CSM152A - Lab 1 Report

Group Members:
Grant Roberts, 405-180-937
Ryan Brennan, 605-347-597
Yu Gao, 804-957-843

Section 5
TA: Kuo
Date Demoed: 1/16/20

1) Introduction:

Our objective for this lab was to create and test a combinational circuit that interprets a 12-bit two's complement representation of an integer and converts it to an 8-bit floating point representation. Below is an abstracted diagram of our module which shows D as our 12-bit input, and the floating point representation as composed of three parts: F, the 4-bit significand, or significant bits of our output, E, a 3-bit two's complement representation of the exponent F will be raised to and S, a single bit that will encode the sign of the significand.

Given the significand F, the 3 bit exponent E and S the sign bit we can represent a floating point number using the following formula:

$$Floating\ Point\ =\ (-1)^{s} * F\ * 2^{E}$$

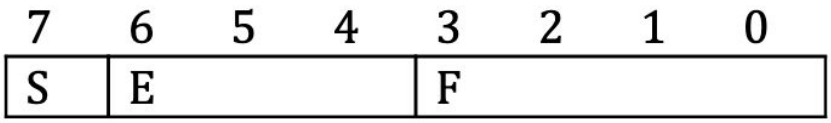

Figure 1: The 8 bit floating point representation

This lab only required simulation in Xilinx and did not require an implementation for the FPGA board.

2) Implementation:

We closely followed the block diagram from the lab specs as it seemed like a logical way to split up the different modules.

