**CS 152A**

**Introductory Digital Design Lab**

**LAB #2**

**Floating Point Conversion**

**February 9, 2015**

**Grade:**

-----------

**Farhad Shahmohammadi**

**Name 1: Anthony Pimentel**

**SID 1: 004019166**

**Name2: Jason Tieu**

**SID2: 304047667**

**Introduction and requirement**

The goal of this experiment was to take a 12-bit analog signal (a number in Two’s Complement Representation (D)) and convert it into an 8-bit Floating Point number. This is relevant to real life because non-linear encodings like Floating Point allow us to represent larger ranges of numbers with fewer bits than linear encodings require, which is useful for compression.

Floating Point format consists of a 1-bit Sign **S** (set to 0 for positive numbers and 1 for negative ones), a 3-bit Exponent **E** (ranging from 0 to 7), and a 4-bit Significand **F** (ranging from 0 to 15). A number in this format can be translated into an integer using the equation .

Unfortunately, some numbers cannot be represented in that format, so we are also instructed to round the result to the nearest floating point representation if we cannot get the exact value.

Lastly, the sizes of the Exponent and Significand put upper and lower bounds on the numbers we can represent. In this case, the smallest number we can represent is -1920 () and the largest is 1920. If we receive input greater than 1920, we are instructed to return 1920 as the result and if it is less than -1920, we are supposed to return -1920.

**Design Description**

We designed the architecture of the Floating Point converter using 4 main modules.

*Convert2CToSM:*

The first module converts the Two’s Complement input (D) into Sign-Magnitude form. This gives us the final output for the S bit and a Magnitude that represents the positive version of the input. We get the magnitude by XOR-ing the input with the most significant bit and adding the most significant bit to the result. This flips the bits and adds 1 when the number is negative and does nothing when the number is positive. There is one case where the magnitude still ends up being negative (when D is -2048), but we handle this in later modules.

*CountLeadingZeros:*

The second module counts the leading zeros in the Magnitude and outputs the first-pass Exponent. We count the leading zeros by simulating a priority encoder (finding the position of the most significant non-zero bit). Then we used the table in the spec (shown below) to map each number of leading zeros to the corresponding Exponent. In the case that the number of leading zeros is 0, we know that the input was -2048, so we set the Exponent to its max value (7 (3’b111)) because we are going to return the most negative number at the end.

