3C7 Assignment 1
David John Neill
16331809

Abstract:

The purpose of this lab was to code a mini arithmetic logic unit using Verilog. Once the code was written and tested using the waveform generator on Vivado we then put the code on the board. But first we had to set up the inputs and outputs. The switches were in the inputs and the LEDs were the outputs. The majority of this code was taken from previous Labs (Labs A-E). However, some of the code from these labs needed to be altered as they were for 8 bits but this ALU was for 6 bits.

Aims:

- To write a mini arithmetic logic unit using Verilog and to test it in Vivado
- To understand how important it is to fully test your design before implementing in hardware.
- To program the board with the code that is fully tested using Vivado

Experimental Set-up:

In this assignment there was only 2 main components that we needed to set up and use:

1. Vivado on a windows computer.
2. A Basys 3 board and a connecting cable to the computer.

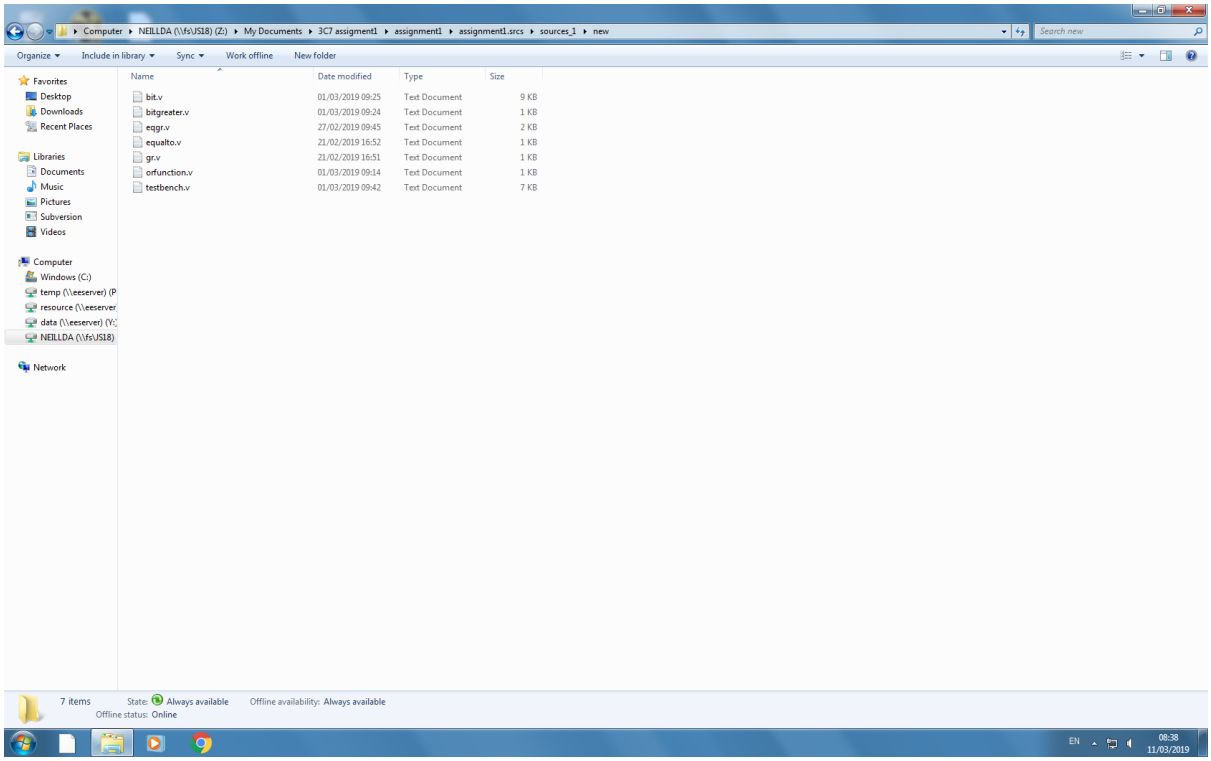
Procedure:

1. Open up Vivado and create a new project and name it.
2. Find all the code from previous labs and add those files to the project.
3. Create a new testbench and add this to simulation sources.
4. Add a top level file which will have the main purpose of calling other files (instantiations), in this file there should be logic. In my case this file was called bit.v.
5. Draw out a block diagram to give yourself a visual representation to work with.
6. Start adding in instantiations to files that you have already written and if the files need changed or altered do this now.
7. Re-evaluate your block diagram and see if anything needs changed.
8. Now finish the code in the bit.v file using logic (do this as it easier for you to read and debug) and once this is done do a small test.
9. Now rewrite the logic you coded (part 9) but this time create files and instantiations.
10. Again do a small test to make the basics of the code is working.
11. Once you are happy with your code after small testing you then begin to write your testbench.
12. When writing the testbench make too sure incorporate all the different possible cases, eg overflow, cout etc….

