ECSE 222 Fall 2022
VHDL Assignment #6 Report: Storage Elements, Sequential Circuits, and Finite State Machines in VHDL

Part 1 – Participants

Karim Elgammal 260920556

Lutming Wang 261006348

Part 2 – Executive Summary

Throughout this lab we have been able to explore implementations of storage elements, and sequential circuits. Furthermore, with these implementations we were introduced to implementations of these circuits using sequential assignment statements in VHDL, types of statements that allow for sequential circuits to be implemented. Up to this point we really have only been using concurrent assignment statements to implement our circuits and so this lab served as a familiarization of the sequential type of assignments. Most of the implementations within this lab are behavioral implementations that take advantage of this introduction to sequential assignment statements. Some have used a structural style of implementation. Moreover, we were able to design a counter, clock divider, a 3-bit counter, a sequence detector and finally a sequence counter. After implementing each circuit, we then used the FPGA board in order to test our working circuits.

Part 3, 4 & 5: Implementation and Results

JK Flip-Flop Storage Element

Throughout this lab, we have implemented two different types of JKFF. First, one that does not take a reset signal into the consideration of the storage element, and secondly one that does. This served to be helpful when implementing the counter as we were able to implement it in a structural method rather than behavioral which was more efficient as the validity of the design seemed more promising (more efficient with testing etc). The JK FF itself was implemented using a behavioral architectural style, allowing us to describe the behavior of the storage element based on the inputs.

```
13  architecture beh of karim_elgammal_jkff is
14  signal temp : std_logic;
15
16  begin
17  process (clk)
18      begin
19      if rising_edge(clk) then
20          if ( J = '0' AND K = '0') then
21              temp <= temp;
22          elsif (J = '0' AND K = '1') then
23              temp <= '0';
24          elsif (J = '1' AND K = '0') then
25              temp <= '1';
26          elsif (J ='1' AND K = '1') then
27              temp <= NOT (temp);
28
29          end if;
30      end if;
31      end process;
32      Q <= temp;
33
34  end beh;
```

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

Figure 1: Implementation of JKFF and Characteristic Table of JKFF

The JKFF was implemented using nested if statements that allowed the logic to be executed when the clock signal changes, and as seen in figure 1, we then go through the different cases/possibilities of the values of the J and K signals. This signal assignment corresponds to the characteristic table of a JKFF provided in figure 1. We used a temporary signal to store the value

of the intermediate output. The karim_elgammal_jkff_Re, is pretty much identical to the above JKFF implementation with the exception of a reset signal; and instead of using if statements for each case, we only used an if statement to detect the clock and the reset signal (active low) and used the case statement to assign the output as per the characteristic table.

```
13  architecture beh of karim_elgammal_jkff_Re is
14
15   signal jk : std_logic_vector (1 downto 0) := "00";
16   signal sigQ : std_logic := '0';
17
18  begin
19
20   jk <= J&K ;
21
22  process(reset, clk)
23   begin
24  if reset = '0' then sigQ <= '0';
25  elsif rising_edge(clk) then
26  case(jk) is
27         when "00" => sigQ <= sigQ;
28         when "01" => sigQ <= '0';
29         when "10" => sigQ <= '1';
30         when others => sigQ <= not (sigQ);
31  end case;
32  end if;
33   end process;
34
35   Q <= sigQ;
36   end beh;
```

Figure 2 – Implementation of JKFF with Reset Signal

Results:

