CPE 166 / EEE270 ADVANCED LOGIC DESIGN LAB 3

Lab Session: Wednesday 2PM - 4:40PM

Student Name: Andrew Robertson

TABLE OF CONTENTS

INTRODUCTION

This lab is very beneficial to learning good approaches to problem solving. Throughout this lab I established a pattern of function recognition and compartmentalization, forming block diagrams (mostly part 2), coding to represent the block diagram structure, and test benching to ensure the output from each individual module is as expected. By doing this, issues are very easy to resolve, and the top-level module should be least of the worry overall. As outlined in the lab book, our goals for these exercises include:

- Structural design strategies using VHDL
- Using Vivado's built in Logic Analyzer
- Finite State machine design using VHDL
- VHDL Simulation
- Downloading VHDL designs to the board

Please note these objectives sometimes parallel those in the lab book rather than keeping on track with them, this is because of different instructions given by Dr. Jing Pang – author of the manual being used. These new instructions are not mentioned in the manual but rather emailed to us directly likely due to the use of a brand-new development board instead of the board the manual was originally written for.

PART 1

DESIGN PURPOSE:

Design and test a 1Hz clock that can count either up or down depending on user input. The output should be user friendly by displaying a recycling 0-9 count on a seven-segment display.

ENGINEERING DATA:

In this part, no block diagram was created due to the very simple nature of it. Stage one is to take in the very fast(100MHz) board clock signal and turn it into a 1 second (1Hz) clock. Professor Pang provided sample code for this task, all I needed to do was modify it a bit to get the code used in Figure 1. In summary, this code counts to 100,000,000. When the counter is below the half way point the output is low, when it is above or equal to the half way point the output is one. When the counter reaches 100,000,000 it is reset to 0. The output of this code can be seen in Figure 5. (Apologies, the input is rapidly changing and appears solid)

Stage two is to increment or decrement an internal variable used to store the value of the number on display. This will be a clock triggered event and will be based off the slower clock. Professor Pang again provided sample code for this task, all I needed to do was modify the bounds to get the code used in Figure 2. This is even simpler then the clock divider. When a positive edge clock signal is recognized we then check to see if we should count up or count down. If counting down, we check to see if the current count is zero. If it is we set it to 9, if it's not we decrement. For counting up we do the same thing but incrementing, and with an upper bound of 9 instead. The source code can be seen in Figure 2, and its output in Figure 6.

Stage three is to convert that value to a series of signals for a seven-segment display, allowing the user to see a visual representation of the stored value. This code, and the top-level module I wrote myself. No internal signals are needed for the seven-segment module because the output there was no intermediate value before the output that needed to be both read from and written to. A case statement addressing values 0-9 and it's corresponding seven segment signals is all that is needed. Figure 3 is the source code, and Figure 7 the output.

SOURCE CODE:

FIGURE 1
(CLOCK DIVIDER)

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_ARITH.ALL;
4   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5   ENTITY clock_div IS
6   PORT
7   (
8   clk : IN STD_LOGIC;
9   clkout : OUT STD_LOGIC);
10  END clock_div;
11
12  ARCHITECTURE behavior OF clock_div IS
13  --SIGNAL tmp : STD_LOGIC_VECTOR(26 downto 0) := "00000000000000000000000000";
14  SIGNAL tmp : integer := 0;
15  --SIGNAL cnt : STD_LOGIC_VECTOR(26 downto 0);
16  --SIGNAL clk2 : std_logic;
17
18  BEGIN
19
20      process (clk)
21      constant meg100 : integer := 100000000;
22      begin
23          if (rising_edge(clk)) then
24              if (tmp = meg100) then
25                  clkout <= '1';
26                  tmp <= 0;
27              elsif(tmp<(meg100/2 - 1)) then
28                  clkout <= '1';
29                  tmp <= tmp + 1;
30              else
31                  clkout <= '0';
32                  tmp <= tmp + 1;
33              end if;
34          end if;
35      end process;
36  --clkout <= clk2;
37  END behavior;
```

FIGURE 2
(COUNTER)

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

entity up_down_counter is
port(
clk, up_down : in std_logic;
cnt : out std_logic_vector(3 downto 0));
end up_down_counter;
architecture arch of up_down_counter is
signal tmp: std_logic_vector(3 downto 0):="0000";
begin

process (clk)
begin
if (rising_edge(clk)) then
    if (up_down = '1') then
        tmp <= tmp + 1;
        if (tmp = "1001") then
            tmp <= "0000";
        end if;
    else
        tmp <= tmp - 1;
        if (tmp = "0000") then
            tmp <= "1001";
        end if;
    end if;
end if;
end process;

cnt <= tmp;
end arch;
```

FIGURE 3
(SEVEN SEGMENT)

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_ARITH.ALL;
4   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5   ENTITY seven_segment IS
6   PORT
7   (
8   int_num : IN STD_LOGIC_VECTOR(3 downto 0);
9   seven_num : OUT STD_LOGIC_VECTOR(7 downto 0));
10  END seven_segment;
11
12  ARCHITECTURE behavior OF seven_segment IS
13  BEGIN
14
15      process (int_num)
16      begin
17      case int_num is
18        when "0000" =>    seven_num <= "11000000";      --dp, g=>a
19        when "0001" =>    seven_num <= "11111001";      --
20        when "0010" =>    seven_num <= "10100100";      --
21        when "0011" =>    seven_num <= "10110000";      --
22        when "0100" =>    seven_num <= "10011001";      --
23        when "0101" =>    seven_num <= "10010010";      --
24        when "0110" =>    seven_num <= "10000011";      --
25        when "0111" =>    seven_num <= "11111000";      --
26        when "1000" =>    seven_num <= "10000000";      --
27        when "1001" =>    seven_num <= "10011000";      --
28        when others =>    seven_num <= "10111111";      --
29      end case;
30      end process;
31  END behavior;
```

FIGURE 4
(TOP)

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

entity top is                                   --port list for top
port(clk, up_d:  in std_logic;
out_display : out std_logic_vector(7 downto 0);
board_display: out std_logic_vector(7 downto 0):="11111110");
end top;

architecture arch of top is                     --define structures (lower level)
                                                --define signals to connect together
signal clk2: std_logic:='0';
signal count: std_logic_vector(3 downto 0):="0000";

                                                --define components
component up_down_counter
    port(clk, up_down : in std_logic;
        cnt : out std_logic_vector(3 downto 0));
end component;

component clock_div
    port(clk : IN STD_LOGIC;
        clkout : OUT STD_LOGIC);
end component;

component seven_segment
    port(int_num : IN STD_LOGIC_VECTOR(3 downto 0);
        seven_num : OUT STD_LOGIC_VECTOR(7 downto 0));
end component;

begin                                           --define chip
up_down_counter1: up_down_counter port map(clk2,up_d,count);
clock_div1: clock_div port map(clk, clk2);
seven_segment1: seven_segment port map(count, out_display);
end arch;
```

SIMULATION WAVEFORM(S):

FIGURE 5
(CLOCK DIVIDER)

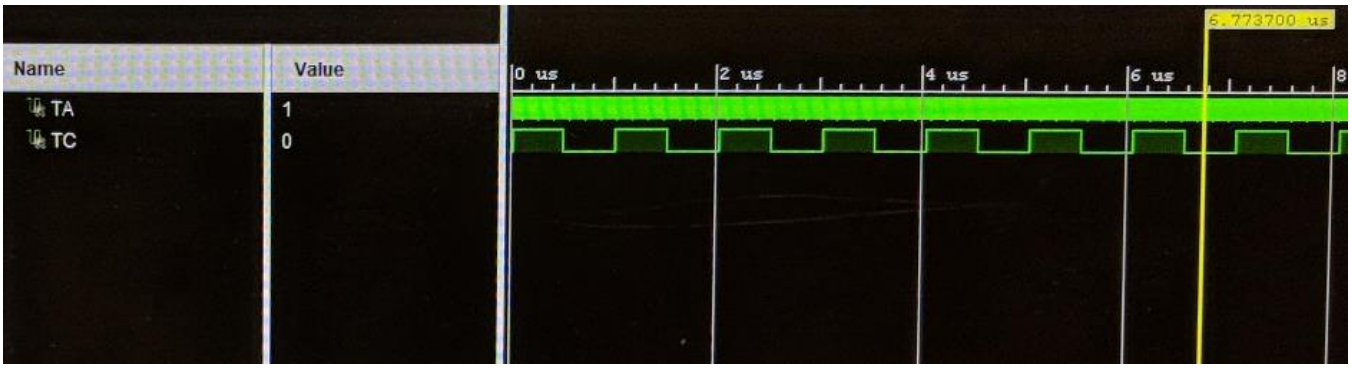


FIGURE 6
(COUNTER)