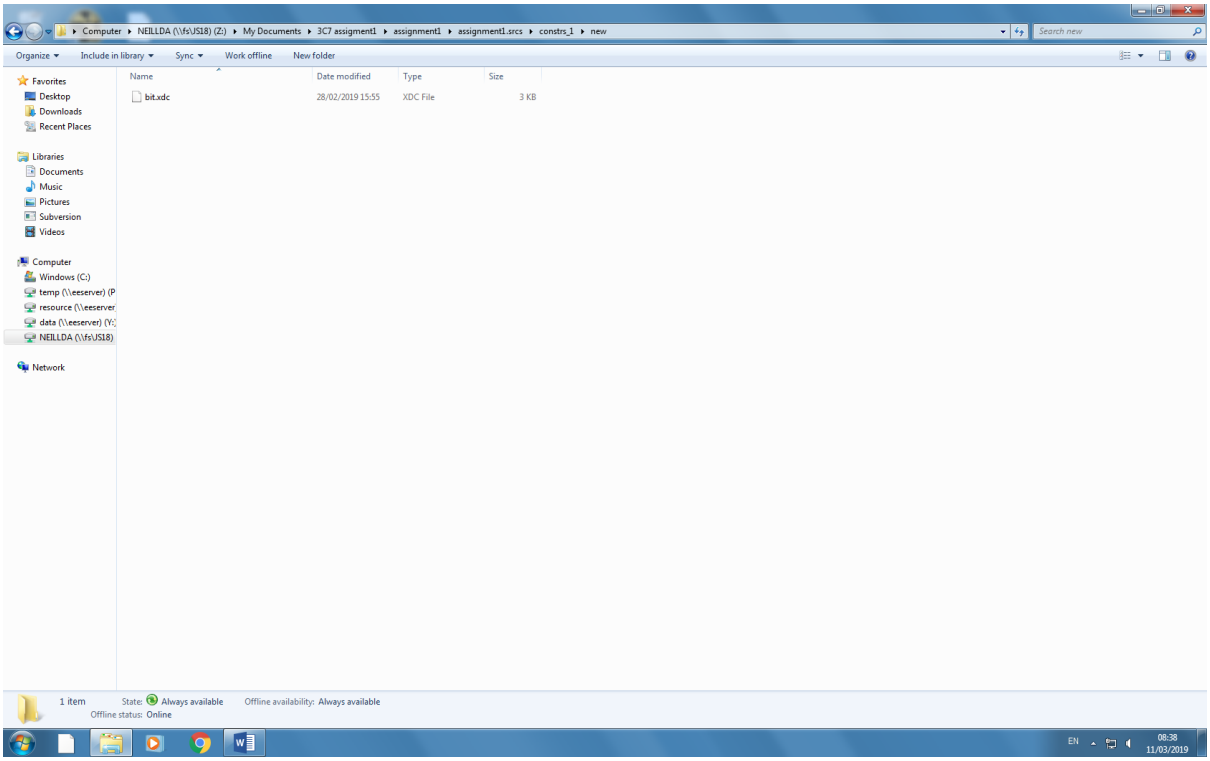
13. Make sure that you test each value of fxn to an appropriate degree.
14. Once you have fully tested your board, now you can program your board with the code.
15. You first need to label your inputs and outputs and change LVCMOS18 to LVCMOS33.
16. Then run all the appropriate implementation, synthesis and bitstream and then program the board using hardware manager.
17. Now test the board using the switches and LEDs that was programmed in step 16.

Below you can see the hierarchy of my design (fig2).
The bit.v file contains all of my instantiations and no logical. The rest of files – orfunction.v , fulladder.v, equalto.v, gr.v and bitgreater.v contain all of my logic that essential makes the ALU work. The testbench is called adder1_tb and you can see under simulation sources. It is in this folder because it used for simulation on Vivado. If you try to upload this file to the board it would cause critical errors.

Then beneath that you can clearly see the Block Diagram.

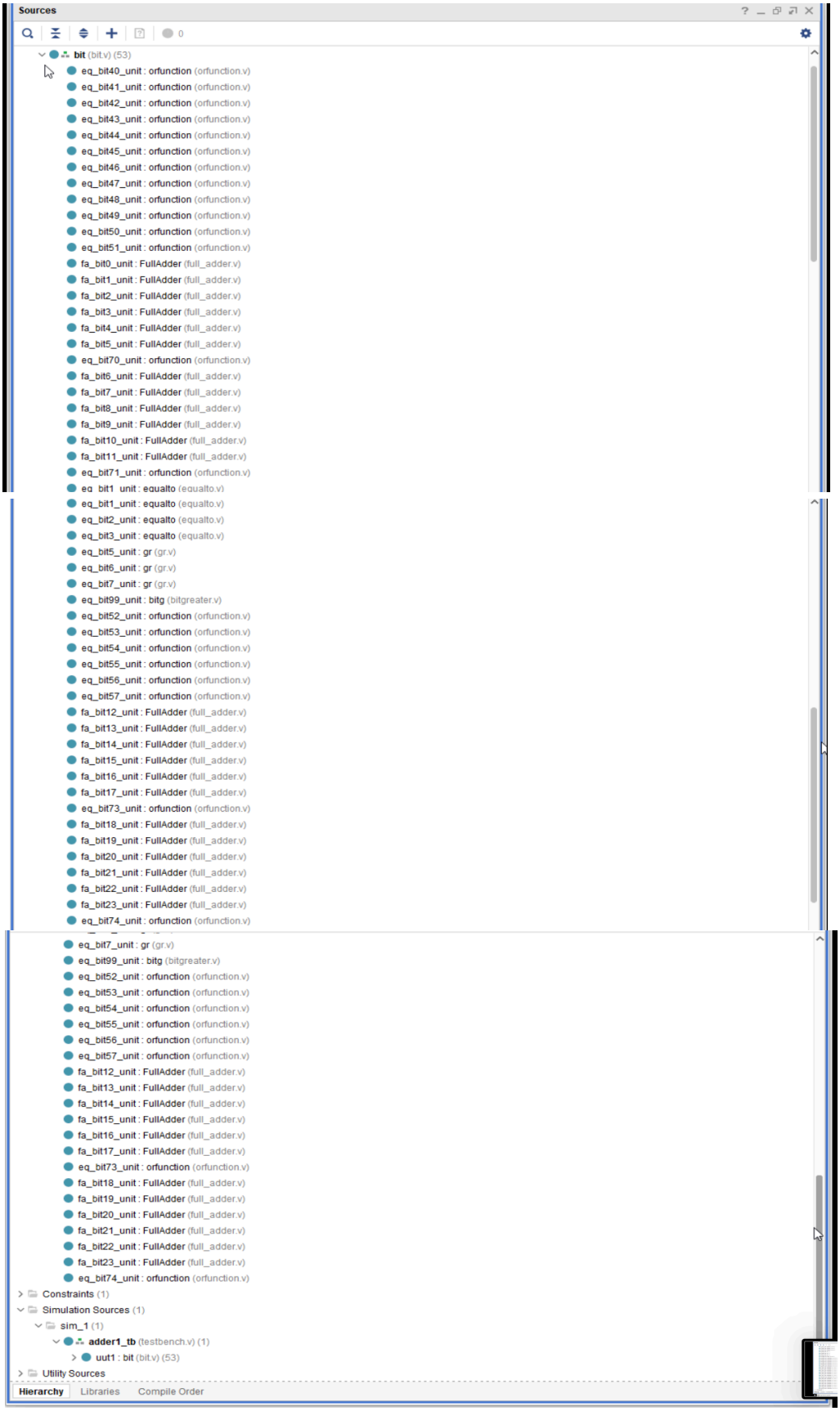


File directory of sources (fig1)

