Team Multiplexers

Alexander Graening (804732740), Xilai Zhang (804796478), Joseph Miller (504744848)

**Lab Report 2 (Team Multiplexers)**

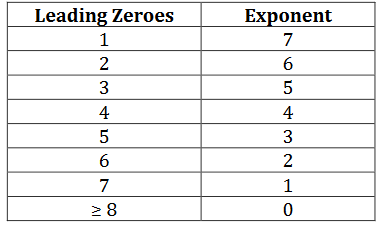
**Section 1: Design Description**

**Overall Description:**

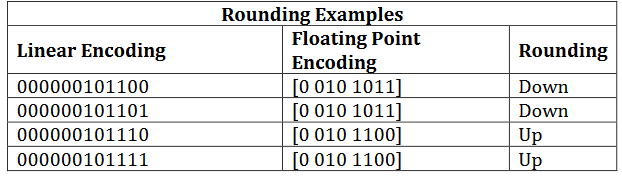
For lab 2, we were tasked with designing a combinational logic circuit to convert a 12-bit, two’s complement signal input to its nearest 8-bit floating point approximation. The format of our floating point numbers is 1-bit allocated to the sign, 3 bits allocated to the exponent and 4-bits allocated to the significand. To determine the value of the exponent to use, we convert any negative number into its sign magnitude equivalent and count the number of leading zeros. We then use the table in figure 2 to determine what value to allocate to the exponent E. After the leading zeros have been counted, we extract the significand F from the 4-bits following the last leading zero. Finally, we examine the fifth bit from the last leading zero to determine whether or not to round the significand up or down as shown in figure 3.



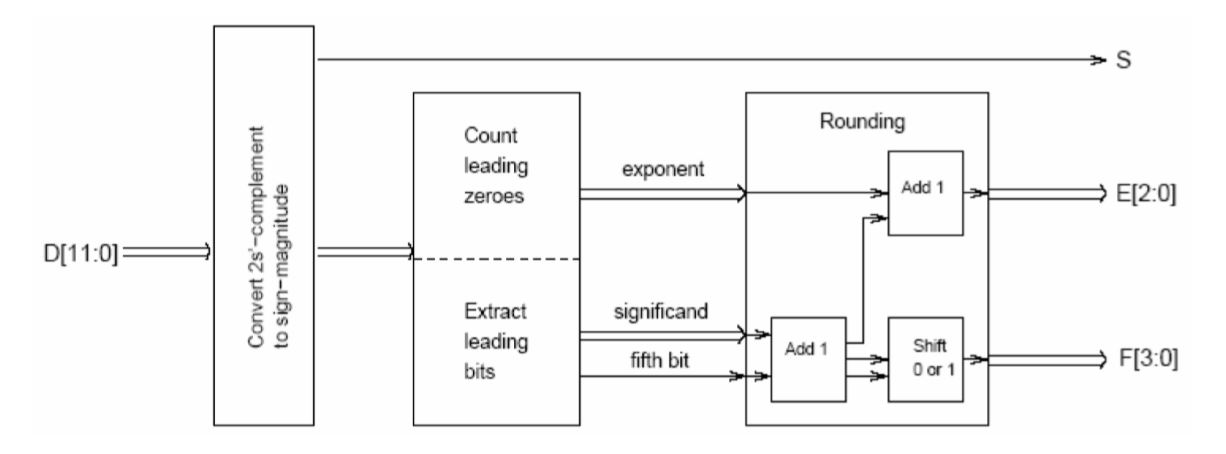
**Figure 1:** An 8-bit Floating Point Representation. S represents the 1-bit sign value, E represents the 3-bit exponent, and represents the 4-bit significand. The value of the floating point number can be calculated using V = (-1)S \* F \* 2E.



**Figure 2:** Table Used to Convert Number of Leading Zeros to Value of Exponent.

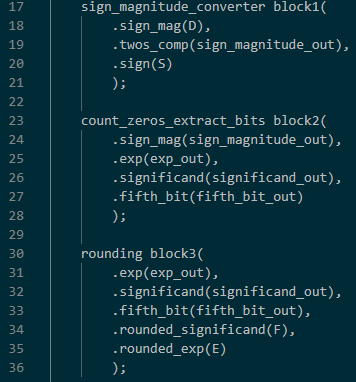


**Figure 3:** Examples for Determining Rounding of Significand Based on Fifth Bit. Since it is zero in the first two columns, we round down for those. Since it is 1 for the last two, they are rounded up instead.



