Name: Chris Baker
UID: 105.180.929

CS M152: Project 2

Floating Point Conversion

Intro

Our world generally operates in analog and continuous signals while computers run in binary (usually discrete). Because of this difference we need to be able to convert signals between those two categories of data. Thus, digital to analog converters (DAC) and analog to digital converters (ADC) are used to either store or read data. In our lab we focus on floating point numbers because they can represent values in between integers, and they can scale to large ranges with the same number of bits. Because of these advantages it can be useful to implement floating point for commercial industries like speech recognition and music editing. Due to this advantage floating point can better distinguish between loud and soft sounds since the average value between its representable numbers is much smaller than that of integers. Therefore, understanding how to perform a transition from linear encoding to nonlinear encodings is an important capability for programmers.

We will implement this lab through an overall module that accepts a 13 bits Two's Complement vector and returns a 9-bit Floating Point representation for that number. We will accomplish this by accepting the Two's Complement bus (D) and convert it to Sign Magnitude form. From the Sign Magnitude we will extract the original Sign Bit and store that for our final Floating-Point Result. Additionally, we handle the special case for -1 before sending our Signed Magnitude number to the Priority Encoder. In the Priority Encoder the 5-bit mantissa, 3 bit exponent, and sixth bit (after leading zeros) will be extracted. Unfortunately, these values are

prone to possible errors from overflow and rounding, so we forward these results into the rounder to handle more special cases. In the rounder we will handle cases where the mantissa may be fully set while the exponent field may be fully set and other such variations. Upon handling possible sources of error, we return the results of a 9-bit Floating Point number. The S represents our sign bit, E represents our 3-bit exponent field, and F represents our 5-bit mantissa.

Design Description

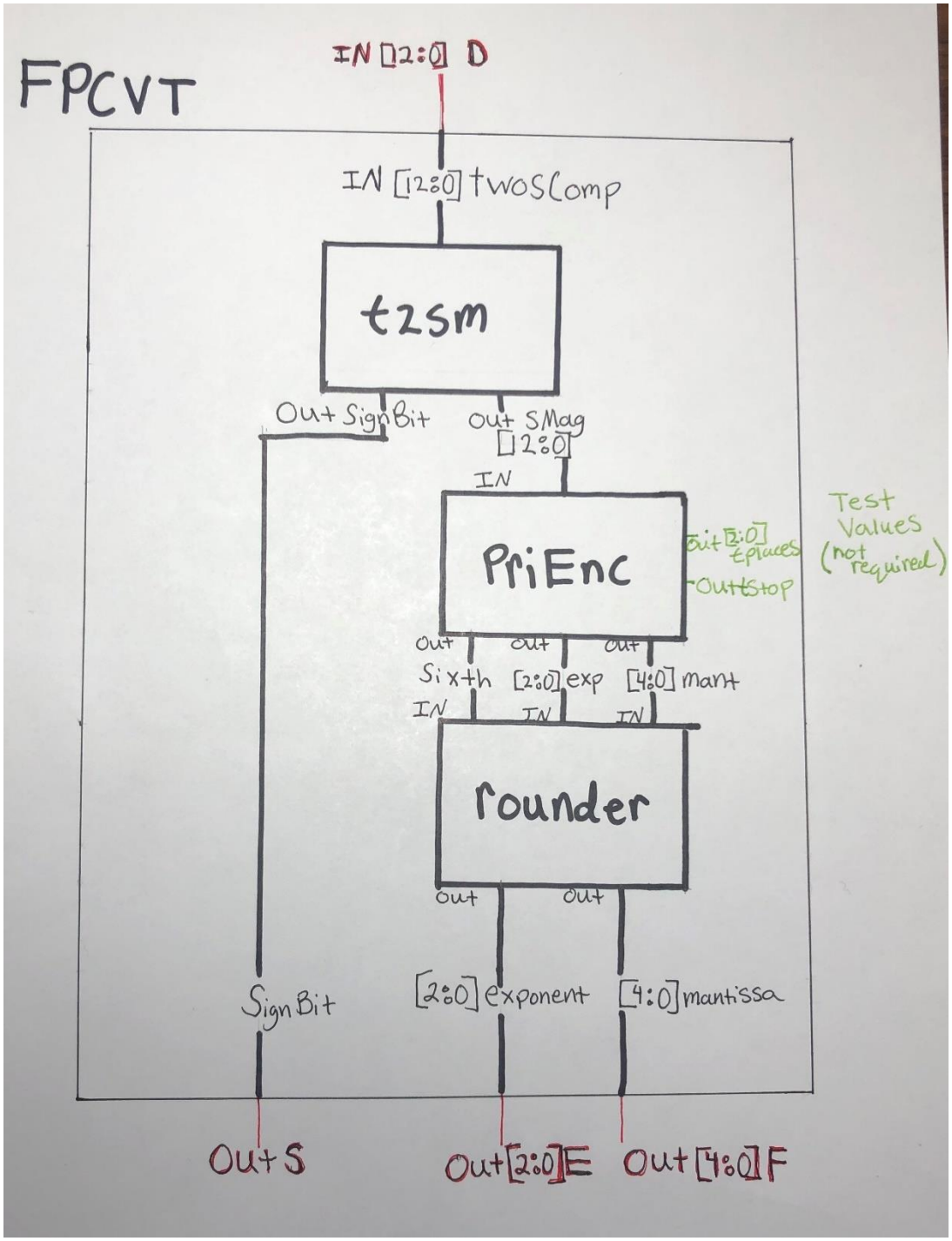
The design for my Floating-Point Conversion process is described through an overarching block that represents the whole module (FPCVT). Contained within the modules are three submodules which perform their respective tasks in taking a 13 bit Twos Complement Number (D) and converting it into a nonlinear Floating Point Number with a sign bit, 3 bit exponent field, and 5 bit mantissa field (S, E, F).

