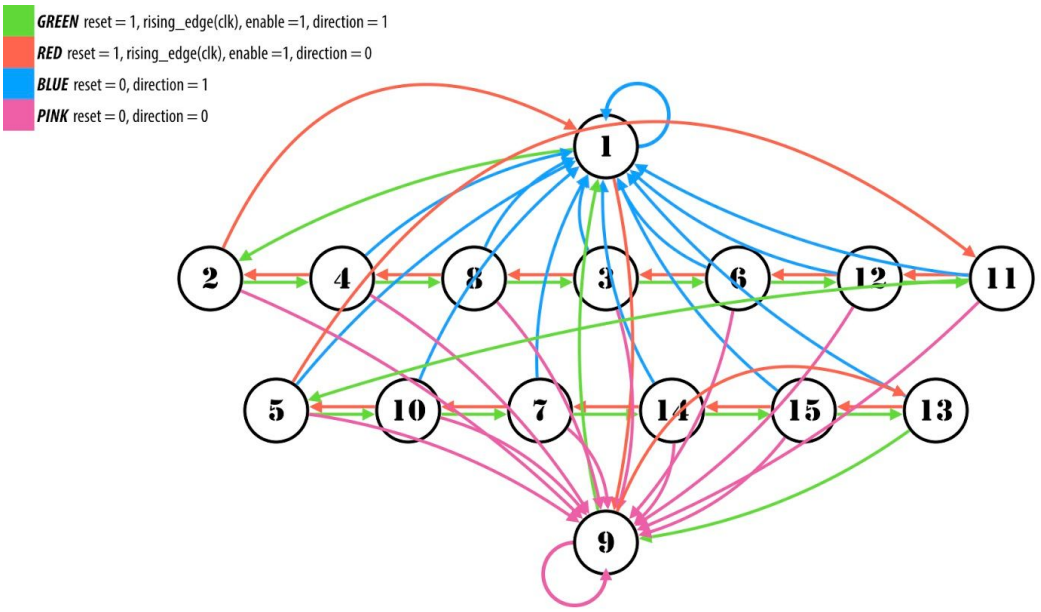
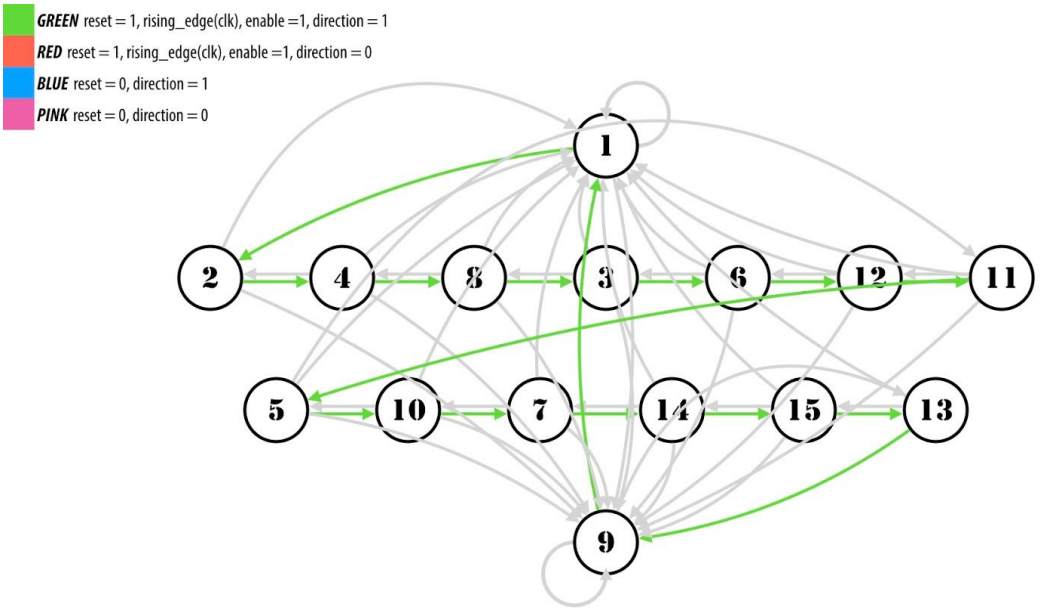
ECSE 222 Lab 2

