ECSE 222: Lab 1 Report

Group #48

Dafne Culha (260785524)
Cheng Lin (260787697)

Due: Mar 11, 2019

Table of Contents

I Introduction

This laboratory experiment introduced the basics of Intel's Quartus Prime and ModelSim softwares through the programming of a binary-to-7-segment LED decoder and a 5-bit adder. In particular, 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 adds two 5-bit binary numbers and displays the inputs and sum on six 7-segment LED displays.

II 7-segment display overview

Each 7-segment decoder on the Altera board resembles the one below (Figure 1). The decoders have seven segments, labelled from 0 to 6, that can be turned on in a special pattern to display a number or letter (Figure 2). The 7-segment displays on the Altera board are "active low," meaning their segments turn on when the input bit is 0. The entire display is controlled by a sequence of 7-bits, where each bit corresponds to a different segment.

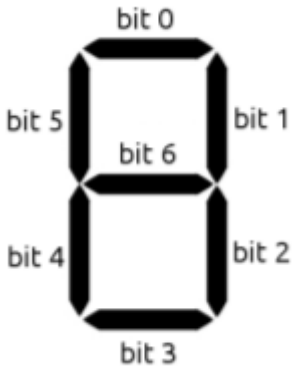


Fig. 1: 7-segment display

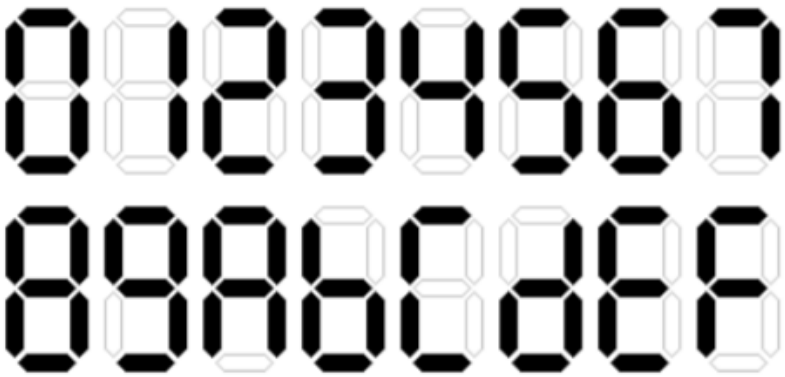


Fig. 2: 7-segment output configurations for hexadecimal outputs

III Binary-to-7-Segment LED decoder

The first required circuit is a binary-to-7-segment LED circuit. The VHDL program for this circuit needs to convert a 4-bit binary input into a sequence of seven bits. Furthermore, these seven bits need to result in the corresponding hexadecimal digit on the 7-segment display.

III.I Circuit description

To determine each 4-bit input's associated 7-bit output, we used Figure 1 and binary-to-decimal number conversion to construct a truth table (Figure 3).

4-bit binary input				7-bit encoded output for 7-seg display (active low)						
w_3	w_2	w_1	w_0	seg_6	seg_5	seg_4	seg_3	seg_2	seg_1	seg_0
0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	0	0	1
0	0	1	0	0	1	0	0	1	0	0
0	0	1	1	0	1	1	0	0	0	0
0	1	0	0	0	0	1	1	0	0	1
0	1	0	1	0	0	1	0	0	1	0
0	1	1	0	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	1	0	0	0
1	0	1	1	0	0	0	0	0	1	1
1	1	0	0	1	0	0	0	1	1	0
1	1	0	1	0	1	0	0	0	0	1
1	1	1	0	0	0	0	0	1	1	0
1	1	1	1	0	0	0	1	1	1	0

Fig. 3: truth table for converting 4-bit binary numbers to decimal digits on a 7-segment display

Next, in the VHDL software, we created the decoder entity. It accepts a 4-bit logic vector (for the 4-bit input) and returns a 7-bit logic vector (for the decoded sequence of 7-bits).

```
12  --create 7 segment decoder entity
13  entity g48_7_segment_decoder is
14      --input is a 4-bit binary number
15      --output is a 7-bit encoded value for the seven segment display
16      Port( code: in std_logic_vector(3 downto 0);
17            segments: out std_logic_vector(6 downto 0));
18  end g48_7_segment_decoder;
19
```

