David Tran, 305109767, Project 1, Lab Section 5

I. Introduction

The goal of Lab1 is to create a Floating-Point converter that converts a 13-bit linear encoding of an analog signal to a 9-bit Floating-Point(FP) Representation. The motivation behind Floating-Point Representation is that linear encoding are not as useful in many applications due to their limited range. Floating-Point Representation enables hardware to represent a more dynamic range of values. The Floating-Point Representation contains a 1-bit Sign Representation, 3-Bit Exponent, and 5-Bit Significand.

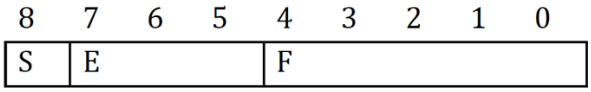


Figure 1: *9-bit Floating-Point Representation* – 1-bit Sign Representation, 3-Bit Exponent, 5-bit Significand

The Floating-Point Representation can be converted to a real number representation (V) in the following manner.

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

At the top level, the floating-point converter (FPCVT) can be represented with the following module.

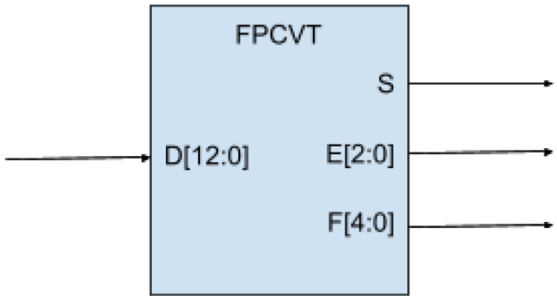


Figure 2: *Top-level design of FPCVT* – Input: 13-bit linear encoding, Output: 1-bit Sign Representation, 3-bit Exponent, 5-bit Significand.

II. Design

The underlying design for the converter takes inspiration from the recommended design in the lab details. The input is handled in three stages. In the first stage, the 13-bit signal is converted from its two's complement representation to a signed-magnitude representation. This extracts the 1-bit Sign Representation. In the second stage, the magnitude of the 13-bit signal is processed to yield the preliminary 3-bit Exponent and 5-bit Significand. Additionally, a 6th bit is extracted to determine rounding. The third stage handles the rounding logic and returns the final 3-bit Exponent and 5-bit Significand.

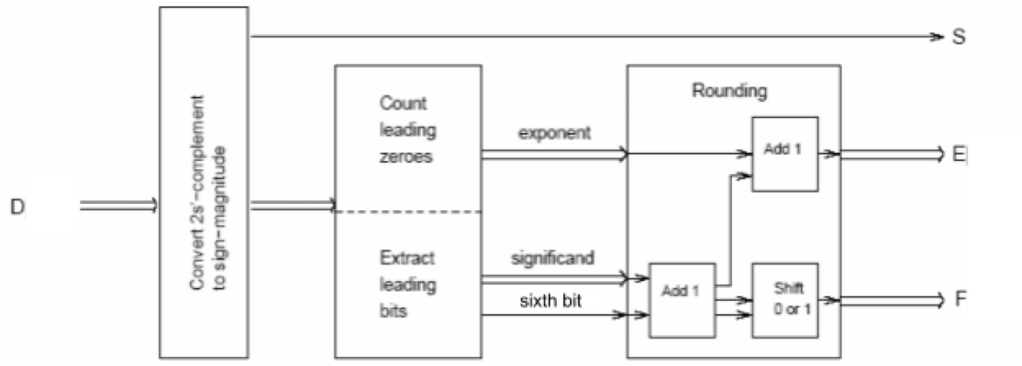


Figure 3: *3-Stage Design of FPCVT* – Stage 1: Convert 2's complement to sign-magnitude, Stage 2: Extract preliminary exponent and significand with 6th bit, Stage 3: Handle rounding logic

In the Verilog hierarchy structure, FPCVT is the top-level module and instantiates 3 submodules to handle all its logic: sign_extractor, float_converter, and rounder.