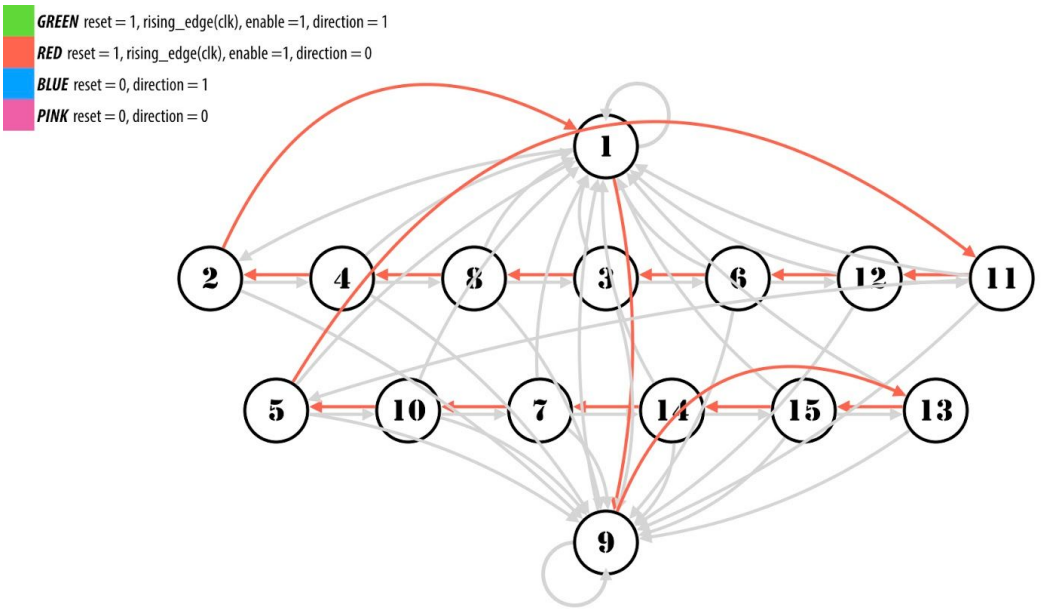
Glen Xu 260767363

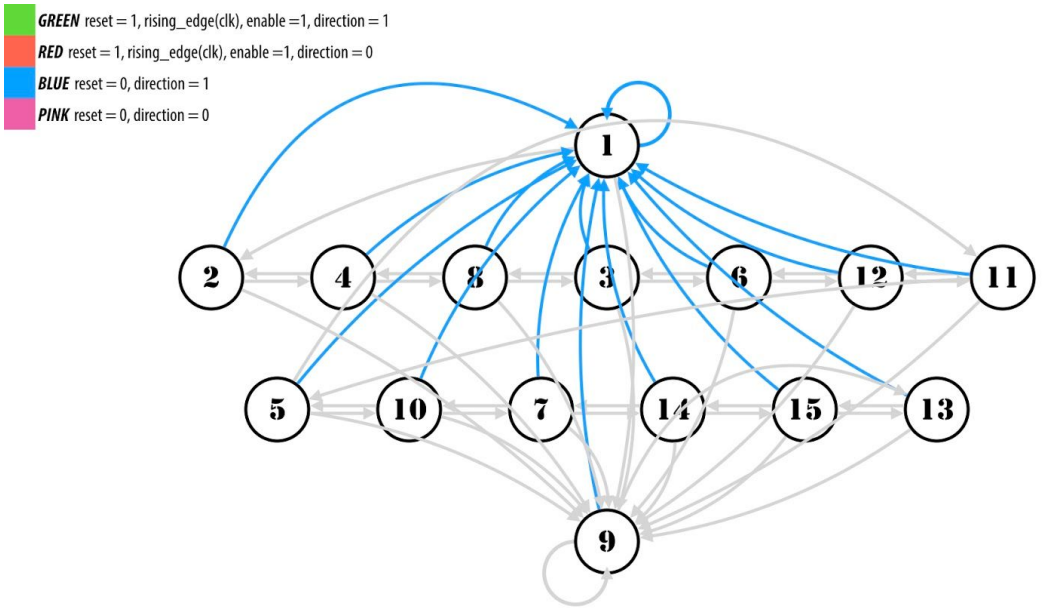
& Cheng Chen 260775674

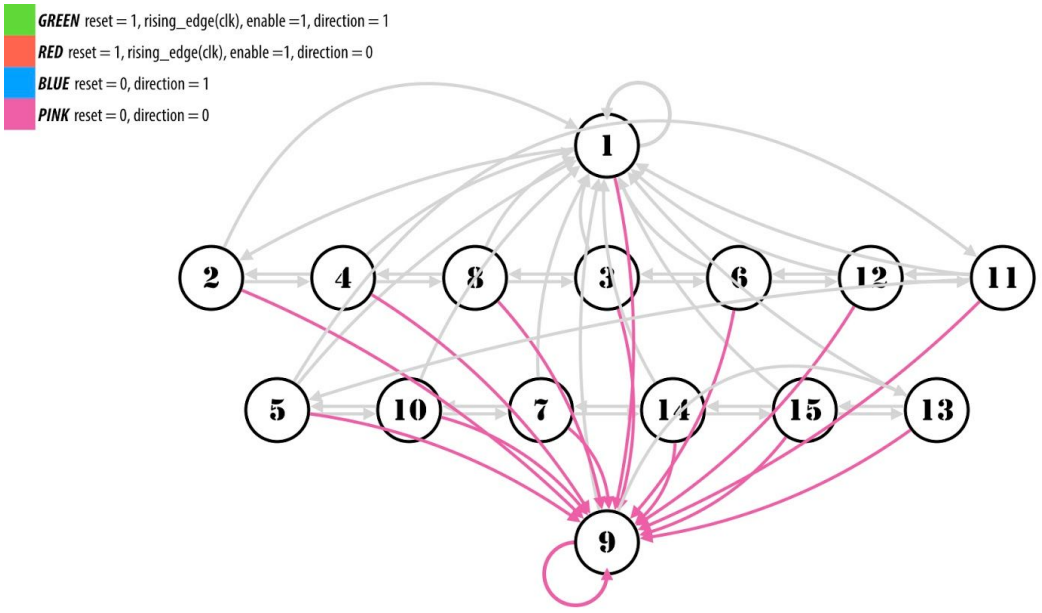
Group 56


GREEN reset = 1, rising_edge(clk), enable =1, direction = 1
RED reset = 1, rising_edge(clk), enable =1, direction = 0
BLUE reset = 0, direction = 1
PINK reset = 0, direction = 0
1
2
4
8
3
6
12
11
5
10
7
14
15
13
9


GREEN reset = 1, rising_edge(clk), enable =1, direction = 1
RED reset = 1, rising_edge(clk), enable =1, direction = 0
BLUE reset = 0, direction = 1
PINK reset = 0, direction = 0
1
2
4
8
3
6
12
11
5
10
7
14
15
13
9


GREEN reset = 1, rising_edge(clk), enable =1, direction = 1
RED reset = 1, rising_edge(clk), enable =1, direction = 0
BLUE reset = 0, direction = 1
PINK reset = 0, direction = 0
1
2
4
8
3
6
12
11
5
10
7
14
15
13
9





FSM Circuit

DESIGN

The FSM circuit is a Moore-style FSM. It can count in both directions, and can be reset to the starting points at anytime. This circuit receives 4 inputs and provides 1 output. The circuit is able to count upwards from 0 to 15 with 4-bit binary numbers. The result from the previous counting process will be stored in order to be added upwards.

The circuit requires a `reset`, a `direction`, an `enable`, and a `clk` as inputs, and a 4-bit vector storing `count` as output. The entity is declared with these 4 ports.

```
entity g56_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 g56_FSM;
```

Then, 15 types called `one`, `two` all the way to `fifteen` are declared in the architecture, and a signal `count_temp` is declared as well to store the result from the previous cycle in the sequential circuit.

```
architecture behaviour of g56_FSM is
  type state_type is (one, two, four, eight, three, six,
twelve, eleven, five, ten, seven, fourteen, fifteen, thirteen,
nine);
```

```
  signal count_temp : state_type := one;
```

Then, the process is implemented. The reset value is not a synchronous signal, and result of the counter will accumulate for every cycle. In the process, we need to know about changes of both of the signals, so they both have to be in the sensitivity list.

```
begin
Process(clk, reset) begin
```

When `reset` equals 0, i.e. when it is active, set `count_temp` to the starting point according to the `direction`. If `direction` equals 0, then set `count_temp` to nine; if `direction` equals 1, then set `count_temp` to one.

