VHDL LAB #6

Rachel Ruddy - 261162987
Natasha Lawford - 261178596
Rachel Bunsick - 261159677

I. Executive Summary:

In this lab, we created a sequence detector to detect the instance of the 4-bit patterns “0010” and “1011” from a given input sequence of 1 bit numbers. The sequence detector utilizes two FSMs, one to detect the former pattern and one to detect the latter pattern. In the next part, we combined the logic of the sequence detector with a counter to keep track of the number of times that a given sequence occurred following each instance. Finally, we used the clock divider we created in Lab #5 to read and count the sequence from a ROM file by reading one bit every one second.

Sequence Detector Code:

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5   entity rachel_ruddy_FSM is
6      Port (seq    : in std_logic;
7            enable : in std_logic;
8            reset  : in std_logic;
9            clk    : in std_logic;
10           out_1  : out std_logic;
11           out_2  : out std_logic);
12  end rachel_ruddy_FSM;
13
14  architecture behavioral of rachel_ruddy_FSM is
15
16      -- States for FSM detecting "1011"
17      type fsm1_state_type is (S0, S1, S2, S3, S4);
18      signal fsm1_state, fsm1_next_state : fsm1_state_type;
19
20      -- States for FSM detecting "0010"
21      type fsm2_state_type is (T0, T1, T2, T3, T4);
22      signal fsm2_state, fsm2_next_state : fsm2_state_type;
23
24      begin
25        process(clk, reset)
26        begin
27           if reset = '1' then
28              fsm1_state <= S0;
29              fsm2_state <= T0;
30           elsif rising_edge(clk) then
31              if enable = '1' then
32                 fsm1_state <= fsm1_next_state;
33                 fsm2_state <= fsm2_next_state;
34              else
35                 fsm1_state <= fsm1_state;
36                 fsm2_state <= fsm2_state;
37              end if;
38           end if;
39        end process;
40
41        -- FSM 1 Behavior
42        process (fsm1_state, seq)
43        begin
44           out_1 <= '0';
45           case fsm1_state is
46              when S0 =>
47                  if seq = '1' then
48                      fsm1_next_state <= S1;
49                  else
50                      fsm1_next_state <= S0;
```