Then, within the architecture of the decoder, we specified the corresponding 7-bit vector for each 4-bit input. We used selected signal assignment to set the output vector because using boolean/logic statements would be more verbose and complicated. This can be seen from the truth table (Figure 3), as there is no observable pattern for any of the segment's values.

```
20  --create architecture for decoder
21  architecture behaviour of g48_7_segment_decoder is
22  begin
23
24      --use single selected signal assignment to set segments vector
25      --using boolean statements to set vectors is too complicated
26
27      segments <= "1000000" when code = "0000" else
28                  "1111001" when code = "0001" else
29                  "0100100" when code = "0010" else
30                  "0110000" when code = "0011" else
31                  "0011001" when code = "0100" else
32                  "0010010" when code = "0101" else
33                  "0000010" when code = "0110" else
34                  "1111000" when code = "0111" else
35                  "0000000" when code = "1000" else
36                  "0010000" when code = "1001" else
37                  "0001000" when code = "1010" else
38                  "0000011" when code = "1011" else
39                  "1000110" when code = "1100" else
40                  "0100001" when code = "1101" else
41                  "0000110" when code = "1110" else
42                  "0001110" when code = "1111";
43
44  end behaviour;
```

III.II Circuit testing

We tested the decoder circuit using the ModelSim software, a program that simulates VHDL circuits and tracks the values of inputs and outputs over time. After compiling the decoder's VHDL code on Quartus Prime, we added the .vhd file to a new project in ModelSim. We then used Quartus Prime's Test Bench Template Writer to create the framework of the .vht file that simulates the decoder. We edited this .vht file to include a "generate_test" process, that looped through all 4-bit binary values and inputted them one at a time into the decoder.

```
58  generate_test : PROCESS
59  BEGIN
60      FOR i IN 0 to 15 LOOP
61          code <= std_logic_vector(to_unsigned(i,4));
62          WAIT FOR 10 ns;
63      END LOOP;
64      WAIT;
65  END PROCESS generate_test;
```

Next, in ModelSim, we ran the .vht simulation specified above. The output can be seen in Figure 4 and 5.

Fig. 4: Simulation plot of decoder circuit (first entry is 4-bit input, second entry is 7-bit output)

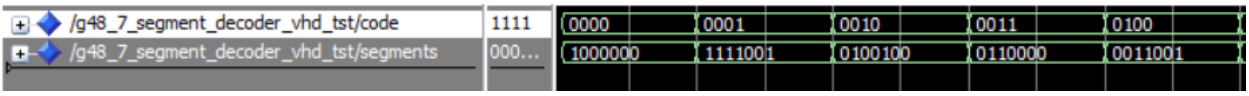


Fig. 5: Close-up of simulation plot

We then verified that all of the outputs (the second row) matched their respective inputs (the first row) by comparing the values with those in the truth table. Because all sixteen input combinations matched, we concluded that our decoder operates correctly.

Furthermore, our decoder is used multiple times in the 5-bit adder circuit that follows. The fact that our adder displays each input and the sum correctly on the Altera board's 7-segment displays also confirms that the decoder works correctly.

IV 5-bit adder

The second required circuit is a 5-bit binary adder. This circuit needs to accept two 5-bit binary numbers (range of 0_{10} to 31_{10}) and output the entire summation (inputs and final sum) in a hexadecimal format on 7-segment displays. Because the two inputs are 5-bit binary numbers, all possible inputs and all possible sums can be represented with two hexadecimal digits each. Therefore, a total of six 7-segment displays (two for one input, two for the other input, and two for the sum) are used.

IV.I Circuit description

We began by declaring an adder entity. It accepts two 5-bit binary numbers (the inputs), and returns three 13-bit vectors. The 13-bit vectors each represent the values of two 7-segment displays. Bit 0 to 6 correspond to one 7-segment, while bit 7 to 13 correspond to another. We have three 13-bit vectors to store the outputs for input A, input B, and the sum of A and B respectively.

```
16  --declare adder entity
17  entity g48_adder is
18      --input is two 5-bit binary numbers
19      --output is three 14-bit vectors corresponding to the 14 inputs for two 7 seg displays each
20      Port(A, B: in std_logic_vector(4 downto 0);
21           decoded_A: out std_logic_vector(13 downto 0);
22           decoded_B: out std_logic_vector(13 downto 0);
23           decoded_AplusB: out std_logic_vector(13 downto 0)
24           );
25  end g48_adder;
26
```

Next, we begin the adder's architecture declaration by importing the 7-segment decoder as a component and instantiating intermediate signals. The intermediate signal AplusB is used to hold the 6-bit sum of the 5-bit inputs, A and B.

```
27  --declare architecture for adder
28  architecture behaviour of g48_adder is
29
30      --import 7 seg decoder component
31      component g48_7_segment_decoder is
32          Port( code: in std_logic_vector(3 downto 0);
33                segments: out std_logic_vector(6 downto 0));
34      end component g48_7_segment_decoder;
35
36      --instantiate a signal to temporarily hold sum of A and B
37      signal AplusB: std_logic_vector(5 downto 0);
38
39      --instantiate 4-bit signals to hold the values for the second (/most significant) digit of each number
40
41      signal A1: std_logic_vector(3 downto 0); --values for most significant digit of A
42      signal B1: std_logic_vector(3 downto 0); --values for most significant digit of B
43      signal AplusB1: std_logic_vector(3 downto 0); --values for most significant digit of A plus B
44
```