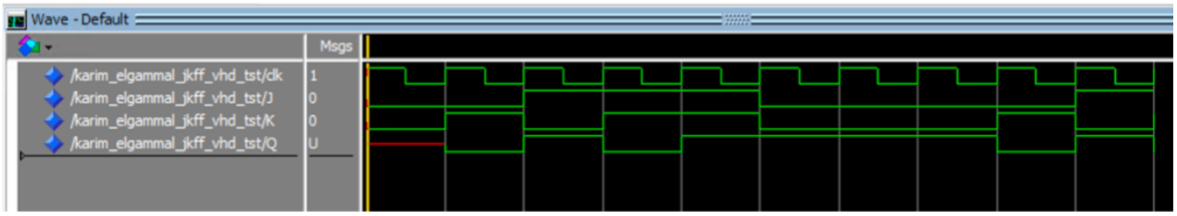


Figure 3- Waveform of JKFF Implementation

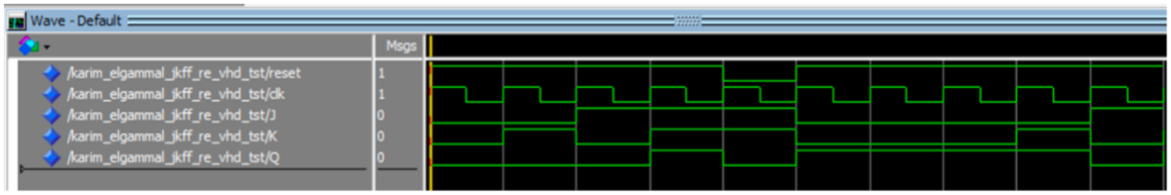
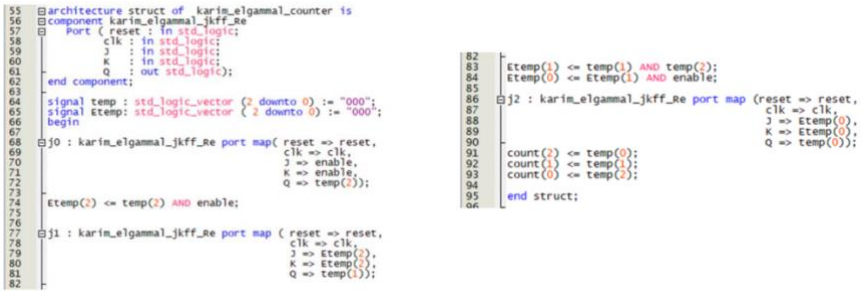


Figure 4 – Waveform of JKFF with Reset

As seen above in figure 3, the JKFF behaves as expected, where depending on the values of J and K, the output is either 0, 1 or latches on to the previous state, or the complement of the previous state. This can be observed at different stages of the wave form and is consistent over our simulation range of 100 ns. The same goes for the JKFF with the reset signal which is seen in figure 4. We can see that once the reset signal is active (low), the output and the JKFF is reset to 0.

Counter

For the implementation of the 3-bit up counter we used a structural implementation that utilized the JKFF with a reset signal as its building blocks. We instantiated the JKFF with reset signal three times as seen in figure 5. We have encountered in class and through the textbook circuits of counters that are implemented using storage units, and within them the outputs of the previous storage elements are fed into the next storage element. We have taken on this model and modified it to take into account the enable signal too, as the separate JKFFs already take care of the reset signal. Furthermore, this has allowed us to keep the specification of an asynchronous reset. We used AND gates in order to allow for an enable signal to control the function of the counter. The enable signal is AND-ed with the outputs of the JKFF when being re-inputted in the following JKFF, allowing us to 'freeze' the counter, while keeping the clock running. This method avoids gating the clock and serves efficient in design.



```
architecture struct of karim_elgammal_counter is
component karim_elgammal_jkff_Re
    Port ( reset : in std_logic;
           clk : in std_logic;
           J   : in std_logic;
           K   : in std_logic;
           Q   : out std_logic);
end component;

signal temp : std_logic_vector (2 downto 0) := "000";
signal Etemp: std_logic_vector ( 2 downto 0) := "000";
begin

j0 : karim_elgammal_jkff_Re port map( reset => reset,
                                      clk => clk,
                                      J => enable,
                                      K => enable,
                                      Q => temp(2));

Etemp(2) <= temp(2) AND enable;


j1 : karim_elgammal_jkff_Re port map ( reset => reset,
                                       clk => clk,
                                       J => Etemp(2),
                                       K => Etemp(2),
                                       Q => temp(1));

Etemp(1) <= temp(1) AND temp(2);
Etemp(0) <= Etemp(1) AND enable;

j2 : karim_elgammal_jkff_Re port map (reset => reset,
                                      clk => clk,
                                      J => Etemp(0),
                                      K => Etemp(0),
                                      Q => temp(0));

count(2) <= temp(0);
count(1) <= temp(1);
count(0) <= temp(2);

end struct;
```

