3C7 Project Report

Student Name:	**Owen Clarke**
Student ID Number:	**16323789**
Assessment Title:	**3C7 Assignment 2**
Lecturer (s):	**Naomi Harte**
Date Submitted	**12/04/2019**

Signed: Owen Clarke

Date: 12/04/2019

Introduction

The purpose of this Assignment was to build upon the design concept of “off-the-shelf” modules that were completed in previous assignments to build a Sequence Detector. This model was similar to Assignment 1 that was due after Reading Week. By building different “circuits” or modules to perform certain tasks, they are combined together to add extra functionality to an FPGA. For the purposes of this assignment, the functionality of the Linear-Feedback Shift Register (LFSR) and the Stop-Watch needed to be combined with a Finite State Machine (FSM) in order to create a sequence detector to spot the pattern “1010”.

Finite State Machine

A finite state machine has a defined number of states based on it’s input, it will either stay in the same state or move/advance to the next state. For the purposes of this assignment, a FSM will be used to detect the input sequence “1010”. Taking the input the Most Significant Bit (MSB) from the LFSR, the FSM will detect whether if it is a 1 or a 0, for the purposes of this design, if the first bit is a 1, it will then move to the next state (State B), if it is a 0, it will stay in the same state (State A). Even as it moves to it’s next state, it will give an output of 0, until it reaches it codeword “1010” where it it will declare an output of 1. This output will then be passed to the Counter Module.

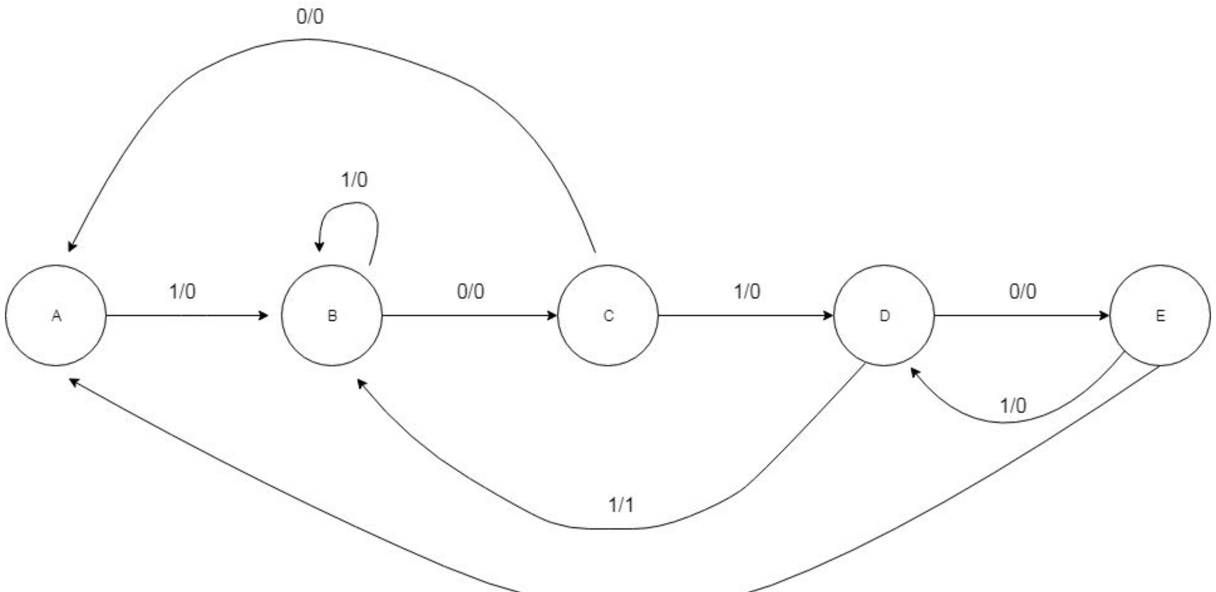


Figure 1: Initial Finite State Machine

1	Finite State Machine for 1010				
2	Initial Design				
3	Current State	Next State		Output	
4		Input 0	Input 1	Input 0	Input 1
5	A	A	B	0	0
6	B	C	B	0	0
7	C	A	D	0	0
8	D	E	B	1	0
9	E	A	D	0	0
10	Finite State Machine for 1010				
11	Minimising				
12	Current State	Next State		Output	
13		Input 0	Input 1	Input 0	Input 1
14	A	A	B	0	0
15	B	C	B	0	0
16	C	A	D	0	0
17	D	E->C	B	1	0
18	E	A	D	0	0
19	Finite State Machine for 1010				
20	Final Design				
21	Current State	Next State		Output	
22		Input 0	Input 1	Input 0	Input 1
23	A	A	B	0	0
24	B	C	B	0	0
25	C	A	D	0	0
26	D	C	B	1	0

The above tables represent the stages that were taken to simply the finite state machine. Note the numbers in the blue on the left hand side are numbers to reference to a specific line in the table. It can be seen that line 7 and line 9 of the table are of similar state. Because of this, Line 9 can be removed and, state E can be removed from the FSM as of line 18. Line 17 show how the states can be simplified to move to an already existing state instead of progressing to the now gone state E. The Finite State Machine can be reduced to just four states instead of the 5. The design for this is a Mealy machine. The output of the states depend on both the current state and the inputs.

Based on my method of simplification by spotting similar lines of next_state, it would appear that the FSM could be reduced even simpler. For example in the Final Design, Line 23 is the same as Line 25 and Line 24 is the same as Line 26. This would mean we would have only 2 states as State C and D would be removed. However this is not possible since we a one would occur every time half the sequence occurs, also State B and State D have different output states which makes them incompatible. For example, if the first two bits were 1 and 0, this would signify that the codeword has been found, when in fact, it has not. There would be no distinguishing between the 2 bit 0 (which produces a 0 output) and the 4th 0 (which produces a 1 output). Below is my final design for the 1010 Sequence Detector FSM.

