Shikha Mody (604922996)

Brendon Ng (304925492)

Armaan Singh (404932331)

COM SCI M152A Lab 5

TA: Logan Kuo

Lab 1 Report

Floating Point Conversion

Introduction

In this lab, we were tasked with creating a converter that would take a 12-bit input in two's complement and convert it to an 8-bit output in floating point format, with sign, exponent, and fraction (mantissa) components. This lab would be only done for simulation (rather than implementation). This way, we could see all possible combinations of inputs in a more organized and clear manner. The purpose of this lab was to not only expand on our Verilog and FPGA implementation skills, but also to make a floating point representation of an otherwise uninterpreted stream of data. The overview of design requirements, as provided in the Lab 1 specification, are as follows:

- One 12-bit input: D[11:0]
- Three outputs:
    - S - sign bit, indicating if the 12-bit encoding is positive or negative
    - E[2:0] - 3-bit exponent, with a maximum value of 7 (3'b111)
    - F[3:0] - 4-bit fraction, with a maximum value of 15 (4'b1111)

These values combine to form a floating point representation of the 12-bit input according to the following formula:

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

One of the challenges with this task was that because the input had more bits than the output, there could be one floating point representation that maps to several 12-bit linear encodings. As outlined in the specification, this is a challenge that comes with value compression.

Other features we had to implement were rounding and a priority encoder. A value got rounded if it did not have an exact floating point representation with the constraints given in this lab; it must be mapped to the nearest floating point value. A priority encoder was the mechanism to determine the exponent (E) value for a given input. It determined the number of leading zeroes and assigned a value to the exponent based on this. The design steps we needed to implement to achieve a floating point representation were as follows:

1. Initially convert the two's complement input to signed magnitude
2. Count the leading zeroes of the signed magnitude number and assign the corresponding exponent value (E) using a priority encoder
3. Calculate the value of the fraction by right shifting the bits in positions 0 through 7
4. Round if necessary: Based on the fifth bit after the last leading zero of the current floating point representation, either round up or keep it the same (round up: increment fraction by 1 – if it overflows, increment the exponent (E) value by 1 and right shift the fraction by one bit)

In summary, the required components of this design were: the three appropriate outputs and all relevant components to achieve the correct floating point representation.

This background information was adapted from the Lab 1 specification.

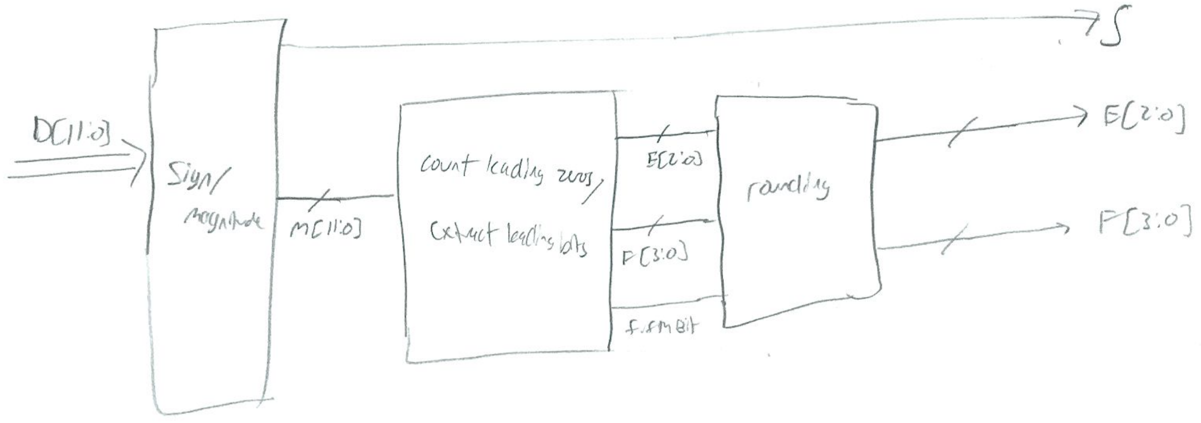
Design Description

Our design for the floating point conversion tool relied on the implementation of a single module, which took as input the 12-bit binary number, as a wire, to be converted, and outputted the floating point value with 3 output registers for the significand, exponent, and sign. To implement this, we first converted the two's complement representation to signed magnitude, then after counting the leading zeros, the significand and fifth bit were stored, and finally the floating point value was rounded based on the fifth bit's value.

