ECSE 222: Lab 3 Report
Finite State Machines

Group #48
Section 007
Dafne Culha (260785524)
Cheng Lin (260787697)

Table of Contents

I Introduction

This laboratory experiment gave us a better understanding of finite state machines (FSMs), Intel's Quartus Prime software, and the ModelSim software through the programming of a bi-directional counter and a multi-mode counter. We used the VHDL language, Quartus Prime's Test Bench Template Writer, ModelSim, and an Altera board to produce the circuits required. The final product was a programmed Altera DE1-SoC board that counts a certain sequence of numbers in either direction and can be controlled with start, stop, and reset pushbuttons.

II Bi-Directional Counter Finite State Machine

The first circuit we designed was a bi-directional counter FSM that counts up/down in the following sequence:

$$1 \longleftrightarrow 2 \longleftrightarrow 4 \longleftrightarrow 8 \longleftrightarrow 3 \longleftrightarrow 6 \longleftrightarrow 12 \longleftrightarrow 11 \longleftrightarrow 5 \longleftrightarrow 10 \longleftrightarrow 7 \longleftrightarrow 14 \longleftrightarrow 15 \longleftrightarrow 13 \longleftrightarrow 9\ (\longleftrightarrow 1 \longleftrightarrow 2 \longleftrightarrow 4 \ldots)$$

Furthermore, the counter will be controlled by an active high enable input, a direction input, and an active low reset input. When the reset input becomes active, the clock will become 1 if it is counting up, and will become 9 if it is counting down (this is indicated by the direction signal).

II.I FSM design

Prior to designing the counter in Quartus Prime, we drew the state diagram for the circuit (Figure 1). Because the counter increments through a loop of 15 values, we designed the FSM to have 15 states labelled A through O. Each of these states corresponds to a unique output *z*, the 4-bit binary representation of the number the state represents. The edges between states are dictated by the value of the direction signal *w*. Lastly, as indicated in the diagram, the state the FSM adopts when reset is pressed is also dependent on the value of *w*.

