ECSE 222 Fall 2022

VDHL Assignment 2 Report: Design and Simulation of Digital Circuits

Part 1 – Participants

Karim Elgammal 260920556

Lutming Wang 261006348

Part 2 – Introduction

Throughout this lab, we implemented/designed a simple 4-bit comparator circuit through a schematic file, then used a Quartus tool to convert the schematic into a VHDL design file, expressing the different components and functionality of said design. Furthermore, we present the testing that was done using a testbench file in two ways, an introductory test, and then an exhaustive test. With a further understanding of digital circuits and now VDHL files/language, we also designed and implemented a 2 to 1 MUX, first in a structural form, and then in the behavioral form. Testbench files were written for both 'styles' and the digital circuit was simulated using Altera-Modelsim to arrive at the waveform of the respective signals we have used throughout the design. Moreover, the implementation of the 4-bit comparator was mostly supplied by the instructions of the given lab, while the implementation of the 2 to 1 MUX, presents us with a more independent implementation.

Part 3 – Methodology

Implementation of a 4-Bit Comparator:

We started off with using the schematic design maker within Quartus to create the logic gate representation of a 4-bit comparator. Essentially a 4-bit comparator has the function of comparing two binary numbers to each other, however it is limited, as assumed with the name, to 4 bit binary numbers. Figure 1 presents the schematic that was designed using the Quartus gate tools. This is the logic circuit for our comparator. And through Quartus we were able to perform an automatic analysis of the circuit to arrive at the Boolean expression/s to implement the circuit. Figure 2 shows the automatically derived structural architecture of the comparator that was designed.

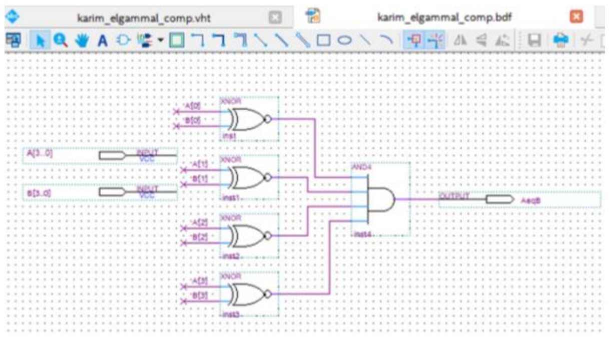
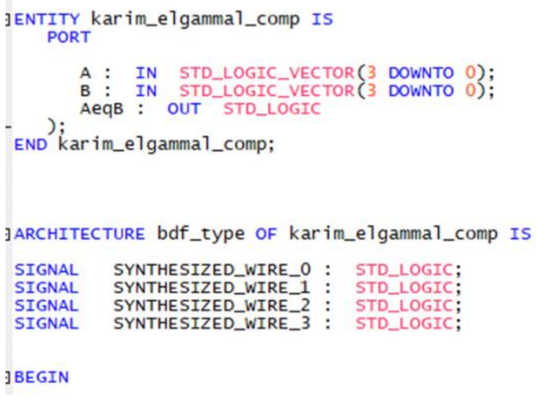


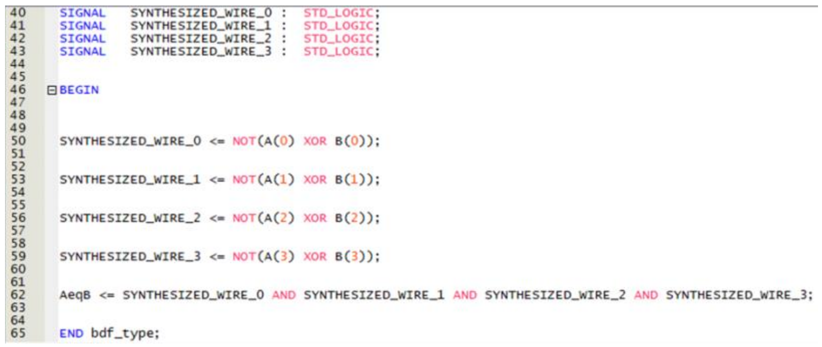
Figure 1 - Schematic Design of 4-Bit Comparator