Below is a schematic which demonstrates the top-level module (FPCVT) and the Two's Complement Input (D) and three corresponding Floating-Point Outputs (S, E, F).

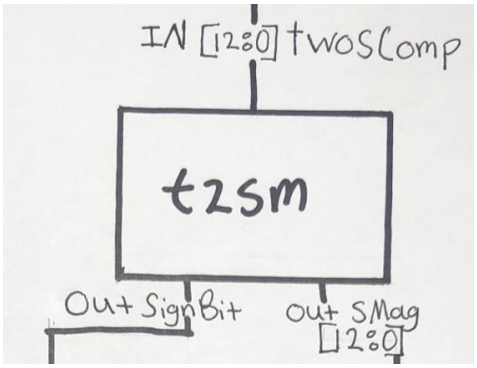
Please note the following schematic also contains two test values I chose in incorporate in this design as I used them check that variables were getting assigned their values appropriately and that my counting loop would stop for the correct cases. The test values are written in green ink and their output is directed to the side of the Priority Encoder for easy identification.

Name: Chris Baker

UID: 105.180.929



Name: Chris Baker
UID: 105.180.929



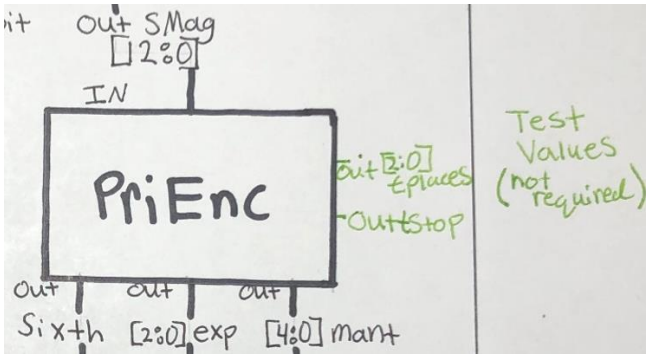
When designing the first submodule I knew from out TA that our first goal was to convert the Two's Complement number into a Sign Magnitude form. Two's Complement stores integers with a sign bit in the MSB position. By doing this the range of the integer is different from that of an unsigned integer. For a 13-bit unsigned integer there is a range from 0 to $2^{13} - 1$ which is derived from the 2^{13} possible permutations of those bits. A signed number would use the MSB to allow representation of negative numbers though, so we now have -2^{13-1} to $2^{13-1} - 1$ as our new range of values with the same amount of permutations available with 13 bits. To convert this to Sign Magnitude I chose to check if the MSB was set.

