ECSE 222 Digital Logic
Lab Assignment III

Group 62
Antoine Wang 260766084
Yicheng Song 260763294

• The state diagram of the FSM

Below is the state diagram of the FSM implemented in our circuit. E represent the enable signal which is active high. D is the direction signal with D = 1 defining the up-counting mode and D = 0 defining the down-counting mode. R is the active-low reset signal. Note that the state diagram presented is simplified since the reset situation from state B to n is not shown since the complexity of the diagram is already abundant. More specifically, for every stage there should be two additional arrows point to the two initial cases (A and O). When reset is active low and the direction is set to up-counting, one of the arrows should be pointing at A. Similarly, the R=0, D=1, the other arrow points at O.

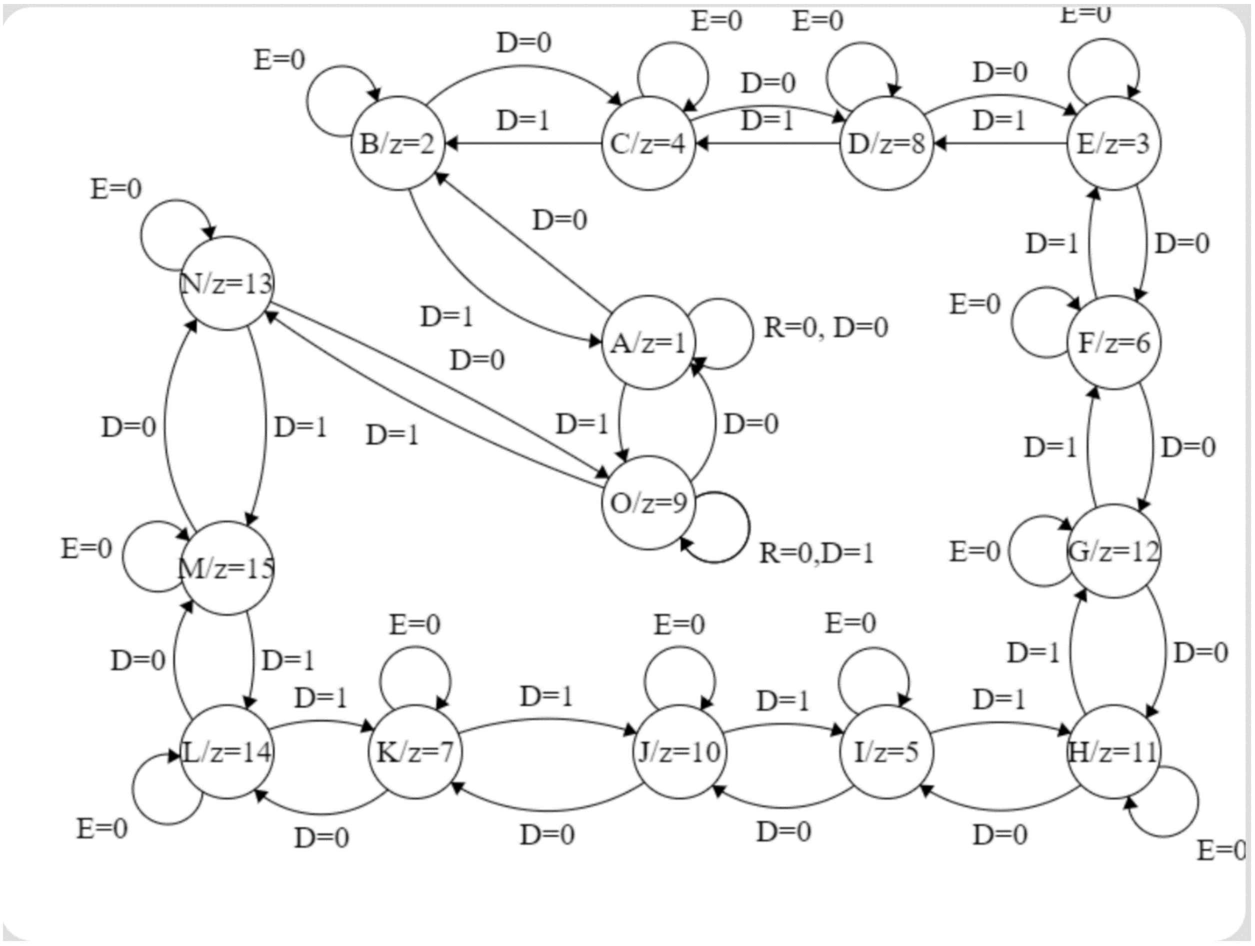
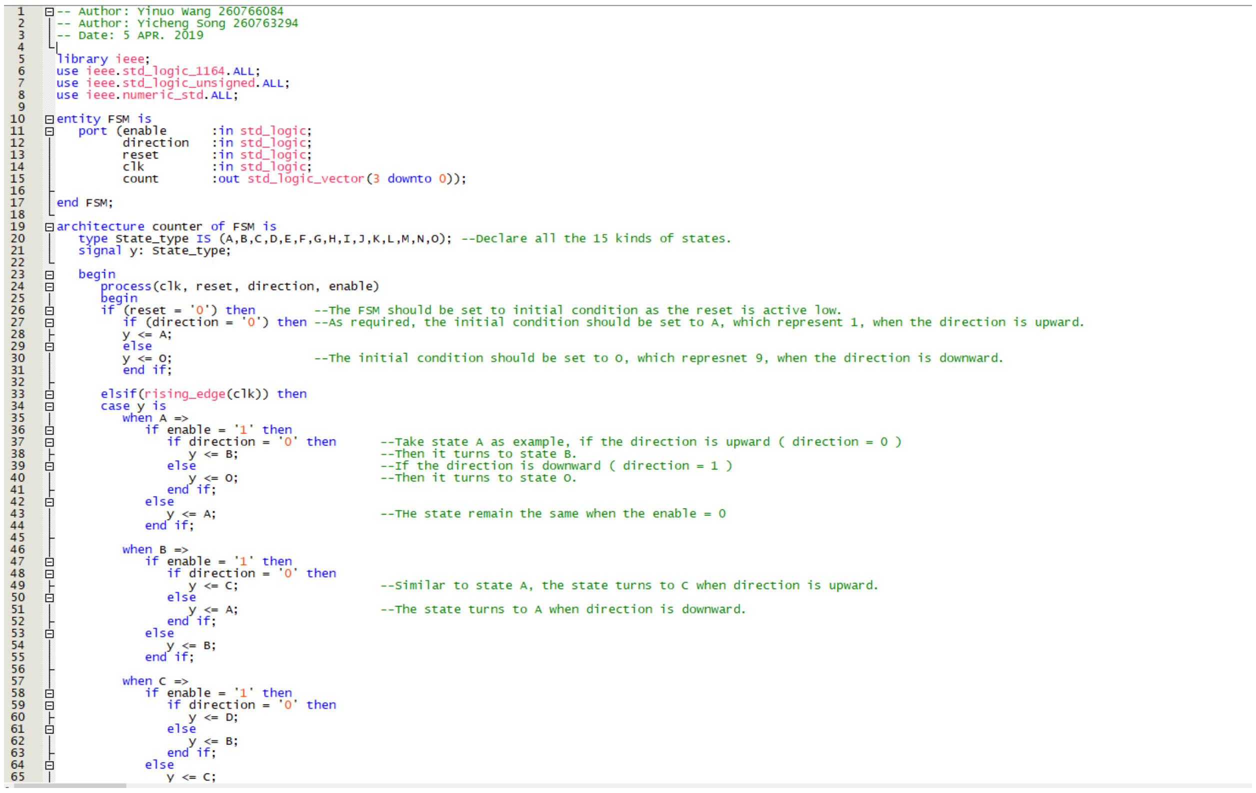


Figure 1. An illustration of the state diagram of the FSM implemented in the multimode counter

• A description of the FSM circuit

The basic logic behind of the FSM is that we listed all 15 states and defined all successors of each states in the VHDL implementation. In the code segment presented below, in the Process 4 signals are included fir checking each stage. When reset is set to be low, the circuit assigns the current stage with the initial stages (A or O) based on the direction signal. If direction is 0, the circuit will operate in positive direction, thus clear to stage A. On the other hand, when direction is 1, the initial stage assigned to the circuit is O.

