CS M152A Lab 1 Report

Jonathan Chau

UID: 705-166-732

TA: Boyuan He

January 24, 2021

1. Introduction and Requirements

In this lab, I want to demonstrate that we can write and implement Verilog code using the Xilinx ISE Software. I show this skill by implementing a simple floating point converter module using a combinational circuit. This circuit takes a 13-bit analog input in two's complement representation (D), and converts it into a 9-bit floating-point output. This output consists of a 1-bit sign representation (S), a 3-bit exponent representation (E), and a 5-bit significand representation (F). The resulting value V represented in our floating-point format can be summarized using this equation below:

$$V = (-1)^S * 2^E * F$$

Note that this value V does not accurately represent D and only represents the floating-point number closest to the original input. There are some requirements as to how the floating-point number should be represented. Preferably, the representation should be normalized, where the most significant bit of the significand is 1. More details will be explained in later sections.

2. Design Description

The module consists of a single 13-bit input (D) and three outputs: the sign (S), the exponent (E), and the significand (F). To streamline the design process, I broke the module into three separate parts. These parts were then connected to form a top module that I will test using a testbench. The project description summarizes up the design of the overall module as shown in Figure 1.

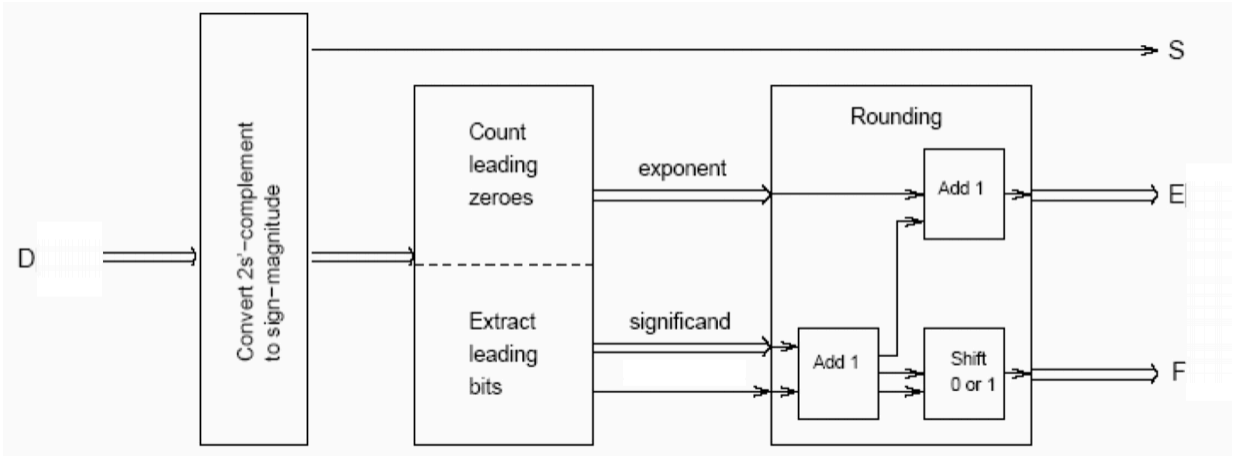

Figure 1. Schematic design of the top module

The first sub-module takes in the 13-bit input (D) and converts it into a sign-magnitude representation. This output is represented by a 1-bit sign (S), and a 13-bit magnitude (output_data). The overall design is simple. First, extract the most significant bit of D and call its output S. If S is zero, then we leave D as is and call it the output data, or the magnitude. If S is one, then we take the two's complement of D and call it the magnitude. We can obtain D's two's complement by flipping all of the bits and adding one to the result. One edge case we have to consider is when the input is -4096. In this case, the sign S is 1 and the magnitude is 4096, but we can't represent this number using the remaining 12-bits as the maximum value with this limitation is 4095. To resolve this, we let the magnitude of -4096 become 4095, the closest magnitude possible. We can achieve this by comparing the two's complement of -4096 to -4096.