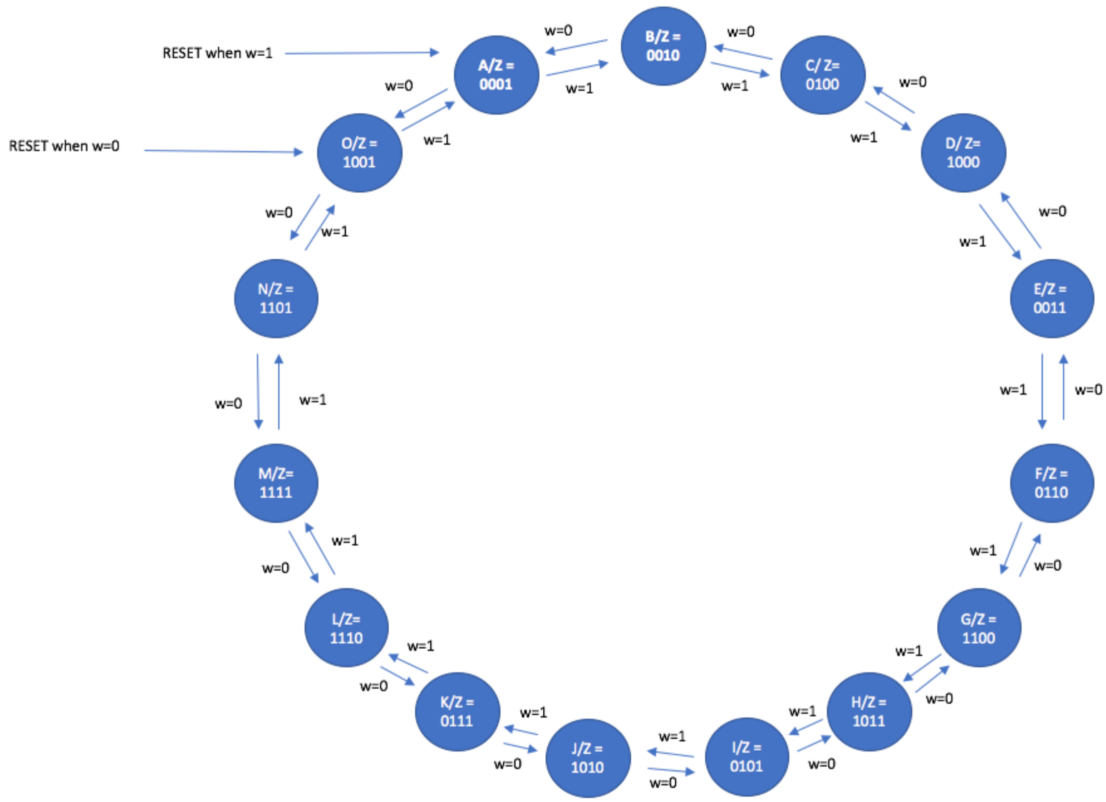


Fig. 1: State diagram for FSM.

Once we drew the state diagram, we began creating the circuit in VHDL. To begin, we created the entity declaration for the FSM, specifying the four inputs and single output of the circuit.

```
entity g48_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 g48_FSM;
```

Next, in the architecture of the counter, we declare a “state_type” signal type and create a variable count_temp that is of that type. Count_temp can therefore take on values A through O, where each letter represents a state in the FSM.

```
--declare architecture for FSM
architecture behaviour of g48_FSM is
   --create a state_type signal type to hold each of the counter's state
   --each consecutive letter corresponds to the following number in the sequence
   --A->1, B->2, C->4, D->8, etc.
   type state_type is (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O);

   --create signal to hold state of count
   signal count_temp : state_type := A;

```

Finally, in the second part of the architecture (below), we create a process that includes the clock and reset inputs as variables on its sensitivity list. This means that the process will execute when there is a change in either of those values. The clock is on the sensitivity list because the counter should increment up/down every clock cycle; the reset value is on the list because it is an asynchronous signal.

Within the process, we first check if the reset input is active (equals 0) and restart the count_temp accordingly (depending on the value of the direction signal).

```
begin

   --declare a process block since this is a sequential circuit
   --define clk and reset in sensitivity list as variables we keep track of
   --all other variables are synchronized with clk

   Process(clk, reset) begin
      --check the value of reset first because it is an asynchronous signal
      --reset is active low
      if(reset = '0') then
         --if direction is 1, we are counting up
         --reset counter to state A
         if (direction = '1') then
            count_temp <= A;
         --otherwise we are counting down, reset counter to state O
         else
            count_temp <= O;
         end if;

```

Otherwise, if the clock has a rising edge and our enable input is equal to 1, we increment count_temp to the next state in the sequence (i.e. A → B, B → C, etc., or O → N, N → M, etc.). This next state is a function of the direction signal, which tells us if we are counting up or down.

We use a case statement to update the value of count_temp. The logic for incrementing count_temp when it equals A in the current cycle is shown below (lines 64-69); the control flow for all other state values are very similar.

```
--check for rising edge of clk
elsif(rising_edge(clk)) then

   --check if enable is on (enable is active high)
   if(enable = '1') then
      --check for counting direction
      --direction = 1 means count up, direction = 0 means count down)
      case count_temp is
         --when count_temp = A, we either increment to B or decrement to O
         when A =>
            if (direction = '1') then
               count_temp <= B;
            else
               count_temp <= O;
            end if;
         when B =>
            if (direction = '1') then
               count_temp <= C;
```

Lastly, outside of the process block, we associate a value to the output signal, count, for every state that count_temp may take. This section of the code corresponds to the combinational circuit for the output logic of the FSM.

```
   --set count output for a given count_temp state
   count <= "0001" when count_temp = A else
            "0010" when count_temp = B else
            "0100" when count_temp = C else
            "1000" when count_temp = D else
            "0011" when count_temp = E else
            "0110" when count_temp = F else
            "1100" when count_temp = G else
            "1011" when count_temp = H else
            "0101" when count_temp = I else
            "1010" when count_temp = J else
            "0111" when count_temp = K else
            "1110" when count_temp = L else
            "1111" when count_temp = M else
            "1101" when count_temp = N else
            "1001" when count_temp = O;

end behaviour;
```

II.II FSM simulation

We used ModelSim to test our FSM and simulate the values of inputs and outputs over time. We compiled our .vhd file on Quartus, added the same file to ModelSim, then used the Quartus Testbench Template Writer to create an empty testbench file. Lastly, we added simulation processes to the .vht file and simulated the circuit on ModelSim.