Contrariwise, when `clk` is on its rising edge and when `enable` = 1, `count_temp` is set to be the next according to the direction. For example, currently `count_temp` is two, if `direction` is 0, then `count_temp` will be set to one; if `direction` is 1, then `count_temp` will be set to four.

In other cases, keep `count_temp` to be the same.

```
if reset = '0' then
  if direction = '0' then
    count_temp <= nine;
  else
```

```
    count_temp <= one;
  end if;
elsif(rising_edge(clk)) then
  if(enable = '1') then
    CASE count_temp is
      when one =>
        if direction = '0' then
          count_temp <= nine;
        else
          count_temp <= two;
        end if;

      when two =>
        if direction = '0' then
          count_temp <= one;
        else
          count_temp <= four;
        end if;

      when four =>
        if direction = '0' then
          count_temp <= two;
        else
          count_temp <= eight;
```

```
  end if;

when eight =>
  if direction = '0' then
    count_temp <= four;
  else
    count_temp <= three;
  end if;

when three =>
  if direction = '0' then
    count_temp <= eight;
  else
    count_temp <= six;
  end if;

when six =>
  if direction = '0' then
    count_temp <= three;
  else
    count_temp <= twelve;
  end if;

when twelve =>
```

```
  if direction = '0' then
    count_temp <= six;
  else
    count_temp <= eleven;
  end if;

when eleven =>
  if direction = '0' then
    count_temp <= twelve;
  else
    count_temp <= five;
  end if;

when five =>
  if direction = '0' then
    count_temp <= eleven;
  else
    count_temp <= ten;
  end if;

when ten =>
  if direction = '0' then
    count_temp <= five;
  else
```

```
    count_temp <= seven;
  end if;

when seven =>
  if direction = '0' then
    count_temp <= ten;
  else
    count_temp <= fourteen;
  end if;

when fourteen =>
  if direction = '0' then
    count_temp <= seven;
  else
    count_temp <= fifteen;
  end if;

when fifteen =>
  if direction = '0' then
    count_temp <= fourteen;
  else
    count_temp <= thirteen;
  end if;
```

```
      when thirteen =>
        if direction = '0' then
          count_temp <= fifteen;
        else
          count_temp <= nine;
        end if;

      when nine =>
        if direction = '0' then
          count_temp <= thirteen;
        else
          count_temp <= one;
        end if;

    end case;
  else
    count_temp <= count_temp;
  end if;
end if;
```

Finally, assign the temporary `count_temp` to `count`. The name of each `count_temp` corresponds to the binary number of `count`.

```
case count_temp is
```

```
  when one => count <= "0001"; --1
  when two => count <= "0010"; --2
  when four => count <= "0100"; --4
  when eight => count <= "1000"; --8
  when three => count <= "0011"; --3
  when six => count <= "0110"; --6
  when twelve => count <= "1100"; --12
  when eleven => count <= "1011"; --11
  when five => count <= "0101"; --5
  when ten => count <= "1010"; --10
  when seven => count <= "0111"; --7
  when fourteen => count <= "1110"; --14
  when fifteen => count <= "1111"; --15
  when thirteen => count <= "1101"; --13
  when nine => count <= "1001"; --9
end case;
end process;
end behaviour;
```

SIMULATION

After making sure that the code is without error by compiling it successfully, a testbench of our code is generated by Quartus. We then edit the testbench by changing the inputs, and simulate the circuit in ModelSim. Since the DE1-SoC board works at a frequency of

50 MHz, we set the clock in our circuit with a period of 20 ns. Note that the `WAIT;` in the default testbench has to be deleted, otherwise the clock will not work properly.

```
BEGIN
  clk <= '1';
  WAIT FOR 10 ns;
  clk <= '0';
  WAIT FOR 10 ns;
```

Then, the other inputs, reset and enable are assigned and changed as well.