Furthermore, the binary values of A, B, and AplusB can be easily converted into a hexadecimal representation by grouping every four consecutive bits and converting them separately (Figure 6).

$$18_{10} = \underbrace{0001}_{=\ 1}\ \underbrace{0010_2}_{2_{16}}$$
$$= 12_{16}$$

Fig. 6: Binary to hexadecimal conversion by grouping

However, A, B, and AplusB are not 8-bit vectors. While bit 0 to 3 of A, B, and AplusB can represent the right-most (or least significant) hexadecimal digit for each of these values, bit 4 (and bit 5) need to be concatenated with zeros to attain a full four bits that a decoder can accept. As a result, we require the intermediate signals of A1, B1, and AplusB1 to hold the concatenated 4-bits that represent each value's left-most (or most significant) hexadecimal digit.

Next, we declare the behaviour of the adder. As mentioned above, we begin by summing the 5-bit inputs of A and B together to obtain a 6-bit sum. The inputs are converted to 6-bits by adding a '0' to their left to accommodate for potential carry-out values.

```
45  begin
46      --add 5-bit inputs like normal 5-bit numbers
47      --concatenate '0' to left of each input to hold carry-out of sum
48      AplusB <= std_logic_vector(unsigned('0' & A) + unsigned('0' & B));
49
50      --assign values to the 4-bit signals for the second (/most significant) digit
51      --append zero's to the left so the values are 4 bits
52      A1 <= "000" & A(4);
53      B1 <= "000" & B(4);
54      AplusB1 <= "00" & AplusB(5 downto 4);
55
56      --instantiate decoder components to convert each 4-digit binary into 7-seg encoded input
57      --first (/least significant) 7-seg takes in the value dictated by bit 3 to 0
58      --second (/most significant) 7-seg takes in the value dictated by the extra signals instantiated above
59      --each decoder outputs to its corresponding "decoded_{}" vector
60
61      --most significant digit of A
62      decoderA1: g48_7_segment_decoder PORT MAP(code => A1,
63                          segments => decoded_A(13 downto 7));
64      --least significant digit of A
65      decoderA0: g48_7_segment_decoder PORT MAP(code => A(3 downto 0),
66                          segments => decoded_A(6 downto 0));
67
68      --most significant digit of B
69      decoderB1: g48_7_segment_decoder PORT MAP(code => B1,
70                          segments => decoded_B(13 downto 7));
71      --least significant digit of B
72      decoderB0: g48_7_segment_decoder PORT MAP(code => B(3 downto 0),
73                          segments => decoded_B(6 downto 0));
74
75      --most significant digit of sum
76      decoderAplusB1: g48_7_segment_decoder PORT MAP(code => AplusB1,
77                          segments => decoded_AplusB(13 downto 7));
78      --least significant digit of sum
79      decoderAplusB0: g48_7_segment_decoder PORT MAP(code => AplusB(3 downto 0),
80                          segments => decoded_AplusB(6 downto 0));
81
82  end behaviour;
```

Following the summation, we assign A1, B1, and AplusB1 their vector values by concatenating an appropriate number of zeros to the left of the bits that represent the most significant hexadecimal digits. We then instantiate six 7-segment decoder components—one for each 7-segment display—and assign the correct input and outputs variables to the port map.

Finally, we implemented the adder circuit on the Altera board. We used Quartus Prime's Pin Planner to associate outputs with pins on the board and then downloaded the circuit with the Quartus Programmer Tool. Once the circuit was downloaded, we could set our 4-bit binary inputs with the slide switches and see the hexadecimal outputs on the 7-segment displays.

IV.II Circuit testing

We tested the adder in the following two manners: through the test bench file in ModelSim and through the FPGA board.

Firstly, we created a test bench .vht file, then compiled and simulated it in ModelSim (Figure 7). Similar to the dencoder's test bench file, we created a "generate_test" process that assigns the inputs consecutive 5-bit binary values. Then, in ModelSim, we verified that the binary outputs of our values were all correct.

Secondly, we tested possible inputs with a programmed Altera board. Specifically, we tested both edge cases and generic cases. We made sure all the numbers were being added and displayed correctly on our board, to confirm that our adder circuit functioned correctly.

