# Lab 2 Report:
# Floating Point Conversion

CS M152A, Potkonjak
Lab 3, Gu
Spring 2017

By,
Eddie Huang, Lucas Jenkins, and Jason Less

## INTRODUCTION AND REQUIREMENTS

In this lab, we were tasked with designing a combinational circuit using ISE software that would receive a 12-bit linearly encoded number and convert it into the nearest 8-bit floating point number. Floating point numbers used in modern computing systems are generally larger in size, but for the purposes of this lab, they were shortened for simplicity. The 8-bit floating point numbers outputted by our combinational circuit were represented using one sign bit, three exponent bits, and four significand bits.

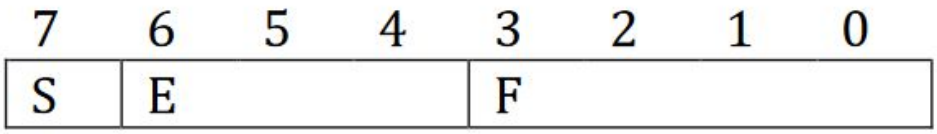


**Figure 1: 8-bit floating bit representation as described above.** The leftmost bit represents the sign bit, the next three bits represent the exponent bit, and the last four bits represent the significant bits. The final value of floating point numbers is calculated as follows: $V = (-1)^S \times (2)^E \times F$.

| Leading Zeroes | Exponent |
|---|---|
| 1 | 7 |
| 2 | 6 |
| 3 | 5 |
| 4 | 4 |
| 5 | 3 |
| 6 | 2 |
| 7 | 1 |
| ≥ 8 | 0 |

**Figure 2: Input format.** In order to implement the circuit, we take the absolute value of all negative decimal numbers, then convert its positive counterpart to a floating point representation, and add the sign bit as necessary. Once this operation is complete, all numbers will have at least one leading zero. The number of leading zeroes determines the value of the 3-bit exponent, as shown above.

The rules set forth in Figure 2 work for almost all values of a 12-bit decimal representation of a number; all negative numbers except for $-2^{11}$ have a positive absolute value that can be represented in the 12-bit linear representation. As such, we must design a special case for this number; if this value is entered, the circuit will return a negative floating point number of the largest possible magnitude.

| Rounding Examples | | |
|---|---|---|
| **Linear Encoding** | **Floating Point Encoding** | **Rounding** |
| 000000101100 | [0 010 1011] | Down |
| 000000101101 | [0 010 1011] | Down |
| 000000101110 | [0 010 1100] | Up |
| 000000101111 | [0 010 1100] | Up |

**Figure 3: Examples of 12-bit linear encodings and their respective 8-bit floating point conversions.** Using the first non-zero bit as a starting point, the fifth bit determines whether the four bit significand is rounded up or down; if the fifth bit is zero, it is rounded down, whereas if the fifth bit is one, it is rounded up.

In most cases, rounding up will result in the three-bit exponent to remain the same. However, when the first four significant bits are "1111" and the fifth bit is also 1, by the rules of rounding up, the significand must be "10000". However, there are only four significand bits, and as such, we omit the final 0 and increase the exponent by one. In the case that the exponent is already 7, and thus cannot be increased to 8 as that would require a four-bit exponent, we simply return the largest possible floating point number.