Firstly, we use the init process to loop over clk values.

```
--init process to loop clock values
init : PROCESS
BEGIN
		--FPGA board's clock has 50 MHz frequency
		--this means one period every 20 ns (so change clk every 10 ns)

		clk <= '1';
		WAIT FOR 1ns;
		clk <= '0';
		WAIT FOR 1ns;
END PROCESS init;
```

Then, we use the always process to test reset, enable and counting up / down functionalities of FSM.

```
BEGIN

		--set initial values
		reset <= '1';
		enable <= '1';
		direction <='1';

		--test reset
		WAIT FOR 45ns;
		reset<='0';
		WAIT FOR 10ns;
		reset <= '1';

		--test enable
		WAIT FOR 34ns;
		enable <= '0';
		WAIT FOR 25ns;
		enable <= '1';

		--test direction
		WAIT FOR 34ns;
		direction <='0';
		WAIT FOR 41ns;
		reset<='0';
		WAIT FOR 2ns;
		reset <='1';

WAIT;
END PROCESS always;
```

Firstly, we set the initial values of reset, enable and direction as ‘1’.

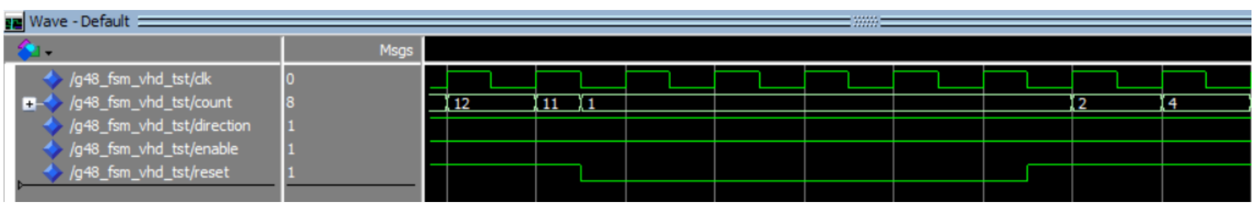
```
BEGIN

                --set initial values
                reset <= '1';
                enable <= '1';
                direction <='1';
```

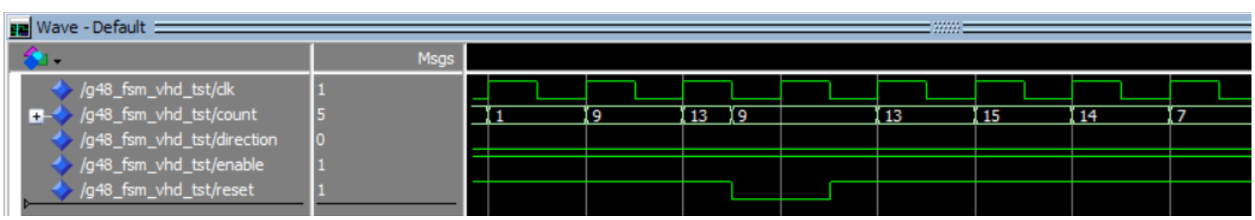
To test reset for both directions, we change its value from ‘1’ to ‘0’.

```
                --test reset
                WAIT FOR 45ns;
                reset<='0';
                WAIT FOR 10ns;
                reset <= '1';
```

To test the reset function while counting upwards, we set the value of reset to ‘0’ while direction has a value of ‘0’. As seen from the waveform below, when reset becomes active, the count returns to 1, as expected.