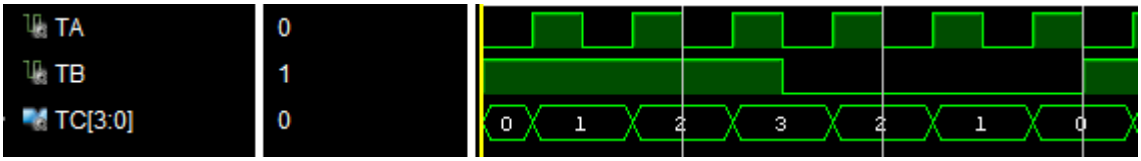


FIGURE 7
(SEVEN SEGMENT)

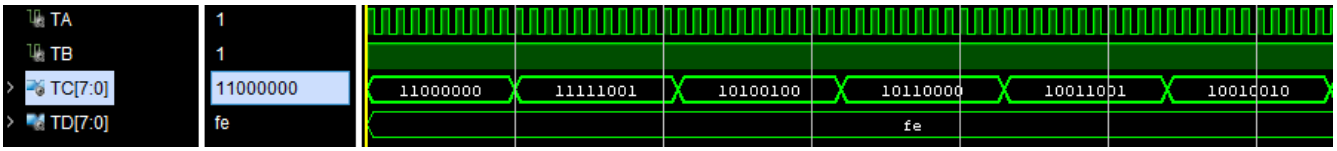
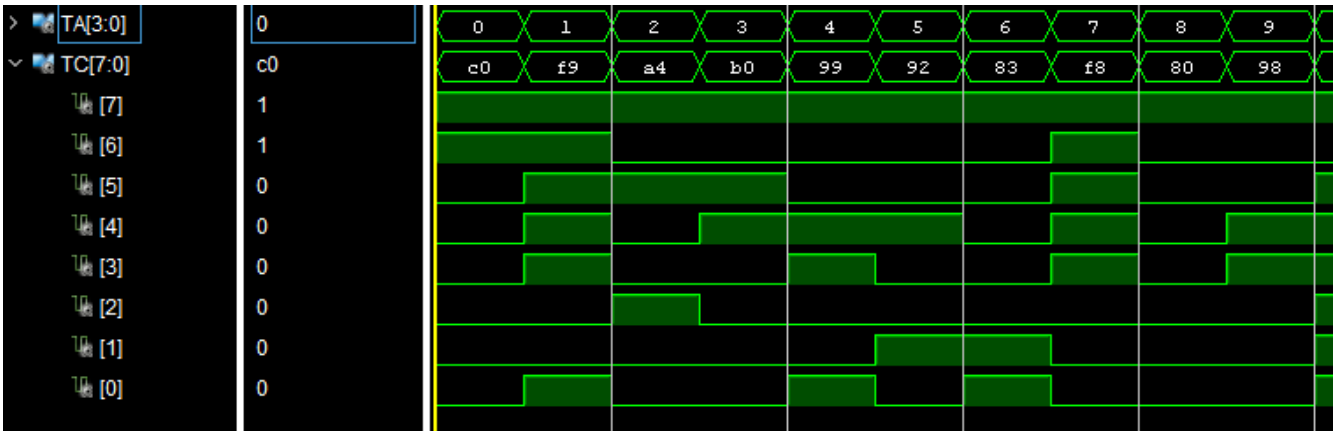


FIGURE 8
(TOP)



RESULT DISCUSSION:

With VHDL being brand new to me at this point, the only real issues I had were all syntax related. The idea was very straight forward but turning that idea into code was very difficult at times. Although a hierarchical design was not required, I felt it would be more beneficial to write for two reasons. Reason one is for the practice while the task was still easy, so it wouldn't be an additional pain later. Reason two is for modularity and ease of testing each individual stage. Knowing that each piece works as expected, to then seeing issues when putting a top-level module together would usually mean the issue is with the way the top and it's testbench is linking the pieces together vs the submodule logic being a problem. For me, this idea laid the groundwork for the rest of the lab sections.

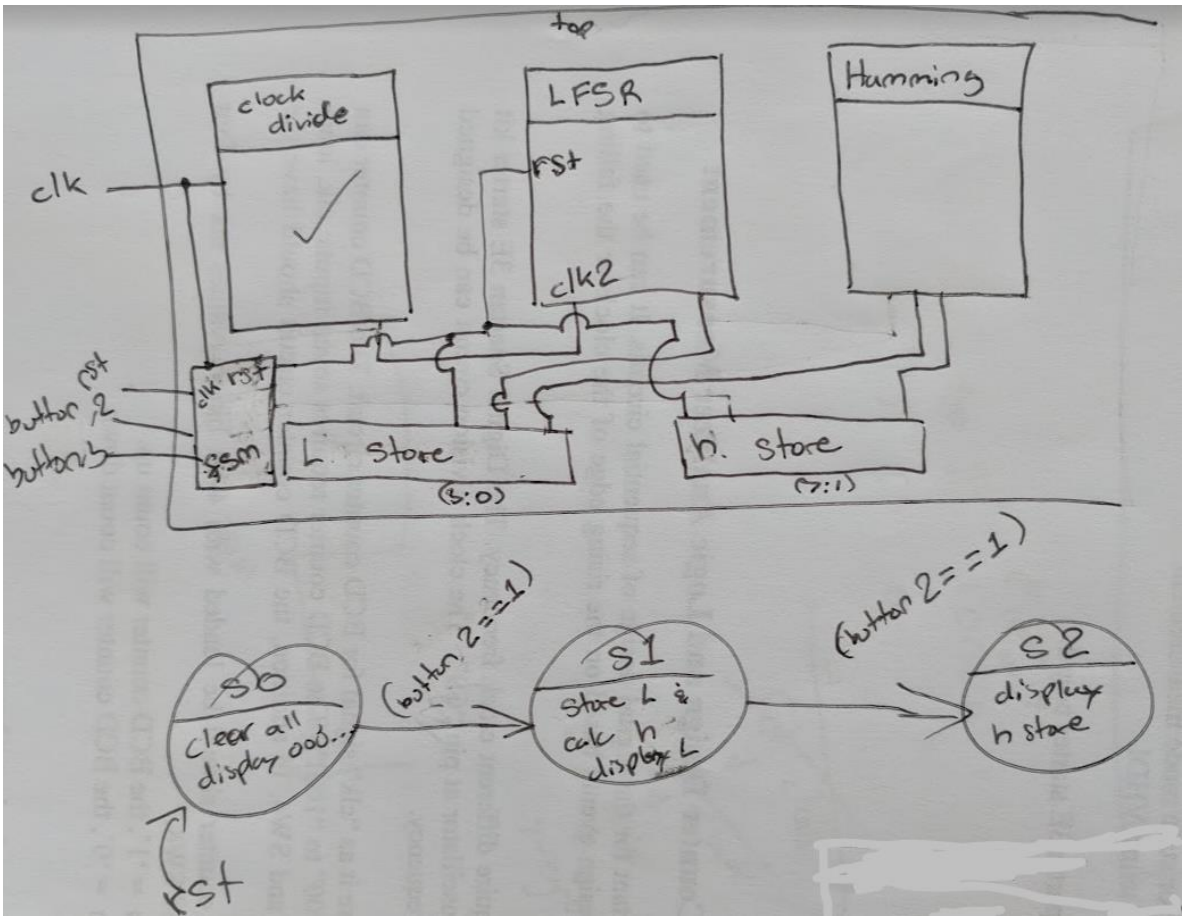
PART 2

DESIGN PURPOSE:

Design and build a circuit that will generate and display the output from a 4 bit LFSR. Using that output, then generate the corresponding hamming code and display it instead.

ENGINEERING DATA:

Below is the block diagram and rough state machine I created before doing anything else with this design. By spending a bit of time putting this together it become very clear what modules would be needed and how they worked together. It's also very easy to see which modules are sequential or combinational this way too, anything with a clock as an input is going to be sequential, and combinational if not. This diagram also served another purpose as a sort of checklist for modules written vs those that need to be written. Last, the diagram made the top module the easiest of all simply as a visual aid for the signals (wires) needed to connect the modules together. (Note the second Button 2 == 1 should be Button 3 == 1, also no load signal is shown from the FSM but it is implemented)

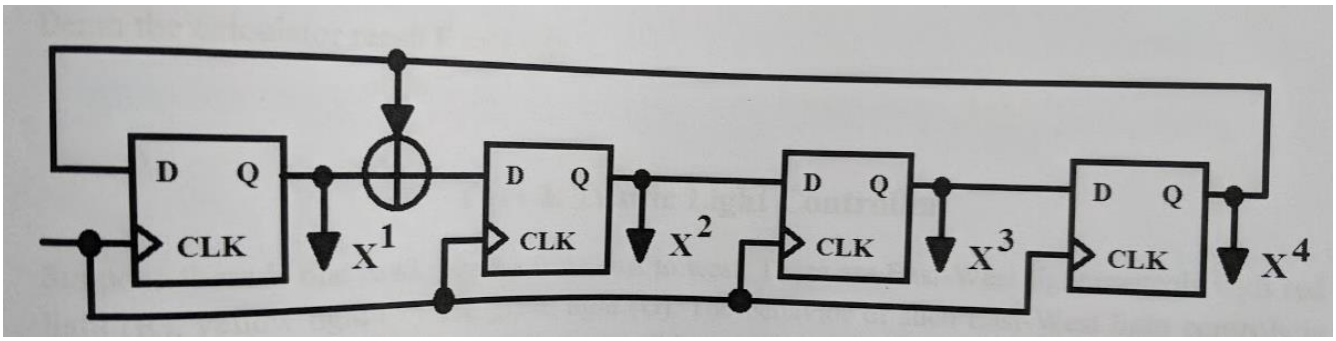


Clock Divider –

This was the exact same clock divider used in Part 1, since the output clock is to be 1Hz again, nothing needed to be changed. Please refer to figures 1 and 5 for the source code and its output, respectively.