Figure 5 - Implementation of 3-Bit Up Counter with Asynchronous Reset Signal and Enable Signal

Results:

As seen in figure 6, we can observe that the waveform of our simulated design follows the specifications that we were aiming for throughout this lab. We can observe firstly that the counter is able to count from 0 to 7 and then cycles back to 0 after it has reached the maximum number represented by 3 bits (1). Also, when the enable input is low (2), the count is held at the count that it was at, this is seen at 001 within our waveform. Finally, we can observe that when the reset input is low, the count is reset back to 0 (3). We can observe these events through the red numbers on the waveform.

Figure 6 - Waveform of 3-Bit Up Counter Implementation

Clock Divider

The clock divider circuit was implemented using a behavioral style that follows the given circuit within the assignment (figure 6B). Using nested if statements, we have essentially behaviorally implemented a down counter from 9 to 0. More specifically, we have been asked to implement this clock divider to release a signal every second, and with a timing environment of 10hz, this equates to a signal every 10 cycles. Therefore, the count T is 10, while the down counter operates on T-1 as shown in figure 6B. The down counter's outputs are then fed into a NOR gate to realize the signal. Therefore, it only emits a signal when it reaches 0.

```
12  architecture beh of karim_elgammal_clock_divider is
13
14  signal temp : std_logic_vector(3 downto 0);
15
16  begin
17
18  process (clk, reset, enable)
19  begin
20  if reset <= '0' then
21      temp <= "1001";
22      elsif rising_edge(clk) then
23          if enable = '1' then
24              if temp = "0000" then
25                  temp <= "1001";
26              else
27                  temp <= std_logic_vector(unsigned(temp)-1);
28              end if;
29          end if;
30      end if;
31  end process;
32
33  en_out <= not( temp(0) or temp(1) or temp(2) or temp(3));
34
35  end beh;
```

Figure 6A – Behavioral Implementation of Clock Divider

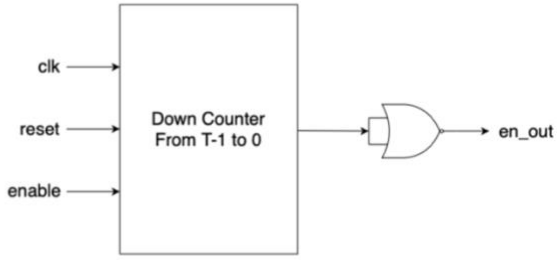


Figure 6B – Logic Circuit of Clock Divider

Results:

As seen from our waveform below, the clock divider asserts a signal every 10 cycles and behaves as expected. The enable and reset signals reset the count within the down counter and enable it respectively.

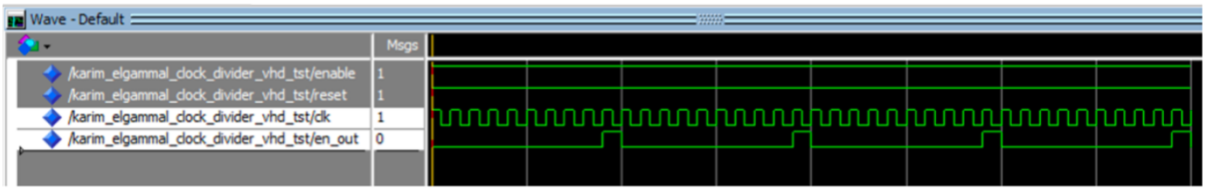


Figure 6C – Waveform of Clock Divider

3-Bit Up Counter Counting in Increments of 1 Second

Using the schematic provided (figure 7B), we were able to join our designs in a structural implementation of a 3-Bit up counter. This counter only receives clock value of 1, every second (or every 10 cycles according to our simulation environment), therefore only counts every 1 second by 1. The output of the clock divider serves as the clock of the counter we designed previously, and the output of that counter is then converted into HEX0, using the 7-segment decoder we