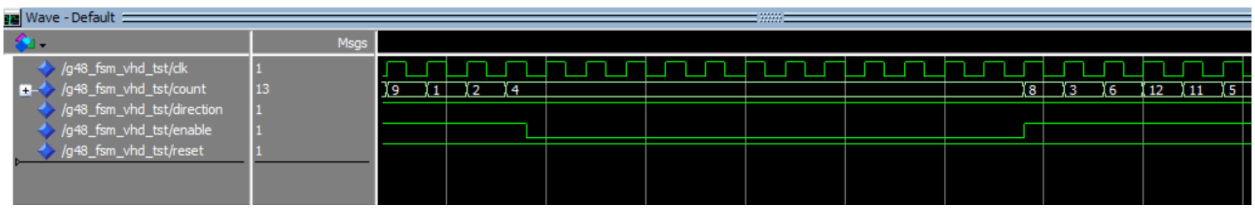
To test the reset function while counting downward, we set the value of reset as ‘0’ while direction has a value of ‘1’. As can be seen from the waveform below, when reset becomes active, the count returns to 9.



Enable controls whether the count value increments or not each clock cycle. To test enable, we change its value from ‘0’ to ‘1’.

```
--test enable
WAIT FOR 34ns;
enable <= '0';
WAIT FOR 25ns;
enable <= '1';
```

As can be seen from the waveform below, the count value is no longer updated at the positive edge of the clock when enable is set to ‘0’. Furthermore, when enable is set back to ‘1’, the count variable begins changing again.

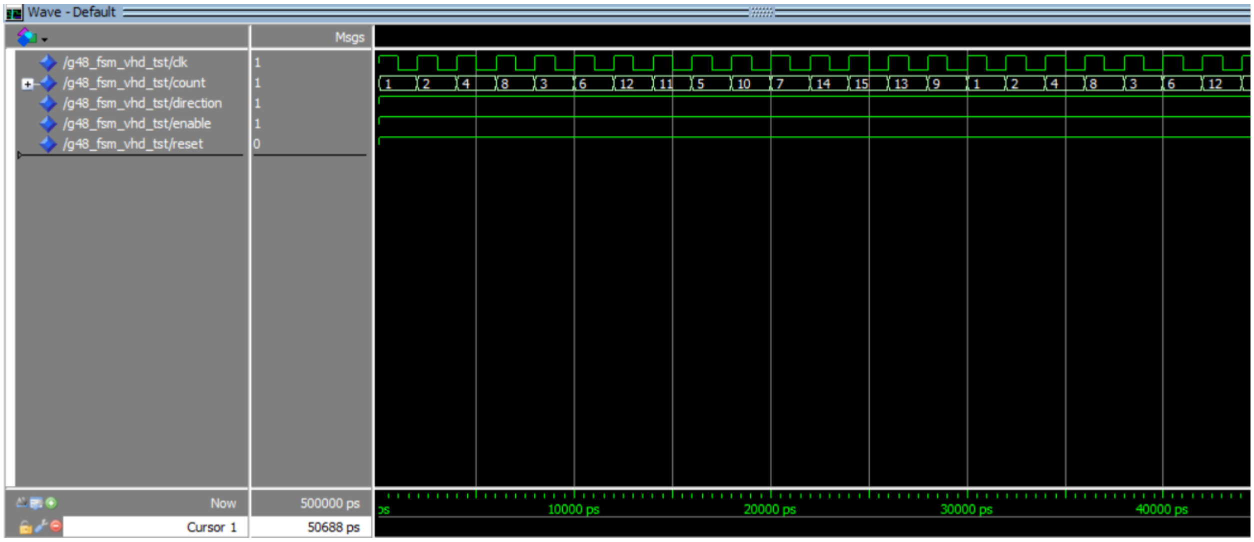


To test if counting up/down functionalities of FSM works overall, we set the direction to ‘0’ and ‘1’ and wait long enough to see if it follows the correct patterns and if it loops:

1↔2↔4↔8↔3↔6↔12↔11↔5↔10↔7↔14↔15↔13↔9 (↔1↔2↔4...)

```
--test direction
WAIT FOR 34ns;
direction <='0';
WAIT FOR 41ns;
reset<='0';
WAIT FOR 2ns;
reset <='1';
```

For example, when direction is set to ‘1’, the count variable increments and loops through the cycle correctly.