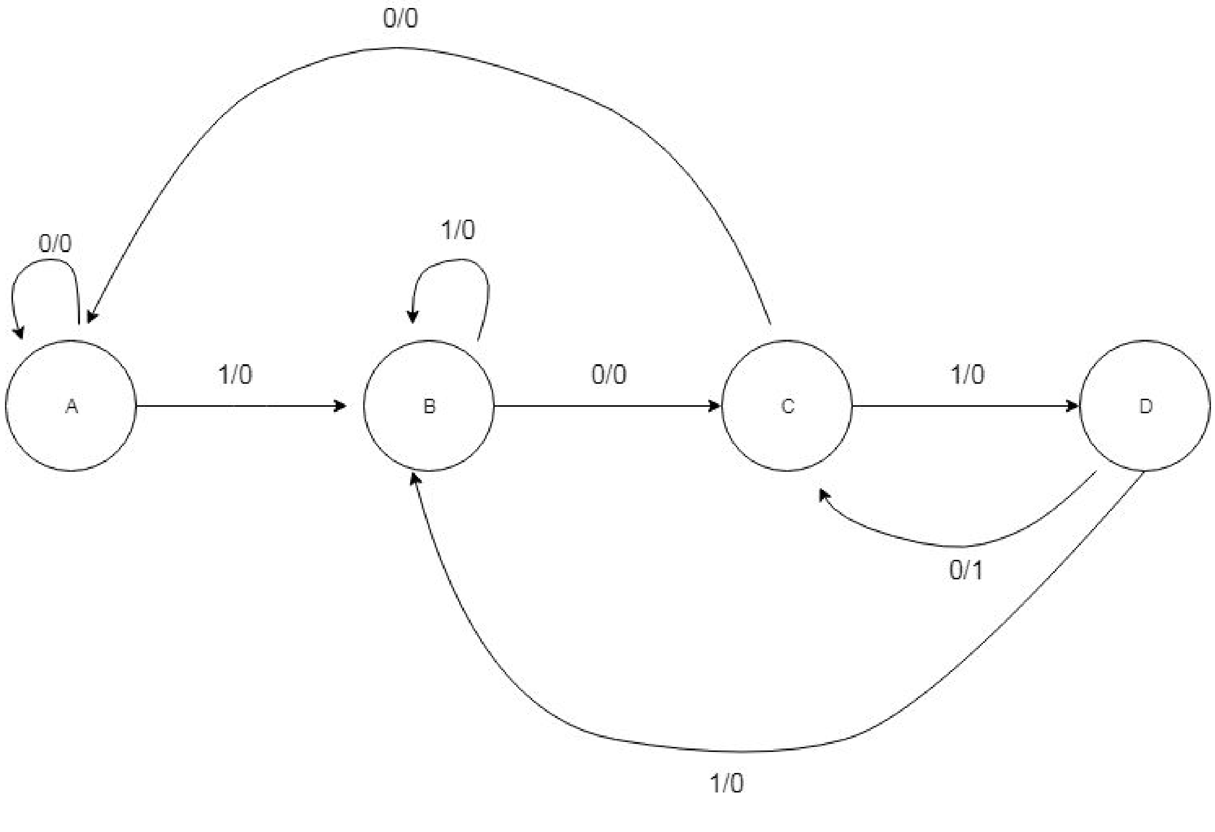


Figure 2: Minimised Finite State Machine

Overlapping

My Finite State Machine is designed to take into account overlapping codewords. For example with the code word, it is very easy for the pattern to repeat. The FSM counts overlapping codewords, for example 101010 would count as 2 codewords being found (the codeword in the Red text and the codeword with the blue background). Once the 4th bit 0 has been detected, the FSM will return to state C. This means that to detect another codeword, it only needs to detect another 1 and then a 0. To deal with Non-Overlapping Sequence Detector, after a codeword has been detected, D would return to state A so the detector would have to iterate through all four states again.

Demo

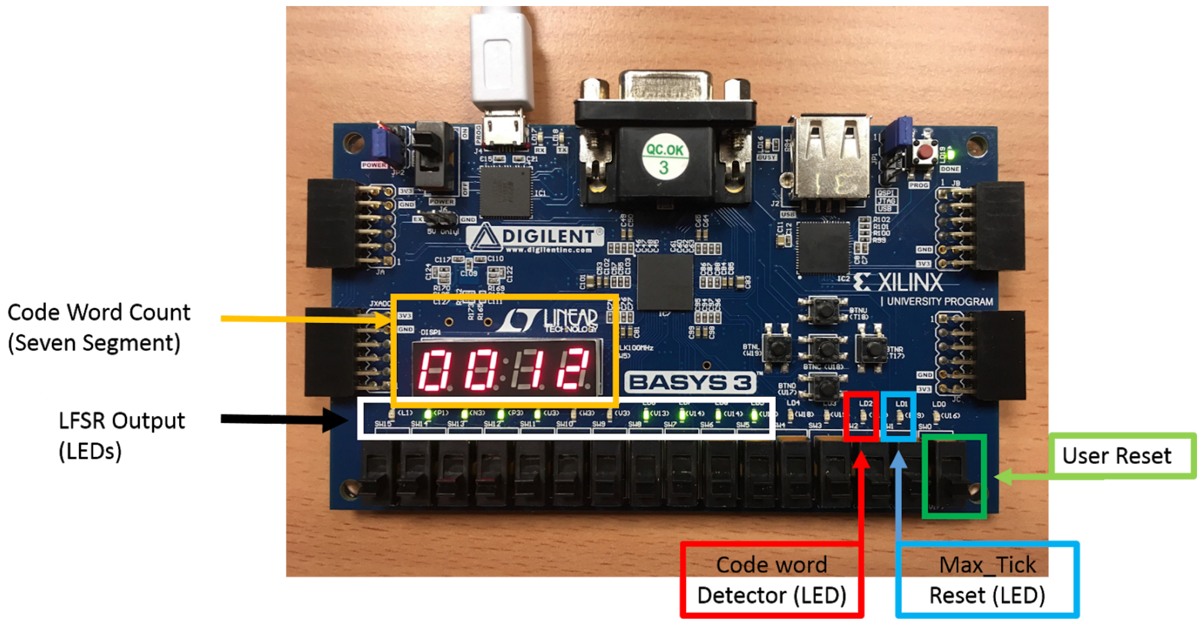


Figure 3: Demo of Board Features

The above image (Figure 3) demonstrates the key features of the FPGA on the Basys 3 Board. The count of how many times the codeword “1010” appears in the Seven Segment Display. This will increment to 128 and then reset. The Code Word Detector LED (LED PIN U19) will flash once a codeword has been detected and the counter on the seven segment display increases. The Max_Tick LED will flash once the Seed value of 11’b11001100101 has been reached by the LFSR and once this flashes, the counter and the Seven Segment Display will go back to zero and start incrementing again. The User_Reset switch is defined on the board as V17. Once this is high, the counter will be reset back to 0 and will remain at 0 until the switch has been deactivated. Once the switch has been set low, the counter and Seven Segment Display will begin to increase again. The LFSR output stream can be seen on the remaining LEDs from U15 being the least significant bit to L1 of the LEDs being the most significant bit. For my project, I am incorporating my tap outputs from Assignment F. This means that bit 10 and bit 8 will produce the XOR output. These are LED’s L1 and N3.

Amount of Times Code Word Appears

From the 1024 cycles (2^{11-1}) , the codeword “1010” appears 128 times. The photos below show that after the Max_tick reg value has been reached, it will reset back to 0.

