Lab 2

CS M152A

Aaron Cheng

Aditya Padmakumar

Introduction

All data is comprised of just streams of bits as 1's and 0's. It is how those streams of bits are interpreted that give data its meaning. There are many different data representations of data including unsigned integers, one's complement, two's complement, and floating point representation. In this lab, the goal was to familiarize ourselves with converting a two's complement representation of a number to a simple floating point representation.

To do this, we used the Xilinx ISE software to design and simulate a combinational circuit that converted a 12-bit two's complement signed integer to its counterpart in 8-bit floating point. The floating point representation consists of a 1-bit sign representation, a 3-bit exponent, and a 4-bit significand, shown below:

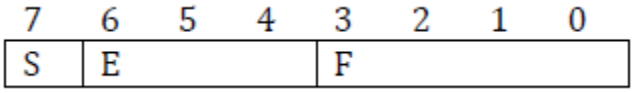

Figure 1: 8-bit floating point representation

The value of the number is then calculated as $V = (-1)^S \times F \times 2^E$, where the S signifies the sign of the number, the significand F ranges from [0000] = 0 to [1111] = 15, and the exponent E ranges from [000] = 0 to [111] = 7.

Lastly, because not all 12-bit two's complement integers can be exactly represented in an 8-bit floating point, we use rounding to assign the closest floating point value to each number.

Design Description

We decided to base our design of the floating point converter on the circuit block diagram that was provided in the lab manual. There are three main modules to our converter. The first module takes as input a 12-bit two's complement signed integer and outputs its sign-magnitude representation. The second module counts the leading zeroes of the integer's magnitude, effectively determining the integer's exponent value, and extracts the significand from the bits following the leading zeroes. The third module handles the rounding of integer and its consequences, such as overflow, in order to produce the most accurate floating point representation of the original input. A high level schematic of our floating point converter is shown below:

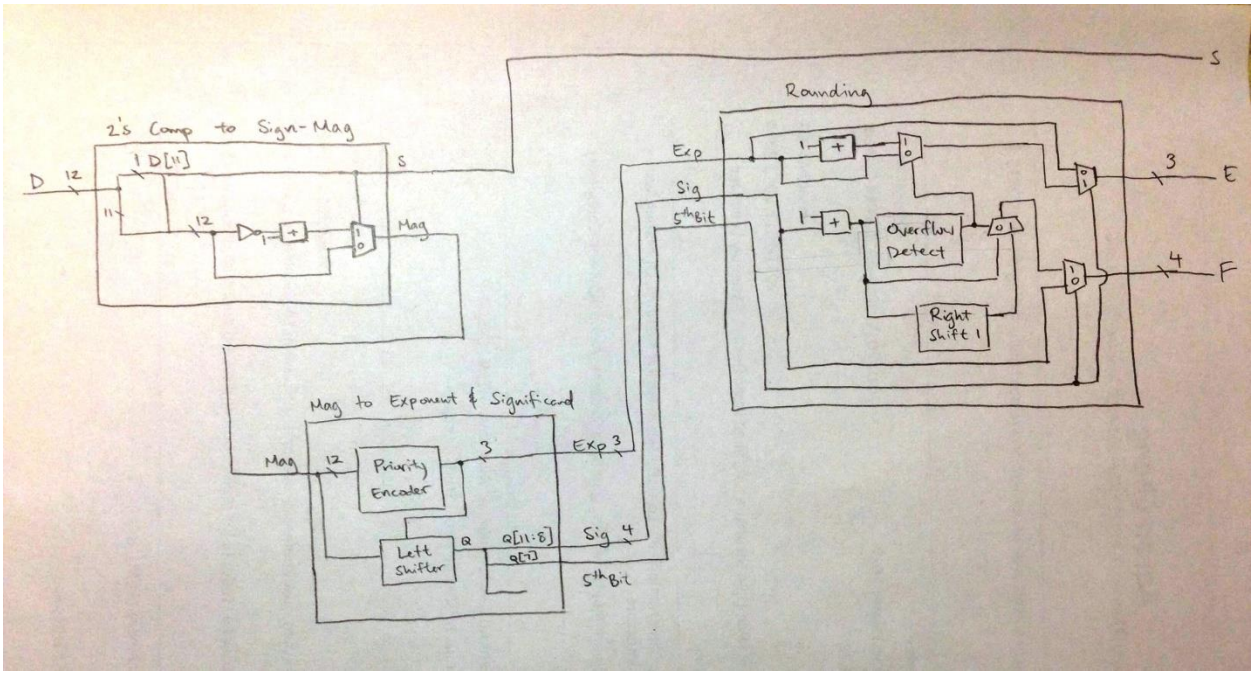

Figure 2: Schematic of converter

In Figure 2, the input to the converter is D, a 12-bit two's complement integer, located in the far left. The outputs of the converter, located in the far right, are S, a bit representing the sign of the integer, E, 3 bits representing the value of the exponent, and F, 4 bits representing the significand.