A similar result is seen when we set the direction variable to '0'. Therefore, by running the FSM on ModelSim, we have confirmed that it works as expected.

III Multi-Mode Counter

The second circuit we built was a multi-mode counter: a sequential circuit that accepts start, stop, and reset inputs to control the FSM that was designed in the previous section. This circuit will accept the Altera board's 50 MHz PLL clock and use a clock divider to assert a change in the FSM every second. Furthermore, the multi-mode counter will use two 7-segment decoders to show the output of the FSM as two-digit decimal numbers on 7-segment LEDs.

III.I Counter design

To begin, we declare the counter as an entity that accepts a start, stop, direction, reset, and clock input and returns two 7-bit vector outputs.

```
--declare an entity with five boolean inputs (start, stop, direction, reset, clk)
--and two 7-bit vector outputs for two 7-seg displays

entity g48_multimode_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 g48_multimode_counter;
```

In the architecture of the counter, we begin by importing the FSM component, the clock divider component, and the 7-segment decoder component in the workspace. It should be noted that the clock divider component in this circuit has a generic T constant of 50000000 and will assert an en_out of 1 every second given a 50 MHz clock input.

```vhdl
--declare architecture for FSM
architecture behaviour of g48_multimode_counter is

   --import the FSM component
   component g48_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 component g48_FSM;

   --import the clock divider component
   component g48_clock_divider is
      Port(enable : in std_logic;
         reset : in std_logic;
         clk : in std_logic;
         en_out : out std_logic);
   end component g48_clock_divider;

   --import 7 seg decoder component
   component g48_7_segment_decoder is
      Port(code: in std_logic_vector(3 downto 0);
         segments: out std_logic_vector(6 downto 0));
   end component g48_7_segment_decoder;

```

Then, we instantiate five signals: one signal to store the output of the clock divider, one signal to store the temporary enable input for the sequential circuit components, one signal to store the output of the FSM, and two signals to store the BCD digits that will be inputted into the 7-segment decoders.

```vhdl
   --declare signal to hold clock from divider
   signal divided_clk : std_logic;
   --declare signal to hold temporary enable, instantiate as 0
   signal enable_temp : std_logic := '0';

   --declare signal to hold temporary count
   signal count_temp : std_logic_vector(3 downto 0);
   --declare signals to hold BCD digits of count_temp
   signal digit0 : std_logic_vector(3 downto 0);
   signal digit1 : std_logic_vector(3 downto 0);

```

Next, we instantiate a clock divider and FSM component in the architecture and specify their port maps. Note that both the clock_divider and the FSM share the same enable and reset input. However, the output of the clock divider acts as the *clock* to the FSM component. This is because the FSM should see a rising edge once every second, the exact frequency in which the clock divider asserts an output of 1. Lastly, the output of the FSM goes to count_temp, which will be used to obtain the digit0 and digit1 vector inputs later on.

```
begin

   --create clock divider
   --clock divider is controlled by reset, clk, and enable_temp variables
   --it outputs divided_clk which acts as the clock for to FSM
   clock_divider : g48_clock_divider PORT MAP(enable => enable_temp,
                                               reset => reset,
                                               clk => clk,
                                               en_out => divided_clk);

   --create an FSM component
   FSM : g48_FSM PORT MAP(enable => enable_temp,
                          direction => direction,
                          reset => reset,
                          clk => divided_clk,
                          count => count_temp);
```

We now create a process that is sensitive to the start, stop, and count_temp inputs. These variables are included on the sensitivity list because the counter needs to respond to any changes in these three variables. The start/stop variables appear because the enable_temp signal is updated each time they change; the count_temp variable is included because digit0 and digit1 need to be updated whenever count_temp increments.