The comparator features 8 inputs for the 8-bit total input of the device (4 for each binary number). The 8 inputs leads to 4 XNOR gates (complementary exclusive OR gate), which then lead to one 4-input AND gate. The structural architecture that was generated encompasses the Boolean logic for the comparator and there are two levels to the schematic design, hence the declaration and use of the 'synthesized wire' signal that allows the unit to first evaluate the XNOR gates, and then the AND gate. Then, these signals are combined in the output AeqB, to give us the final output. Figure 2.2 shows the Boolean logic that is described using the synthesized wire signals and the combined output signal.



```
ENTITY karim_elgammal_comp IS
    PORT
    (
        A :  IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
        B :  IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
        AeqB :  OUT  STD_LOGIC
    );
END karim_elgammal_comp;


ARCHITECTURE bdf_type OF karim_elgammal_comp IS

SIGNAL   SYNTHESIZED_WIRE_0 :  STD_LOGIC;
SIGNAL   SYNTHESIZED_WIRE_1 :  STD_LOGIC;
SIGNAL   SYNTHESIZED_WIRE_2 :  STD_LOGIC;
SIGNAL   SYNTHESIZED_WIRE_3 :  STD_LOGIC;


BEGIN
```

Figure 2.1 Definition of Comparator Inputs and Outputs



```
SIGNAL   SYNTHESIZED_WIRE_0 :  STD_LOGIC;
SIGNAL   SYNTHESIZED_WIRE_1 :  STD_LOGIC;
SIGNAL   SYNTHESIZED_WIRE_2 :  STD_LOGIC;
SIGNAL   SYNTHESIZED_WIRE_3 :  STD_LOGIC;


BEGIN



SYNTHESIZED_WIRE_0 <= NOT(A(0) XOR B(0));


SYNTHESIZED_WIRE_1 <= NOT(A(1) XOR B(1));


SYNTHESIZED_WIRE_2 <= NOT(A(2) XOR B(2));


SYNTHESIZED_WIRE_3 <= NOT(A(3) XOR B(3));


AeqB <= SYNTHESIZED_WIRE_0 AND SYNTHESIZED_WIRE_1 AND SYNTHESIZED_WIRE_2 AND SYNTHESIZED_WIRE_3;


END bdf_type;
```

Figure 2.2 Structural Architecture of the Comparator

This implementation of the comparator was tested using two approaches; firstly an introductory testing script, and then an exhaustive testing script. The introductory testing script simply inputs a small range of binary numbers into the ‘logic_vector’ type input, A and B. The exhaustive test script however, tests the full range of possible binary numbers that are composed of 4-bits for both inputs, that is 256 possible arrangements of inputs. The output is then displayed using the Altera-Modelsim, simulation tool (fig). From the figure below, we can see that the output of our testbench simulation provides us with what we expect, an output that compares the two binary numbers, and outputs the value 1, only when the two numbers are of the same magnitude.

As seen in figure 3.1, we can see the introductory testing supplies us with the outcomes that we had expected, i.e., an output of 1 when the magnitude of the two binary numbers is the same. Furthermore, figure 3.2 presents the exhaustive testing, and a clear pattern is emerged using this exhaustive testing, where we see the shifts in the green line correspond to when the values of the two binary numbers are the same. Figure 3.3 examines the simulation plot of the exhaustive testing in more detail.