implemented in VHDL 4. This allows us to display the value of the count on the FPGA board. This implementation was the most efficient as it only required us to connect the respective inputs

```
architecture struct of karim_elgammal_wrapper is

component karim_elgammal_counter
port (enable : in std_logic;
      reset : in std_logic;
      clk : in std_logic;
      count : out std_logic_vector (2 downto 0));
end component;

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

component seven_segment_decoder
port ( code : in std_logic_vector (2 downto 0);
segments_out : out std_logic_vector (6 downto 0));
end component;

signal clkdiv_to_counter : std_logic;
signal counter_to_decoder : std_logic_vector(2 downto 0);
```

```
begin

clkdiv : karim_elgammal_clock_divider port map(enable => enable,
                                               reset => reset,
                                               clk => clk,
                                               en_out => clkdiv_to_counter);

cntr : karim_elgammal_counter port map(enable => enable,
                                       reset => reset,
                                       clk => clkdiv_to_counter,
                                       count => counter_to_decoder);

dcdr : seven_segment_decoder port map(code => counter_to_decoder,
                                      segments_out => HEX0);
end struct;
```

Figure 7A – Structural Implementation of Wrapper for 3-Bit Up Counter

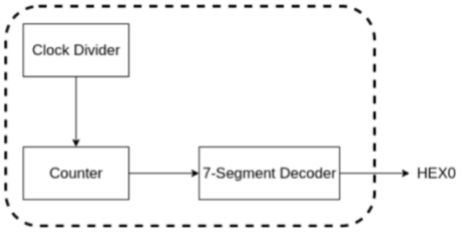


Figure 7B – Schematic of Wrapper for 3 Bit Up Counter

and outputs to each other using temporary carrier signals seen in figure 7A.

Results:

As observed from figure 7C, the 3-bit up counter follows the required behavior described within the instructions of the assignment. The counter adds to the count every 10 cycles, as long as the enable and reset signals are high. Then the counter recycles back to the value of 0 when it has reached the maximum number 7 (represented by 3 bits). Furthermore, we can see from figures 7D and 7E, that the counter also behaves as expected when the enable signal or reset signals are low. We can see that when the enable signal is low, the counter holds the value of the count and then continues counting as soon as the enable signal returns to 1. The asynchronous reset can be observed when the reset signal is 0; the count resets to 0 as soon as the reset signal is low, without respect to the clock, making it an asynchronous component of the design.

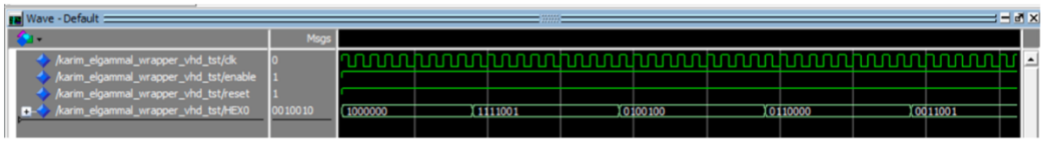


Figure 7C – Waveform Showcasing Count Feature

Figure 7D – Waveform Showcasing Enable Functionality

Figure 7E – Waveform Showcasing Reset Functionality

Critical Path:

The critical path analysis for this entity, follows the same method of finding the critical path of the comparator. We placed timing constraints on each of the inputs/outputs of the component in an SDC file as shown in figure 7F. Then upon compiling the timing analysis, we received the violating paths that occur within the design. Our results seem to indicate a path that lies within the clock divider. The reiteration that takes places during the processes of the clock divider seem to be violating paths even with modifying the timing constraints from a range of 5ns to 100ns. As seen below we can gather that the critical path then, is from the clk signal, and ends at the HEX0 output, whichever index (as it is works like a LUT with the seven segment decoder). The biggest difference, or slack is then -3.811, which allows us to believe that the delay of the critical path, is 5 + 3.811 = 8.811. We can also observe the total UI utilization and total pins needed for the design within figure 7F.