If the MSB was set, then I had a negative number. To respond to this, I would negate the bits and add 1 to create a positive signed magnitude interpretation of that number. However, this could be problematic in the case of the most negative 13-bit number (-2^{13-1} = -4096). I proceeded to handle this by checking if the MSB was still set since this is the only case that would appear negative twice (due to an asymmetric range of negative to positive signed numbers). If this was the case, then I set all the bits of the number to 1, so I would have the sign bit as 1 and use the rest of the bits as the magnitude. If the number did not consecutively

Name: Chris Baker
UID: 105.180.929

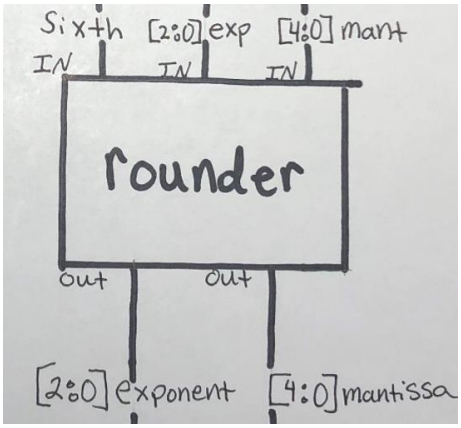
return negative, then I knew it was any negative number other then the most negative 13-bit number. In this case I set the value in the MSB to 1 to later extract that sign bit.

If the MSB was not set to begin with then we did not need to perform any manipulation on the number. Thus, we finish this submodule by collecting the sign bit from the MSB and sending the 13-bit Signed Magnitude number to the Priority Encoder.



The Priority Encoder was responsible for identifying the number of leading zeros, assigning the 5 bits following the last lead zero as the mantissa field, and storing the 6th bit from the last zero to later determine round up behavior. To count the zeros, I decided to work inwards from the 12th bit. I made this distinction based on the sign bit being stored already and to improve execution slightly. I also set a register to be able to determine when we have reached seven zeros and stop the loop for efficiency as well. Using this loop, we obtained the number of leading zeros and right after we set the mantissa bits accordingly based on our stopping position from the loop. We also used that position to locate and grab the sixth bit and store it into a register. I ran test variables like tplace and tstop to help determine if my variables were getting assigned values correctly and to identify if my loop was exiting correctly based on different signed magnitude numbers. The exponent value was determined by the table

provided in our Lab Manual. From the table I decided to start my exponent count at 7 and decrement it as the number of leading zeros increased. This module finished its purpose by assigning the sixth bit, 3-bit exponent field, and 5-bit mantissa field. These outputs would become the inputs to the rounder.

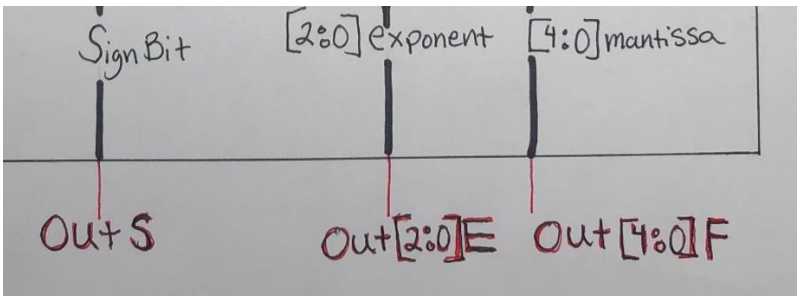


The rounder holds an especially important job of determining when to round up or truncate as well as checking for errors and edge cases. The sixth bit communicates the intention to round up the significand by adding 1. However, the issue can arise where our mantissa is already fully set. Planning for this I decided to start by checking for a set sixth bit.

Next, I test if the mantissa is fully set. This is important because adjusting for the overflow in the mantissa requires us to set the MSB of the mantissa field and add 1 to the exponent field. Thus, if the mantissa is fully set, I must also check if the exponent field is fully set. If all three conditions are true (sixth = 1, exp = 111, mant = 11111), then the only option is to declare this number as the max possible magnitude representation of floating point. To clarify, we keep the exponent and mantissa fully set and pass those values onward. This is because we have something like (x 111 11111) which is $-1^x * 2^7 * 2^{5-1} = \pm 3968$ depending

on the sign bit. Alternatively, if the exponent field was not fully set then we just need to overflow the mantissa, change the MSB to 1, and increment the exponent field. This is because the mantissa can only hold 5 bits, so overflowing it would require 6 bits to represent the true value. We adjust for this by right shifting the mantissa (dividing it by 2) and then incrementing the exponent field (multiplying by 2).

