. System Verilog Design Code

```
//RTL Model for Linear Feedback Shift Register
module lfsr
#(parameter N = 4) // Number of bits for LFSR
(
 input logic clk, reset, load_seed,
 input logic[N-1:0] seed_data,
 output logic lfsr_done,
 output logic[N-1:0] lfsr_data
);

//student to add implementation for LFSR code
logic[N-1:0] temp;
logic a;

always @(posedge clk or negedge reset)
begin
        if(reset == 0)
                lfsr_data <= 0;
        else if(load_seed == 1)
                lfsr_data <= seed_data;
        else
                lfsr_data <= temp;
```

```
end

always_comb
begin
        temp = lfsr_data;
        case(N)
                2 :     a = temp[N-1] ^ temp[N-2];
                3 :     a = temp[N-1] ^ temp[N-2];
                4 :     a = temp[N-1] ^ temp[N-2];
                5 :     a = temp[N-1] ^ temp[N-3];
                6 :     a = temp[N-1] ^ temp[N-2];
                7 :     a = temp[N-1] ^ temp[N-2];
                8 :     a = temp[N-1] ^ temp[N-3] ^ temp[N-4] ^ temp[N-5];
        endcase
        temp = temp << 1;
        temp[0] = a;
        if(lfsr_data == seed_data)
                lfsr_done = 1;
        else
                lfsr_done = 0;
end

endmodule: lfsr
```

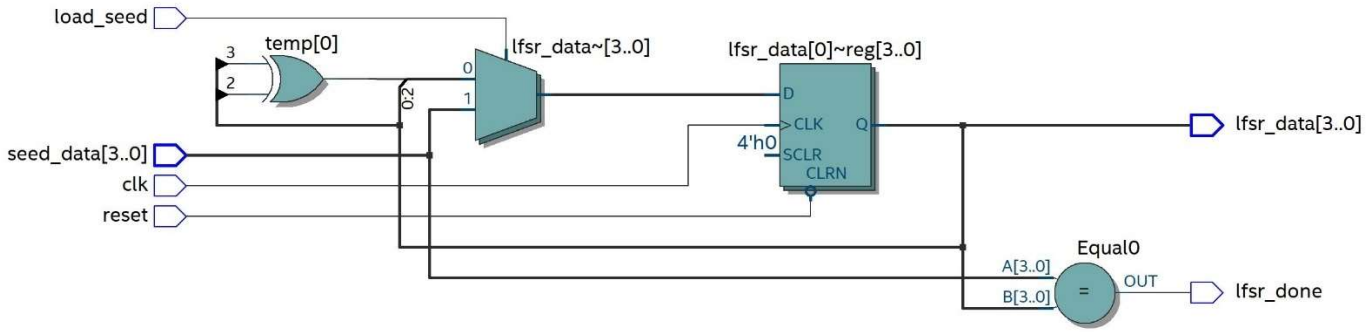
b. Testbench code

```
`timescale 1ns/1ns
//LFSR Testbench Code
module lfsr_testbench;
 parameter N = 4;
 logic clock;
```

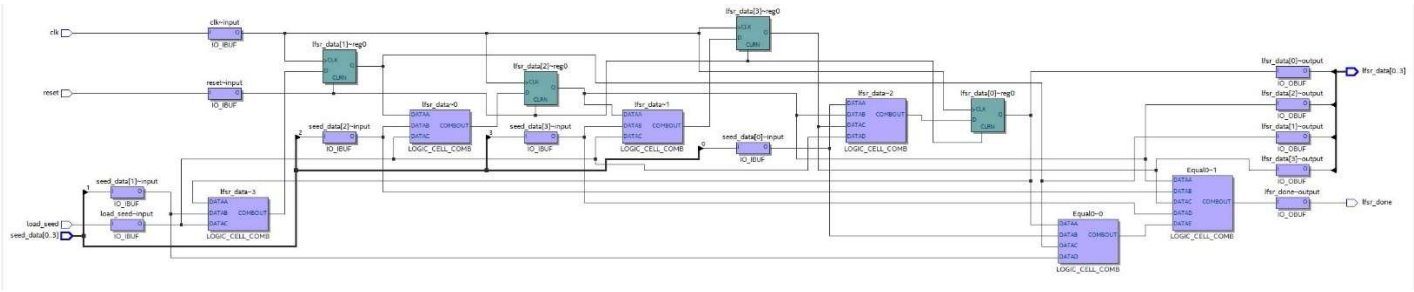
```
logic [N-1:0] lfsr_data, seed_data;
logic lfsr_done;
logic reset, load_seed;

lfsr #(.N(N)) design_inst(
 .clk(clock),
 .reset(reset),
 .load_seed(load_seed),
 .seed_data(seed_data),
 .lfsr_data(lfsr_data),
 .lfsr_done(lfsr_done)
);

initial begin
 // Initialize Inputs
 reset = 0;
 load_seed = 0;
 clock = 0;
 seed_data = 4'b0000;

 // Wait 10 ns for global reset to finish and start counter
 #10;
 reset = 1;

 #10;
 load_seed = 1;
 seed_data = 4'b1111;

 #20;
 load_seed = 0;
```

```
 #350;


 // terminate simulation
$finish();
end


// Clock generator logic
always@(clock) begin
  #10ns clock <= !clock;
end


// Print input and output signals
initial begin
 $monitor(" time=%0t,  reset=%b  clk=%b  load_seed=%b  count=%d", $time, reset, clock, load_seed, lfsr_data);
end


endmodule
```

2. RTL schematic



3. Post map schematic



4. Resource Utilization

➔ The resource utilization for the designed four-bit Linear Feedback Shift Register (LFSR) is given on the next page.