LFSR –

Source code is Figure 13, output is 21. Using the given diagram below, a sequential circuit needed to be created to serve as both the intermittent output to the LCD's, and the input to the hamming module. This is just a normal shift register, but with the right most flip flop feeding into the left most with a XOR gate thrown in.



Hamming –

Source code is Figure 12, output is 20. A good reference behind the idea of this code is here https://en.wikipedia.org/wiki/Hamming_code. It is a form of error detection based off the creation of parity bits for a certain pattern of bits from the nibble to be checked. The parity values can be created with XOR gates, and as such the only knowledge needed to accomplish this is the combination of input bits to create each parity bit.

FSM –

Source code is Figure 14, output is 19. Please note, an additional state was added beyond what is shown in the block diagram. What I wanted to be able to do was clearly see the LFSR output before moving on the hamming output so hand calculations can be done to double check the output. To achieve this, I added a state that is stepped in to by flipping button 2 down. This state had no other purpose but to turn the load signal off for DFFA so it can be viewed.

H store (DFFB) –

Source code is Figure 11, output is 18. The only difference between DFFA and DFFB is the bit width. They both have load and clear functionality so reset can clear the registers and load can control when they accept new signals and when they are latched.

L store (DFFA) –

Nearly identical to DFFB. Source code is Figure 10, output is 17.

Top –

Source code is Figure 15, output is 23. The only notable thing that needed to be done here that may not be obvious from the block diagram is concatenating 0's to the start of the LFSR output to be assigned to the 7 bit LED output.

SOURCE CODE:

FIGURE 9
(DFFA)

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity dffa is
5
6   port ( load, clear, clk  : IN std_logic;
7          din : IN std_logic_vector(3 downto 0);
8          qout : OUT std_logic_vector(3 downto 0) );
9
10  end dffa;
11
12  architecture imp of dffa is
13  begin
14
15      process (clk)
16      begin
17          if (rising_edge(clk)) then
18              if (clear = '1') then
19              qout <= "0000";
20              end if;
21              if (load = '1') then
22              qout <= din;
23              end if;
24          end if;
25      end process;
26
27  end imp;
```

FIGURE 10
(DFFB)

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity dffb is
5
6  port ( load, clear, clk  : IN std_logic;
7         din : IN std_logic_vector(7 downto 1);
8        qout : OUT std_logic_vector(7 downto 1) );
9
10 end dffb;
11
12 architecture imp of dffb is
13 begin
14     process (clk)
15         begin
16         if (rising_edge(clk)) then
17             if (clear = '1') then
18             qout <= "0000000";
19             end if;
20             if (load = '1') then
21             qout <= din;
22             end if;
23         end if;
24     end process;
25
26 end imp;
```

FIGURE 11
(HAMMING)

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   ENTITY hamming IS
4   PORT(
5           four_bit_in : in std_logic_vector(3 downto 0);
6           seven_bit_out : out std_logic_vector(6 downto 0)
7   );
8   END hamming;
9
10  ARCHITECTURE ham OF hamming IS
11  signal parity : std_logic_vector(2 downto 0);
12  BEGIN
13
14
15  parity(0) <= four_bit_in(0) xor four_bit_in(1) xor four_bit_in(3);
16  parity(1) <= four_bit_in(0) xor four_bit_in(2) xor four_bit_in(3);
17  parity(2) <= four_bit_in(1) xor four_bit_in(2) xor four_bit_in(3);
18
19  seven_bit_out <= four_bit_in(3) & four_bit_in(2) & four_bit_in(1) &
20                   parity(2) & four_bit_in(0) & parity(1) & parity(0);
21
22  END ham;
```

FIGURE 12
(LFSR)

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  ENTITY lfsr_circuit IS
4  PORT (
5      CLK: IN STD_LOGIC;
6      rst: in std_logic;
7      lfsr_out: OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
8      );
9   end entity lfsr_circuit;
10
11  ARCHITECTURE design OF lfsr_circuit IS
12     signal x : std_logic_vector(3 downto 0) :="0001";
13 begin
14     lfsr_out(0) <= x(3);
15     lfsr_out(1) <= x(2);
16     lfsr_out(2) <= x(1);
17     lfsr_out(3) <= x(0);
18
19     process(clk)
20     begin
21        if rising_edge(clk) then
22           if (rst ='1') then
23              x <= "1000";
24           else
25              x(0) <= x(3);
26              x(1) <= x(0) XOR x(3);
27              x(2) <= x(1);
28              x(3) <= x(2);
29           end if;
30        end if;
31     end process;
32  end design;
```

FIGURE 13
(MUX)

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4  use IEEE.std_logic_arith.all;
5
6  entity MUX is
7
8  port (
9          S: IN std_logic;
10         D0 : IN STD_LOGIC_VECTOR(3 downto 0);
11         D1 : IN STD_LOGIC_VECTOR(7 downto 1);
12         muxOut : OUT std_logic_VECTOR(7 downto 1)
13       );
14 end MUX;
15
16 architecture beh of MUX is
17     begin
18
19     process (D1, D0, S)
20         begin
21             if(S = '1') then
22                 muxOut <= D1;
23             else
24                 muxOut <= "000" & D0;
25             end if;
26
27     end process;
28 end beh;
```

FIGURE 14
(FSM)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY fsm IS
PORT(
        SW0, SW1, SW2, CLK : in std_logic;
        CLR, LDA, LDB, MUX : out std_logic
);
END fsm;

ARCHITECTURE finite OF fsm IS
TYPE state IS (s0, s1, s2, s3);
signal cs, ns: state := s0;

BEGIN

process (cs, SW0, SW1, SW2)      --ns block
begin
case cs is
  when s0 =>
    if (SW1 = '1') then
      ns <= s1;
    else
      ns <= s0;
    end if;
  when s1 =>
    if (SW1 = '0') then
      ns <= s2;
    else
      ns <= s1;
    end if;
  when s2 =>
    if (SW2 = '1') then
      ns <= s3;
    else
      ns <= s2;
    end if;
  when s3 =>
    if (SW0 = '1') then
      ns <= s0;
    else
      ns <= s3;
    end if;
  when others => ns <= s0;
end case;
end process;

process (clk)                    --cs block
begin
if(rising_edge(clk)) then
    if (SW0 = '1') then
        cs <= s0;
    else
        cs <= ns;
    end if;
end if;
end process;
```

```

process (cs)                --signal block
begin
case cs is
  when s0 =>
    CLR <='1';
    MUX <='0';
    LDA <='0';
    LDB <='0';
  when s1 =>
    CLR <='0';
    MUX <='0';
    LDA <='1';
    LDB <='0';
  when s2 =>
    CLR <='0';
    MUX <='0';
    LDA <='0';
    LDB <='0';
  when s3 =>
    CLR <='0';
    MUX <='1';
    LDA <='0';
    LDB <='1';
  when others =>
    CLR <='1';
    MUX <='0';
    LDA <='0';
    LDB <='0';
end case;
end process;

END finite;
```

FIGURE 15
(TOP)

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_ARITH.ALL;
4   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6   entity top is                                   --port list for top
7   port(clk, switch0, switch1, switch2:  in std_logic;
8   out_display : out std_logic_vector(7 downto 1);
9   end top;
10
11  architecture arch of top is                     --define structures (lower level)
12                                                  --define signals to connect together
13  signal clk2: std_logic;
14  signal reset: std_logic;
15  signal loada: std_logic;
16  signal loadb: std_logic;
17  signal mux_s: std_logic;
18  signal lfsr_o: std_logic_vector(3 downto 0);
19  signal hamming_out: std_logic_vector(7 downto 1);
20  signal dffa_out: std_logic_vector(3 downto 0);
21  signal dffb_out: std_logic_vector(7 downto 1);
22
23
24                                                  --define components
25
26  component dffa is
27
28  port ( load, clear, clk  : IN std_logic;
29           din : IN std_logic_vector(3 downto 0);
30         qout : OUT std_logic_vector(3 downto 0) );
31
32  end component;
33
34  component dffb is
35
36  port ( load, clear, clk  : IN std_logic;
37           din : IN std_logic_vector(7 downto 1);
38         qout : OUT std_logic_vector(7 downto 1) );
39
40  end component;
41
42  component clock_div IS
43
44  PORT ( clk : IN STD_LOGIC;
45      clkout : OUT STD_LOGIC);
46
47  END component;
48
49  component fsm IS
50  PORT(
51          SW0, SW1, SW2, CLK : in std_logic;
52          CLR, LDA, LDB, MUX : out std_logic
53  );
54  END component;
```