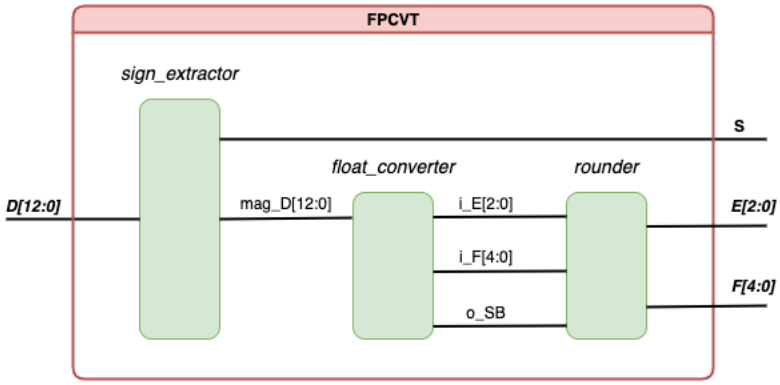


Figure 4: *Verilog Structure of FPCVT* – 3 submodules: sign_extractor, float_converter, and rounder

III. Implementation

a. Sign Extractor

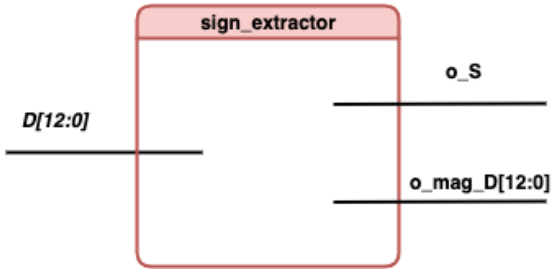


Figure 5: *Sign Extractor Submodule*

The sign bit o_S comes easily as it is simply the 13th bit (D[12]) of the input. The magnitude requires some combinatorial logic, because the two's complement is handled differently for

positive versus negatives values. For positive values, the most-significant-bit (MSB) is 0 and we do not need to do anything. For negative values, the MSB is 1 and we need to invert the bits and add 1 to obtain the magnitude of the value.

$$o\_mag\_D = \sim D + 1$$

Figure 6: *Two's complement of D*

One edge case that the sign_extractor is designed around is the case of the most negative number (1_0000_0000_0000). Because two's complement representation is asymmetric, we cannot adequately represent the most negative value in signed magnitude value. Instead, when the extractor encounters this special case, the extractor assigns the o_mag_D to be 0_1111_1111_1111.

b. Float Converter

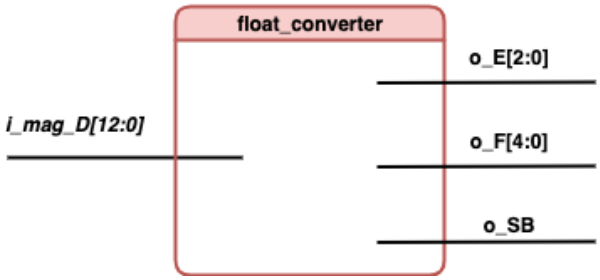


Figure 6: *Float Converter Submodule*

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

Table 1: *Exponent Encoding*

The exponent is related to the number of leading 0's in front in the magnitude and is encoded in the following table above. The significand is 5 bits following the last 0, and the rounding bit (o_SB) is the 6th bit after the 0. For bit values with exponent equal to 0, the significand are the 5 least significant bits, and the rounding bit is 0.