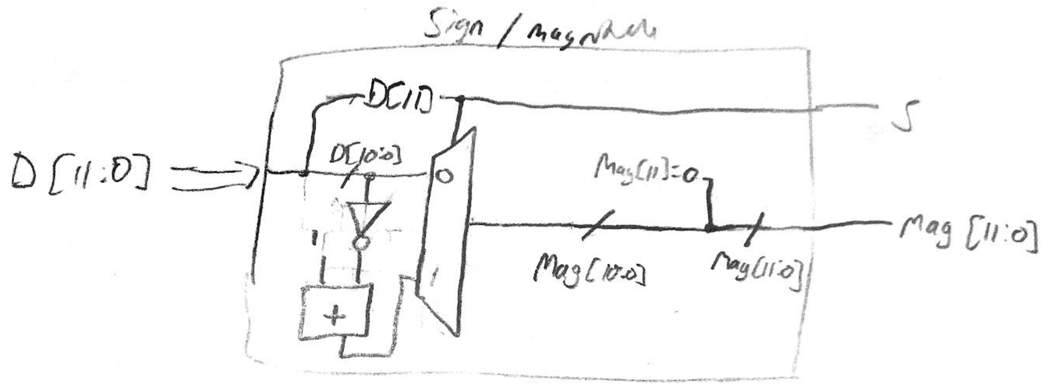
The conversion from two's complement to signed magnitude was fairly simple. By checking the most significant bit in the original number, we knew whether it was positive or negative, and if positive we did not modify the number. Otherwise, the input was converted to signed magnitude by negating the bits and adding one. The sign bit was also stored separately to be outputted.

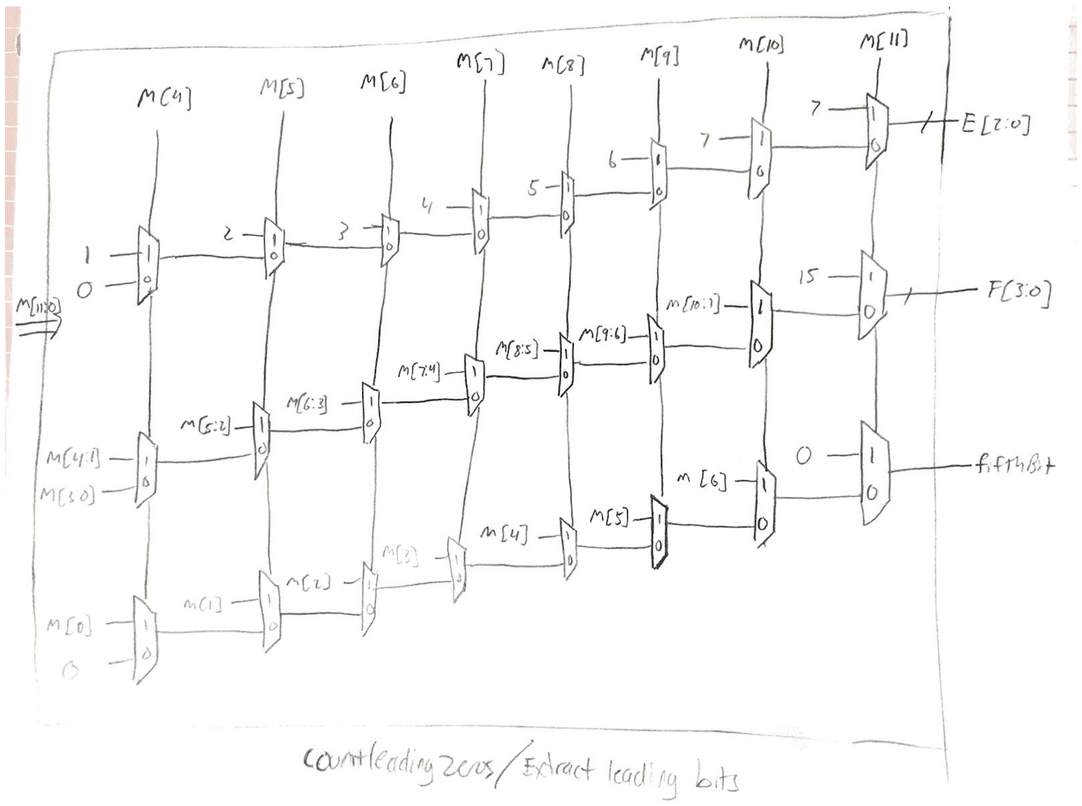
To count the leading zeros, we implemented a priority encoder using a nested if statement which would check the bits one at a time, starting with the most significant bit. If it is zero, then check the next bit, and so on until either a one bit is found, or there are at least 8 leading zeros, which would correspond to an exponent of 0 in the floating point value. Also, when the last leading zero is reached, the next 4 bits are stored as the significand value for the floating point number, and the fifth bit value is stored to be used later in rounding.