```
51                  end if;
52              when S1 =>
53                  if seq = '0' then
54                      fsm1_next_state <= S2;
55                  else
56                      fsm1_next_state <= S1;
57                  end if;
58              when S2 =>
59                  if seq = '1' then
60                      fsm1_next_state <= S3;
61                  else
62                      fsm1_next_state <= S0;
63                  end if;
64              when S3 =>
65                  if seq = '1' then
66                      fsm1_next_state <= S4;
67                  else
68                      fsm1_next_state <= S2;
69                  end if;
70              when S4 =>
71                  if seq = '1' then
72                      fsm1_next_state <= S1;
73                      out_1 <= '1';
74                  else
75                      fsm1_next_state <= S2;
76                      out_1 <= '1';
77                  end if;
78              when others =>
79                  fsm1_next_state <= S0;
80          end case;
81      end process;
82
83      -- FSM 2 Behavior
84        process (fsm2_state, seq)
85        begin
86           out_2 <= '0';
87           case fsm2_state is
88              when T0 =>
89                  if seq = '0' then
90                      fsm2_next_state <= T1;
91                  else
92                      fsm2_next_state <= T0;
93                  end if;
94              when T1 =>
95                  if seq = '0' then
96                      fsm2_next_state <= T2;
97                  else
98                      fsm2_next_state <= T0;
99                  end if;
100             when T2 =>
101                 if seq = '1' then
102                     fsm2_next_state <= T3;
103                 else
104                     fsm2_next_state <= T2;
105                 end if;
106             when T3 =>
107                 if seq = '0' then
108                     fsm2_next_state <= T4;
109                 else
110                     fsm2_next_state <= T1;
111                 end if;
112             when T4 =>
113                 if seq = '1' then
114                     fsm2_next_state <= T0;
115                     out_2 <= '1';
116                 else
117                     fsm2_next_state <= T2;
118                     out_2 <= '1';
119                 end if;
120             when others =>
121                 fsm2_next_state <= T0;
122         end case;
123     end process;
124
125 end behavioral;
```

The sequence detector code defines a new type of variables of state type to represent each of the five states for our two FSMs. First, we create a clock process which is triggered at the change of the clock input or reset input. We hand the reset case and additionally change the state of the FSM on the rising edge of the clock. Then, based on the state diagrams attached below the testbench, the logic for the state transitions along with the output is used in a process which is triggered when the state is changed.

Sequence Detector Testbench:

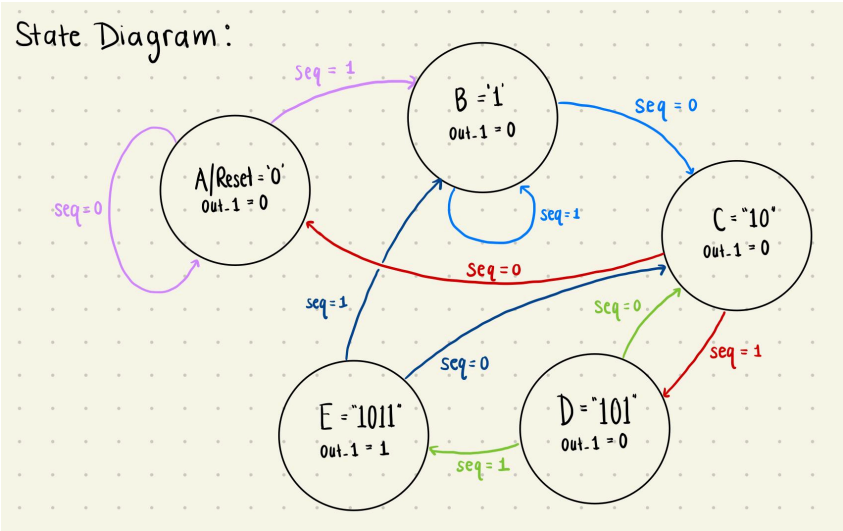
```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5   entity FSM_tb is
6   end FSM_tb;
7
8   architecture behavioral of FSM_tb is
9
10      component rachel_ruddy_FSM
11          Port (seq    : in std_logic;
12                enable : in std_logic;
13                reset  : in std_logic;
14                clk    : in std_logic;
15                out_1  : out std_logic;
16                out_2  : out std_logic);
17      end component;
18
19      signal seq_tb : std_logic := '0';
20      signal enable_tb : std_logic := '0';
21      signal reset_tb : std_logic := '0';
22      signal clk_tb : std_logic := '0';
23      signal out_1tb : std_logic := '0';
24      signal out_2tb : std_logic := '0';
25
26      constant clk_period : time := 10 ns;
27
28      begin
29
30      uut: rachel_ruddy_FSM
31          port map (
32                seq => seq_tb,
33                enable => enable_tb,
34                reset => reset_tb,
35                clk => clk_tb,
36                out_1 => out_1tb,
37                out_2 => out_2tb
38            );
39
40      clk_process: process
41      begin
42          while true loop
43              clk_tb <= '0';
44              wait for clk_period / 2;
45              clk_tb <= '1';
46              wait for clk_period / 2;
47          end loop;
48      end process;
49
```

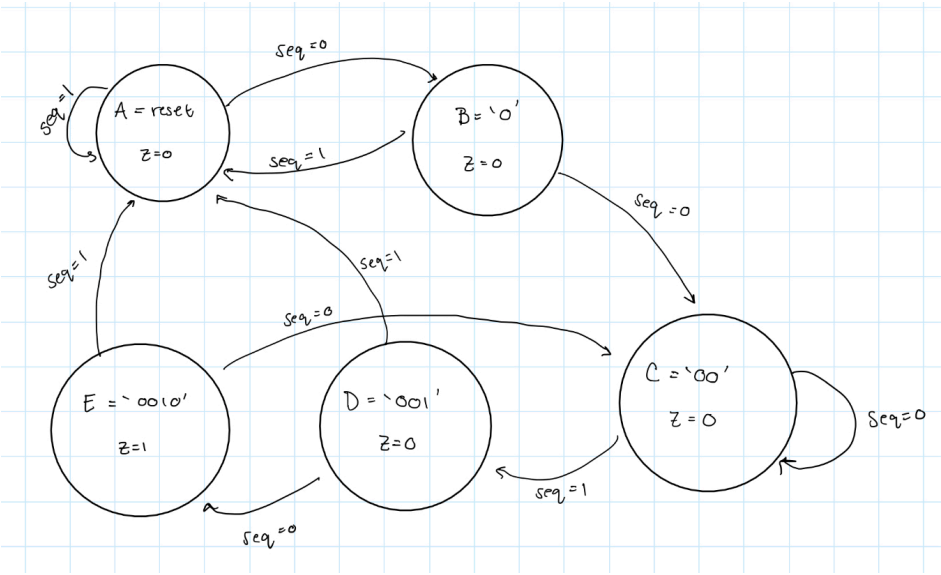
```
50      stimulus_process: process
51      begin
52
53      -- initialize
54      reset_tb <= '0';
55      wait for clk_period * 2;
56
57      reset_tb <= '1';
58      wait for clk_period * 2;
59
60      -- enable FSM
61      enable_tb <= '1';
62      wait for clk_period * 2;
63
64      seq_tb <= '0';
65      wait for clk_period * 2;
66
67      seq_tb <= '0';
68      wait for clk_period * 2;
69
70      seq_tb <= '1';
71      wait for clk_period * 2;
72
73      seq_tb <= '0';
74      wait for clk_period * 2;
75
76      seq_tb <= '1';
77      wait for clk_period * 2;
78
79      seq_tb <= '1';
80      wait for clk_period * 2;
81
82      seq_tb <= '0';
83      wait for clk_period * 2;
84
85      seq_tb <= '1';
86      wait for clk_period * 2;
87
88      seq_tb <= '0';
89      wait for clk_period * 2;
90
91      seq_tb <= '1';
92      wait for clk_period * 2;
93
94      seq_tb <= '1';
95      wait for clk_period * 2;
96
97      seq_tb <= '0';
```

