CSM152a Lab 1 Report: Floating Point Conversion

Team:
Gajan Nagaraj
Frank Ren
Everett Sheu

Professor: Miodrag Potkonjak
TA: HongXiang Gu
Lab 1

Introduction:

The goal of lab 1 was to use Xilinx ISE software to design, implement, and test a two's complement to floating point convertor in Verilog HDL. As seen in Figure 1, such a system takes a 12-bit linear encoding of an analog signal and converts it into a compounded 8-bit floating point representation.

For this lab, the input is a 12-bit number in two's complement representation. Two's complement is a binary representation of numbers where the negative of a binary number is represented by taking the complement and adding one. The output of the convertor will be a simplified floating point representation consisting of 1-bit indicating the sign of the number (S), 3-bits for the exponent (E), and 4-bits for the significand (F). This floating point output will represent the number in the following manner: $(-1)^S \cdot F \cdot 2^E$.

The floating point convertor for this lab will be based on simulation only. This means that our convertor will not be synthesized and run on the FPGA. Instead we used the Xilinx ISE program to simulate the VHDL code on the lab workstation itself.

For future reference, our floating-point converter will use the pin description as described in the following table:

D [11 : 0]	Input data in Two's Complement Representation. D0 is the Least Significant Bit (LSB). D11 is the Most Significant Bit (MSB).
S	Sign bit of the Floating Point Representation.
E [2 : 0]	3-Bit Exponent of the Floating Point Representation.
F [3 : 0]	4-Bit Significand of the Floating Point Representation.

Table 1: Pin description of our floating-point converter. It details the inputs and outputs of our module implementation.

Design Description:

For the design of the floating point converter, we implemented three different modules to help us convert 12-bit two's complement to 8-bit floating point: twotosig, priEncode, and round. We then combine the three modules together in our main module FPCVT.

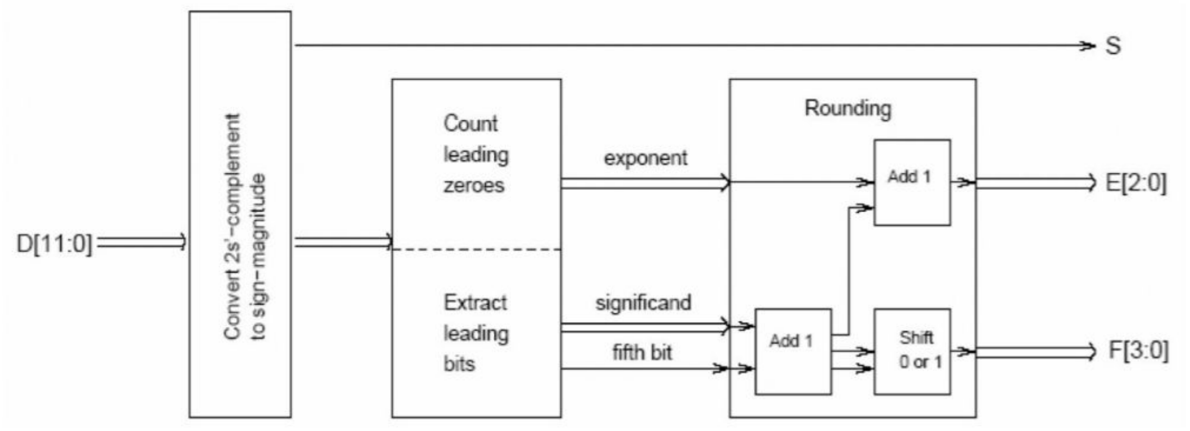


Figure 1: The Overall Design Diagram of Floating Point Converter. The 3 separate modules are shown in the figure and our main module combine all 3 into our main module FPCVT.

Two's Complement to Sign Magnitude (twotosig):

The first step in floating point conversion is the twotosig module which takes a 12-bit two's complement representation input (D[11:0]) and converts it into sign magnitude representation. In sign magnitude representation, the most significant bit indicates the sign of the number and the rest of the bits indicate the magnitude of the number. Therefore to convert from two's complement to sign magnitude we must first detect if D is positive or negative.