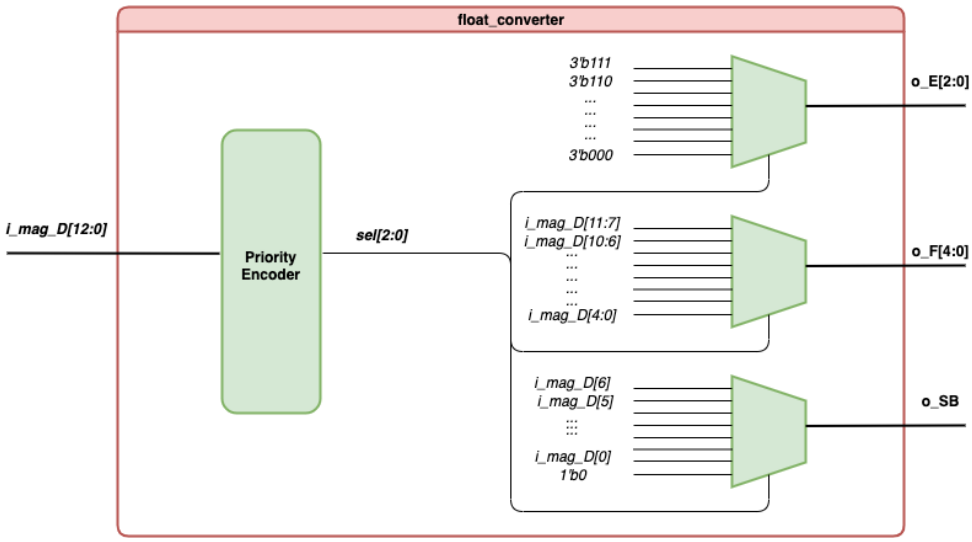


Figure 7: *Detailed implementation of float_converter* – Priority encoder converts 13-bit input to 3-bit selection bits for multiplexers

The float_converter is implemented as a priority encoder that takes in the 13-bit input and converts to a 3-bit output that is used as the selection bits for 3 multiplexers, each corresponding to one respective output. This works because the o_E, o_F, and o_SB can only take on 8 different combination of values.

c. Rounder

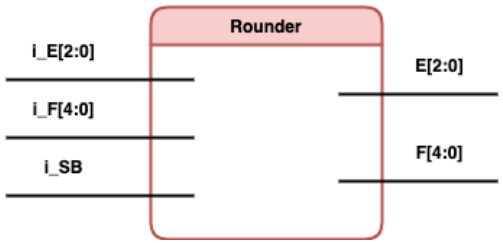


Figure 8: *Rounder Submodule*

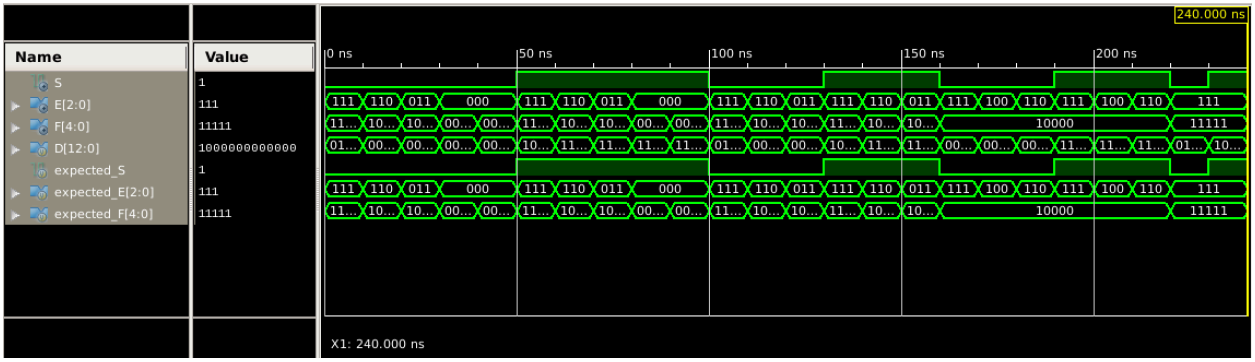
Rounder implements the logic for rounding. The rounder adds i_SB to i_F[4:0]. The overflow bit of the sum is used to apply a right shift to the value sum of the two. This resulting value of this process is F[4:0]. Additionally, the overflow bit of the sum is added to i_E[2:0] to obtain E[2:0]. In the edge case where this leads to an overflow in exponent, both E and F are set the maximum possible value represented by Floating-Point Representation (E = 3'111, F=5'b111111).

d. Top Level: FPCVT

The top level module, FPCVT, simply instantiates the 3 submodules and connects their wire to obtain its output. This is shown in detail in Figure 4.

IV. Testbench

To facilitate the iterative approach that I took in developing the project, I developed testbenches for each module along the way. In doing so, I was able to catch mistakes and correct them early on in the process. For the final testbench of the top module (FPCVT), I aggregated all the tests I had developed through the process into one large test. The results are as follow.