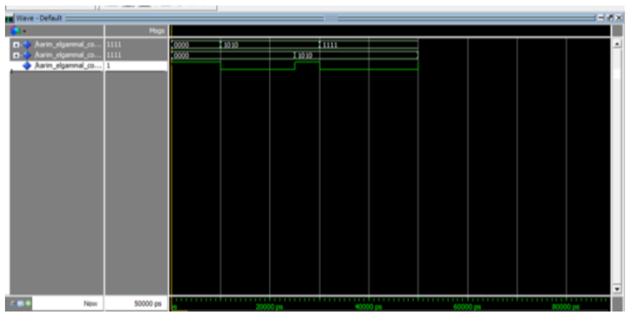


Figure 3.1 Introductory Testing Outcome

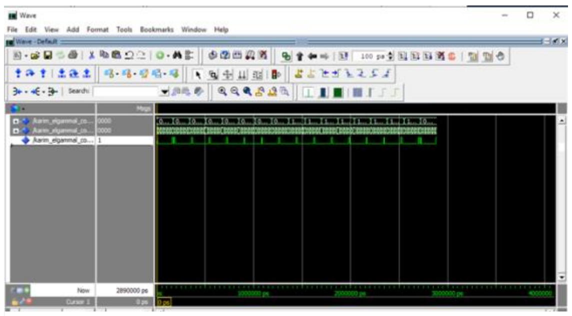


Figure 3.2 Exhaustive Testing Outcome

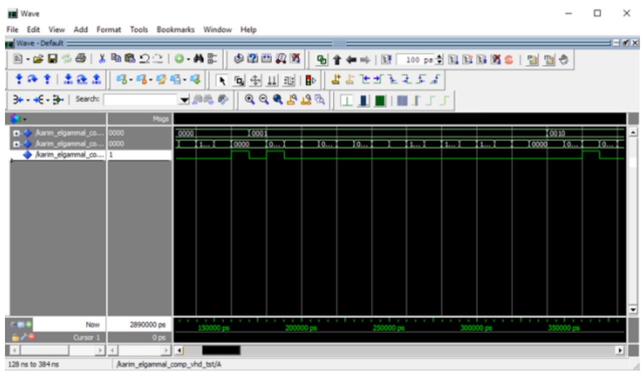
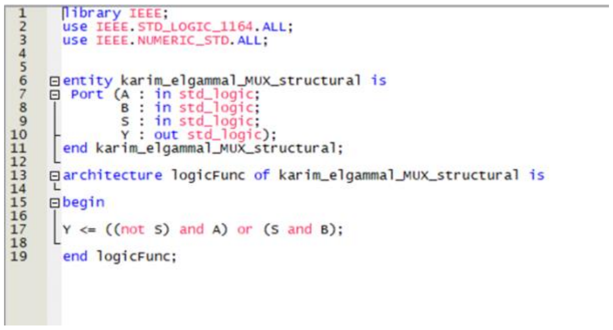


Figure 3.3 Exhaustive Testing Outcome 2

Implementation of a 2 to 1 Multiplexer:

A Multiplexer is used to combine multiple signals, into one output channel; within this specific example the 2 to 1 multiplexer combines two signals into one output channel using a selector signal that is also considered an input signal within our component design. Firstly, we started with a declaration of the entity and defining the ports that are associated with the input signals and the single output signal. Throughout this lab, we have been asked to implement the multiplexer through two methods, first a structural representation, and then a behavioral representation.

Structural Implementation of the 2 to 1 Multiplexer:

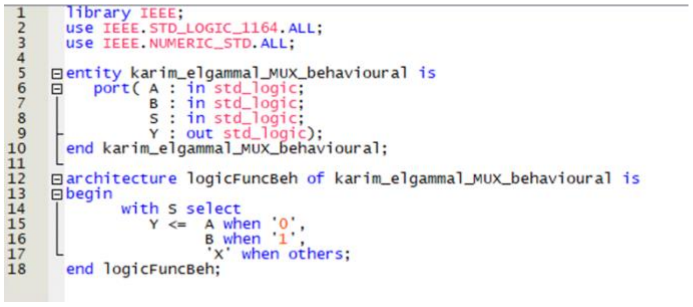


```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5
6   entity karim_elgammal_MUX_structural is
7    Port (A : in std_logic;
8          B : in std_logic;
9          S : in std_logic;
10         Y : out std_logic);
11  end karim_elgammal_MUX_structural;
12
13  architecture logicFunc of karim_elgammal_MUX_structural is
14
15  begin
16
17  Y <= ((not S) and A) or (S and B);
18
19  end logicFunc;
```