➔ We observe that 6 combinational ALUTs are used to store the needed data.

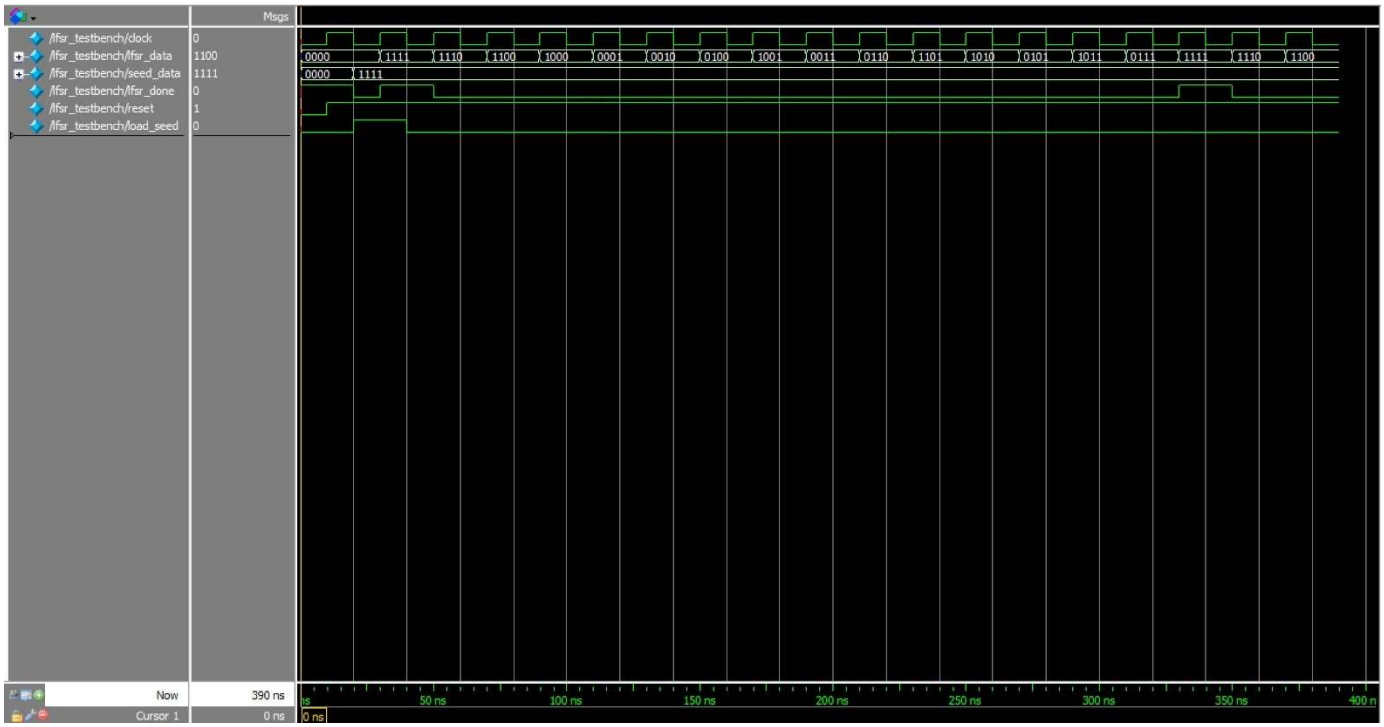
➔ We also see that 4 dedicated logic registers are also used to store the temporary data on which operations are performed.

➔ One 5-input function, two 4-input function and three <=3-input functions are used.

➔ Total I/O count is 12, 4 each from seed_data and lfsr_data & one each from clk, load_seed, reset and lfsr_done.

	Resource	Usage
1	Estimated ALUTs Used	6
1	-- Combinational ALUTs	6
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	4
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	6
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	1
4	-- 4 input functions	2
5	-- <=3 input functions	3
8		
9	Combinational ALUTs by mode	
1	-- normal mode	6
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	6
12		
13	Total registers	4
1	-- Dedicated logic registers	4
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	12
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	lfsr_data[2]~reg0
21	Maximum fan-out	4
22	Total fan-out	51
23	Average fan-out	1.50

5. Simulation Result for N=4



- ➔ The simulation waveform for a 4-bit LFSR is shown above.
- ➔ A 4-bit LFSR is a design which generates a four-bit random number by feeding the XOR of some of the previously generated bits to the LSB position and shifting the rest of the bits to the left by one position. The random number generated from the design exhibits pseudo-random behavior, that is, the numbers are generated based on the initial value and for each initial seed value there is a deterministic pattern of counting.
- ➔ The random number sequence repeats after a given number of iterations (clock cycles). For instance, the sequence repeats after 15 counts for a four-bit LFSR.
- ➔ The reset to the design is asynchronous and *seed_data* is passed to the output when *load_seed* signal in '1'.
- ➔ The LSB for a 4-bit LFSR is obtain through XOR of 2 MSBs.
- ➔ Let us try to analyze from the waveform and see if our design exhibits the intended behavior.
    - The *reset* and *load_seed* becomes 1 at around 10ns so the design takes the input from the *seed_data* at the next clock edge (30ns) and relays it to the output.
    - The LFSR takes the value 1111 at 30ns. The XOR the two MSB would be- 1 ^ 1 = 0. Shifting the bits left by one position and putting the XOR value at the LSB, we get the next value to be 1110
    - Similarly calculating the subsequent values-
        - Previous value = 1110; XOR of MSBs: 1^1 = 0; next value = 1100
        - Previous value = 1100; XOR of MSBs: 1^1 = 0; next value = 1000
        - Previous value = 1000; XOR of MSBs: 1^0 = 1; next value = 0001