If the numbers match, we feed the result to the multiplexer to select 4095 as the resulting magnitude. The design for the first sub-module is shown in Figure 2.

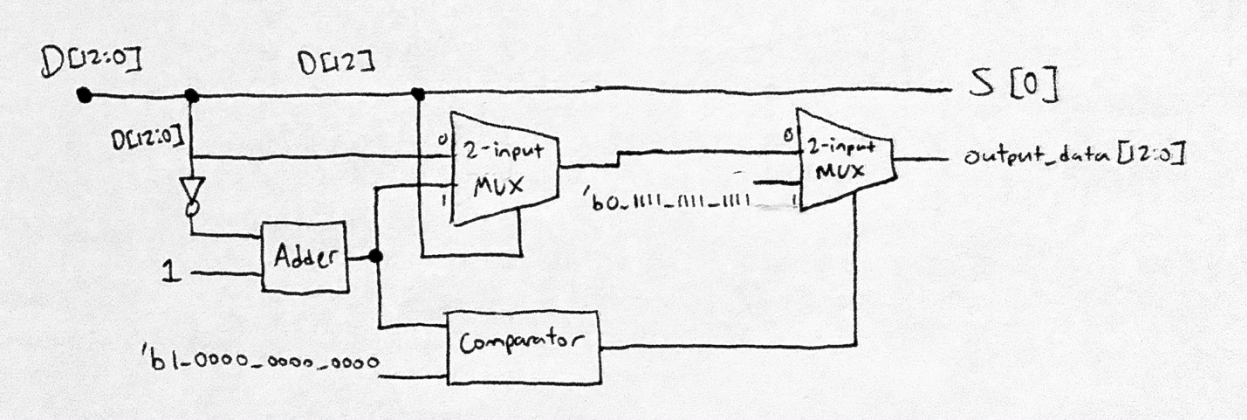

Figure 2. Schematic design of the Two's-complement to Sign-magnitude

The second sub-module takes in the 13-bit magnitude (output_data) as the input, and extracts the first five bits and a possible sixth bit to help round to the nearest floating point representation. The outputs are the 3-bit exponent (exp) the 5-bit significand (sig) and the 1-bit sixth bit. We get the exponent (exp) by placing output_data[11:4], or o[11:4] into a priority encoder. Zero is returned to the exponent if only o[4] is 1, or all of the eight bits are zero. To get the significand (sig), we make 8 masks that cover 5-bits of the magnitude and feed them to the first multiplexer. The value of the exponent selects which of the eight inputs gets passed as the significand (sig). This is equivalent to my Verilog implementation where I right shifted the input by the value represented by the exponent (exp) and taking the last five bits as the significand (sig). The sixth bit is obtained by extracting the sixth bit after the leading zeros. However, if the exponent is zero, the sixth bit is zero because the significand takes up the five least significant bits, leaving the sixth bit with nothing. The design for the second sub-module is shown below in Figure 3.

