CS M152A Lab 2

**Introduction and Requirement**

For this lab, we created a combinational circuit that converts a 12-bit linear 2’s complement encoding into a compounded 8-bit Floating Bit representation. When inputting the 12 bit 2’s complement encoding (which in our case is d[11:0]) we should get the following segments of output for the floating bit representation: the sign bit (s) , 3-bit exponent portion (output\_e[2:0]), and 4-bit significand portion (output\_s[3:0]). Hence, our task was to extract the sign bit, and compute the exponent and significand based on the instructions of the lab. To retrieve the value of the floating point representation, we use the following equation: (-1)^S \* F \* 2^E. Altogether, we would get a result like this from a high level:

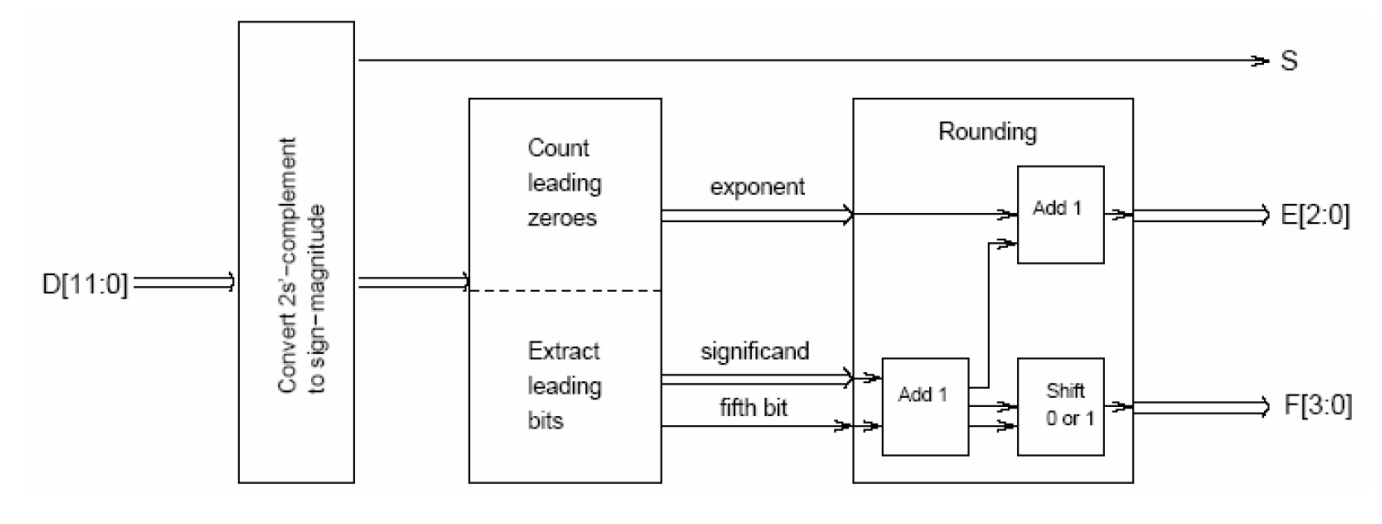


One thing to keep note of is there is the possibility of some decimal numbers having multiple Floating Point Representations. For example, the value 56 has two valid FP Representations: 00110111 and 00101110 (7 \* 2^3 = 56 vs. 14 \* 2^2). The preferred representation is the one with the MSB of the significand being a 1 (in this case, it would be the second FP representation would be the preferred. This is known as normalization.

An important aspect of this lab was the concept of rounding, since certain numbers in 12-bit 2’s complement don’t necessarily translate over to 8-bit floating point representation. Hence, we had to account for rounding up/down when necessary, which in our case was dependent on the fifth bit after the significand. If the fifth bit is 0, then the significand is the first four bits. If it is 1, we add 1 to the significand. We did have to account for two edge cases in the chance there is overflow: 1) If the max significand is rounded up, we had to set the signficand to 1000 and increment the exponent and 2) If the exponent is exponent is incremented past its max, then just set the significand and exponent to all 1’s.