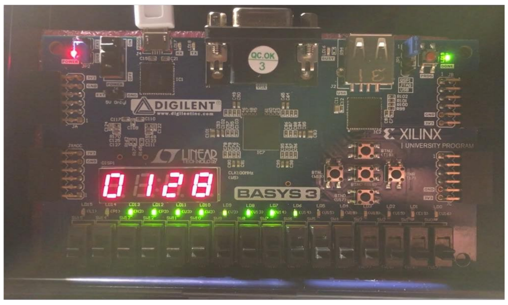


Figure 4: Max Number of Codewords

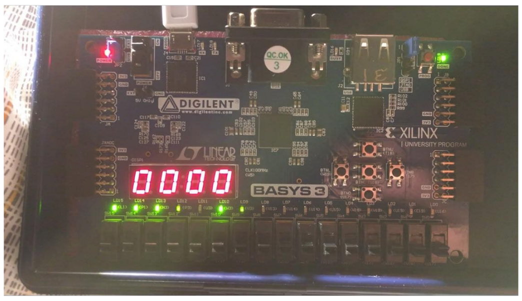
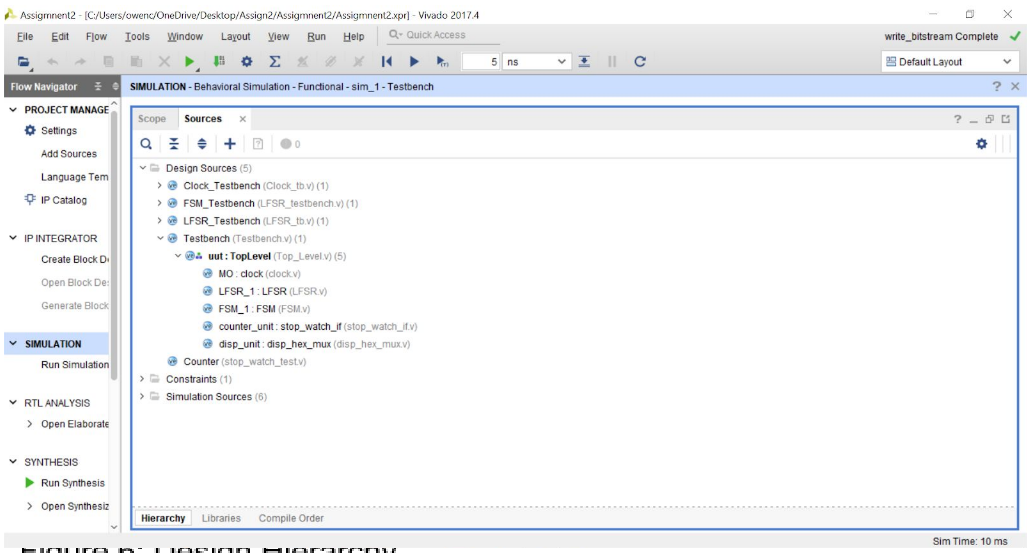


Figure 5: Counter Resetting to 0

Design Hierarchy

Figure 6 is an image of the Vivado program is a screen capture of the the Hierarchy of my FPGA design. A testbench file used for Simulations passes wires into the Top Level Module which instantiates the 5 modules: A clock module, the LFSR module, the Finite State Machine Module, a counter Module and a Seven Segment Display Module.

I have various other testbenches I set up while creating the Sequence Detector to ensure correct functionality of each individual module before they were combined together. Previously I had both my *counter_unit* and *disp_unit* with a “Counter” module. However I was getting errors when displaying to the board because I was passing in the wrong Clock into the counter. It was receiving the boards 100Mhz clock which was too fast to properly increment the tick signals. This issue was solved by bringing all modules up to the *TopLevel* and pass the correct clocks into each module, the 100Mhz clock for the display and the


Assignment2 - [C:/Users/owenc/OneDrive/Desktop/Assign2/Assignment2/Assignment2.xpr] - Vivado 2017.4

write_bitstream Complete

SIMULATION - Behavioral Simulation - Functional - sim_1 - Testbench

```
Design Sources (5)
  Clock_Testbench (Clock_tb.v) (1)
  FSM_Testbench (LFSR_testbench.v) (1)
  LFSR_Testbench (LFSR_tb.v) (1)
  Testbench (Testbench.v) (1)
    uut : TopLevel (Top_Level.v) (5)
      MO : clock (clock.v)
      LFSR_1 : LFSR (LFSR.v)
      FSM_1 : FSM (FSM.v)
      counter_unit : stop_watch_if (stop_watch_if.v)
      disp_unit : disp_hex_mux (disp_hex_mux.v)
  Counter (stop_watch_test.v)
Constraints (1)
Simulation Sources (6)
```

Sim Time: 10 ms

Figure 6: Design Hierarchy

slowed down *clk* (output from the Clock Module) into the *counter_unit* module.

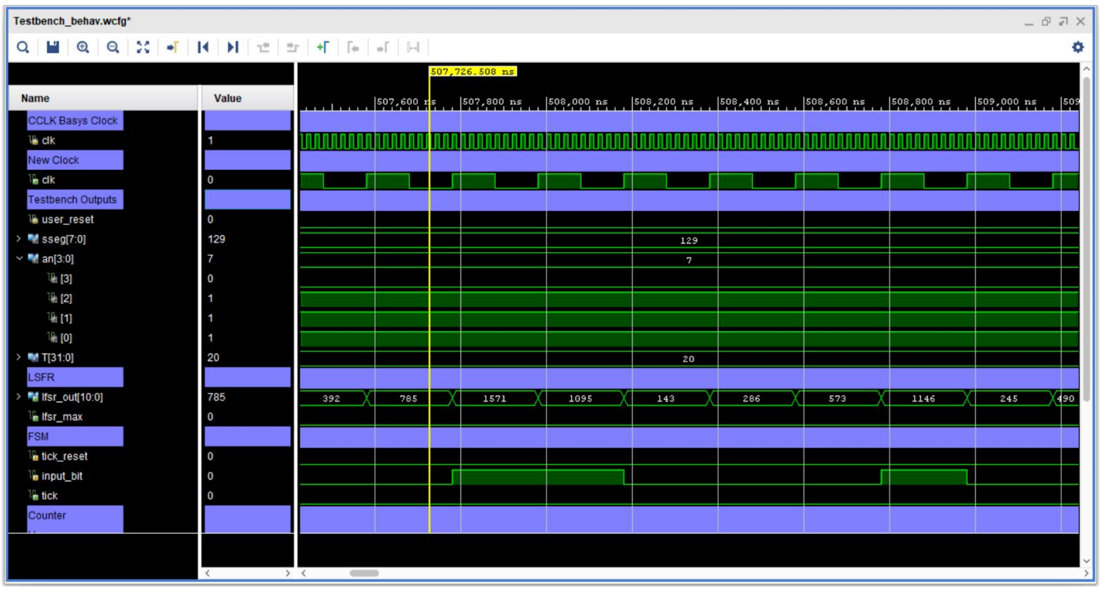
Controlling the Clock

It can be seen in the schematic in the next section (Figure 8), that the Basys board 100Mhz clock is passed through a clock divider module and is reduced to by a scale of 5,000,000. The FPGA code for this module was taken from the “Tutorials on Using Clocks” handout provided with Assignment F. The output of this module is a 1Hz clock signal instead of the 100Mhz. As stated above, I had issues passing the wrong clock into the Counter module which meant it would increment straight to 3200 and then 6400 and back to 0 within a second. It works by utilising the existing clock, and counts upwards till it reaches the clock scale and inverts the clk signal.