Figure 4.1 Structural Implementation

As seen from figure 4.1, the structural implementation includes the entity declaration of our component, where we define the type of signals (A, B, S and Y) and defining weather they are inputs or outputs. This shapes the ‘blackbox’ of karim_elgammal_MUX_structural. Then, the architecture description in a structural method which essentially tells the program, what the functionality of this multiplexer is. Because this is the structural representation; a Boolean expression is the only thing that is required to represent the function of the karim_elgammal_MUX_structural. The Boolean logic is derived using the fact that a 2 to 1 multiplexer only emits the signal A when the selector signal S, is equal to 0, and emits signal B otherwise. From line 17, we can see that the Boolean logic is combined into the output Y, using the signal assignment <=.

Behavioral Implementation of the 2 to 1 Multiplexer:



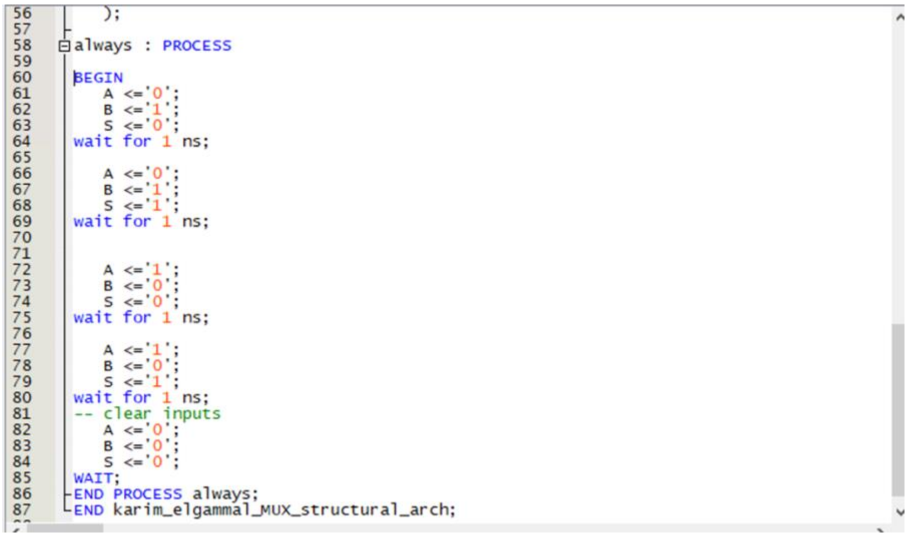
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity karim_elgammal_MUX_behavioural is
    port( A : in std_logic;
          B : in std_logic;
          S : in std_logic;
          Y : out std_logic);
end karim_elgammal_MUX_behavioural;