**Figure 4:** Overall Schematic for Implementation of Floating Point Converter. (This figure is directly from the assignment.) We take in a 12 bit input D which is first fed into the module for converting 2’s complement to sign magnitude. The output from this module is then fed into a new module that counts leading zeros and extracts the significand bits. Finally, a rounding module is fed in the exponent, significand and fifth bit from the prior module and rounds the value for the significand accordingly.

**FPCVT (Top Module):**



**Figure 5:** Code From the Main FPCVT Module. The code shown here connects the inputs and outputs for our modules necessary to create the floating point converter. We used three main blocks to accomplish this task: a sign magnitude converter, a block that counts and extracts leading zeros, and a block that handles rounding.

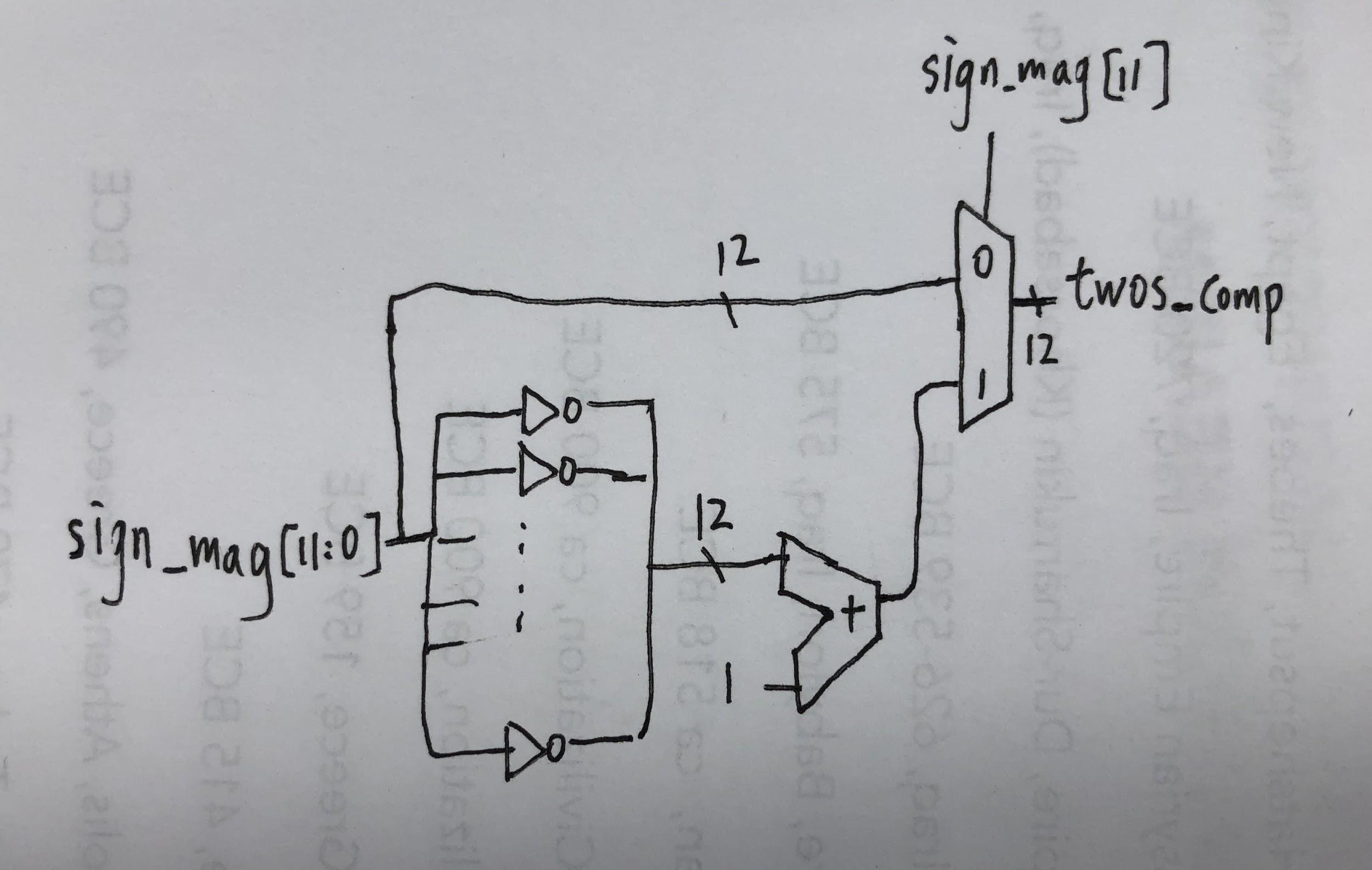
The main idea behind this module is to link the diagram found in figure 4 to verilog modules. We handle wiring up all the necessary inputs and outputs to the three main modules here. We have as input to this module D, the 12-bit 2’s complement signal we wish to convert to floating point. For output we have S (the sign bit), E (the 3-bit exponent) and F (the 4-bit significand). We also have 4 wires used to hook the modules together which include: sign\_magnitude\_out (our 12 bit converted sign magnitude signal), exp\_out (our temporary exponent value pre-rounding), significand\_out (our temporary significand value pre-rounding), and fifth\_bit\_out (the fifth bit).