If the most significant bit (D[11]) is zero then D is positive, else it is negative. If D is positive, then D is also our sign magnitude representation. If D is negative, our sign magnitude representation is the complement of D plus 1. The exception is when D is 12'b1000_0000_0000. Here, D represents the most negative number possible in two's complement form. In this case, taking the complement and adding one does not result in an equivalent representation in sign magnitude form. Thus, we manually set the output sign magnitude representation to 12'b1111_1111_1111.

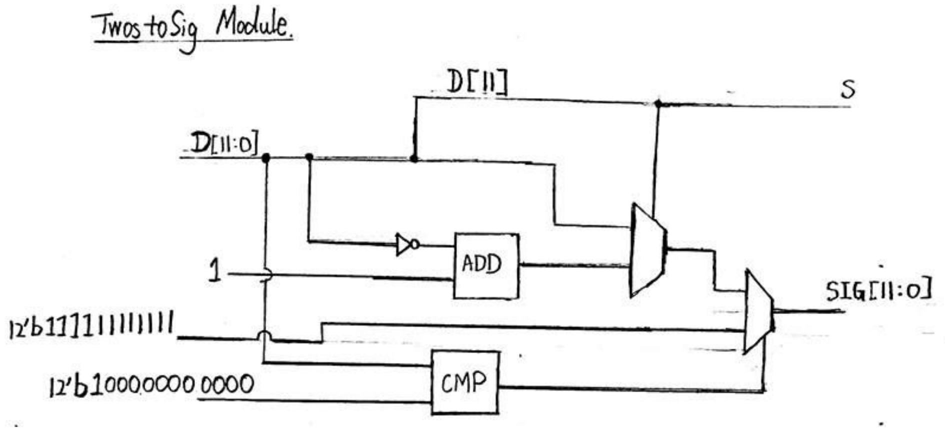
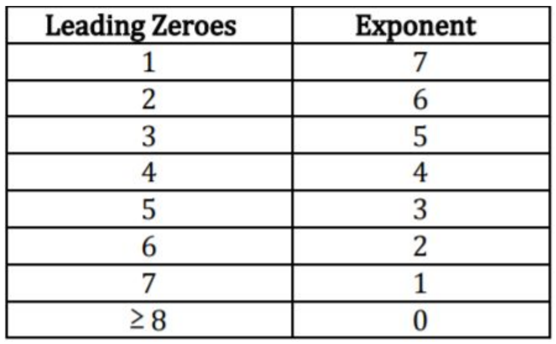


Figure 2: Schematic of twotosig module. We have four inputs and two outputs in this module. We use a adder to add the inverted input D[11:0] and 1 if the number is negative. We also use a conditional statement to compare the input with our special case to determine the final representation.

Priority Encoder/Count Leading Zeros (priEncode):

The Priority Encoder takes in the sign magnitude output of the twotosig module and outputs the preliminary exponent and significand values of the floating point representation. In order to transform the sign magnitude representation to floating point, we must count the number of leading zeros to obtain the floating point exponent and magnitude of the number.

Figure 3, provided by the lab manual, maps the number of leading zeros to the value of the exponent. We implemented this logic using a priority encoder which locates the first “1” in the sign magnitude representation starting from the most significant bit. Using the location of the first “1” we can calculate the number of leading zeros, and thus map the corresponding exponent. Then, starting from the location of the first “1”, four bits are extracted as the significand.



Leading Zeroes	Exponent
1	7
2	6
3	5
4	4
5	3
6	2
7	1
≥ 8	0

Figure 3: Table that details the conversion of leading zeros to exponent value of the floating-point number

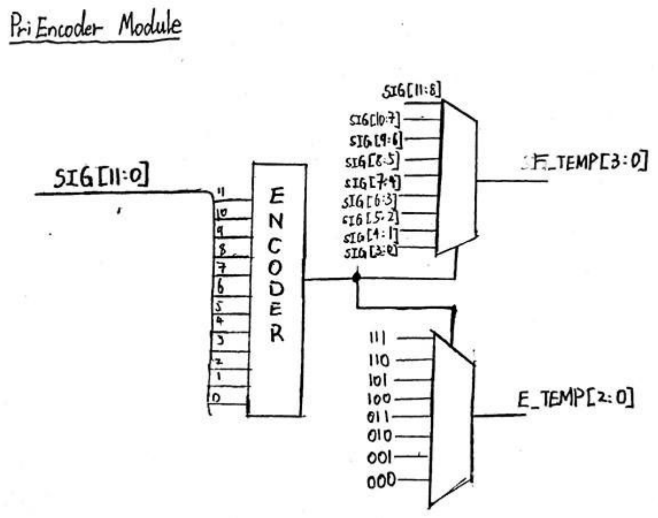


Figure 4: Schematic of priEncode module. The priority encoder takes in the sign magnitude and find the first "1" in the bits, then it uses that information to decides the exponent and significand value of the floating point representation.