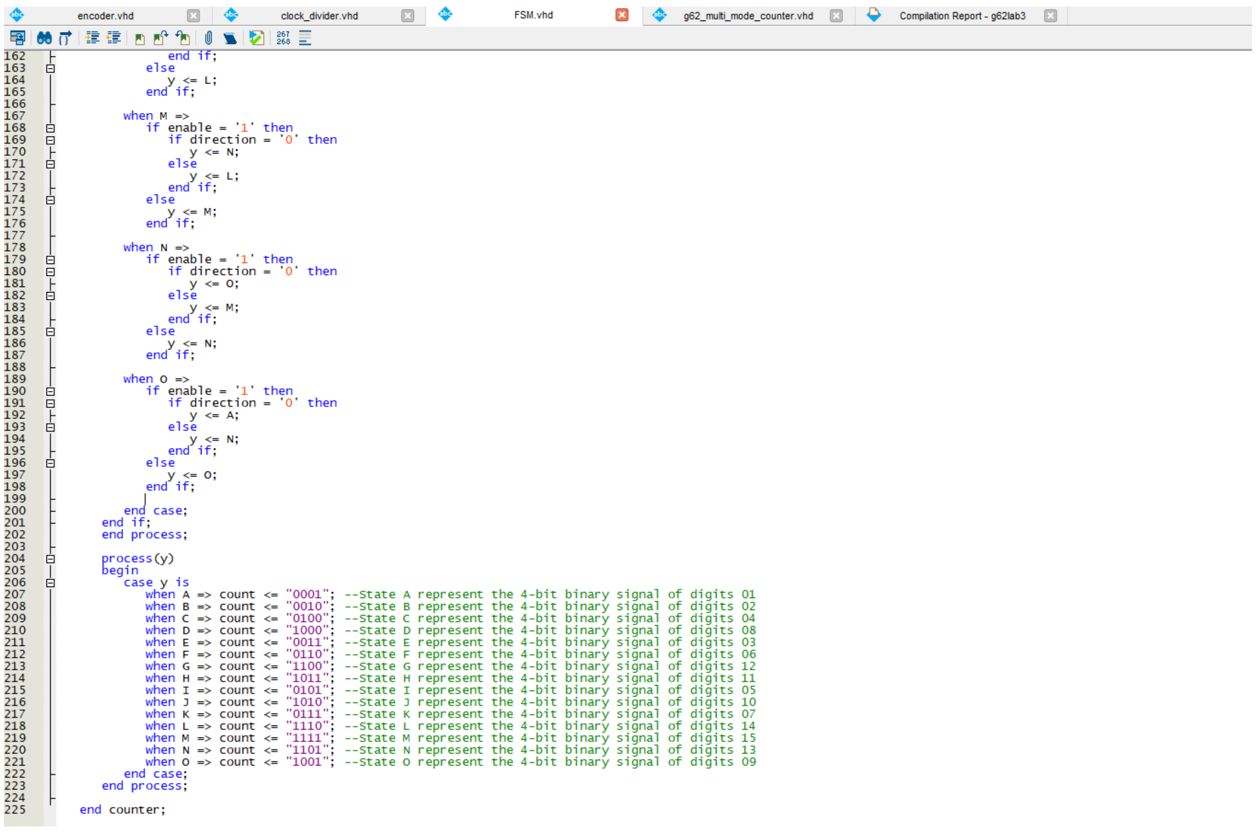
Then, when reset is being nullified (R=1), the circuit behaves base on the positive edge of the clock signal (clk). As described in the following "elsif" block, whenever a rising edge is met, the circuit itself select the current stage. Then, in the next layer of "if" statement, enable signal is considered to determine whether the current stage should change or stays the same. If the enable is high, meaning the status is active, the next layer of "if" will be reached. The sequence of the counting by the state machine will be decided based on the input direction signal. '1' will count the series of signal in forward direction and '0' is the backward direction. An example is the case A block presented below. A will proceed to either the stage after it (B) or before it (O) based on the direction signal.



```vhdl
-- Author: Yinuo Wang 260766084
-- Author: Yicheng Song 260763294
-- Date: 5 APR. 2019

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

entity FSM is
   port (enable      :in std_logic;
         direction   :in std_logic;
         reset       :in std_logic;
         clk         :in std_logic;
         count       :out std_logic_vector(3 downto 0));

end FSM;

architecture counter of FSM is
   type State_type IS (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O); --Declare all the 15 kinds of states.
   signal y: State_type;

   begin
      process(clk, reset, direction, enable)
      begin
      if (reset = '0') then        --The FSM should be set to initial condition as the reset is active low.
         if (direction = '0') then --As required, the initial condition should be set to A, which represent 1, when the direction is upward.
         y <= A;
         else
         y <= O;                   --The initial condition should be set to O, which represnet 9, when the direction is downward.
         end if;

      elsif(rising_edge(clk)) then
      case y is
         when A =>
            if enable = '1' then
               if direction = '0' then      --Take state A as example, if the direction is upward ( direction = 0 )
                  y <= B;                   --Then it turns to state B.
               else                         --If the direction is downward ( direction = 1 )
                  y <= O;                   --Then it turns to state O.
               end if;
            else
               y <= A;                      --THe state remain the same when the enable = 0
            end if;

         when B =>
            if enable = '1' then
               if direction = '0' then
                  y <= C;                   --Similar to state A, the state turns to C when direction is upward.
               else
                  y <= A;                   --The state turns to A when direction is downward.
               end if;
            else
               y <= B;
            end if;

         when C =>
            if enable = '1' then
               if direction = '0' then
                  y <= D;
               else
                  y <= B;
               end if;
            else
               y <= C;
```

Figure 1. A segment of the states in FSM VHDL implementation

A final step to complete the FSM is that all the tokens from A to O defining each state should be decoded into legitimate output signals containing binary representations. Thus, at the end of the FSM implementation a selection block is added. Based on the current states, the corresponding binary value will be returned. The 4-bit binary number, named as “count” signal, will later be used in LED decoding.