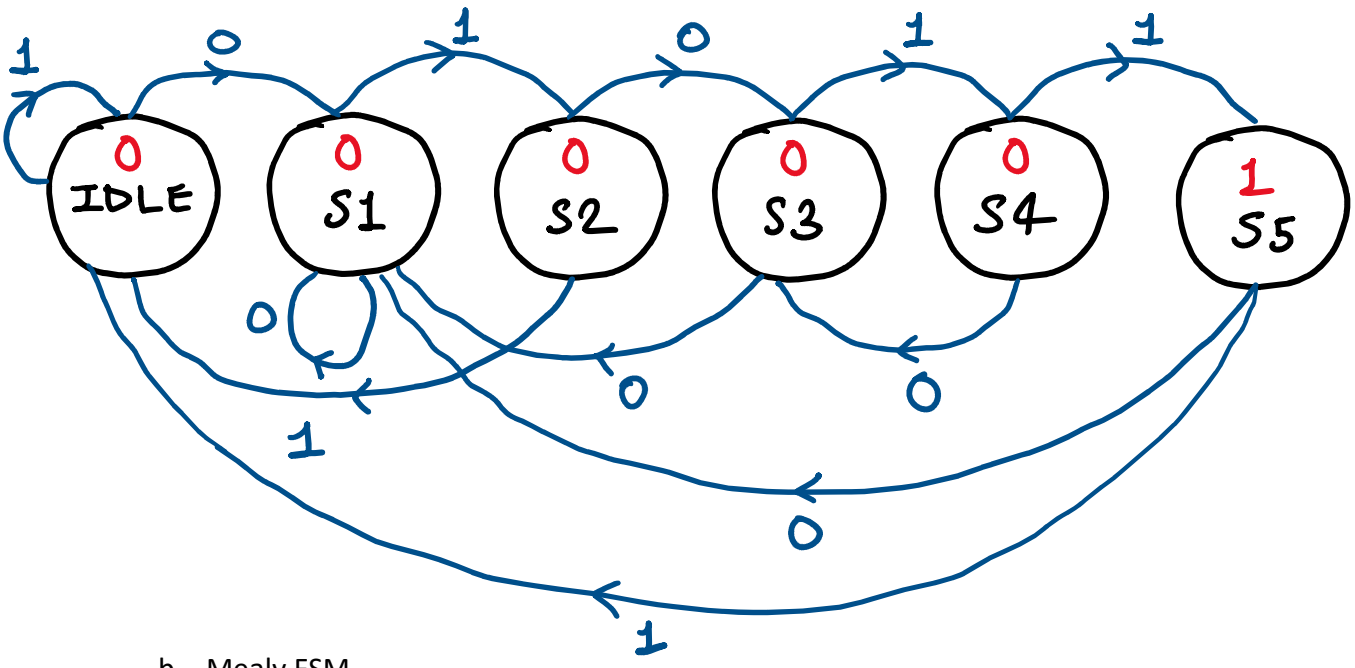
        - Previous value = 0001; XOR of MSBs: 0^0 = 0; next value = 0010
    - This goes on unless a *reset* or *load_seed* pulse is encountered.
    - Kindly observe that the pattern starts to repeat at 330ns (after 15 cycles) and we get a value of one at *lfsr_done* output.

➔ Hence, we observe that the design exhibits the intended behavior as interpreted theoretically. This can also be observed from the transcript below.