**Sign Magnitude Conversion Module:**

The general idea behind this module is to convert an input two’s complement signal into a sign magnitude output, as well as output the sign bit S for the final output. To convert to sign magnitude, all that is needed is for the signal to be inverted and incremented by one. We are only to convert if the original signal is negative, so we check if the sign bit of the original signal is 1 to determine whether or not to convert.



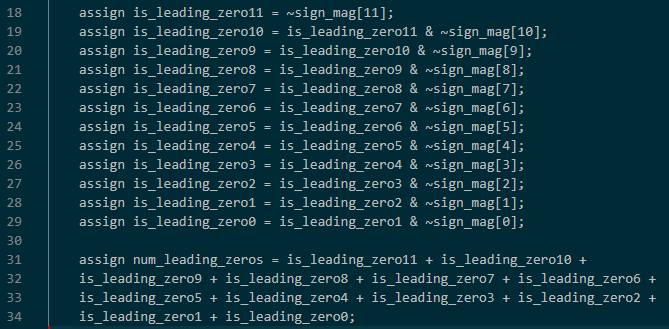
**Figure 6:** Code Implementing Sign Magnitude Conversion. This ternary condition operation boils down to if the leading bit of the input two’s complement signal is 1, convert it to sign magnitude. (Note: The confusing naming is a result of a naming error on our part where the inputs and outputs to this module were swapped. Twos\_comp refers to the sign magnitude output and sign\_mag refers to the original two’s complement signal.)



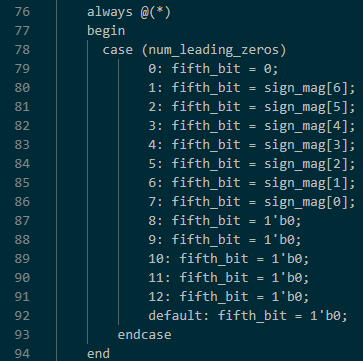
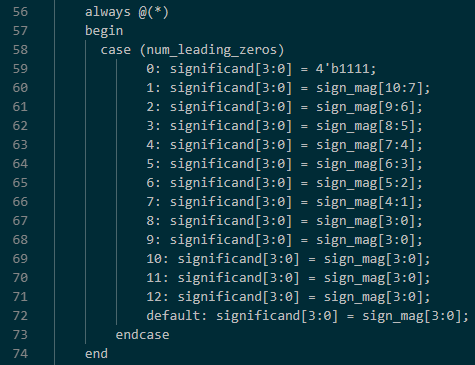
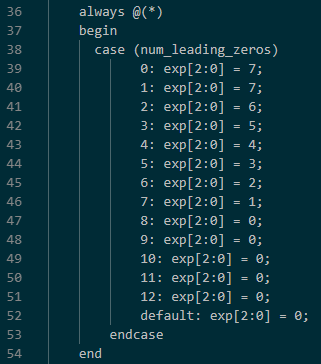
**Figure 7:** Sign Magnitude Conversion. If the number is positive, simply switch it to the output. Else, invert the bits and add one, before switching it to the output.

