Lab 2 Report—Floating Point Conversion

Karthik Ramesh 104687637

Chungchhay Kuoch 004843308

Introduction:

Our goal for this lab was to convert a 12-bit two's-complement number into the best approximation of an 8-bit floating-point representation. We had three components for the 8-bit number.

We did not use the FPGA board for this lab; we only used the simulations for this lab. We tested our code using a test bench file we made with different input values and checking that the output for the corresponding inputs and outputs were proper/matching our calculations.

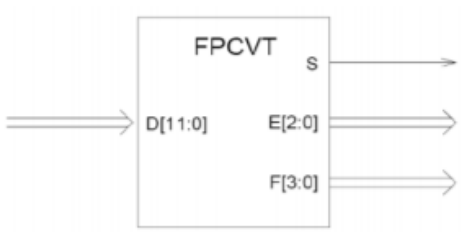
The diagrams below show how we represent 12 bits into 8 bits number:

MSB (7^{th} bit): signed bit = S;

6^{th}-3^{rd} bits: significand = F;

2^{nd}-0^{th} bits: exponent = E.

→ 8-bit number = $(-1)^S * F * 2^E$



FPCVT Pin Descriptions	
D [11 : 0]	Input data in Two's Complement Representation. D0 is the Least Significant Bit (LSB). D11 is the Most Significant Bit (MSB).
S	Sign bit of the Floating Point Representation.
E [2 : 0]	3-Bit Exponent of the Floating Point Representation.
F [3 : 0]	4-Bit Significand of the Floating Point Representation.

Design Description:

There were three modules we implemented. We connected all these modules together in a "master" file that called all the functions from each of the files and put it all together. Based on the design diagram, we created a module for converting from twos-complement to signed magnitude, another for counting leading zeros & extracting the leading bits, and lastly for rounding any numbers that are not able to be represented properly by the 8-bit floating point representation, due to problems like overflow.

Connecting module

This is the master module, with figure 1. We can see how the files all connect. There is a 12-bit two's-complement number that we convert into an 8-bit floating-point representation. We

split the project into three main modules that do the work: converting to signed magnitude, counting/extracting, and rounding.

Below is the overall look of each of those files such as 2s'-complement, count and extract, and rounding.

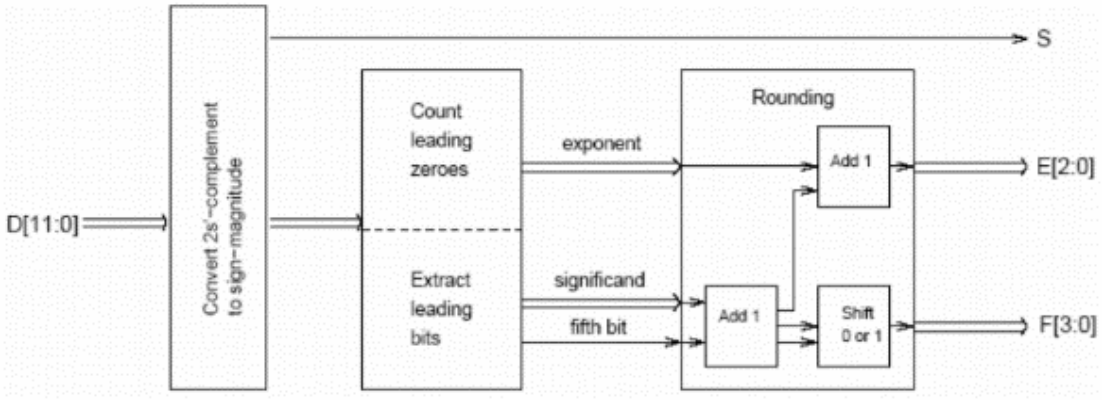


Figure 1: Overall look of the master module's design.

Two's Complement to Signed Magnitude

In this module, we focused on converting from a two's complement number into signed magnitude. We did this by checking the most significant bit to see if it was positive or negative. If the bit was 0, it was positive, we did nothing, and just passed on the number as it is. However, the bit showed a 1, it meant it was negative, and we used the formula in (Eq 2) to convert it to a proper signed magnitude number:

$-x = \sim x + 1$ …….(Eq 2)

There were two outputs for this module: the signed bit, and the rest of the converted signed magnitude number.