**Design Description**

The float-point conversion circuit utilized the block diagram shown below. In order to modularize the program and logic, we decided to divide the float-point conversion with three modules. The first module takes care of the initial input and converts the 2's complement input to sign-magnitude, the second module counts the leading zeroes and extracts the leading bits, and the third module manages the rounding portion of the circuit and outputs the 3-bit exponent and 4-bit significand. At the end, a module integrated these three modules and allowed the float-point conversion circuit to operate as intended.

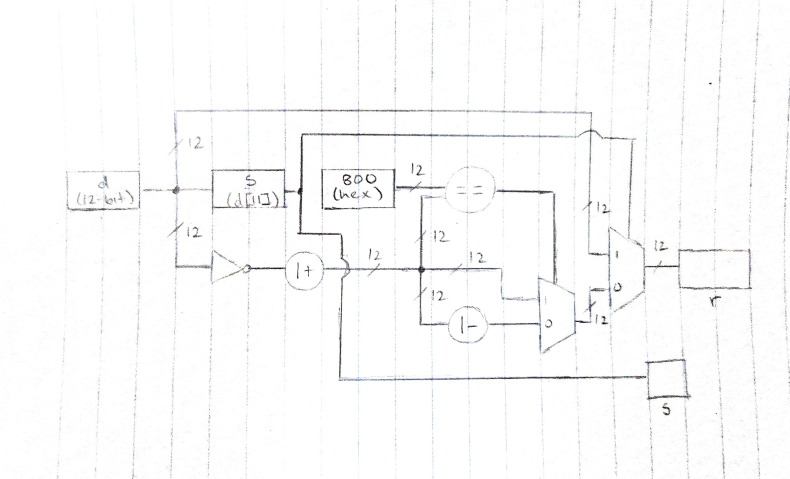


**First Module - Convert 2's Complement to Sign's Magnitude (convert2sign.v)**

| module convert2sign(input [11:0] d, output [0:0] s, output [11:0] r); |
| --- |

The first module's main objective is to convert the 12-bit two’s-complement input to sign-magnitude representation. As the module header shows above, the module receives a 12-bit two's-complement input (d) and outputs the sign bit (s) alongside the 12-bit sign-magnitude representation (r). In order to do this, the module first extracted the sign bit from the 12-bit two's complement input and assigned the bit value to output s. Then, to check if the input requires any modifications, the module questions whether or not the sign bit is 0 or 1. If the sign bit is 0, the module simply assigns the 12 bits of d to output r because nonnegative numbers remain unchanged. However, if the sign bit is 1, then the module changes the negative input to its absolute value by taking the complement of all the input bits and adding 1. One important consideration is that an edge case exists where the input value is -2048. If this occurs, the computation will incorrectly convert the input to the 12-bit value of 100000000000 instead of 011111111111. Thus, to fix this edge case, whenever the edge case is detected, the module manually reverts the sign-magnitude representation to the correct representative value.

The schematic for the first module is also provided below.



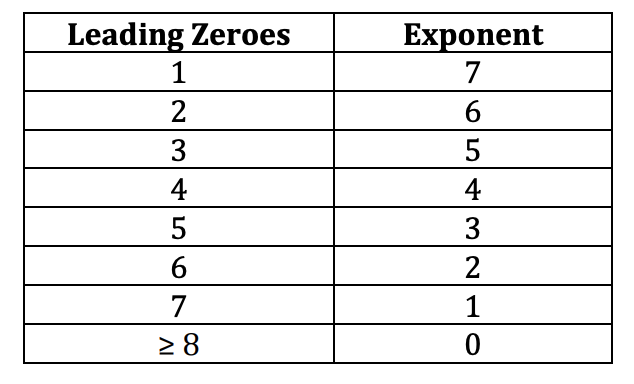
**Second Module - Count Zeroes and Extract Leading Bits (linearToFB.v)**

| module linearToFB (input [11:0] r, output reg [2:0] e, output reg [3:0] s, output reg f); |
| --- |