**Count Zeros Extract Bits Module:**

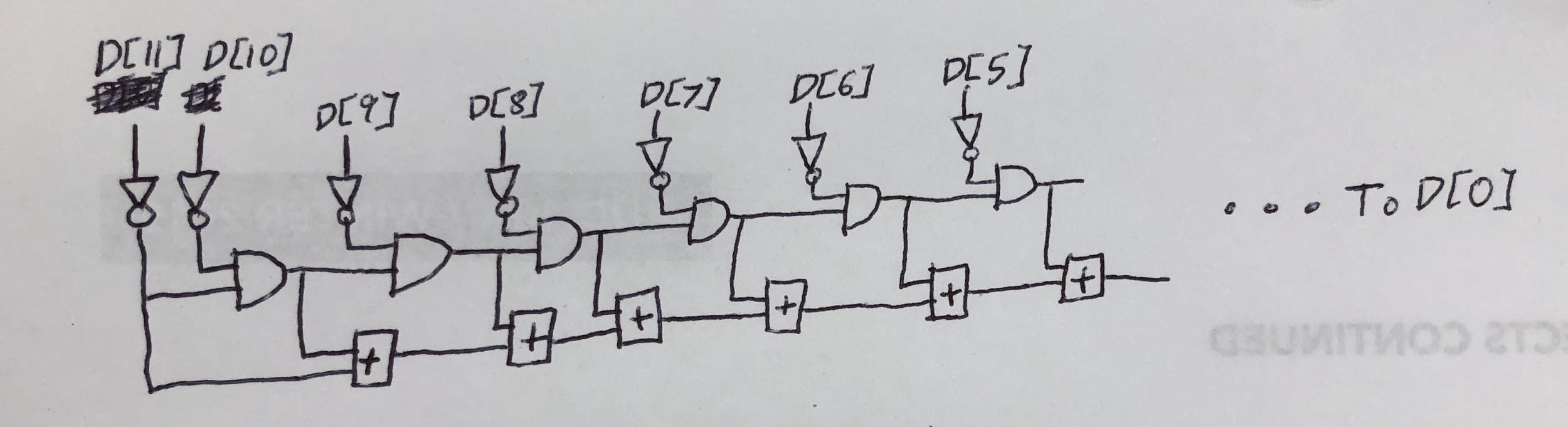
The main idea behind this block is determining how many leading zeros we have on the input signal, and using this to map what the fifth\_bit, significand, and exponent should be before rounding is done. The schematic in figure 10 is the best way to understand our design, but here is an attempt to describe our design in words. To count number of leading bits, take the inverse of first bit and name it is\_leading\_zero11. Now if we AND the inverse of the second bit with is\_leading\_zero11 we will get is\_leading\_zero10, which tells us whether the first two bits are 0. The pattern is as follow: for the kth bit, we AND the inverse of the kth bit with is\_leading\_zero(k+1) to generate is\_leading\_zero(k), which tells us whether the first bit until the kth bit are all zeros. Now we add is\_leading\_zero(k) for k from 11 to 0 to generate num\_leading\_zeros, which tells us the number of leading zeros. Now that we have this, the value of exponent is defined by num\_leading\_zeros directly, with assignment based on the chart in figure 2. Significand[3:0] is defined as sign\_mag[11- num\_leading\_zeros:8 - num\_leading\_zeros] when num\_leading\_zeros is smaller than 8. In all other situations, the significand is defined as the last four bits. The fifth bit is assigned in a similar manner with sign\_mag[7 - num\_leading\_zeros] when num\_leading\_zeros is between 1 and 7 and 0 otherwise.



**Figure 8:** Verilog code handling counting of leading zeros. Logic is described above.



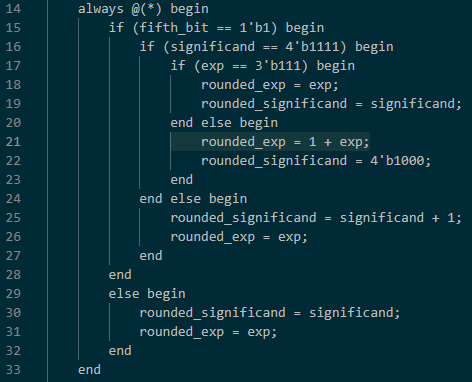
**Figure 9:** Statements Implementing Exponent Value Assignment Based on Chart in Figure 3 (Left), Value for Significand (Center), and Value for Fifth Bit (Right).



