



McGill  
UNIVERSITY

# ECSE 222 Digital Logic Lab #1

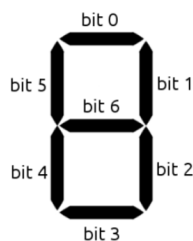
Alex Gruenwald, 260783506

Kaustav Das Sharma, 260772982

Group #50

**A description of the 7-segment decoder circuit. Explain why you used the selected signal assignment instead of the conditional signal assignment?**

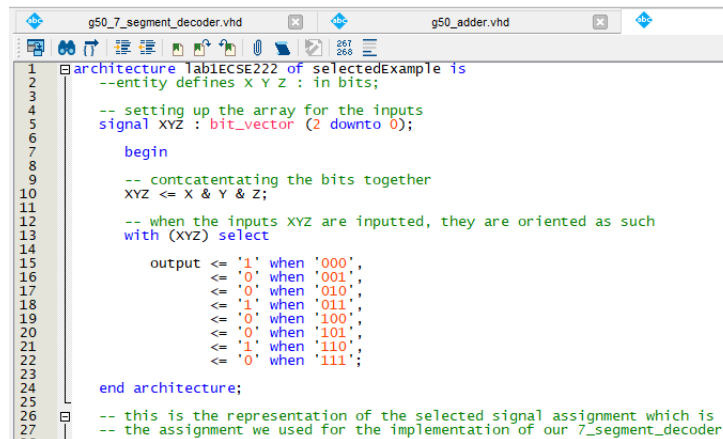
The `7_segment_decoder` is the VHDL code in our lab which defines how the displays on the Altera DE1-SoC board will display the values between 0 – 9 and A - F. Given the initial entity code, all that was required was to code a switch statement (different name in VHDL), going through each possible combination of 4 bits ranging from 0 – 9 and A – F. When a specific value in this range was given, it corresponded to the 7-bit representation of how it would be displayed on the Altera board. For example, if we wanted to represent a  $7_{10}$  in bits, it would be written  $0111_2$ . With this being the input for the case, the output would be 7 bits corresponding to the bit locations in Figure 1. If we want to create a '7' in the display, the bits corresponding to position 0, 1, and 2 must be one. Because we are basing our display on an active low (active when 0) these 3 bits must be '0' to actually display the right quadrant. The output for this example would be a 7-bit result  $\Rightarrow$  '1111000' where the least significant bit represents bit 0 and the most significant bit represents bit 6 in Figure 1.



**Figure 1:** Bit values and corresponding orientation on the display

When coding the case statement, there were two ways to implement the signal assignments – conditional or selected. Both of these assignments result in the same output but are implemented differently in code. A comparison of the two different types of implementation are shown below in Figure 2 and Figure 3 (random examples not related to actual decoder code). From observation, we can immediately see the difference between the two assignments where the selected signal assignment uses a switch case (similar to one in Java) and the conditional one uses an if / else statement. Implementing the conditional signal assignment implies that every statement must be checked in order to find a statement that matches for each loop. This is incredibly inefficient for the VHDL `7_segment_decoder` code shown in Figure 4, because there are 16 cases. When considering the selected signal

assignment, the switch cases allow for each input to find the right statement almost immediately, ensuring a quick and efficient implementation of the code.