architecture logicFuncBeh of karim_elgammal_MUX_behavioural is
begin
        with S select
            Y <=  A when '0',
                  B when '1',
                  'X' when others;
end logicFuncBeh;
```

Figure 4.2 Behavioral Implementation

The behavioral implementation of the multiplexer follows the same declaration of the entity, with the same respect to the ports that are used in the structural implementation. The difference within this implementation is the architecture of the entity; while we used a Boolean expression to describe the functionality of the multiplexer, in this behavioral implementation, we described the outputs related to each kind of input that could be received. This is done using a select statement which describes the possible set of entries from S, {0,1}. It also considers the possibility of receiving an 'other' signal, or an undefined signal. In each case of an input from S, the select statement assigns the output with an input as described above in the explanation of a 2 to 1 multiplexer. So, when S is 0, A is the output, while if S is 1, B is the output. Additionally, the behavioral implementation outputs X when S is ant other value outside of the domain {0,1}, where X is an unknown value within the std_logic type.

Part 4 & 5 – Results

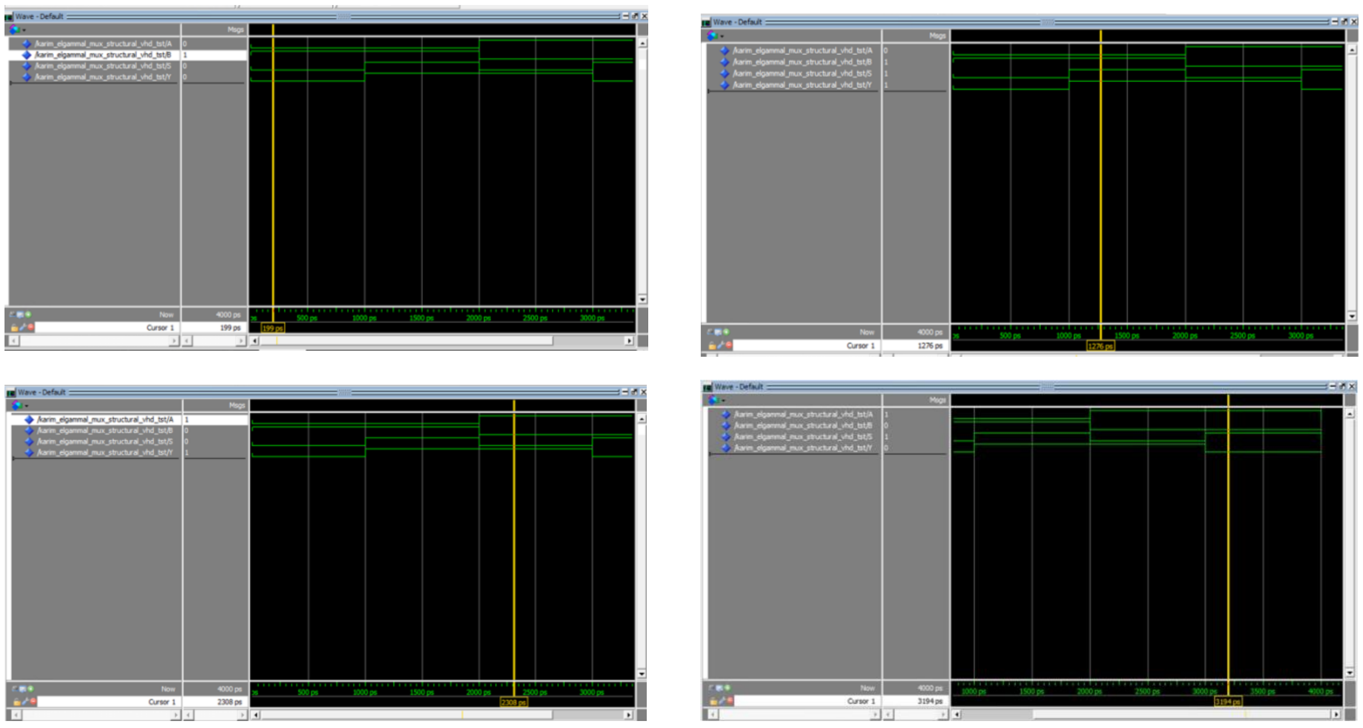
These implementations were evaluated using a testbench file that iterates through the possible values that can be placed as input. This mirrors the introductory testing that was used with the 4-bit comparator above. As seen from figure 4.3, the testbench code essentially inputs all the possible inputs of the device at hand. Both the structural and behavioral implementation are tested using the same method of a variety of inputs; the only difference between the two scripts is the entity that it is pointing to, for the simulation process.



```
    );

always : PROCESS

BEGIN
    A <='0';
    B <='1';
    S <='0';
wait for 1 ns;

    A <='0';
    B <='1';
    S <='1';
wait for 1 ns;


    A <='1';
    B <='0';
    S <='0';
wait for 1 ns;