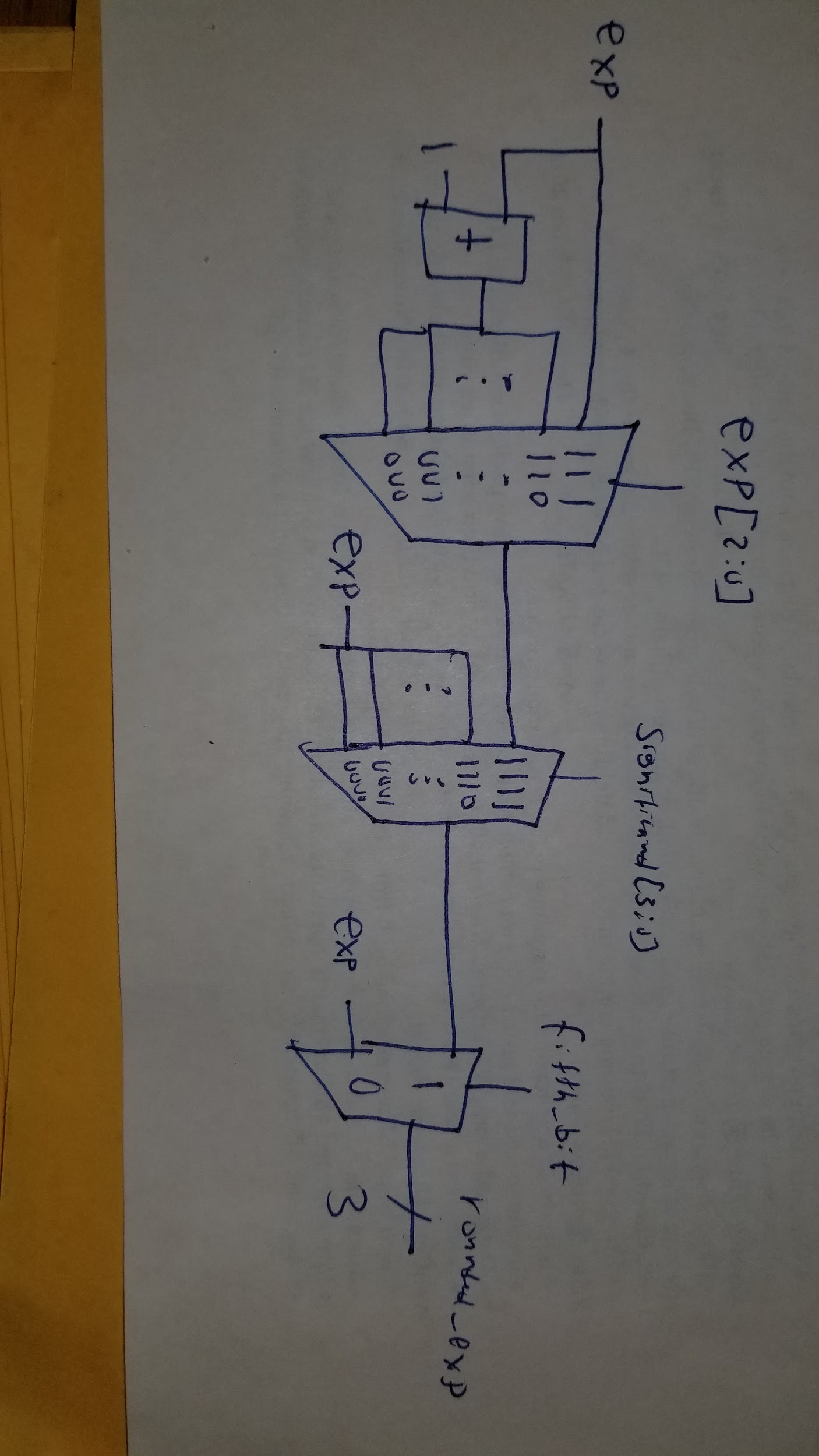
**Figure 10:** Combinational Logic Used to Count Number of Leading Zeros. Each of the 12 bits from the sign magnitude version of the original two’s complement input are used as inputs in this block. Note that the outputs of all the and gates after the first 1 will be 0, so the final sum will be the number of zeros. The size of the adders has been omitted in this diagram for clarity although the size of the adders would change slightly as they are connected from left to right and the maximum sum requires 4 bits.