In my previous assignment, I could not understand why the output clk was not showing on the simulation until I realised, my simulation was running for Nanoseconds while the new clock had a rising edge every second. To view the new slowed down clock value (clk), I decreased the clock scale to a value of 5 instead of 5,000,000. Below (Figure 7) is a clock signal with a clock scale of 5. The Waveform simulation shows the slowed down clock signal by clock scale of to a frequency of 10Mhz.

$$clkscale \ = \ \frac{desired\,Period}{2\,x\,(CCLK\ period)} \ = \ \frac{CCLK\ frequency}{2\,x\,(desired\ Frequency)}$$

$$clk \ = \ \frac{100*10^6}{2*5} = 10\text{Mhz Frequency}$$



Name	Value
CCLK Basys Clock	
clk	1
New Clock	
clk	0
Testbench Outputs	
user_reset	0
sseg[7:0]	129
an[3:0]	7
[3]	0
[2]	1
[1]	1
[0]	1
T[31:0]	20
LSFR	
lfsr_out[10:0]	785
lfsr_max	0
FSM	
tick_reset	0
input_bit	0
tick	0
Counter	

Figure 7: Slowed Down Clock

Schematic of Design

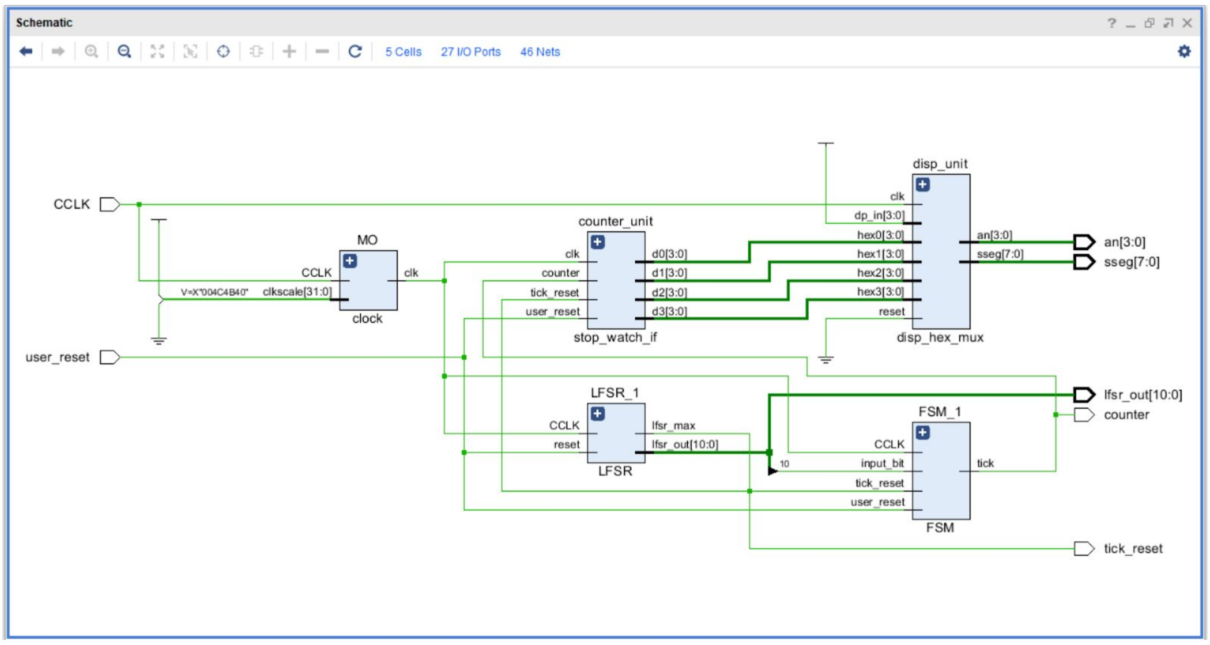


Figure 8: Design Schematic

Figure 8 is the Elaborated Design from Vivado showing how all the inputs and outputs of the different modules are connected. The CCLK gets passed into the Clock Module and then the output clk (which has been slowed down by a scale of 5000000) is used for every module apart from the display. The *counter* output is used as an input into the *counter_unit* but is also outputted as an LED on the board. The output flashes every time a codeword has been detected. This is the same for *tick_reset* which is the output from the LFSR module but becomes an input for the *counter_unit* and an LED output on the board. The tick reset is will flash high once the seed value has been reached in the LFSR, telling the other modules that it is starting a new cycle.