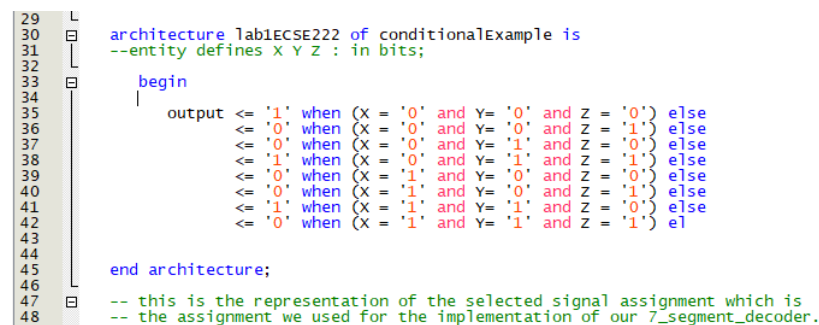


```

1 architecture lab1ECSE222 of selectedExample is
2   --entity defines X Y Z : in bits;
3
4   -- setting up the array for the inputs
5   signal XYZ : bit_vector (2 downto 0);
6
7   begin
8
9   -- concatenating the bits together
10  XYZ <= X & Y & Z;
11
12  -- when the inputs XYZ are inputted, they are oriented as such
13  with (XYZ) select
14
15    output <= '1' when '000',
16             <= '0' when '001',
17             <= '0' when '010',
18             <= '1' when '011',
19             <= '0' when '100',
20             <= '0' when '101',
21             <= '1' when '110',
22             <= '0' when '111';
23
24  end architecture;
25
26 -- this is the representation of the selected signal assignment which is
27 -- the assignment we used for the implementation of our 7_segment_decoder.
28

```

Figure 2: VHDL code representing how selected signal assignment is written



```

29 L
30 L
31 L
32 L
33 L
34 L
35 L
36 L
37 L
38 L
39 L
40 L
41 L
42 L
43 L
44 L
45 L
46 L
47 L
48 L
30 architecture lab1ECSE222 of conditionalExample is
31   --entity defines X Y Z : in bits;
32
33   begin
34
35     output <= '1' when (X = '0' and Y = '0' and Z = '0') else
36                <= '0' when (X = '0' and Y = '0' and Z = '1') else
37                <= '0' when (X = '0' and Y = '1' and Z = '0') else
38                <= '1' when (X = '0' and Y = '1' and Z = '1') else
39                <= '0' when (X = '1' and Y = '0' and Z = '0') else
40                <= '0' when (X = '1' and Y = '0' and Z = '1') else
41                <= '1' when (X = '1' and Y = '1' and Z = '0') else
42                <= '0' when (X = '1' and Y = '1' and Z = '1') el
43
44   end architecture;
45
46 -- this is the representation of the selected signal assignment which is
47 -- the assignment we used for the implementation of our 7_segment_decoder.
48

```

Figure 3: VHDL code representing how conditional signal assignment is written

To summarize our decision for using selected signal assignments, we did it for implementation efficiency. When considering large amounts of input bits, using the conditional assignment will result in a large waiting time for each if/else statement to be found. Using a selected signal assignment ensures a quick comparison and output for each loop.

**A discussion of how the 7-segment decoder circuit was tested, showing representative simulation plots. How do you know that the circuit works correctly?**

To test the 7-segment decoder circuit, we compiled the VHDL file on ModelSim and extracted the signals into the wave-viewer to run and visualise the output. The results of this simulation are shown **Figure 4 & 5**, and it shows that each of the corresponding outputs are decoded correctly. For example, the “0” represents the output on the display of the Altera and below it are the bits that need to be activated in order to produce it. These bits are “active-low” which implies that 0 represents the respective quadrant being “on”.

