Class:	**CPE300L Digital Systems Architecture and Design 1001**		Semester:	**Fall 2024**
Points		Document author:	**Narek Kalikian**	
		Author's email:	**kalikn1@unlv.nevada.edu**	
		Document topic:	**Postlab 3**	
Instructor's comments:				

1. Introduction / Theory of Operation

The main two types of finite state machines (FSMs) are Moore FSMs and Mealy FSMs. The primary difference between these two types of FSMs is that Moore machines have outputs that are solely dependent on the current state of the FSM, Mealy machines have outputs that are dependent both on the current state of the FSM and the current input(s). Moore FSMs generally require more states but are overall easier to design than Mealy FSMs. Moore FSMs generate states and output values synchronously whereas Mealy FSMs handle these processes asynchronously.

When designing an FSM, the first thing you should start with is the state diagram. By carefully reading the requirements of the state machine and establishing each state, state transitions based on specific inputs, and outputs where applicable, you are able to create an accurate visual representation of how the machine should operate. This makes implementing the functionality of the FSM through Verilog, and eventually on hardware, much easier because you have something to go off of. The same way you wouldn't build hardware without first drawing necessary schematics, you shouldn't attempt the programming portion of the FSM design without first drawing out the state diagram.

2. Description of Experiments

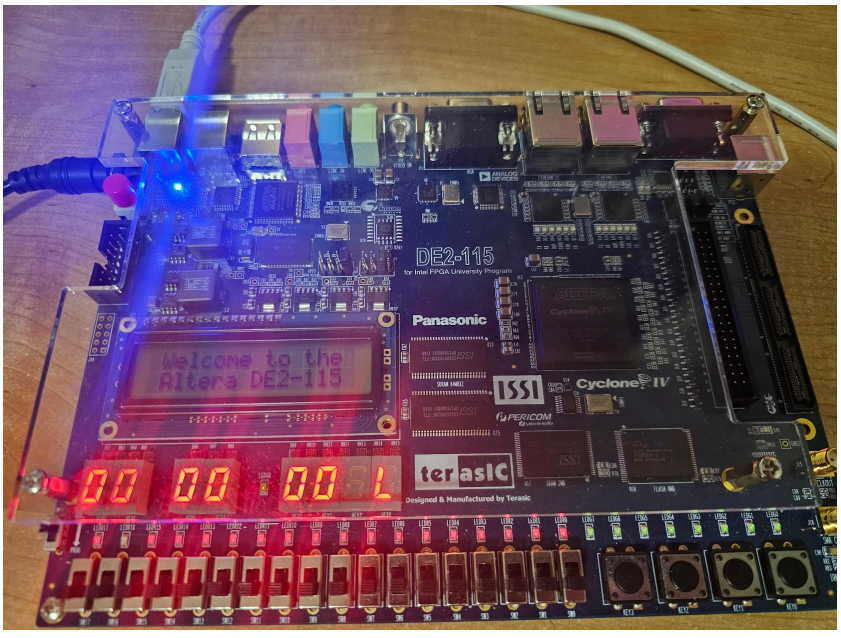
Experiment 1 - Verilog Code:

```
module digital_lock (input wire [5:0] in, input wire clk, input wire reset,
output reg [6:0] sevenseg, output reg [6:0] sevenseg7, output reg [6:0] sevenseg6,
output reg [6:0] sevenseg5, output reg [6:0] sevenseg4, output reg [6:0] sevenseg3,
output reg [6:0] sevenseg2, output reg unlock);

parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5, S6 = 6;

reg [2:0] current_state, next_state;

reg [24:0] clk_divider;
reg slow_clk;

always @ (posedge clk)
begin
clk_divider <= clk_divider + 1;
slow_clk <= clk_divider[24];
end

always @ (posedge slow_clk or posedge reset)
begin
if(reset)
current_state <= S0;
else current_state <= next_state;
end

always @ (*)
begin

unlock = 0;

case(current_state)
S0:
begin
if (in[5] == 1)
next_state = S1;
else
next_state = S0;
end
S1:
begin
if (in[4] == 1)
next_state = S2;
else
next_state = S0;
end
S2:
begin
if (in[3] == 0)
next_state = S3;
else
next_state = S0;
end
S3:
begin
if (in[2] == 1)
next_state = S4;
else
next_state = S0;
end
S4:
begin
if (in[1] == 1)
next_state = S5;
else
next_state = S0;
end
S5:
begin
if (in[0] == 0)
next_state = S6;
else
next_state = S0;
end
S6:
begin
unlock = 1;
if (reset)
next_state = S0;
else
next_state = S6;
end
default: next_state = S0;

endcase

case(in[5])
0: sevenseg7 <= 7'b1000000;
1: sevenseg7 <= 7'b1111001;

default: sevenseg7 <= 7'b1111111;

endcase

case(in[4])
0: sevenseg6 <= 7'b1000000;
1: sevenseg6 <= 7'b1111001;

default: sevenseg6 <= 7'b1111111;

endcase

case(in[3])
```