Waveform

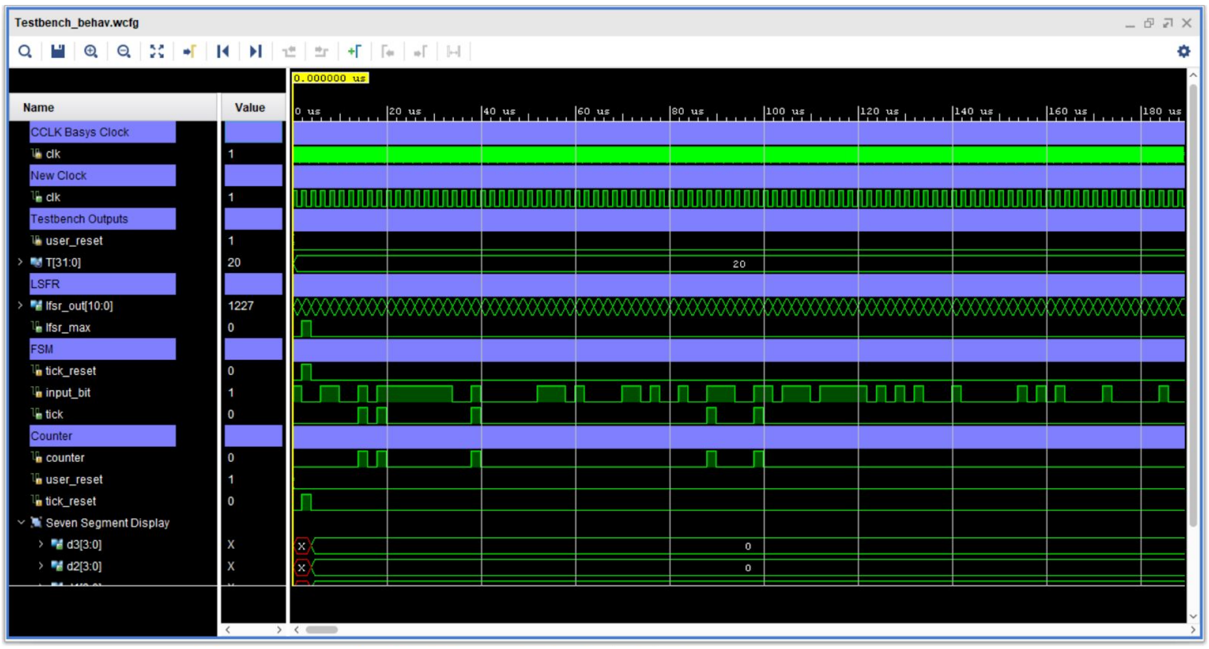


Figure 9: Waveform 1

In figure 9 is a the first 180us seconds of the simulation. The first 10 cycles of the CCLK clock (Basys Board 100Mhz clock), the User Input is set high. Since the first value in the LFSR is the seed value, the tick_reset is also set high. This means that there is no inputs to the Seven Segment Display. The top two waveforms are the CCLK clock signal and the the slowed down clock signal. This simulation has a clkscale of 50 in order to see the Sequence detector work within the small simulation time. With a *clkscale* of 5,000,000, the slowed down clock (clk) would have a period of 1 second, while the simulation only lasts for 10ns. Hence, no visible inputs or calculations could be seen on the waveform.

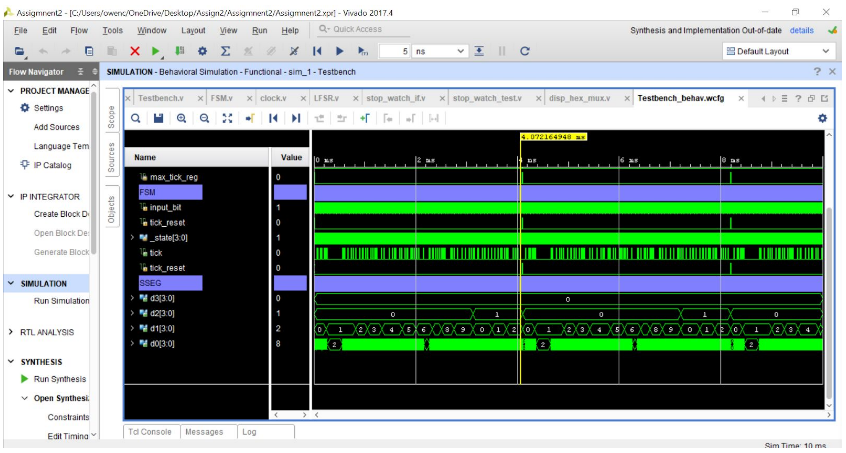
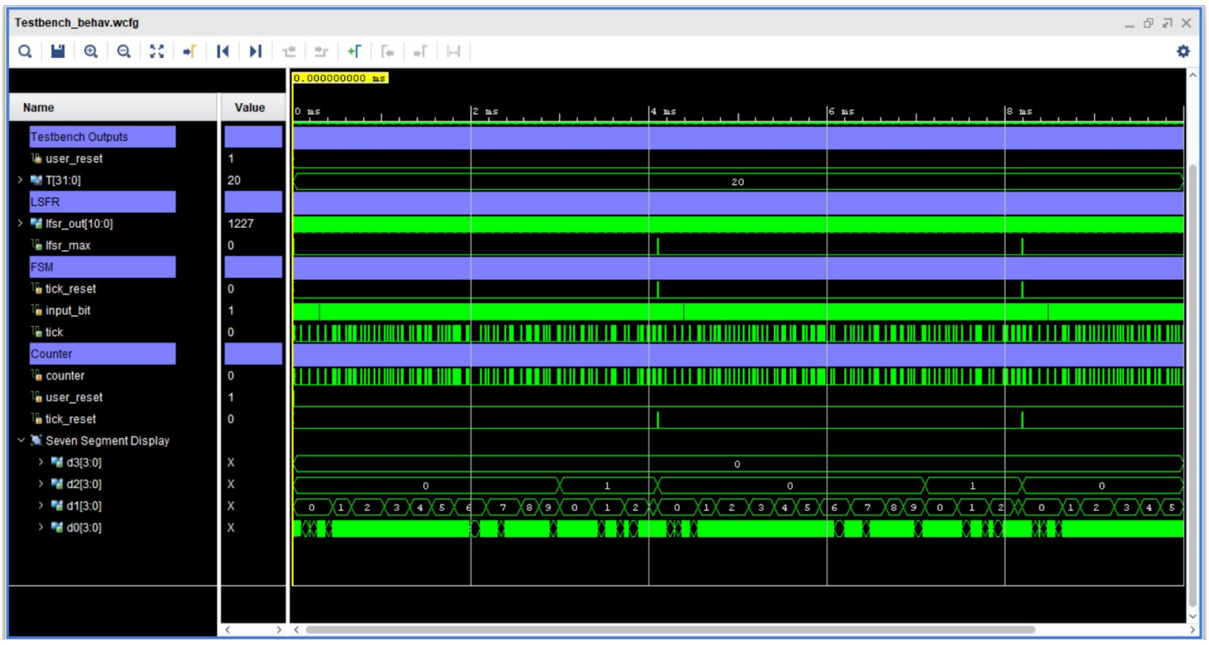


Figure 10: Waveform 2 - Max Code Count shown on Seven Seg in Waveform

It can be seen that the LSFR_Max has the same waveform as the tick_reset. This is also the same for the *tick* and the *counter.* This is because I have different inputs and outputs from each of the different modules in the Waveform. This was for testing and making sure that each module was correctly receiving the right values. The input_bit is the most significant bit from the LFSR that is passed into the Finite State Machine. Over the course of the simulation, the value of *input_bit* changes as it generated by the XOR function from the tap (bit 10 and bit 8 XOR'd together) values in the LFSR.

Figure 10 shows the values that will be displayed on the Seven Segment Display before the *tick_reset* goes high and resets the counter back to 0. This matches the output from the Seven Segment display when the code has been loaded onto the Basys board.