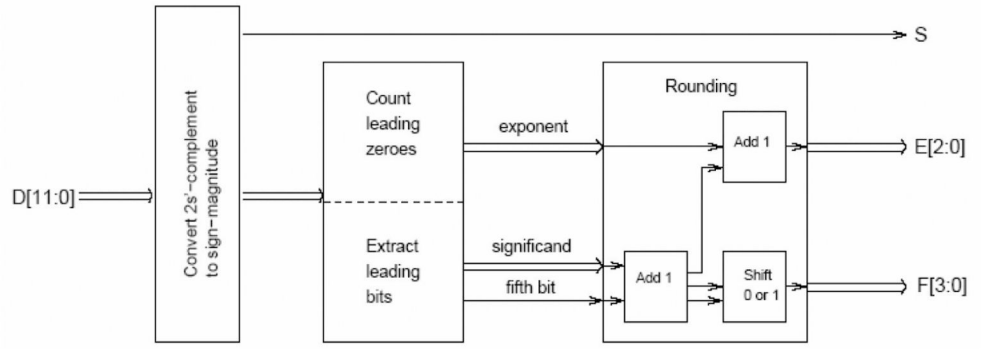

Figure 2: Overall block diagram for floating point conversion from lab 1 specs

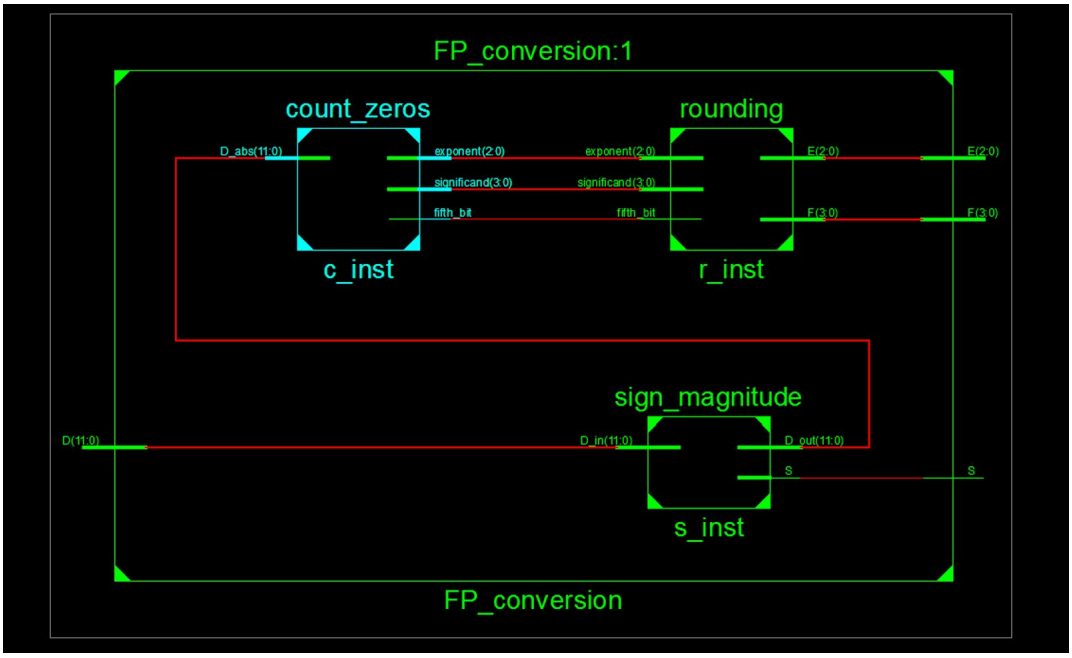

Figure 3: Screenshot from our synthesis schematic

2.1) Sign_magnitude module:

This module was responsible for converting the 12-bit two's compliment representation into sign-magnitude representation. This would be done by extracting the first bit which determines the sign in two's compliment. We would then take this bit and determine if the number was positive or negative. There are two possible outcomes for this. If the bit is determined to be positive, we keep the magnitude the same. If it is negative we negate it and add one. This can be seen in the following code snippet.

```
assign S = D_in[11];
assign D_out = (S==1'b1)? ~D_in + 1 : D_in;
```

Figure 4: Code snippet for assigning the sign bit

2.2) Count_zeros module:

This Module was responsible to count the number of leading zeros in the first 8 out of the 12 most significant bits. As shown in the chart below, where D7~D0 represents 8 most significant bits as input, and Q2~Q0 represents the number of leading zeros in binary. For example, when D7~D0 = 0000 0001, the corresponding output Q2~Q0 = 3'b111, which is 7 in decimal. The simplified logic expression for Q2~Q0 are shown below.

Priority Encoder

Input								Binary Output		
D7	D6	D5	D4	D3	D2	D1	D0	Q2	Q1	Q0
0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	1	x	1	1	0
0	0	0	0	0	1	x	x	1	0	1
0	0	0	0	1	x	x	x	1	0	0
0	0	0	1	x	x	x	x	0	1	1
0	0	1	x	x	x	x	x	0	1	0
0	1	x	x	x	x	x	x	0	0	1
1	x	x	x	x	x	x	x	0	0	0

The output counts how many zeros are there in the input

$$Q_0 = \sum(1,2,4,6)$$
$$Q_0 = (\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3\overline{D}_2\overline{D}_1D_0 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3D_2 + \overline{D}_7\overline{D}_6\overline{D}_5D_4 + \overline{D}_7D_6)$$
$$Q_0 = (\overline{D}_7\overline{D}_5\overline{D}_3\overline{D}_1D_0 + \overline{D}_7\overline{D}_5\overline{D}_3D_2 + \overline{D}_7\overline{D}_5D_4 + \overline{D}_7D_6)$$
$$Q_0 = (\overline{D}_7(\overline{D}_5(\overline{D}_3\overline{D}_1D_0 + \overline{D}_3D_2 + D_4) + D_6))$$