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

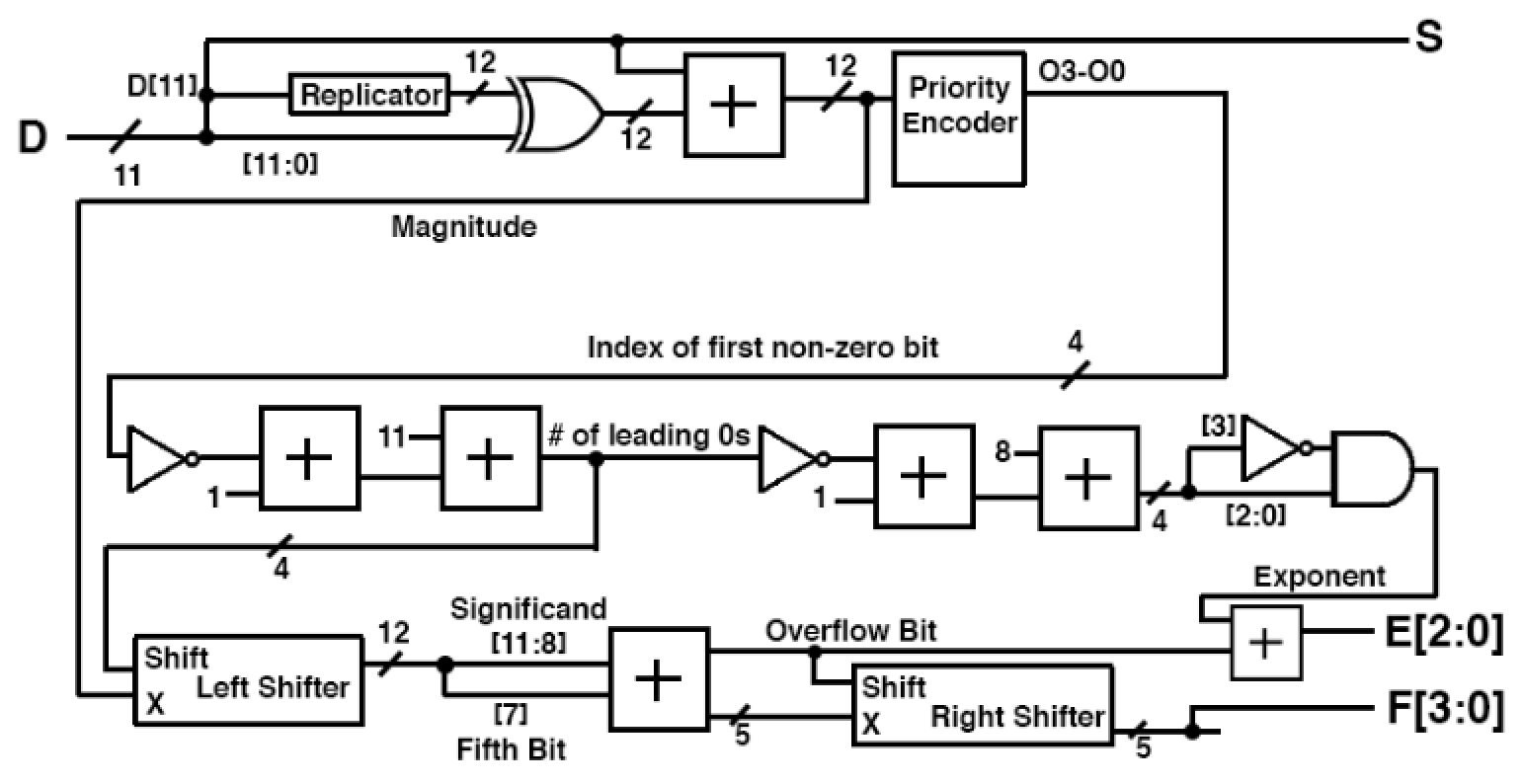
*ExtractLeadingBits:*

The third module extracts the first 4 bits (after the leading zeros) from the Magnitude to output the first-pass Significand and it also extracts the 5th bit (after the leading zeros) for use in the next module. It does this by left-shifting the Magnitude by the number of zeros and then using the 4 most significant bits for the first-pass Significand and the 5th most significant bit for the 5th bit. There are two special cases in this module. When the number of leading zeros is ≥ 8, we are supposed to use the least significant 4 bits of the Magnitude as the Significand and 0 as the 5th bit. Also, when the number of leading zeros is 0, we know that the input was -2048, so we set up the Significand to be the maximum value (15 (4’b1111)) and the 5th bit to be 1 to guarantee that we return the most negative number in the next module.

*Rounding:*

The final module rounds the Exponent and Significand based on their own values and the value of the 5th bit. Usually, this just involves adding the 5th bit to the first-pass Significand. However, there are some special cases. If the Significand is 15 (4’b1111) and the 5th bit is 1, we overflowed, so the final Significand (F) is usually 8 (4’b1000). However, if the Exponent is also 7 (3’b111), then we return the maximum value we can represent (Exponent = 7 (3’b111) and Significand = 15 (4’b1111)). If the Sign bit S was negative, then this is the most negative value we can represent.

*The schematics for the system architecture*



**Simulation Documentation**

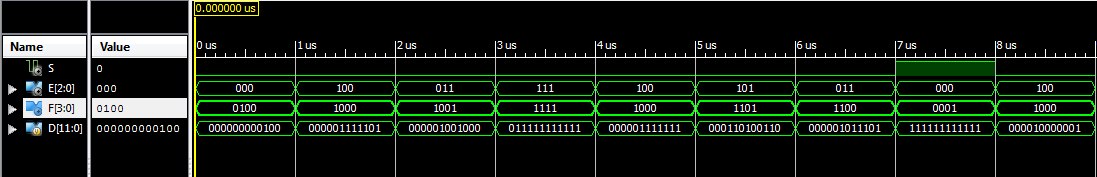
We designed our testbench to print out the sign (S), exponent (E) and Significand (F) for each 12-bit linear encoding. We were given input to use for our testbench. The following waveforms correspond to the nine linear encodings we were given to test with.