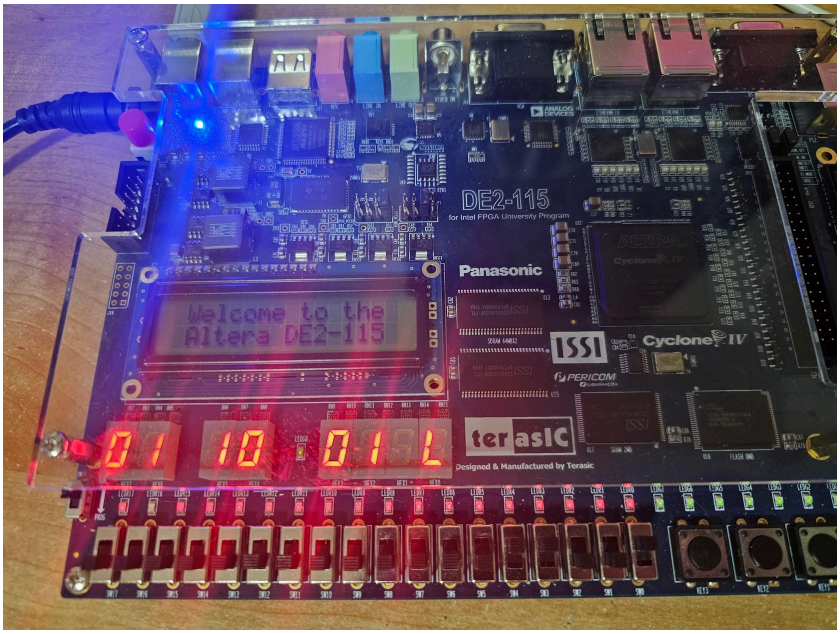
```
103     0: sevenseg5 <= 7'b1000000;
104     1: sevenseg5 <= 7'b1111001;
105
106     default: sevenseg5 <= 7'b1111111;
107
108     endcase
109
110     case(in[2])
111     0: sevenseg4 <= 7'b1000000;
112     1: sevenseg4 <= 7'b1111001;
113
114     default: sevenseg4 <= 7'b1111111;
115
116     endcase
117
118     case(in[1])
119     0: sevenseg3 <= 7'b1000000;
120     1: sevenseg3 <= 7'b1111001;
121
122     default: sevenseg3 <= 7'b1111111;
123
124     endcase
125
126     case(in[0])
127     0: sevenseg2 <= 7'b1000000;
128     1: sevenseg2 <= 7'b1111001;
129
130     default: sevenseg2 <= 7'b1111111;
131
132     endcase
133
134     case(unlock)
135     0: sevenseg <= 7'b1000111;
136
137     1: sevenseg <= 7'b1000001;
138
139     default: sevenseg <= 7'b1111111;
140
141     endcase
142     end
143     endmodule
```

Experiment 1 - DE2 Board Pictures:

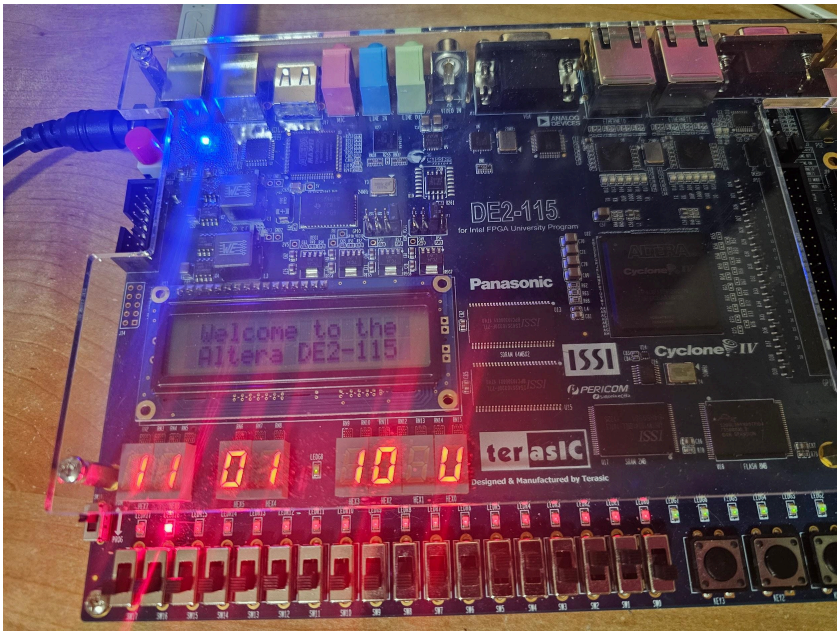
Start: Combination = 000000



Incorrect Combo: Combination = 011001



Correct Combo: Combination = 110110



Experiment 1 - RTL View:

Experiment 1 - Video: Attached to Canvas Submission

Experiment 1 - Video Explanation: First, the wrong combination is input as a test. The digital lock stays locked (L). Then, reset switch is asserted. Next, the correct combination is input and the digital lock unlocks (U). The input combination is changed from the correct combination to show that the digital lock stays unlocked. Finally, the reset switch is asserted again, which resets the digital lock and returns to being locked (L).

