Lab 2 - Floating Point Conversion

Fnu Shiva Sandesh, Junyoung Kim, and Brandon Tai

May 4, 2017

M152 A - Miodrag Potkonjak

TA: Brinda Alluri

Introduction And Requirement

For this lab, we didn't need to use FPGA to simulate output of the signals, but we were required to design and test a combinational circuit that converts an encoded 12 bit analog signal into compounded floating point using Xilinx ISE to implement the design in Verilog HDL.

In order to convert the encoded 12 bit analog signal into the compounded floating point, we had to follow the table,

FPCVT Pin Descriptions	
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.

Figure 1: FPCVT Pin Descriptions

The above table shows that how to implement an analog number (12 bits) into a digital number (8 bits). The 8-bit number can represent either an unsigned number range from 0 to 255 or a signed number range from -128 to 127 by using two's complement representation. Because 7 or 8 bits of precision is not enough to represent or capture of dynamic range of sounds, we can only use this compounded floating point representation to commercial use.

7	6	5	4	3	2	1	0
S	E			F			

The value represented by an 8-Bit Byte in this format is:

$$\mathbf{V = (-1)^S \times F \times 2^E}$$

The **S**-Bit signifies the **Sign** of the number. The 4-Bit **Significand**, **F**, ranges from [0000] = 0 to [1111] = 15, and the 3-Bit **Exponent**, **E**, ranges from [000] = 0 to [111] = 7. The following table shows the values corresponding to several Floating Point Representations.

Floating Point Representation Examples		
Floating Point Representation	**Formula**	**Value**
[0 000 0000]	0×2^0	0
[1 010 1010]	-10×2^2	-40
[0 011 0111]	7×2^3	56
[0 010 1110]	14×2^2	56

Figure 2: Floating Point Representation Examples

Design Description

Our algorithm design is broken up into four main sections: unsigned magnitude conversion, calculation of E, calculation of F, and rounding computation.

Unsigned Magnitude Conversion: First, the most significant bit is saved in S (sign bit). Then, the absolute value of D is calculated. So, if $D < 0$, D is negated using the formula $D = \sim D + 1$.

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

Table 1: Leading Zeros and Exponent Table

Calculation of E: As shown in the table above, the exponent, E, is directly related to the number of leading zeros. So, E is calculated by counting the number of zeros up to 8, and setting $E = 8 - num\_zeroes$.

Calculation of F: Next, the mantissa, F, is calculated. F is the four most significant bits of the number. So, since the number of leading zeroes was calculated in the previous step, the next four bits are saved as F.

Rounding Computation: Finally, any rounding is computed by first seeing if the 5th most significant bit is a 1 (because then the number must be rounded up). If it is, 1 is added to the value of F. If F was the maximum 0xF, then F overflows to 0x8, and 1 is added to the value of E. if E was also the maximum of 0x7, then F and E are both reverted to their maximum values of 0xF and 0x7 respectively.