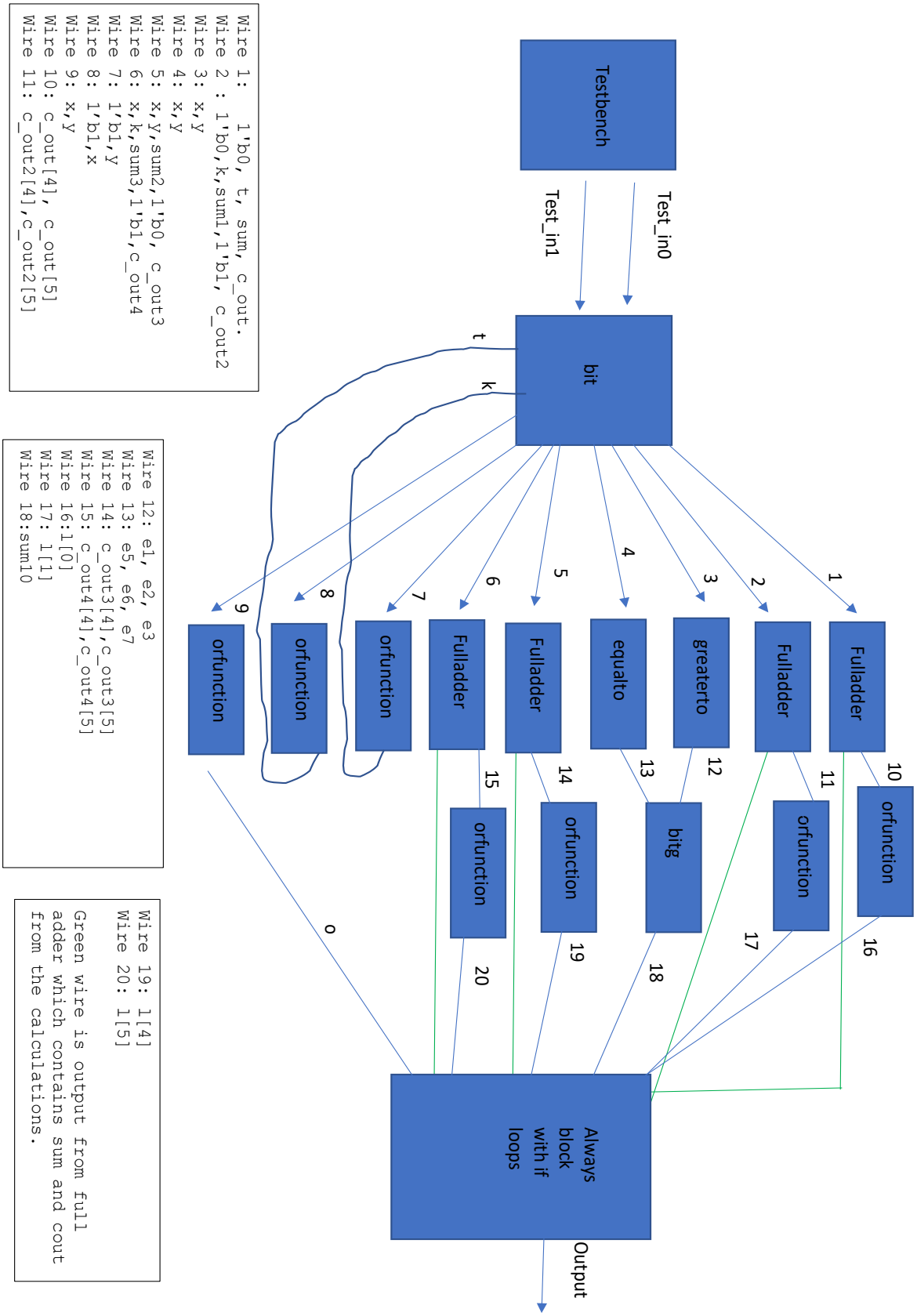


File directory of constraints (fig2)

Hierarchy of my design (fig3).



Block Diagram (fig4).




```
Wire 1:    1'b0, t, sum, c_out.
Wire 2 : 1'b0,k,sum1,1'b1, c_out2
Wire 3: x,y
Wire 4: x,y
Wire 5: x,y,sum2,1'b0, c_out3
Wire 6: x,k,sum3,1'b1,c_out4
Wire 7: 1'b1,y
Wire 8: 1'b1,x
Wire 9: x,y
Wire 10: c_out[4], c_out[5]
Wire 11: c_out2[4],c_out2[5]
```

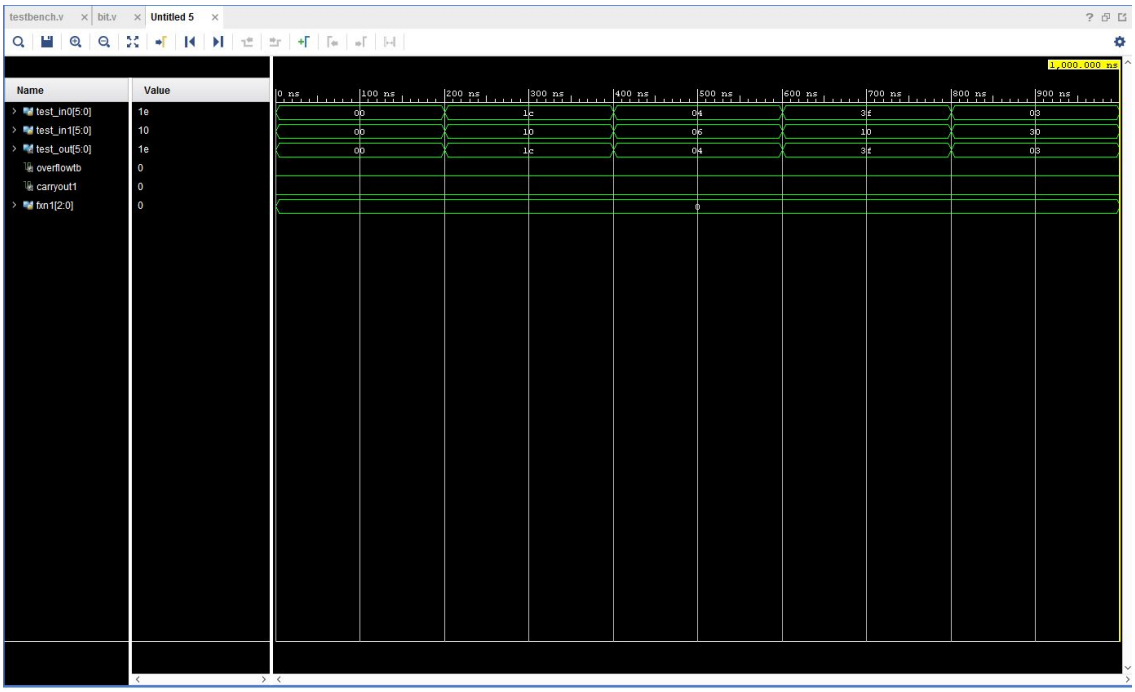
```
Wire 12: e1, e2, e3
Wire 13: e5, e6, e7
Wire 14: c_out3[4],c_out3[5]
Wire 15: c_out4[4],c_out4[5]
Wire 16:1[0]
Wire 17: 1[1]
Wire 18:sum10
```

```
Wire 19: 1[4]
Wire 20: 1[5]

Green wire is output from full
adder which contains sum and cout
from the calculations.
```