To set the enable_temp value in the process block, we begin by checking whether start = 0 or stop = 0 (both are active low). Depending on whether start or stop are active, enable_temp (an active high signal) is set to 1 or 0 respectively. The enable_temp signal controls whether or not the clock divider and the FSM increment each clock cycle.

```
   Process(start, stop, count_temp) begin

      --check value of start (active low)
      if(start = '0') then
         enable_temp <= '1'; --if start = 0, enable stopwatch

      --check value of stop (active low)
      elsif(stop = '0') then
         enable_temp <= '0'; --if stop = 0, turn enable off

      end if;
```

Alternatively, if start and stop are both 1 and the process block is activated by a change in count_temp, the circuit will update the values of digit0 and digit1. Because the outputs of HEX0 and HEX1 must correspond to a 7-segment display's decimal representation of count_temp, digit0 and digit1 are encoded in BCD format.

To convert count_temp into two decimal digits, we have to first check if count_temp is greater than 1001_2 (or 9_{10}). If count_temp is greater than 9, digit0 is corrected by adding 0110_2 (or 6_{10}) to it, and digit1 becomes 0001_2. Alternatively, if count_temp is not greater than 9, digit0 becomes count_temp and digit1 becomes 0. These if-statements generate the correct inputs for the two 7-segment decoders used in the next code snippet.

```
      --convert count_temp to BCD digits stored in digit0 and digit1
      --start by checking if count_temp is greater than 9
      if(count_temp > "1001")then
      --if count_temp greater than 9, convert to BCD
         digit0 <= std_logic_vector(unsigned(count_temp) + "0110"); --conver to BCD by adding 6
         digit1 <= "0001";

      else
      --if count_temp is less than 9, set digit0 = count_temp and digit1 = 0
         digit0 <= count_temp;
         digit1 <= "0000";
      end if;

   end Process;
```

Finally, at the end of the architecture, we create two 7-segment decoder entities. The inputs to these decoders are digit0 and digit1, the two 4-bit BCD variables whose values are set in the process block. The outputs to these decoders are HEX0 and HEX1, which are fed into the 7-segment LED displays on the FPGA.

```
   --decoder for counter 0
   decoder0: g48_7_segment_decoder PORT MAP(code => digit0,
                         segments => HEX0);
   --decoder for counter 1
   decoder1: g48_7_segment_decoder PORT MAP(code => digit1,
                         segments => HEX1);

end behaviour;
```

III.II Counter testing

We tested the inputs with a programmed Altera FPGA and verified the outputs displayed by the board. We pushed the start button and waited to make sure the values displayed are correct and the circuit loops in both directions. We checked if the buttons are working correctly and if our circuit started, stopped, and reset as expected. Lastly, we verified the values of the clock reset for both directions.