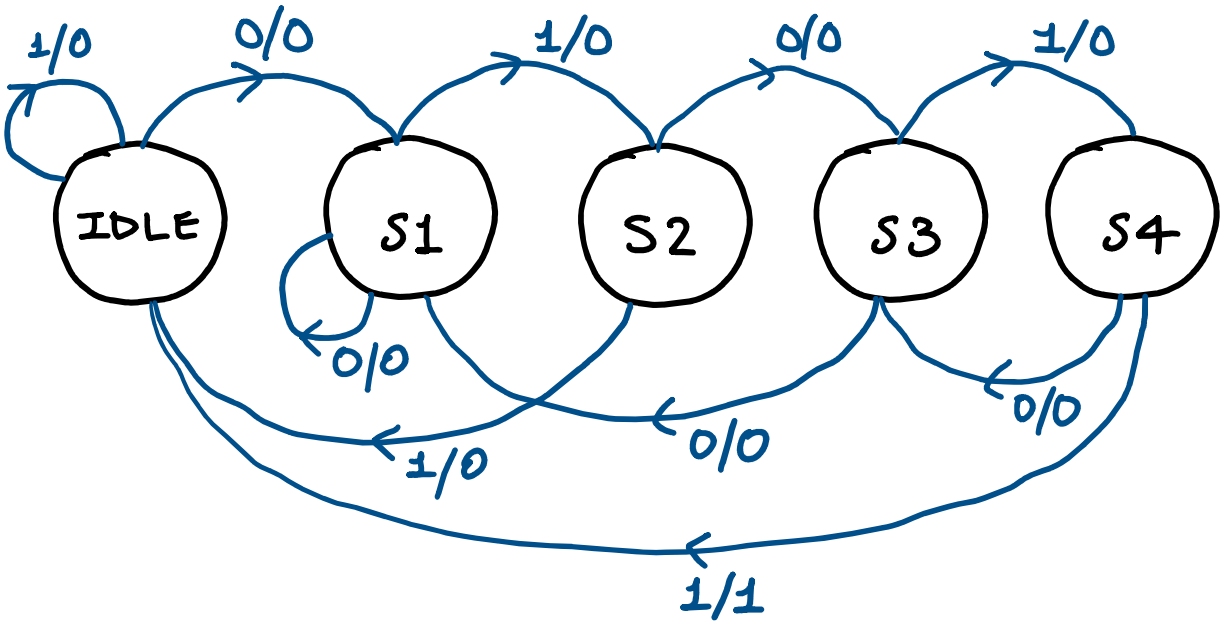
```
VSIM 6> run -all
#  time=0,   reset=0  clk=0  load_seed=0  count= 0
#  time=10,  reset=1  clk=1  load_seed=0  count= 0
#  time=20,  reset=1  clk=0  load_seed=1  count= 0
#  time=30,  reset=1  clk=1  load_seed=1  count=15
#  time=40,  reset=1  clk=0  load_seed=0  count=15
#  time=50,  reset=1  clk=1  load_seed=0  count=14
#  time=60,  reset=1  clk=0  load_seed=0  count=14
#  time=70,  reset=1  clk=1  load_seed=0  count=12
#  time=80,  reset=1  clk=0  load_seed=0  count=12
#  time=90,  reset=1  clk=1  load_seed=0  count= 8
#  time=100,  reset=1  clk=0  load_seed=0  count= 8
#  time=110,  reset=1  clk=1  load_seed=0  count= 1
#  time=120,  reset=1  clk=0  load_seed=0  count= 1
#  time=130,  reset=1  clk=1  load_seed=0  count= 2
#  time=140,  reset=1  clk=0  load_seed=0  count= 2
#  time=150,  reset=1  clk=1  load_seed=0  count= 4
#  time=160,  reset=1  clk=0  load_seed=0  count= 4
#  time=170,  reset=1  clk=1  load_seed=0  count= 9
#  time=180,  reset=1  clk=0  load_seed=0  count= 9
#  time=190,  reset=1  clk=1  load_seed=0  count= 3
#  time=200,  reset=1  clk=0  load_seed=0  count= 3
#  time=210,  reset=1  clk=1  load_seed=0  count= 6
#  time=220,  reset=1  clk=0  load_seed=0  count= 6
#  time=230,  reset=1  clk=1  load_seed=0  count=13
#  time=240,  reset=1  clk=0  load_seed=0  count=13
#  time=250,  reset=1  clk=1  load_seed=0  count=10
#  time=260,  reset=1  clk=0  load_seed=0  count=10
#  time=270,  reset=1  clk=1  load_seed=0  count= 5
#  time=280,  reset=1  clk=0  load_seed=0  count= 5
#  time=290,  reset=1  clk=1  load_seed=0  count=11
#  time=300,  reset=1  clk=0  load_seed=0  count=11
#  time=310,  reset=1  clk=1  load_seed=0  count= 7
#  time=320,  reset=1  clk=0  load_seed=0  count= 7
#  time=330,  reset=1  clk=1  load_seed=0  count=15
#  time=340,  reset=1  clk=0  load_seed=0  count=15
#  time=350,  reset=1  clk=1  load_seed=0  count=14
#  time=360,  reset=1  clk=0  load_seed=0  count=14
#  time=370,  reset=1  clk=1  load_seed=0  count=12
#  time=380,  reset=1  clk=0  load_seed=0  count=12
# ** Note: $finish    : C:/Users/ITSloaner/Downloads/ECE111/Mini_project3/MP3/lfsr/lfsr_testbench.sv(40)
#    Time: 390 ns  Iteration: 0  Instance: /lfsr_testbench
```

Part-2: Moore and Mealy FSM for Sequence Detector

1. State Transition Diagram
    a. Moore FSM



    b. Mealy FSM



➔ The state transition diagrams for Moore and Mealy FSM are given above.

- ➔ The Moore FSM is comprised of 6 states. The output of the design becomes 1 when it takes the state S5. The IDLE state is the default state. Every subsequent state corresponds to an input bit according to the desired pattern – “01011”.
- ➔ The Mealy FSM comprises of 5 states. The default is the IDLE state. Please note that the FSM output is 1 only when it is in S4 state and an input value of 1 is given; the system transitions to IDLE state after this. Every other state transition occurs with an output of 0 and is according to the sequence in our intended pattern.