    A <='1';
    B <='0';
    S <='1';
wait for 1 ns;
-- clear inputs
    A <='0';
    B <='0';
    S <='0';
WAIT;
END PROCESS always;
END karim_elgammal_MUX_structural_arch;
```

Figure 4.3 Structural Implementation Testbench Script

From the simulation plots, we can observe the functionality that we described within the architecture of the entity. Below are the simulation plots for the structural implementation of the multiplexer. Using the placement of the yellow cursor and the table of values on the left of the plot we can examine the results received by this implementation. And so, we can see clearly that the output is strictly dependent on the input of S, where the output is always A when S = 0, and B whenever S = 1. The results are nearly identical when the behavioral implementation is simulated which is seen in figure 6.



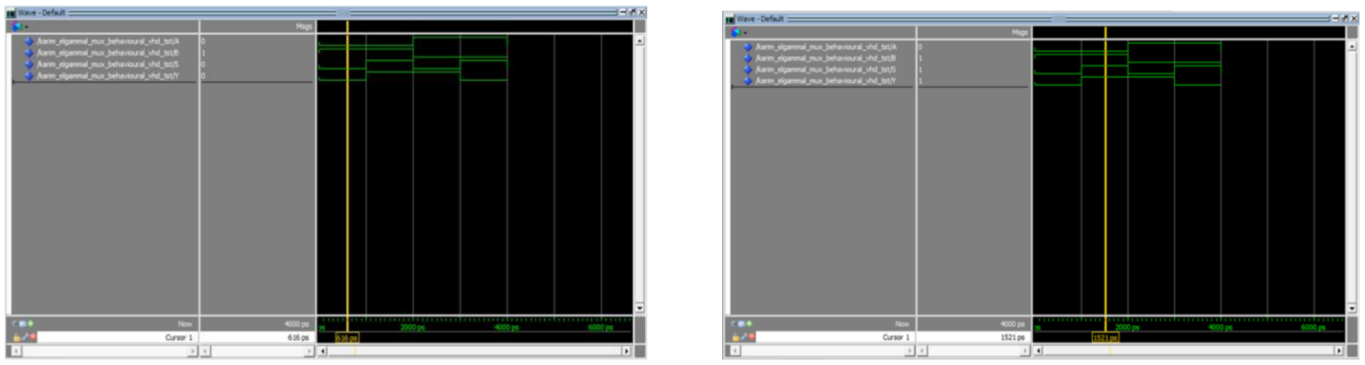
```
Wave - Default                                   Msgs
/karim_elgammal_mux_structural_vhd_tst/A          0
/karim_elgammal_mux_structural_vhd_tst/B          1
/karim_elgammal_mux_structural_vhd_tst/S          0
/karim_elgammal_mux_structural_vhd_tst/Y          0
Now      4000 ps
Cursor 1  199 ps
```

```
Wave - Default                                   Msgs
/karim_elgammal_mux_structural_vhd_tst/A          0
/karim_elgammal_mux_structural_vhd_tst/B          1
/karim_elgammal_mux_structural_vhd_tst/S          1
/karim_elgammal_mux_structural_vhd_tst/Y          1
Now      4000 ps
Cursor 1  1276 ps
```

```
Wave - Default                                   Msgs
/karim_elgammal_mux_structural_vhd_tst/A          1
/karim_elgammal_mux_structural_vhd_tst/B          0
/karim_elgammal_mux_structural_vhd_tst/S          0
/karim_elgammal_mux_structural_vhd_tst/Y          1
Now      4000 ps
Cursor 1  2308 ps
```

```
Wave - Default                                   Msgs
/karim_elgammal_mux_structural_vhd_tst/A          1
/karim_elgammal_mux_structural_vhd_tst/B          0
/karim_elgammal_mux_structural_vhd_tst/S          1
/karim_elgammal_mux_structural_vhd_tst/Y          0
Now      4000 ps
Cursor 1  3194 ps
```

Figure 5 Structural Implementation Simulation Plots



```
Wave - Default                                   Msgs
/karim_elgammal_mux_behavioural_vhd_tst/A         0
/karim_elgammal_mux_behavioural_vhd_tst/B         1
/karim_elgammal_mux_behavioural_vhd_tst/S         0
/karim_elgammal_mux_behavioural_vhd_tst/Y         0
Now      4000 ps
Cursor 1  616 ps
```