**Rounding Module:**

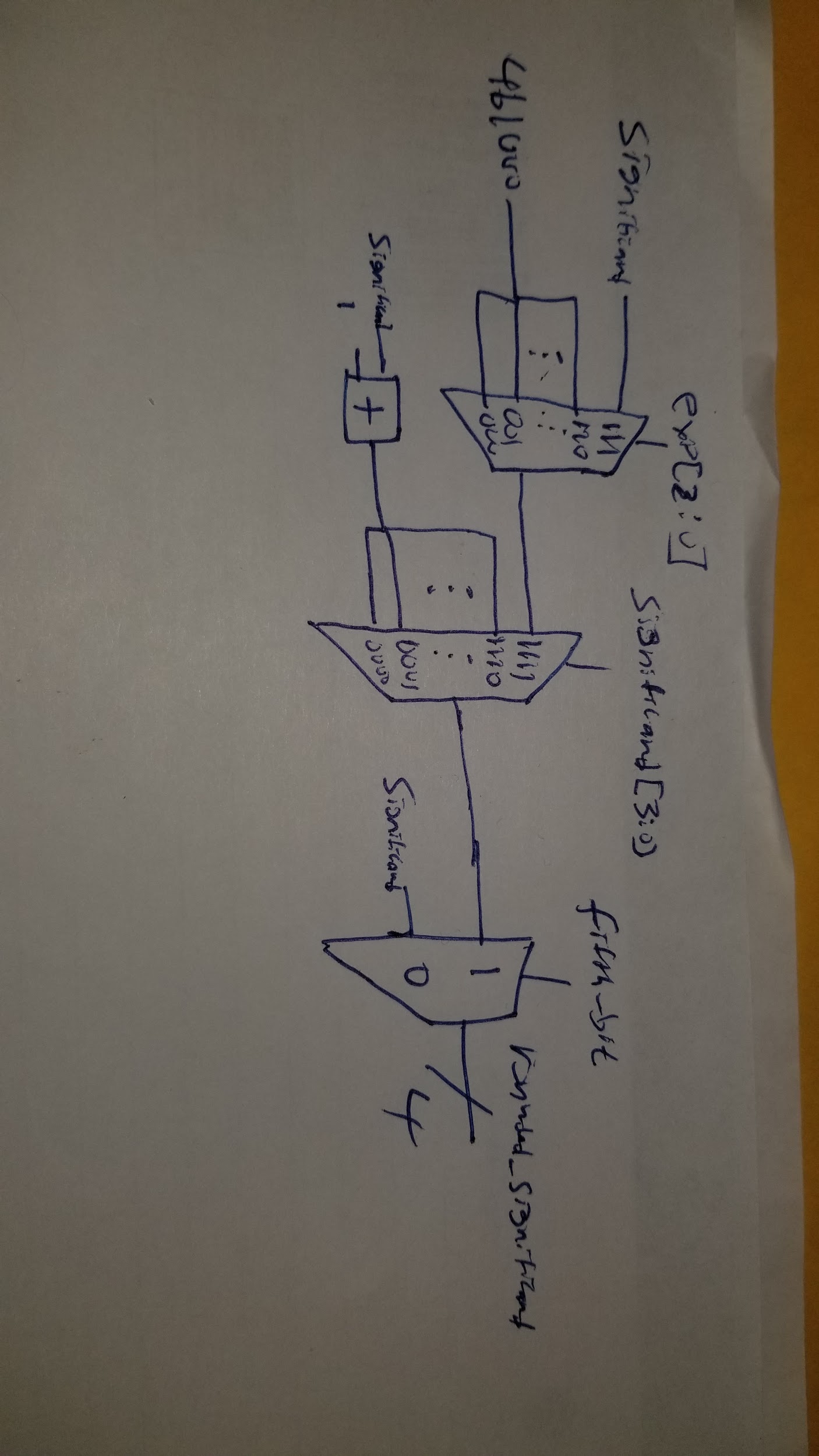
The main idea behind our design for the rounding module is case analysis. We round the significant bit up when the fifth bit is 1 and down when 0. We also run into the following two issues when rounding up. First, if the significand is the maximum 4’b1111, and if exponent is also the maximum 3’b111, we cannot raise the exponent to compensate for the decrease in significand, so we keep the number as large as possible, namely, do nothing. Second, if the significand is 4’b1111 and exponent has not yet reached its maximum, we cannot set the significand to 5’b10000, but we can do something similar by making it a 4’b1000 and adding a 1 to the exponent acting as a left shift. Otherwise, rounding acts normally and since the significand has not reached the maximum of 4’b1111, so we simply increase it by 1. If the fifth bit is 0, we round down, which involves doing nothing since the significand we received was already the rounded down version (all bits past the 5 we received are zero since we don’t have them).