```
55
56  component hamming IS
57  PORT(
58          four_bit_in : in std_logic_vector(3 downto 0);
59          seven_bit_out : out std_logic_vector(6 downto 0)
60  );
61  END component;
62
63  component lfsr_circuit IS
64  PORT (
65      CLK: IN STD_LOGIC;
66      rst: in std_logic;
67      lfsr_out: OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
68      );
69  end component;
70
71  component MUX is
72
73  port (
74          S: IN std_logic;
75          D0 : IN STD_LOGIC_VECTOR(4 downto 1);
76          D1 : IN STD_LOGIC_VECTOR(7 downto 1);
77          muxOut : OUT std_logic_VECTOR(7 downto 1)
78         );
79  end component;
80
81
82  begin                                          --define chip
83  dffa1: dffa port map(loada, reset, clk2, lfsr_o, dffa_out);
84  dffb1: dffb port map(loadb, reset, clk2, hamming_out, dffb_out);
85  clock_div1: clock_div port map(clk, clk2);
86  fsm1: fsm port map(switch0, switch1, switch2, clk2, reset, loada, loadb, mux_s);
87  hamming1: hamming port map(dffa_out, hamming_out);
88  lfsr_circuit1: lfsr_circuit port map(clk2, reset, lfsr_o);
89  MUX1: MUX port map(mux_s, dffa_out, dffb_out, out_display);
90  end arch;
```

FIGURE 16
(ENABLED LINES IN THE CONSTRAINT FILE)