The second module's main objective is to perform the basic linear to floating point conversion. As the module header shows above, the module receives a 12-bit sign-magnitude input r and outputs the 3-bit exponent e, 4-bit significant s, and the 1-bit fifth bit. In order to do this, the module utilized multiple if statements to accurately count the number of leading zeroes; we structured the if statements like a priority encoder (as shown below).

| **If Statement Code Portion** |
| --- |
| if (r[11] == 1) begin  e = 3'b111;  s = 4'b1111;  f = 1'b1;  end else if (r[10] == 1) begin  e = 3'b111;  s = r[10:7];  f = r[6];  end else if (r[9] == 1) begin  .  .  . |

Depending on the number of leading zeroes, we assigned the three output values within each of the "if statement" code blocks. To compute the exponent value, the module simply used the table below. The module calculated the significant output by extracting the four bits starting from the leading zero bit, and the fifth bit output was assigned by extracting the bit five spots away from the leading zero bit.



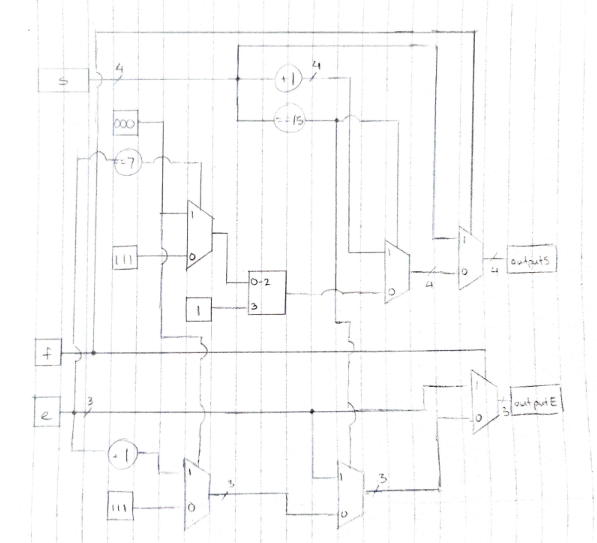
**Third Module - Rounding (rounding.v)**

| module rounding(input [2:0] e, input [3:0] s, input [0:0] f, output reg [2:0] outputE, output reg [3:0] outputS); |
| --- |

The third module's main objective is to perform the rounding of the Floating Point Representation. As the module header shows above, the module receives a 3-bit exponent input, 4-bit significand input, 1-bit fifth bit input, and outputs a 3-bit rounded exponent and 4-bit rounded significand. In order to do this, the module first evaluates the fifth bit value. If the fifth bit following the last leading zero is 1, then the significand is incremented by 1. Then, the module checks if the significand overflows; if it does, then the significand is shifted right one bit and the exponent is incremented by 1 as well. If the fifth bit following the last leading zero is 0, then every value remains the same since no rounding is needed.

The edge case to consider for rounding is the scenario where the fifth bit is 1, the significand input is 1111, and the exponent input is 111; since literally all the values will overflow in this scenario, the module must simply not increment any value and keep every value the same.

The schematic for the third module is also provided below:

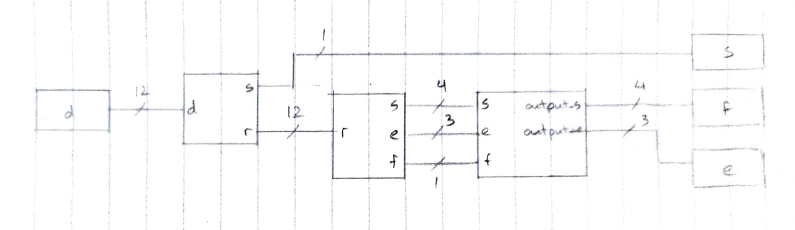


**Final Module - Integrating the 3 Modules (FloatingPointConverter.v)**

| module FloatingPointConverter(input [11:0] d, output [0:0] s, output [2:0] e, output [3:0] f); |
| --- |

The final module's main objective is to seamlessly integrate the three modules mentioned above and serve as the high-level "controller" of the circuit's input and outputs. As the module header shows above, the module receives a 12-bit 2's complement input d and outputs the 1-bit sign bit s, 3-bit exponent e, and 4-bit significand f. The module essentially calls the three modules and accurately assigns the inputs and outputs for the three modules to provide the final circuit's floating point conversion output.