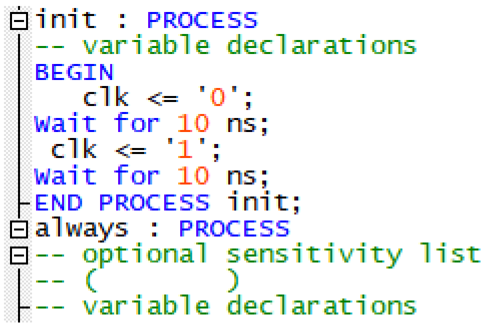


```vhdl
            end if;
         else
            y <= L;
         end if;

      when M =>
         if enable = '1' then
            if direction = '0' then
               y <= N;
            else
               y <= L;
            end if;
         else
            y <= M;
         end if;

      when N =>
         if enable = '1' then
            if direction = '0' then
               y <= O;
            else
               y <= M;
            end if;
         else
            y <= N;
         end if;

      when O =>
         if enable = '1' then
            if direction = '0' then
               y <= A;
            else
               y <= N;
            end if;
         else
            y <= O;
         end if;

      end case;
   end if;
   end process;

   process(y)
   begin
      case y is
         when A => count <= "0001"; --State A represent the 4-bit binary signal of digits 01
         when B => count <= "0010"; --State B represent the 4-bit binary signal of digits 02
         when C => count <= "0100"; --State C represent the 4-bit binary signal of digits 04
         when D => count <= "1000"; --State D represent the 4-bit binary signal of digits 08
         when E => count <= "0011"; --State E represent the 4-bit binary signal of digits 03
         when F => count <= "0110"; --State F represent the 4-bit binary signal of digits 06
         when G => count <= "1100"; --State G represent the 4-bit binary signal of digits 12
         when H => count <= "1011"; --State H represent the 4-bit binary signal of digits 11
         when I => count <= "0101"; --State I represent the 4-bit binary signal of digits 05
         when J => count <= "1010"; --State J represent the 4-bit binary signal of digits 10
         when K => count <= "0111"; --State K represent the 4-bit binary signal of digits 07
         when L => count <= "1110"; --State L represent the 4-bit binary signal of digits 14
         when M => count <= "1111"; --State M represent the 4-bit binary signal of digits 15
         when N => count <= "1101"; --State N represent the 4-bit binary signal of digits 13
         when O => count <= "1001"; --State O represent the 4-bit binary signal of digits 09
      end case;
   end process;

end counter;
```

Figure 2. The assignment of values based on current states in the FSM VHDL code

• A discussion of how the FSM circuit was tested, showing representative simulation plots. How do you know that these circuits work correctly?

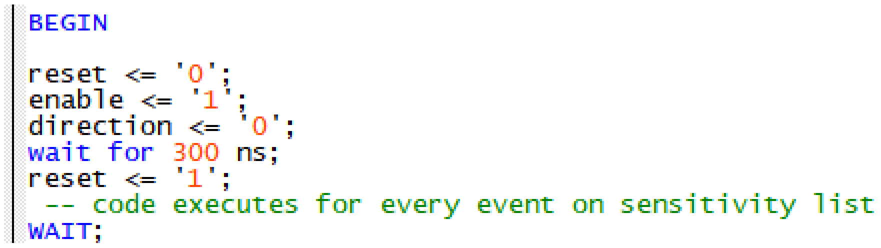
We test the counter by giving inputs to simulate the real conditions and comparing the simulated results with our expected ones.



```
init : PROCESS
-- variable declarations
BEGIN
   clk <= '0';
wait for 10 ns;
 clk <= '1';
wait for 10 ns;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (         )
-- variable declarations
```

Figure 1. Simulation of clock signal with frequency of 50M Hz.

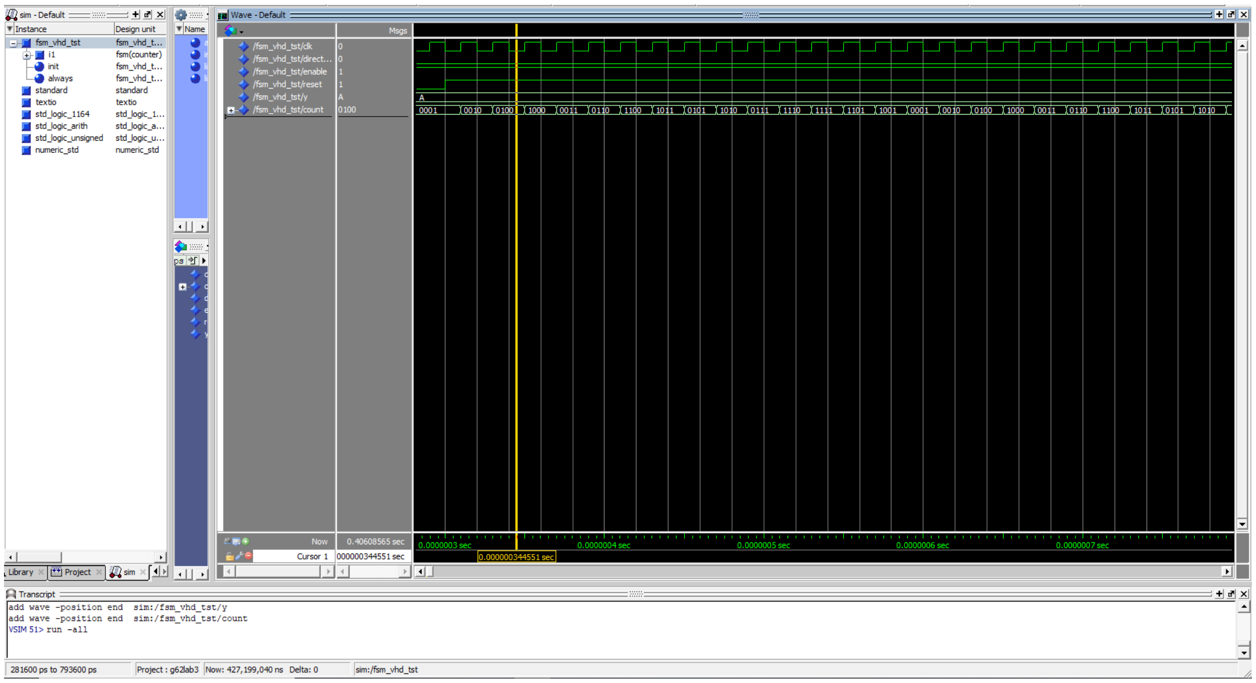
We set the clock signal with frequency of 50 million to simulate the real one by setting the value of clock value changing in period of 10 nanoseconds. The initial reset is set as ‘0’, enable is set as ‘1’ and the direction is ‘0’ for the first 300 nanoseconds. With this group of initial conditions, we can check whether the reset works. As reset is active low and direction is set to ‘0’ the state will be set to A under any condition, which means the output should always be ‘0001’ during this period. After 300 nanoseconds, the reset is set to ‘1’ for testing whether the FSM works properly. Since we have set initial state to A by previous step and direction is ‘0’, the state should be passed to B at the next rising edge of clock signal. By similar manner, the state would be passed to C after B, and so on, which means the output should be the 4-bit binary signal of the number starting from 1 in the queue in order from left to right.



```
BEGIN

reset <= '0';
enable <= '1';
direction <= '0';
wait for 300 ns;
reset <= '1';
 -- code executes for every event on sensitivity list
WAIT;
```

Figure 2. Simulation of initial condition of inputs: reset, enable and direction.