If the mantissa was not fully set, then we can safely add 1 and leave the exponent field unchanged. Lastly, if the sixth bit was never set, then we already have our correct values to return through the overall top-level module.



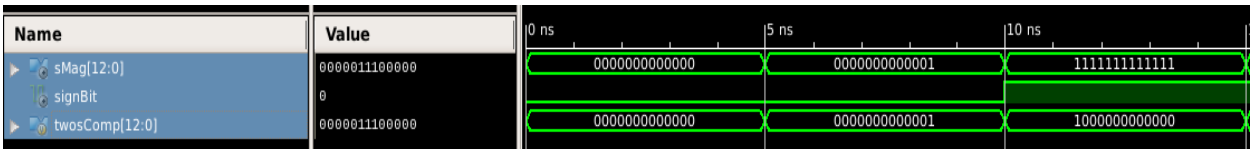
Ultimately, the first module stores and forwards the sign bit to our output S. The first module also converts the Two's Complement number into Signed Magnitude form. The next two submodules filter the bits to correctly choose the mantissa and exponent fields. Each submodule contributed to the final three outputs which makeup the 9-bit Floating Point interpretation of our original number.

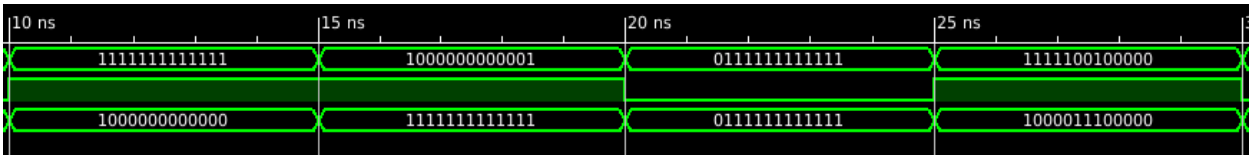
Simulations and Tests

The following tests were conducted in increments of 5 units of time based on the default timescale of 1ns/1ps. For each module there includes several tests a few of which are

required for checking more error prone cases from rounding and overflows. Other tests are performed for general correctness and to simply understand the results of our code.

Test 1: Two's Complement to Signed Magnitude (t2sm):





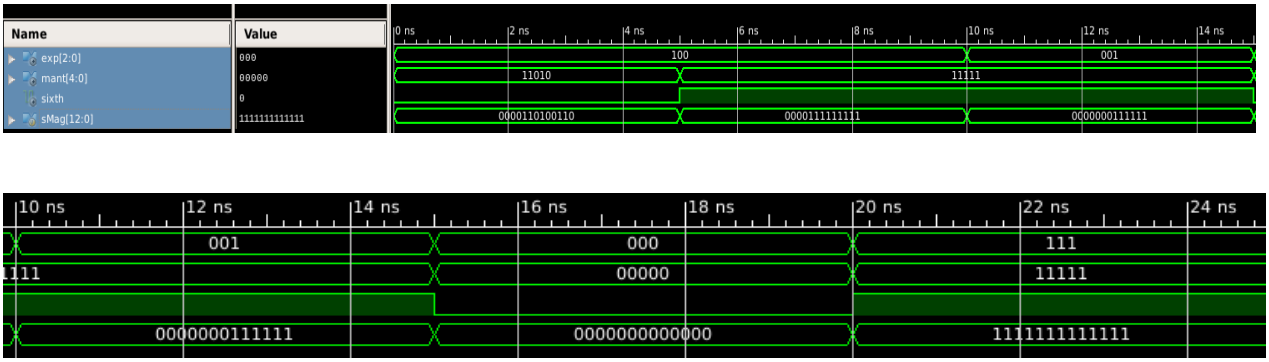
In the following we will see that the resulting MSB is determined by the original MSB. All the bits are negated and then 1 is added to the result to determine the magnitude. Special test case for most negative value included (maps to itself due to asymmetric range for signed int).