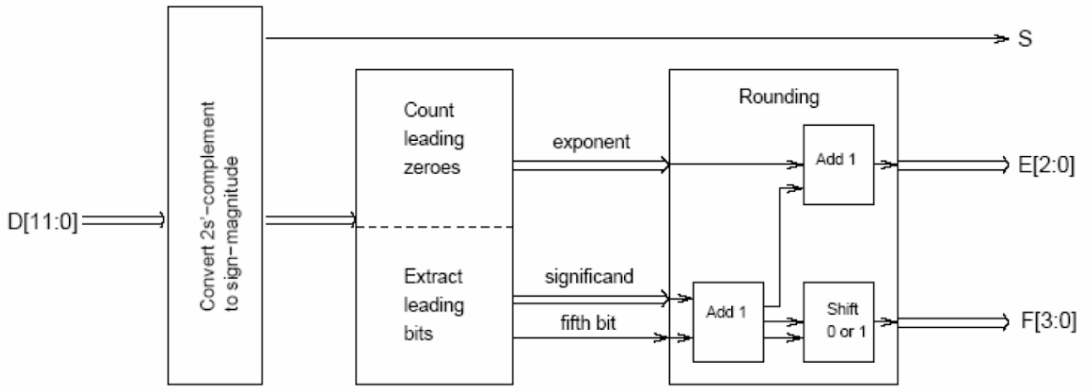


**Figure 4: Overall Design Concept.** The circuit takes in a 12-bit decimal integer, (denoted D[11:0]) runs it through the circuit, and outputs a floating point number, represented via the sign bit S, the 3-bit exponent E[2:0], and the 4-bit significand F[3:0].

## DESIGN DESCRIPTION

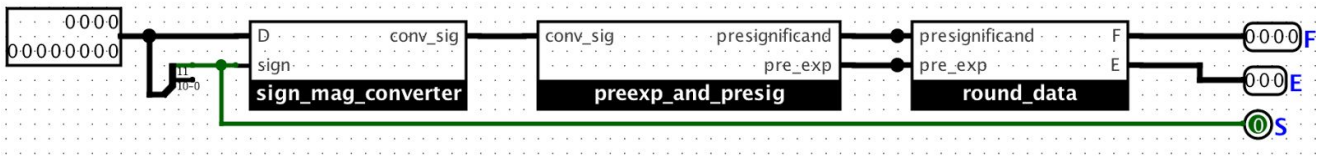


**Figure 5: High level diagram of our design concept**

The high-level design of our 12 bit to 8 bit floating point converter follows the design described both in the spec and in the above introduction. We split the floating point converter into 4 separate files and three sub-modules, which are described in detail below: a module to extract the sign bit and deal with negative numbers, a module to extract basic information about the

significand and the exponent, and a module to round this basic information to the nearest 8 bit floating-point value. In all modules, the main method of representing control flow in Verilog if statements was through comparisons linked to multiplexer select bits.

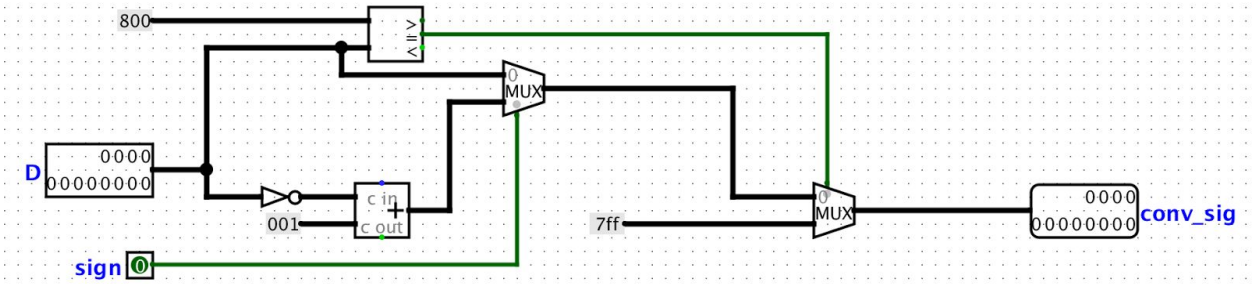


**Figure 6: Diagram of logic for sign_mag_converter**

The first module is simple - it checks whether the sign bit of D, the input, is 1. If it is, it uses a multiplexer to select D inverted with one added to it (which converts a negative 2's complement number to its positive counterpart). We know that D is negative if the first bit is zero, so we want to perform this inversion. We also check for the edge case when D is a 1 followed by eleven 0's, an edge case where D's positive counterpart does not fit in twelve bits, so we simply convert it to the highest positive number available (which overflows anyway). This new signal, which is possibly inverted if D was originally negative, is sent as an output to conv_sig.