```
 97        seq_tb <= '0';
 98        wait for clk_period * 2;
 99
100        seq_tb <= '1';
101        wait for clk_period * 2;
102
103        seq_tb <= '1';
104        wait for clk_period * 2;
105
106        seq_tb <= '0';
107        wait for clk_period * 2;
108
109        -- try reset
110        reset_tb <= '1';
111        wait for clk_period * 2;
112
113        seq_tb <= '0';
114        wait for clk_period * 2;
115
116        seq_tb <= '0';
117        wait for clk_period * 2;
118
119        seq_tb <= '1';
120        wait for clk_period * 2;
121
122        seq_tb <= '0';
123        wait for clk_period * 2;
124
125        seq_tb <= '0';
126        wait for clk_period * 2;
127
128        wait for clk_period * 20;
129
130        wait;
131        end process;
132
133    end behavioral;
134
```

The sequence detector testbench tests exhaustively the functionality of the detector by feeding it a specific input sequence to confirm that each state transition occurs as expected and that the expected output matches the functionality. Additionally, the reset and enable functions are tested to ensure that the behavior is what we expect.

State Diagrams for FSMs:





Above are the state diagrams for each sequence.

Sequence counter:

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5   entity rachel_ruddy_sequence_detector is
6       Port (seq : in std_logic;
7             enable: in std_logic;
8             reset : in std_logic;
9             clk : in std_logic;
10            cnt_1 : out std_logic_vector(2 downto 0); -- Count for "1011"
11            cnt_2 : out std_logic_vector(2 downto 0) -- Count for "0010"
12            );
13  end rachel_ruddy_sequence_detector;
14
15  Architecture Behavioral of rachel_ruddy_sequence_detector is
16
17      component rachel_ruddy_FSM is
18          Port (seq    : in std_logic;
19                enable : in std_logic;
20                reset  : in std_logic;
21                clk    : in std_logic;
22                out_1  : out std_logic;
23                out_2  : out std_logic);
24      end component;
25
26      signal out_1, out_2: std_logic := '0';
27      signal count_1, count_2: unsigned(2 downto 0) := "000";
28
29      begin
30
31      FSM_inst: rachel_ruddy_FSM -- instantiate the sequence detector
32          Port map (seq => seq,
33                    enable => enable,
34                    reset => reset,
35                    clk => clk,
36                    out_1 => out_1,
37                    out_2 => out_2
38                   );
39
40      Process (clk, reset)
41          begin
42              If reset = '0' then --overrides everything:
43                  count_2 <= "000"; --reset both counts to 0
44                  count_2 <= "000";
45              elsif rising_edge(clk) then -- if not reset, then check to see what happens on rising edge
46
47                  If enable = '1' then --if enable is on and sequence detected, add 1 to count(s)
48                      if out_1 = '1' then
49                          count_1 <= count_1 + 1;
50                      end if;
51                      If out_2 = '1' then
52                      count_2 <= count_2 + 1;
53                      end if;
54                  end if;
55              end if;
56      end process;
57
58          cnt_1 <= std_logic_vector(count_1);
59          cnt_2 <= std_logic_vector(count_2);
60
61  end Behavioral;
```