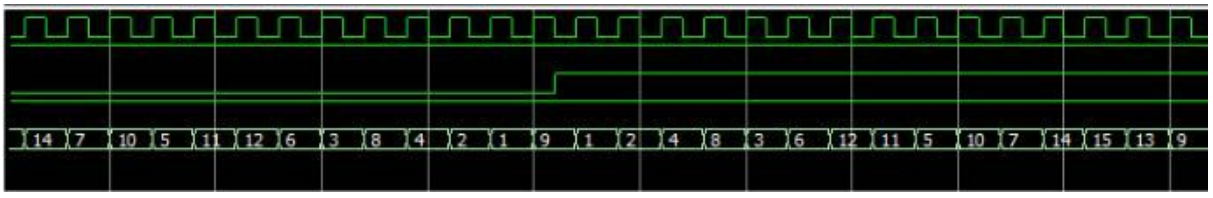
```
BEGIN
  reset <= '1';
  enable <= '0';
  direction <= '0';
  WAIT FOR 2 s;
  reset <= '0';
  enable <= '0';
  direction <= '0';
  WAIT FOR 2 s;
  reset <= '0';
  enable <= '1';
  direction <= '0';
  WAIT FOR 5 s;
```

```
  direction <= '1';
WAIT;
```

The wave graph shows that the `reset`, `direction` and `enable` inputs are functioning normally. The inputs change accordingly to the assignment, and the output stays constant whenever `enable` is 1. The `reset`, `direction` and `enable` are changed independently as well to see if each of them functions as expected, and they did. Only when `reset` equals 0 and `enable` equals to 1 simultaneously, the output value starts to accumulate at the rising edge of `count`.

Also, when `direction` changes, the `count` does change into another direction.

These show that the simulation is successful.



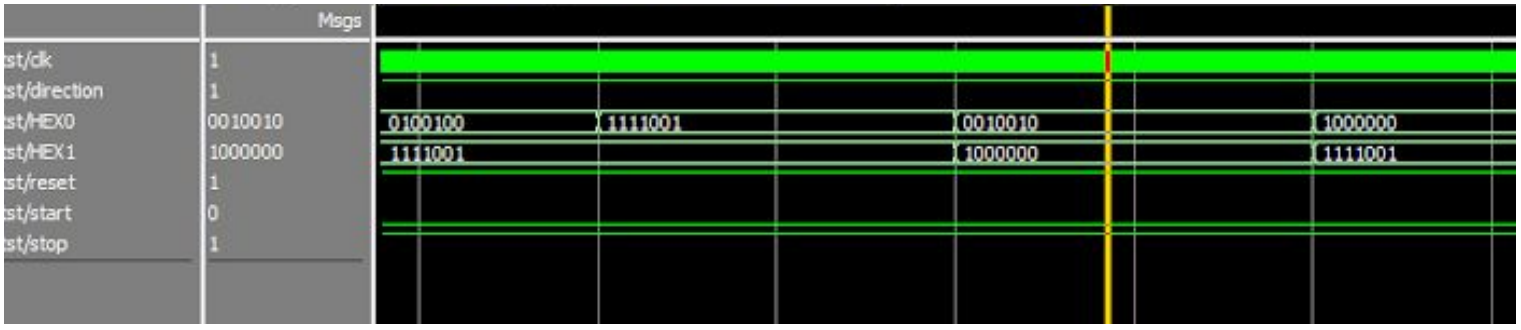
Multi Mode Counter

The multi mode counter combines the finite state machine, clock divider and 7-segment decoder circuits we implemented before with five inputs which are clk, start, stop, reset and direction. Start, stop and reset are active low. For direction, 0 indicates going to the left and 1 indicates going to the right in the diagram given in the lab description. The outputs of this circuit are 2 digits in hexadecimal form that shows the current state of the circuit. The clock

divider creates a clock for the counter to count up so that the digits change at every one second. As described in the finite state machine circuit, the counter will count to the next state according to the sequence defined in the finite state machine. Every time the counter counts from one state to the next state, the 7-segment decoders display the state on the FPGA board.