Similarly:

$$Q_1 = \sum(0,1,4,5)$$
$$Q_1 = (\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3\overline{D}_2\overline{D}_1D_0 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3\overline{D}_2D_1 + \overline{D}_7\overline{D}_6\overline{D}_5D_4 + \overline{D}_7\overline{D}_6D_5)$$
$$Q_1 = (\overline{D}_7\overline{D}_6\overline{D}_3\overline{D}_2D_0 + \overline{D}_7\overline{D}_6\overline{D}_3\overline{D}_2D_1 + \overline{D}_7\overline{D}_6D_4 + \overline{D}_7\overline{D}_6D_5)$$
$$Q_1 = (\overline{D}_7(\overline{D}_6(\overline{D}_3\overline{D}_2(D_0 + D_1) + D_4 + D_5)))$$

$$Q_2 = \sum(0,1,2,3)$$
$$Q_2 = (\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3\overline{D}_2\overline{D}_1D_0 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3\overline{D}_2D_1 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4\overline{D}_3D_2 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_3)$$
$$Q_2 = (\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_0 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_1 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_2 + \overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4D_3)$$
$$Q_2 = (\overline{D}_7\overline{D}_6\overline{D}_5\overline{D}_4(D_0 + D_1 + D_2 + D_3))$$

The exponent are calculated by subtracting Q[2:0] from 8. The condition is set for most negative number 12'b 1000 000 000. Since the two's complement is itself, and the sign bit is still 1, therefore in this case the exponent is expected to be all 1, which is 7 (3'b 111).

```
assign exponent = (D_abs[11] == 0)? 8-Q : 7;
```

The significand bits are assigned based on the location of leading zeros.

```
assign exponent = (Q == 3'b001) ? D_abs[10:7]:
                          (Q == 3'b010) ? D_abs[9:6] :
                          (Q == 3'b011) ? D_abs[8:5] :
                          (Q == 3'b100) ? D_abs[7:4] :
                          (Q == 3'b101) ? D_abs[6:3] :
                          (Q == 3'b110) ? D_abs[5:2] :
                          (Q == 3'b111) ? D_abs[4:1] :
                          (D_abs[11] == 1 )? 4'b1111:
                          D_abs[3:0];
```

The way we handle 12'b 1000 0000 0000 is adding a condition for sign bit, only the most negative number's sign bit is still 1 after two's complement. Therefore in this case the significand is expected to be all 1, which is 4'b1111 in the code.

The way to get fifth bit number is similar to the significand.

2.3) Rounding Module:

The rounding module is responsible to round up or down depends on the given fifth bit from count_zero module. There are 3 cases need to be considered, the first case is the fifth bit, either a 0 or 1; the second case is significant bits, either 4'b 1111 or not; the third case is exponent bits, either 3'b111 or not.

In the first case, if the fifth bit is 0, the module won't do anything, but simply output exponent and significand it received from count zero module. Otherwise, it goes to the second case.

In the second case, if the significand bits are not 4'b 1111, which means rounding up doesn't cause overflow, significand need to increase by 1. Otherwise it goes to the third case.

In the third case, if the exponent bits are not 3'b 111, which means rounding up doesn't cause overflow, exponent need to increase by 1 and significand becomes 4'b 1000. Otherwise change all the bits of exponent and significand to 1.

```
if (fifth_bit == 0)
            begin
                E <= exponent;
                F <= significand;
            end

            else
            begin

                if (significand == 4'b1111)
                begin

                    if (exponent == 3'b111)
                    begin
                        E <= exponent;
                        F <= significand;
                    end

                    else
                    begin
                        E <= exponent + 1;
                        F <= 4'b1000;
                    end
                end

                else
                begin
                    E <= exponent;
                    F <= significand + 1;
                end
            end
 end
```

2.4) FP_Conversion:

The top module is responsible for organizing the sign_magnitude, count_zeros and rounding modules. It organizes them in a way such that the inputs from one are the outputs from another so they flow logically. It was interesting to note how quickly this started to resemble object oriented programming but in a hardware language. This can be seen in the following code snippet as we call each of the three modules.

```
sign_magnitude s_inst( .D_in(D),
                       .D_out(outB_to_inC),
                                                    .S(S));

// exponent, significand, fifth_bit
wire [2:0] outC_to_inD_0;
wire [3:0] outC_to_inD_1;
wire outC_to_inD_2;
count_zeros c_inst( .D_abs(outB_to_inC),
                                                .exponent(outC_to_inD_0),
                                                .significand(outC_to_inD_1),
                                                .fifth_bit(outC_to_inD_2));

rounding r_inst(.exponent(outC_to_inD_0),
                                          .significand(outC_to_inD_1),
                                          .fifth_bit(outC_to_inD_2),
                                          .F(F),
                                          .E(E));
```

Figure 5: Code snippet from FP_conversion module showing how each of the other 3 modules are called and outputs passed as inputs

3) Testing

The testing of our program was rather simple. We simply ran through all possible inputs and checked for consistency in the outputs in FP_conversion to make sure that our entire program was working together. We did this by declaring a 12 bit register D and initializing it to 0 at first. We then ran a forever statement adding one to D each time. This is shown in the following code snippet.

```
forever #100 D = D + 1;
```

Figure 6: Code snippet for our testing analysis

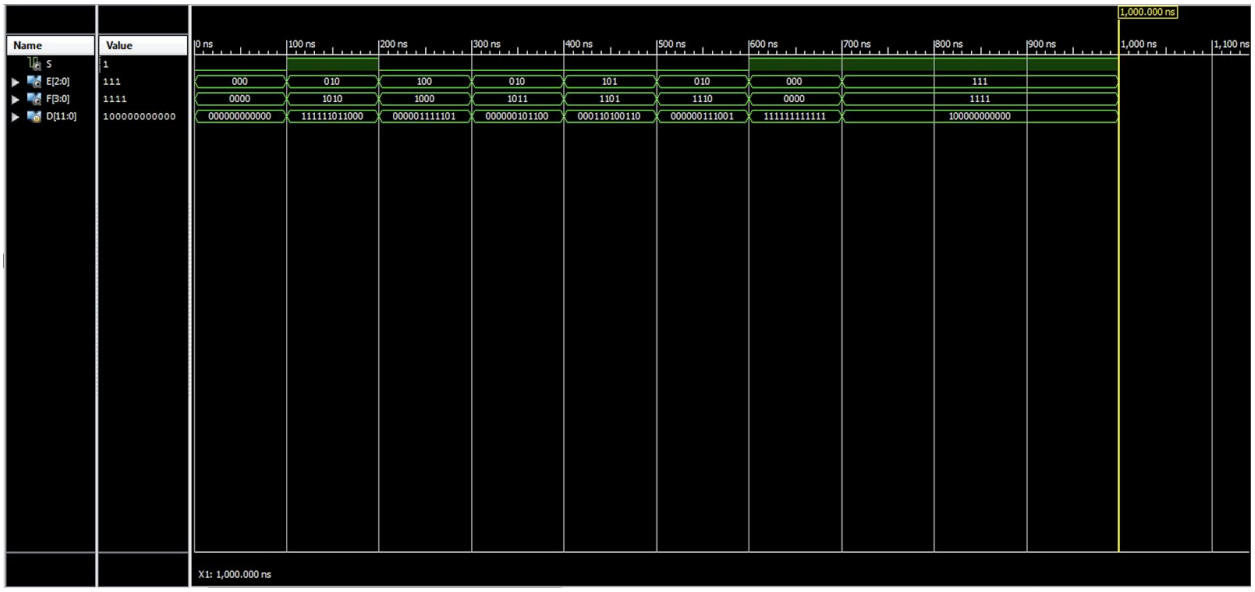

Figure 7: Screenshot of the output for the testbench

We examined some outputs more closely than others. For the cases where no rounding was required we largely glanced at the first few and then moved onto the cases where rounding happened and really made sure the output was correct. Our TA provided us with the idea to loop through and check all cases starting from 0, as we originally were only checking some tricky edge cases. Being able to see every possible input in a row really helped for sanity checking as we were just able to scroll left and examine each output.