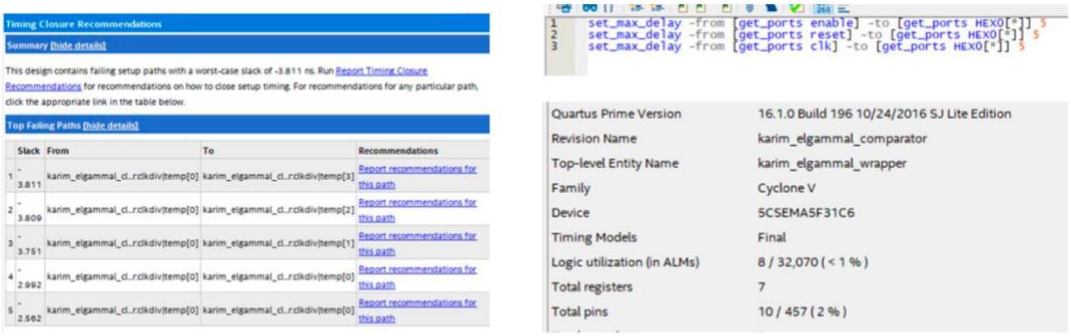


Figure 7F – Timing Analysis of Wrapper Entity

Sequence Detector

For this part of the lab, we have been asked to implement a sequence detector that detects two patterns from a single input. The first sequence we are detecting is 1011 and the second is 0100. The detector has two outputs which correspond to the first sequence and the second sequence respectively. To realize this component, we have created two sub-components that resemble a Moore-Type FSM, one to detect 1011 and the other to detect 0100. We define a type of state within our design file to navigate between the different states that our FSM will have. As seen in figure

8, we use the type of state in conjunction with case statements that allow us to evaluate the state that we are in and which steps we can take from the state being examined. We also utilize the if-else statements VHDL supplies in order to conditionally execute the code based on the reset and enable signal. We then use these sub-components to create our sequence detector. This is done by instantiating one of each sub-FSMs and connecting their inputs to the sequence input and each of their respective outputs to out_1 and out_2 respectively.

```
architecture beh of karim_elgammal_miniFSM is
type state is (A0, A1, A2, A3, A4);

Signal currentState : state;

begin
process (clock, reset)

begin
if (reset = '0') then
    currentState <= A0;

    elsif falling_edge(clock) then

if enable = '1' then

    case currentState is

        when A0 =>
            if P = '1' then
                currentState <= A1;
                elsif P = '0' then
                currentState <= A0;
                end if;

        when A1 =>
            if P = '1' then
                currentState <= A1;
                elsif P = '0' then
                currentState <= A2;
                end if;
```

```
architecture beh of karim_elgammal_FSM is

component karim_elgammal_miniFSM --for 1011
Port (clock :  in std_logic;
      enable : in std_logic;
      P :      in std_logic;
      reset :  in std_logic;
      O :      out std_logic);
end component;

component karim_elgammal_miniFSM2 -- for 0010
Port (clock :  in std_logic;
      enable : in std_logic;
      P :      in std_logic;
      reset :  in std_logic;
      O :      out std_logic);
end component;

begin

miniFSM1 : karim_elgammal_miniFSM port map( clock => clk,
                                            enable => enable,
                                            P => seq,
                                            reset => reset,
                                            O => out_1);

miniFSM2 : karim_elgammal_miniFSM2 port map ( clock => clk,
                                              enable => enable,
                                              P => seq,
                                              reset => reset,
                                              O => out_2);


end beh;
```

Figure 8 – Implementation of FSM/Sequence Detector

Results:

After writing a testbench that asserts the sequences described above, we ran the waveform simulation in order to examine the behaviour of our sequence detector. As seen from figure 9 and 10, we are successfully determining when a pattern occurs; and even when these respective patterns overlap. We further performed tests to all the input signal combinations to ensure that our enable and reset functionalities are operating. Further, we chose to implement the FSM based on the falling edge of the clock, this is to ensure that once a pattern has been fed, the indicator for that respective pattern is only asserted at the next clock cycle as described in the lab instructions. Further, we can examine that out_1 detects 1011, and out_2 detects 0010.