**D: 100**

**S: 0**

**E: 000**

**F: 0100**

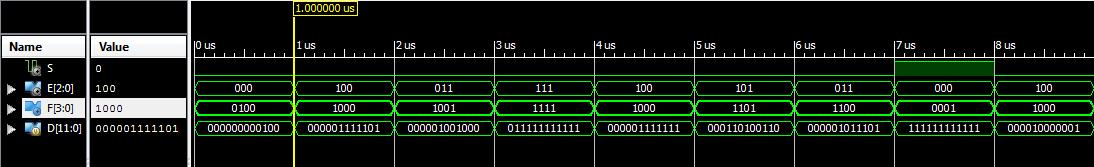


**D: 000001111101**

**S: 0**

**E: 100**

**F: 1000**

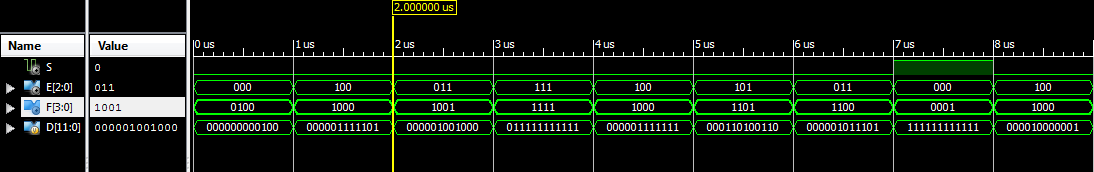


**D:** **01001000**

**S: 0**

**E: 011**

**F: 1001**

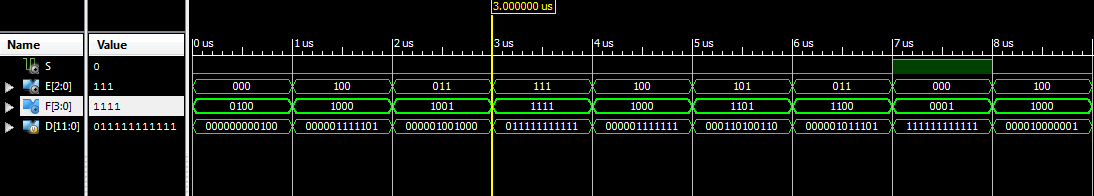


**D: 011111111111**

**S: 0**

**E: 111**

**F: 1111**

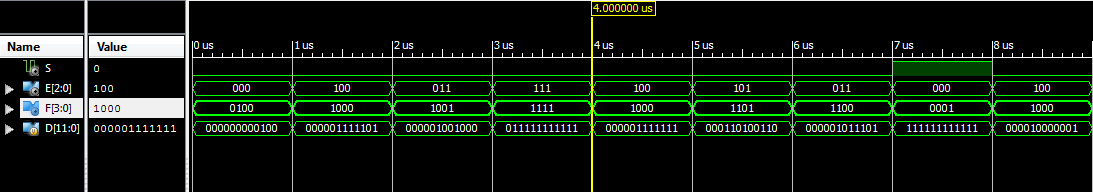


**D: 01111111**

**S: 0**

**E: 100**

**F: 1000**

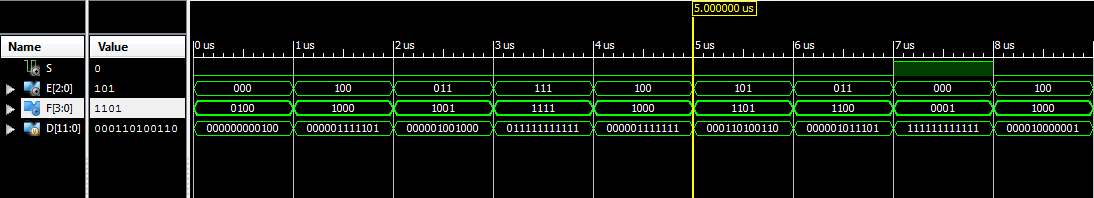


**D: 000110100110**

**S: 0**

**E: 101**

**F: 1101**

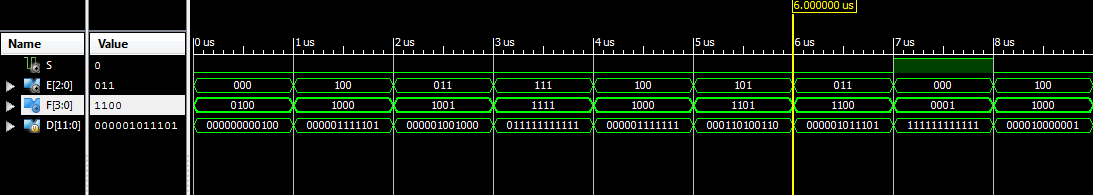


**D: 01011101**

**S: 0**

**E: 011**

**F: 1100**

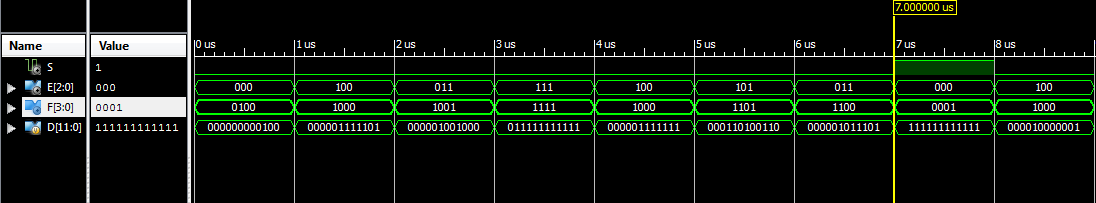


**D: 111111111111**

**S: 1**

**E: 000**

**F: 0001**

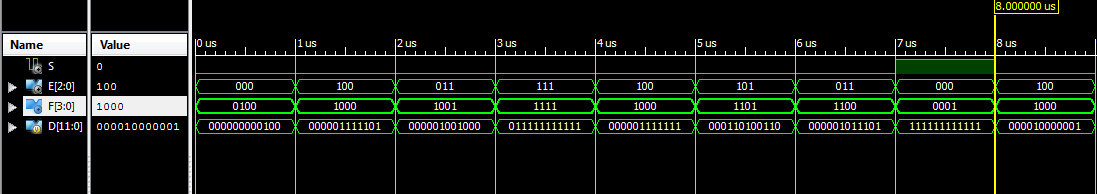


**D: 10000001**

**S: 0**

**E: 100**

**F: 1000**



**Conclusion**

We had some issues figuring out how to implement our design circuit into Verilog code. Since Verilog is a HDL, using if and loop statements are tricky. Overall, we were able to write a straightforward solution that gave us the correct outputs.

**Individual Contribution**

Both of us were present during all lab times and worked on the assignment as a team. We each contributed 50% to the lab and report. Anthony did the first half of the report, while Jason did the second half.