We also had a test bench for each of our modules

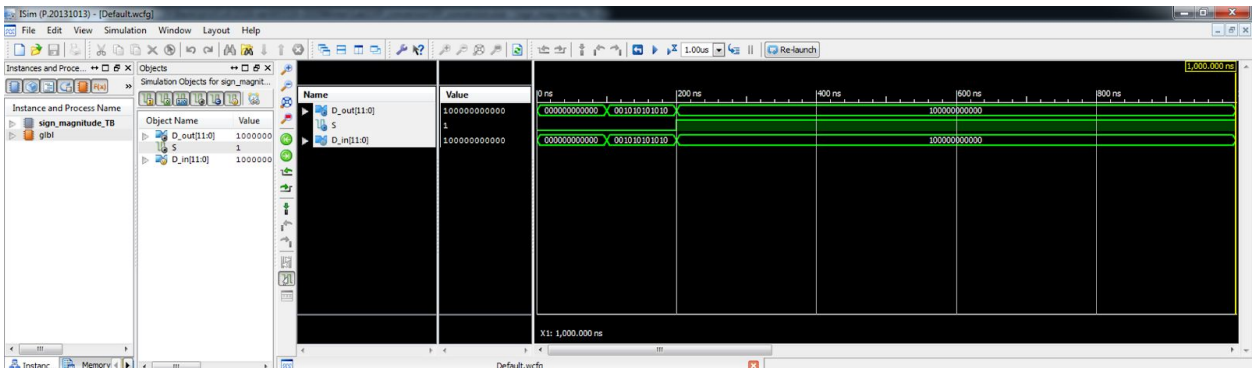

Figure 8: Screenshot of the output for the testbench of sign_magnitude

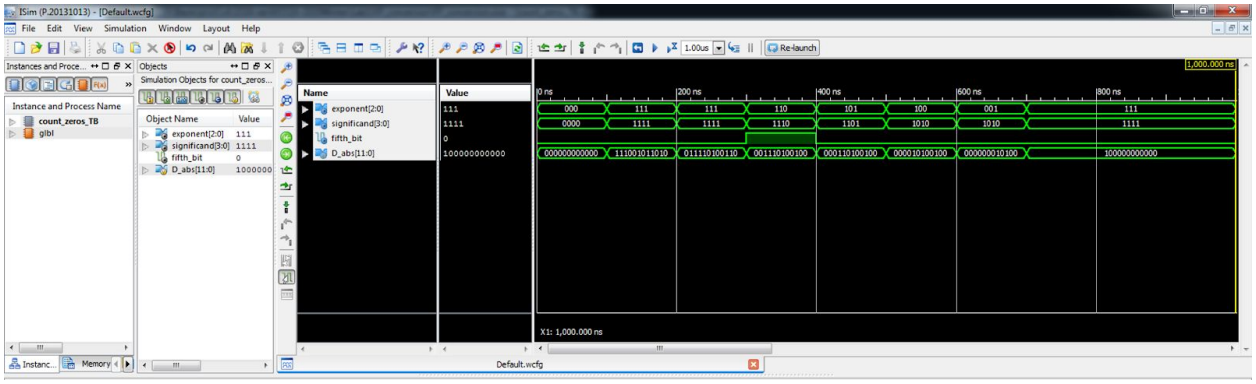

Figure 9: Screenshot of the output for the testbench of count zeros

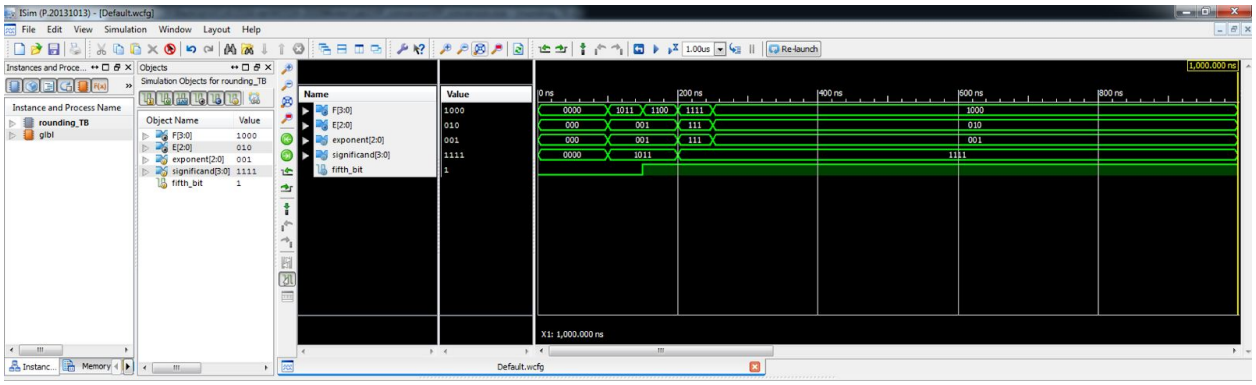

Figure 10: Screenshot of the output for the testbench of rounding module

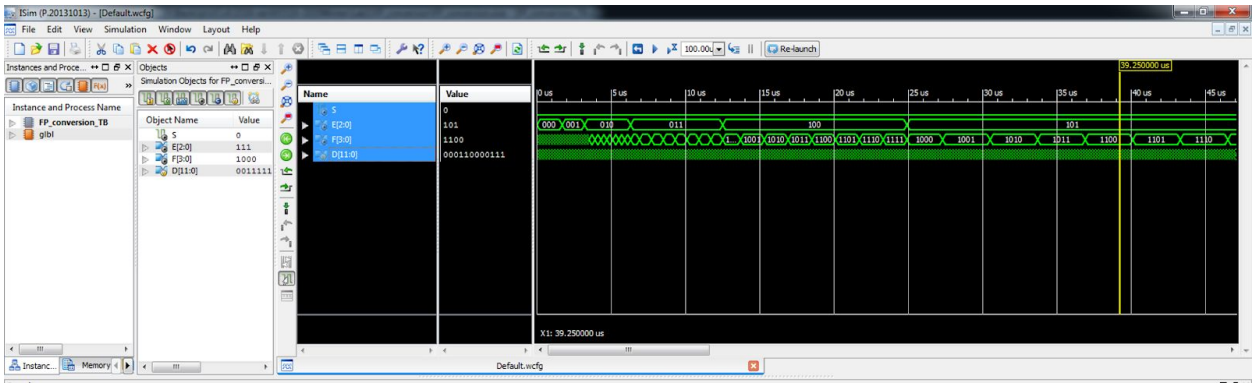

Figure 11: Screenshot of the output for the testbench of the final result

4) Conclusion

This was a great introduction lab to get us writing our own code and modules in Verilog using Xilinx. As lab0 was more of a walkthrough it was interesting to attack a problem that was more open ended on implementation. The schematic provided in the lab specs was a great starting point and helped us divide our modules up logically. As was mentioned previously, by modularizing our implementation, it really helped to abstract pieces of our converter so we could write seperate test benches and check that each piece was working properly.

As this was the first lab that we implemented start to finish, there were a few issues we ran into. We originally struggled with where to start and how to get writing code. We initially went back and did

some hand encoding of floating point numbers to be sure logically we had the right idea and then we started to implement the sign and magnitude module. By starting off on just a piece of the overall converter it really helped to get going on the implementation. It was also a new process of writing the test benches. For the final test bench of the converter we didn't think to check all the inputs and their outputs. We were really focusing on edge cases that we knew could cause issues. Instead, as was mentioned in the test bench section, our TA gave us the advice of looping through all inputs and checking outputs systematically, this helped us greatly in making sure our implementation was correct.