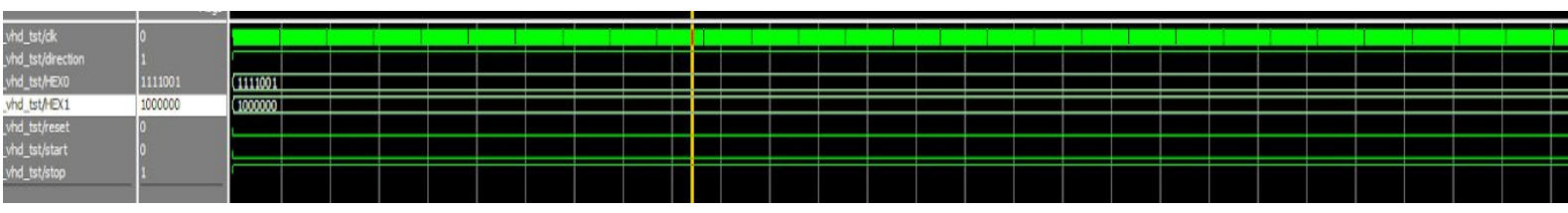
Testing of Multi Mode Counter

At first, we tested the circuit on ModelSim. After that, we used the FPGA board to test it. To test it with ModelSim, we wrote a testbench code that set direction to 0, start to 0 and stop and reset to 1.



The figure above is a screenshot of the ModelSim simulation. It shows that after certain time period, the state changes according to the sequence in finite state machine from one to another. Also, when stop is set to be 0, the number is always the current state.

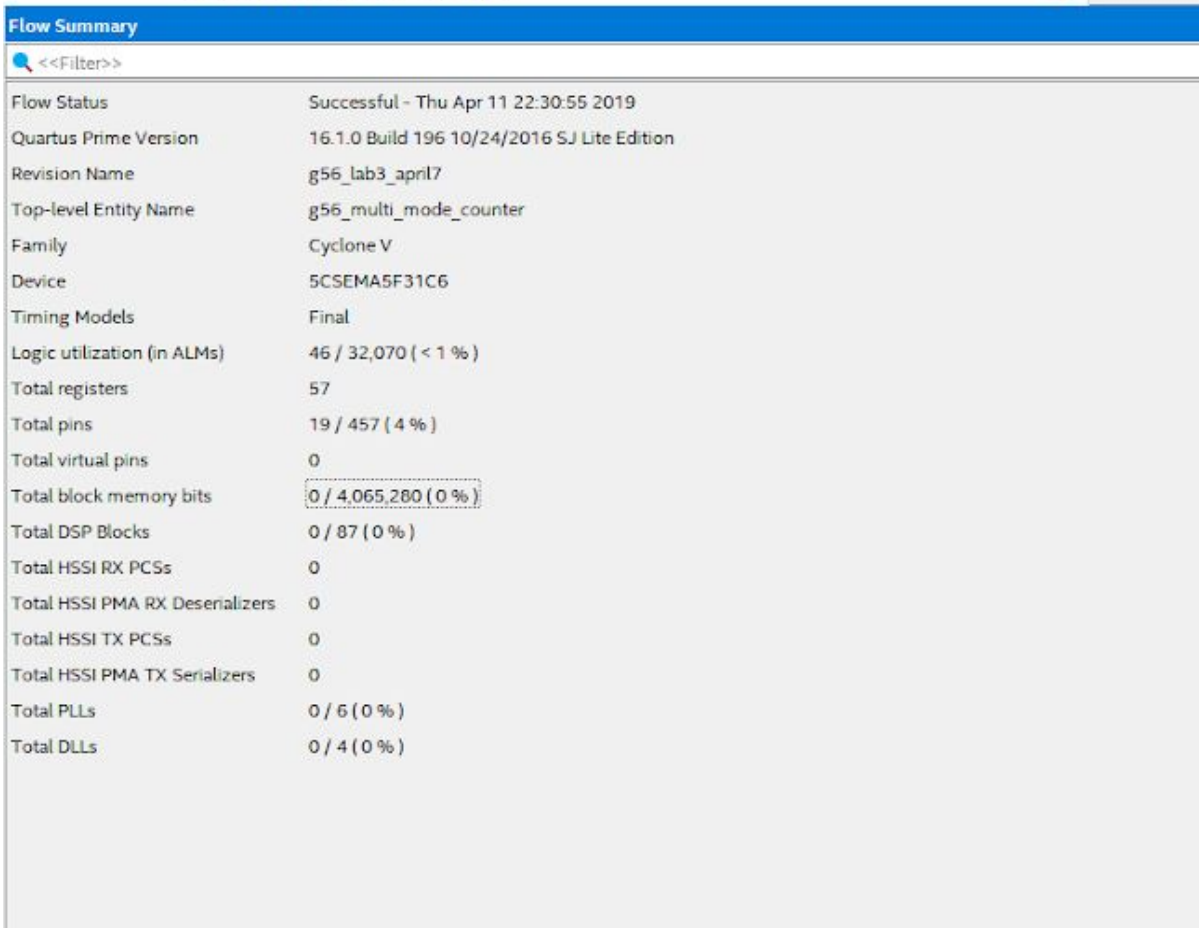
To test reset, we set start and reset to 0(active) and stop to 1.