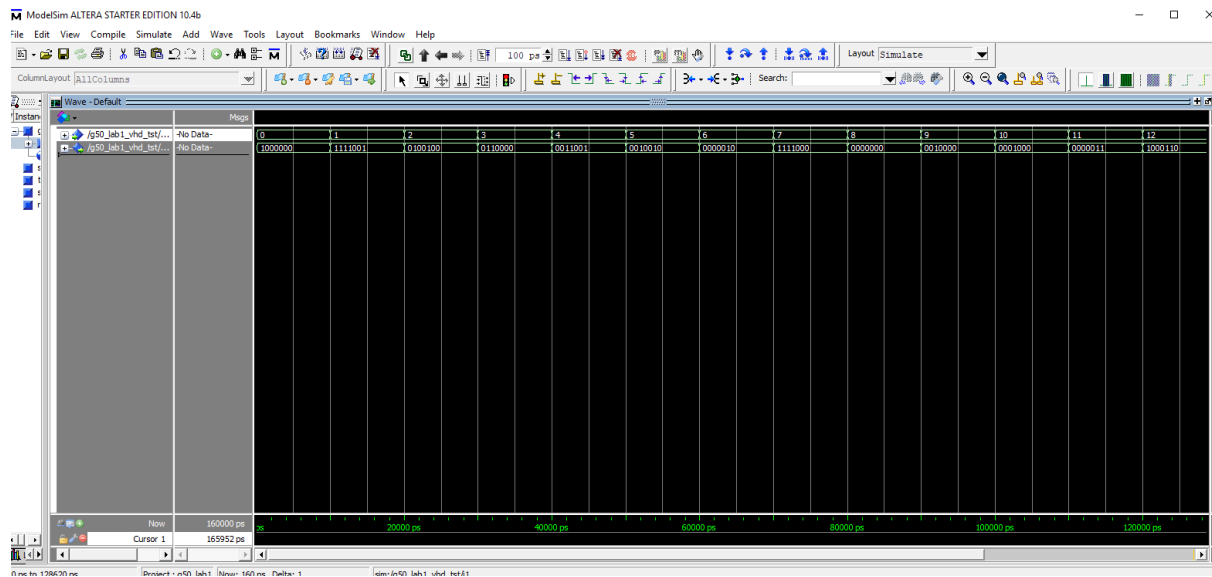


Figure 4: Simulated 7\_segment\_decoder (left-view)

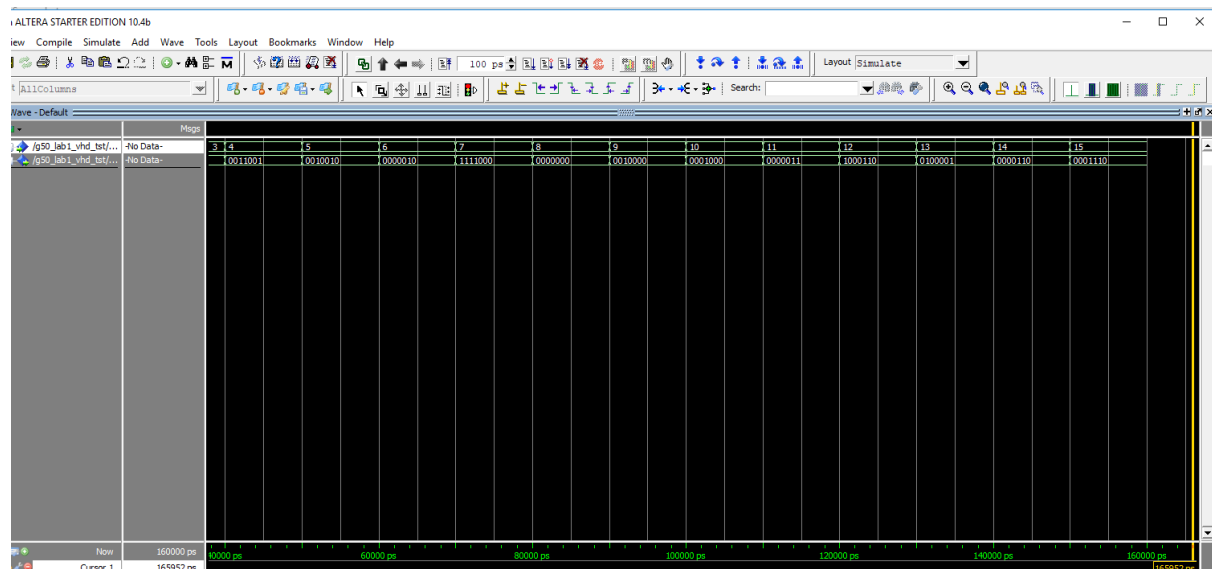
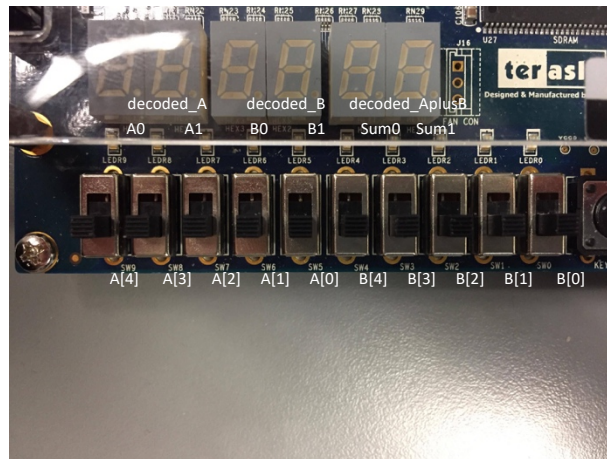