The sequence counter instantiates the FSM to detect the two given sequences from an output bit, but it additionally keeps track of the number of times that each output for the sequences has been 1.

Sequence counter testbench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_tb is
end counter_tb;

architecture behavioral of counter_tb is

    component rachel_ruddy_sequence_detector
       Port (seq    : in std_logic;
          enable : in std_logic;
          reset  : in std_logic;
          clk    : in std_logic;
          cnt_1  : out std_logic;
          cnt_2  : out std_logic);
    end component;

    signal seq_tb : std_logic := '0';
    signal enable_tb : std_logic := '0';
    signal reset_tb : std_logic := '0';
    signal clk_tb : std_logic := '0';
    signal cnt_1tb : std_logic := '0';
    signal cnt_2tb : std_logic := '0';

    constant clk_period : time := 10 ns;

    begin

    uut: rachel_ruddy_sequence_detector
       port map (
             seq => seq_tb,
             enable => enable_tb,
             reset => reset_tb,
             clk => clk_tb,
             cnt_1 => cnt_1tb,
             out_2 => cnt_2tb
          );

    clk_process: process
    begin
       while true loop
          clk_tb <= '0';
          wait for clk_period / 2;
          clk_tb <= '1';
          wait for clk_period / 2;
       end loop;
    end process;

    stimulus_process: process
    begin

    -- initialize
```

```
    reset_tb <= '1';
    wait for clk_period * 2;

    report "Starting stimulus process" severity note;
    reset_tb <= '0';
    wait for clk_period * 2;


    reset_tb <= '1';
    wait for clk_period * 2;

    -- enable FSM
    enable_tb <= '1';
    wait for clk_period * 2;

    -- sequence: 1011001001010
    -- cnt1 : 1011
-- cnt2 : 0010
    seq_tb <= '1';
    wait for clk_period * 2;

    seq_tb <= '0';
    wait for clk_period * 2;

    seq_tb <= '1';
    wait for clk_period * 2;

    seq_tb <= '1'; -- sequence 1 detected- cnt_1 = 001
    wait for clk_period * 2;

    seq_tb <= '0';
    wait for clk_period * 2;

    seq_tb <= '0';
    wait for clk_period * 2;

    seq_tb <= '1';
    wait for clk_period * 2;

    seq_tb <= '0';
    wait for clk_period * 2; -- sequence 2 detected- cnt_2 = 001

    seq_tb <= '0';
    wait for clk_period * 2;

seq_tb <= '1';
    wait for clk_period * 2;

    seq_tb <= '0';  -- sequence 2 detected again - cnt_2 = 010
    wait for clk_period * 2;