The following figure shows our logic design implementation for converting from two's complement to signed magnitude. Note: due to overflow, we had to manually code for the case of the most negative number: -2048 a case wherein the ending 8-bit floating representation number showed up as the most negative number possible: 11111111.

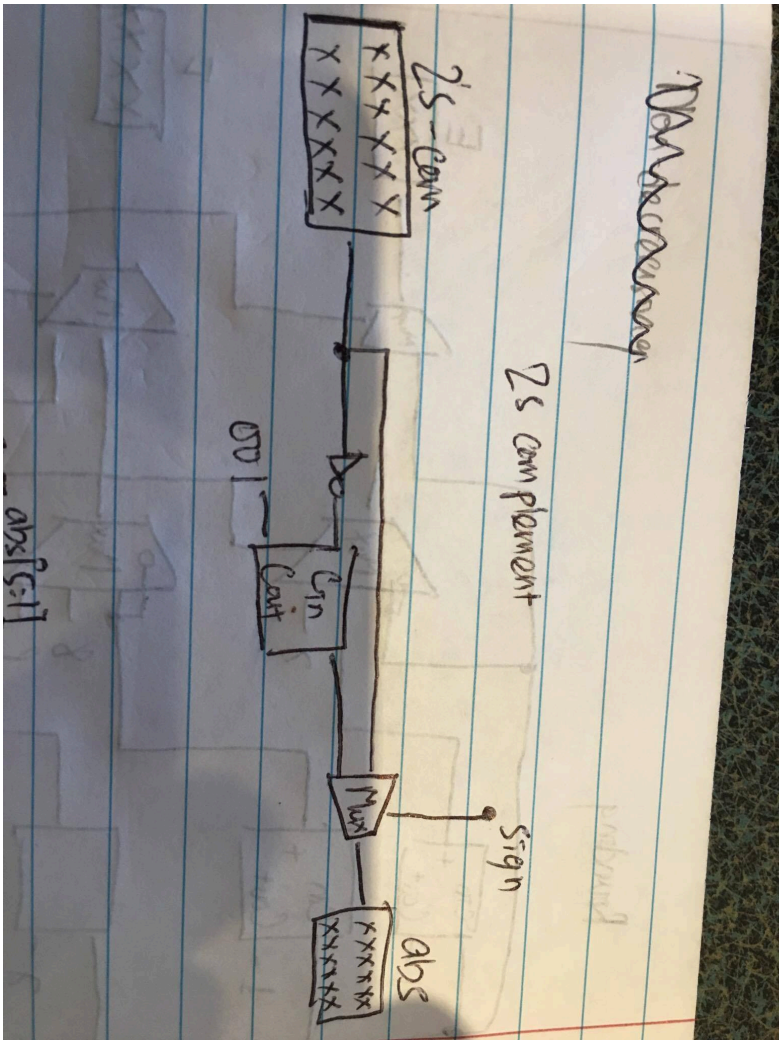


Figure 2: the logic design implementation for this module

Count and Extract

This module takes in the signed bit and signed magnitude output from the first module as input. It outputs a temporary exponent, temporary significand, and a fifth bit status that will be used later for rounding.