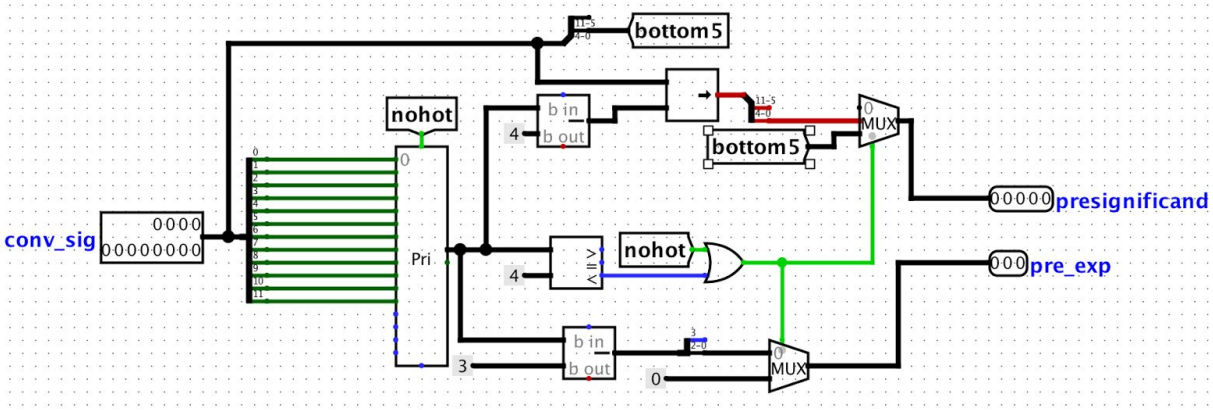


**Figure 7: Diagram of logic for preexp_and_presig module**

The next module takes the conv_sig output from the previous module, and outputs a basic significand value (presignificand) and exponent value (pre_exp), which need to be rounded to obtain the final value. Although our Verilog code did not create a priority encoder in the traditional way, we still maintained idea of capturing information about where the first one in the input from the MSB down. If the first one was at a position < 4, we treated it as if it were at position 4, as if it is at position <=4 we can represent it exactly in the 5 bit presignificand. We then used (the location of the first 1)-3 to calculate the value of the pre_exp in the floating point value, the power to which to raise the significand is determined by three bits before the location of the first 1 in the input. We then looked at either the first 4 bits after the first 1 if the position of the first 1 was greater than 4, or the bottom 5 bits if the position of the first one was at position

less than 4, but using a multiplexer and shifter to look at the bits. We figured out the number of bits to shift by subtracting 4 from the position of the first 1 (the obvious number to shift by if one is looking for the 5 bits after and including the first one).