2. Code
    a. System Verilog Moore FSM code

```
// Moore Sequence Detector for 01011
module seq_det_moore(
 input logic clk,
 input logic rst,
 input logic in,
 output logic out
);

// Students to add code for Moore Sequence Detector

//Parameters to define FSM state encodings
localparam[5:0] IDLE=6'b000001,
                S1=6'b000010,
                S2=6'b000100,
                S3=6'b001000,
                S4=6'b010000,
                S5=6'b100000;

//Current state and next state variables
logic[5:0] present_state, next_state;

//Sequential logic for present state
always_ff@(posedge clk)
```

```
begin
	if(!rst)
		present_state <= IDLE;
	else
		present_state <= next_state;
end

//Combinational logic for next state and output
always@(present_state,in)
begin
	case(present_state)
		IDLE: begin
			if(in==0) next_state = S1;
			else next_state = IDLE;
		end
		S1: begin
			if(in==1) next_state = S2;
			else next_state = S1;
		end
		S2: begin
			if(in==0) next_state = S3;
			else next_state = IDLE;
		end
		S3: begin
			if(in==1) next_state = S4;
			else next_state = S1;
		end
		S4: begin
			if(in==1) next_state = S5;
			else next_state = S3;
		end
```

```
                S5: begin
                        if(in==0) next_state = S1;
                        else next_state = IDLE;
                end
                default: next_state = IDLE;
        endcase
end

always@(present_state)
begin
        case(present_state)
                S5: out = 1;
                default: out = 0;
        endcase
end

endmodule: seq_det_moore
```

b. System Verilog Mealy FSM Code

```
// Mealy Sequence Detector for 01011
module seq_det_mealy(
 input logic clk,
 input logic rst,
 input logic in,
 output logic out
);

// Students to add code for Mealy Sequence Detector

//Parameters to define FSM state codings
```

```
localparam[4:0] IDLE=5'b00001,
                                        S1=5'b00010,
                                        S2=5'b00100,
                                        S3=5'b01000,
                                        S4=5'b10000;

//Current state and next state variables
logic[4:0] present_state, next_state;

//Sequential Logic for present state
always_ff@(posedge clk)
begin
        if(!rst)
                present_state <= IDLE;
        else
                present_state <= next_state;
end

//Combinational logic for next state and output
always@(present_state, in)
begin
        case(present_state)
                IDLE: begin
                        if(in==0) begin
                                next_state = S1;
                                out = 0;
                        end
                        else begin
                                next_state = IDLE;
                                out = 0;
                        end
```

```
end
S1: begin
        if(in==1) begin
                next_state = S2;
                out = 0;
        end
        else begin
                next_state = S1;
                out = 0;
        end
end
S2: begin
        if(in==0) begin
                next_state = S3;
                out = 0;
        end
        else begin
                next_state = IDLE;
                out = 0;
        end
end
S3: begin
        if(in==1) begin
                next_state = S4;
                out = 0;
        end
        else begin
                next_state = S1;
                out = 0;
        end
end
```

```
                S4: begin
                        if(in==1) begin
                                next_state = IDLE;
                                out = 1;
                        end
                        else begin
                                next_state = S3;
                                out = 0;
                        end
                end
                default: begin
                        next_state = IDLE;
                        out = 0;
                end
        endcase
end

endmodule: seq_det_mealy
```

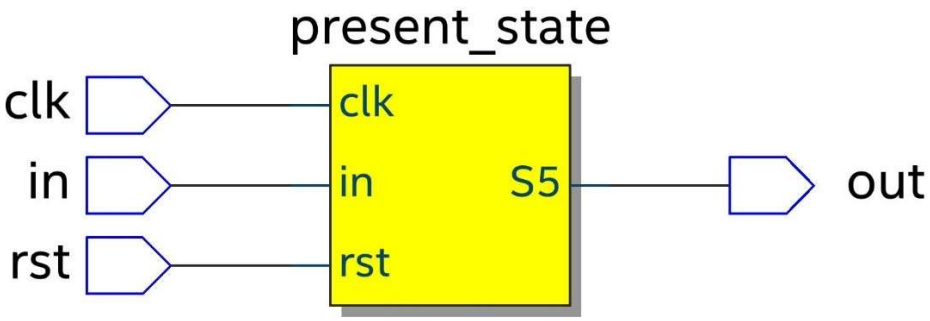
c. Testbench Code

```
`timescale 1ns/1ns
//Sequence Detector Testbench

module seq_det_tb;
 parameter NUM_BITS = 6;

 logic clk, reset;
 logic out_moore,out_mealy,load_seed,lsfr_done;
 logic[NUM_BITS-1:0] seed, data;

  lfsr #(.N(NUM_BITS)) lfsr1(
```