Screenshot 1: *Waveform for test of FPCVT* – 24 tests each 10 ns intervals; tests encompass 16 common use cases and 8 edge cases

```
time=          0, D=0110100100000, S=0, E=111, F=11010, expected_S=0, expected_E=111, expected__F=11010
time=         10, D=0010101000000, S=0, E=110, F=10101, expected_S=0, expected_E=110, expected__F=10101
time=         20, D=0000010100000, S=0, E=011, F=10100, expected_S=0, expected_E=011, expected__F=10100
time=         30, D=0000000000010, S=0, E=000, F=00010, expected_S=0, expected_E=000, expected__F=00010
time=         40, D=0000000000001, S=0, E=000, F=00001, expected_S=0, expected_E=000, expected__F=00001
time=         50, D=1001011100000, S=1, E=111, F=11010, expected_S=1, expected_E=111, expected__F=11010
time=         60, D=1101011000000, S=1, E=110, F=10101, expected_S=1, expected_E=110, expected__F=10101
time=         70, D=1111101100000, S=1, E=011, F=10100, expected_S=1, expected_E=011, expected__F=10100
time=         80, D=1111111111110, S=1, E=000, F=00010, expected_S=1, expected_E=000, expected__F=00010
time=         90, D=1111111111111, S=1, E=000, F=00001, expected_S=1, expected_E=000, expected__F=00001
time=        100, D=0110101100000, S=0, E=111, F=11011, expected_S=0, expected_E=111, expected__F=11011
time=        110, D=0010101100000, S=0, E=110, F=10110, expected_S=0, expected_E=110, expected__F=10110
time=        120, D=0000010100100, S=0, E=011, F=10101, expected_S=0, expected_E=011, expected__F=10101
time=        130, D=1001010100000, S=1, E=111, F=11011, expected_S=1, expected_E=111, expected__F=11011
time=        140, D=1101010100000, S=1, E=110, F=10110, expected_S=1, expected_E=110, expected__F=10110
time=        150, D=1111101011100, S=1, E=011, F=10101, expected_S=1, expected_E=011, expected__F=10101
time=        160, D=0011111100000, S=0, E=111, F=10000, expected_S=0, expected_E=111, expected__F=10000
time=        170, D=0000011111100, S=0, E=100, F=10000, expected_S=0, expected_E=100, expected__F=10000
time=        180, D=0001111110100, S=0, E=110, F=10000, expected_S=0, expected_E=110, expected__F=10000
time=        190, D=1100000100000, S=1, E=111, F=10000, expected_S=1, expected_E=111, expected__F=10000
time=        200, D=1111100000100, S=1, E=100, F=10000, expected_S=1, expected_E=100, expected__F=10000
time=        210, D=1110000001100, S=1, E=110, F=10000, expected_S=1, expected_E=110, expected__F=10000
time=        220, D=0111111111111, S=0, E=111, F=11111, expected_S=0, expected_E=111, expected__F=11111
time=        230, D=1000000000000, S=1, E=111, F=11111, expected_S=1, expected_E=111, expected__F=11111
```

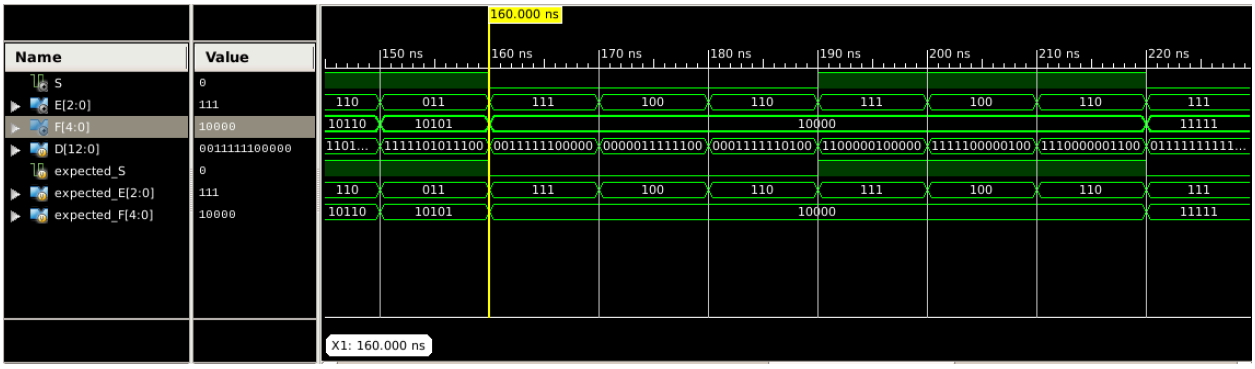
Screenshot 2: *Console output of results from testbench*