```
7   set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
13  set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { switch2 }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
14  set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { switch1 }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
15  set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { switch0 }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
33  set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { out_display[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
34  set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { out_display[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
35  set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { out_display[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
36  set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { out_display[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
37  set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { out_display[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
38  set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { out_display[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
39  set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { out_display[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
```

SIMULATION WAVEFORM(S):

FIGURE 17
(DFFA)

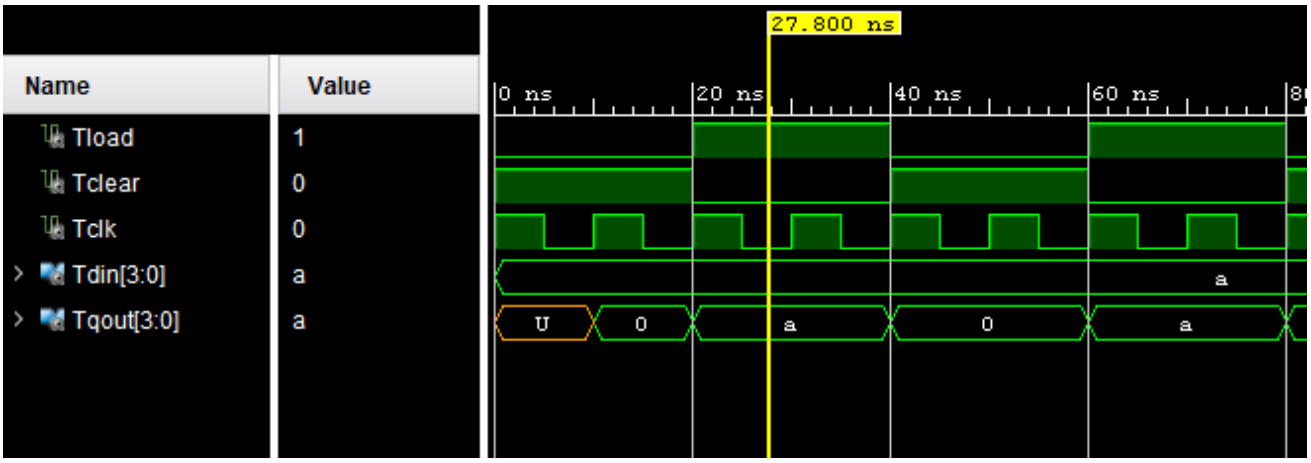


FIGURE 18
(DFFB)

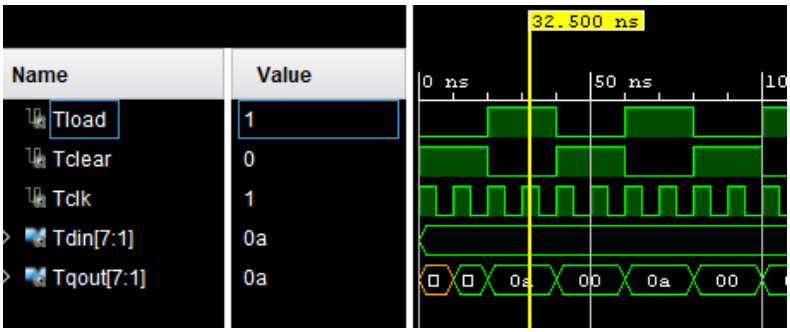


FIGURE 19
(FINITE STATE MACHINE)

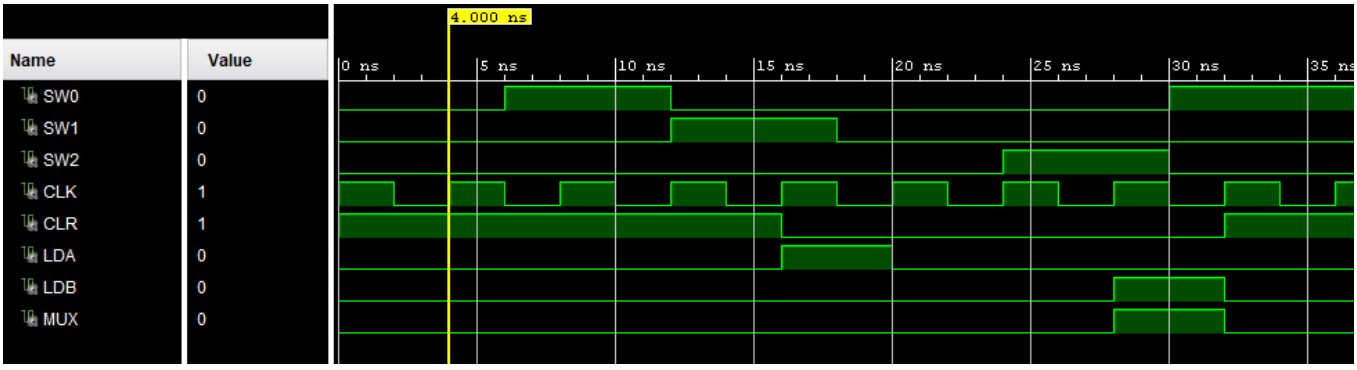


FIGURE 20
(HAMMING)

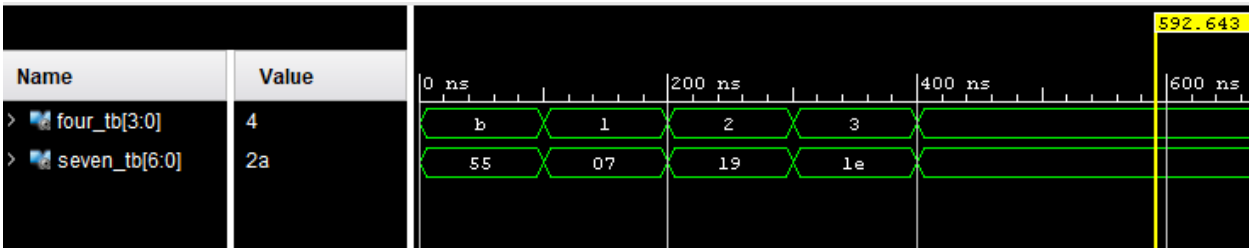


FIGURE 21
(LFSR)

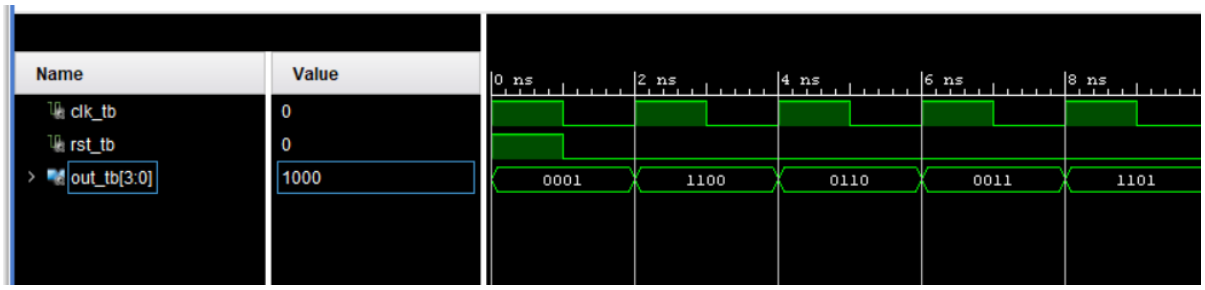


FIGURE 22
(MUX)

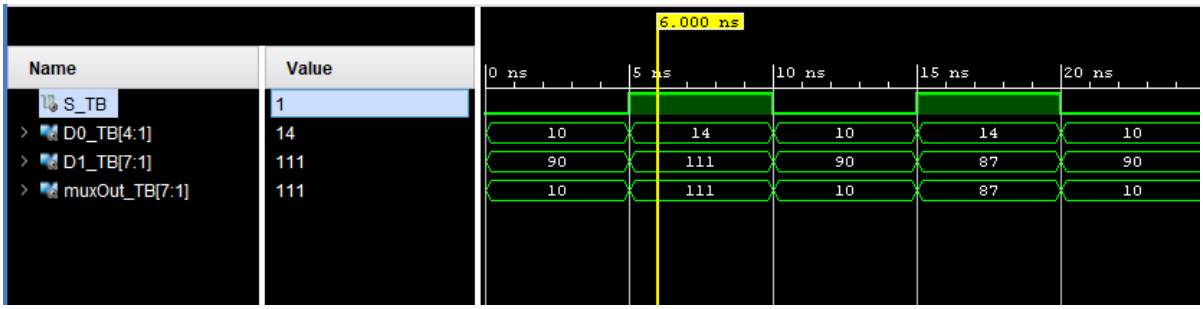
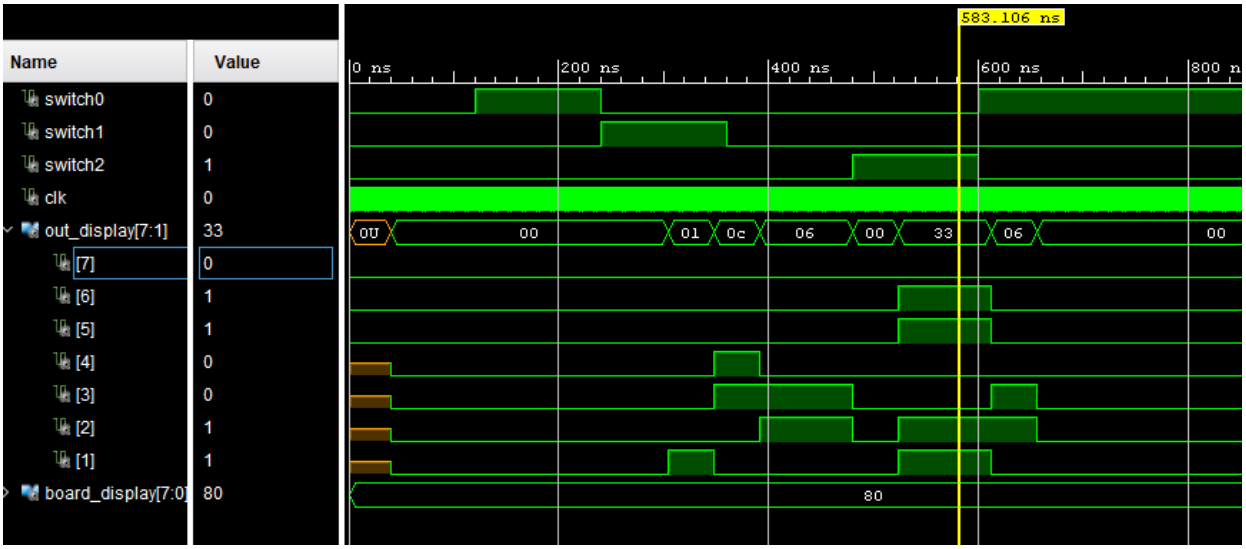


FIGURE 23
(TOP)



RESULT DISCUSSION:

The block diagram was a life saver in many ways for this part of lab 3. Having it near to keep me on track every step of the way is much more useful than it sounds. It served as a visual aid for the top-level module, a checklist for writing sub modules, and as a motivational tool when checking completed parts off. This part is where most of time was spent in this lab. VHDL was still new, giving way to an interesting new set of syntax errors, and the level of complexity was much greater than the last part. Given this information, I feel this was where I learned the most out of this lab.

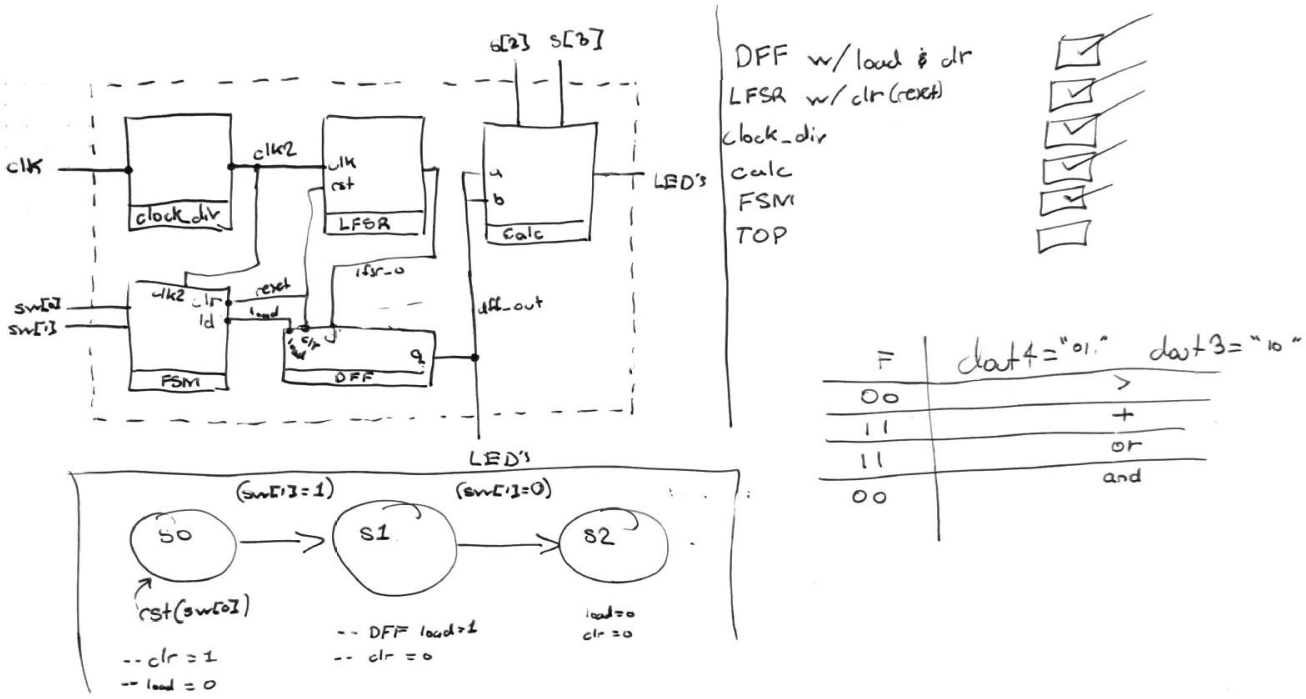
PART 3

DESIGN PURPOSE:

Design a calculator with functions to process two values in the following ways: find which is greater, add, logic OR, logic AND. The input for the calculator will be from a 4-bit LFSR driven by a 1Hz clock. Use one switch for reset and another switch to allow for first allowing LFSR input, and then locking the value in. Two more switches are to be used to control the calculator function.

ENGINEERING DATA:

This was my quick block diagram to get the project started:



From the given description I decided to use clock divider, LFSR, calculator, DFF, and FSM modules to meet the given specs. The clock division requirements were the same as our past parts, so I was able to reuse the same code. Please refer to figures 1 and 5 for the source code and its output, respectively. The LFSR also had the same spec requirement as before. Source code is Figure 13, output is 21. Last, the DFF also had the same requirement as a past lab part. Source code is Figure 10, output is 17. The three new items are the finite state machine, calculator, and top module.

Calculator – Achieved by using a case statement for the combinations of B3 and B4 possible seen below.

I learned I was unable to use concatenation in the case variable portion, so a vector signal needed to be created that performed the concatenation early. I also learned that the sensitivity list for the process with this case statement needed to watch the new vector signal instead of the B3 and B4 variables directly due to delay.

Table 3-1. Calculator function

Inputs		Output Calculator Function
b3	b4	
0	0	F = dout4 > dout3
0	1	F = dout4 + dout3
1	0	F = dout4 or dout3
1	1	F = dout4 and dout3

Finite State Machine –

This was essentially the same as part two of this lab, only minor changes needed to be made like the number of switches to use and using two sets of outputs since we want to see LFSR and calculator output at the same time.

Top –

Again, very near the top used for part two. Simply added the two new components listed above and removed all the unused ones like the MUX. Last was to make the ports reflect the new block diagram.

SOURCE CODE:

FIGURE 24
(CALC)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

ENTITY calc IS
PORT
(
B3, B4 : IN STD_LOGIC;
dout3, dout4 : IN STD_LOGIC_vector(1 downto 0);
F : OUT STD_LOGIC_vector(1 downto 0)
);
END calc;

ARCHITECTURE behavior OF calc IS
signal temp : std_logic_vector(1 downto 0);
BEGIN
    temp <= B3 & B4;
    process (temp, dout3, dout4)
    begin
        case (temp) is
            when "00" => if (dout4 > dout3) then
                         F <= "01";
                         else
                         F <= "00";
                         end if;
            when "01" => F <= (dout4 + dout3);
            when "10" => F <= (dout4 or dout3);
            when "11" => F <= (dout4 and dout3);
            when others => F <= "00";
        end case;
    end process;
END behavior;
```