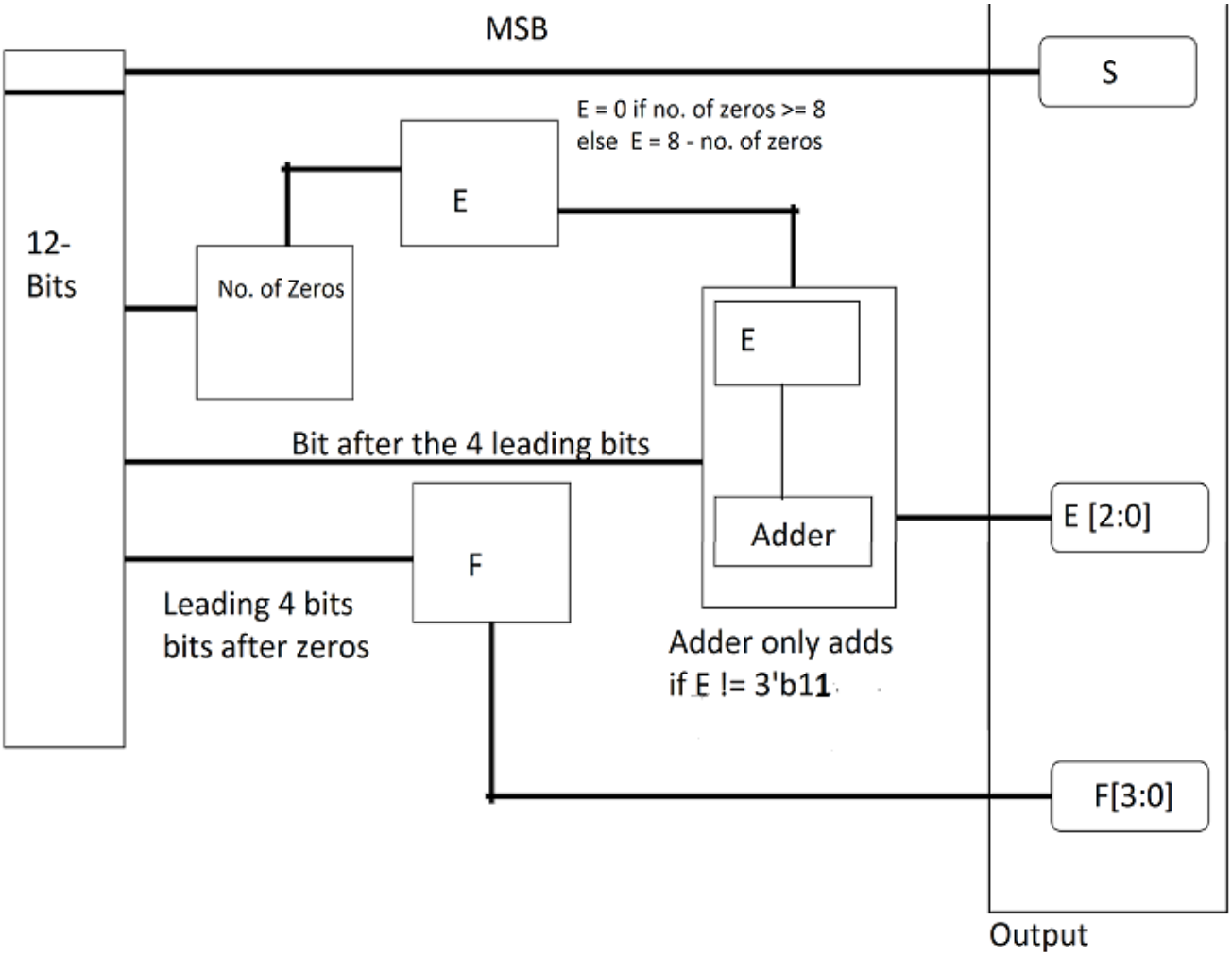

Figure 3: Depicting the logic in modules

Simulation Documentation

For this project we had an extensive test cases which covered a variety of possible 12 bit entries, especially the corner cases, in our test bench code. The first time we ran the code we had errors most of which were because the 'break' statement was not being recognized as a legal command. Therefore, it kept throwing syntax error for it. In order to fix it we had to remove all the break statements in our code. In order to keep the logic same we defined an explicit variable 'brk' which worked as a flag, similar to the break statement. This fixed all of the syntactical errors in the code. However, when we ran our code we found out that our results were not what we were expecting. We realized that there is a pattern in the inputs that was making our test cases fail. So the error was that whenever we had more than seven leading zeros in our input the simulations produced a wrong value of exponent and significant. We realized that the error was caused because of an error in the size of the register we were using to to hold the number of zeros. Initially, the size of the register was 3 bits therefore whenever our input had more than 7 leading zeros it would count up to 7 (3'b111) it would reset to zero.

Since the calculations for the mantissa and exponent depends on the number of zero, thus incorrect number of zeros was resulting in the calculation of the mantissa and exponent to be

wrong as well. When we fixed the results were as we expected. The following table have the test cases we had and the corresponding results we got after the simulations.

12 Bit-Input (D[11:0])	*Sign Bit (S)*	*3 - Bit Exponent(E[2:0])*	*4-Bit Significand (F [3 : 0])*
0000000000000	0	000	0000
000011110000	0	100	1111
000001111000	0	011	1111
100011110000	1	111	1110
111111111111	1	000	0001
011111000000	0	111	1111
000000101110	0	010	1100
000000101101	0	010	1011
000000101110	0	010	1011
000000101110	0	010	1100
000000101111	0	010	1100

Table 2: Test Cases in our Testbench

After this we ran the test-bench code that was provided to us by our TA, the following snippet are the results of the simulation results of our code. When we ran this test bench code, we got zero error the very first time we ran it. This concludes that the logic that was used in our code is correct and it covered all the cases, including all the corner cases.