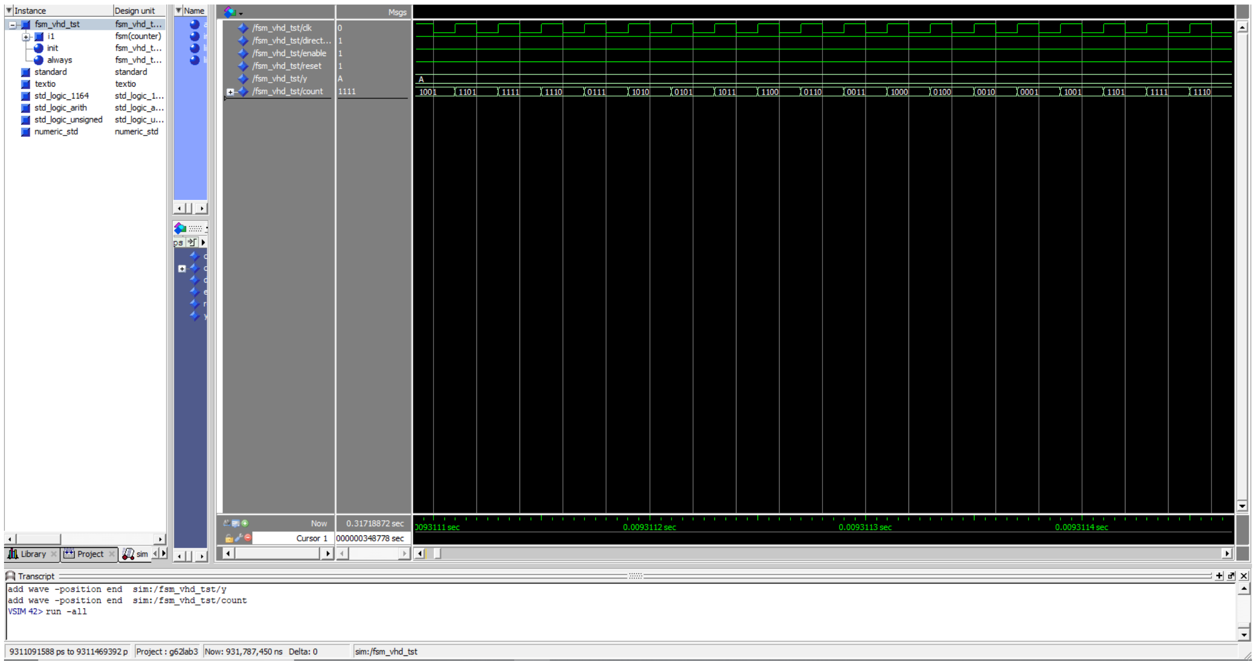
As can be seen from the simulation plots, two modes are implemented with the testing bench, Figure 3 below demonstrates the modelSim simulation of the code provided in figure 2. The direction is 0 thus the entire circuit counts in the positive direction. In figure 4 the direction signal is tweaked into 1, making the wave to behave in down-counting manner.



```
add wave -position end  sim:/fsm_vhd_tst/y
add wave -position end  sim:/fsm_vhd_tst/count
VSIM 51> run -all
```

```
281600 ps to 793600 ps | Project : g62lab3 | Now: 427,199,040 ns  Delta: 0 | sim:/fsm_vhd_tst
```

Figure 3. FSM simulation plot in up-counting manner.



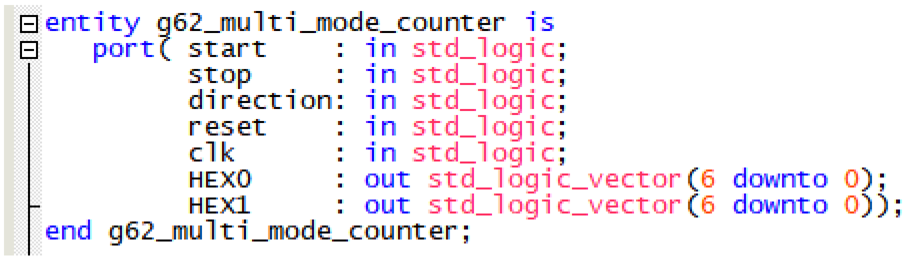
```
add wave -position end  sim:/fsm_vhd_tst/y
add wave -position end  sim:/fsm_vhd_tst/count
VSIM 42> run -all
```

```
9311091588 ps to 9311469392 p | Project : g62lab3 | Now: 931,787,450 ns  Delta: 0 | sim:/fsm_vhd_tst
```

Figure 4. FSM simulation plot in down-counting manner.

• A description of the multi-mode counter circuit.

At the very beginning the inputs start, stop, reset which control the progress of the counter, and the input direction which control the direction of the counter, are declared together with the clock signal clk and two 7-segment-coded outputs HEX0 and HEX1.

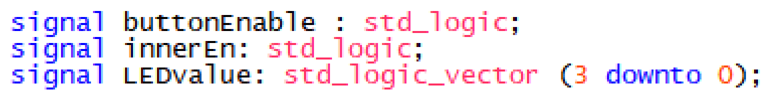


```
entity g62_multi_mode_counter is
    port( start     : in std_logic;
          stop      : in std_logic;
          direction: in std_logic;
          reset     : in std_logic;
          clk       : in std_logic;
          HEX0      : out std_logic_vector(6 downto 0);
          HEX1      : out std_logic_vector(6 downto 0));
end g62_multi_mode_counter;
```

Figure 1. Declaration of all inputs and outputs.

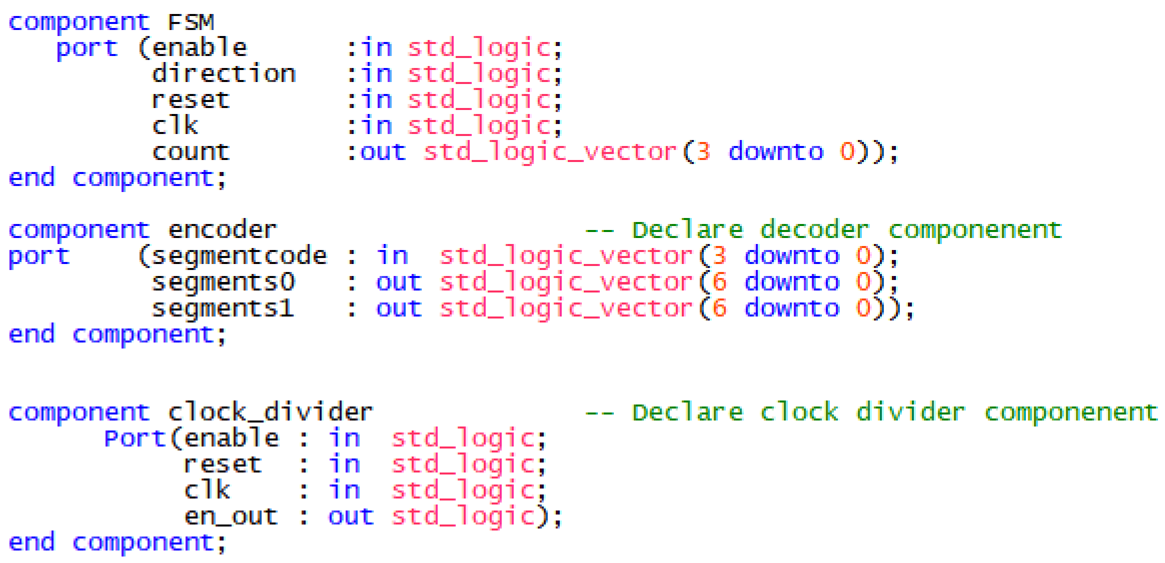
For building the architecture of multi-mode counter, we first declare a signal buttonEnable for storing the state of start and stop button, a signal innerEn which is produced by clock divider and functioned as enable signal for the FSM, and a 4-bit signal LEDvalue for storing the 4-bit binary output of FSM.