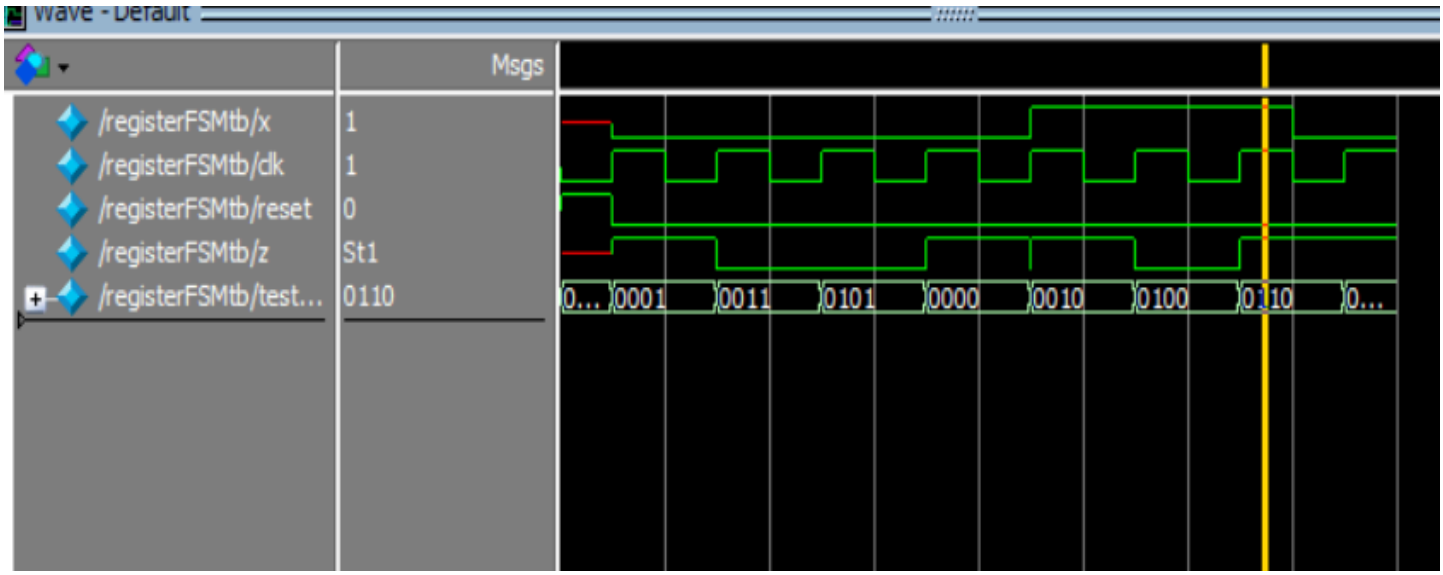
Experiment 2 - Verilog Code:

```
module registerFSM (input x, input clk, input reset, output reg z, output reg [3:0] test);

parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5, S6 = 6;

reg [2:0] current_state, next_state;

always @ (posedge clk or posedge reset)
if(reset)
current_state <= S0;
else current_state <= next_state;

always @(*) begin
    case (current_state)
        S0:
  begin
            test = 4'd0;
            if (x == 0) next_state = S1; else next_state = S2;
            z = (x == 0) ? 1 : 0;
        end
        S1:
  begin
            test = 4'd1;
            if (x == 0) next_state = S3; else next_state = S4;
            z = (x == 0) ? 1 : 0;
        end
        S2:
  begin
            test = 4'd2;
            if (x == 0) next_state = S4; else next_state = S4;
            z = (x == 0) ? 0 : 1;
        end
        S3:
  begin
           test = 4'd3;
            if (x == 0) next_state = S5; else next_state = S5;
            z = (x == 0) ? 0 : 1;
        end
        S4:
  begin
             test = 4'd4;
            if (x == 0) next_state = S5; else next_state = S6;
            z = (x == 0) ? 1 : 0;
        end
            S5:
    begin
               test = 4'd5;
                if (x == 0) next_state = S0; else next_state = S0;
                z = (x == 0) ? 0 : 1;
            end
            S6:
    begin
                test = 4'd6;
                if (x == 0)
                next_state = S0;
                z = 1;
            end
      endcase
end

endmodule
```

Experiment 2 - Testbench Code:

```
module registerFSMtb;

  reg x;
  reg clk;
  reg reset;
  wire z;
  wire [3:0] teststate;

  registerFSM uut (
    .x(x),
    .clk(clk),
    .reset(reset),
    .z(z),
    .teststate(teststate)
);

initial begin
  // S0
  reset = 1;
  clk = 0;
  #100;
  // S1
  reset = 0;
  clk = 1;
  x = 0;
  #100;
  clk = 0;
  #100;
  // S3
  clk = 1;
  #100;
  clk = 0;
  #100;
  // S5
  clk = 1;
  #100;
  clk = 0;
  #100;
  // Back to S0
  clk = 1;
  #100;

  // S0
  clk = 0;
  #100;
  // S2
  clk = 1;
  x = 1;
  #100;
 clk = 0;
 #100;
 // S4
 clk = 1;
 #100;
 clk = 0;
 #100;
 // S6
 clk = 1;
 #100;
clk = 0;
x = 0;
#100;
// Back to S0
clk = 1;
#100;

end
initial begin
    $monitor("At time %t: clk=%b, reset=%b, x=%b, z=%b, current state=%d", $time, clk, reset, x, z, teststate);
end
endmodule
```

Experiment 2 - Waveform Simulation:



Experiment 2 - Simulation Console:

```
# Loading work.registerFSMtb(fast)
VSIM 42> run -all
# At time                    0: clk=0, reset=1, x=x, z=x, current state= 0
# At time                  100: clk=1, reset=0, x=0, z=1, current state= 1
# At time                  200: clk=0, reset=0, x=0, z=1, current state= 1
# At time                  300: clk=1, reset=0, x=0, z=0, current state= 3
# At time                  400: clk=0, reset=0, x=0, z=0, current state= 3
# At time                  500: clk=1, reset=0, x=0, z=0, current state= 5
# At time                  600: clk=0, reset=0, x=0, z=0, current state= 5
# At time                  700: clk=1, reset=0, x=0, z=1, current state= 0
# At time                  800: clk=0, reset=0, x=0, z=1, current state= 0
# At time                  900: clk=1, reset=0, x=1, z=1, current state= 2
# At time                 1000: clk=0, reset=0, x=1, z=1, current state= 2
# At time                 1100: clk=1, reset=0, x=1, z=0, current state= 4
# At time                 1200: clk=0, reset=0, x=1, z=0, current state= 4
# At time                 1300: clk=1, reset=0, x=1, z=1, current state= 6
# At time                 1400: clk=0, reset=0, x=0, z=1, current state= 6
# At time                 1500: clk=1, reset=0, x=0, z=1, current state= 0
```