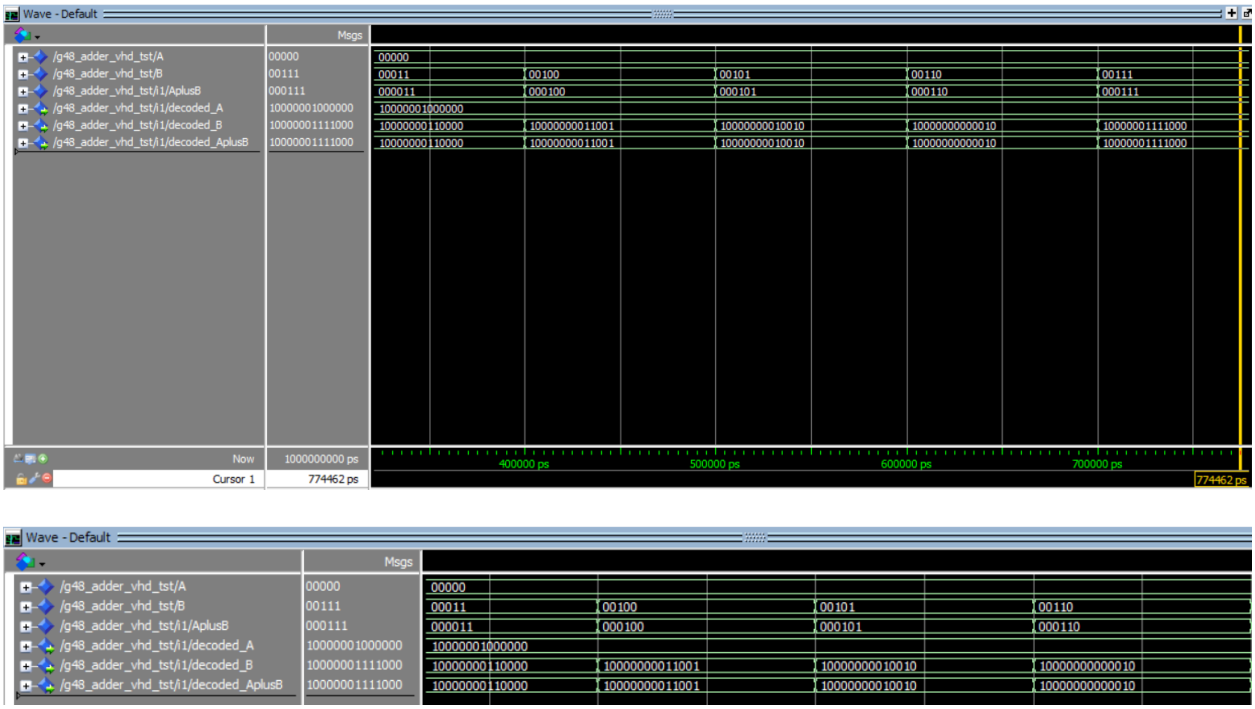
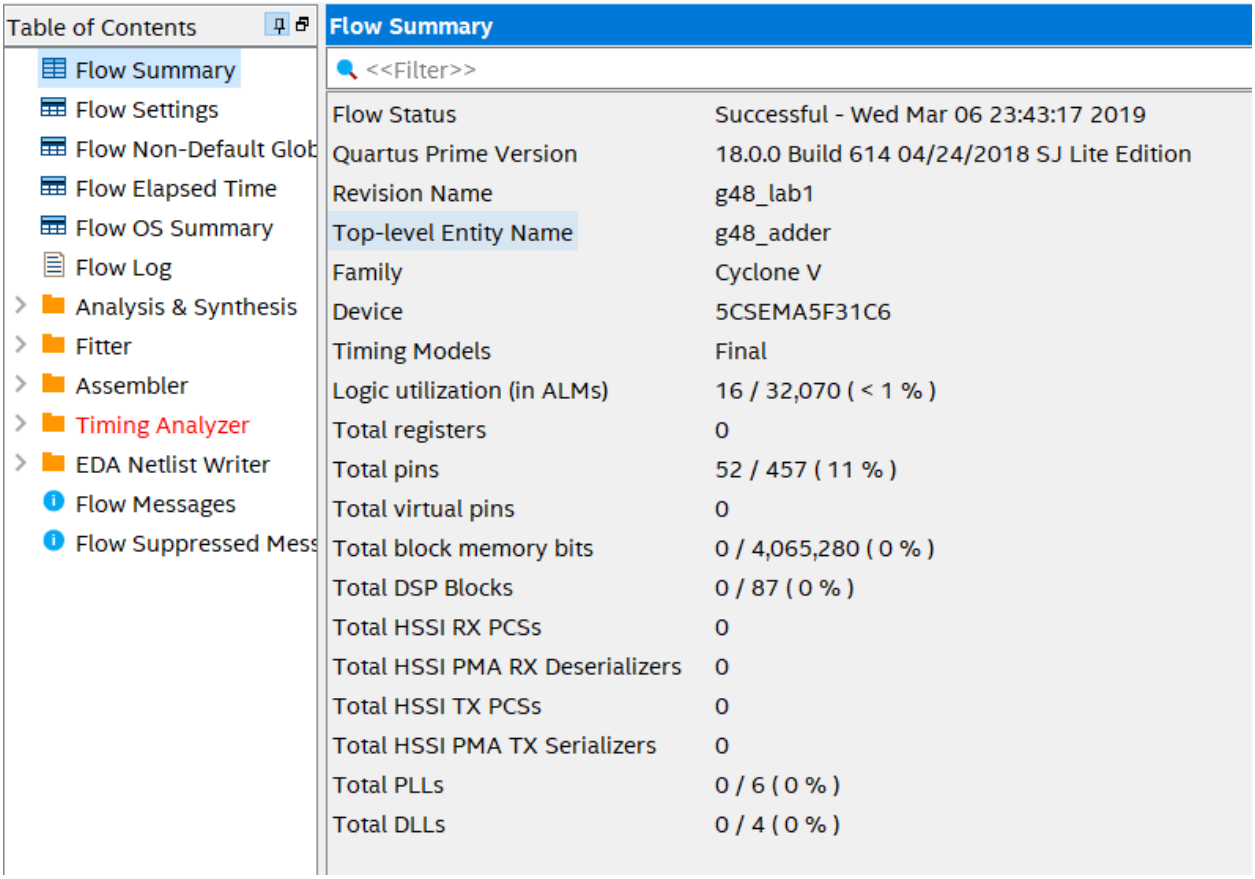