IV FPGA Resource Utilization

Flow Summary

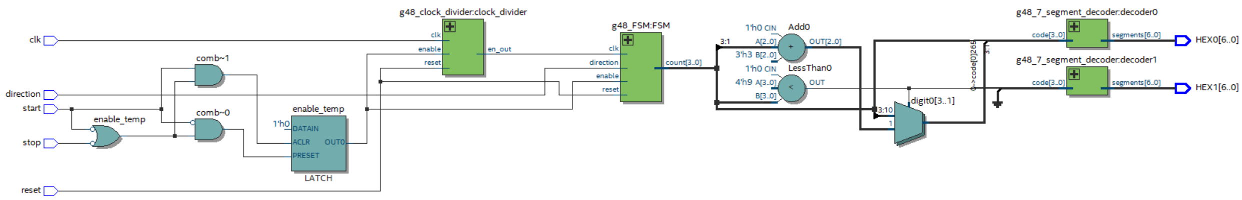
<<Filter>>

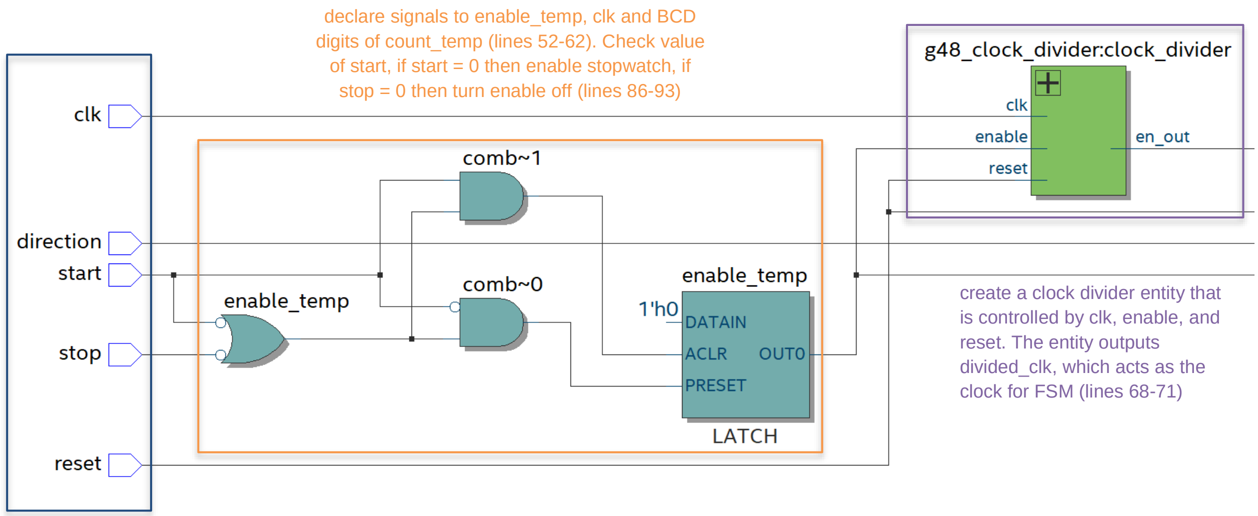
Flow Status	Successful - Tue Apr 02 23:38:32 2019
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	g48_lab3
Top-level Entity Name	g48_multimode_counter
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	55 / 32,070 (< 1 %)
Total registers	62
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 %)

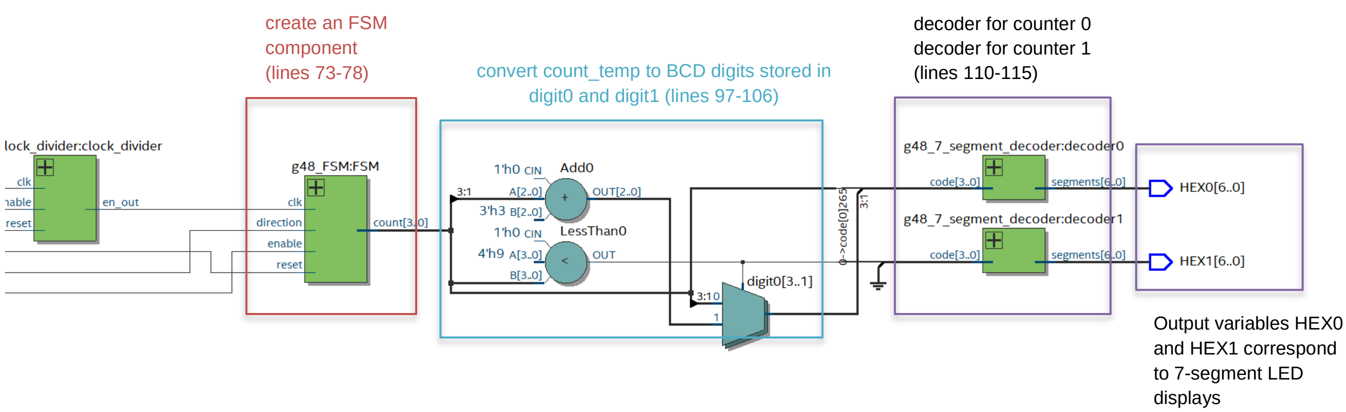
Logic utilization measures how much of the device is used to implement our circuit. As seen above, our compilation flow summary indicates that our logic utilization is less than 1%. This means that we have used very little of the board's total resources to implement our counter.

V RTL Schematic of Multi-mode counter

The following is the RTL schematic of the multi-mode counter. For clarity, the schematic will be split up into two sections and each component's function (and corresponding lines of code) will be labelled on the visuals below. In summary, each green component represents a circuit entity (clock divider, FSM, or decoders). As well, the blue square is a register to store the enable input, and the other circuit components either check for the start/stop values or convert the temporary count signal to BCD format.