Rounding (round):

Finally, our floating-point converter must be able to deal with rounding properly. The round module implements rounding by examining the single bit following the 4 bits of the significand. We will refer to this bit as the fifth bit. The fifth bit tells us whether to round up or down. If it is 0, the nearest number is obtained by truncation – simply using the first four bits. If, on the other hand, the fifth bit is 1, the representation is obtained by rounding the first four bits up – by adding 1.

Next, we must deal with the issue when the significand is saturated and can no longer be incremented. This is done by resetting the significand to zero and incrementing the exponent. For example,

000001111101→	0	3	10000	OOPS! →	0	4	1000

However, there are some issues that we run into when this is done. Namely, there is a problem when rounding is required but the preliminary values of both the significand and exponent are saturated. In this scenario, neither value can be incremented any further. The solution is to preserve the saturated preliminary values of the exponent and significand for use as the output, thus using the largest possible floating point representation available to us.

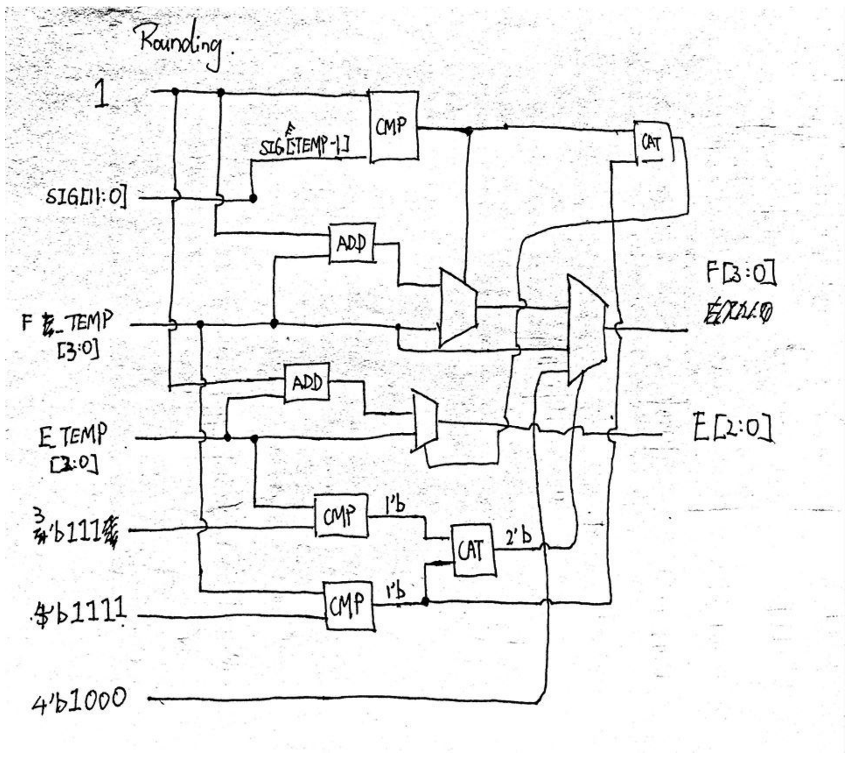


Figure 5: Schematic of round module. The rounding module takes the sign magnitude output of the twotosig module and the preliminary exponent and significand outputs of the priEncode module as inputs. It uses these inputs to decide if rounding is required based on the value of the fifth bit following the significand and the preliminary values of the exponent and significand.

Simulation Documentation:

In order to assess the correctness of our floating-point converter implementation, we created our own testbench with test cases to validate our module. Each test case was chosen specifically to test aspects of both the converter as a whole and the sub-modules by themselves.