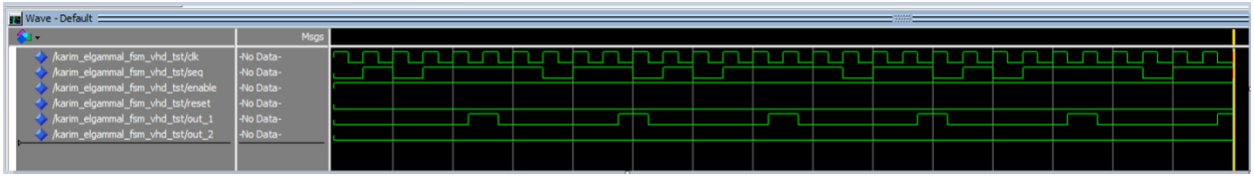


Figure 9 – Waveform for FSM Showcasing Detection of 1011

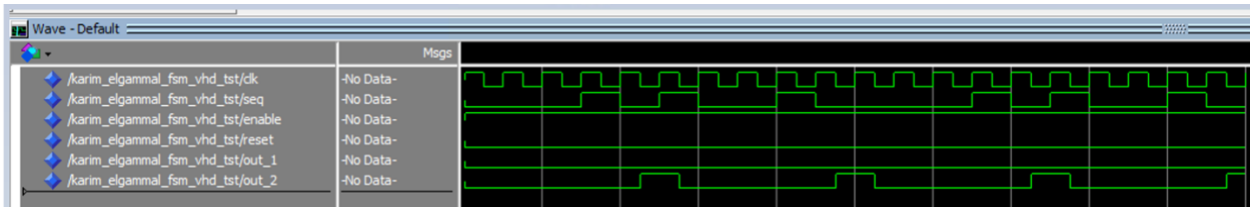


Figure 10 – Waveform of FSM Showcasing Detection of 0010

Question 4: Why is it better to use two FSMs, rather than one, in the implementation of the sequence detector from Section 9?

Answer: It is optimal to use two FSMs to implement our sequence detector as it allows us to interpret and anticipate less states than necessary. Further, if we only used one FSM, we might not be able to detect the occurrence of both patterns at the same time. We generate an FSM that detects 1011 and one that detects 0010 and they do this concurrently, at the same time.

Sequence Counter

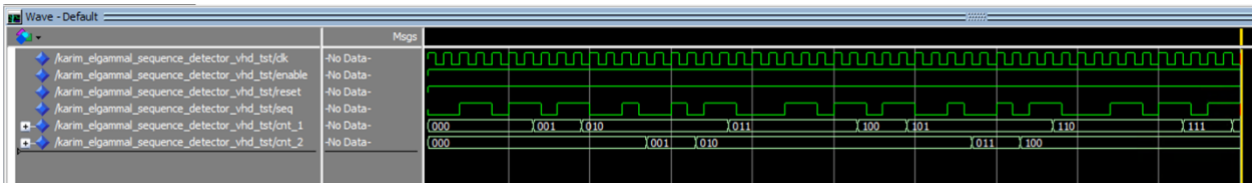
By utilizing the previously implemented circuits of our sequence detector and the 3 bit up counter; we are able to realize a circuit which detects the sequences and additionally counts the occurrence of said sequences. This is realized by instantiating one sequence detector and two 3 bit up counters, one for each sequence. We also utilize temporary signals which connect the detector's output to the 3-bit counter's clock signal. This asserts a count every time one of the sequences are recognized. We then feed the output of the encasing circuit as the output of the count. This logic can be examined in figure 11.

```
component karim_elgammal_FSM
    port( seq : in std_logic;
          enable: in std_logic;
          reset : in std_logic;
          clk : in std_logic;
          out_1 : out std_logic;
          out_2 : out std_logic);
end component;

component karim_elgammal_counter
Port (enable : in std_logic;
      reset : in std_logic;
      clk : in std_logic;
      count : out std_logic_vector (2 downto 0));
end component;

signal counterdriver1 : std_logic;
signal counterdriver2 : std_logic;

begin

seq_detect : karim_elgammal_FSM port map( seq => seq,
                                          enable => enable,
                                          reset => reset,
                                          clk => clk,
                                          out_1 => counterdriver1,
                                          out_2 => counterdriver2);

counter1 : karim_elgammal_counter port map( enable => enable,
                                            reset => reset,
                                            clk => counterdriver1,
                                            count =>  cnt_1);

counter2 : karim_elgammal_counter port map( enable => enable,
                                            reset => reset,
                                            clk => counterdriver2,
                                            count => cnt_2);
end beh;
```