Then the FSM, encoder and clock_divider are imported and declared for future utilization.



```
signal buttonEnable : std_logic;
signal innerEn: std_logic;
signal LEDvalue: std_logic_vector (3 downto 0);
```

Figure 2. Declaration of signals for the following architecture.



```
component FSM
   port (enable       :in std_logic;
         direction    :in std_logic;
         reset        :in std_logic;
         clk          :in std_logic;
         count        :out std_logic_vector(3 downto 0));
end component;

component encoder                        -- Declare decoder componenent
port     (segmentcode : in  std_logic_vector(3 downto 0);
         segments0    : out std_logic_vector(6 downto 0);
         segments1    : out std_logic_vector(6 downto 0));
end component;


component clock_divider                  -- Declare clock divider componenent
      Port(enable : in  std_logic;
           reset  : in  std_logic;
           clk    : in  std_logic;
           en_out : out std_logic);
end component;
```

Figure 3. Declarations for importing all components we need.

A clock_divider stage cd is first constructed for the basic time dividing. It takes the state of button buttonEnable as enable, inputs reset, and clk as reset and clock signal, and produce an innerEn which represent the divider of time.

The FSM stage fsm1 followed the clock_divider is then built. The innerEn which represent '1' with interval of 1 second is functioned as input enable for the fsm1. With taking the direction value and the reset value together with the clock signal clk, fsm1 should produce a 4-bit binary signal which represent the value of the current state in binary numbers.

With the 4-bit binary signal given by fsm1, we could get the two 7-segment-coded signals HEX0 and HEX1 which represent the binary numbers in digital form by importing them to the Decoder stage.

```
cd: clock_divider port map(buttonEnable, reset, clk, innerEn); --map to a clock divider

fsm1: FSM port map(innerEn, direction, reset, clk,  LEDvalue);
Decoder: encoder port map(LEDvalue,HEX0,HEX1);
```

Figure 4. Map to clock_divider, FSM and encoder.

We store the state of buttons by using the signal buttonEnable. At every rising edge we check the state of the start and stop buttons. If the start button is pressed but the stop one is not then the counter should start counting so the buttonEnable for this condition should be '1', which means when the input start is '0' and stop is '1' then the buttonEnable should be set to '1', vice versa, and for all other conditions the buttonEnable should keep the same as previous state.

```
if (rising_edge(clk)) then
   if(start ='1' and stop = '0') then
         buttonEnable <= '0'; --stop is pressed, enable set to zero, the entire system froze from clock divider
   elsif(start ='0' and stop = '1') then
         buttonEnable <= '1';  -- start is pressed. Clk_divider starts and activate entire system
   else
         buttonEnable <= buttonEnable;
   end if;
end if;
```

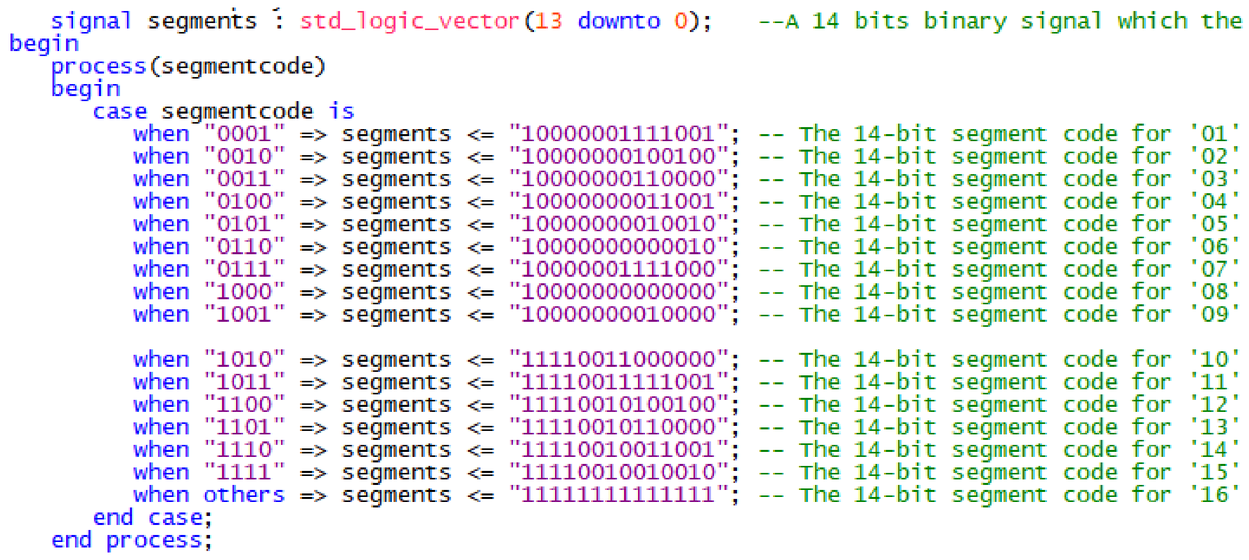
Figure 5. Logic of storing button state.

It should be reminded that the clock_divider circuit is similar to that of lab2 whereas the encoder circuit is far different from that of lab1. In lab3 we are supposed to display numbers on the board with digital numbers instead of hexadecimal numbers, which means we cannot use the same 7-segment code we used in lab1. As a result, we change the output of encoder to two 7-segment code which should be in the range of 0 to 9.

```
Port (  segmentcode : in  std_logic_vector(3 downto 0);     --Input numbers represented by a 4-bit binary signal.
        segments0   : out std_logic_vector(6 downto 0);     --The 7-segment code of the first digit of the state.
        segments1   : out std_logic_vector(6 downto 0));    --The 7-segment code of the second digit of the state.
```

Figure 6. Declaration of inputs and outputs of encoder circuit.

