CS M152A
Lab 1

Michael Wu, Haoyu Yun, Jennie Zheng

Introduction and Requirement

We will build a combinational circuit to translate a *12-bit two's-complement number* to the nearest *8-bit floating-point number*. In this way, we can produce a compressed representation of the number.

We implement the circuit in Verilog in a **FPCVT** module.

The input is a 12-bit two's-complement number. The 8-bit floating-point number consists of a 1-bit sign, 3-bit exponent, and 4-bit significand. The outputs are the values of these three components.

If the exponent is not zero, we round the 12-bit number to the nearest possible floating point value based on the value of the bit following the significand.

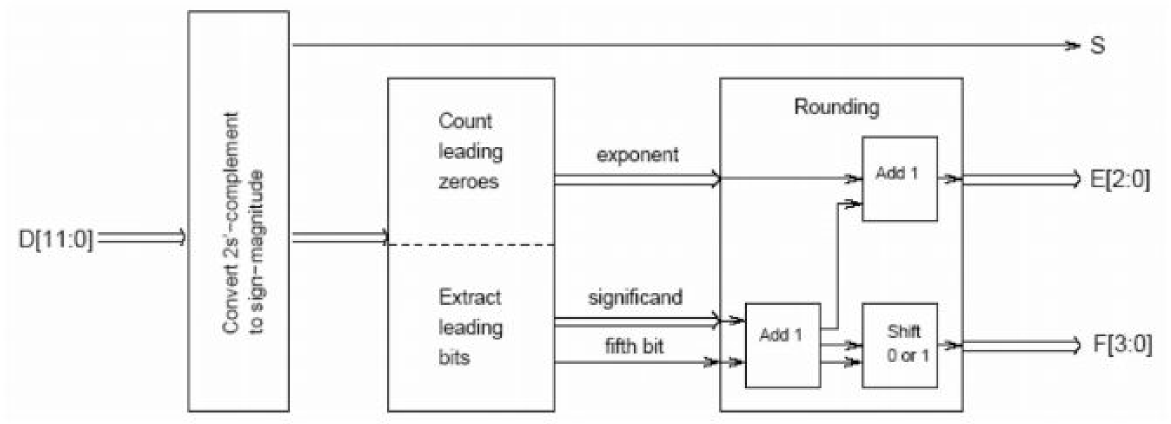
There are two categories of special cases to consider.

With the input -2048 (1000 0000 0000), we cannot properly convert this to sign-magnitude due to overflow. In this case, we just return the smallest floating point number (1 111 1111).

With inputs greater than or equal to 1984 (0111 1100 0000) or lesser than or equal to -1984 (1000 0100 0000), rounding would force the exponent to 8. In this case, we just return the largest floating point number (0 111 1111) or smallest floating point number (1 111 1111) based on the sign.

Design Description

This circuit consists of four core modules, three modules for each step of the conversion process and one module to link them together.



Basic Module Architecture

Sign Magnitude Module

We first convert the two's-complement number to its signed-magnitude representation with a **signMagnitude** module. The input is the 12-bit two's-complement number and the output is the 12-bit magnitude and a 1-bit sign.

The sign is equivalent to the most significant bit of the input. If the most significant bit of the input is 0, then the magnitude is unchanged. If it is 1, however, then the magnitude is one plus the complement of the input.

Exponent and Significand Extraction Module

We then extract the 3-bit exponent and 4-bit significand from this number, as well as a fifth bit for rounding purposes, through a **countZerosExtractData** module.

The input is a 12-bit magnitude and the output is a 3-bit exponent, 4-bit significand, and 1-bit fifth bit used for rounding in the next module.

We determine the exponent by counting the number of leading zeroes -- the fewer leading zeroes there are, the larger the number and its corresponding exponent. The significand is either the four bits starting from the first 1 or the least significant four bits. The fifth bit is the next most significant bit not used in the significand; if the significand includes the four least significant bits, the fifth bit is 0.

Rounding Module

We obtain the components of the floating-point number by rounding the exponent, significand, and fifth bit in a **round** module.

The input is a 3-bit exponent, 4-bit significand, and 1-bit fifth bit. The output is the rounded 3-bit exponent and 4-bit significand.

If the fifth bit is 0, no rounding is needed. If the fifth bit is 1, we must round up the significand by increasing it by 1. If the significand is maximum, or 1111, we would round up (to 10000) which does not fit into our 4-bit significand representation. We deal with this overflow by right-shifting this value (setting the significand to 1000) and adding 1 to the exponent.

Combinational Circuit Module

We put it all together in the **FPCVT** module.

The input is a 12-bit two's-complement number. The 8-bit floating-point number consists of a 1-bit sign, 3-bit exponent, and 4-bit significand, which are the three outputs of the module.