As shown in the simulation, the circuit is always at the initial state since the reset is active. To test direction, we set start and direction to 0 and reset and stop to 1, we found that the circuit is counts to the left. When we set direction to 1 and others remain unchanged, the circuit counts to the right.

To test the circuit on the board, at first we pushed start, the circuit counts to the right as specified by the finite state machine. When we pushed stop, the counter stopped working. Then we flipped the direction using the switch, the circuit counts to a opposite direction. After pushing reset, everything resets to the initial states which are 1 or 9 depending on the direction. Thus, we can conclude that the circuit is functioning properly.

FPGA Resource Utilization



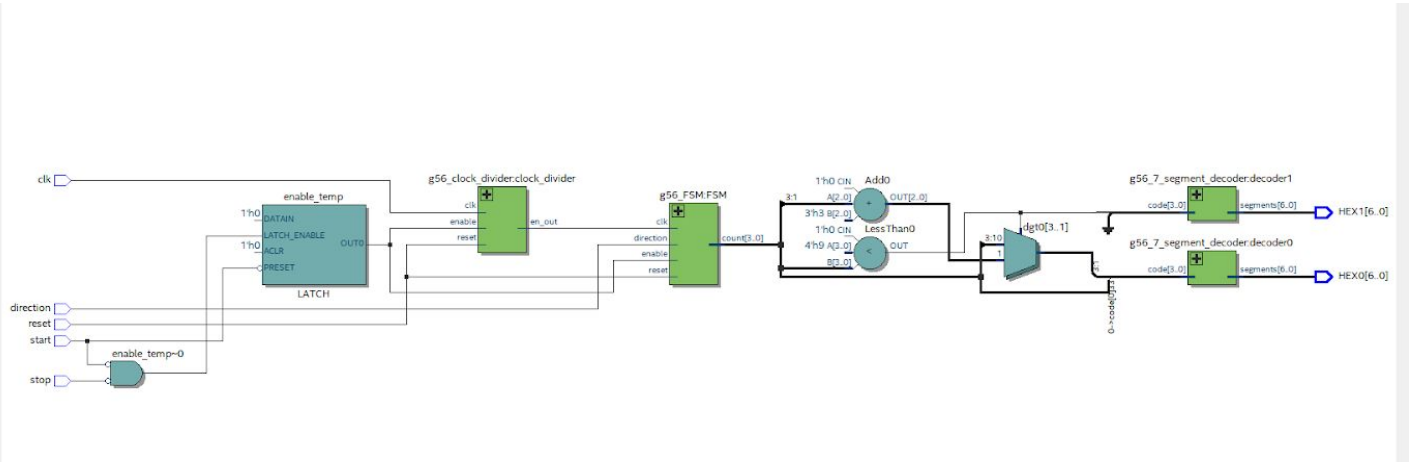
Flow Summary

<<Filter>>

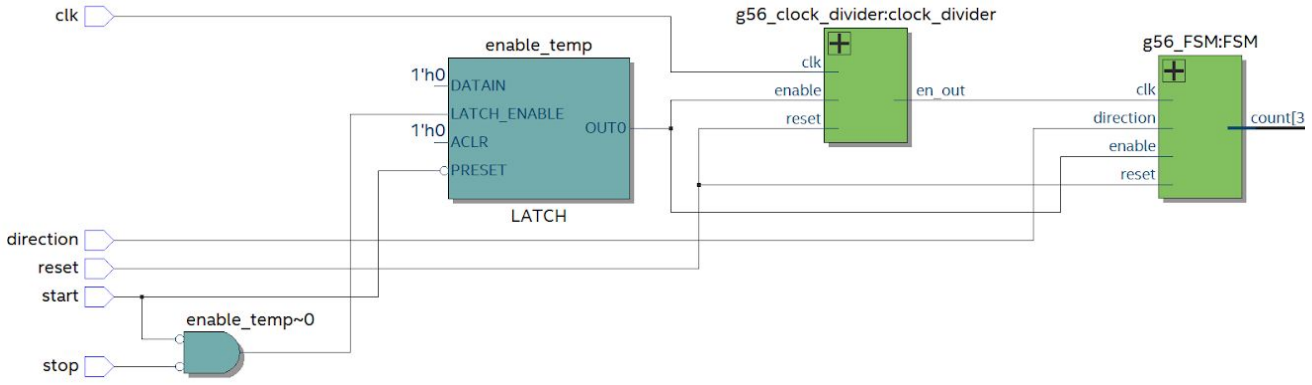
Flow Status	Successful - Thu Apr 11 22:30:55 2019
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g56_lab3_april7
Top-level Entity Name	g56_multi_mode_counter
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	46 / 32,070 (< 1 %)
Total registers	57