In rounding the decimal, the main case is to increment the significand if the fifth bit is one. However, should the significand overflow (i.e. the 4 bit value stored in the register is 4'b1111), then instead the exponent should be rounded up and the significand set to 8. This ensures the floating point representation is normalized, which prefers that the most significant bit of the significand is 1 whenever there are multiple floating point representations of a number.

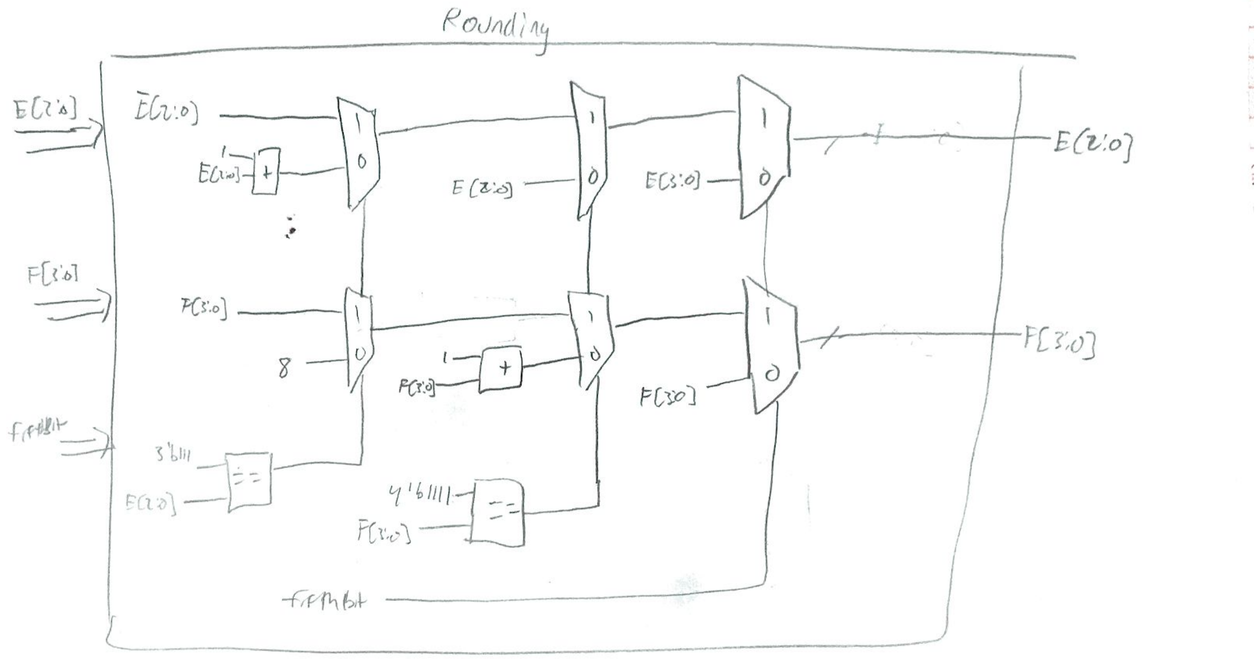


Above: Top Level Design
Below: Sign/Magnitude block implementation





Above: Count leading zeros/extract leading bits block implementation
Below: Rounding block implementation