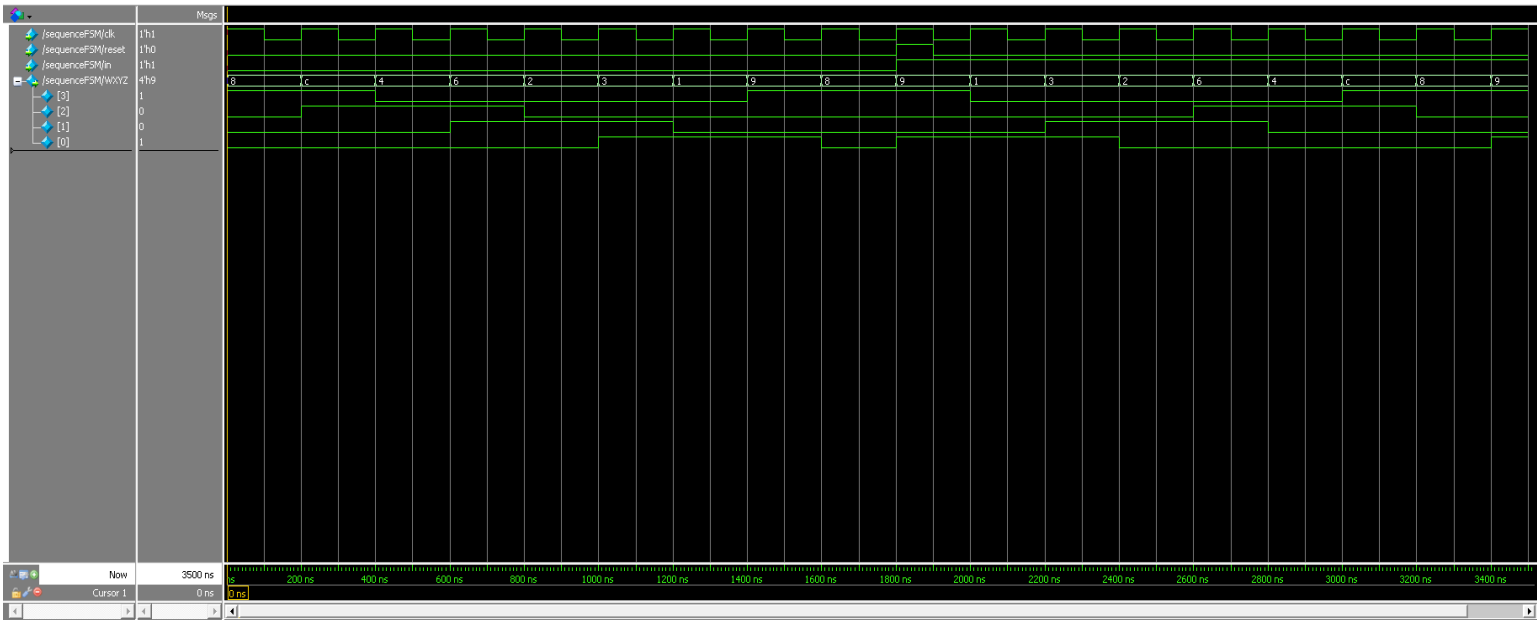
Experiment 3 - Verilog Code:

```
module sequenceFSM(input clk, input reset, input in, output reg [3:0] WXYZ, output reg [6:0] sevenseg0,
output reg [6:0] sevenseg1, output reg [6:0] sevenseg2, output reg [6:0] sevenseg3, output reg [6:0] sevenseg7);

 parameter S0_0 = 4'b1000, S1_0 = 4'b1100, S2_0 = 4'b0100, S3_0 = 4'b0110,
           S4_0 = 4'b0010, S5_0 = 4'b0011, S6_0 = 4'b0001, S7_0 = 4'b1001;

 parameter S0_1 = 4'b1001, S1_1 = 4'b0001, S2_1 = 4'b0011, S3_1 = 4'b0010,
           S4_1 = 4'b0110, S5_1 = 4'b0100, S6_1 = 4'b1100, S7_1 = 4'b1000;

 reg [3:0] current_state, next_state;

 reg [26:0] clk_divider;
 reg slow_clk;

 always @ (posedge clk)
begin
clk_divider <= clk_divider + 1;
slow_clk <= clk_divider[26];
end

 always @(posedge slow_clk or posedge reset)
begin
    if (reset)
        current_state <= (in == 0) ? S0_0 : S0_1;
    else
        current_state <= next_state;
end
 always @(*)
begin
    if (in == 0)
    begin
        case (current_state)
            S0_0: next_state = S1_0;
            S1_0: next_state = S2_0;
            S2_0: next_state = S3_0;
            S3_0: next_state = S4_0;
            S4_0: next_state = S5_0;
            S5_0: next_state = S6_0;
            S6_0: next_state = S7_0;
            S7_0: next_state = S0_0;
            default: next_state = S0_0;
        endcase
    end
    else if (in == 1)
    begin
        case (current_state)
            S0_1: next_state = S1_1;
            S1_1: next_state = S2_1;
            S2_1: next_state = S3_1;
            S3_1: next_state = S4_1;
            S4_1: next_state = S5_1;
            S5_1: next_state = S6_1;
            S6_1: next_state = S7_1;
            S7_1: next_state = S0_1;
            default: next_state = S0_1;
        endcase
    end
end

 always @(*)
begin
    WXYZ = current_state;

    case(WXYZ[0])
    0: sevenseg0 <= 7'b1000000;
    1: sevenseg0 <= 7'b1111001;
    endcase

    case(WXYZ[1])
    0: sevenseg1 <= 7'b1000000;
    1: sevenseg1 <= 7'b1111001;
    endcase

    case(WXYZ[2])
    0: sevenseg2 <= 7'b1000000;
    1: sevenseg2 <= 7'b1111001;
    endcase

    case(WXYZ[3])
    0: sevenseg3 <= 7'b1000000;
    1: sevenseg3 <= 7'b1111001;
    endcase

    case(in)
    0: sevenseg7 <= 7'b1000000;
    1: sevenseg7 <= 7'b1111001;
    endcase

end
endmodule
```