Figure 5: Simulated 7\_segment\_decoder (right-view)

A description of the adder circuit. How many 7-segment decoder instances did you use in your design and why?

The adder is directly connected to the LED displays on the Altera board. Since the LED displays on the board are grouped in twos, we labelled a grouping with two variables (A0 and A1) and used the size of the `std_logic_vector` to separate each display in the grouping. For example, (13 downto 0) can be separated into one range from (13 downto 7) corresponding to A0 and (6 downto 0) corresponding to A1. With the range separated, all six displays can then be accessed.

When considering the switches, they are separated into two sections, `std_logic_vector A` and `B` where `A` represents the first five switches and `B` represents the second grouping of switches as seen in **Figure 6**.



**Figure 6:** vectors `A` and `B` locations as seen on the switches

Depending on which switch is activated, a binary value, based on the location of the switch in the grouping of 5, will be sent to another vector `A_pad` for an `A` switch or `B_pad` for a `B` switch where the 5 bit binary value will be concatenated with an initial  $000_2$  to make an 8 bit output, which is then separated into two 4-bit groupings for the displays. The creation of the 4-bit value can then be inputted into the `7_segment_coder` and turned into a 7-bit combination which is read as a hexadecimal value on the display. For the 5-bit vector of `A` or `B`, initially seen as  $00000_2$ , the switch location on the board corresponds to the location of the binary value that will be changed to a 1.

For example, `B[2]` which is the third switch as seen in **Figure 6**, will change the 3<sup>rd</sup> bit in the 5 bits:  $00100_2$ . This is then concatenated with  $000_2$  in the `B_pad` vector to make  $00000100_2$  which is then split into two groups of 4-bits:  $0000_2$  and  $0100_2$ . Concatenating a  $000$  allows for an easy split of the now 8-bit value as the `7_segment_coder` requires at most 4 bits as an input and not 5. Also, the 5<sup>th</sup> bit can now easily be represented on the second display as it composed of 4-bits as well. These values are then inputted into the `7_segment_decoder` where  $0000_2$  ( $0_{10}$ ) will be converted as  $1000000_2$  and  $0100_2$  ( $4_{10}$ ) will be converted to  $0011001_2$ . Now, with the correct binary representation of a hexadecimal number, the `7_segment_decoder` outputs the 7-bit value into the LED display variables called

decoded\_A, decoded\_B, and decoded\_AplusB as seen on the display in **Figure 6**. For the final step, the decoded\_B and the decoded\_AplusB displays are updated with the input bits where the range from (13 downto 7) will have '1000000<sub>2</sub>' and the (6 downto 0) range will have '0011001<sub>2</sub>' which results in a display of '01' for both the decoded\_B display as well as the decoded\_AplusB display.

If the A switches were activated with the B switches, the same steps would apply to A, except now the decoded\_AplusB would have an extra step of adding the two binary digits together. In this addition, 6 bits (unsigned) of space are allocated in case the addition goes over the 5-bit limit.