FIGURE 25
(FSM)

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  ENTITY fsm IS
4  PORT(
5          SW0, SW1, CLK : in std_logic;
6          CLR, LD : out std_logic
7  );
8  END fsm;
9
10 ARCHITECTURE finite OF fsm IS
11 TYPE state IS (s0, s1, s2);
12 signal cs, ns: state := s0;
13
14 BEGIN
15
16 process (cs, SW0, SW1)
17 begin
18 case cs is
19   when s0 =>
20     if (SW1 = '1') then
21       ns <= s1;
22     else
23       ns <= s0;
24     end if;
25   when s1 =>
26     if (SW1 = '0') then
27       ns <= s2;
28     else
29       ns <= s1;
30     end if;
31   when s2 =>
32       ns <= s2;
33   when others => ns <= s0;
34 end case;
35 end process;
36
37 process (clk)
38 begin
39 if(rising_edge(clk)) then
40     if (SW0 = '1') then
41         cs <= s0;
42     else
43         cs <= ns;
44     end if;
45 end if;
46 end process;
47
48 process (cs)
49 begin
50 case cs is
51   when s0 =>
52     CLR <='1';
53     LD  <='0';
54   when s1 =>
55     CLR <='0';
56     LD  <='1';
57   when s2 =>
58     CLR <='0';
59     LD  <='0';
60   when others =>
61     CLR <='1';
62     LD  <='0';
63 end case;
64 end process;
65
66 END finite;
```

FIGURE 26
(TOP)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;

entity top is                                   --port list for top
port(clk, switch0, switch1, switch2, switch3:  in std_logic;
lfsr_display : out std_logic_vector(3 downto 0);
calc_display : out std_logic_vector(1 downto 0));
end top;

architecture arch of top is                     --define structures (lower level)
                                                --define signals to connect together
signal clk2: std_logic;
signal reset: std_logic;
signal load: std_logic;
signal lfsr_o: std_logic_vector(3 downto 0);
signal dff_out: std_logic_vector(3 downto 0);

                                                --define components

component dffa is

port ( load, clear, clk  : IN std_logic;
         din : IN std_logic_vector(3 downto 0);
        qout : OUT std_logic_vector(3 downto 0) );

end component;

component clock_div IS

PORT ( clk : IN STD_LOGIC;
     clkout : OUT STD_LOGIC);
```

```
END component;

component fsm IS
PORT(
        SW0, SW1, CLK : in std_logic;
        CLR, LD : out std_logic
);
END component;

component lfsr_circuit IS
PORT (
    CLK: IN STD_LOGIC;
    rst: in std_logic;
    lfsr_out: OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
    );
end component;

component calc IS
PORT
(
B3, B4 : IN STD_LOGIC;
dout3, dout4 : IN STD_LOGIC_vector(1 downto 0);
F : OUT STD_LOGIC_vector(1 downto 0)
);
END component;


begin
lfsr_display <= dff_out;                                    --define chip
dffa1: dffa port map(load, reset, clk2, lfsr_o, dff_out);
clock_div1: clock_div port map(clk, clk2);
fsm1: fsm port map(switch0, switch1, clk2, reset, load);
lfsr_circuit1: lfsr_circuit port map(clk2, reset, lfsr_o);
calc1: calc port map(switch2, switch3, dff_out(1 downto 0), dff_out(3 downto 2), calc_display);

end arch;
```

FIGURE 27
(ENABLED ITEMS IN CONSTRAINT FILE)

```
7   set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
13  set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { switch3 }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
14  set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { switch2 }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
15  set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { switch1 }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
16  set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { switch0 }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
33  set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { lfsr_display[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
34  set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { lfsr_display[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
35  set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { lfsr_display[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
36  set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { lfsr_display[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
47  set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { calc_display[0] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
48  set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { calc_display[1] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]
```

SIMULATION WAVEFORM(S):

FIGURE 28
(CALC)

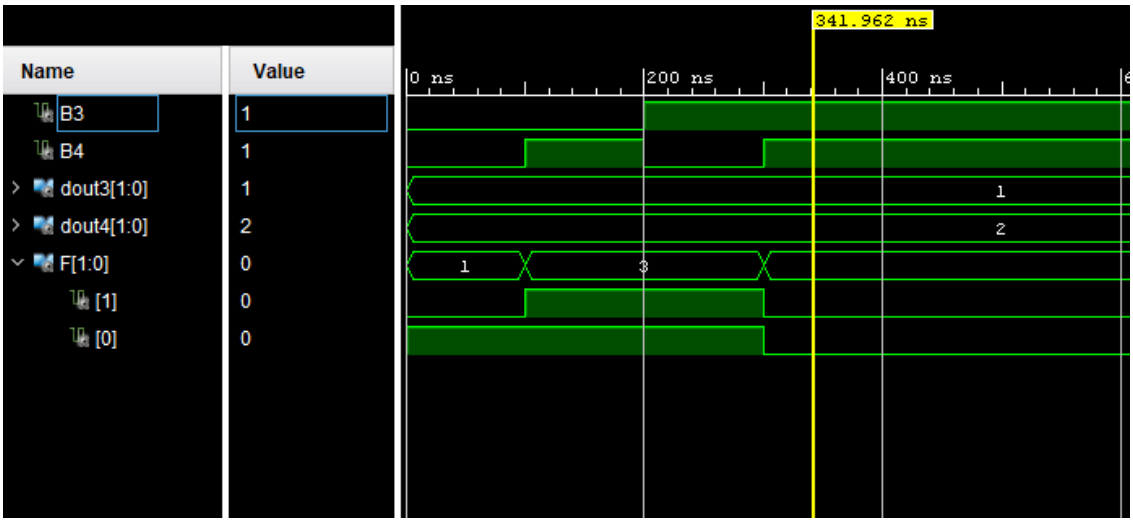


FIGURE 29
(FSM)

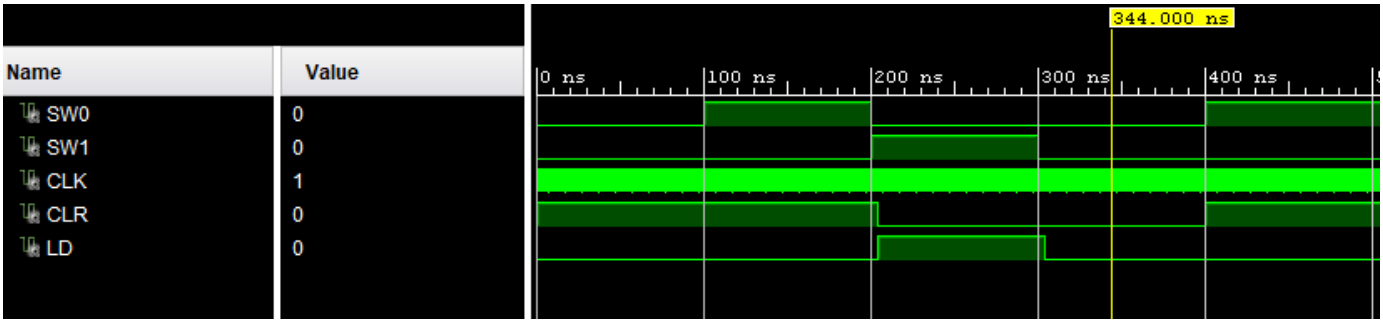
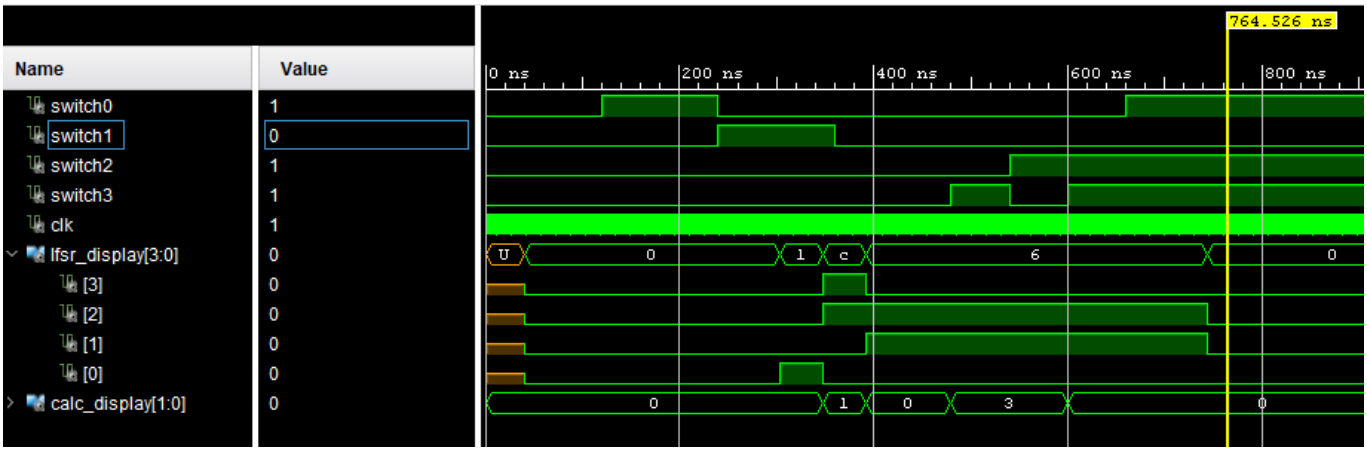