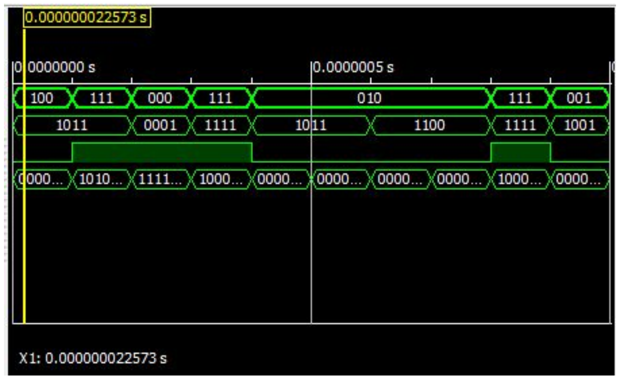


Figure 6: Waveform output of our own testbench. From top to bottom, each waveform represents the values of the E[2:0], F[3:0], S, and D[11:0] respectively. The waveform output of our testbench is detailed more thoroughly in the **Figure 7**.

Test Case	Input	Output	Pass/No Pass

1	D = 12'b0000_1010_1010	S = 0
		E = 100
		F = 1011
			PASS

2	D = 12'b1010_0100_0100	S = 1
		E = 111
		F = 1011
			PASS

3	D = 12'b1111_1111_1111	S = 1
		E = 000
		F = 0001
			PASS

4	D = 12'b1000_0000_0000	S = 1
		E = 111
		F = 1111
			PASS

5	D = 12'b0000_0010_1100	S = 0
		E = 010
		F = 1011
			PASS

6	D = 12'b0000_0010_1101	S = 0
		E = 010
		F = 1011
			PASS

7	D = 12'b0000_0010_1110	S = 0
		E = 010
		F = 1100
			PASS

8	D = 12'b0000_0010_1111	S = 0
		E = 010
		F = 1100
			PASS

9	D = 12'b1000_0010_1111	S = 1
		E = 111
		F = 1111
			PASS

10	D = 12'b0000_0001_0001	S = 0
		E = 001
		F = 1001
			PASS

Figure 7: Table detailing the test cases used and their results.

Test Cases 1 through 4 were used to ensure the proper functionality of our twotosig module, which is validated by the correct outputs of S, E, and F for each case. Cases 1,2 and 3 validate that the module correctly passes along the correct sign bit output and provides a valid conversion to sign magnitude encoding. Case 4 validates the edge case of the smallest negative number (-2048). The logic pertaining to this case is detailed earlier in our Design Description section.

Test Cases 5 and 6 were used to ensure the proper functionality of our priEncode module, which is validated by the correct outputs of E and F for each case. The test cases that were chosen were basic inputs that required no special rounding logic. Therefore, they specifically tested the ability for the module to count the number of leading zeros and extract the leading bits to obtain the exponent and significand respectively.

Test Cases 5 through 9 were used to ensure the proper functionality of our round module, which is validated by the correct outputs of E and F for each case. Cases 5 and 6 required no rounding and, therefore, tests the basic case of keeping the exponent and significand values as is. Cases 7, 8, and 9 required rounding since the fifth bit following the significand bits is a 1 in each case. Cases 7 and 8 test basic rounding, which increments the significand without needing to modify the exponent. Case 9 tests the edge case in which both the significand and the exponent are full. This, like the base case should keep the two values as is.

Finally, Test Case 10 was implemented simply because our old and incorrect implementation of the rounding module was not checking the least significant bit when it was the fifth bit. Therefore, it was used to test the overall correctness of the floating point converter.

Conclusion:

In this lab we successfully designed, implemented, and tested a robust two's complement to floating point convertor in Verilog HDL. Our implementation can take any 12-bit two's complement representation as input and output the corresponding 8-bit floating point representation. Our convertor was composed of three modules: twotosig, priEncode, and round. The twotosig module converts two's complement representation to sign magnitude representation and also extracts the sign of the number. priEncode counts the number of leading zeros of the sign magnitude representation and computes the preliminary exponent and significand values of the floating point representation. The round module inspects the value of the fifth bit following the significand and the preliminary values of the exponent and significand to decide if they need to be rounded. These three individually implemented modules put together form our floating point convertor.

Additionally, from this lab, we learned the limitations and conventions of Verilog design as compared to other programming languages. We also learned how to effectively use the Xilinx ISE software to test our implementation as well as debug our modules individually more efficiently. Our solution fulfilled all the requirements of the lab specifications and covered all special cases. We did not encounter any issues passing the grading testbench.