Experiment 3 - Testbench Code:

```
module sequenceFSM_tb;

    reg clk;
    reg reset;
    reg in;

    wire [3:0] WXYZ;
    sequenceFSM uut (
        .clk(clk),
        .reset(reset),
        .in(in),
        .WXYZ(WXYZ)
    );

    initial begin
        clk = 1;
        reset = 0;
        in = 0;

        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100
        clk = 0;
        #100
        clk = 1;
        #100 $stop;
    end

initial begin
    $monitor("At time %t: clk=%b, reset=%b, in=%b, WXYZ=%b", $time, clk, reset, in, WXYZ);
end

endmodule
```

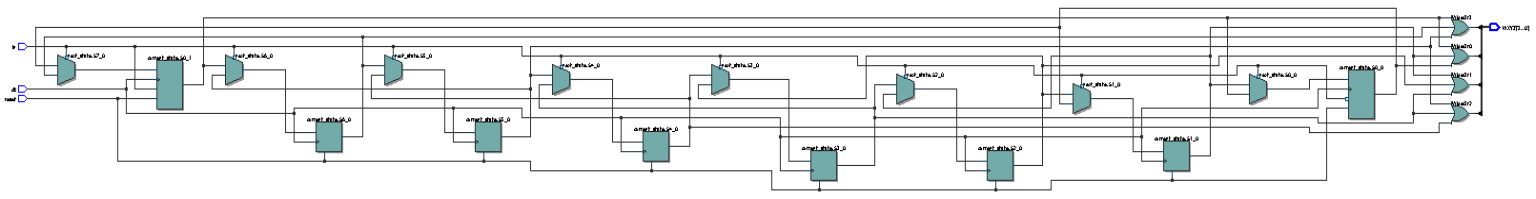
Experiment 3 - Waveform Simulation:



Experiment 3 - Simulation Console:

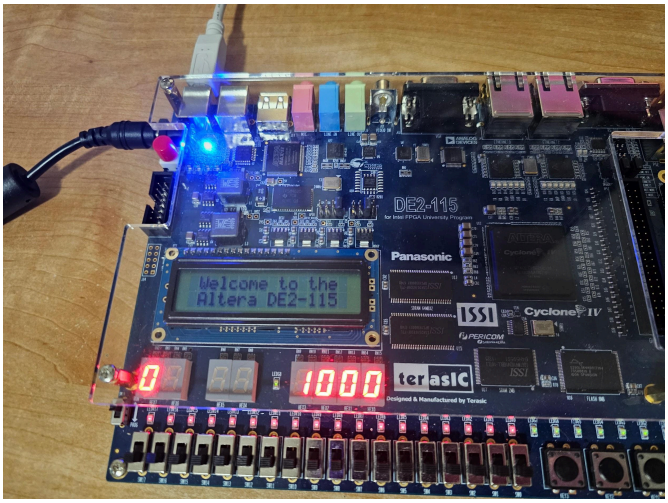
```
# Loading work.sequenceFSM_tb(fast)
VSIM 106> run -all
# At time                    0: clk=1, reset=0, in=0, WXYZ=1000
# At time                  100: clk=0, reset=0, in=0, WXYZ=1000
# At time                  200: clk=1, reset=0, in=0, WXYZ=1100
# At time                  300: clk=0, reset=0, in=0, WXYZ=1100
# At time                  400: clk=1, reset=0, in=0, WXYZ=0100
# At time                  500: clk=0, reset=0, in=0, WXYZ=0100
# At time                  600: clk=1, reset=0, in=0, WXYZ=0110
# At time                  700: clk=0, reset=0, in=0, WXYZ=0110
# At time                  800: clk=1, reset=0, in=0, WXYZ=0010
# At time                  900: clk=0, reset=0, in=0, WXYZ=0010
# At time                 1000: clk=1, reset=0, in=0, WXYZ=0011
# At time                 1100: clk=0, reset=0, in=0, WXYZ=0011
# At time                 1200: clk=1, reset=0, in=0, WXYZ=0001
# At time                 1300: clk=0, reset=0, in=0, WXYZ=0001
# At time                 1400: clk=1, reset=0, in=0, WXYZ=1001
# At time                 1500: clk=0, reset=0, in=0, WXYZ=1001
# At time                 1600: clk=1, reset=0, in=0, WXYZ=1000
# At time                 1700: clk=0, reset=0, in=0, WXYZ=1000
# At time                 1800: clk=1, reset=1, in=1, WXYZ=1001
# At time                 1900: clk=1, reset=0, in=1, WXYZ=1001
# At time                 2000: clk=0, reset=0, in=1, WXYZ=1001
# At time                 2100: clk=1, reset=0, in=1, WXYZ=0001
# At time                 2200: clk=0, reset=0, in=1, WXYZ=0001
# At time                 2300: clk=1, reset=0, in=1, WXYZ=0011
# At time                 2400: clk=0, reset=0, in=1, WXYZ=0011
# At time                 2500: clk=1, reset=0, in=1, WXYZ=0010
# At time                 2600: clk=0, reset=0, in=1, WXYZ=0010
# At time                 2700: clk=1, reset=0, in=1, WXYZ=0110
# At time                 2800: clk=0, reset=0, in=1, WXYZ=0110
# At time                 2900: clk=1, reset=0, in=1, WXYZ=0100
# At time                 3000: clk=0, reset=0, in=1, WXYZ=0100
# At time                 3100: clk=1, reset=0, in=1, WXYZ=1100
# At time                 3200: clk=0, reset=0, in=1, WXYZ=1100
# At time                 3300: clk=1, reset=0, in=1, WXYZ=1000
# At time                 3400: clk=0, reset=0, in=1, WXYZ=1000
# At time                 3500: clk=1, reset=0, in=1, WXYZ=1001
# ** Note: $stop    : C:/altera/13.0sp1/sequenceFSMtb.v(110)
#    Time: 3600 ns  Iteration: 0  Instance: /sequenceFSM_tb
# Break at C:/altera/13.0sp1/sequenceFSMtb.v line 110
```