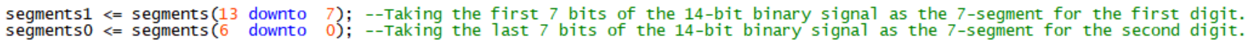
With the 14-bit signal segments declared, we transfer the 4-bit binary input to relative 14-bit 7-segment-coded signal.



```
   signal segments : std_logic_vector(13 downto 0);   --A 14 bits binary signal which the
begin
   process(segmentcode)
   begin
      case segmentcode is
         when "0001" => segments <= "10000001111001"; -- The 14-bit segment code for '01'
         when "0010" => segments <= "10000000100100"; -- The 14-bit segment code for '02'
         when "0011" => segments <= "10000000110000"; -- The 14-bit segment code for '03'
         when "0100" => segments <= "10000000011001"; -- The 14-bit segment code for '04'
         when "0101" => segments <= "10000000010010"; -- The 14-bit segment code for '05'
         when "0110" => segments <= "10000000000010"; -- The 14-bit segment code for '06'
         when "0111" => segments <= "10000001111000"; -- The 14-bit segment code for '07'
         when "1000" => segments <= "10000000000000"; -- The 14-bit segment code for '08'
         when "1001" => segments <= "10000000010000"; -- The 14-bit segment code for '09'

         when "1010" => segments <= "11110011000000"; -- The 14-bit segment code for '10'
         when "1011" => segments <= "11110011111001"; -- The 14-bit segment code for '11'
         when "1100" => segments <= "11110010100100"; -- The 14-bit segment code for '12'
         when "1101" => segments <= "11110010110000"; -- The 14-bit segment code for '13'
         when "1110" => segments <= "11110010011001"; -- The 14-bit segment code for '14'
         when "1111" => segments <= "11110010010010"; -- The 14-bit segment code for '15'
         when others => segments <= "11111111111111"; -- The 14-bit segment code for '16'
      end case;
   end process;
```

Figure 7. Transfer from 4-bit binary number to relative 14-bit 7-segment-coded number.

By splitting the 14-bit signal into two 7-bit signals, we can express the input number in decimal on the FPGA board.



```
segments1 <= segments(13 downto  7); --Taking the first 7 bits of the 14-bit binary signal as the 7-segment for the first digit.
segments0 <= segments(6  downto  0); --Taking the last 7 bits of the 14-bit binary signal as the 7-segment for the second digit.
```

Figure 8. Split of the 14-bit signal.

• A discussion of how the multi-mode counter circuit was tested on the FPGA board.

With all codes compiled and mapped, we can have the multi-mode counter circuit tested.

First, we need to check whether all three buttons work correctly. As being set, KEY2 on the board refers to start, KEY1 refers to stop and KEY0 refers to reset. The three buttons are tested by pressing buttons from KEY2 to KEY0 in order. The counter started counting after KEY2 is pressed and stopped after KEY1 is pressed. KEY0 was tested twice to check whether the counter could be reset to correct to correct initial conditions with different directions.

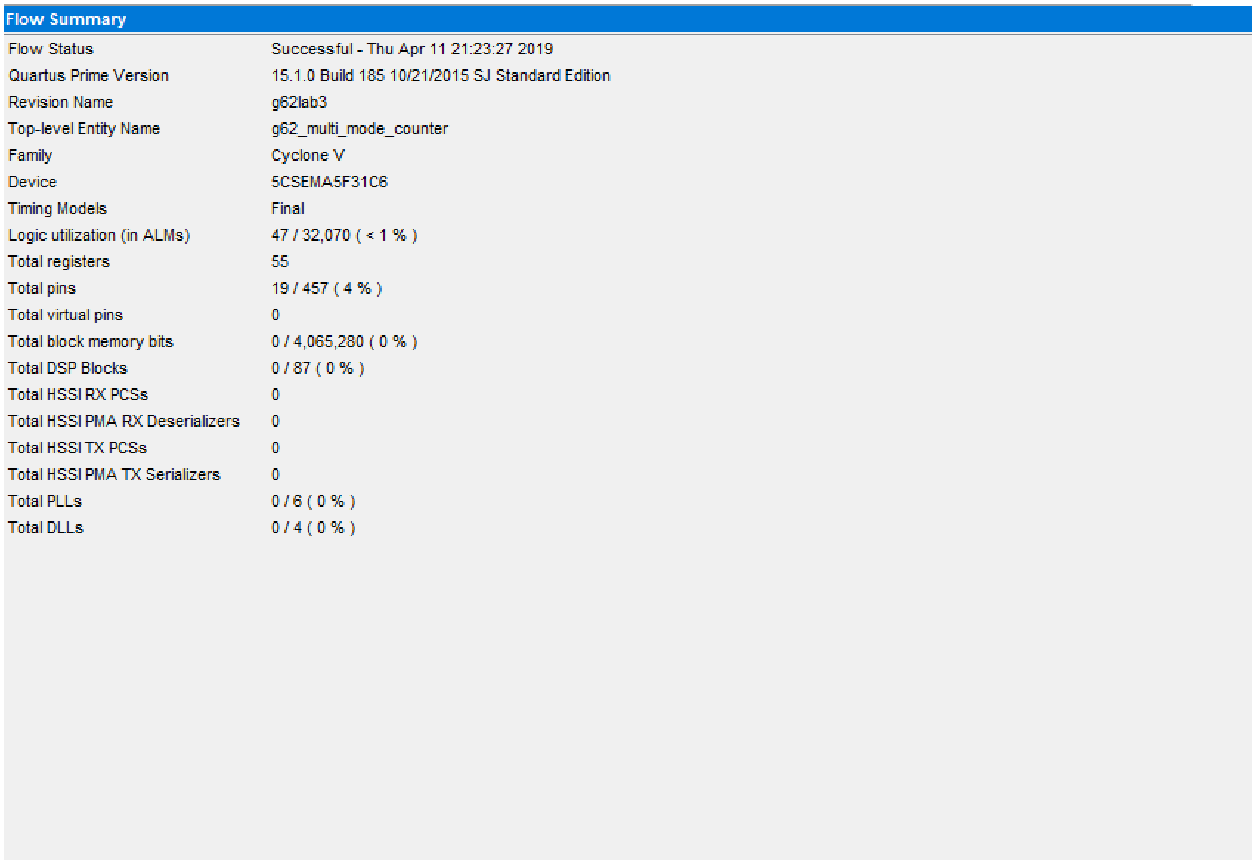
Then the direction switch should be tested. On our board, KEY0 refers to the direction switch. As we've set '0' for upward direction and '1' for downward direction and the switch on FPGA board is active-high, the display on the board should go up when switched down and go down when switched up. The change of direction should be able to implement at any state of the FSM, which means the direction should change whenever the switch changes.

At last we need to check whether all numbers displayed correctly and in the correct order. The numbers should be displayed as 2-bit digital numbers, for instance, 01. The interval between each number should be 1 second and the sequence should be reset to the correct initial condition with reference to current direction.

By checking all these 3 aspects carefully, there is no obvious error.

• A summary of the FPGA resource utilization (from the Compilation Report's Flow Summary) and the RTL schematic diagram for the multi-mode counter circuit. Clearly specify which part of your code maps to which part of the schematic diagram.

Below is the flow summary of the multimode counter circuit. More registers are used comparing to lab 2 since the states of the FSM increased in this lab. More bits need to be stored. However, the resources involved in the implementation is not abundant. Also, the number of used pins in the board is less then lab 2 since only 2 LED lights are activated for the counter to work.