Simulation Documentation

Because Lab 1 was simulation-based rather than implementing this logic on an FPGA, we tested our software design using its waveforms on simulated data inputs and forcing different constants. Throughout the course of development, we incrementally tested and simulated our design to ensure the different parts of the module were functioning. For example, one of the first mechanisms we tested was the clocks. We wanted to make sure that the clock was indeed flipping from 0 to 1 and back again at the correct time increments. Then, we built on this framework to see if the sign bit was changing properly. To test this, we set the 12-bit input (D) in our test bench to 0 then 1 then -1 with delays in between each change so that we could clearly see the differences. This was the methodology for the rest of our tests: test every possible case for output.

Regarding general test cases for the entire module, we ramped up from the incremental tests during development. First, we simulated a sanity to make sure that the base cases were working properly. These included testing the 12 bit binary encoding equivalents of the numbers 0, 1, and -1. We also tested another random positive and negative number to check their floating point bit representations. Then, as a more thorough check, we simulated all of the test cases provided and explained in the Lab 1 specification. This included other conversions to check from 12 to 8 bit floating point encoding, cases to round up, cases to round down, and the requirements to handle overflow. Next, we did a thorough check of transitions. This included checking the priority encoder's functionality, like detecting a change in the exponent field of the output. This also included testing wraparound negative values (i.e. overflow processing). For example, we tested the two's complement equivalent of 3000 which is outside the range of representable numbers in 12 bits. This means that 3000 would essentially be a negative number, so we checked that the output fields were consistent with what they should be. Also, we simulated different cases of rounding, including up and down. For all rounding cases, we tested the number that needed to be rounded up, and its equivalent number (bitwise equivalent) that did not need rounding; this way, we could see that the bit pattern of the output was the exact same, meaning that the rounded and actual values were the same. Lastly, we checked the outer range of a 12-bit bitwise encoding (maximum and minimum possible). Hence, we could check the majority of cases through this thorough mechanism of checking for all of the requirements of the specification. These specific test cases are reflected in Figure 1, which shows a zoomed in view. Our final simulation involved using a loop in our test bench to test all possible values that the 12-bit linear encoded input could take (ranging from -2047 to 2048). By adding delays between each switch in value, we could see the progression of each field in the floating point representation (sign, exponent, fraction). We also randomly tested points in this output to ensure accuracy in converting the two's complement input to floating point output. These waveforms are displayed in Figure 2, and a magnified view of when the sign (S) goes from negative to positive in Figure 3.