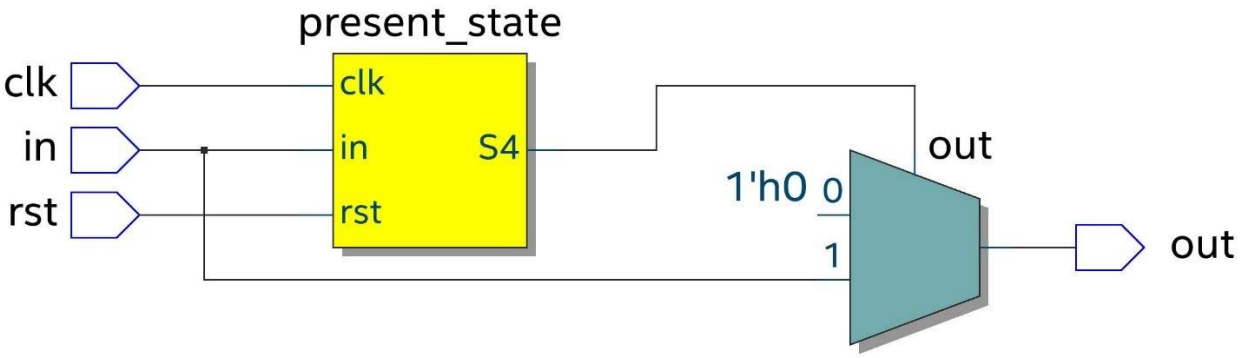
```
  .clk(clk),
  .reset(reset),
  .load_seed(load_seed),
  .seed_data(seed),
  .lfsr_data(data),
  .lfsr_done(lfsr_done)
 );

        seq_det_moore moore(
        .clk(clk),
        .rst(reset),
        .in(data[0]),
        .out(out_moore)
        );
        seq_det_mealy mealy(
        .clk(clk),
        .rst(reset),
        .in(data[0]),
        .out(out_mealy)
        );

initial begin
// Initialize Inputs
reset = 0;
// Wait 10 ns for global reset to finish and start counter
#10;
reset = 1;
load_seed = 1;
seed = 6'b000111;
#10;
load_seed = 0;
```

```
$display("Starting");
#5000ns;
$display("Completed");
// terminate simulation
$finish();
end

initial begin
	clk =0;
	forever #5 clk=!clk;
end

always_comb begin
	if(data[4:0] == 5'b01011) begin
		$display(" time=%0t, data=%6b",$time, data);
	end
end

wire mealy_right, moore_right;
assign mealy_right = (data[4:0] == 5'b01011) ? out_mealy : 1;
assign moore_right = (data[5:1] == 5'b01011) ? out_moore : 1;

always_ff @(posedge clk) begin
	if(!mealy_right) $display("mealy failed");
	if(!moore_right) $display("moore failed");
end


endmodule: seq_det_tb
```

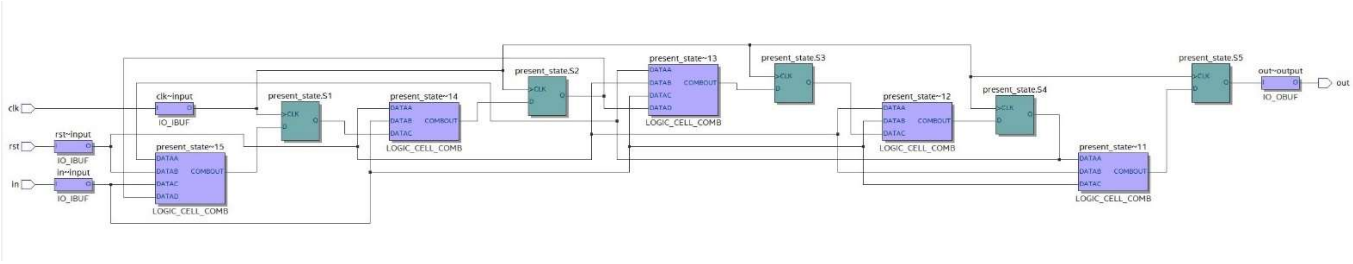
3. RTL schematic
    a. Moore FSM



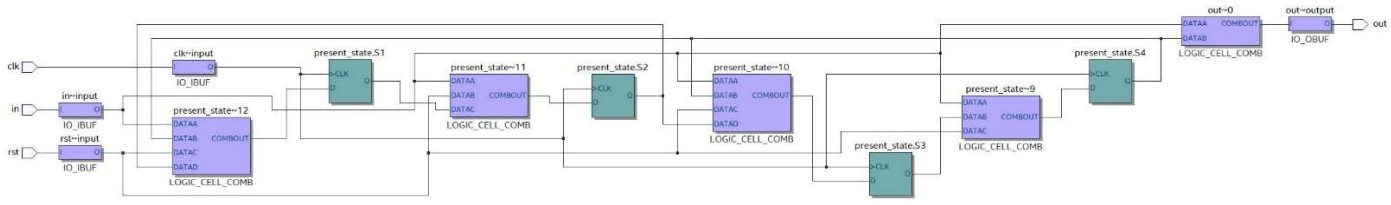
    b. Mealy FSM



4. Post-map schematic
    a. Moore FSM



    b. Mealy FSM



5. Resource Usage
    a. Moore FSM

	Resource	Usage
1	Estimated ALUTs Used	5
1	-- Combinational ALUTs	5
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	5
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	5
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	0
4	-- 4 input functions	2
5	-- <=3 input functions	3
8		
9	Combinational ALUTs by mode	
1	-- normal mode	5
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	5
12		
13	Total registers	5
1	-- Dedicated logic registers	5
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	4
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	rst~input
21	Maximum fan-out	5
22	Total fan-out	32
23	Average fan-out	1.78

b. Mealy FSM

	Resource	Usage
1	Estimated ALUTs Used	5
1	-- Combinational ALUTs	5
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	4
3		
4	Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	5
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	0
3	-- 5 input functions	0
4	-- 4 input functions	2
5	-- <=3 input functions	3
8		
9	Combinational ALUTs by mode	
1	-- normal mode	5
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	5
12		
13	Total registers	4
1	-- Dedicated logic registers	4
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	4
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	in~input
21	Maximum fan-out	5
22	Total fan-out	29
23	Average fan-out	1.71

➔ Comparison

- The number of ALUTs used are 5 in both the design which indicates that both the design has same number of inputs and outputs.
- The number of I/O are also same (4 each), clk, in, rst and out.
- Total combinational functions used are also same, that is, 5 each for the combinational block operations.

- The only difference in the resource usage comes from the dedicated logic registers, wherein the Moore FSM uses 5 registers whereas the Mealy FSM uses 4. The extra register accounts for the extra state in the Moore FSM. The Moore FSM has a total of 6 states whereas the Mealy FSM comprises of 5 states, hence an extra register would be required in case of Moore to store the data corresponding to the one extra state. Hence, Moore FSM has a higher overall resource usage value.