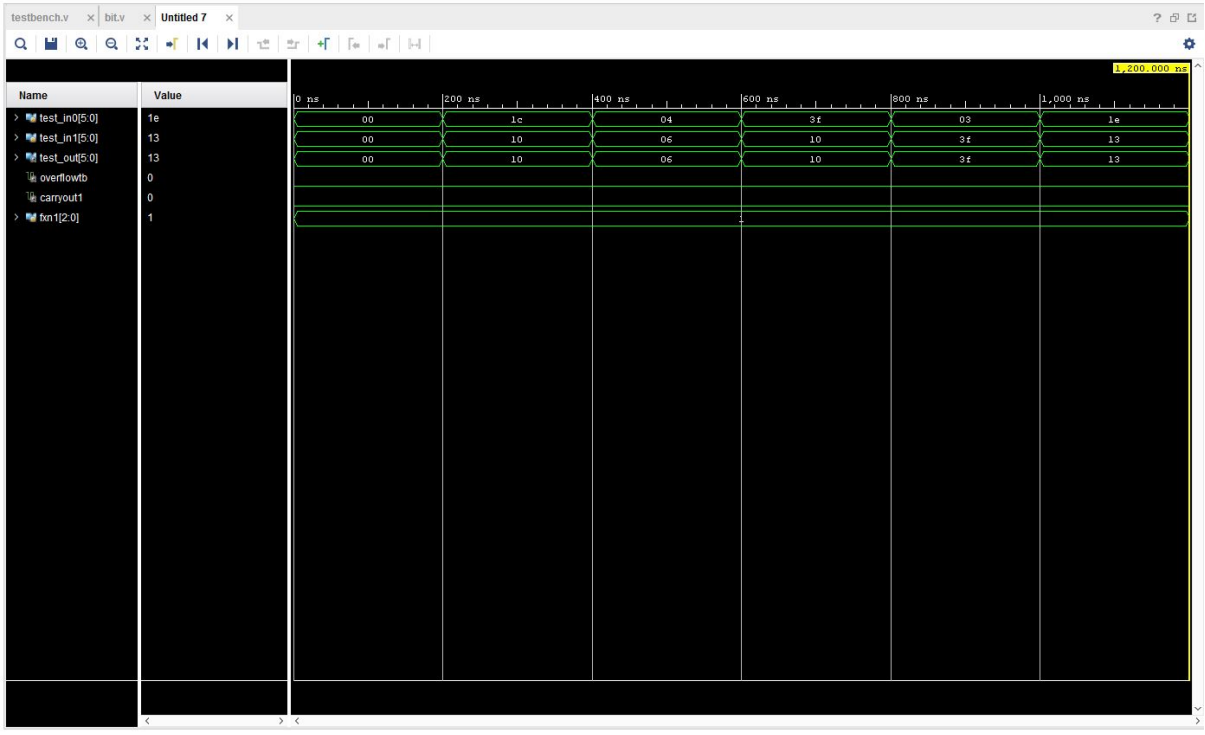
Results:

A



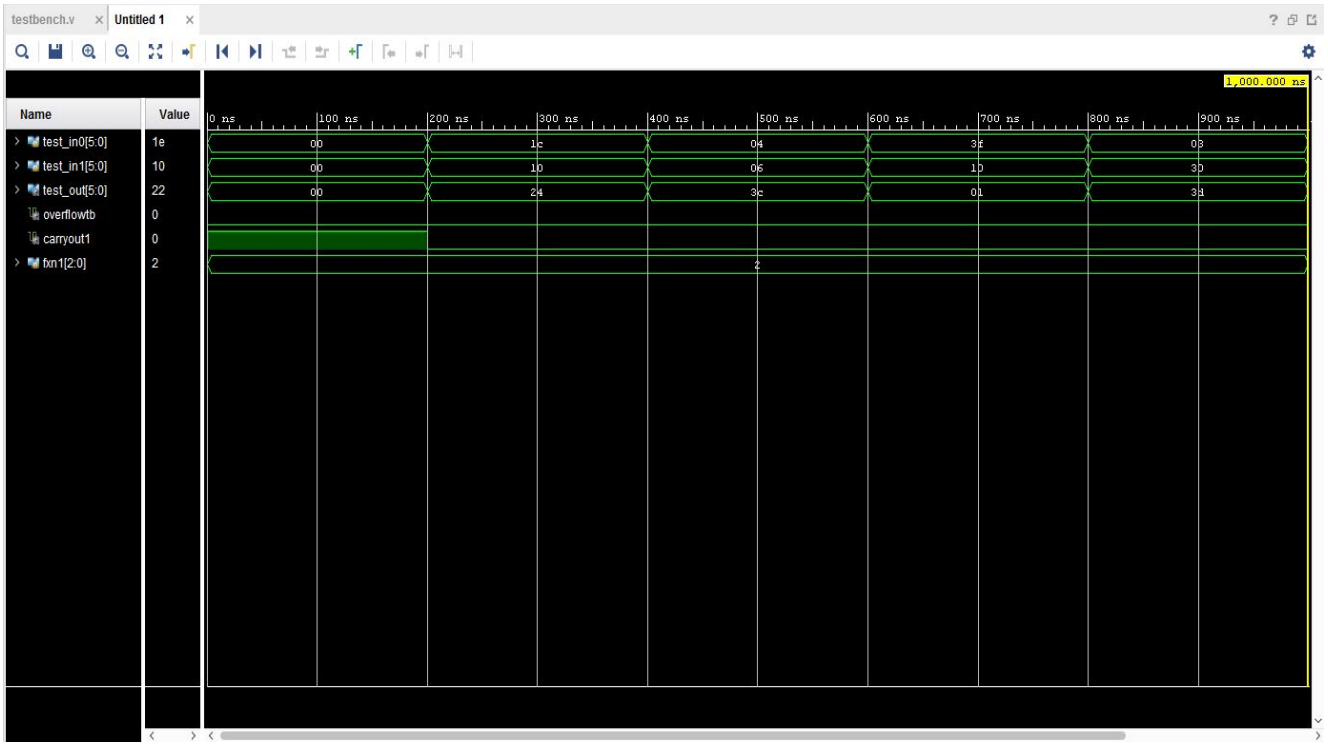
Fxn = 000 (fig5)