For example, '10000<sub>2</sub>' + '10000<sub>2</sub>' = '100000<sub>2</sub>' which now represents '20<sub>16</sub>'

To answer the second part of the question, 6 instances of the 7\_segment\_decoder were used in the implementation of the adder code where all 6 instances can be seen in **Figure 6**. The reason for this implementation is because we are adding 5-bit values together. This 5-bit output is an issue for the input into the 7\_segment\_decoder because the 7\_segment\_decoder outputs a 7-bit binary value that represents a hexadecimal value. Hexadecimal values are written in pairings of 4-bits, so a 5-bit input would not result in a single hexadecimal value. To account for this, we concatenate '000<sub>2</sub>' to the end of each A, B, and sum vector resulting in each having a total of 8-bits. This is then split into two 4-bit variables (as seen in **Figure 6**), where each is then turned into a hexadecimal representation for the display. Since there are three groupings of displays, one for A, B and the sum, and each display has two hexadecimal values, there needs to be a total of 6 displays, where each display has an instance of the 7\_segment\_decoder linked to it.

### A discussion of how the adder circuit was tested?

To test the adder circuit, we tested various combinations of switch configurations, however for the purpose of this report, here are three specific test cases. Figure 7 represents the scenario where all the switches are turned on. As explained previously, the first four LED displays are the numbers (in hexadecimal) being summed to get the output in the last two LED displays. Hence for this test case, we have  $(1F_{16} \approx 31_{10}) + (1F_{16} \approx 31_{10}) = (3E_{16} \approx 62_{10})$ . Figure 8 is the scenario where the switches are turned off, which means it's  $(00_{16} \approx 0_{10}) + (00_{16} \approx 0_{10})$ .

$0_{10}) = (00_{16} \approx 0_{10})$ . Lastly, Figure 9 shows a specific test case triggering  $(01_{16} \approx 1_{10}) + (01_{16} \approx 1_{10}) = (02_{16} \approx 2_{10})$ .