Fig. 7: ModelSim simulation plots of 5-bit adder

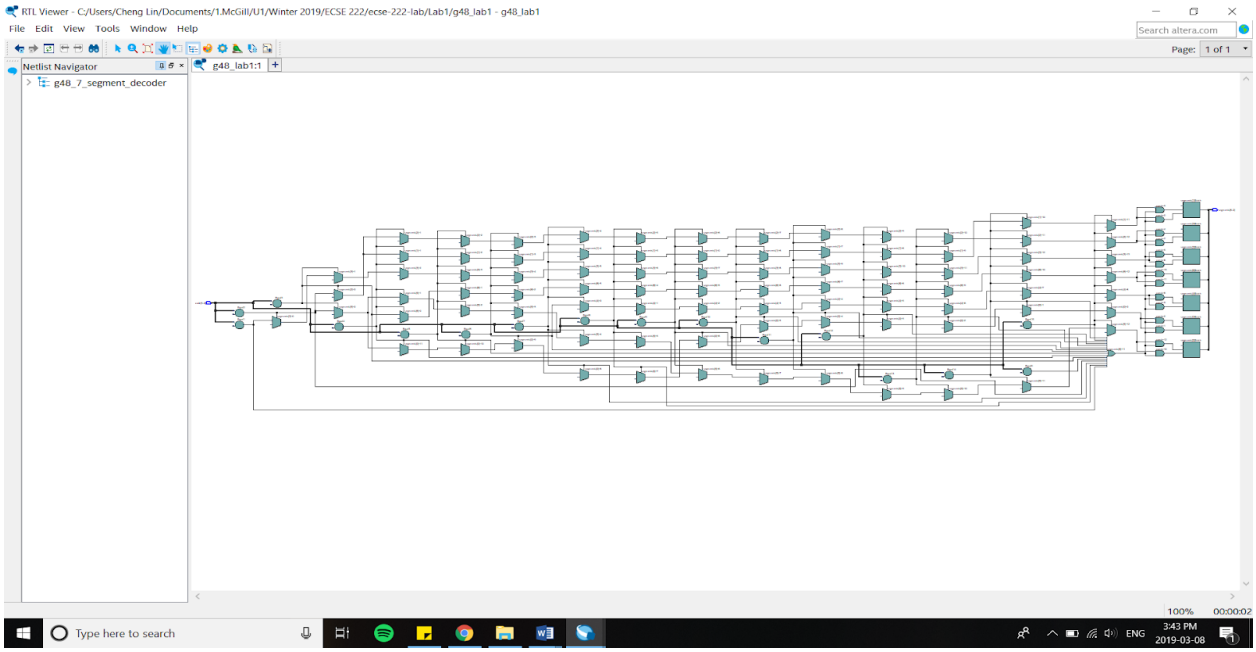
V FPGA Resource Utilization



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 adder.