Figure 11 – Implementation of Sequence Counter

Results:

We then performed waveform simulation of the sequence counter. We have designed our testbench to match that of the desired behaviour mentioned within the lab instructions. This is to assess the behaviour of our components. As seen below in figure 12, we match the desired behaviour by counting the instances of each of the patterns through cnt_1 and cnt_2. Further we can examine that the count maintains its state or value if the same pattern has not been detected again.



clock:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
seq:	0	0	1	1	0	1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
out_1:	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	2	2	2	2	3	3
out_2:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2

Figure 12 – Waveform Analysis of Sequence Counter & Desired Behaviour

Final Wrapper

Finally, we are asked to implement a circuit that can count a sequence coming from a signal being asserted every 1 second. Then this count is presented on the HEX displays on the FPGA board. Further, we implement the reset and enable functionalities to enable/disable the function of our circuit and to reset the status of our counter. The unit that supplies the sequence was supplied to us as ROM (a random-access memory). We utilize this unit within our design to match the schematic in figure 13. And so, we use the previously designed circuits and instantiate them to match this schematic. However, through the implementation, we feed into our ROM a clock signal that matches to 1 second of real time; and so, we use the clock divider circuit to output that rate. We do this by setting our T to 50000000; this is to account for the 50MHz clock frequency of our device. By this design, we receive a bit of information from the ROM every one second of real time. We also feed the output of our clock divider into the clock input of our sequence counter; this is to synchronize the clock frequency of all the components within our design. We then instantiate two instants of the seven-segment decoder in order to display the 3 bit value we receive from the sequence counter on to the hex display. The functional part of our code is presented in figure 13.

```
begin
clk_div : karim_elgammal_clock_divider port map( enable => enable,
                                                 reset => reset,
                                                 clk => clk,
                                                 en_out => clkdiv_to_ROM);

seqgen : ROM port map(clk => clkdiv_to_ROM,
                      reset => reset,
                      data => rom_to_sequence);

seqcounter : karim_elgammal_sequence_detector port map( seq => rom_to_sequence,
                                                        enable => enable,
                                                        reset => reset,
                                                        clk => clkdiv_to_ROM,
                                                        cnt_1 => cnt1,
                                                        cnt_2 => cnt2);

hexdis0 : seven_segment_decoder port map (code => cnt1,
                                          segments_out => HEX0);

hexdis5 : seven_segment_decoder port map (code => cnt2,
                                          segments_out => HEX5);

end struct;
```

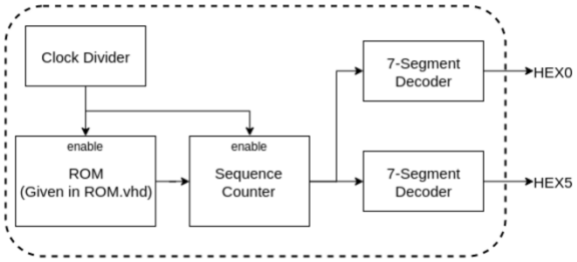


Figure 13 – Wrapper Logic and Functional Code

Results:

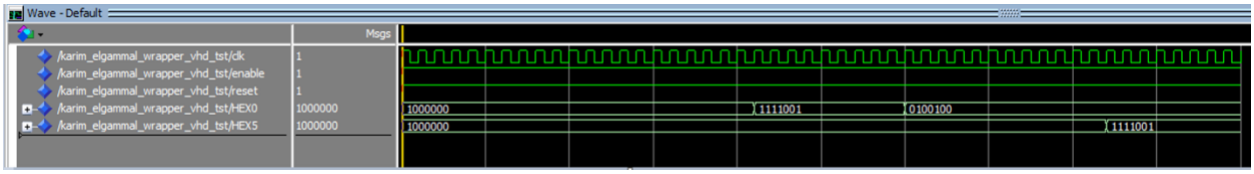


Figure 14 – Waveform of Wrapper Simulation

We can observe from our results above that our wrapper circuit behaves how we would expect it to based on our desired behaviour section of the lab instructions.