    --reset test! cnt_1 = 0, cnt_2 = 0
```

```
105     --reset test! cnt_1 = 0, cnt_2 = 0
106
107     reset_tb <= '0';
108     wait for clk_period * 2;
109
110     reset_tb <= '1';
111     wait for clk_period * 2;
112     -- finished reset test
113
114  -- Last sequence:  001011011
115
116     seq_tb <= '0';
117     wait for clk_period * 2;
118
119     seq_tb <= '0';
120     wait for clk_period * 2;
121
122     seq_tb <= '1';
123     wait for clk_period * 2;
124
125     seq_tb <= '0';
126     wait for clk_period * 2; -- sequence 2 detected cnt_2 = 001
127
128     seq_tb <= '1';
129     wait for clk_period * 2;
130
131     seq_tb <= '1';
132     wait for clk_period * 2; -- sequence 1 detected cnt_1 = 001
133
134     seq_tb <= '0';
135     wait for clk_period * 2;
136
137     seq_tb <= '1';
138     wait for clk_period * 2;
139
140     seq_tb <= '1';
141     wait for clk_period * 2; -- sequence 1 detected cnt_1 = 010
142
143     wait for clk_period * 20;
144     wait;
145     end process;
146
147  end behavioral;
148
```

The sequence counter testbench iterates through different edge cases to observe the behavior of the sequence detector depending on the reset and enable. The same sequence of input bits is used from the sequence detector testbench, however we are also able to check to see whether the counter is functioning as expected.

Wrapper Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity rachel_ruddy_wrapper is
    Port (reset   : in std_logic;
          clk     : in std_logic;
          HEX0    : out std_logic_vector(6 downto 0);
          HEX5    : out std_logic_vector(6 downto 0));
end rachel_ruddy_wrapper;

architecture behavior of rachel_ruddy_wrapper is

    component ROM is
       port(
        clk : in std_logic;
        reset : in std_logic;
        data : out std_logic);
    end component;


    component seven_segment_decoder is
       Port   (code : in std_logic_vector (3 downto 0);
               segments_out: out std_logic_vector(6 downto 0));
    end component;


    component rachel_ruddy_sequence_detector is
       Port (seq : in std_logic;
             enable: in std_logic;
             reset : in std_logic;
             clk : in std_logic;
             cnt_1 : out std_logic_vector(2 downto 0); -- Count for "1011"
             cnt_2 : out std_logic_vector(2 downto 0) -- Count for "0010"
         );
    end component;


    component rachel_ruddy_clock_divider is
       port (enable  : in std_logic;
          reset   : in std_logic;
          clk     : in std_logic;
          en_out  : out std_logic);
    end component;

    signal enable : std_logic := '1';
    signal clk2   : std_logic := '0';
    signal data   : std_logic := '0';
    signal cnt_1  : std_logic_vector(2 downto 0);
```

```
50      signal cnt_2  : std_logic_vector(2 downto 0);
51      signal u_cnt_1  : unsigned(3 downto 0);
52      signal u_cnt_2  : unsigned(3 downto 0);
53      signal new_cnt_1  : std_logic_vector(3 downto 0) := "0000";
54      signal new_cnt_2  : std_logic_vector(3 downto 0) := "0000";
55      signal decoded_cnt1   : std_logic_vector(6 downto 0);
56      signal decoded_cnt2   : std_logic_vector(6 downto 0);
57
58      begin
59
60      i1: rachel_ruddy_clock_divider port map (enable, reset, clk, clk2);
61      i2: ROM port map (clk2, reset, data);
62      i3: rachel_ruddy_sequence_detector port map (data, enable, reset, clk2, cnt_1, cnt_2);
63
64      u_cnt_1 <= '0' & unsigned(cnt_1);
65      new_cnt_1 <= std_logic_vector(u_cnt_1);
66      u_cnt_2 <= '0' & unsigned(cnt_2);
67      new_cnt_2 <= std_logic_vector(u_cnt_2);
68
69      i4: seven_segment_decoder port map (new_cnt_1, decoded_cnt1);
70      i5: seven_segment_decoder port map (new_cnt_2, decoded_cnt2);
71
72      HEX0 <= decoded_cnt1;
73      HEX5 <= decoded_cnt2;
74
75  end behavior;
76
```

The wrapper code instantiates the clock divider from Lab #5 and the output is used as the clock for both the ROM instance and the sequence detector instance. The sequence detector reads the output of the ROM as its input and detects the given sequences. Then, the count of the number of occurrences is mapped to 2 different seven-segment decoders so that they can be displayed on the board.

Wrapper Testbench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity wrapper_tb is
end wrapper_tb;

architecture behavioral of wrapper_tb is

    component rachel_ruddy_wrapper is
        Port (reset   : in std_logic;
              clk     : in std_logic;
              HEX0    : out std_logic_vector(6 downto 0);
              HEX5    : out std_logic_vector(6 downto 0));
    end component;

    signal reset_tb : std_logic := '0';
    signal clk_tb : std_logic := '0';
    signal HEX0_tb : std_logic_vector(6 downto 0);
    signal HEX5_tb : std_logic_vector(6 downto 0);

    constant clk_period : time := 10 ns;

    begin

    uut: rachel_ruddy_wrapper
        port map (
              reset => reset_tb,
              clk => clk_tb,
              HEX0 => HEX0_tb,
              HEX5 => HEX5_tb
            );

