Anthony Humay, Andrew Musk, Clayton Green

M152A Lab 6

Mastorakis

Lab 1 Report

*Introduction and requirement (10%)*

*Summarize background information about the lab and the detailed design requirements. It’s very important to make sure you are designing the right thing before starting.*

**Introduction**

In this lab, we built a combinational circuit that took in a 12-bit two’s complement representation of an integer and converted it to a 8-bit floating point. The 8-bit float consists of 3 components: a one bit sign, a three bit exponent, and a four bit mantissa. Using a floating point representation allows for a greater range of numbers compared to an equal amount of bits in two’s complement, but at the cost of having more rounding issues and therefore less specificity (an exponent of 111 means the number can be 27\*mantissa, which is larger than two’s complement).

This process entailed converting two’s complement to sign magnitude, counting the number of leading zeros, and using that information to get the leading bits. If the leading bits would overflow when incremented by one, we would need to increase the exponent by 1 (as this has the same effect). If increasing the exponent resulted in another overflow, this time of the exponent, then we were told to leave it at that overflow.

To get the leading zeros, a priority encoder would be perfect for finding the index of the first high bit, which can be used to calculate how many zeros preceded it.

*2. Design description (15%). Document the design aspects including the basic description of the design, modular architecture, interactions among the modules, and interface of each major module. You should include schematics for the system architecture. You can also include figures for state machines and Verilog code when needed.*

**Design Description**

The main module was abbreviated FPCVT for “floating-point conversion,” which had a 12 bit input which is the two’s complement version of the number, and had three outputs: sign (1 bit), exponent (3 bits), and significand (4 bits). This module instantiates each of the submodules.

The first module called by FPCVT is SignMagnitudeConverter, which takes in the two’s complement number and converts it into a single sign bit and a magnitude. This module is implemented by checking to see if the number is positive or negative. If it is negative, the bits must be flipped, and then added to one. If it is already positive, nothing needs to be done. The sign bit and magnitude are then returned.

Next, GetExponentSignificand is called. It takes in the magnitude as an input. It then starts by checking where the most significant 1 bit is in the magnitude using an if/else ladder. Based on where this first 1 is, it then pulls the 4 bits of the significand from the magnitude using a case statement on a local variable that stores the location of the first bit. Next, based on where we found the significand, we check the next lowest bit in order to check if we will need to round. The exponent is then simply set to be the variable we used to store the location of the most significant 1 bit. We also deal with the special case of when the most significant 1 bit is in the magnitude[11] position. Now, the exponent, significand, and our variable telling us whether rounding is required are returned to FPCVT.

The final module called is Round, which deals with the required rounding. First, we check if the exponent, significand, and if the most significant carry bit are all 1’s, and if they are, we set the exponent to 7 and significand to 15. Otherwise, we check if the most significand carry bit is 1, and if it is, we set the significand to increment by 1. Otherwise, we just leave it as is. Now we check if significand[4] is a 1, and if it is, we increment our exponent by one and right shift the significand by one. Finally, we return the significand and exponent.

We now have our sign, exponent, and significand, rounded if needed.

*3. Simulation documentation (10%). Document all the simulation efforts (what requirements are tested and what the test cases are), document bugs found during simulation, and provide simulation waveforms.*

**Simulation Documentation**

To test base functionality, we tested generic positive and negative numbers. These were numbers not on the edges, like 25 and -36. To ensure that these were correct, we verified with our own calculators and used the formula:

V = -1S \* F \* 2E

Initially, we ran into an issue where the sign was not getting assigned to negative numbers. This was due to the first submodule that dealt with assigning the sign not actually being assigned at the end of the module.

Also, there was a major error in how we were counting the zeros; we tried to recursively go through and check until we realized how much easier and simpler it was just to look for the first one with a switch statement.

We wanted to ensure that the edge cases were tested, so we made sure that input = 0 gave us 0 000 0000.

We tested the positive bound, input = 2047, to ensure it gave us 0 111 1111 (positive max).

We tested the negative bound, input = -2048, to ensure that it gave us 1 111 1111 (negative max).

Also, during testing we didn’t know we had to put time weights in between each new input. This was a conundrum until we added 100 nanosecond ones.

*4. Conclusion (5%). Summary of the design. Difficulties you encountered, and how you dealt with them. General suggestions for improving the lab, if any.*

**Conclusion**

We divided our main FPCVT module into three sub modules, one for each major computational task of the conversion. These modules handled finding the sign, exponent/mantissa, and rounding respectively. The first module, finding the sign, first converted from two’s complement to signed magnitude, and then assigned the sign value to high or low. The exponent/mantissa were also assigned in the module. Assigning values is something that we were unfamiliar with, so we experienced errors in the beginning trying to assign the sign, exponent and mantissa in the main FPCVT module.

We did not realize that, unlike in our object oriented past experiences, the variable names of the inputs and outputs of the three modules mattered and had to match what you had made when you “instantiated” the modules originally. We quickly figured this out.

Also, we forgot to take into account overflow in the exponent. We added the last extra if statement in the exponent/mantissa module to deal with this.