Lab 1: Floating Point Conversion

CS M152A Lab 3
TA: Gu, Hongxiang
April 22, 2019

Nathan Marsh,
Denise Wang,

Introduction

The goal of this lab is to design a combinational circuit that converts a 13-bit linear encoding of an analog signal into a compounded 9-bit floating point representation. This 9-bit floating point output is represented by 1-bit sign representation, a 3-bit exponent, and a 5-bit significand, which is given by the value. Due to the compression operation, there are more input bits than output bits, so it is likely that multiple linear encodings can be mapped to the same 9-bit floating point representation. It is also likely that there are multiple linear encodings that do not have floating point representations, so they are mapped and rounded to the closest floating point encoding. This design consisted of three modules. The first module converts the 13-bit two's-complement input to sign-magnitude representation. For the second module, a priority encoder was implemented to count the leading zeros, which performs the basic linear to floating point conversion. The last block performs rounding of the floating point representation. These modules were designed in Verilog HDL using the Xilinx ISE software.

Design Description

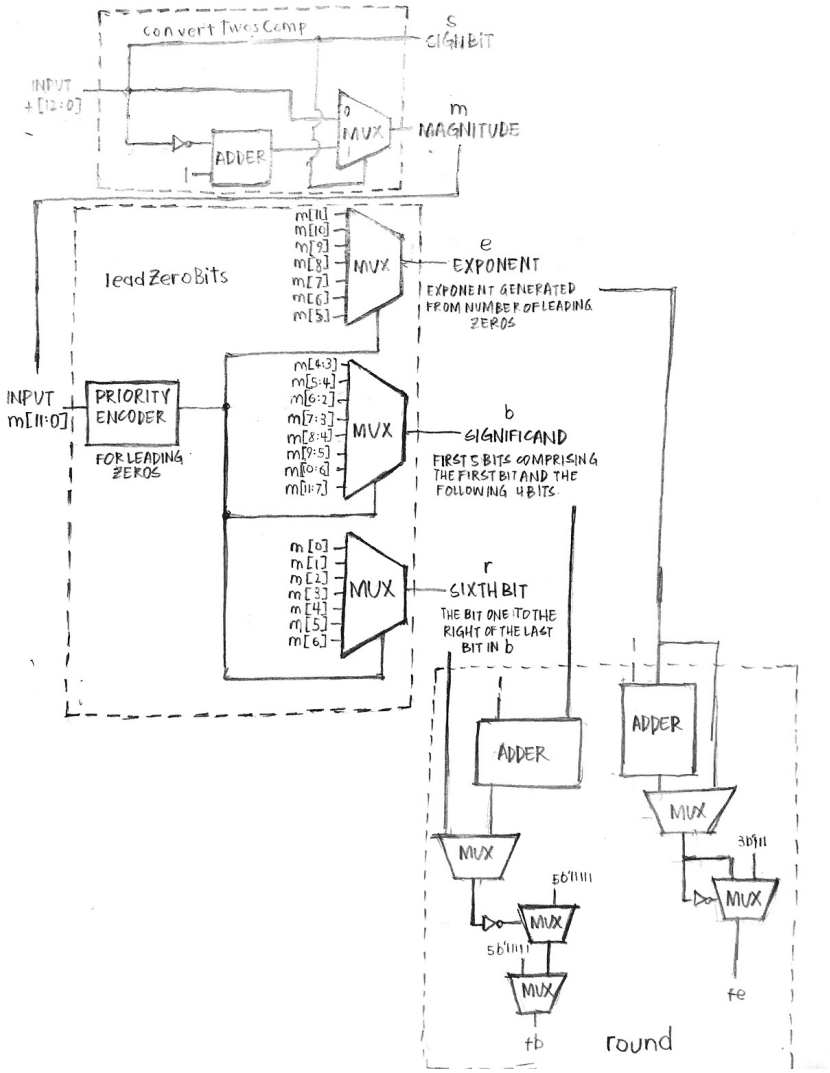

Figure 1.1- Design of each submodule.

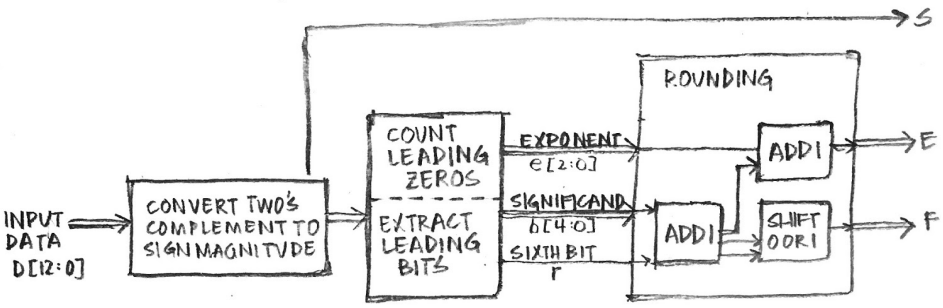

Figure 1.2- Overall design of FPCVT module.