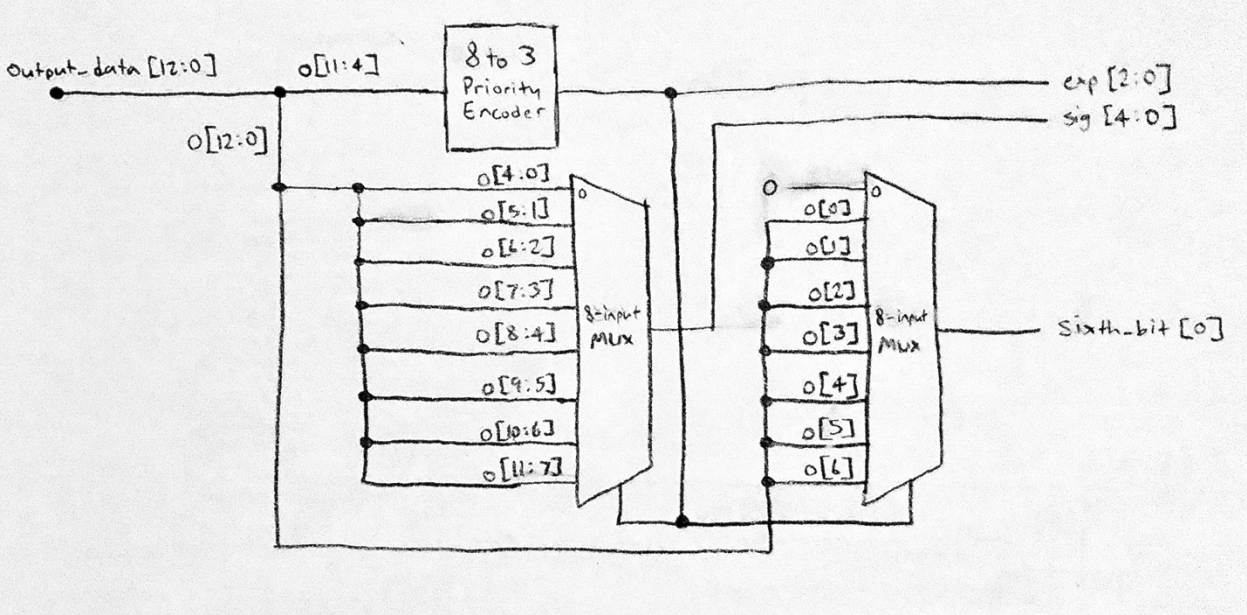

Figure 3. Schematic design of the counting leading zeros and extracting leading bits module

The third sub-module takes in three inputs, the exponent, the significand, and the sixth bit, and adds one if the sixth bit is one. The outputs produced are the exponent (E) and the significand (F). If adding one to the significand causes it to overflow, add one to the exponent and set the significand to [10000]. If adding 1 cause both the significand and exponent to overflow, set the significand and exponent to their maximum values and return them as F and E respectively. The design for the third sub-module is shown below in Figure 4.

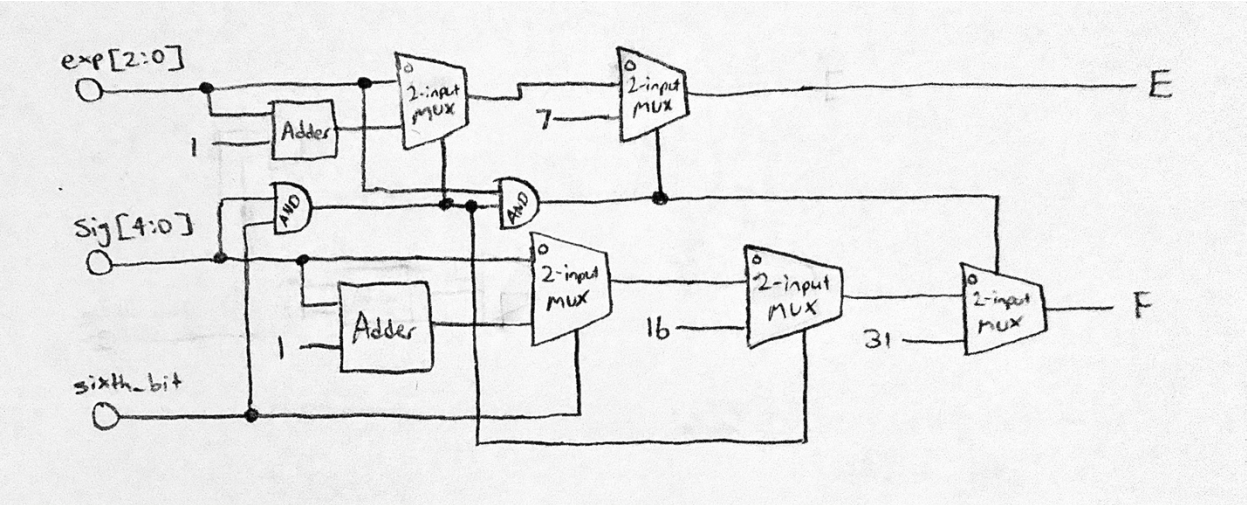

Figure 4. Schematic design of the rounding module

3. Simulation Documentation

Testing was conducted using the Xilinx ISE environment. I first tested the sub-modules to see if they are working properly before simulating the top module for a final check. To smoothen the testing process, the initial and final inputs are set to 0. Starting at 100ns, each test case below lasts 10 nanoseconds before going over to the next test case.