B



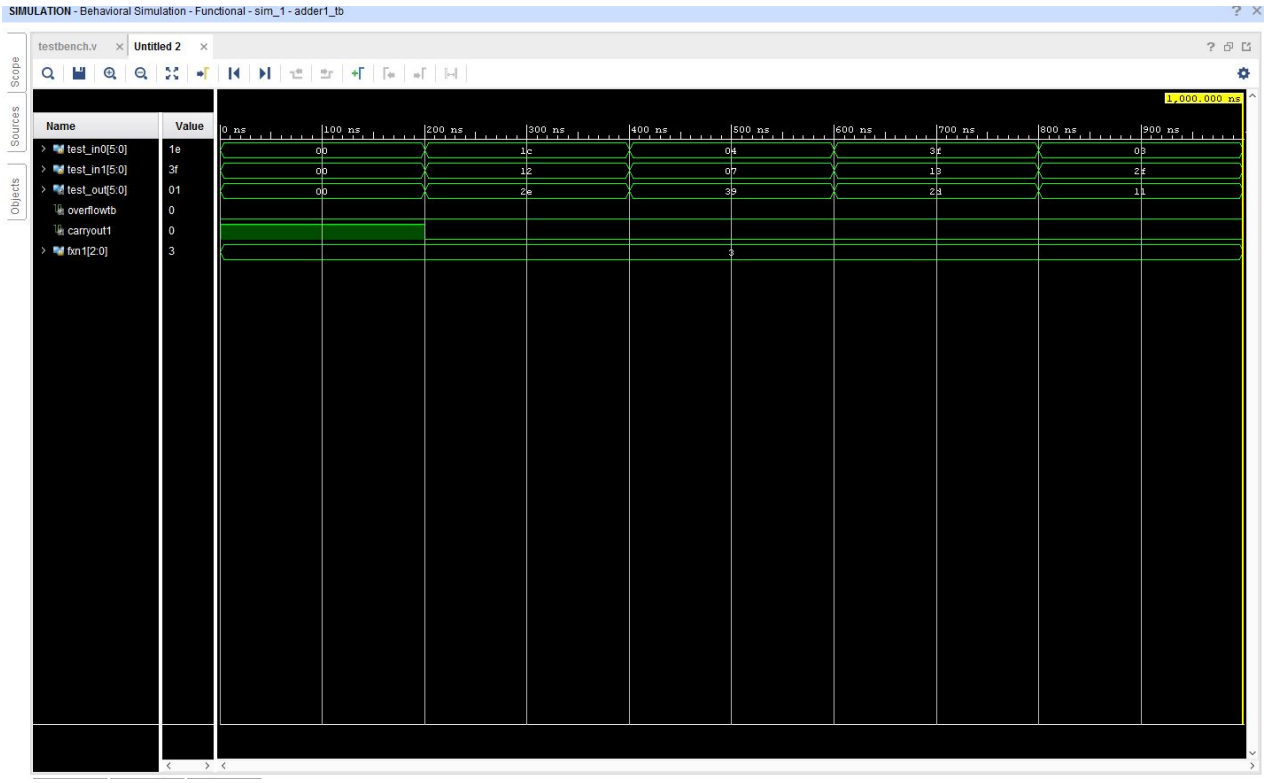
Fxn = 001 (fig6)