The first module takes as input D, a 12-bit two's complement representation. The most significant bit is extracted and outputted as S, the sign of the integer. This output S becomes the output S of the whole converter. If the extracted bit is 0, the 12 bits are left alone, and if the extracted bit is 1, the 12 bits are flipped and added by 1, producing the output Mag, the magnitude of the integer.

This first module's output then becomes the input of the second module, which converts the magnitude to the original integer's exponent value and significand. The second module first counts the number of leading zeroes in the magnitude and determines the exponent based on the outcome, outputting it as Exp. It then retrieves the first four bits after the leading zeroes as the output Sig, and also grabs the next bit after those as output FifthBit, which will be used to determine whether to round or not.

The third and final module takes the second module's outputs, Exp, Sig, and FifthBit, as inputs. If FifthBit is 1, this module increments Sig by 1. If Sig overflows, then it right shifts Sig by one bit and increments Exp. The resulting Exp and Sig are then outputted as E and F, respectively. If FifthBit is 0, then Exp and Sig are outputted as is. The outputs E and F become the outputs of the whole converter.

Simulation Documentation

After the design of each module, we conducted basic tests on the module to ensure the module was functioning as required.
There was no need to observe any waveform during simulation as the only output we required was displayed on the console.

TCtoSM: (Two's compliment to Signed Magnitude)
Expected Result: Given a 12 bit binary input (D), we expect this module to convert this input to 12 bit signed magnitude representation (Mag). Additionally, we expect a single bit output (S) to contain of the sign of the output.

Special Cases:
For the largest negative number: -2048, the result looks like -2048 instead of +2048.

Test Cases:

Input (D)	Output (Mag)	Output (S)	Success?
000000000000	000000000000	0	Y
011111111111	011111111111	0	Y
111111111111	000000000001	1	Y
100000000001	011111111111	1	Y
100000000000	100000000000	1	Y

MagtoEF:
Expected Result: Given a 12 bit binary input (Mag), we expect this module to count the leading zeros and accordingly extract the leading bits. This consists of calculating the exponent (Exp), significant (Sig) and extracting the "fifth bit".

Special Cases:
For a Mag with no leading zero's, since the Lab Description was unclear on what to do specifically in this case, we assign Exp to 7, Sig to 15 and FifthBit to 0. (Smallest possible floating point value)

Test Cases:

Input (Mag)	Output (Exp)	Output (Sig)	Output (FifthBit)	Success?
000000000000	000	0000	0	Y
011111111111	111	1111	1	Y
000000000001	000	0001	0	Y
011110111111	111	1111	0	Y
011110000000	111	1111	0	Y
011111000000	111	1111	1	Y
000110111111	101	1101	1	Y
100000000000	111	1111	0	Y

Rounding:

Expected Result: Using the outputs from the previous module, we expect this module to alter our current Exp and Sig based on the value of the FifthBit and finally give us our final Exponent (E) and Significand (F).

Special Cases:

For an Exp of 7 and Sig of 15, while rounding, the exponent could be larger than our maximum possible exponent. Therefore we assign E to 7 and F to 15. (Largest possible values)

Test Cases:

Input (Exp)	Input (Sig)	Input (FifthBit)	Output (E)	Output (F)	Success?
000	0000	0	000	0000	Y
111	1111	1	111	1111	Y
000	0001	0	000	0001	Y
001	1111	1	010	1000	Y
101	1101	1	101	1110	Y
100	1111	0	100	1111	Y
100	1111	1	101	1000	Y

Once we were able to confirm all the modules were outputting expected results, we combined all the modules and then tested the entire system together.

Test Cases:

Input (D)	Output (S)	Output (E)	Output (F)	Success?
000000000000	0	000	0000	Y
000000101100	0	010	1011	Y
000000101101	0	010	1011	Y
000000101110	0	010	1100	Y
000000101111	0	010	1100	Y
111111111111	1	000	0001	Y
100000000000	1	111	1111	Y
011111111111	0	111	1111	Y

Conclusion

Based on the tests we designed for this project and the demo we passed, we can confidently say that our design successfully converts 12-bit two's complement signed integers into 8-bit floating-point values.

During the lab we dealt with the following problems:

- Usage of priority encoder vs. manually counting number of zeros: We decided that using a 16-bit priority encoder would unnecessarily increase the complexity of our proposed solution.
- Handling corner cases: Some corner cases for this lab were not defined in the lab manual and hence we were forced to define our own expected values.