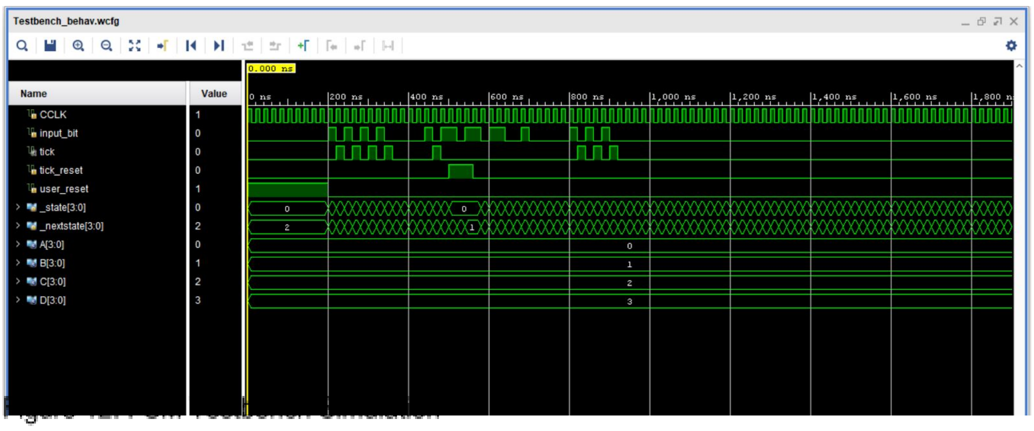
Name	Value
Testbench Outputs	
user_reset	1
T[31:0]	20
LSFR	
lfsr_out[10:0]	1227
lfsr_max	0
FSM	
tick_reset	0
input_bit	1
tick	0
Counter	
counter	0
user_reset	1
tick_reset	0
Seven Segment Display	
d3[3:0]	X
d2[3:0]	X
d1[3:0]	X
d0[3:0]	X

Figure 11: Waveform 3

The third waveform (Figure 11) displays the entire 10us simulation. The tick_reset can be seen as the output from the LFSR module *ifsr_max* and is then seen lower down in the waveform in th FSM and Counter as *tick_reset*. The amount of times the codeword has been detected can be seen in the FSM section and the Counter Section. This is because the *tick* from FSM is an input for the Counter module. Here the Sequence Detector has gone through two cycles, as the tick_reset has been set high.

FSM Test Bench

The test vectors for this testbench can be found in the Appendix after the module code. This testbench simulates inputting a stream of random 1's and 0's to replicate what



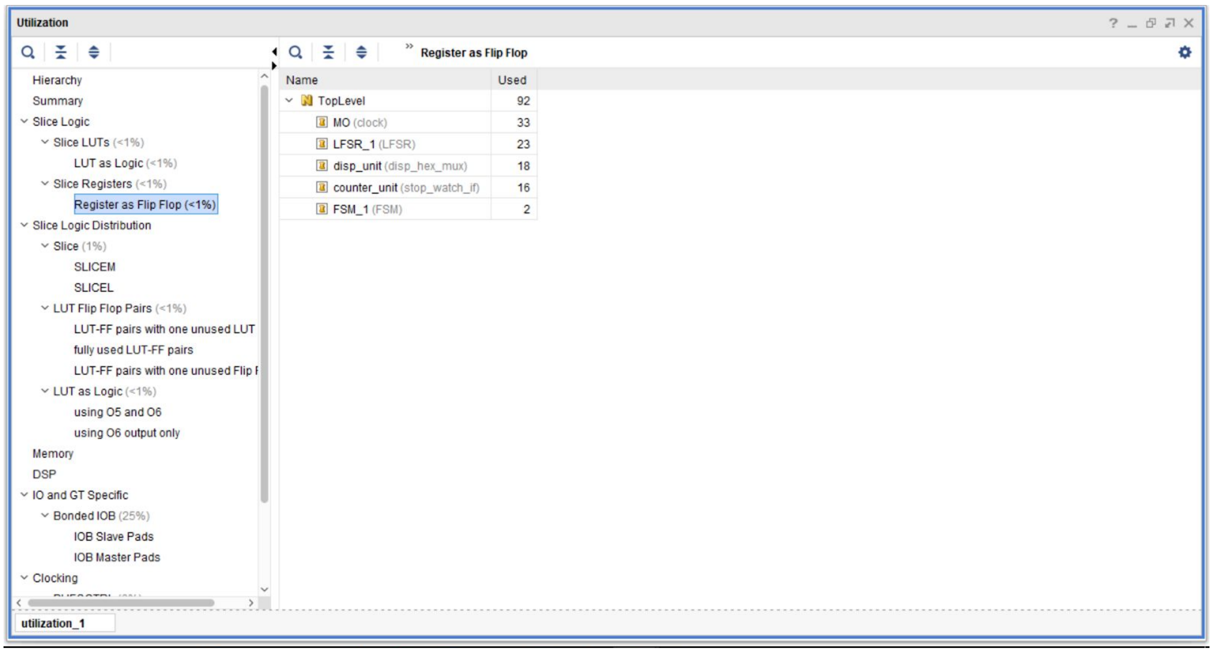
Name	Value
CCLK	1
input_bit	0
tick	0
tick_reset	0
user_reset	1
_state[3:0]	0
_nextstate[3:0]	2
A[3:0]	0
B[3:0]	1
C[3:0]	2
D[3:0]	3

Figure 12: FSM Testbench Simulation

the LFSR would output. The states A,B,C and D are assigned the value 0,1,2 and 3 respectively.

I do not have any other test vectors for the module as each one relied on user inputs or outputs to perform their functionality. The smaller testbenches that I created, only had a clock as an input testvector since the LFSR only requires a clock signal. If the results in the simulation looked off, I would use the smaller test benches, but this would simulate the same inputs just without any other module connected. I used my FSM testbench to make sure that the right output was being given. However this was the only module I felt could be given controllable outputs which we could test.

Utilisation Report / Registers as Flip Flops


Utilization

- Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (<1%)
    - LUT as Logic (<1%)
  - Slice Registers (<1%)
    - Register as Flip Flop (<1%)
- Slice Logic Distribution
  - Slice (1%)
    - SLICEM
    - SLICEL
  - LUT Flip Flop Pairs (<1%)
    - LUT-FF pairs with one unused LUT
    - fully used LUT-FF pairs
    - LUT-FF pairs with one unused Flip F
  - LUT as Logic (<1%)
    - using O5 and O6
    - using O6 output only
- Memory
- DSP
- IO and GT Specific
  - Bonded IOB (25%)
    - IOB Slave Pads
    - IOB Master Pads
- Clocking

Register as Flip Flop

Name	Used
TopLevel	92
MO (clock)	33
LFSR_1 (LFSR)	23
disp_unit (disp_hex_mux)	18
counter_unit (stop_watch_if)	16
FSM_1 (FSM)	2

utilization_1

Figure 13: Utilisation Report

Conclusion

The code for my Sequence detector worked successfully on the board and was resetting. There were a few small problems with the display and the clock values but my other modules were taken straight from the previous assignments and placed into the design with minimal changes. My small bitsize meant I had a small codeword count of 128 which would then reset back to 0 correctly. I became more aware of blocking and non-blocking statements when building the FSM and other modules as I became more familiar with each of their functionality and purpose such as non-blocking running on the Posedge of the clock signal updating the next state of the FSM. Overall, I am happy with my design, I feel it has a logical implementation with a few extra key features such as the counter and tick_reset LED to indicate the start of a new cycle.