VI RTL Schematics

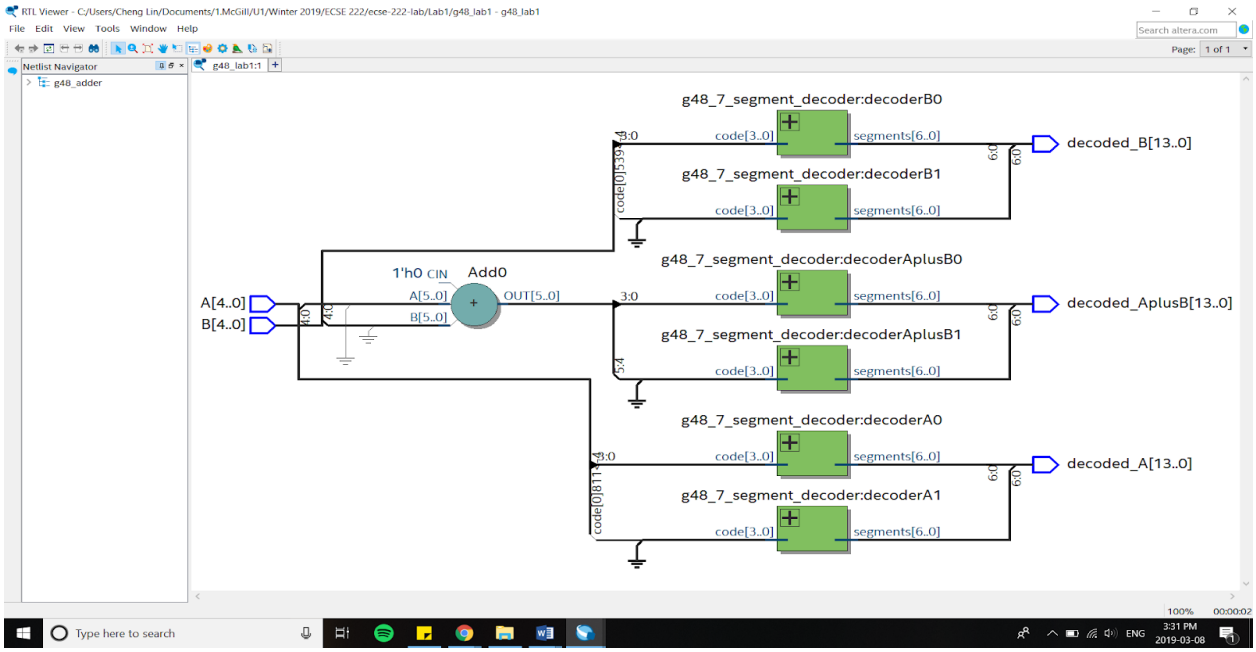
VI.I RTL schematic for the 7-segment decoder



The RTL schematic for the 7-segment decoder consists of many multiplexers. Each set of multiplexers corresponds to a single selected signal assignment statement. Since we have many of these assignment statements in our code, we have many multiplexers in our RTL schematic.

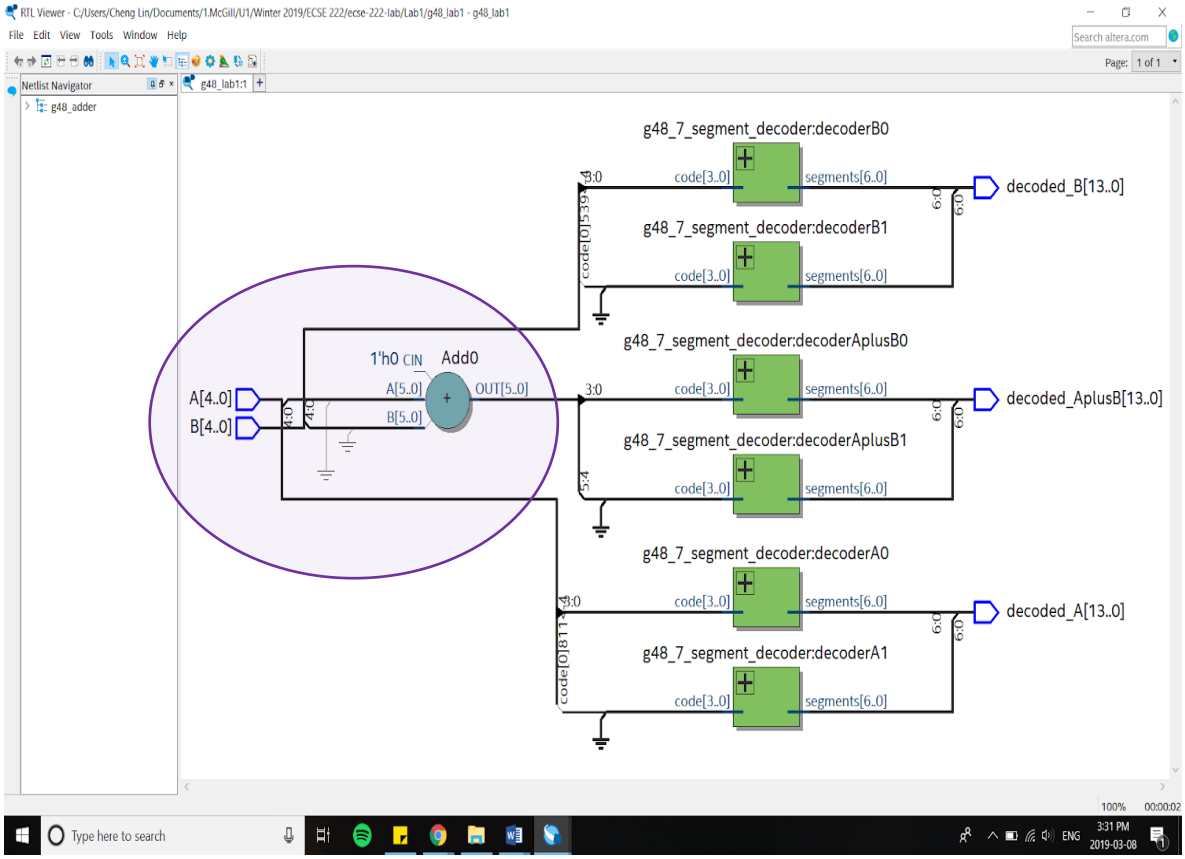
```
segments <= "1000000" when code = "0000" else
                        "1111001" when code = "0001" else
                        "0100100" when code = "0010" else
                        "0110000" when code = "0011" else
                        "0011001" when code = "0100" else
                        "0010010" when code = "0101" else
                        "0000010" when code = "0110" else
                        "1111000" when code = "0111" else
                        "0000000" when code = "1000" else
                        "0010000" when code = "1001" else
                        "0001000" when code = "1010" else
                        "0000011" when code = "1011" else
                        "1000110" when code = "1100" else
                        "0100001" when code = "1101" else
                        "0000110" when code = "1110" else
                        "0001110" when code = "1111";
```