1. twosComp = 'b0;
    a. Testing mechanics with a simple test: Stores 0 for MSB, flips all 0's to 1's, then adds 1 to overflow to all 0's
2. #5 twosComp = 'b1;
    a. Second Simple Test: 1 becomes 1'b1 with the MSB 0 since it was positive
3. #5 twosComp = 'b1000000000000;
    a. Special Case: This is the most negative value and becomes the most negative value again for Two's Complement. However, when we convert it to Sign Magnitude, we want it to be the most negative value in that representation. So we store MSB, adjust the magnitude as necessary and put the 1 back into MSB.

b. Most Neg in Two Comp should become (All 1s mag, and sign bit 1)

4. #5 twosComp = 'b1111111111111;
    a. Verifying that -1 becomes 1 for a simple check that negation occurs, +1 is added, and MSB is stored and replaced
5. #5 twosComp = 'b0111111111111;
    a. Test middle case with positive MSB, we see that all 0s become 1, and then we add 1 to LSB, set MSB. This is correct and verifies positives are working correctly.
6. #5 twosComp = 'b1000011100000;
    a. Test middle case using negative MSB, verifying it works for negatives as well.

Test 2: Priority Encoder (PriEnc):



In the following tests we were able to use stop to identify when we had accumulated 7 leading zeros. We also used the tplaces as testing place holder for the number of spots we reserved for the mantissa.

1. sMag = 422;

Name: Chris Baker
UID: 105.180.929

   a. Test the Lab Manual example to make sure this is performing correctly compared with the results given in the manual. We do have exp = 4, so this was useful to validate exponent is working as intended.

2. sMag = 13'b0000111111111;
   a. Test to make sure that counting zeros and exponents are still performing correctly

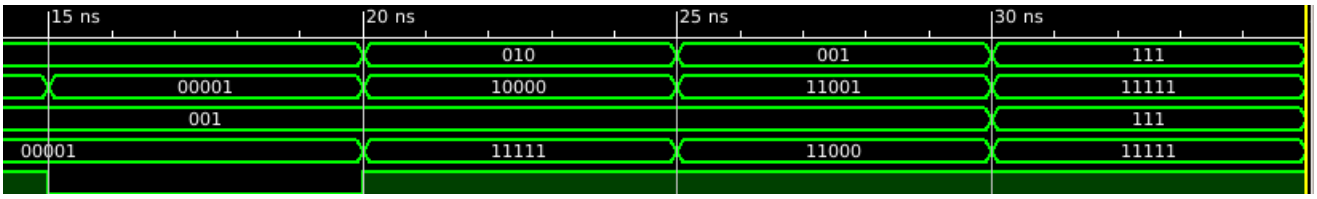
3. #5 sMag = 13'b0000000111111;
   a. This was a great test to build upon the first two. Firstly, because I have 7 leading zeros the expected exponent is 1. We did attain both values as described by the relationship in the chart for exponents and leading zeros in the Lab Manual.

4. #5 sMag = 13'b0000000000000;
   a. Test used to verify the mantissa and exponent values would both be 0 if there were not values set. This implies the sixth bit and other variables are being set as intended. For example, the least 5 bits become the mantissa and are all zero. I also verified that seven zeros were counted with another test variable.

5. #5 sMag = 13'b1111111111111;
   a. Tests that all 1's deliver the mantissa and exponent correctly. Note to get true signed magnitude values we must pass them through the T2SM submodule. So, for the testbench here I used more positive tests to verify

correctness and tested more in future models with the signed magnitude conversion.

Test 3: Rounder (rounder):



In the following tests I was able to verify that setting the sixth bit triggers the Mantissa to increase by 1. Further, by overflowing the mantissa we can increase the exponent by 1 and set the MSB of the mantissa to 1 and leave the rest of the bits as 0. Additionally, when the sixth bit is off, then no rounding occurs. We also test the special case of a fully set mantissa and exponent which returns fully set bit vectors for both fields.