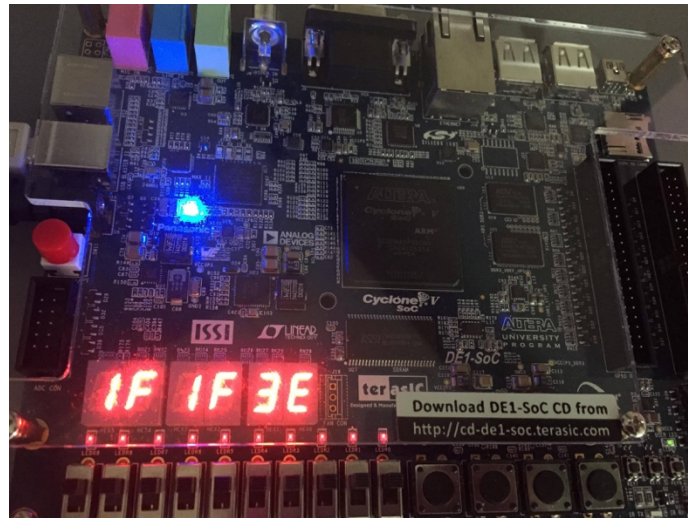


Figure 7: All switches on

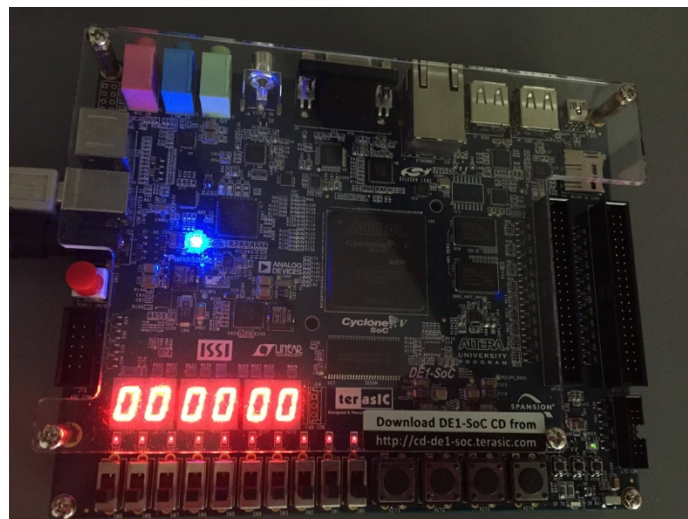


Figure 8: All switches off

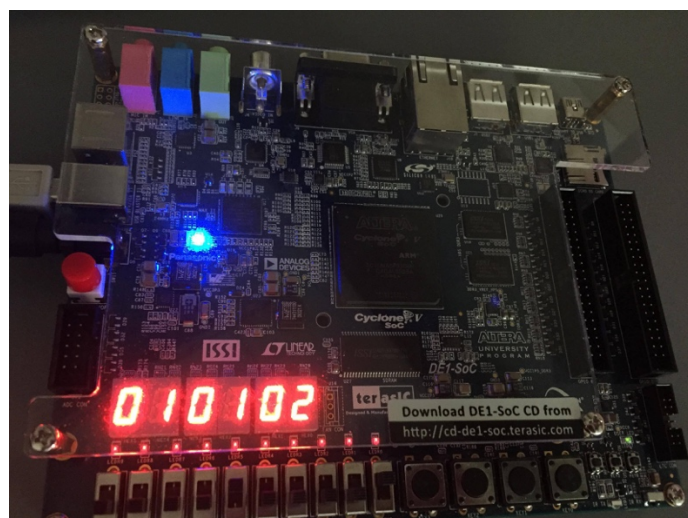


Figure 9: Basic adder test case

A summary of the FPGA resource utilization (from the Compilation Report's Flow Summary) and the RTL schematic diagram for both the 7-segment decoder and the adder circuits. Clearly specify which part of your code maps to which part of the schematic diagram?

Figure 10 corresponds to the compilation flow report summary outputted from our VHDL code in Altera. Figure 11 and 12 corresponds to resulting RTL schematic diagrams from our code.

Flow Summary	
Flow Status	Successful - Tue Mar 12 13:11:47 2019
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Standard Edition
Revision Name	g50_lab1
Top-level Entity Name	g50_adder
Family	Cyclone V
Device	SCSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	16 / 32,070 (< 1 %)
Total registers	0
Total pins	52 / 457 (11 %)
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 %)