The design for the floating-point converter included three submodules that converted twos complement into sign magnitude, to convert magnitude into significand and exponent bits, and rounding. The final module connected the three submodules to obtain the desired 13-bit two's complement input to 9-bit floating point representation.

In the first module called convertTwosComp, we converted a 13-bit two's complement number into a 1-bit sign and 12-bit magnitude number using simply wires and ternary operators. The sign bit was obtained by taking the leading bit of the two's complement. Generating the magnitude required more logic because taking the two's complement of the negative minimum number is still the same number. Using ternary operators, we took the 12 least significant bits if the number was positive, took the twos complement number if it was negative, and the all ones for the magnitude if it was the smallest negative number.

The next module called leadZeroBits takes a 12-bit magnitude number and finds the 3-bit exponent, 5-bit significand, and 1-bit for rounding. First, we obtained the exponent using a 3:8 priority encoder. We implemented the priority encoder by calculating each bit using boolean logic with the 7 most significant bits of the magnitude, then concatenating all three bits. With the exponent calculated, we used an always block and cases statements to determine which 5-bits to get for the significand. Inside the always block we used another case statement to find the 1-bit for rounding as well based on the exponent.

The last submodule called round takes a 3-bit exponent, 5-bit significand, and 1-bit for round and outputs the rounded 3-bit exponent and 5-bit significand using only wires and ternary operators. To round the significand, we did nothing to the input if the round bit was zero or the 5-bit significand and exponent were all ones, added one to the significand if the round bit was one, right shifted and added one to the significand if it was all ones and the exponent was not all ones. For the exponent, if the rounding bit was one, the significand were all ones and the exponent was not all ones we added one, but otherwise we kept it the same.

The final floating-point converter module called FPCVT takes one 13-bit two's complement input and outputs a 1-bit sign, 3-bit exponent, and 5-bit significand as specified by creating instances and connecting the submodules previously described. Other than the inputs and outputs we needed four wires for the intermediate magnitude, exponent, significand and round bits. First, we create an instance of the convertTwosComp module by inputting the 13-bit twos complement input and outputting the final sign bit and the intermediate 12-bit

magnitude. Next, we instantiate the leadZeroBits module with input the intermediate 12-bit magnitude and the output intermediate 3-bit exponent, 5-bit significand, and 1-bit for rounding. The last submodule round takes as input the intermediate 3-bit exponent, 5-bit significand, and 1-bit for round and outputs the final 3-bit exponent and 5-bit significand. After connecting all the modules as described, we were able to complete the functionality as exactly as required.

Simulation Documentation

Test Case	Result	Pass/No Pass
Positive input value t= 13'b0000011001100	Remains as is s= 0 m= 000011001100	Passed
Negative input value t= 13'b1001001001100	Bits are inverted, 1 added s= 1 m=110110110100	Passed
Maximum positive input t=13'0111111111111	Remains as is s= 0 m=111111111111	Passed
Minimum negative input t=13'1000000000000	Converted to largest value s=1 m=111111111111	Passed

Table 1.1- convertTwosComp Test Cases. The table displays the test cases used for the leadZeroBits module. The first column shows the 13-bit two's complement input values (t) that were tested, the second column shows results of the test cases with the correct values for the sign bit (s) and magnitude (m), and the last column shows whether the test cases passed or did not pass.

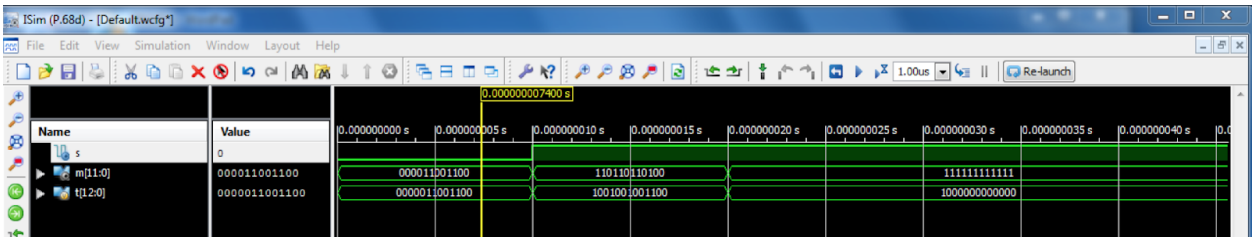

Figure 1.3- convertTwosComp Simulation. This screenshot displays the waveform of the convertTwosComp module. The s represents the sign bit, the m[11:0] represents magnitude, and the t[12:0] represents the two's complement number. The right side represents the resulting simulation of each of the test cases inputted in Table 1.1.