1. exp = 0; sMag = 0; sixth = 1; #5
    a. We create a mock test by setting the sixth bit. This tells our code to round up the Significand. The significand performs this by adding 1. Thus, our Mantissa gives the correct value.
2. exp = 1; sMag = 0; sixth = 1; #5

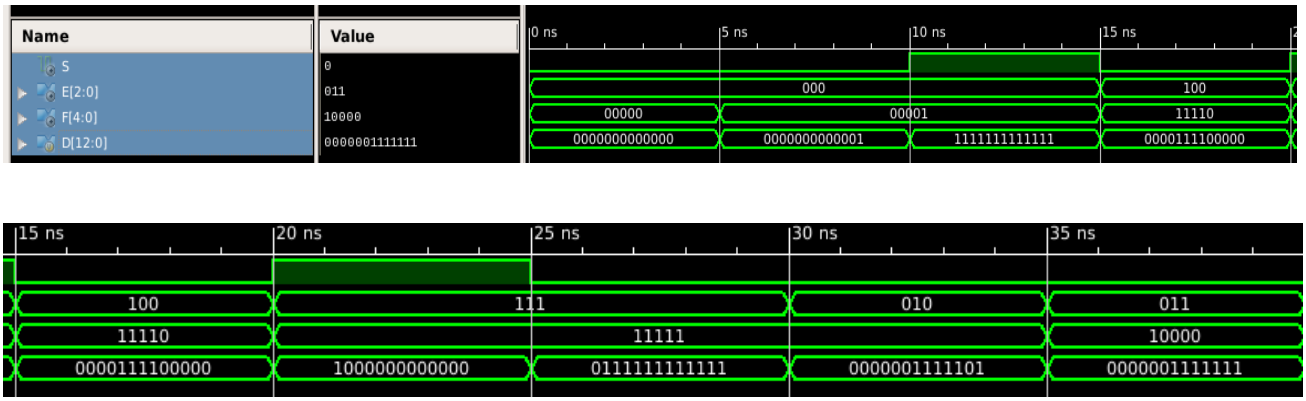
Name: Chris Baker
UID: 105.180.929

   a. Test rounding with an exponent value assigned. The exponent sets to 1 as intended and the mantissa still achieves the rounded value of becoming 1.

3. exp = 1; sMag = 1; sixth = 1; #5
   a. Test all variables set simultaneously. This demonstrates that the mantissa still rounds correctly when there is a value already set within its bit vector. The exponent is unchanged as expected.
4. exp = 1; sMag = 1; sixth = 0; #5
   a. This test shows that rounding does not take place when the sixth bit is not set. Thus, our Mantissa remains at a value of 1. The exponent is unchanged as well.
5. exp = 1; sMag = 'b11111; sixth = 1; #5
   a. We now start full mantissa tests. We overflow the mantissa and must resolve this by setting the mantissa to 10000 and increasing the exponent by 1. Mantissa does become set to 10000 as intended and exponent went from 001 to 010 as intended.
6. exp = 1; sMag = 'b11000; sixth = 1; #5
   a. Test the rounding in another case. This is meant to check accuracy at a higher mantissa. We get the correct result of 11001. Now we can proceed to test the major round case.
7. exp = 7; sMag = 'b11111; sixth = 1; #5
   a. We have fully set the mantissa and exponent. Further, the sixth bit is set. Our intended result is to set the max floating-point value in this case. Which would be (x 111 11111). Our exponent and mantissa return as both remaining fully set

demonstrating they are the exception to our earlier rules of rounding and incrementing.

Test 4: Overall Top-Level Module: Floating Point Conversion (FPCVT):



In our overall module we are mainly checking for accuracy in these examples to determine if we need to fix something from one of the sub modules that prepare the data for this execution.

1. D = 13'b0000000000000; #5;
    a. Test all 0's for a simple check. We expect all fields to be zero and we received that result
2. D = 13'b0000000000001; #5;
    a. Another simple check to determine if Sign Bit changes unexpectedly for a positive value. The mantissa, exponent, and sign bit are the correct results.
3. D = 13'b1111111111111; #5;
    a. We now test -1 to determine how this test applies to negative values as well as the opposite of our previous value (1). The sign bit has been set correctly and our

mantissa is still 1. The exponent is not set. Therefore, $-1^1 * 2^0 * 1 = -1$. Correct result.