Experiment 3 - RTL View:

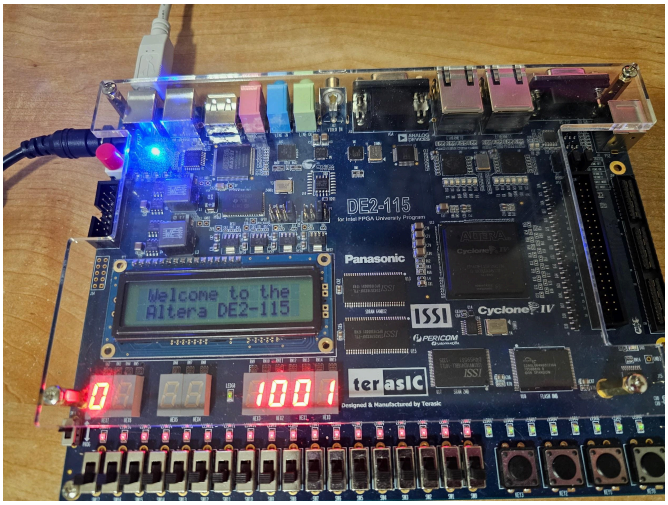


Experiment 3 - DE2 Board Pictures:

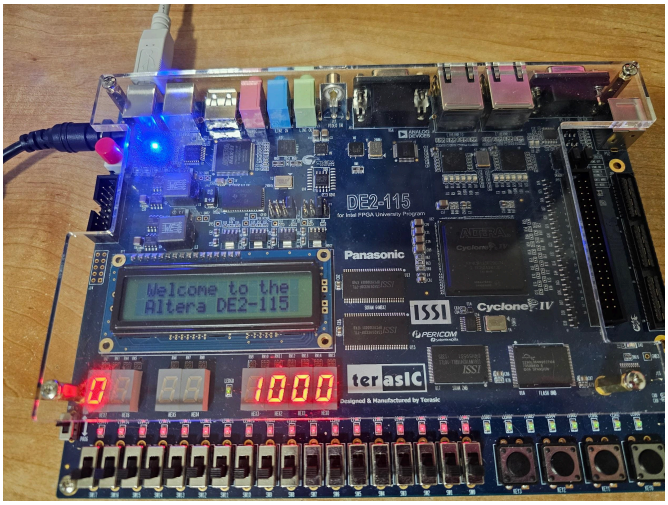
Start of A = 0 Sequence - WXYZ = 1000



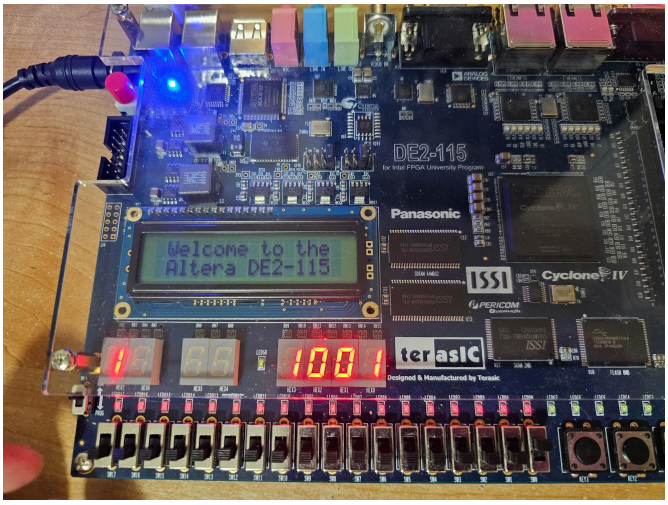
End of A = 0 Sequence - WXYZ = 1001



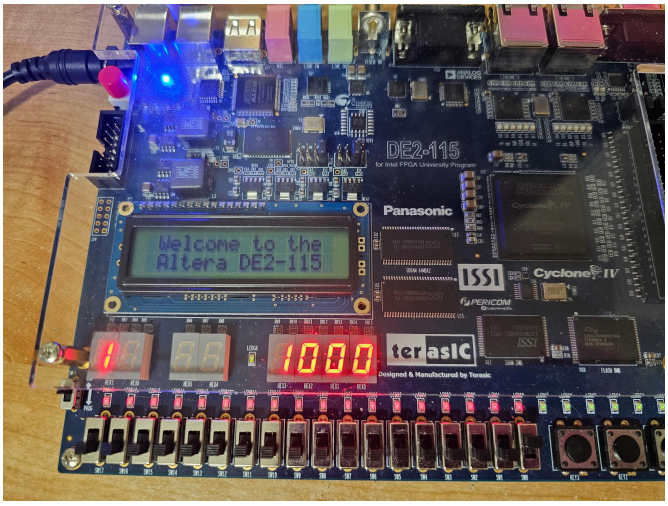
A = 0 Sequence Restarts- WXYZ = 1000



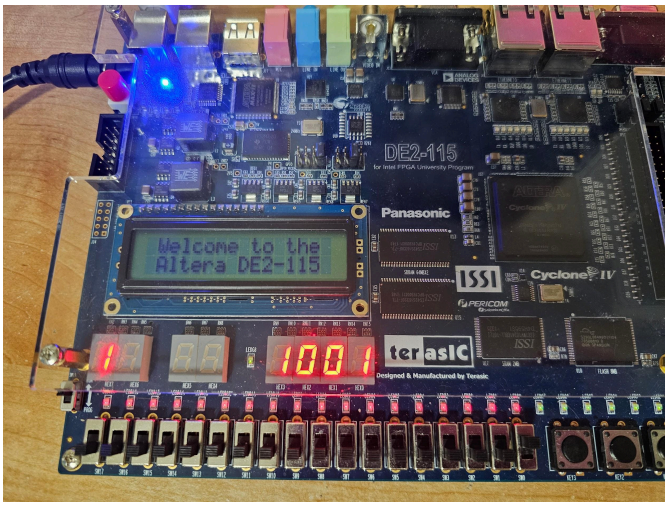
Start of A = 1 Sequence - WXYZ = 1001



End of A = 1 Sequence - WXYZ = 1000



A = 1 Sequence Restarts- WXYZ = 1001



Experiment 3 - Video: Attached to Canvas Submission

Experiment 3 - Video Explanation: Input A starts low. The on-board seven-segment displays go through the 4-bit sequence for A=0. Once the first sequence has gone all the way through and it starts looping back to the beginning again, the A switch is set high and the reset switch is asserted and deasserted. This resets the FSM logic all while setting A high so it can now run through the A=1 sequence, which it does and once again shows on the seven-segment displays.

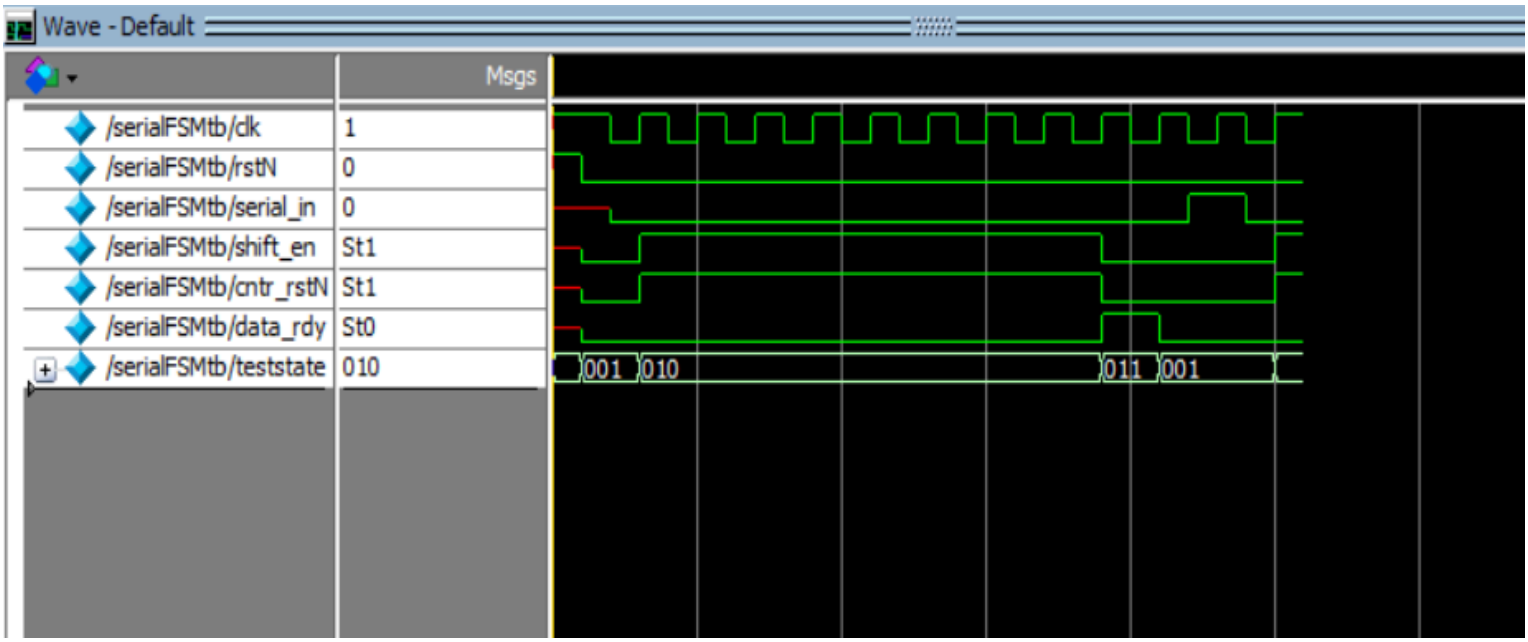
Experiment 4 - Verilog Code:

```
module serialFSM (input clk, input rstN, input serial_in,