Test Case	Result	Pass/No Pass
2 leading zeros m= 12'b010110000000	Bits shifted successfully e= 110 b= 10110 r= 0	Passed
3 leading zeros m= 12'b001100101000	Bits shifted successfully e= 101 b= 11001 r= 0	Passed
5 leading zeros m = 12'b000011011010	Bits shifted successfully e= 011 b= 11011 r= 0	Passed
10 leading zeros m= 12'b000000000100	Zero bits shifted e= 000 b= 00100 r= 0	Passed

Table 1.2- leadingZeroBits Test Cases. The table displays the test cases used for the leadZeroBits module. The first column shows the 12-bit input values (m) that were tested, the second column shows results of the test cases with the correct values for the exponent (e), significand (b), and the bit following the last in b (4), and the last column shows whether the test cases passed or did not pass.

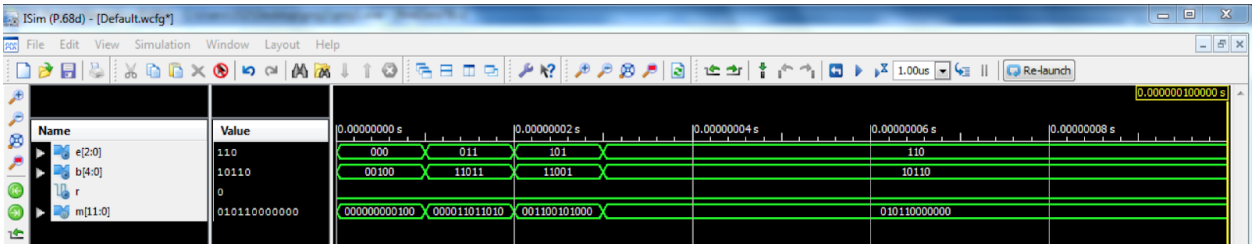

Figure 1.4- leadZeroBits Simulation. This screenshot displays the waveform of the leadZeroBits module. The e[2:0] represents the exponent generated from the number of leading zeros, the b[4:0] represents the first 5 bits comprising the first one and the following four bits, the r represents the bit one to the right of the last bit in b and the m[11:0] represents the 12-bit magnitude. The right side represents the resulting simulation of each of the test cases inputted in Table 1.2.

Test Case	Result	Pass/No Pass
e= 010 b= 11010 r= 0	Rounds successfully, 5-bit significand fe= 010 fb= 11010	Passed
e= 010 b= 11010 r= 1	Rounds successfully, 5-bit significand fe= 010 fb= 11011	Passed
e= 111 b= 11111 r= 1	Rounds successfully, overflow case fe= 111 fb= 11111	Passed

Table 1.3- round Test Cases. The table displays the test cases used for the round module. The first column shows the 3-bit exponent as specified earlier above (e), the 5-bit input as specified earlier above (b), and the round as specified earlier above (r), the second column shows results of the test cases with the final exponent (fe) and the final 5-bit significand (fb), and the last column shows whether the test cases passed or did not pass.

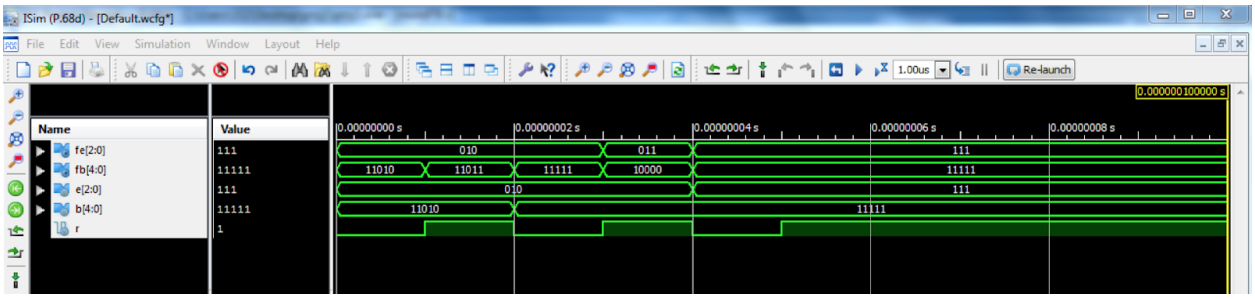

Figure 1.5- round Simulation. This screenshot displays the waveform of the round module. The fe[2:0] represents the final 3-bit exponent, the fb[4:0] final 5-bit significand, the e[2:0] represents the exponent, the b[4:0] represents the 5-bit input, and the r represents the round. The right side represents the resulting simulation of each of the test cases inputted in Table 1.3.