a. Common Cases

The first 16 cases were used to test the common use cases for the floating-point converter. The first 10 cases test for conversion of both positive and negative values with no rounding bit. The next 6 test for conversion with an addition rounding bit, but not overflow into the exponent. For these cases, we can note that the significand is increased by 1.

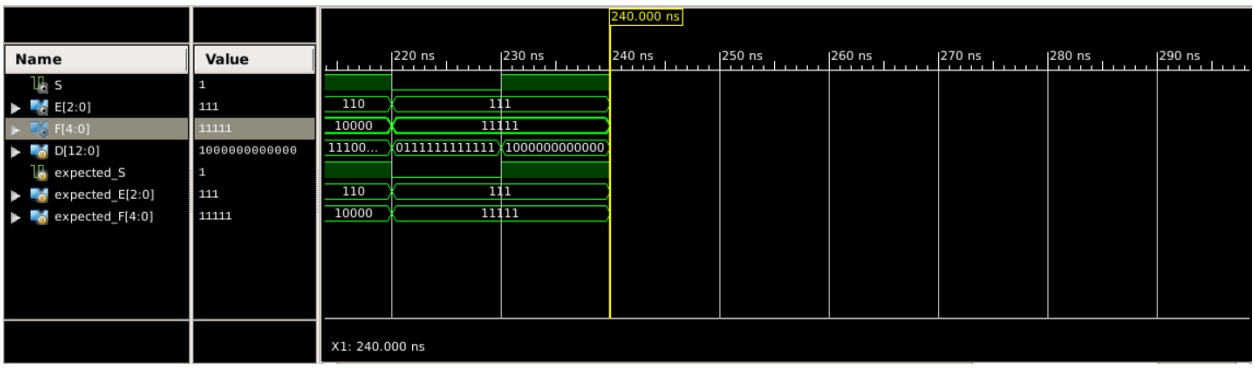
b. Edge Cases

For our edges cases, we have 6 tests for rounding with overflow into the exponent – 3 for positives and 3 for negatives. Note that in these tests the expected value for the significand is always 5'b10000. Additionally the exponent is increased by one.



Screenshot 3: *Rounding overflow in exponent, (160 ns – 220 ns)* – 6 tests for rounding overflow in exponent that results in adding 1 to exponent and significand to be 1'b10000

The last 2 edge cases test the two extremes: the largest negative value (1_0000_0000_0000) and the largest positive value (0_1111_1111_1111). In both these cases, the exponent should be 3'b111 and the significand should be 5'b11111.



Screenshot 4: *Largest negative and positive values, (220 ns – 240 ns) – We can see that both yield the largest possible 9-bit Floating-Point Representation*

V. Conclusion

When I implemented the design, the project synthesizes with two warnings. However, these warnings are not really source for concern because they are about bit truncation. These resulted from me assigning largest bits arrays to smaller bit arrays to obtain the least significant bits.

FPCVT Project Status (10/25/2020 - 10:48:42)			
Project File:	Project1.xise	**Parser Errors:**	No Errors
Module Name:	FPCVT	**Implementation State:**	Placed and Routed
Target Device:	xc6slx16-3csg324	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	2 Warnings (0 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Screenshot 4: *Design Summary* – 2 warnings from bit truncation.

Lab 1 taught me the value of thinking modularly in Verilog and developing comprehensive testbenches to catch edge cases. What I found interesting about the Floating-Point Representation was that it cannot represent some numbers accurately. It would be interesting to see how accurately numbers are represented with a largest bit-sized Floating-Point Representation.