VI.II RTL schematic for adder



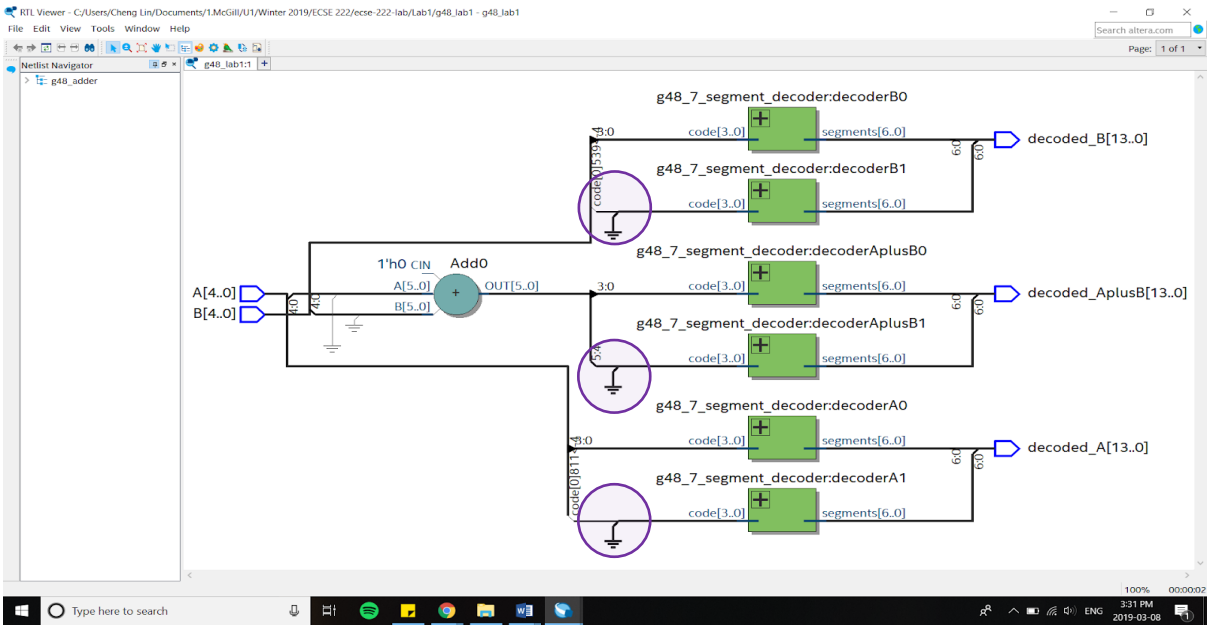
The left-most part of the 5-bit adder circuit (see marked area below) corresponds to the summing of the 5-bit inputs. As well, we concatenate a '0' to left of each input to hold the carry out of the sum.

```
AplusB <= std_logic_vector(unsigned('0' & A) + unsigned('0' & B));
```