Test Case	Result	Pass/No Pass
Positive input value D= 13'b 0000011001100	S= 0 E= 011 F= 11010	Passed
Negative input value D= 13'b 1001001001100	S= 1 E= 111 F= 11011	Passed
-1 input D= 13'b 1111111111111	S= 1 E= 000 F= 00001	Passed
Maximum positive input D=13'0111111111111	S= 0 E= 111 F= 11111	Passed
Minimum negative input D=13'1000000000000	S= 1 E= 111 F=11111	Passed

Table 1.4- FPCVT. The table displays the test cases used for the FPCVT module. The first column shows the 13-bit two's complement input values (D) that were tested, the second column shows results of the test cases with the correct values for the sign bit (S), the 3-bit exponent (E), and the 5-bit significand (F), and the last column shows whether the test cases passed or did not pass.

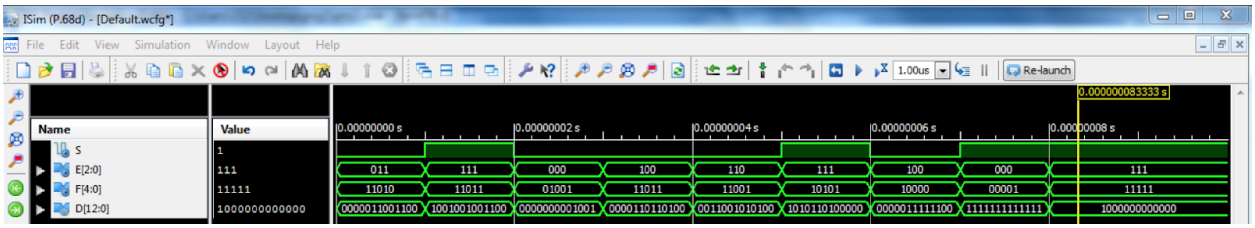

Figure 1.6- FPCVT Simulation. This screenshot displays the waveform of the FPCVT module. The S represents the sign bit, E[2:0] represents the exponent, the F[4:0] represents the significand, and the D[12:0] represents the input data. The right side represents the resulting simulation of each of the test cases inputted in Table 1.4.

Conclusion

The floating-point conversion module comprises three submodules that wired together. First, the input is converted from a 13-bit twos complement number to a 1-bit sign and a 12-bit magnitude representation. Next, with the 12-bit magnitude we calculated the 3-bit exponent, unrounded 5-bit significand, and a rounding bit based on the number of leading zeros, the following five bits, and the bit after those five bits. In the last submodule, we input the 5-bit significand, the 3-bit exponent, and the 1-bit rounding number to round and then change the final 5-bit significand and sometimes the 3-bit exponent if the rounding bit was asserted. Finally, the floating-point conversion module linked all of these modules together to get the 9-bit specified output of 1-bit sign bit coming from the first module, and the 3-bit exponent and 5-bit significand coming from the last submodule described.

For this first Verilog lab, we experienced difficulties with the “x” unknown data value, case statements, and always blocks. When we initially tried to compile our code, we attempted to use case statements without inserting them into the always block because we wanted to use the functionality of case without using registers. After changing some wires to registers and putting our case statements in our always blocks we were able to make the file compile; however, the logic in our case statements was still flawed. In an attempt to make a priority encoder, we implemented our case statements with a certain number of zeros followed by a one followed by enough “x” bit to fill the 12-bit magnitude. We hoped that the compiler would treat these “x” bits like don’t cares but, in reality 0 and 1 both failed to match the “x” bit and we were left always taking the default case. To fix it, we used a combinational priority encoder with assign statements and bit logic then used the generated exponent as the input to our other leading zero dependent inputs: the 5-bit significand and the rounding bit. At this point, all individual inputs we tested worked perfectly. However, when we attempted to pass the script to test our functionality, we got seven thousand out of nine thousand errors. After some clever thinking, we realized that our always block sensitivity list only contain the exponent which meant that the for all the values with the same exponent, no new value would be generated. After changing the sensitivity list to a star instead of the exponent we passed all test cases.

To improve the lab, I would suggest doing a more common conversion that is actually performed in a computer. While the simplicity was nice for a first lab, learning to convert from a 16-bit floating point representation to a 32-bit floating point representation might be more valuable because this conversion is done frequently in computer science. Doing this conversion

would also learn about how floating point is actually stored in a computer following the IEEE standard with a bias for the exponent and a longer mantissa. Apart from the practicality, this lab seemed almost perfect for students just learning Verilog because it required attention to detail to some edge cases, but the general case was simple enough to understand and combined both logic and module driven modules.