4. D = 13'b0000111100000; #5;
    a. We test a middle case for in between behavior awareness. Since we have 4 leading zeros our exponent should be 7 – 4 = 4. The exponent is 100 which is 4. The first 5 bits after last lead zero are 11110 which matches the mantissa shown. Sixth bit is 0 = 0. No rounding. All is well so far.
5. D = 13'b1000000000000; #5;
    a. Test most negative number. A lot goes on here in the submodules. This value should be converted to (1 1111 1111 1111) by the first submodule that changes to Sign Magnitude. From here we extract our MSB as the sign bit which is 1. This is correctly shown for S=1 above. Then the exponent field is set to 7 since we have all 1s counting from left. Since we have all 1s the mantissa is also going to be all 1s. Everything is correct. Our final analysis is that $-1^1 * 2^7 * 31 = -3968$. This makes sense since we had the most negative 2's complement number and changed it to Sign Magnitude to make the most negative floating-point number.
6. D = 13'b0111111111111; #5;
    a. Like the previous test, but we check the value for a max positive number. This returns S = 0 since it is a positive number. This achieves the max positive number as intended. Therefore, this verifies that rounding and overflow behavior is adjusted correctly for a fully set mantissa and exponent field.
7. D = 13'b0000001111101; #5;

   a. Unlike the previous test with everything fully set in mantissa and exponent; we will check if they work for only one fully set. This sets up the next test to prove that rounding and mantissa overflow is working as intended. Mantissa is 11111, exponent is 010.

8. D = 13'b0000001111111; #5;
   a. We verify rounding and exponent correct behavior. The sixth bit after leading zeros is set here. Mantissa is adjusted to 10000 and the exponent becomes 011.

I performed several other tests as well that verify my submodules are working as intended to correctly yield top level appropriate results for FPCVT.

Conclusion

The Floating-Point Conversion Module was largely based from the Lab Manual for this project. The overall design was given such that we should be able to take a 13-bit Two's Complement input by the variable name of D. Upon passing bit vector D through the module we should be able to create a nonlinear representation of that number. The fields that make up the floating-point interpretation are the significand (also referred to as mantissa), exponent, and sign bit. The significand is a 5-bit vector, the exponent is a 3-bit vector, and the sign bit is represented as a pin or 1 bit.

Bit vector D passed into our first submodule which converted the value into a Signed Magnitude representation. This submodule sent the sign bit from the MSB as a return value to the main module and the Signed Magnitude number was passed to the next submodule. In the Priority Encoder leading zeros were counted, 6^{th} bit was identified, the mantissa bits were

collected, and the exponent value was set. The sixth bit, exponent field, and mantissa field may be correct for some numbers, but there were overflow and rounding errors that were possible. Because of this we fed those as inputs to the rounder submodule. In the rounder we correctly handled cases where floating point fields were fully set and other variations of factors. To resolve the issues derived from overflow and rounding we implemented logical shifts, value setting, and incrementing. We also checked for edge cases and set values to max positive or negative when appropriate. The result was a 9-bit floating point representation of the original bit vector D. The fields contained in the 9-bit vector are the sign bit, mantissa, and exponent.

Working through this lab I encountered several difficulties when trying to run test bench simulations. There were many times where uninterpreted values appeared when I ran the simulations. I kept making changes to code but noticed sometimes the values appeared regardless of those changes. I realized that I was clicking the wrong file and running the test bench incorrectly. Once I figured out this issue then I had problems with incorrect assignments. This was made worse by several compiler errors like HDL 806 and HDL 44. The HDL 806 error occurred when I first tried to make a series of if statements to determine counting zeros and setting values in the Priority Encoder. I checked documentation and it appeared I did not properly open or close a process such as a conditional, however the code was messy and long, so I use a loop perform the same process in a simpler manner. The HDL 44 error dealt with incorrectly using data types in conditionals or loops. I decided to change variables and use constants to avoid these problems altogether. Solving these issues in the first submodule made the following modules less error filled. I resolved most logic issues by writing values on paper and performing mini tests as I wrote the code. I realized upon testing that I almost forgot to

Name: Chris Baker
UID: 105.180.929
include the adjustment for mantissa overflow but was paying attention to the side work I was doing on paper to check my results.

This lab was an important because of the common application of converting digital and analog signals. The real world operates in real time continuous signals, but to interpret data we sometimes work on analysis in discrete numbers. Moreover, commercial industries benefit from the precision and range offered by floating point numbers where speech recognition and music are more efficiently controlled.

One suggestion for the lab would be to rewrite the first few sentences explaining the Input Format Section. I had to reread that part several times to understand what was meant by the leading zeros and exponent explanation. I was able to understand the intention numerically, but the writing explanation could be made clearer. My last suggestion for the lab is to have a pdf or helpful study sheet with similar operations or modules created. Further, understanding how modules link across multiple pages and testbenches would be appreciated greatly.