output reg shift_en, output reg data_rdy, output reg cntr_rstN, output reg [2:0] teststate );

parameter RESET = 2'b00, WAIT = 2'b01, LOAD  = 2'b10, READY = 2'b11;

reg [2:0] current_state, next_state, downcount;

always @ (posedge clk or negedge rstN)
begin

if (rstN)
        current_state <= RESET;
else
        current_state <= next_state;
end

always @ (*)
begin

next_state = current_state;

case (current_state)

RESET:
begin
        teststate = 3'd0;
        next_state = WAIT;
end
WAIT:
begin
        teststate = 3'd1;
        shift_en = 0;
        cntr_rstN = 0;
        data_rdy = 0;
if (serial_in == 0)
        next_state = LOAD;
end
LOAD:
begin
        teststate = 3'd2;
        cntr_rstN = 1;
        shift_en = 1;
if (downcount == 3'b000)
        next_state = READY;
end
READY:
begin
        teststate = 3'd3;
        shift_en = 0;
        cntr_rstN = 0;
        data_rdy = 1;
                next_state = WAIT;
end

endcase
end

always @ (posedge clk or negedge rstN)
begin

if(rstN)
        downcount <= 3'b111;
else if (current_state == LOAD)
        downcount <= downcount - 1;
else
        downcount <= 3'b111;

end

endmodule
```