    clk_process: process
    begin
        while true loop
            clk_tb <= '0';
            wait for clk_period / 2;
            clk_tb <= '1';
            wait for clk_period / 2;
        end loop;
    end process;

    stimulus_process: process
    begin

        reset_tb <= '0';
        wait until rising_edge(clk_tb);

        reset_tb <= '1';
        wait until rising_edge(clk_tb);

        wait for 300 sec;

        wait;

    end process;

end behavioral;
```

The wrapper testbench allows the wrapper instance to run for 300 seconds to observe its behavior as the data from the ROM file is read.

II. Questions:

1. **Why is it better to use two FSMs, rather than one, in the implementation of the sequence detector from Section 4?**

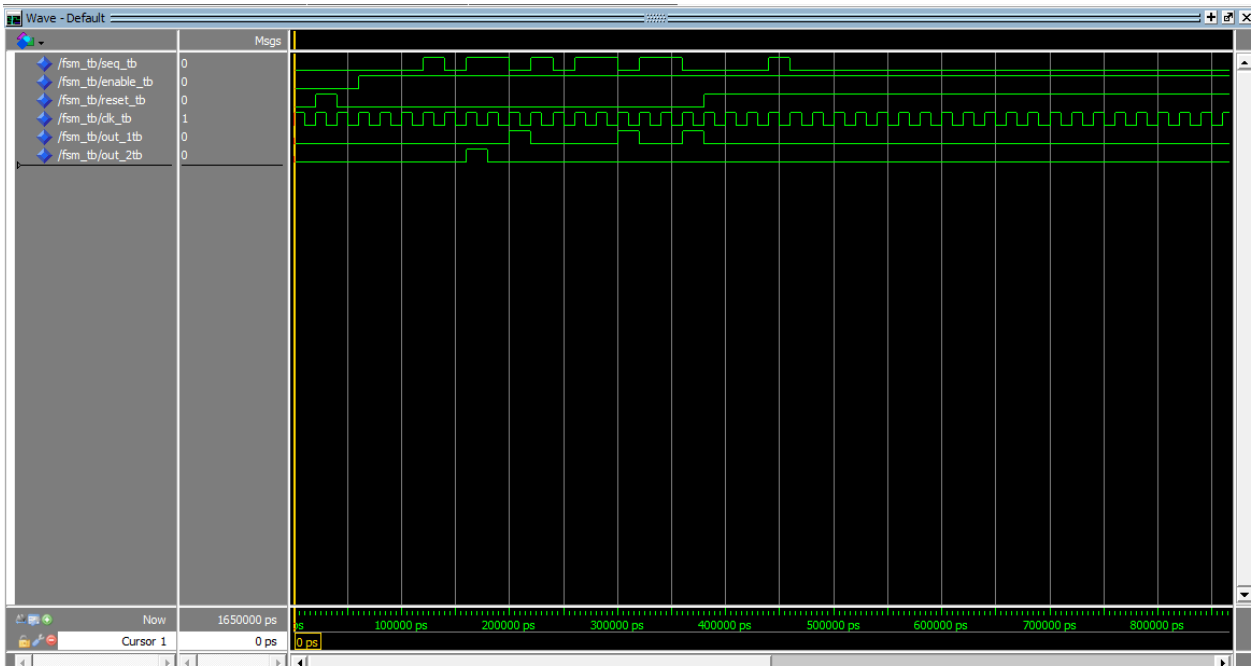
If we were to use only one FSM to detect both sequences, it would be extremely difficult to determine different state transitions since we only have one input variable, and since the sequences are not the same, receiving 1 vs 0 will lead to conflicting states.

2. **Report the number of pins and logic modules used to fit your design on the FPGA board**

Flow Status	Successful - Mon Dec 09 16:30:51 2024
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	rachelLab6
Top-level Entity Name	rachel_ruddy_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	51 / 32,070 (< 1 %)
Total registers	51
Total pins	16 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

III. Schematics and simulation results (and explanations/labels of figures/important points on the simulation plots)

Sequence Detector Waveform:



IV. Conclusions:

This lab introduced us to the concept of incorporating FSMs to create circuits which have both combinational and sequential elements. When approaching a more complex problem like this, it is important to make sure that you understand the logic and state transitions before implementation to reduce the amount of time spent debugging. Additionally, it is important to consider the efficiency of how you set up each FSM to reduce the complexity of your design.