-A



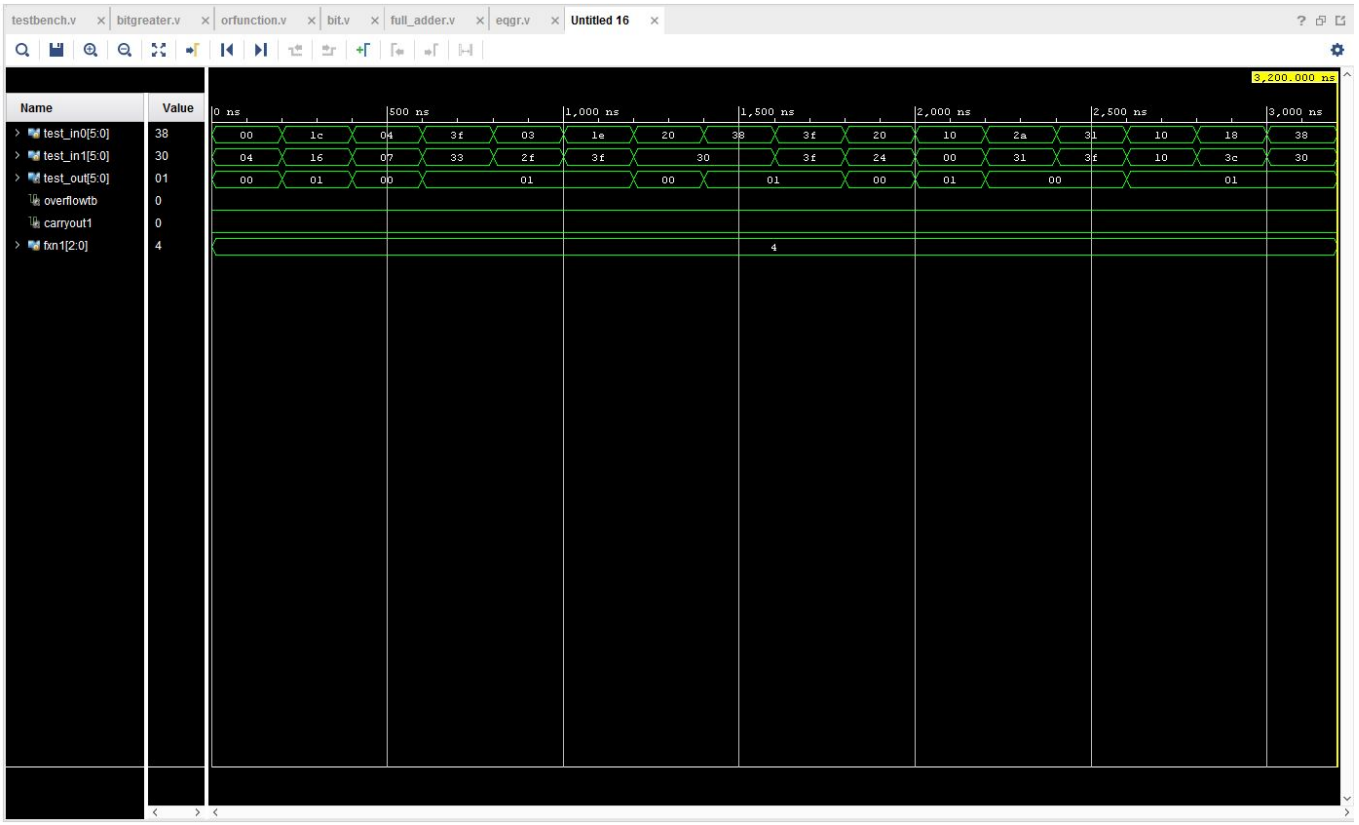
Fxn = 010 (fig7)

-B



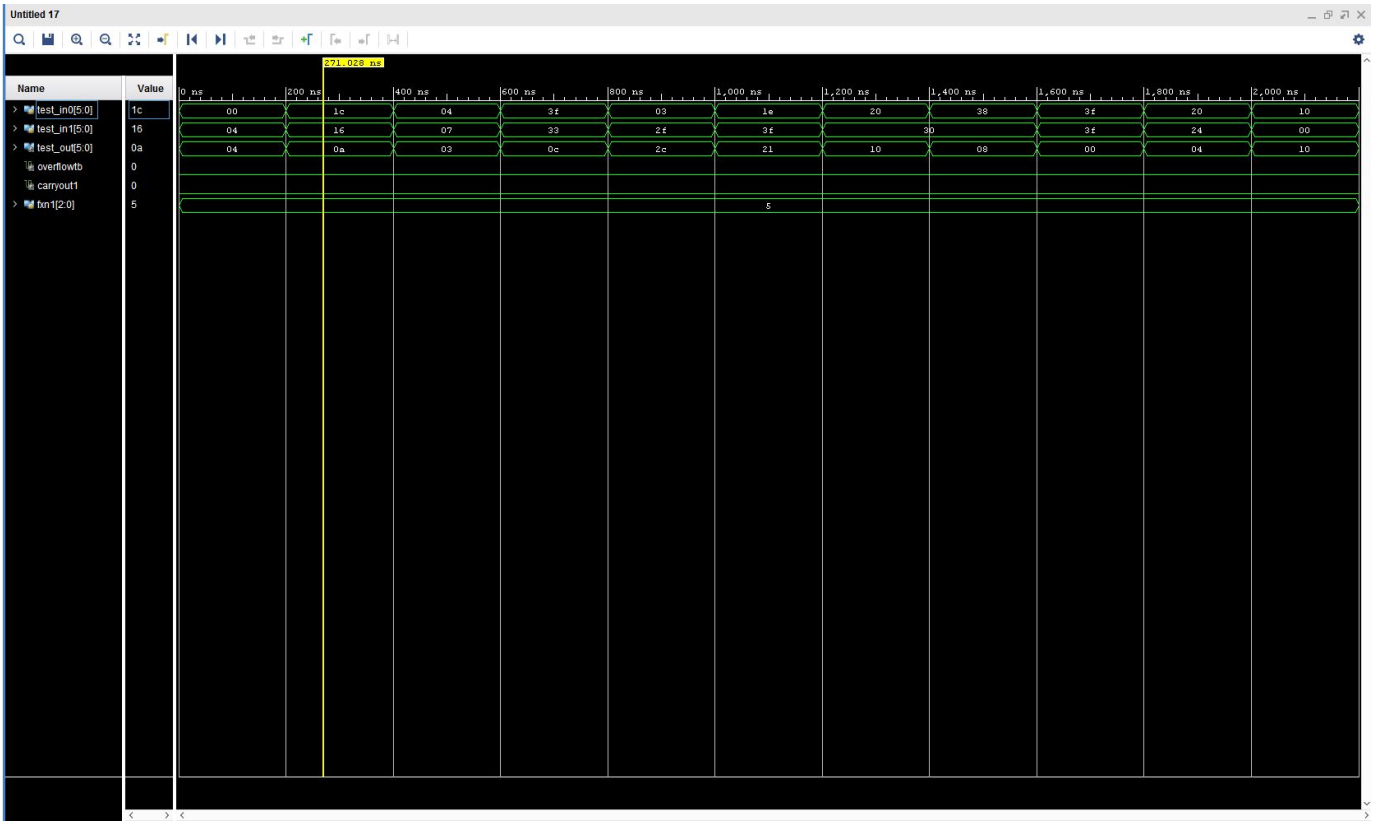
Fxn = 011 (fig8)