Experiment 4 - Testbench:

```
module serialFSMtb;

reg clk;
reg rstN;
reg serial_in;
wire shift_en;
wire cntr_rstN;
wire data_rdy;
wire [2:0] teststate;

serialFSM uut(
        .clk(clk),
        .rstN(rstN),
        .serial_in(serial_in),
        .shift_en(shift_en),
        .cntr_rstN(cntr_rstN),
        .data_rdy(data_rdy),
        .teststate(teststate)
);

initial begin
        // Reset state
        rstN = 1;
        clk = 1;
        #100;
        // Wait state
        rstN = 0;
        #100;
        clk = 0;
        serial_in = 0;
        #100;
        // Load state
        clk = 1;
        #100;
        // 1 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 2 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 3 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 4 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 5 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 6 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 7 clock cycle
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 8 clock cycle -> Ready state
        clk = 0;
        #100;
        clk = 1;
        #100;
        // 1 more clock cycle -> Back to Wait state
        clk = 0;
        #100;
        clk = 1;
        #100;
        serial_in = 1;
        clk = 0;
        #100;
        clk = 1;
        #100;
        serial_in = 0;
        clk = 0;
        #100;
        clk = 1;
        #100;
end
always @ (teststate) begin
        case (teststate)
                3'd0: $display("At time %t: In RESET state", $time);
                3'd1: $display("At time %t: Now In WAIT state", $time);
                3'd2: $display("At time %t: Now In LOAD state", $time);
                3'd3: $display("At time %t: Now In READY state", $time);
        endcase
end
initial begin
    $monitor("At time %t: clk=%b, rstN=%b, serial_in=%b, shift_en=%b, cntr_rstN=%b, data_rdy=%b", $time, clk, rstN, serial_in, shift_en, cntr_rstN, data_rdy);
end
endmodule
```

Experiment 4 - Waveform Simulation:



Experiment 4 - Simulation Console:

```
# Loading work.serialFSMtb(fast)
VSIM 185> run -all
# At time                    0: In RESET state
# At time                    0: clk=1, rstN=1, serial_in=x, shift_en=x, cntr_rstN=x, data_rdy=x
# At time                  100: Now In WAIT state
# At time                  100: clk=1, rstN=0, serial_in=x, shift_en=0, cntr_rstN=0, data_rdy=0
# At time                  200: clk=0, rstN=0, serial_in=0, shift_en=0, cntr_rstN=0, data_rdy=0
# At time                  300: Now In LOAD state
# At time                  300: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                  400: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                  500: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                  600: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                  700: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                  800: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                  900: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1000: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1100: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1200: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1300: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1400: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1500: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1600: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1700: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1800: clk=0, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
# At time                 1900: Now In READY state
# At time                 1900: clk=1, rstN=0, serial_in=0, shift_en=0, cntr_rstN=0, data_rdy=1
# At time                 2000: clk=0, rstN=0, serial_in=0, shift_en=0, cntr_rstN=0, data_rdy=1
# At time                 2100: Now In WAIT state
# At time                 2100: clk=1, rstN=0, serial_in=0, shift_en=0, cntr_rstN=0, data_rdy=0
# At time                 2200: clk=0, rstN=0, serial_in=1, shift_en=0, cntr_rstN=0, data_rdy=0
# At time                 2300: clk=1, rstN=0, serial_in=1, shift_en=0, cntr_rstN=0, data_rdy=0
# At time                 2400: clk=0, rstN=0, serial_in=0, shift_en=0, cntr_rstN=0, data_rdy=0
# At time                 2500: Now In LOAD state
# At time                 2500: clk=1, rstN=0, serial_in=0, shift_en=1, cntr_rstN=1, data_rdy=0
```

3. Questions and Answers

1. Disadvantage of Mealy: Mealy FSMs have outputs that are dependent on current input(s). They also generate these outputs asynchronously. For these reasons, Mealy FSMs are more susceptible to common glitches and not always getting correct outputs right away. These issues are overcome by Moore FSMs because their outputs are independent of current input values and are only based on the machine's current state. Additionally, their outputs are generated synchronously.

2. FSM Encoding Styles: The encoding style of an FSM is the way that states in the machine are encoded/represented. It is significant because it affects the FSM's performance in terms of speed, resource/component usage, and potentially even power consumption. There are a few different kinds of FSM encoding styles. The most straightforward and probably most common one is binary encoding which uses the minimum number of bits required to encode states and does so sequentially using binary (000, 001, 010, 011…). There is also one-hot encoding which uses one bit to encode each state in the machine and follow a pattern (000001, 000010, 000100, 001000…). Gray coding is another FSM encoding style which only changes successive states by one bit in a pattern (000, 001, 011, 010, 110…).

3. Combinational vs Sequential Sections: The combinational section is responsible for handling the next state logic of the FSM. That is, deciding what state the FSM should transition to next or if it should remain in the same state based on the current state and possibly also the inputs. The sequential section is responsible for storing that current state of the FSM.

4. Conclusions

During this lab, we got a refresher on how finite state machines operate, particularly in Verilog and on an FPGA board. We learned more about how FSMs can handle different functionalities and be applied in certain systems to handle all sorts of logic. We also learned more about the differences between Moore and Mealy FSMs. We were able to teach ourselves how to write testbenches for FSMs to more easily test state transitions and processes of the FSMs.