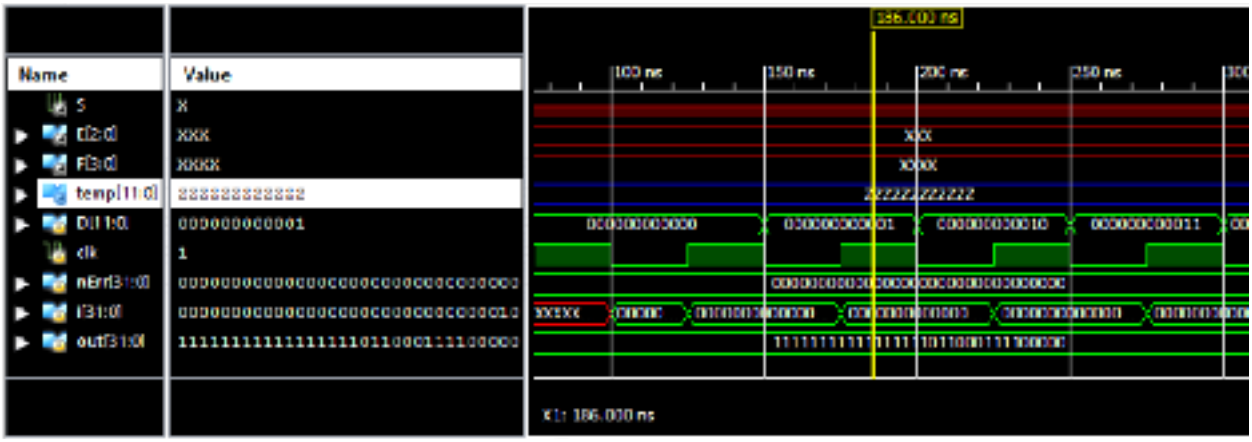

Figure 4: Par1/2 of the test Bench Simulation results. (with test bench provided by TA)

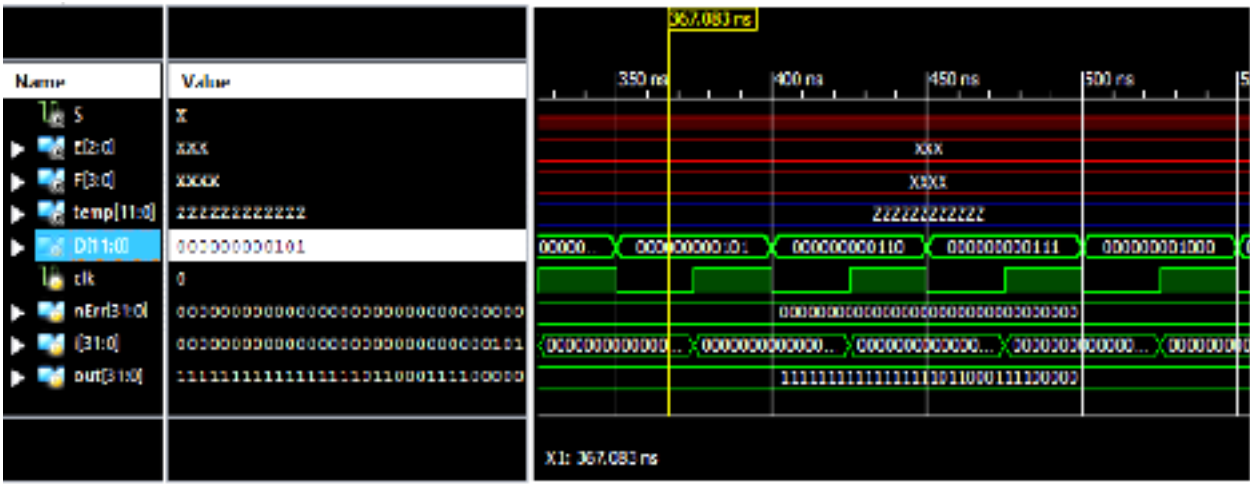

Figure 5: Part 2/2 of the test Bench Simulation results. (with test bench provided by TA)

Conclusion

From this lab exercise, we learned how to convert a 12-bit number into an 8-bit compounded floating point number using the FPCVT Pin Descriptions table. In order to achieve our goal, we designed our circuit using four main parts: two's complement, 3-bit exponent for calculating E, 4-bit significant of F, and rounding computation. Because Verilog HDL doesn't support break statement the way we expected (such as in C/C++ languages), we had to implement our own break statement logic into the circuit. However, it was a learning experience in terms of how the numbers can be modelled in the computer system. We also learned the relation between the number of bits in system and the accuracy with which a number can be represented. Since in this lab, the number of bit with which we were representing 12 bits was just 8 bits, there was a significant amount of rounding in some cases. This error would definitely be detrimental in a real system. This lab was very useful in improving our understanding of the concepts of conversion between different representations.