6. Simulation Transcript

```
ModelSim> vsim work.seq_det_tb
# vsim work.seq_det_tb
# Start time: 13:58:00 on Nov 15,2021
# Loading sv_std.std
# Loading work.seq_det_tb
# Loading work.lfsr
# Loading work.seq_det_moore
# Loading work.seq_det_mealy
add wave sim:/seq_det_tb/*
VSIM 8> run -all
# Starting
#  time=105, data=001011
#  time=285, data=101011
#  time=735, data=001011
#  time=915, data=101011
#  time=1365, data=001011
#  time=1545, data=101011
#  time=1995, data=001011
#  time=2175, data=101011
#  time=2625, data=001011
#  time=2805, data=101011
#  time=3255, data=001011
#  time=3435, data=101011
#  time=3885, data=001011
#  time=4065, data=101011
#  time=4515, data=001011
#  time=4695, data=101011
# Completed
# ** Note: $finish    : C:/Users/ITSloaner/Downloads/ECE111/Mini_project3/MP3/FSM_final/seq_det_tb.sv(53)
#    Time: 5020 ns  Iteration: 0  Instance: /seq_det_tb
# 1
# Break in Module seq_det_tb at C:/Users/ITSloaner/Downloads/ECE111/Mini_project3/MP3/FSM_final/seq_det_tb.sv line 53
```

7. Simulation Waveforms
    a. Zoomed-in

➔ The zoomed in simulation waveform is given below.

➔ One hot encoding is used to assign values to each state in both FSM.

➔ The input bit (*data[0]*), data of LFSR (*data*) and FSM outputs (*out_moore* & *out_mealy*) are clearly depicted in the simulation waveform.

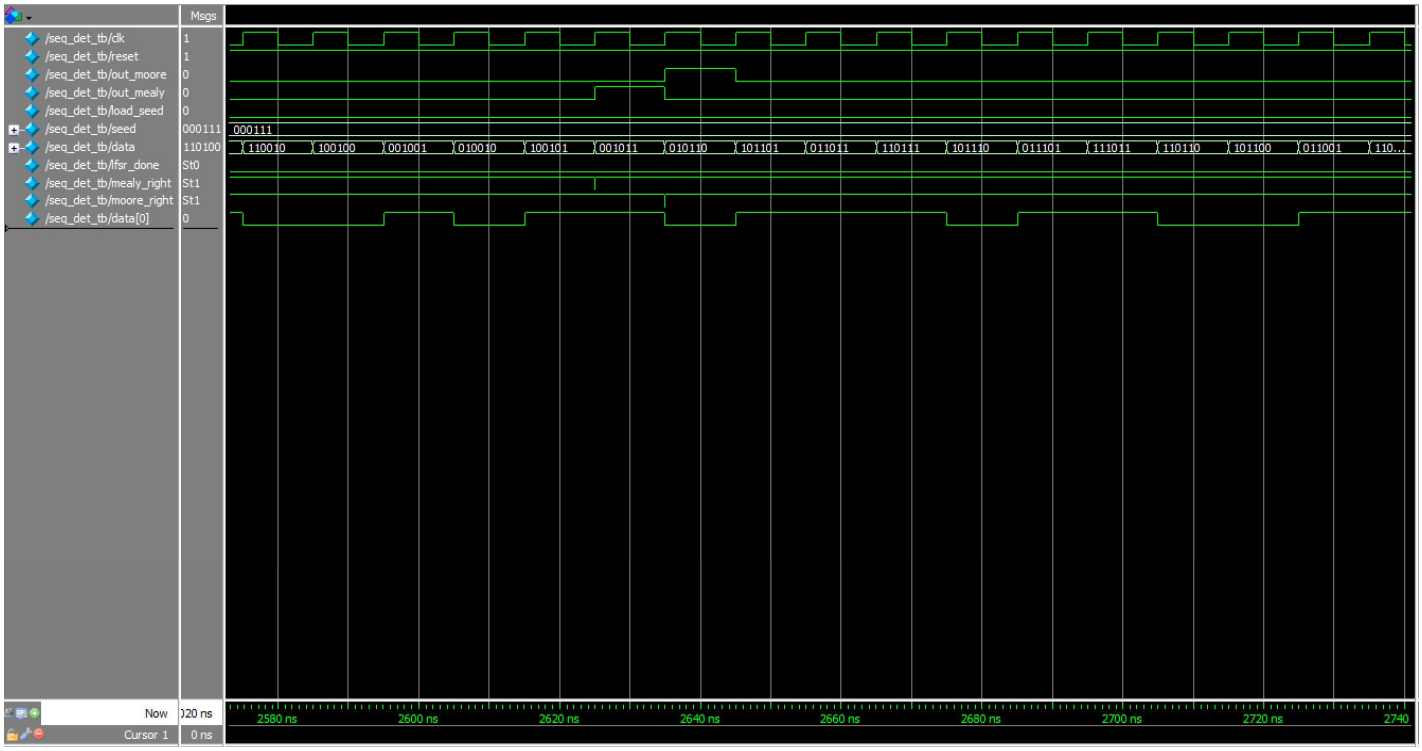
➔ The intended sequence to be detected is “01011”. We can clearly observe that from 2625ns to 2645ns, this required sequence is achieved and the value of output becomes 1 correspondingly, proving the correctness of the sequence detector.