Link for Video of Sequence Detector Reaching Max Tick

https://drive.google.com/file/d/18-F7a-MEwamlda0BXcRYa81uG3L7NQK0/view?usp=sharin
g

Appendix: Code

Toplevel Test Benchfile

```
`timescale 1 ns/10 ps

module Testbench();

//Declaring all the values
localparam T = 20; // Clock Period as spec

reg clk, user_reset;
wire counter, tick_reset;
wire[7:0] sseg; //Our clock signal and Res
wire[3:0] an;

 // Display counter on Seven Segment Displ
//UUT instantiation/
TopLevel uut
  (.CCLK(clk),.user_reset(user_reset),.an(
// Listing 4.19

always
begin
    clk = 1'b1; // Set the clock high
    #(T/2); //Half a period

    clk = 1'b0; //Set the Clock Low
    #(T/2); //For half a period
end

initial
begin

    user_reset <= 1'b1;
    #(T*10);
    user_reset <= 1'b0;
end
```

//Top Level Module Code

```
module TopLevel(
input CCLK, user_reset,
output wire tick_reset,//Defines the clock input and the Reset //
Defines the output of the LED's
output wire [3:0] an,
output wire [7:0] sseg, //Output for each Seven Segment Display
output wire[10:0] lfsr_out, //LFSR 11 bit output for LED's
output wire counter  //This will display on the Seven Seg Display
everytime the FSM is correct
);
//reg clk;
 //LFSR output
//wire tick_reset; //Reset when max cycles reached
reg lsfr_max; // Goes high when 2^n-1 cycles has occured
wire clk; //Define the clock
wire[3:0] d0,d1,d2,d3; //Outputs for the Seven Segment Units


//-----------------------------------------------------------------------
----------
//                              Clock Signal
//-----------------------------------------------------------------------
------------
clock MO(.CCLK(CCLK),.clkscale(5000000),.clk(clk)); //Used to scale the
clock to a scale of 5000000

//-----------------------------------------------------------------------
-------------
//                        Linear Feedback Shift Register
//-----------------------------------------------------------------------
------------

LFSR LFSR_1
(.CCLK(clk),.reset(user_reset),.lfsr_out(lfsr_out),.lfsr_max(tick_reset)
);
//-----------------------------------------------------------------------
----------
//                            Finite State Machine
//-----------------------------------------------------------------------
----------

FSM FSM_1
(.CCLK(clk),.input_bit(lfsr_out[10]),.user_reset(user_reset),.tick_reset
(tick_reset), .tick(counter));
```

```
stop_watch_if counter_unit
      (.clk(clk),.user_reset(user_reset),.tick_reset(tick_reset),
//Calculate how many times codeword appears based on counter input
       .d3(d3),.d2(d2), .d1(d1), .d0(d0),.counter(counter));

disp_hex_mux disp_unit   (.clk(CCLK), .reset(1'b0),.hex3(d3), .hex2(d2),
.hex1(d1), .hex0(d0),.dp_in(4'b1111), .an(an), .sseg(sseg));

endmodule
```

Clock.v module

```
module clock(input CCLK,input[31:0] clkscale, output reg clk);

reg [31:0] clkq =0;//Create 32 bit register to increment
// Set the clock signal to 0 //Assign the clock scale
initial
begin
clk = 1'b1; //Set Clock to high
end

always@(posedge CCLK) //On the positive Edge of the Clock
    begin
        clkq = clkq+1;  //Increment register
            if (clkq >= clkscale) //if the clkq counter reaches the
scale
                begin
                    clk = ~clk; //Invert the clock signal
                    clkq = 0;//Reset the counter
                 end
     end

endmodule
```

LFSR.v code

```
module LFSR
(
input wire CCLK, reset,//Defines the clock input and the Reset
output wire[10:0] lfsr_out, // Defines the output of the LED's
output wire lfsr_max //Defines the output for max_tick signal
);
//signal declaration
wire clk; // The output from the Clock.v
```

```
reg [10:0] lfsr_reg; // The current value of stored in the register
reg [10:0] lfsr_next; // The value of the next values in the register
reg[10:0] lfsr_seed; //The seed value using for reset
reg max_tick_reg;// The value to set if 2^n-1 has been reached
reg lfsr_tap; // The value of the taps are stored here


always @(posedge CCLK, posedge reset)
begin
lfsr_seed <= 11'b11001100101;//The Seed value

if (reset)//If the Reset has been activated
    lfsr_reg <= 11'b11001100101; // Set the register to the seed value

else
    lfsr_reg <= lfsr_next; //Store the next value in the lsfr_reg value
end
//Always block for calculating the tap values

always @(posedge CCLK)

    if(lfsr_reg ==  11'b11001100101)
        max_tick_reg <= 1'b1;
    else
        max_tick_reg <= 1'b0;


always @*
begin
    // generate the feedback by XOR of tap 8 and 10
    lfsr_tap = lfsr_reg[8] ^ lfsr_reg[10];
    // feedback goes into 0 position. Other bits shift up
    lfsr_next = {lfsr_reg[9:0],lfsr_tap }; //Combine both the first 9
bits and the new tap value together
end // next state logic
// output logic
assign lfsr_out = lfsr_next; // Set the output to the value stored in
the register
assign lfsr_max = max_tick_reg; //Set the count output to the
maxt_tick_reg
endmodule
```

FSM.v Finite State Machine Code

```
module FSM(
              input input_bit, // Bit from the LSFR machine
              input CCLK, //Clock signal
              input user_reset, // User Reset
              input tick_reset, //Tick Reset, when cycles have completed
              output reg tick   // Output signal from FSM

);

reg[3:0] _state, _nextstate; //Define current and next state

// symbolic state declaration
localparam A = 4'b0000, B = 4'b0001, C =4'b0010, D = 4'b011; //Set the
value for each state, ie value of 3 means we are at state D

// state register
always @(posedge CCLK, posedge user_reset, posedge tick_reset) //On the
positive clock, tick reset or User Reset
begin
if (user_reset || tick_reset) //If User Reset or Tick Reset from LFSR is
activated
_state <= A; //Set to State A
else
_state <= _nextstate;// Progress to next state
end

// next-state logic and output logic
always @*
begin
    _nextstate = _state; // default state: the
    tick = 1'b0; // default output: 0

case (_state)
A:
    if (input_bit == 1'b1) //If 1
        begin
              tick = 1'b0; //Output is 0 - Codeword not detected
              _nextstate = B; // Move to state B
        end
    else  // If input bit is a 0
        begin
              tick = 1'b0; //Output is 0 - Codeword not detected
              _nextstate = A; //Stay in same state
        end

B:
    if (input_bit  == 1'b0)
        begin
                tick = 1'b0; //Output is 0 - Codeword not detected
               _nextstate = C; //Move to State C
        end
    else  // If input bit is a 0
        begin
               tick = 1'b0; //Output is a 0 - Codeword not detected
               _nextstate = B; //Stay in State B
        end



C:
    if(input_bit  == 1'b1)
        begin
                tick = 1'b0; //Output is a 0 - Codeword not detected
                _nextstate = D; //Move to State D
        end

    else //Input bit is a 0
        begin
                tick = 1'b0; // Output is 0 - Codeword not detected
                _nextstate = A;
        end
D:
    if(input_bit  == 1'b0)
        begin
                tick = 1'b1; // Codeword detected
                _nextstate = C; //Move back to state C

        end
     else //If bit is 1
        begin
                tick = 1'b0; //Codeword not detected
                _nextstate = B; //Move back to state B
        end


default: _nextstate = A; //Default case is A

endcase

end
```

```
endmodule
```