Secondly, this part of our code assigns values to the 4-bit signals for the left (most significant) digits of each 7-segment display pair. The circuit also appends ‘0’s to the left so the values are 4 bits; concatenating the ‘0’s corresponds to the ground connections in the RTL diagram as marked.

```
A1 <= "000" & A(4);
B1 <= "000" & B(4);
AplusB1 <= "00" & AplusB(5 downto 4);
```

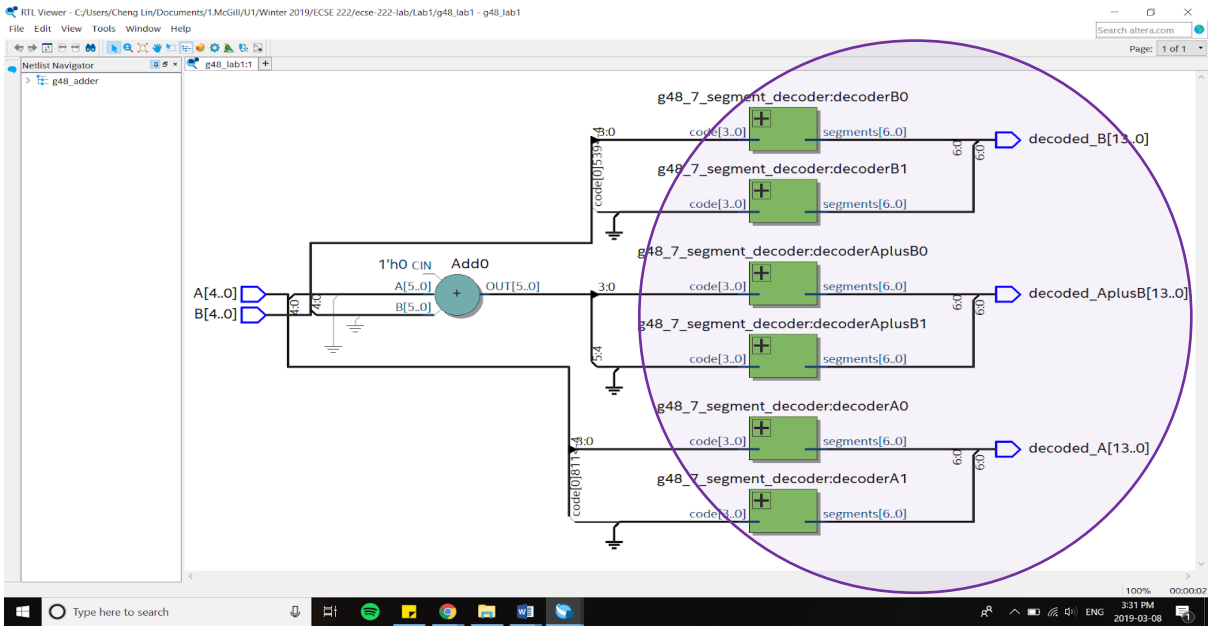


Lastly, this part of our code corresponds to the decoder entities in the diagram as marked.

```
--most significant digit of A
decoderA1: g48_7_segment_decoder PORT MAP(code => A1,
                                          segments => decoded_A(13 downto 7));
--least significant digit of A
decoderA0: g48_7_segment_decoder PORT MAP(code => A(3 downto 0),
                                          segments => decoded_A(6 downto 0));

--most significant digit of B
decoderB1: g48_7_segment_decoder PORT MAP(code => B1,
                                          segments => decoded_B(13 downto 7));
--least significant digit of B
decoderB0: g48_7_segment_decoder PORT MAP(code => B(3 downto 0),
                                          segments => decoded_B(6 downto 0));

--most significant digit of sum
decoderAplusB1: g48_7_segment_decoder PORT MAP(code => AplusB1,
                                          segments => decoded_AplusB(13 downto 7));
--least significant digit of sum
decoderAplusB0: g48_7_segment_decoder PORT MAP(code => AplusB(3 downto 0),
                                          segments => decoded_AplusB(6 downto 0));
```



Specifically, we declare 6 decoder entities (one for each 7-segment display), and create a port map that assigns the outputs to their respective bit sequences in decoded_A, decoded_B, and decoded_AplusB.

VII Conclusion

For this laboratory assignment, we programmed a binary-to-7-segment LED decoder and a 5-bit adder in the VHDL language and tested our product by using Quartus Prime's Test Bench Template Writer, ModelSim, and the Altera board. We were able to produce and verify our final product: a programmed Altera DE1-SoC board that adds two 5-bit binary numbers and displays the full summation on six 7-segment LEDs in hexadecimal. Overall, this experiment taught us the basics of VHDL, Intel's Quartus Prime and ModelSim, and downloading programs onto an FPGA board.