```
Wave - Default                                   Msgs
/karim_elgammal_mux_behavioural_vhd_tst/A         0
/karim_elgammal_mux_behavioural_vhd_tst/B         1
/karim_elgammal_mux_behavioural_vhd_tst/S         1
/karim_elgammal_mux_behavioural_vhd_tst/Y         1
Now      4000 ps
Cursor 1  1521 ps
```

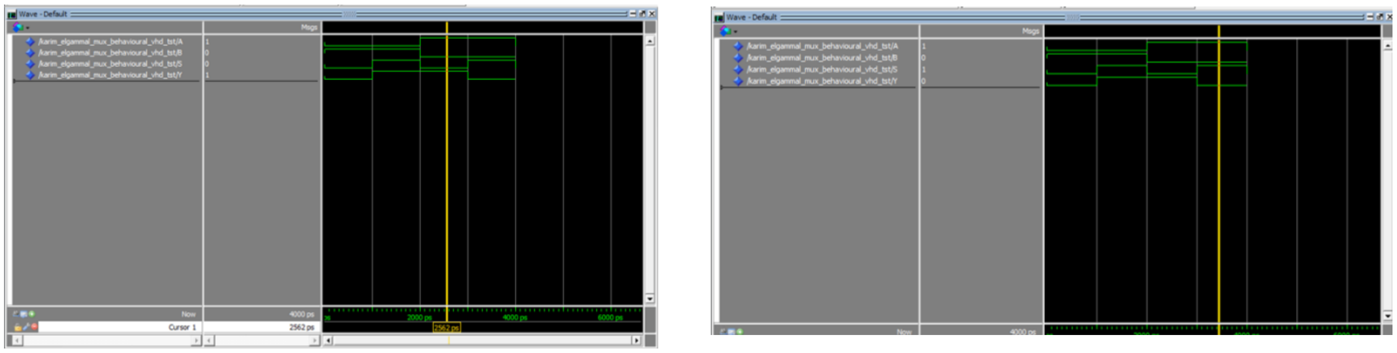

Wave - Default
Msgs
/karim_elgammal_mux_behavioural_vhd_tst/A 1
/karim_elgammal_mux_behavioural_vhd_tst/B 0
/karim_elgammal_mux_behavioural_vhd_tst/S 0
/karim_elgammal_mux_behavioural_vhd_tst/Y 1
Now 4000 ps
Cursor 1 2562 ps
2000 ps
4000 ps
6000 ps
Wave - Default
Msgs
/karim_elgammal_mux_behavioural_vhd_tst/A 1
/karim_elgammal_mux_behavioural_vhd_tst/B 0
/karim_elgammal_mux_behavioural_vhd_tst/S 1
/karim_elgammal_mux_behavioural_vhd_tst/Y 0
Now 4000 ps

Figure 6 Behavioral Implementation Simulation Plots

Part 5 – Conclusions

From the results above, we can conclude that first, the exhaustive testing should be sought after when implementing such entities, this is especially because more complex entities will involve more complex inputs that will require evaluation of signals that are 'continuous'. The exhaustive testing uses a loop to iterate through the values of each bit of the binary number, and while this method of testing, was not used to validate the 2 to1 Multiplexer, the introductory test deemed to be acceptable especially since each input was of a magnitude of 1 bit. Overall, throughout this lab, we were able to explore the implementations of both a MUX and a comparator, both units are used throughout electronic devices in order to drive logic circuits. This physical implementation can be summarized using the table below, which gives us an insight on how many pins are required for each device, and for both types of implementations. As seen below the total number of pins from both implementations are equal; and this might lead us to believe that they carry the same efficiency/cost when physically implemented, however our limitations of a pretty minimal design narrow our conclusion to only include the MUX in our similarity conclusion.

	AeqB	2-to-1 Multiplexer	
	Schematic	Structural	Behavioral
Logic Utilization (in ALMs)	2/32070	1 / 32,070	1 / 32,070
Total Pins	9	4	4