Flow Summary	
Flow Status	Successful - Thu Apr 11 21:23:27 2019
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Standard Edition
Revision Name	g62lab3
Top-level Entity Name	g62_multi_mode_counter
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	47 / 32,070 (< 1 %)
Total registers	55
Total pins	19 / 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 %)

Figure 1: A summary of the FPGA resource utilization (from the Compilation Report's Flow Summary)

Code Mapping and RTL Diagrams:

Below is the overall RTL diagram of the multimode counter. The blue components are responsible of handling and processing the button and switch input. The logic will tweak the button input to become the enable to be used later in the clock divider. Then, the first sequential component, which is the clock divider, takes the buttonEnable signal along with the reset and clk signal to divide up the 50MHz frequency. The modified frequency, which is 1Hz, are rippled into the FSM component as enable to trigger the change of different stages. The output is exported at any time into the new decoder that we implement specifically for this lab. The 4-bit binary representation now corresponds to 2 decimal numbers, thus a 14-bit LED representation. The exact mapping code please refer to figure 4 in the fourth section.

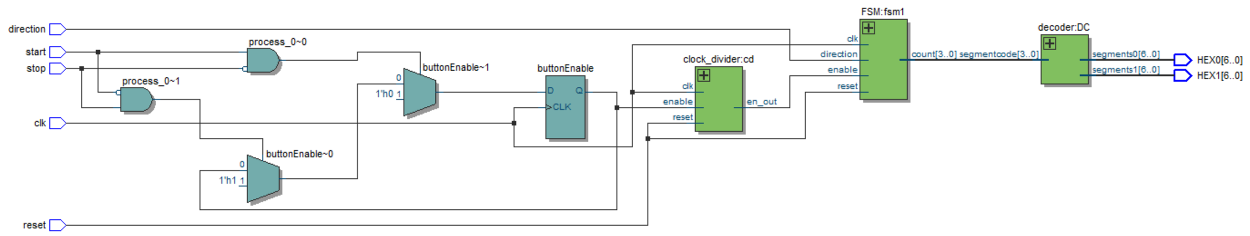
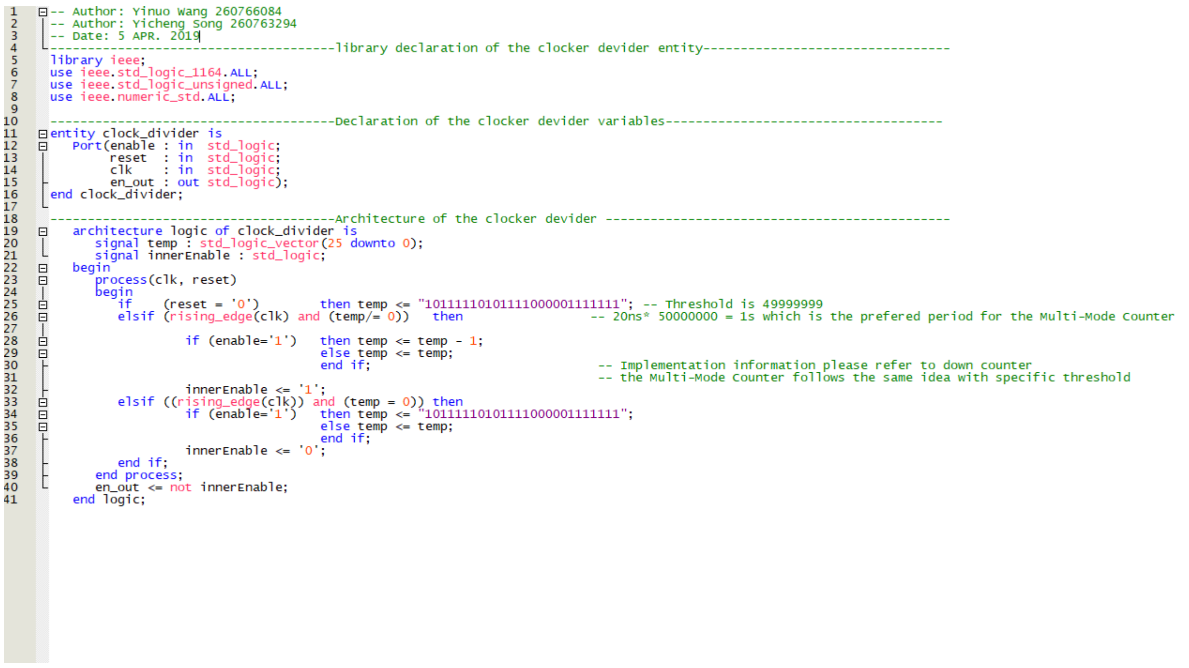


Figure 2: RTL diagram of the overall multimode counter circuit

For an entity of the clock divider, the corresponding RTL diagram is presented below. Note that the logic is same as that in lab 2. The reason why the RTL diagram is so extensive is that the increase threshold needs a 26-bit binary number (to represent 49999999). The code and the corresponding circuit diagram are shown as figure 3-4.



```
-- Author: Yinuo Wang 260766084
-- Author: Yicheng Song 260763294
-- Date: 5 APR. 2019
--------------------------------------library declaration of the clocker devider entity--------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.ALL;

--------------------------------------Declaration of the clocker devider variables------------------------------------
entity clock_divider is
   Port(enable : in  std_logic;
        reset  : in  std_logic;
        clk    : in  std_logic;
        en_out : out std_logic);
end clock_divider;

--------------------------------------Architecture of the clocker devider ---------------------------------------------
   architecture logic of clock_divider is
      signal temp : std_logic_vector(25 downto 0);
      signal innerEnable : std_logic;
   begin
      process(clk, reset)
      begin
         if    (reset = '0')        then temp <= "10111110101111000001111111"; -- Threshold is 49999999
         elsif (rising_edge(clk) and (temp/= 0))   then                      -- 20ns* 50000000 = 1s which is the prefered period for the Multi-Mode Counter

                  if (enable='1')   then temp <= temp - 1;
                                    else temp <= temp;
                                    end if;                              -- Implementation information please refer to down counter
                                                                         -- the Multi-Mode Counter follows the same idea with specific threshold
                  innerEnable <= '1';
         elsif ((rising_edge(clk)) and (temp = 0)) then
                  if (enable='1')   then temp <= "10111110101111000001111111";
                                    else temp <= temp;
                                    end if;
                  innerEnable <= '0';
         end if;
      end process;
      en_out <= not innerEnable;
   end logic;
```