3.1 Sub-Module Tests

3.1.1 Two's-Complement to Sign-Magnitude Converter

To check for basic functionality, I gave random test inputs to this submodule. As seen below in Figure 5, the nonnegative numbers, which are 0, 1, and 4095, all produce the expected sign output 0. For negative numbers -4096, -4095, -27, and -1023, the sign bit returns the expected value of 1.

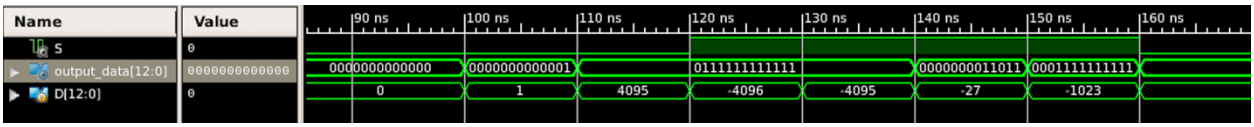

Figure 5. Waveform Simulation of the Two's-Complement to Sign-Magnitude Converter Sub-Module

For the magnitude I have to inspect each test case individually to confirm that the sub-module is working. 0 should get 'b0, and -27 should get 'b11011 as the magnitude. These results pass and are shown in Figure 5. The special case I need to consider is when D = -4096. I have said that the sign bit is 1, but the most positive number represented in the signed 13 bits is 4095, so the resulting output_data should return [0_1111_1111_1111]. The above figure confirms this special case and thus, the first sub-module is fully functional.

3.1.2 Leading Bits Extractor

In this sub-module, the input is limited to positive numbers 0 to 4095. Luckily testing this module is easier than the previous module. We check the significand by comparing the sig output to the first five bits after the leading zeros and see if they match. We also check if the sixth bit

after the leading zeros match with the resulting sixth_bit, or that it is zero if the significand occupies the five least significant bits. Checking if the exponent is correct is a bit harder, as we get the value by subtracting eight from the number of leading zeros. If there are eight or more leading zeros, set the exponent value to zero. Figure 6 below shows the results.

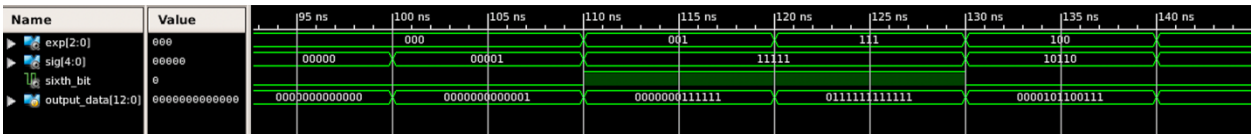

Figure 6. Waveform Simulation of the Leading Bits Extractor Module

The latter three cases check out, in both cases. The ones we have to worry about are when the input is 0 or 1. Thankfully, those cases check out as well, as both the exponent and sixth_bit are displayed as 0 and the significand matches the last five bits.

3.1.3 Rounding Sub-Module

For the third and final sub-module, I simply check to see if the sixth_bit adds one to the significand or if it does not. In a normal case, where the exponent is ‘b010 and the significand is ‘b01010, if the sixth_bit is zero, then E takes the value of the exponent input, ‘b010, and F takes the value of the significand input, ‘b01010. If the sixth_bit was one, then F adds one to the significand input and returns the sum, ‘b01011, and the exponent remains unchanged, as the significand did not overflow. As seen in the other cases where the sixth_bit is zero, the output E is identical to the input exp, and the output F is identical to the input sig. Figure 7 illustrates these results.