A>=B



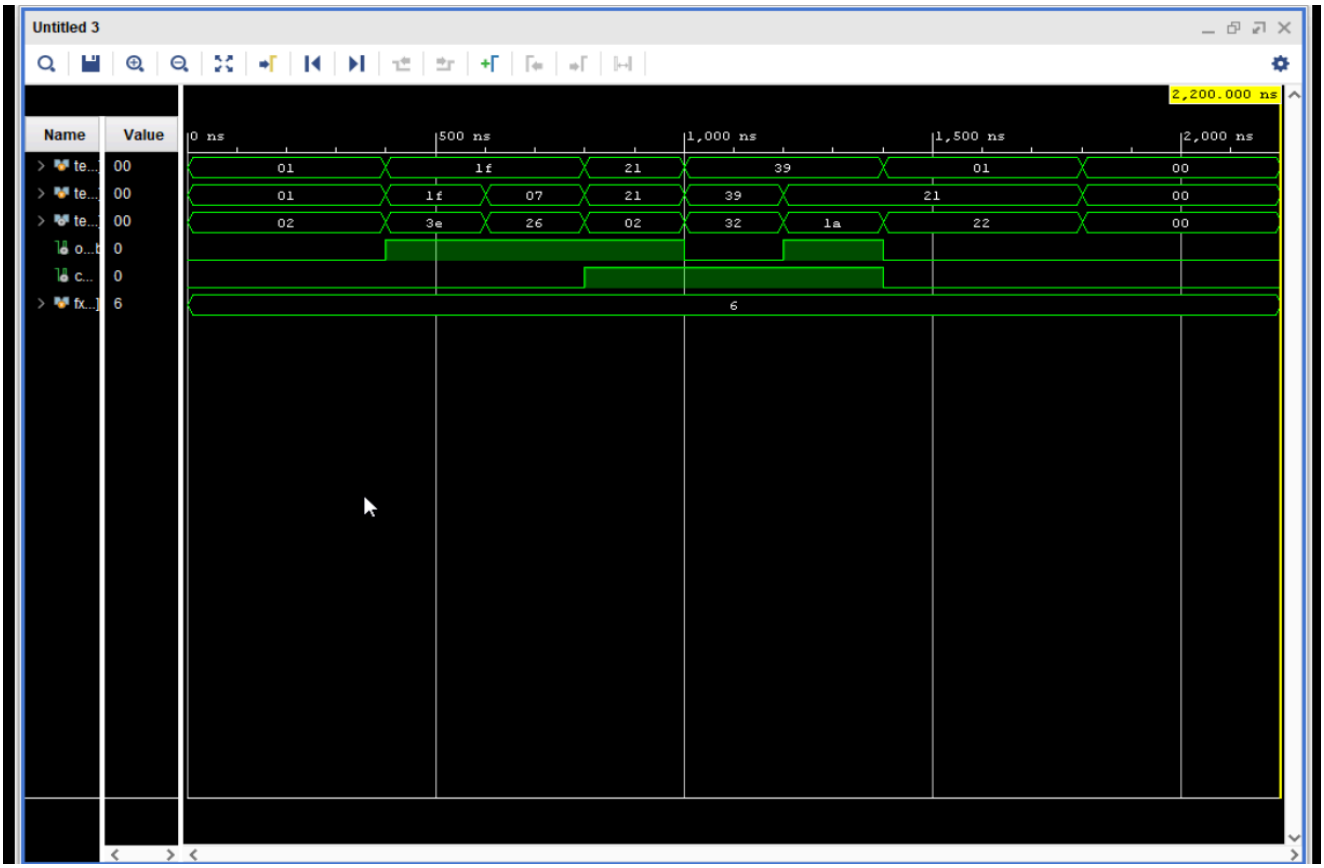
Fxn = 100(fig9)

A^B



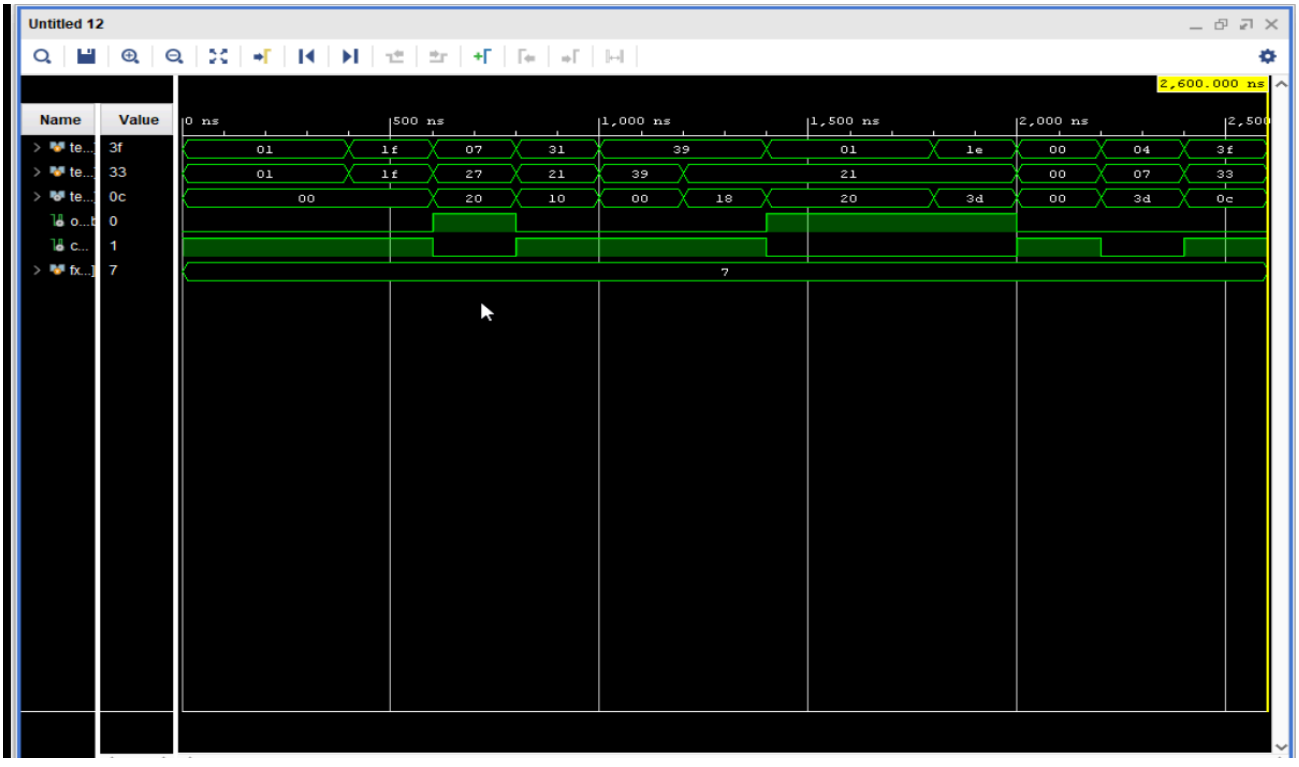
Fxn = 101 (fig10)