FIGURE 30
(TOP)



RESULT DISCUSSION:

I am very glad this part of the last was completed last. The hierarchical design of separate modules used in all the previous parts made this one a one-day project simply due to the amount of reuse that was possible. Unexpectedly, the most difficult module was the calculator due to nuanced reasons I didn't previously think of and minor syntax issues. The nuanced issues was a lack of proper library use to accommodate comparison and addition. The syntax issue was one small enough for Vivado not to pick up directly, but rather in the form of telling me my top module was not a type despite it being defined and successfully synthesizing. It turned out I was missing a right parenthesis after the last port declaration in my port list. Since this was a vector, a quick glance made it appear the port list was fine since it began in a left parenthesis, ended in a right with a semicolon after, and had semicolons following all other lines.

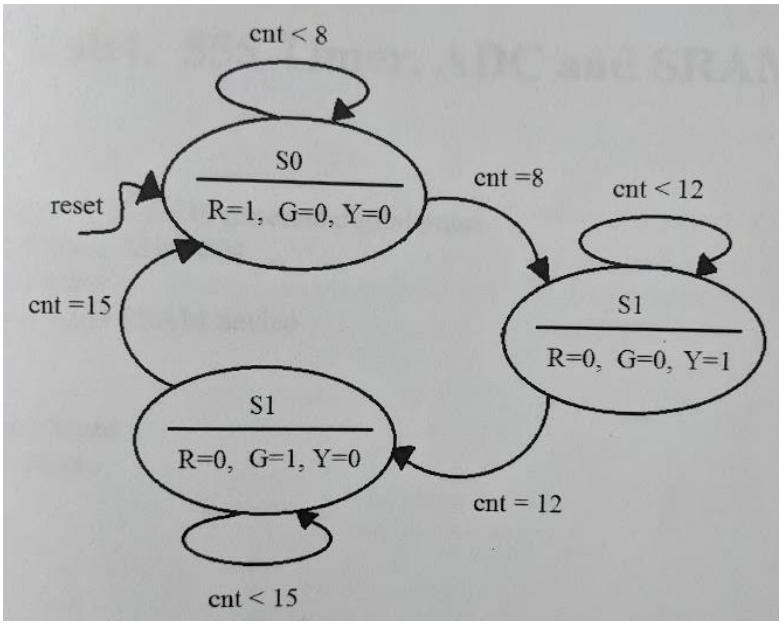
PART 4

DESIGN PURPOSE:

Design a counter based finite state machine that models a stoplight.

ENGINEERING DATA:

This part of the lab was insanely straight forward. Not only did we only work with a finite state machine and it's testbench, but Professor Pang also provided the code needed with only minor changes to be our responsibility. Due to this, I won't have much to report for this section. Below is the figure given as a visual aid in the lab manual:



With the code given, all that we needed to do was change the times that the program progressed from one state to another. The source code provided, with this minor change can be seen in figure 31. Its output is in figure 32. The only differentiator that this state machine uses in comparison to what we have done so far is use a counter as a trigger from changing state instead of a switch or button.

SOURCE CODE:

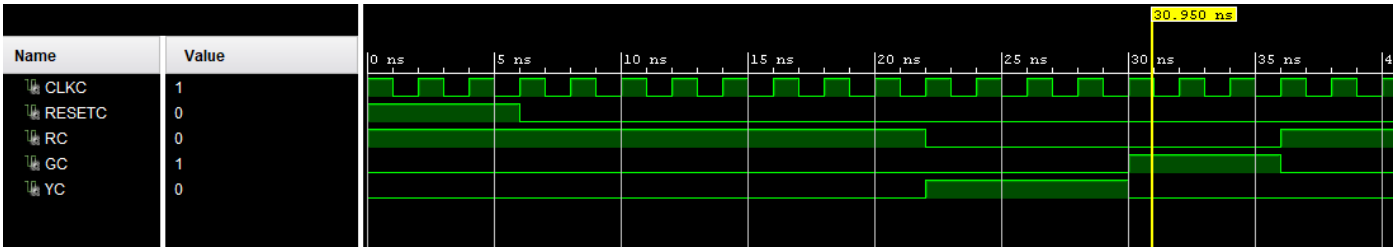
FIGURE 31
(FSM)

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.std_logic_unsigned.all;
4   use IEEE.std_logic_arith.all;
5
6   entity fsm is
7
8   port ( clk, reset : IN std_logic;
9          R,G,Y : OUT std_logic );
10
11  end fsm;
12
13  architecture beh of fsm is
14
15  type state_type is (s1,s2,s3);
16
17  signal state: state_type ;
18
19  signal count : STD_LOGIC_VECTOR(3 downto 0);
20  begin
21
22
23  process (clk,reset)
24  begin
25  if (reset ='1') then
26     count <= "0000";
27     state <=s1;
28  elsif (clk='1' and clk'event) then
29    case state is
30    when s1 =>
31          count <= count + 1;
32          if (count < 8) then
33            state <= s1;
34          elsif (count = 8) then
35            state <= s2;
36          end if;
37
38    when s2 =>
39          count <= count + 1;
40          if (count < 12) then
41            state <= s2;
42          elsif (count = 12) then
43            state <= s3;
44          end if;
45
```

```
46      when s3 =>
47            count <= count + 1;
48            if (count < 15) then
49              state <= s3;
50            elsif (count = 15) then
51              state <= s1;
52            end if;
53      when others => state <= s1;
54      end case;
55
56    end if;
57
58    end process;
59
60    process(state)
61
62     begin
63
64    case state is
65
66      when s1 =>
67            R <='1';
68            G <='0';
69            Y <='0';
70      when s2 =>
71            R <='0';
72            G <='0';
73            Y <='1';
74
75      when s3 =>
76            R <='0';
77            G <='1';
78            Y <='0';
79
80      when others=>
81            R <='0';
82            G <='0';
83            Y <='0';
84
85    end case;
86
87    end process;
88
89    end beh;
```

SIMULATION WAVEFORM(S):

FIGURE 32



RESULT DISCUSSION:

This lab section, including writing the testbench, took maybe an hour start to finish. It would not have been difficult to design from scratch, but I believe Professor Pang was feeling how pressured most other students are right now and wanted to make this section a walk in the park.

PART 5

DESIGN PURPOSE:

Design RAM that has controls for reading and writing to 16 addresses. Write “1010” to each address then read that data from the same addresses.

ENGINEERING DATA:

Again, everything was handed to us for this section. In an email Professor Pang provided the code seen in figures 33, 34 and 35 minus some small changes that needed to be made. This time I was slightly relieved to get this email, the design instructions were very brief and I was somewhat unclear what to do. Given this code, I now have an idea how to approach something like this in the future.

The code originally provided had an 8 bit data register but we were told to store a four bit value of 1010. There were two ways to accomplish this so I chose the slightly more involved way to still force myself to learn more about this code. The first way is to simply concatenate 4 zeroes before 1010 in top, and everything else could be left untouched. The way I chose to attack this is to change the data register to 4 bits wide instead. This required changing the port declarations as well as the data being written in top. From doing this I learned that the conv_integer() function outputs 8 bits. This meant using (3 downto 0) to access only its lower 4 bits for the data segment.

Figure 36 displays the output seen when running the simulation feature in Vivado as we have been so far, while figure 37 shows the captured data in Vivado’s built in logic analyzer (ILA).

SOURCE CODE:

FIGURE 33
(SRAM)

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity sram is
6   port (
7           address :in    std_logic_vector ( 3 downto 0);
8           din     :in    std_logic_vector ( 3 downto 0);
9           dout    :out   std_logic_vector ( 3 downto 0);
10          cs      :in    std_logic;
11          we      :in    std_logic;
12          oe      :in    std_logic
13      );
14  end sram;
15
16  architecture beh_sram of sram is
17
18  type memory is array (0 to 15)of std_logic_vector (7 downto 0);
19  signal mem : memory ;
20
21  begin
22
23
24        MEM_WRITE:
25        process (address, din, cs, we)
26        begin
27           if (cs = '1' and we = '1') then
28              mem(conv_integer(address)) <= "0000" & din;
29           end if;
30        end process;
31
32
33        MEM_READ:
34       process (address, cs, we, oe, mem)
35       begin
36        if (cs = '1' and we = '0' and oe = '1')  then
37          dout <= mem(conv_integer(address))(3 downto 0);
38        else
39          dout <=  (others => 'Z' );
40
41        end if;
42        end process;
43
44    end beh_sram;
```