We pass the 12-bit two's-complement number through the **signMagnitude** module to obtain the 1 bit sign and 12-bit magnitude. Then, we pass the 12 magnitude bits to the **countZerosExtractData** module to extract a 3-bit exponent, 4-bit significand, and 1-bit fifth bit. Finally, we pass these outputs through a **round** module to obtain a rounded 3-bit exponent and 4-bit significand.

For the special cases, we detect them with *if* statements and manually set their exponent and significand.

Simulation Documentation

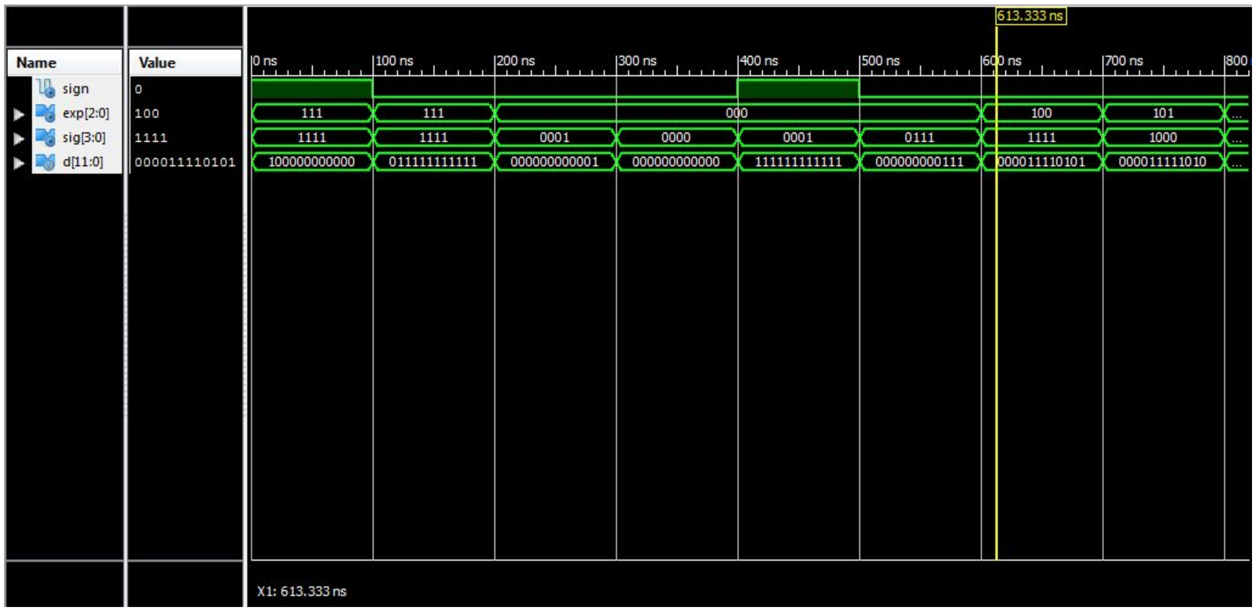
We decided to test each of the individual modules in order to ensure that the code worked properly. We ran into three separate bugs in three different modules.

First, in the **signMagnitude** module, we accidentally flipped the result so that we returned negative numbers as is and positive numbers as their negative complement. This was a very simple one character fix: we changed a 0 into a 1 in the if statement.

Secondly, in the **countZerosExtractData** module, we forgot to define the length of the two variables which counted the number of leading zeros. This caused the compiler to assume the two variables were 1-bit when in reality they needed to go up to 4-bit to store numbers ranging from 0 to 11.

Thirdly, **FPCVT** module, we forgot to declare two variables. This resulted in the two variables getting implicitly initiated into 1-bit values when they were supposed to be 3-bit and 4-bit long values, respectively. Again, this was a simple fix of declaring the variables.

Overall, we realized that all of our bugs were careless mistakes. Although they were very minor, they cost us valuable time. In order to speed up the testing process, we divided the four modules into two groups of two and had two people test two modules each. One person tested on an online EDA playground and the other tested in the ISE Suite. Although the two testing environments were not identical, they were close enough for our purposes and we managed to catch all our bugs. Interestingly, EDA playground compiled and ran code significantly faster than the ISE Suite because the EDA playground was smaller and used less overhead.



Waveform of successful floating point conversion simulation.

Conclusion

We designed a combinational circuit to convert a two's-complement number to floating-point. We take the magnitude of the number, extract an exponent and significand based on the number of leading zeroes (or position of most significant 1 bit, if any), and then round them based on a fifth bit.

We wrote the circuit in Verilog. We split the task up into modules, which we combined to generate our floating-point number.

The primary difficulties were in familiarizing ourselves with the ISE Suite programming environment, especially in learning how simulation worked. We were satisfied with our modular design of the circuit.