A+B



Fxn = 110 (fig11)

A-B



Fxn = 111 (fig12)

Discussion:

Looking at my waveforms I can see that my results are correct and that the values shown on the waveform are the same as the expected values. When some of these were tested on the board they also worked correctly. For values of fxn >=100 I took the test bench from the corresponding lab and then added in a few more tests just to confirm that the was working. My testing I think is very thorough and covers all the different possibilities. I have attached my results to the submission as I did not put all the tests into waveform as it would be to difficult to see. Each function I felt had the appropriate number of tests where completed. The more complicated the arithmetic the more test cases it received. However, when looking through my results I did get one answer I wasn't expecting. When looking at the -A and -B waveforms you can see that you get carryout when you want the minus number of 0. This is actually correct if you look at calculations below:

0= 000000

To get minus you need to flip bits and then add one:

```
000000->1111111
                 111111
                 +      1
                 ----------
             1   000000
```

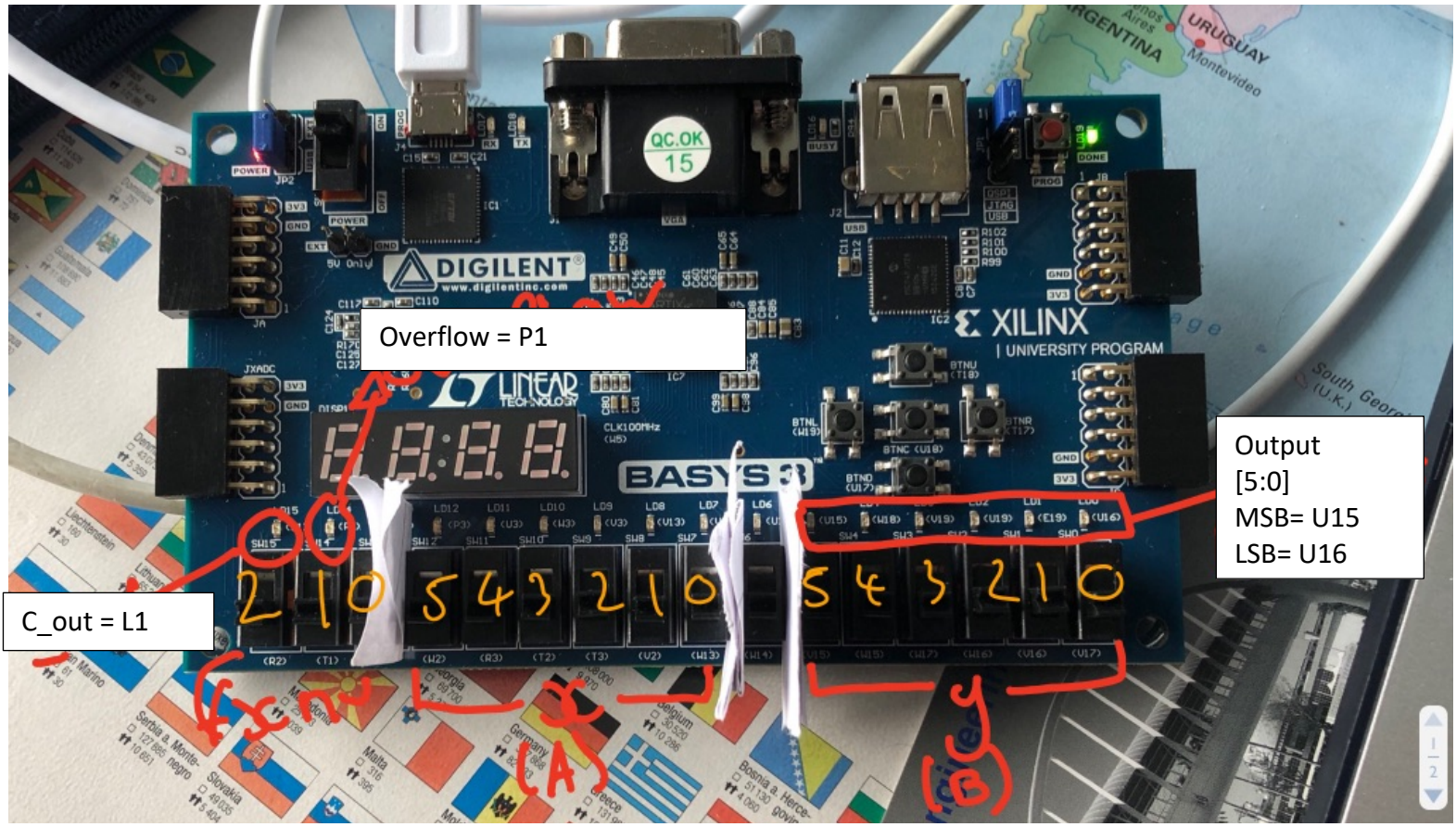
Therefore carry out is turned high.

If I wanted to improve my design I could have used the 7 segment display however this would have been very complicated to do

Modifications:

The only modifications that were made was to the 8 bit equal to or greater than. I changed this because the logic equation that I had would only work for 8 bits. So I went back to my notes that I used to develop the 8 bit equation and changed it so it would now work for 6 bits.
I primarily used the same testbenches that I used for each previous lab, the only difference being I added more test cases to prove that it was definitely full testing and working correctly.

Demo:



When the switches are down it means it's a zero and when switches are up they are a 1.

As you can see from the diagram above all the switches and LEDs that are used are labelled. The corresponding values are coded in my bit.xdc file.
To recap:
Fxn = R2-> U1, MSB = R2
X (A) = W2-> W13, MSB = W2
Y(B) = V15 -> V17, MSB = V15
OUTPUT = U15->U16, MSB =U16
Overflow = P1
C_out = L1

Conclusion:

In conclusion my ALU worked exactly as it should have. It worked correctly for all the test cases and any edge cases. I am confident that it would work for any set of numbers that user could possibly enter into it. My ALU also works correctly on the board with no errors. If

this was the real world of work, I would be happy to move on to the next stage of production.

I have attached the code to a different file as I feel it would make this document unnecessary long.