FIGURE 34
(FSM)

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.std_logic_unsigned.all;
4   use IEEE.std_logic_arith.all;
5
6   entity sram_fsm is
7     port ( clk, reset : IN    std_logic;
8            address : OUT     std_logic_vector(3 downto 0);
9            cs, we, oe: OUT   std_logic );
10  end sram_fsm;
11
12  architecture fsm_beh of sram_fsm is
13    type state_type is (idle, s1,s2,s3,s4);
14    signal state: state_type ;
15    signal cnt: std_logic_vector(3 downto 0);
16  begin
17
18    cs <= '1';
19    address <= cnt;
20
21    process (clk,reset)
22    begin
23      if (reset ='1') then
24        state   <= idle;
25        cnt     <= "0000";
26
27      elsif (clk='1' and clk'event) then
28        case state is
29      when idle =>
30          state <= s1;
31          cnt   <= "0000";
32
33          when s1 =>
34          state <= s2;
35          cnt   <= "0000";
36
37          when s2 =>
```

```
38          cnt   <=   cnt + 1;
39          if (cnt  < 15) then
40             state <= s2;
41          else
42             state <= s3;
43          end if;
44
45          when s3 =>   state <= s4;
46             cnt  <= "0000";
47
48       when s4 =>
49          cnt  <= cnt + 1;
50          state <= s4;
51
52
53       when others =>
54          cnt     <= "0000";
55          state   <= s1;
56
57        end case;
58       end if;
59    end process;
60
61    process(state)
62    begin
63       case state is
64          when idle => we <= '0'; oe <= '0';
65          when s1 => we <= '1'; oe <= '0';
66          when s2 => we <= '1'; oe <= '0';
67          when s3 => we <= '0'; oe <= '1';
68          when s4 => we <= '0'; oe <= '1';
69          when others => we <= '0'; oe <= '0';
70       end case;
71    end process;
72
73 end fsm_beh;
```

FIGURE 35
(TOP)

```
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4   use WORK.ALL;
5
6   entity top_sram is
7    Port ( clk, reset: in std_logic;
8         data:  inout std_logic_vector(3 downto 0)
9    );
10  end top_sram;
11
12  architecture Behavioral of top_sram is
13
14  signal cs, we, oe: std_logic;
15  signal  wdata:     std_logic_vector(3 downto 0);
16  signal address:    std_logic_vector(3 downto 0);
17
18  attribute mark_debug: string;
19  attribute keep: string ;
20  attribute mark_debug of address: signal is "true";
21  attribute mark_debug of wdata: signal is "true";
22
23  begin
24
25   data <= "1010" when ( we='1' and oe='0' )  else "ZZZZ";
26   wdata <= data;
27
28
29  g1:  entity sram ( beh_sram )
30       port map ( address => address, din => data, dout => data,
31                  cs => cs, we => we,  oe => oe );
32
33  g2:  entity sram_fsm ( fsm_beh )
34       port map ( clk=>clk, reset=>reset,
35           address=> address,
36           cs => cs, we => we, oe => oe );
37
38
39  end Behavioral;
```

SIMULATION WAVEFORM(S):

FIGURE 36
(STEP 1)

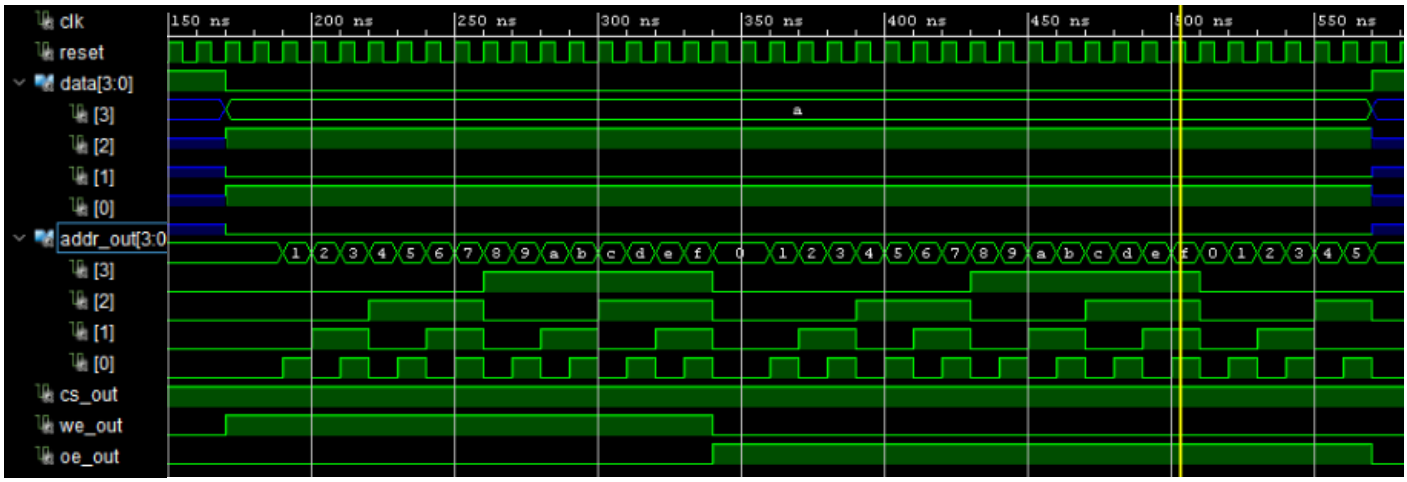
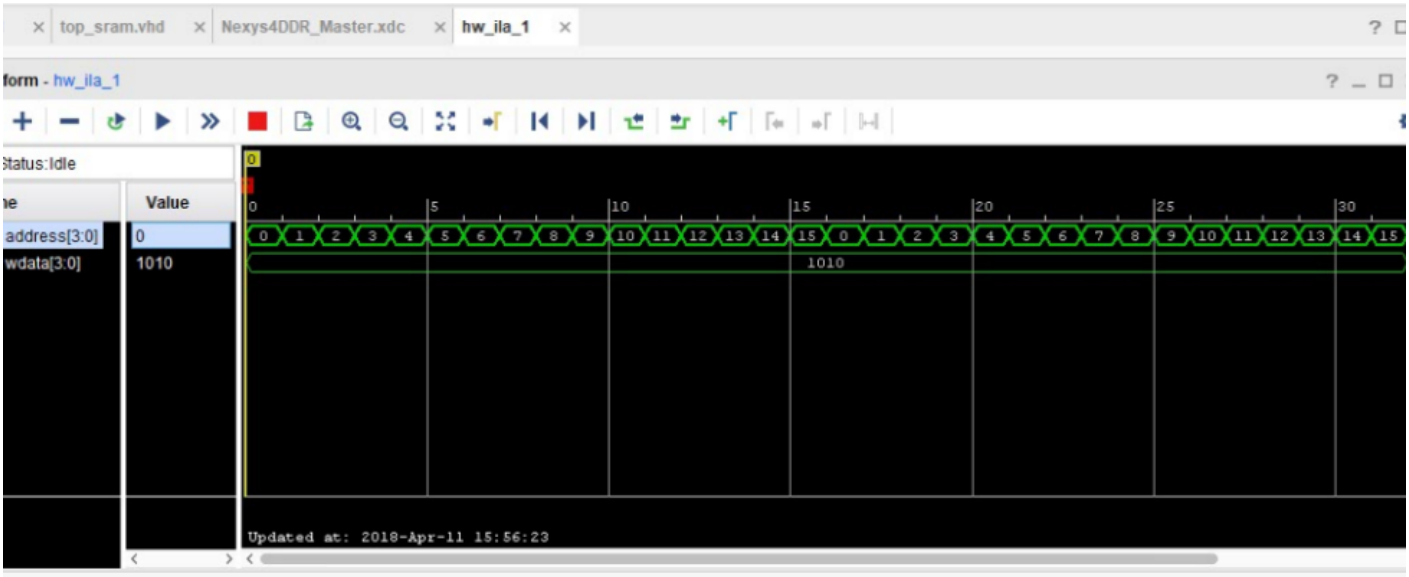


FIGURE 37
(STEP 2)



RESULT DISCUSSION:

Being handed code is always bittersweet. Sure, the workload is much lighter but I always learn a lot from struggling, finding an answer, and finally revising it. I don't feel as though I actually accomplished much this section beside learning how to setup for, and launch the ILA.