The schematic for the high level of the entire Verilog module is provided below:



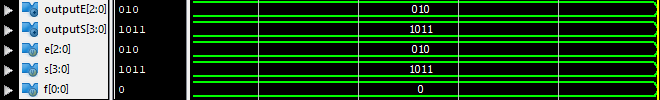
**Simulation Documentation**

We had 3 separate Verilog modules for each block and tested them individually. The first block converts the 12-bit two’s-complement input to sign-magnitude representation, the file we used for testing it is called convert2signTB.v. The waveform of one of the test cases (input is 000000101100 or 44 in decimal equivalent), that we used is given below where s holds the sign bit and r is the place holder for the input d which is the input data in Two’s Complement Representation. As seen below, we have found the sign bit to be 0, which is correct since 44 is positive.

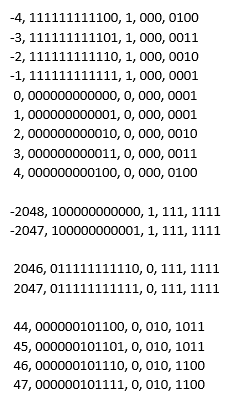


The second testbench, linearToFBTB, tests the basic linear to Floating Point conversion. Continuing with the previous test case, the waveform of one of the test cases that we used is given below, where s is the 4-bit significand, E is the 3-bit exponent portion, and F refers to the sign bit. Since there are 6 leading zeroes, that corresponds to a value of 2 (or 010) for the exponent piece. In turn, the significand should be 1011 since that is the next four bits following the four zeroes. Finally, the fifth bit after the significand is 0. All of these are computed correctly by our program as seen below.



The third test bench, rounding, tests the rounding of the Floating Point Representation. The waveform of our previous test case that we used is given below, where outputE is the 3 bit exponent, and outputS is the 4 bit significant. Since our fifth bit is 0, there was no need for us to round, and hence we are done with the final step of the conversion. 

We have a final module which uses all 3 of these modules to obtain our final 8-bit Floating Point representation. We tested this file for numbers ranging from -2048 to 2047, paying close attention to edge cases. Tricky edge cases that we found were when the decimal is -1 or 0, or the beginning and end of our range which is -2048 and 2047. To check that these values were correct, we manually calculated the floating point representations for some of the values using the same steps as the lab. The console output for the edge cases and a few general cases is given below (from left to right, it is the decimal representation of the number, 12 bit two’s complement, sign bit, exponent portion, and significand portion):



One bug that we found during testing while looking at the simulation waveforms was that our program was not giving the correct 4 bit significand in the case of the sign bit being 0. Upon closer inspection, we realized that in our third block, we have started with “if f==1” i.e. when the sign bit is one, but we never wrote the else statement for this which will look at the case when the sign bit is zero. Other bugs that we encountered involved the usage of wires vs. registers, in which we realized that we would need to use wires if we wanted the output of one module to be the input of another module. Finally, we were having trouble having the for loop working to test our various test inputs, in which we were attempting to see the waveforms of all the different inputs/outputs. However, we realized later on that a better way to go about it would be to use the console output with the help of the “display” function.

**Conclusion**

To summarize, we compartmentalized the problem for this lab into 3 different stages: 1) extracting the sign bit from the MSB of the 12-bit 2’s complement input (if negative, we invert the rest of the bits). 2) Retrieving the exponent encoding depending on the amount of leading zeroes. 3) Rounding based on the value of the fifth bit (as well as handling edge cases). When designing each module and testing them with our test benches, we encountered several problems, which were mainly centered around how to use proper Verilog syntax and how linking between the modules worked. These problems that came up were quickly resolved through searching online forums about the problems we faced as well as asking the TA for guidance. In the end, we ultimately learned quite a bit about the inner workings behind conversion between two’s complement and floating point representations. We also realized how manageable this lab was when separating each step of the process into its own module, which may prove to be useful for future labs to come.