Counter stop_watch_if.v

```
// Listing 4.18
module stop_watch_if
   (
    input wire clk,counter,user_reset,tick_reset, //Inputs
    output wire [3:0] d3, d2, d1, d0 //Output Hex
   );

   reg [3:0] d3_reg,d2_reg,d1_reg ,d0_reg;
   reg [3:0] d3_next, d2_next,d1_next,d0_next;
   // body
   // register
   always @(posedge clk)
   begin
      //ms_reg <= ms_next;
      d3_reg <= d3_next;
      d2_reg <= d2_next;
      d1_reg <= d1_next;
      d0_reg <= d0_next;
   end

  always @*
  begin
              // default: keep the previous value
              d0_next = d0_reg;
              d1_next = d1_reg;
              d2_next = d2_reg;
              d3_next = d3_reg;
         if (user_reset||tick_reset)
            begin
                d0_next = 4'b0; // Set the values to all 0's: 0000
                d1_next = 4'b0;
                d2_next = 4'b0;
                d3_next = 4'b0;
            end

         if (counter) // If not clear, incremenent the clock
                 if (d0_reg != 9)//While the units do not go past 9
                    begin
                    d0_next = d0_reg + 1;
                    end
                     //Increase units by 1
                 else              // reach XX9
                    begin
                       d0_next = 4'b0; // Reset the units to 0
                       if (d1_reg != 9) // While the tens do not
increase past 9
                          d1_next = d1_reg + 1; //Increase tens by 1
                       else       // reach X99
                          begin
                             d1_next = 4'b0; //Reset the tens to 0
                             if (d2_reg != 9) //While the hundreds do
not increase past 6
                                d2_next = d2_reg + 1; //Increase
hundreds by 1
                             else // reach 999
                                begin
                                  d2_next = 4'b0; // Reset the hundreds
to 0
                                    if(d3_reg !=9) //While the thousands
do not increase past 9
                                            d3_next = d3_reg+1; //
Increase thousands by 1
                                    else
                                             d3_next = 4'b0; // Reset
thousands to 0
                                 end
                          end
                    end

              end

   // output logic
   assign d0 = d0_reg; // Assign the units output to the SSEG
   assign d1 = d1_reg; // Assign the tens output to the SSEG
   assign d2 = d2_reg; // Assign the Hundreds output to the SSEG
   assign d3 = d3_reg; // Assign the Thousands output to the SSEG

endmodule
```

Disp_hex_mux.v Seven Seg Code

// Listing 4.15

```
module disp_hex_mux
   (
    input wire clk, reset,
    input wire [3:0] hex3, hex2, hex1, hex0,  // hex digits
    input wire [3:0] dp_in,             // 4 decimal points
    output reg [3:0] an,  // enable 1-out-of-4 asserted low
    output reg [7:0] sseg // led segments
   );

   // constant declaration
   // refreshing rate around 800 Hz (50 MHz/2^16)
   localparam N = 18;
   // internal signal declaration
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   reg [3:0] hex_in;
   reg dp;

   // N-bit counter
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         q_reg <= 0;
      else
         q_reg <= q_next;

   // next-state logic
   assign q_next = q_reg + 1;

   // 2 MSBs of counter to control 4-to-1 multiplexing
   // and to generate active-low enable signal
   always @*
      case (q_reg[N-1:N-2])
         2'b00:
            begin
               an =  4'b1110;
               hex_in = hex0;
               dp = dp_in[0];
            end
         2'b01:
            begin
               an =  4'b1101;
               hex_in = hex1;
               dp = dp_in[1];
            end
         2'b10:
            begin
               an =  4'b1011;
               hex_in = hex2;
               dp = dp_in[2];
            end
         default:
            begin
               an =  4'b0111;
               hex_in = hex3;
               dp = dp_in[3];
            end
       endcase

   // hex to seven-segment led display
   always @*
   begin
      case(hex_in)
         4'h0: sseg[6:0] = 7'b0000001;
         4'h1: sseg[6:0] = 7'b1001111;
         4'h2: sseg[6:0] = 7'b0010010;
         4'h3: sseg[6:0] = 7'b0000110;
         4'h4: sseg[6:0] = 7'b1001100;
         4'h5: sseg[6:0] = 7'b0100100;
         4'h6: sseg[6:0] = 7'b0100000;
         4'h7: sseg[6:0] = 7'b0001111;
         4'h8: sseg[6:0] = 7'b0000000;
         4'h9: sseg[6:0] = 7'b0000100;
         4'ha: sseg[6:0] = 7'b0001000;
         4'hb: sseg[6:0] = 7'b1100000;
         4'hc: sseg[6:0] = 7'b0110001;
         4'hd: sseg[6:0] = 7'b1000010;
         4'he: sseg[6:0] = 7'b0110000;
         4'hf: sseg[6:0] = 7'b0111000;
         default: sseg[6:0] = 7'b1111111;

     endcase
     sseg[7] = dp;
   end

endmodule
```

// TestBench for FSM module

```
`timescale 1 ns/10 ps
```

```
module FSM_Testbench();

//Declaring all the values
localparam T = 20; // Clock Period as specified


reg clk, user_reset, tick_reset; //Our clock signal and Reset Signal
wire tick;
reg input_bit;  // Display counter on Seven Segment Display

```

//UUT instantiation

```
FSM uut

(.CCLK(clk),.user_reset(user_reset),.input_bit(input_bit),.tick(tick),.t
ick_reset(tick_reset)); //Used to define the inputs for the testbench

always
begin
    clk = 1'b1; // Set the clock high
    #(T/2); //Half a period

    clk = 1'b0; //Set the Clock Low
    #(T/2); //For half a period
end

initial
begin
    user_reset = 1'b1;//Set the Reset High
    tick_reset = 1'b0;
    input_bit <= 1'b0;
    #(T*10);// Wait for 10 Periods
    user_reset = 1'b0; // Set the Reset to Low


    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); tick_reset <=1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); tick_reset <=1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;
    @(posedge clk); input_bit <= 1'b1;
    @(posedge clk); input_bit <= 1'b0;




end

 endmodule
```