Logic Utilization, Total Pin and Critical Path:

Flow Status	Successful - Thu Dec 01 22:39:16 2022
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Standard Edition
Revision Name	karim_elgammal_comparator
Top-level Entity Name	karim_elgammal_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	51 / 32,070 (< 1 %)
Total registers	52
Total pins	17 / 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 %)

	Slack	From	To	Recommendations
1	-7.025	karim_elgammal_cl...:clk_div|temp[9]	karim_elgammal_cl...:clk_div|temp[8]	Report recommendations for this path
2	-7.021	karim_elgammal_cl...:clk_div|temp[9]	karim_elgammal_cl...:clk_div|temp[9]	Report recommendations for this path
3	-7.021	karim_elgammal_cl...:clk_div|temp[9]	karim_elgammal_cl...:clk_div|temp[7]	Report recommendations for this path
4	-6.922	karim_elgammal_cl...clk_div|temp[12]	karim_elgammal_cl...:clk_div|temp[8]	Report recommendations for this path
5	-6.918	karim_elgammal_cl...clk_div|temp[12]	karim_elgammal_cl...:clk_div|temp[9]	Report recommendations for this path

	Clock	Setup	Hold	Recovery
1	Worst-case Slack	-7.025	0.000	N/A
1	clk	-7.025	0.000	N/A
2	karim_elgammal_clock_divider:clk_div|temp[0]	-4.432	0.238	N/A
3	karim_elgammal_sequence_detector:seqcounte...elgammal_minFSM2:minFSM2|currentState.A4	-1.153	0.258	N/A
4	karim_elgammal_sequence_detector:seqcounte..._elgammal_minFSM:minFSM1|currentState.A4	-1.129	0.277	N/A
2	Design-wide TNS	-169.397	0.0	0.0
1	clk	-114.625	0.000	N/A
2	karim_elgammal_clock_divider:clk_div|temp[0]	-49.886	0.000	N/A
3	karim_elgammal_sequence_detector:seqcounte...elgammal_minFSM2:minFSM2|currentState.A4	-2.839	0.000	N/A
4	karim_elgammal_sequence_detector:seqcounte..._elgammal_minFSM:minFSM1|currentState.A4	-2.935	0.000	N/A

Figure 15 – Logic Analysis for Wrapper Circuit And Timing Analysis

The critical path analysis for this entity, follows the same method of finding the critical path of the comparator. We placed timing constraints on each of the inputs/outputs of the component in an SDC file. Then upon compiling the timing analysis, we received the violating paths that occur within the design. Our results seem to again, indicate a path that lies within the clock divider. The reiteration that takes places during the processes of the clock divider seem to be violating paths

even with modifying the timing constraints from a range of 5ns to 100ns. We can gather that the critical path then, is from the clk signal, and ends at the HEX0 output, whichever index (as it is works like a LUT with the seven segment decoder). The biggest difference, or slack is then -7.025, which allows us to believe that the delay of the critical path, is 5 + 7.025 = 12.025. We can also observe the total UI utilization and total pins needed for the design within figure 15.

Part 6: Conclusions

Throughout this investigation, we have been able to successfully implement storage elements and finally, a counter. Most of our designs, other than the smaller elements utilize structural implementation of code. This allowed for an efficient design process that gave way for easy debugging. Further, we have incorporated the requirements of the design successfully. However, the critical path analysis seems to be inconclusive for the 3-bit up counter; this is due to the discrepancies within the clock environment. As we have set it up for a 10hz environment, the analysis was slightly complex and required more time for complete evaluation. Finally, we have been able to demonstrate our working circuit on to our FPGA board. A sample demonstration is supplied in video format along with this report.