**Figure 11:** Code Implementing Rounding Logic in Verilog. We choose to handle edge cases and rounding based on a series of if statements.



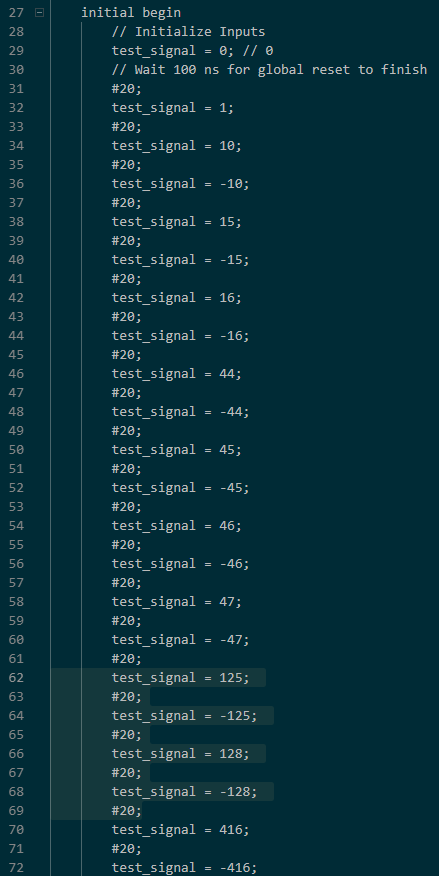
**Figure 12:** Diagram Describing How to Calculate the Exponent with Combinational Logic. In this, if the fifth bit is zero we get the exponent rounded down. If the fifth bit is one and significand is not all ones, we get the exponent left the same as it was when it came in. If the significand is all ones and fifth bit is one, the signal selected will be 111 if exponent is already all ones or exp+1 if not.

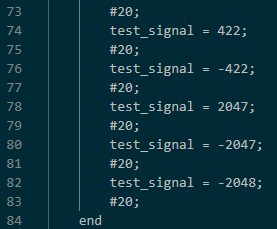


**Figure 13:** Diagram Describing How to Calculate the Significand With Combinational Logic. In this, if the fifth bit is zero, no rounding is performed so the significand passed through. If the fifth bit is 1 and the significand is not 1111, the significand is rounded up so it gets a one added to it. If the fifth bit is one and the significand is 1111, then it is set to 1000 if the exponent is not 111 and 1111 if exponent is 111.