Figure 3: VHDL code of the clock divider

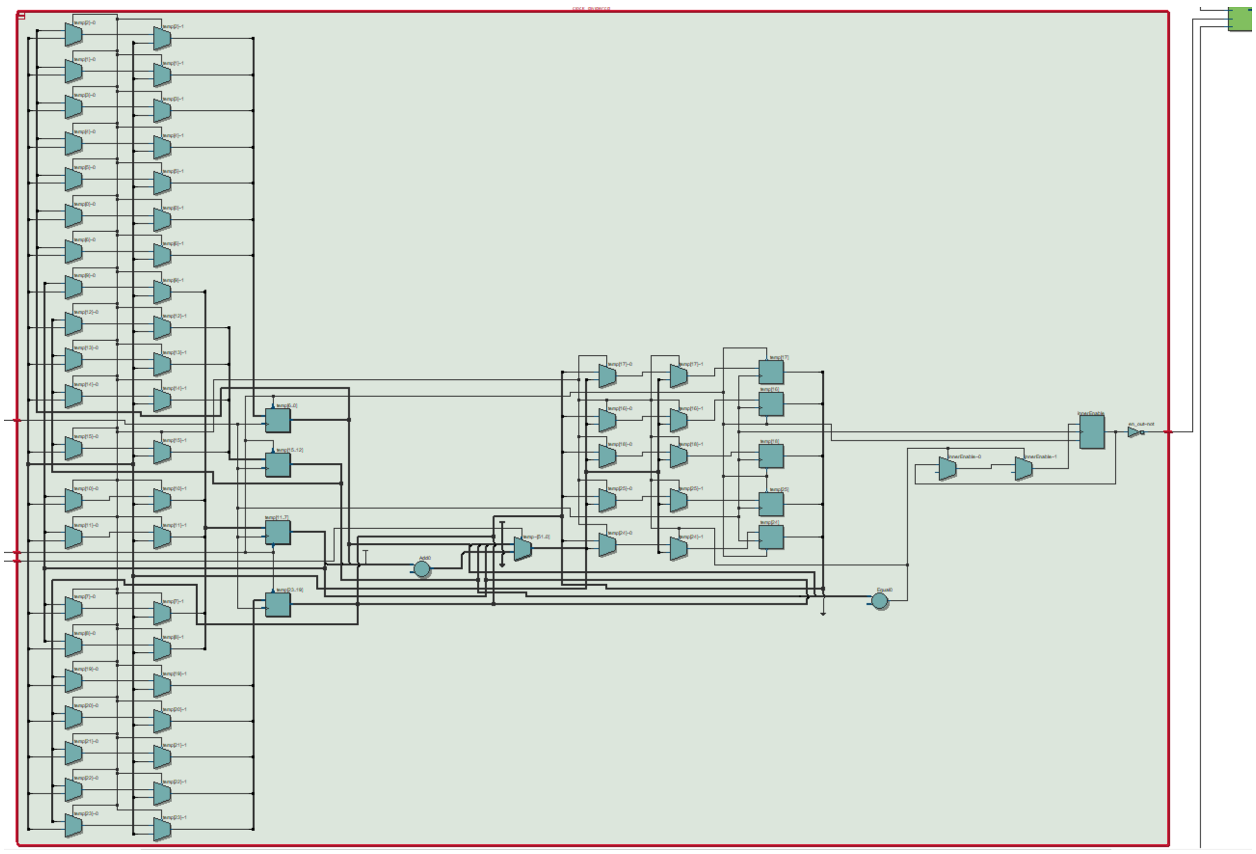


Figure 4: RTL diagram of the clock divider

For an entity of the FSM, the RTL diagram is present below. It is quite extensive since 15 cases are included all in the single entity.

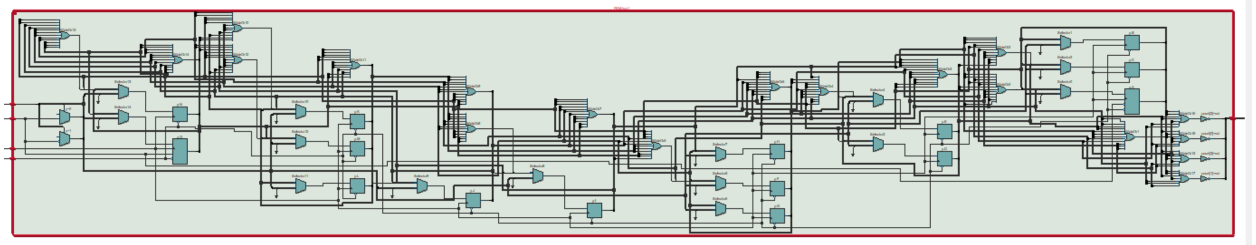


Figure 5: RTL diagram of the FSM

For an entity of the decoder, for this lab the modified decoder can handle a single 4-bit input into a 14-bit LED representation. Note that the central segment of the MSB of the LED digit will always be non-active (since number 0 or 1 never lights up the central segment), only 13 multiplexers are needed as shown in the RTL diagram below.

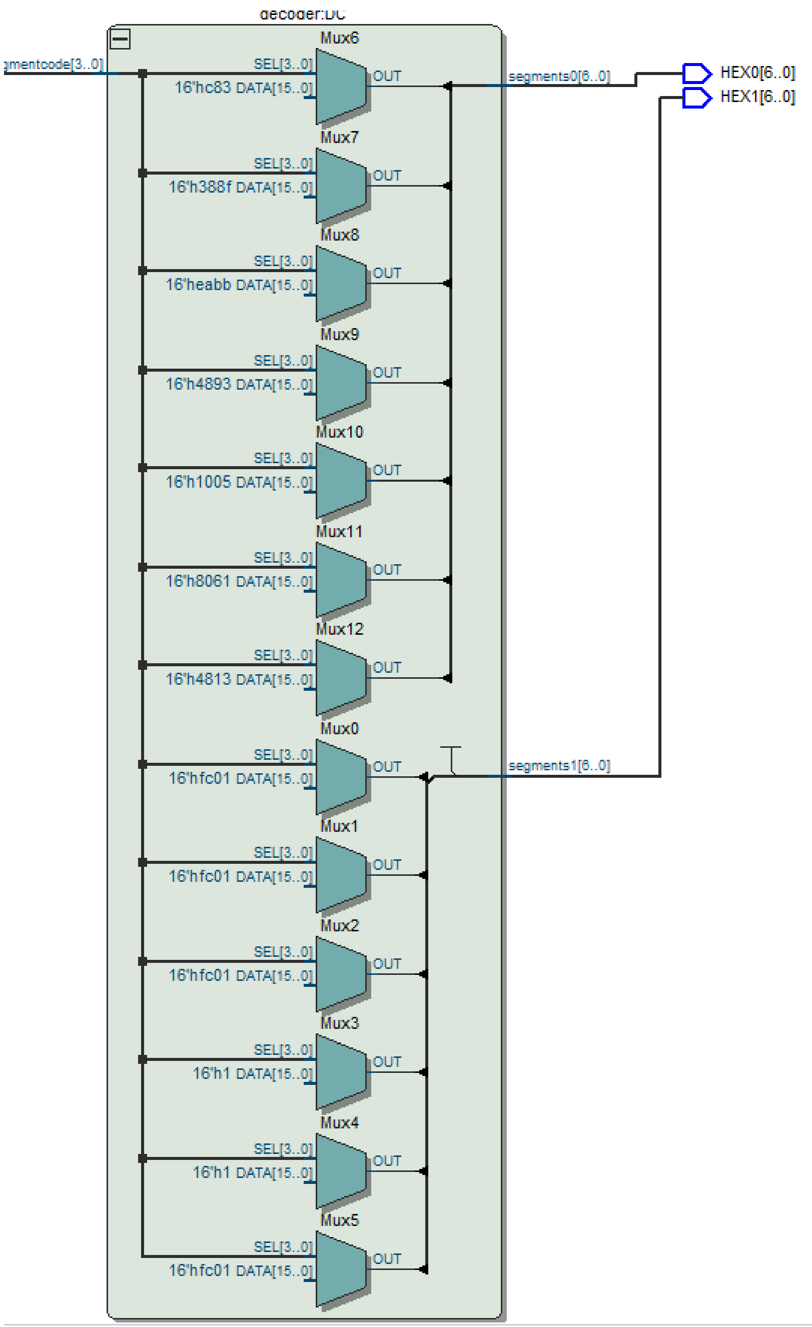


Figure 6: RTL diagram of the decoder (from 01 to 15)