Megan Pham
505313300
CS M152A Lab 2
Dipti Sahu

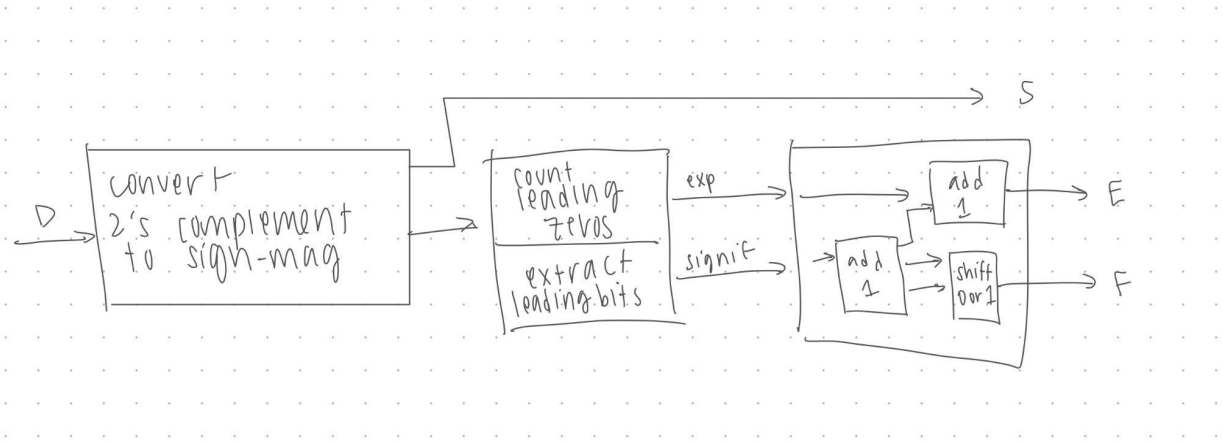
Lab 1 Report

Introduction

Not only are binary numbers used to represent integer numbers, but also they can be seen to exemplify noninteger fractional numbers through floating-point representation. This representation normally consists of 32 bits and can also be seen through scientific notation. An example of this would be 1.01011101*2^5, which would equal to 0 10001000 10111010000000000000000 in floating point. These bits are then divided into three sections: sign, exponent, and mantissa/significand. In a binary number, the most significant bit (the leftmost part of the number) is dubbed as the sign bit. 0 represents a positive number, and 1 is a negative number. The exponent section is usually eight bits long and is the exponent that is shown in scientific notation. Lastly, the mantissa, or significand, field is 23 bits long and consists of the rest of the number.

Module Design

In this lab, we had to convert a 13 bit number. For my design, I made it the same as the diagram presented as an example in class as seen below.



As noted in the assignment document, there are three main parts to designing a floating point conversion, which ended up being my three submodules: converting our input into sign-magnitude representation, counting the leading zeroes and extracting leading bits, and rounding.

The first block allowed me to change all the negative binary numbers into nonnegative binary numbers. I made a module named *convert* with the input binary number and the outputs as the sign bit and the converted number. If the most significant bit was zero, then there is no need to change the number. Taking into consideration the most negative number 1 0000 0000 0000 (-4096) would not work when changing the negative into a positive, I had to check that test case and convert it to the most positive number instead 0 1111 1111 1111. After checking that test case, if it didn't pass the above criteria, then the input is a regular negative number. To convert into a positive number, I take the complement of the input and add one.

In the second block, I used a priority encoder, as what was recommended in class, to count the leading zeroes. This was implemented with a bunch of if and else-if statements. The basic logic was that, starting from the most significant bit and going to the least significant bit, if the bit was a one, then there were no more leading zeros. The significand would then be wherever that one bit was and its next three bits, making four bits total. As the significand output can be obtained by right shifting the most significant input bits from bit positions 0 to 7, that would indicate the exponent. For example, let's say the one after the leading zeros was at bit position 10. That means that the significand would be the input[10:6], and the exponent would be equal to 6 (as input[11]==1 would have exp==7). Then, I made sure to keep track of the sixth bit for the rounding module. Obviously, if by bit four, if there hasn't been a one after the leading zeros, the significand becomes the last four bits of the input, the exponent would be equal to zero, and the sixth bit would also be zero.