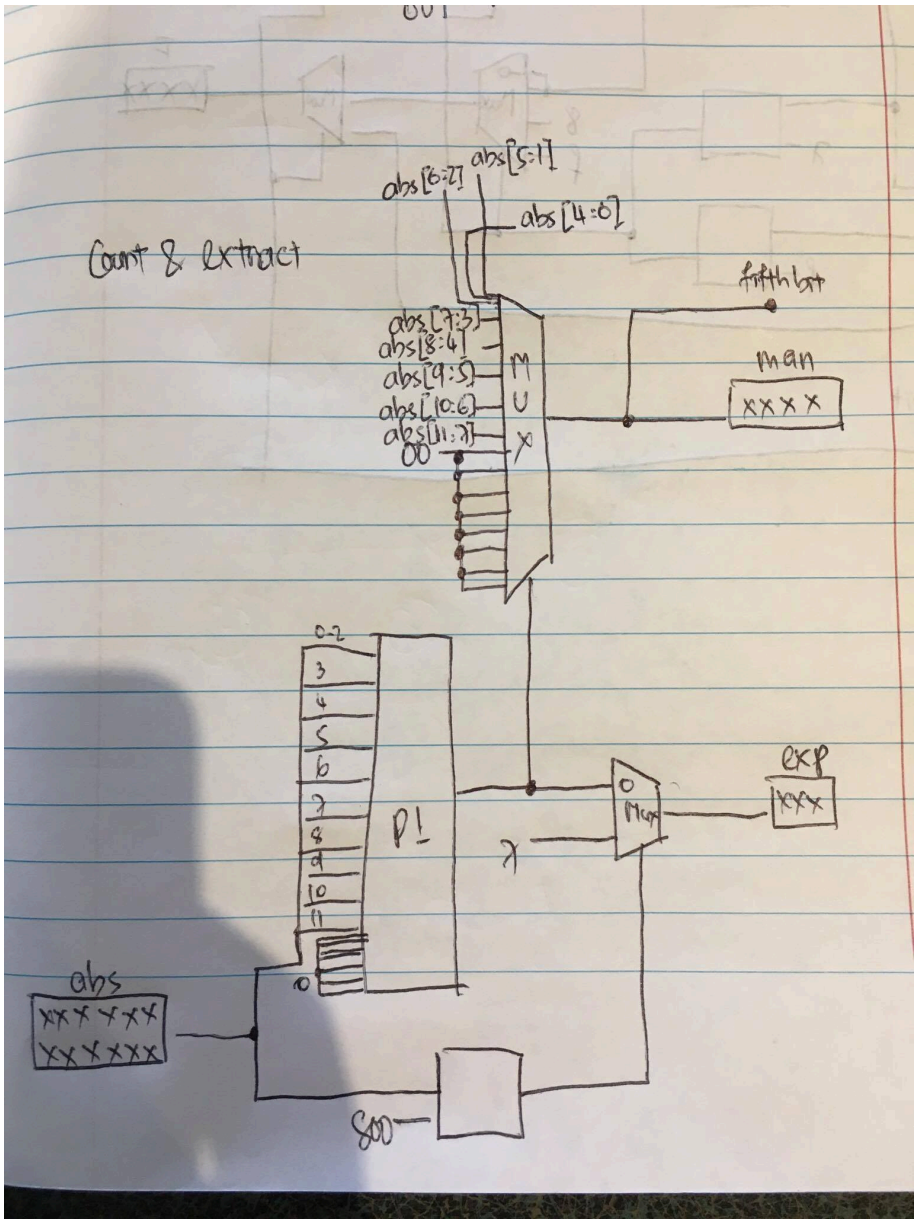


Figure 3: the logic design implementation for this module

It evaluates the temporary exponent and significand by finding the number of leading zeros.

For the significand, we found the spot for the leading zero and used the next four bits as the significand. The very next bit was the fifth bit, which we will use for identifying overflow in the next module. For the exponent, we used the following logic show in figure 3:

Leading Zeroes	Exponent
1	7
2	6
3	5
4	4
5	3
6	2
7	1
≥ 8	0

Figure 4: The number of leading zeros to the exponent.

We then outputted the fifth bit, temporary significand, and temporary exponent from this module.

Rounding

In this module, we used the output from the last module as input. Our output for this module are the signed bit : S, final significand: F, and final exponent: E.

From figure 5, we can see that if the fifth bit input is a 1, then we know there is going t be overflow so we prepare for rounding. We add one to the significand, and if it overflows, then add one to the exponent and right shift the significand to 1000. If both overflow, then output all 1's. However, if the fifth bit is a 0, then we do not need to round and set the input significand and exponent to the final significand and exponent.