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 %)

The device suggests what device we are using to run the VHDL code. Logic utilization tells how much of the device used to implement our circuit which is 1%. Total pins shows how many pin assignments we used. The first line tells the user that the compilation is successful.

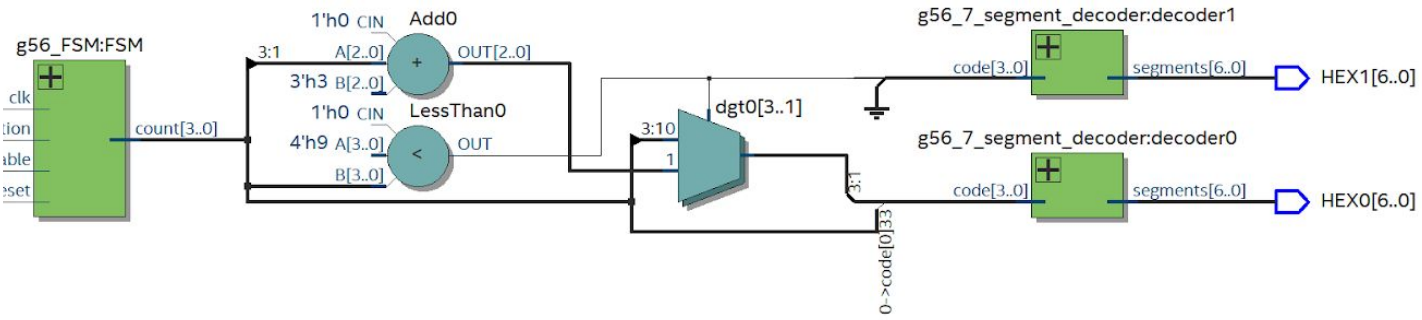
RTL Schematic Diagram



The figure above gives the whole circuit for the multi mode counter. This circuit includes one clock divider, one finite state machine and two 7-segment decoders.



The first part of this circuit is shown above. The clock divider circuit takes a clock as input and outputs another clock which controls the FSM's clock so that the FSM changes to next state every one second. As long as the the enable is active and reset is inactive for the FSM, the FSM will loop in the cycle written in the code.



The above screenshot show the second part of the multi mode counter circuit. In this sub circuit, the FSM outputs a value which then gets decoded by the 7-segment decoders. The 7-segment decoders then display the value on the board in decimal format.