The last part of this floating point conversion module is rounding, which depends on the sixth bit from the previous block. If the sixth bit is a zero, then we make no changes to the significand and exponent. However, if the sixth bit is a one, then we have to check for overflow. If the significand is 11111 and the exponent is 111, then there is an overflow, and we keep the exponent and significand the same (111 11111). However, if the exponent is not 111, then we can handle the significand by shifting right by one and adding one to the exponent number. Without worrying about overflow now, usually, we can add one to the significand and keep the exponent the same.

Finally, I called all these submodules into my main module FPCVT. This module takes in the input and outputs the sign, exponent, and significand: the three parts of the floating point representation.

Testbench Design

To code my testbench, I had the reg input and outputted the sign, exponent, and significand wires. To test my code, I used the examples from the assignment document which included:

0 0000 0110 1100 → 0 010 11011
0 0000 0110 1101 → 0 010 11011
0 0000 0110 1110 → 0 010 11100
0 0000 0110 1111 → 0 010 11100

Then, I made sure to check the edge cases, which included:

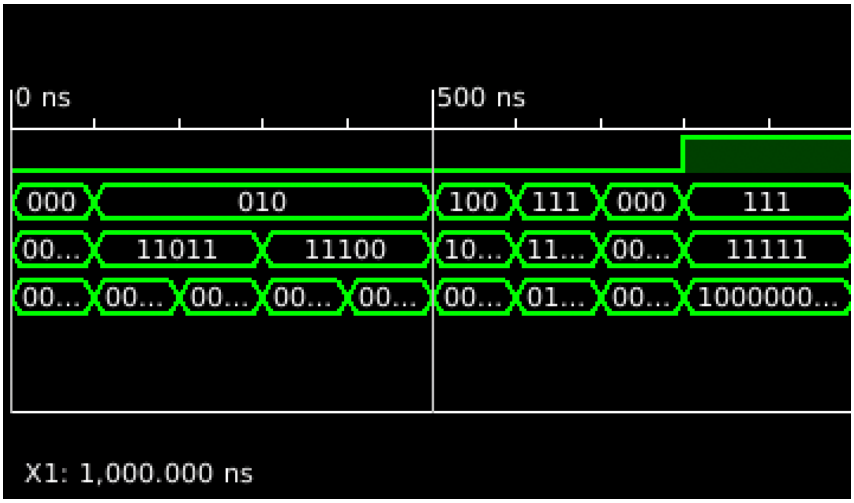
0 0000 1111 1101 //overflow case → 0 100 10000
0 1111 1111 1111 //largest positive number → 0 111 11111
0 0000 0000 0001 //smallest positive number → 0 000 00001
1 0000 0000 0000 //most negative number → 1 111 11111

My module passed all test cases. To validate these numbers and calculations, I double-checked the first four cases with the assignment examples (which had the floating point conversions given). For the last four cases, I checked by converting the numbers into floating point conversion myself.

ISE Design Overview Summary Report

https://drive.google.com/file/d/15btfM18mugDxkzV4qcGkcNJEeJ_l6x5O/view?usp=sharing

Simulation Waveforms



Conclusion

Converting a binary number to a floating point is tedious, yet allows engineers and mathematicians to utilize noninteger binary numbers in their work. In this lab, I have learned how to use Xilinx and create a floating point converter by breaking up the problem into smaller subproblems: the three blocks from my design module. The most difficult part of this lab was trying to get started with the Virtual Machine as my mouse kept bugging out, and I was unable to press anything. After many attempts of googling and resetting the VM, it finally worked. In addition, since I was not used to the syntax, it was hard for me to quickly type in the code for the submodules without doing research.