The following is another copy of our VHDL code for line references in the RTL diagram:

```
1   --ECSE 222 Lab 3
2   --Group 48
3   --Dafne Culha (260785524), Cheng Lin (260787697)
4   --01 Apr 2019
5
6
7   library IEEE;
8   use IEEE.STD_LOGIC_1164.ALL;
9   use IEEE.NUMERIC_STD.ALL;
10  use work.ALL;
11
12  --declare an entity with five boolean inputs (start, stop, direction, reset, clk)
13  --and two 7-bit vector outputs for two 7-seg displays
14
15  entity g48_multimode_counter is
16          Port (start : in std_logic;
17                      stop : in std_logic;
18                      direction : in std_logic;
19                      reset : in std_logic;
20                      clk : in std_logic;
21                      HEX0 : out std_logic_vector(6 downto 0);
22                      HEX1 : out std_logic_vector(6 downto 0));
23
24  end g48_multimode_counter;
```

```vhdl

--declare architecture for FSM
architecture behaviour of g48_multimode_counter is

	--import the FSM component
	component g48_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 component g48_FSM;

	--import the clock divider component
	component g48_clock_divider is
		Port(enable : in std_logic;
			reset : in std_logic;
			clk : in std_logic;
			en_out : out std_logic);
	end component g48_clock_divider;

	--import 7 seg decoder component
	component g48_7_segment_decoder is
		Port(code: in std_logic_vector(3 downto 0);
			segments: out std_logic_vector(6 downto 0));
	end component g48_7_segment_decoder;

	--declare signal to hold clock from divider
	signal divided_clk : std_logic;
	--declare signal to hold temporary enable, instantiate as 0
	signal enable_temp : std_logic := '0';


	--declare signal to hold temporary count
	signal count_temp : std_logic_vector(3 downto 0);
	--declare signals to hold BCD digits of count_temp
	signal digit0 : std_logic_vector(3 downto 0);
	signal digit1 : std_logic_vector(3 downto 0);

begin

	--create clock divider
	--clock divider is controlled by reset, clk, and enable_temp variables
	--it outputs divided_clk which acts as the clock for to FSM
	clock_divider : g48_clock_divider PORT MAP(enable => enable_temp,
									reset => reset,
									clk => clk,
									en_out => divided_clk);

	--create an FSM component
	FSM : g48_FSM PORT MAP(enable => enable_temp,
									direction => direction,
									reset => reset,
									clk => divided_clk,
									count => count_temp);

```

```
        --declare a process block since this is a sequential circuit with memory
        --define start and stop in sensitivity list as variables we keep track of

        Process(start, stop, count_temp) begin

                --check value of start (active low)
                if(start = '0') then
                        enable_temp <= '1'; --if start = 0, enable stopwatch

                --check value of stop (active low)
                elsif(stop = '0') then
                        enable_temp <= '0'; --if stop = 0, turn enable off

                end if;

                --convert count_temp to BCD digits stored in digit0 and digit1
                --start by checking if count_temp is greater than 9
                if(count_temp > "1001")then
                --if count_temp greater than 9, convert to BCD
                        digit0 <= std_logic_vector(unsigned(count_temp) + "0110"); --conver to BCD by adding 6
                        digit1 <= "0001";

                else
                --if count_temp is less than 9, set digit0 = count_temp and digit1 = 0
                        digit0 <= count_temp;
                        digit1 <= "0000";
                end if;

        end Process;

        --decoder for counter 0
        decoder0: g48_7_segment_decoder PORT MAP(code => digit0,
                                                 segments => HEX0);
        --decoder for counter 1
        decoder1: g48_7_segment_decoder PORT MAP(code => digit1,
                                                 segments => HEX1);

end behaviour;
```

VII Conclusion

For this laboratory experiment, we designed a finite state machine and a multi-mode counter on an Altera board which can count a certain sequence of numbers in either direction and can be controlled by start, stop, reset variables. Overall, this lab experiment gave us a better understanding of finite state machines (FSMs), Intel's Quartus Prime software, and the ModelSim software.