Figure 10: Compilation flow summary report

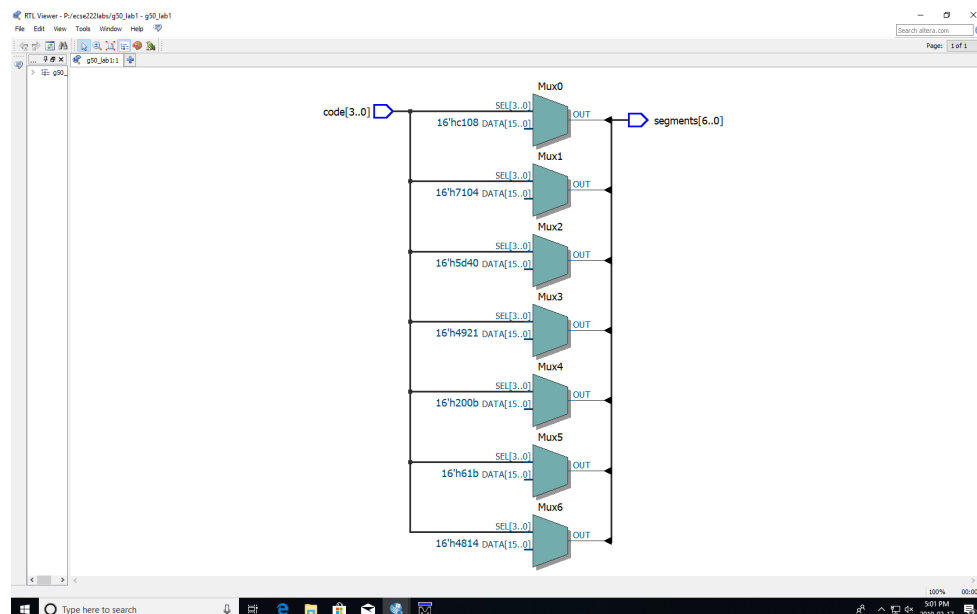
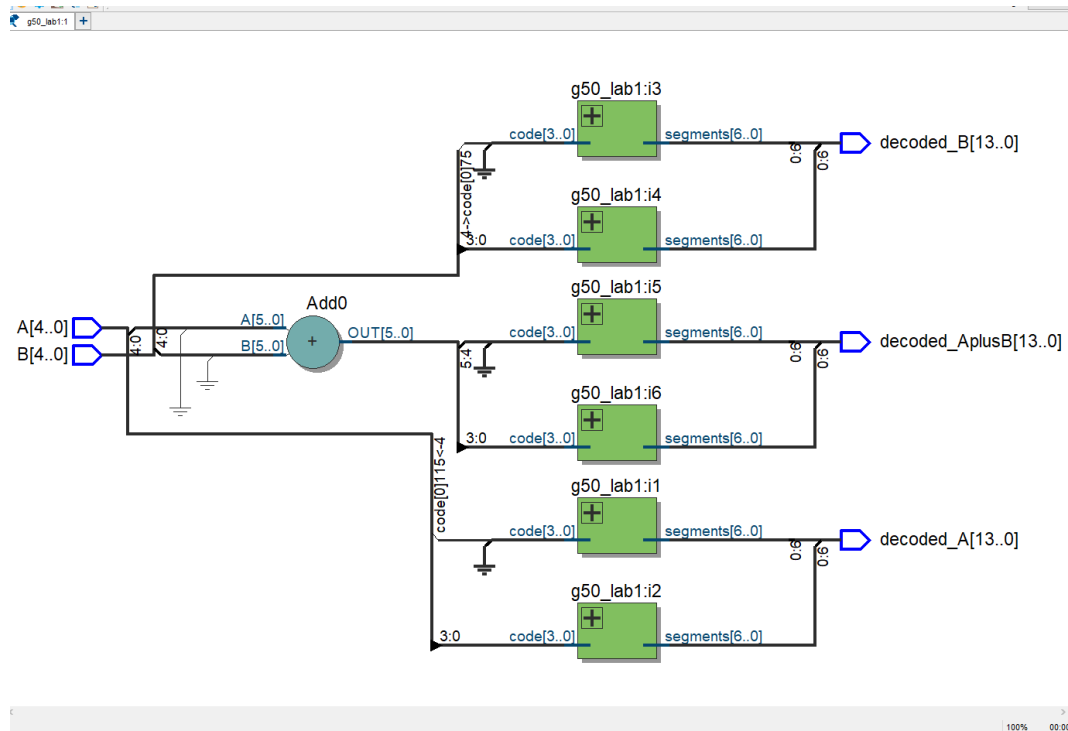


Figure 11: RTL Schematic for decoder circuit



The code[3..0] in the schematic diagram above corresponds to the input which is a 4 bit value, which then goes through a series of multiplexers which transforms the input to the desired 7 segment display representation (segments[6,0]), by turning active low bits into 0s.



**Figure 12:** RTL Schematic for adder circuit

In the schematic diagram above, the A[4..0] and B[4..0] corresponds to the switches on the hardware, which then has three pathways. The first corresponds to the decoder which parses the input signal and displays the resultant hexadecimal into i3 and i4. The same happens in the bottom pathway, except it displays onto i1 and i2. The middle pathway corresponds to the display which adds A and B, so it goes through the adder and then parses it to a hexadecimal and displays on i5 and i6.