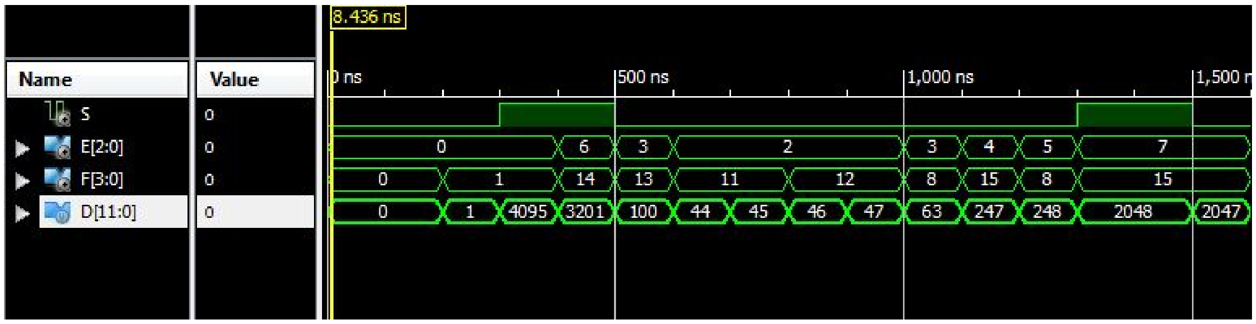


Figure 1: General Test Cases

From left to right: Zero, 1, -1, normal negative (-895), normal positive (100), Rounding down, rounding down, rounding up, rounding up, rounding up causing exponent increase, all ones in the mantissa (247), increase exponent but maintain leading 1 in mantissa (248), negative max, positive max

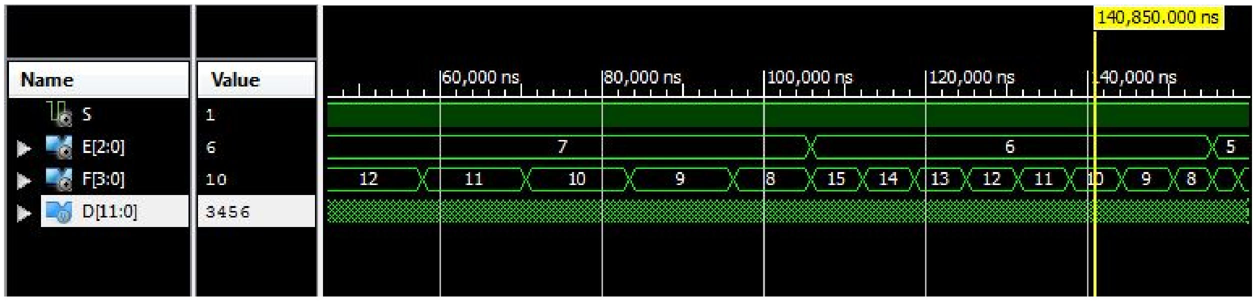


Figure 2: A chunk of the test for every number possible

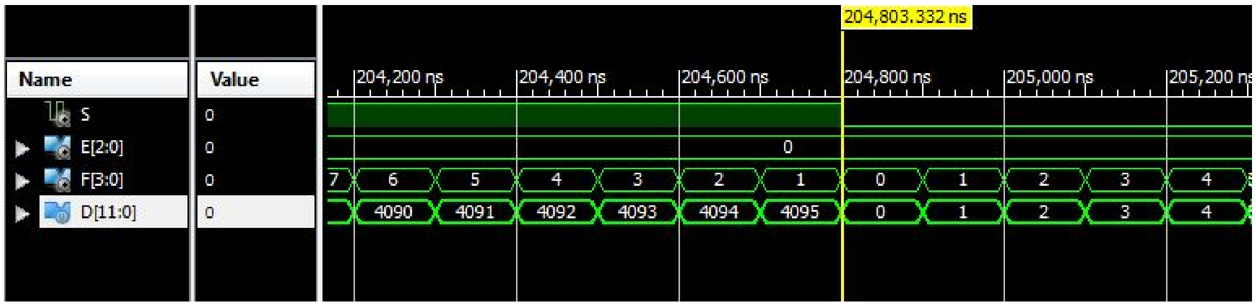


Figure 3: Zoomed in of sign switch for every number possible

Conclusion

In summary, the design of our lab followed the specification's block diagram and requirements in a sequential manner. Our simulation strategy was to test portions of our module incrementally, then do general test cases for all possibilities of 12-bit input. We encountered difficulties in debugging certain transitions (e.g. overflow of rounding), but by isolating the region where the error was occuring in our code, we were able to find the source of the bug and fix it appropriately. Also, we simulated different cases by also forcing certain values in the waveform; we then added these to our testbench for a consistent and thorough simulation. Generally, this lab could be improved with the addition of a bias. This way, the floating point representation would be consistent with what we learn in other classes (namely EE M16). Nonetheless, this lab was helpful in refining our Verilog and simulation skills, and implementing a useful floating point converter.