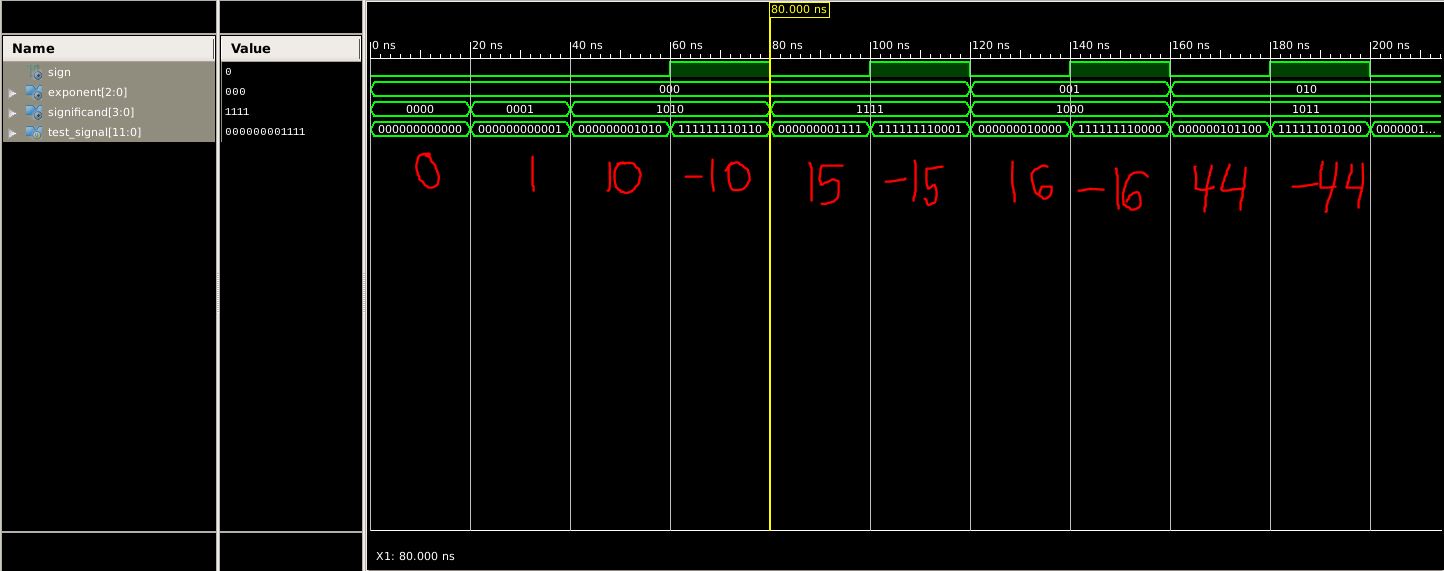
**Section 2: Simulation Documentation**

The main testing methodology we had when it came to our testbench was to ensure that not only did the design work for typical cases, but also for the edge cases as well. We needed to check to make sure the output waveform matched the floating point value we expected it to go to according to the specifications laid out in the lab manual. We knew of the important edge case of -2048 which would give the same value when converted to two's complement. We treated this case as special in the counting zeros module by including a case where there weren’t any leading zeros. This allowed us to set this value appropriately to the maximum possible integer available in our floating point converter. Other important numbers to test included positive values, negative values, values that set significand to 1000 and incremented the exponent, values large enough to set the value to max int and values that should be rounded down. Once we came up with our set of numbers that satisfied one or more of the above test cases, we wrote them up into a testbench and verified the binary output matched what would be expected from our values. Once we ran this testbench, it was quickly obvious that something was very wrong: The exponents and significands were not correctly set. This led to a quick reexamining of the code to determine what bug could be causing this. Since the value for exponent didn’t seem to match what should be expected from the number of leading zeros, the rounding module seemed like a likely culprit for the mistake. Upon reexamining this module, it became clear the issue was counting the leading zeros. We forgot one of the bits to be summed on line 31. This caused our value for num\_leading\_zeros to be wrong as well as all values that depended on it. With this quick fix, the testbench passed all cases and we felt confident our design was correct

.



**Figure 14:** Code for testbench including all numbers tested for our testbench. We used an idea of positive values followed by negative values to quickly check negative cases as the waveform would only differ by the sign bit.



  
**Figure 15:** Output Waveforms Generated From Running the Testbench. This shows some of the largest and smallest magnitude values we used to test along with all the edge cases we tested. We wrote in which binary numbers correspond to their decimal equivalents for easier reading. We verified each of our test cases by hand to ensure there were no mistakes in the code.

**Section 3: Conclusion**

The lab was a success. We managed to properly test and implement the floating point converter well within the time we were allotted and passed the golden testbench on the first run through. We learned a lot about creating our own verilog code from scratch and how to properly modularize our verilog code to make the overall design easier to follow and modify. We also discovered the importance of testbenches and how to properly examine each possible edgecase that can arise in a given project.