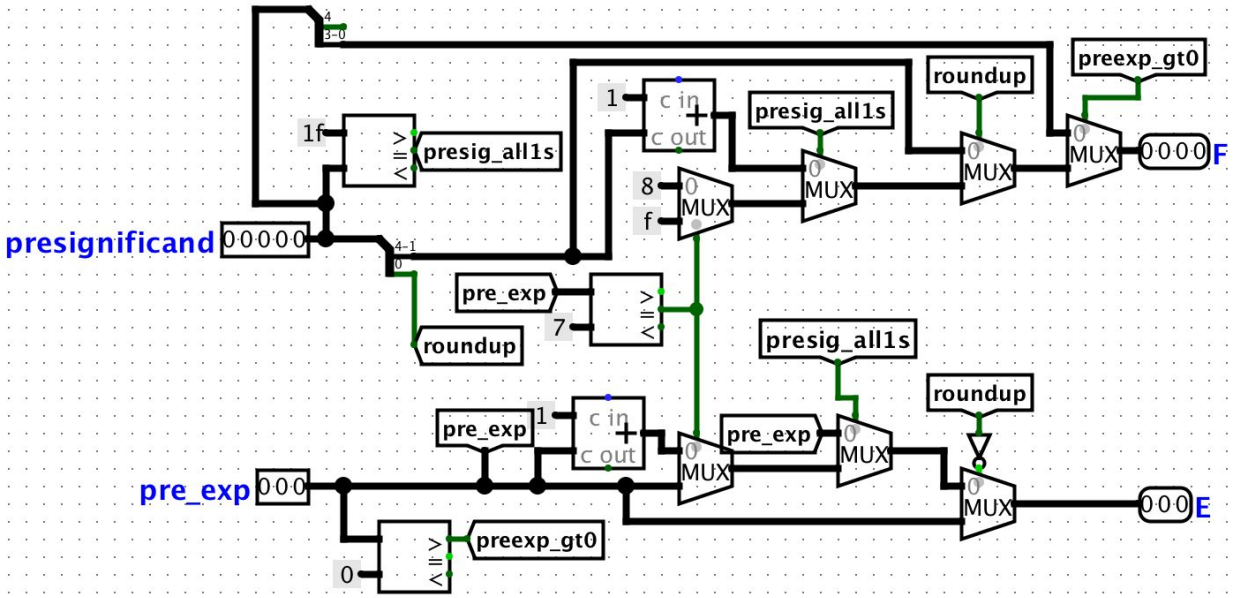


**Figure 8: Diagram of logic for round_data module**

Although the logic diagram for the round_data module appears complicated, the underlying design principle is relatively simple, the diagram just must account for many edge cases. The basic principle is to look at the LSB of the pre_significand. If it is 1, we round the presignificand to presignificand+1, while if it is 0, we leave the presignificand as is. This is so that our final output significand, F, is as close as possible to the input D (as the fifth bit in the presignificand marks the 50% cutoff point between presignificand and presignificand+1). The majority of the edge case problems revolve around cases when the LSB is 1 and we try to round up. However, if the upper four bits of presignificand are all 1's AND the lowest bit is a 1, when rounding up, presignificand would overflow. Thus, we change presignificand to this overflow value and add one to the output exponent, E. We do this as long as the exponent is not already at its maximum value, in which case we leave both the exponent and significand at their given value, as this is the maximum value representable in our eight-bit float, and this is what the spec says to do with overflow. We deal with these problems by using comparators to check if values will overflow, and using multiplexers with the select bits to these multiplexers being the output of the comparisons. The last remaining edge case to deal with comes when the pre_exp is zero, and there are only at most 4 significant bits. In this case, even if the LSB is 1, we do not want to round up, as we can represent this value exactly in the 4 bit significand F. Thus, we ensure that the output F for this value is always the least significant 4 bits of presignificand and that the exponent E remains 0 by comparing the significand with 0 and using the output of this as input to multiplexers.

## SIMULATION DOCUMENTATION

As the purpose of the lab only required the conversion of a twelve-bit two's complement number into a floating point approximation, it didn't require the use of the Nexys3 board. The entire project was designed and tested using Xilinx ISE simulation. As discussed in previous sections, our project consisted of four modules: one top module (FPCVT), and three submodules (sign_mag_converter, preexp_and_presig, and round_data). As this was a relatively simple project, there wasn't a large amount of tests that needed to be done, and there were not a whole lot of edge cases. However, we still did thorough tests on each of our modules in isolation, and as a modular system, to make sure that our program achieved the desired result (i.e. receiving a 12-bit signed magnitude value as input and extracting the sign-bit, exponent field (3 bits), and significand (4 bits)).

First of all, to test the sign_mag_converter module, we simply gave as input to the module different 12-bit two's complement numbers to make sure that the module properly converted them to their signed magnitude counterparts. Note that the signed-bit was extracted in the top module, so this module only converts it to the positive counterpart, and the sign-bit is incorporated in the final output values. The test cases used are listed below:

| Test Case | Result | Pass/No Pass |
|---|---|---|
| Positive input values | Correctly identifies positive input, and leaves as is | Pass |
| Negative input values | Inverts all bits, and then increments by 1 (i.e. absolute value) | Pass |
| Smallest possible value (-2,048) | Deals with overflow, by just assigning largest value (2,047) | Pass |

**Figure 9: Table showcasing the test cases used for the sign_mag_converter submodule. Note that the edge case of the smallest possible value to deal with overflow was handled in this module.**

Next, we tested the preexp_and_presig module by passing in as input the converted signed magnitude number, and checked to make sure that it correctly gave as output the five-bit pre-significand value and the three-bit pre-exp value. As stated in the last section, we used a pre-significand value that is five bits long, as the fifth bit is used for rounding purposes, which is handled in our round_data module. In addition, we have a pre-exp value as our round_data module deals with a special case (i.e. when all five of the pre-significand bits are 1) to deal with overflow.

| Test Case | Result | Pass/No Pass |
|---|---|---|
| Seven leading zeros or less | Properly shifts the five desired bits (pres-significand) and extracts them. Also, extracts pre-exp (based off of leading zeros) | Pass |
| Eight leading zeros or more | No shifting necessary, concatenates extra bit to make five bits. Pre-exp is 0 | Pass |