➔ Observe from the waveform that the sequence gets detected in the Mealy FSM one clock pulse before the Moore FSM. Let us understand the reason for this.

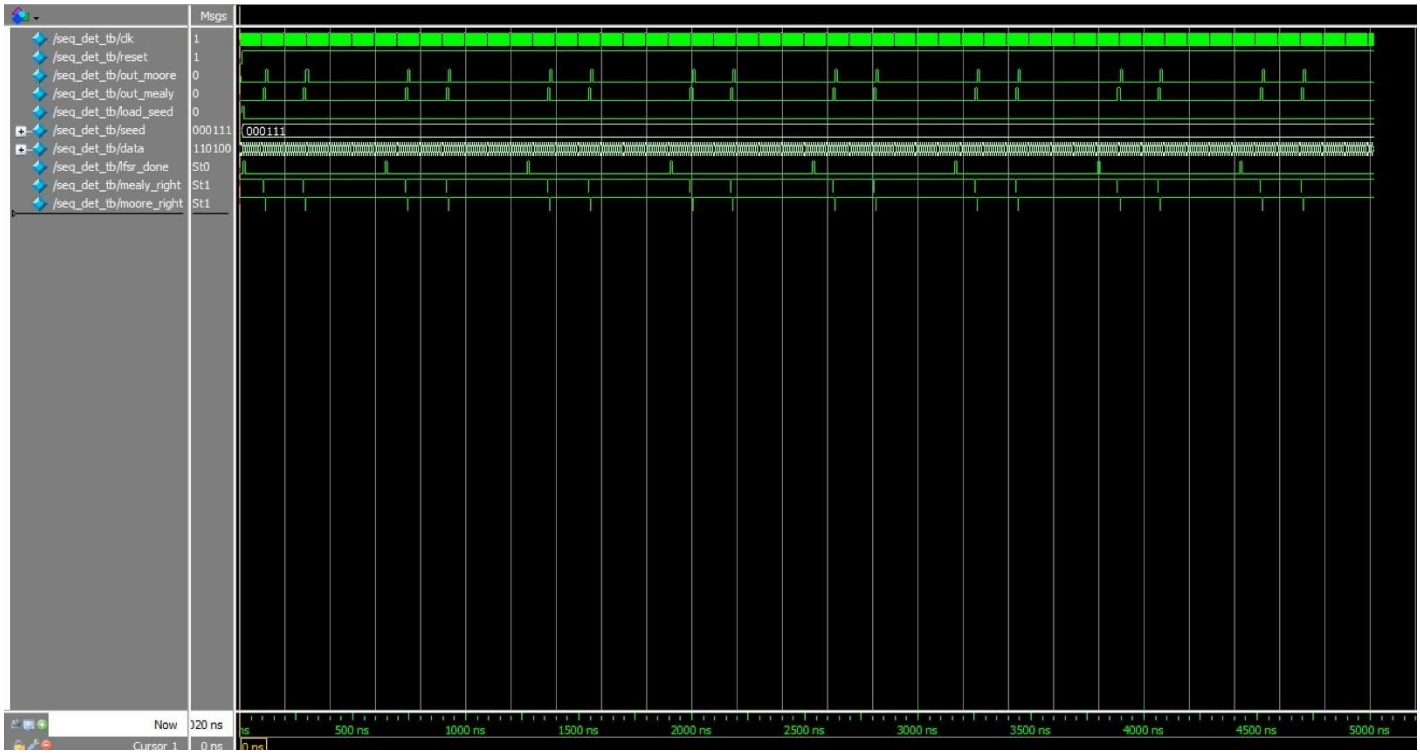
- For the case of Moore FSM, the state transitions are bound by the clock pulse. To determine the corresponding next state in accordance with our intended sequence, we have written a case statement. The statement takes the *present_state* and *in* as the input

variables and gives out the next state accordingly. The system transitions onto the subsequent state only at next positive clock edge.

- Also, the output becomes one only when the Moore FSM is in state S5 (100000).
- Therefore, when the final correct bit of our sequence '1' arrives, the design decides that the next state is going to be S5 (through case statement) but could not transition to that state until the next positive clock edge. On the next positive clock, the FSM transitions to state S5, the output takes the value 1, but at the same time new input data value also arrives (through which the system can take decision about the next state).
- For the case of Mealy FSM too, the state transitions are bound by the clock pulse. To determine the corresponding next state in accordance with our intended sequence, we have written a case statement. The statement takes the *present_state* and *in* as the input variables and gives out the next state and output accordingly. The system transitions to the next state only on the subsequent positive clock edge but the output value is given out immediately as it is part of the combinational logic and does not depend on clock edge.
- Hence, when the Mealy FSM is in state S4 (10000) and it receives an input of '1', the output immediately takes the value '1' without waiting for the clock edge to arrive. The next state is chosen accordingly (IDLE) but the state transition happens only on the subsequent positive clock edge.
- Therefore, the Moore FSM lags Mealy FSM by one clock period in giving the corresponding output value.



➔ The zoomed-out version of the simulation schematic is shown below.



➔ The sequence flushed in by the LFSR keeps repeating after a given interval of time (63 cycles). This can be inferred from the recurring spike in the *lfsr_done* signal.

➔ Correspondingly, our intended sequence “01011” also keeps repeating after 63 cycles. This is evident from the recurring spike in the *out_moore* and *out_mealy* pulse.