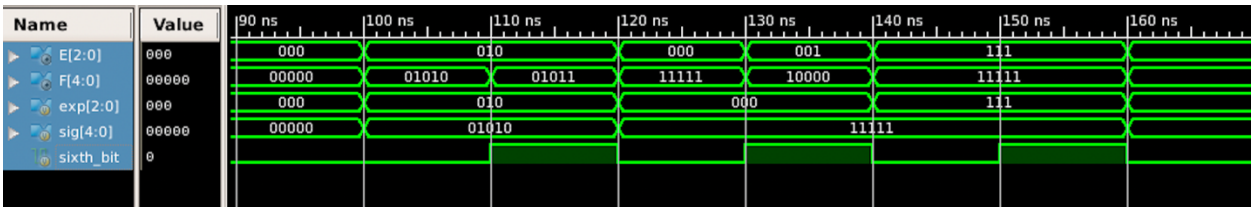

Figure 7. Waveform Simulation of the Rounding Sub-Module

However, there are two instances where the significand is ‘b11111 and the sixth_bit is 1. In this case, the output F will return 10000 and the exponent is incremented by one to produce the new output E. This is seen in the case at 130 ns. However, the case at 150 ns shows when the exponent is ‘b111, the significand is ‘b11111, and the sixth_bit is 1. Overflowing the exponent would result in E and F becoming 0 and 10000 respectively, which does not accurately represent the input. We want their values to be the largest possible, so E takes ‘b111 and F gets ‘b11111.

3.2 Top Module Test

After verifying that the three sub-modules produce correct outputs, it is time to put the combined top module to the test. As expected, every nonnegative number returns 0 as their sign bit S, and every negative number returns 1 for that bit. The test cases at 90, 100, 110, and 120ns test for basic functionality. At 130 ns, the sixth bit is tested. The special case -4096 should return the same result as -4095, since 4096 adopts the magnitude of 4095. Finally, at 170ns, we test the input -2047. As expected, the signed bit is 1, since -2047 is negative. However, the magnitude, 2047, is represented as ‘b0_0111_1111_1111. We count two leading zeros and get 6 as our exponent value. The significand is ‘b11111, and the sixth_bit is also 1. Since the significand overflows, the resulting significand is now ‘b10000, as correctly observed. We also add one to the exponent to make the number 7, or ‘b111, as seen below. Figure 8 illustrates these results.

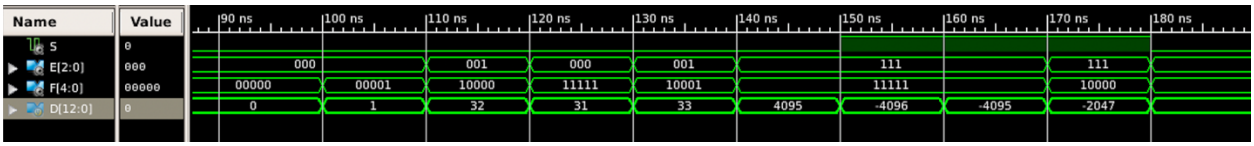

Figure 8. Wave Simulation of the Overall Floating Point Converter Module

One bug that I found during this lab is when the sixth_bit would always get the least significant bit if the exponent was 0. I adjusted the Verilog code a bit so that the sixth_bit is always zero when the exponent (exp) is zero.

4. Conclusion

By completing this lab, I learned how to familiarize myself with Verilog and the Xilinx ISE by building a module and testing its functions using the testbench files. I was able to develop my module using three separate parts that come together to compress a 13-bit input to a 9-bit floating-point representation. These three parts include the Two's Complement to Sign-Magnitude Converter, the Leading Bits Extractor, and the Rounding Module.

I faced some hardships while doing this project, including how to start implementing the modules and how I should streamline my design-thinking process, as I wasn't very familiar with the electrical engineering mindset. With the help of my TA, I was able to resolve my issues and finish my project by thinking like a computer scientist and reviewing previous slides presented during lecture.