**Figure 10: Table showcasing the test cases used for the preexp_and_presig submodule. There were only two general test cases required for this module as shown in the table.**

Lastly, the round_data module was tested by giving as input our pre-significand and pre-exp values acquired from the preexp_and_presig module, and then making sure that it gives as output the four-bit significand and three-bit exp field values. In addition, we made sure that the proper rounding was done depending on the fifth pre-significand bit.

| Test Case | Result | Pass/No Pass |
|---|---|---|
| Pre-significand: any<br>Pre-exp: any<br>(except for edge cases) | Rounds according to the fifth bit of pre-significand, and gives four-bit significand. Exp is equal to pre-exp | Pass |
| Pre-significand: 11111<br>Pre-exp: any (except for 111) | Overflow case, so rounds up setting significand to 1000, and increments exp field by 1 | Pass |
| Pre-significand: 11111<br>Pre-exp: 111 | Overflow case, but already max significand and exp field, so leaves as is | Pass |

**Figure 11: Table showcasing the test cases used for the round_data submodule. This submodule had a special overflow edge case that needed special testing for. This overflow occurred when all five bits of the pre-significand were 1, so the exp and significand field needed to be adjusted accordingly.**

After testing each submodule in isolation, and making sure that they all worked properly, a final test was done on the entire system worked as a whole, and that all submodules complied to their interface specifications. As all of the test cases/edge cases were covered when testing the individual submodules, the same input values were tested with the top module to check for the correct outputs: sign-bit, significand, and exp fields.


**Figure 12: This waveform showcases the simulation for a subset of the test cases used to test our program. The image consists of the waveform with the input, linear_enc_sign (D), and the three outputs: sign (S), exp (E), and the significand (F). All values are in binary, but as shown, the program correctly reads in the 12-bit input value, and properly gets the most approximate 8-bit floating point counterpart.**

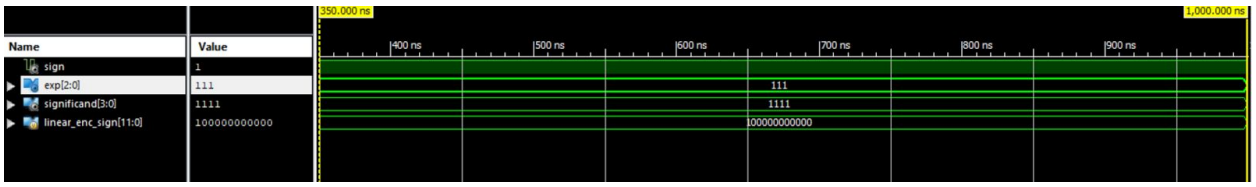


**Figure 13: This waveform showcases the simulation for one of the edge cases for this project: the smallest possible value (i.e. -2,048). The image consists of the waveform with the input, linear_enc_sign (D), and the three outputs: sign (S), exp (E), and the significand (F). As shown, our modular system correctly identifies that it is the smallest negative number, and then converts it to the most approximate 8-bit floating point value possible (i.e. -1,920).**

## CONCLUSION

This lab taught our group how to build a floating point converter using Verilog, and utilizing Xilinx to simulate our design. Unlike in previous labs, this was our first lab where we wrote all of the modules from scratch, and created our own test bench (despite being simple) to test our design implementation. For this project, we chose to create three submodules that handled each of the blocks discussed towards the end of the specification. Then, we used a top module to bring the three modules together, which read in a twelve-bit input value, and converted it to its eight-bit floating-point counterpart.

As this was the first lab where we actually wrote some Verilog code, there were some problems encountered along the way. However, this relatively simple lab was a nice chance to learn the Verilog syntax, and get familiar with the Xilinx ISE. For example, we had wrote all of our modules in class using a text editor, but when we went to compile it in Xilinx, there was a decent amount of error/warning messages. These errors/warnings were quickly resolved by reading the Xilinx error messages, and changing the code accordingly.

With this lab, our group had successfully created the required floating-point converter, first by testing it with our own testbench in simulation, and then using the TA's testbench during demo day.