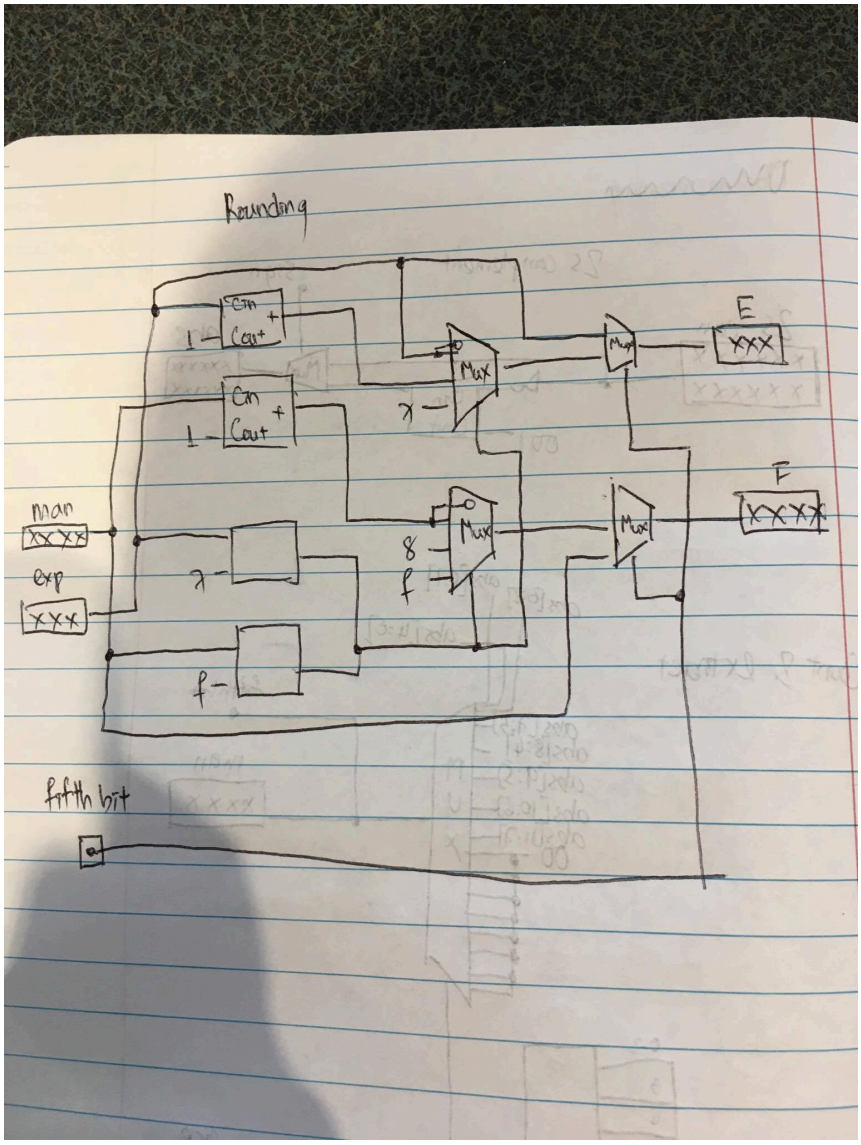


Figure 5: the logic design implementation for this module

Test Bench

Here are the test bench cases we tested. We wanted to make sure that our code took care of every case, and here are some examples:

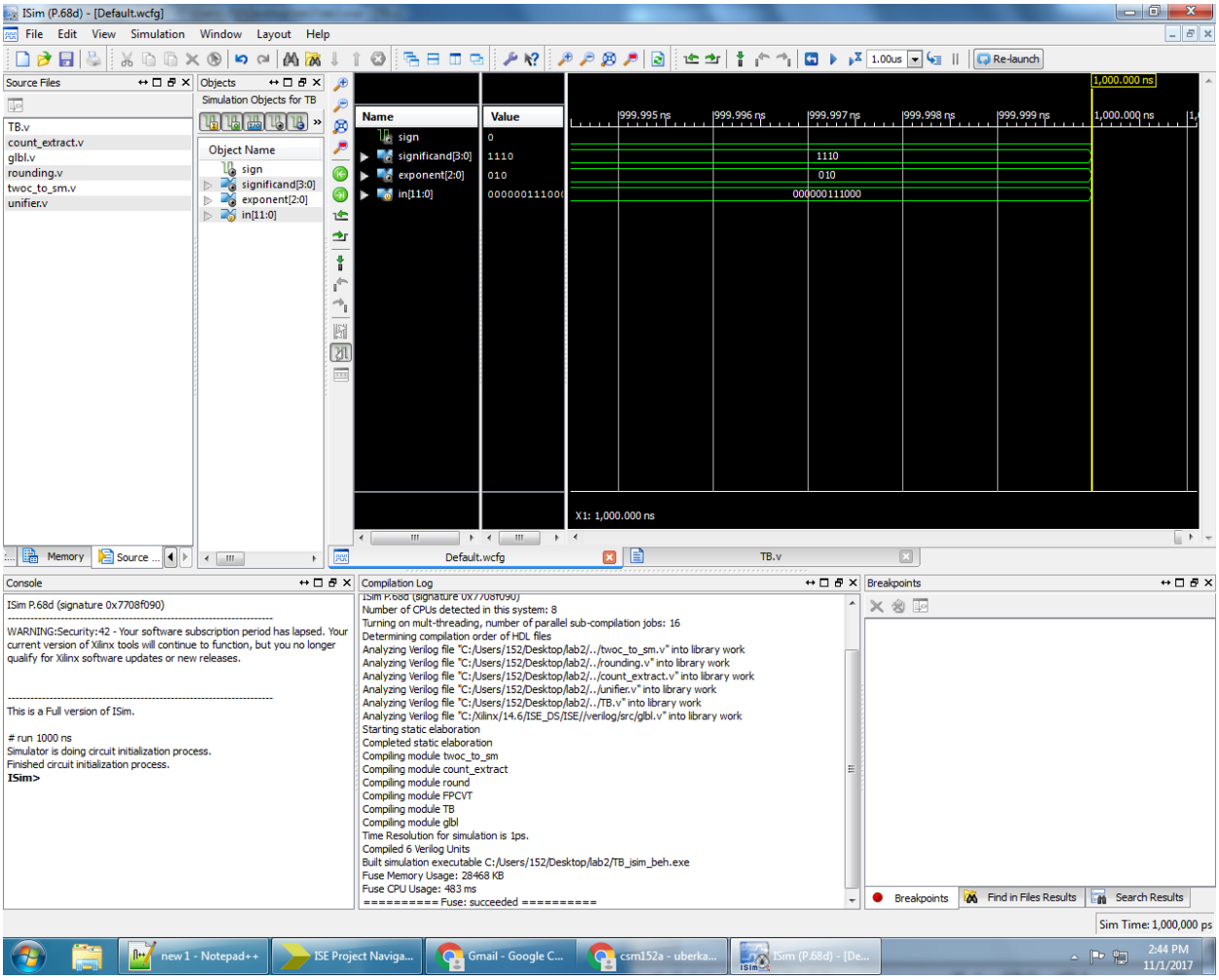


Figure 9: When inputting 56

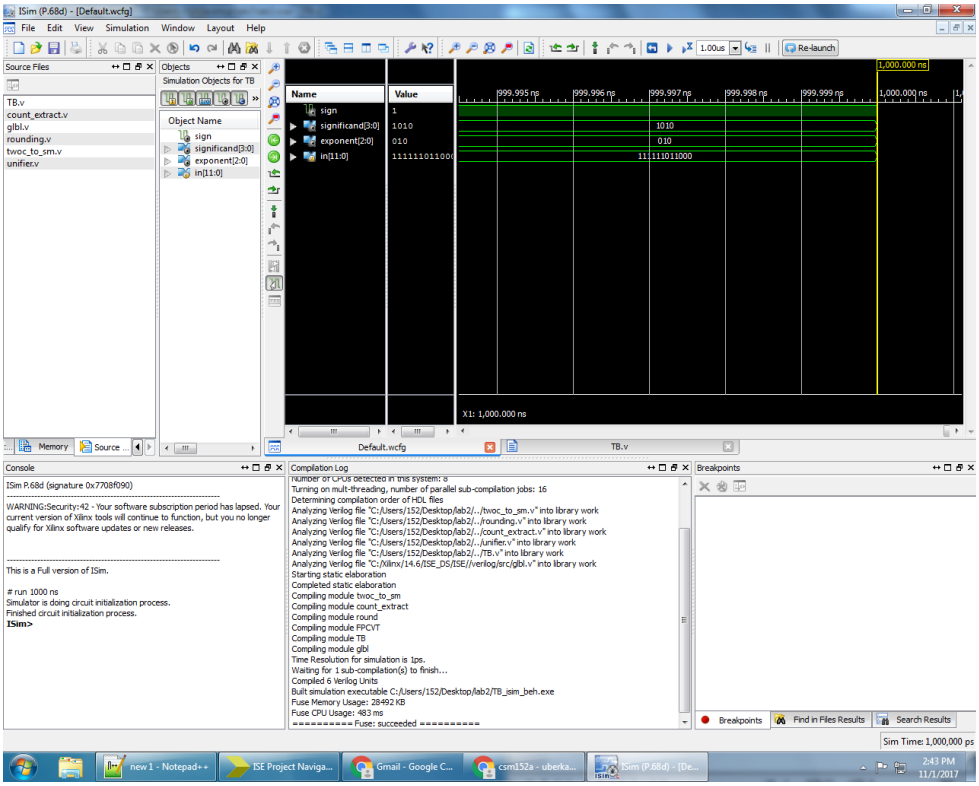


Figure 10: When inputting -40

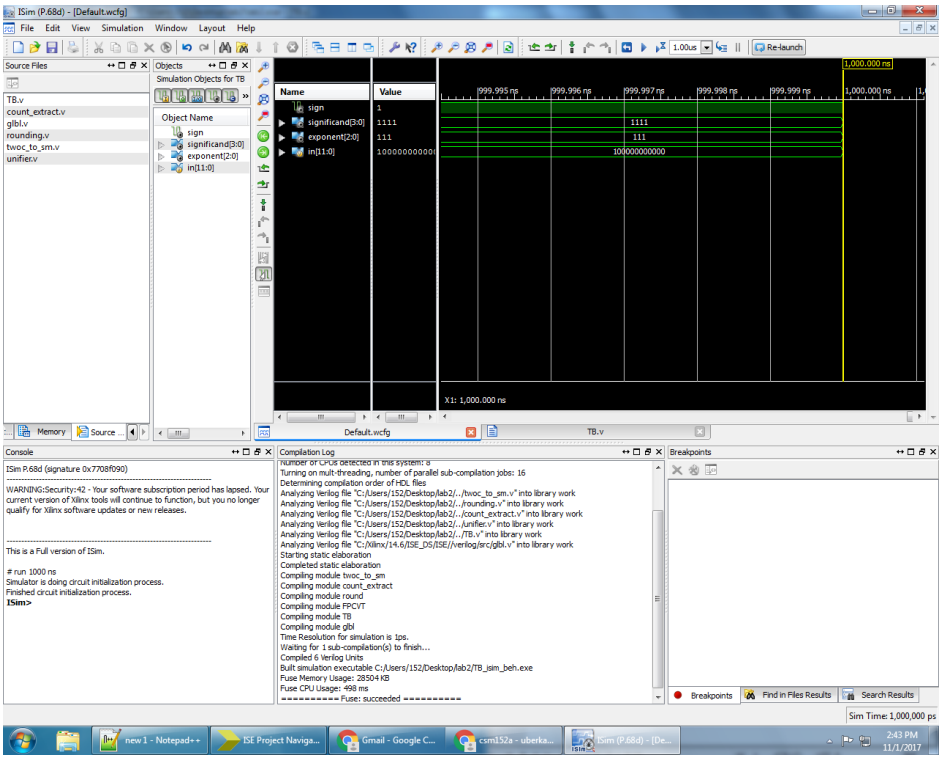


Figure 6: When inputting 12'b100000000000

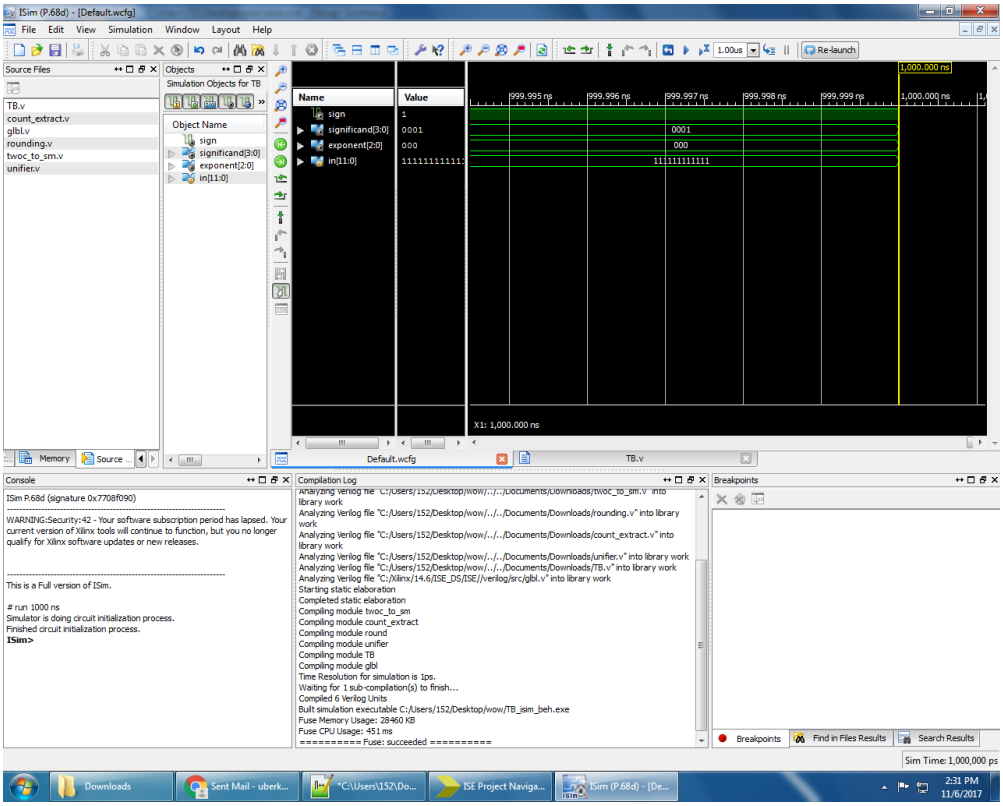


Figure 7: When inputting -1

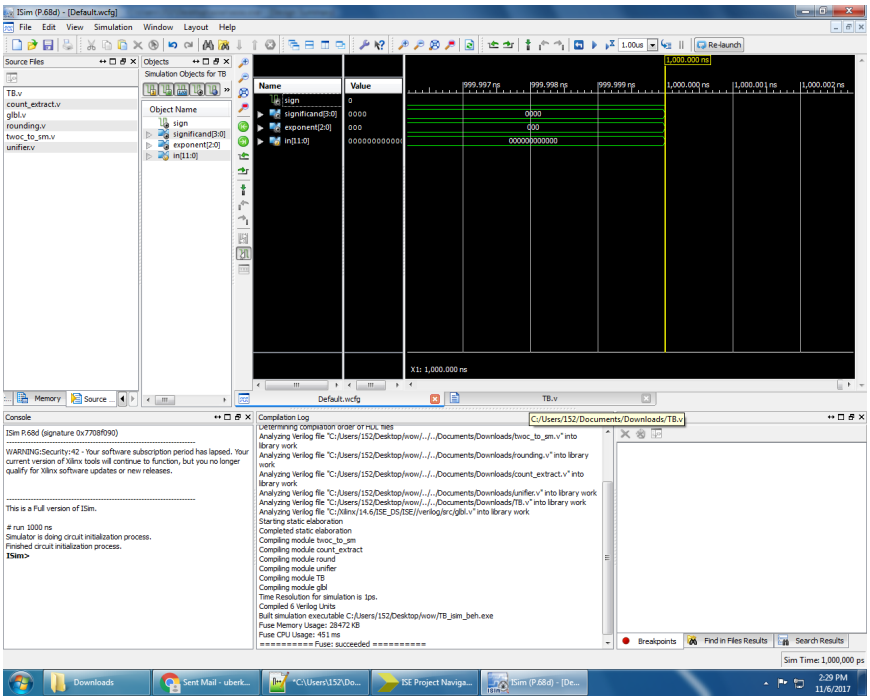


Figure 8: When inputting 0 base case.

We tested our code with a simple test case 12'b0 (Figure 8) to make sure we got the base case correctly. And it turned out to be correct. We then continued to test another simple case which is -1 (all 1), figure 7, which turned out to be correct. We then continued testing the numbers that were given in the lab (-40, 56) which are the normal cases. If this case is correct, the other numbers should be also correct. After taking a long time of coding and configuring it, we got the correct outputs. However, we did not realize there was still one error until we asked TA to test our code. When inputting 12'b100000000000 (Figure 6), he wanted it to be all 1s for S, E, and F. However, we somehow got a random value. Therefore, we had to modify our code in order to make it work properly with these case. Figure 6 shows this case works properly by outputting all 1s since it is an overflow case.

We tested all of these cases because:

Figure 8 is the base case, we must past this in order to do more complex things.

Figure 7 is another simple case with a negative number. Therefore, we tested it to see if converting negative number is correct or not.

Figure 6 is the part that we failed for the first test, and we modified it to take care of the overflow.

Inputting -40, 56 (Figure 9, 10) that were given in the lab which are the normal cases. Passing this case means passing other numbers.

Conclusion

From this lab, we gained valuable knowledge on how to link up modules and test things module-by-module so we know exactly where our error was when we made an error. Additionally, it helped us get more familiar with the Xilinx software and taught us how to use it better.