
CS M152A: Introductory Digital Design Laboratory

# Lab #1

# Floating Point Conversion

Name: Andrew Juarez, Atibhav Mittal

UID: 504-572-352, 804-598-987

Lab Section: 4

Due Date: Oct 19, 2017

## 1.1: Introduction and Requirement

The goal of this lab was to convert a 12-bit two's complement analog to an 8-bit floating-point representation. The output format of the 8-bit floating point representation is as follows: the most significant bit represents the sign bit (S), the next three bits represent the exponent (E), and the least significant four bits represent the significand (F). In mathematical form, it is shown by the following equation:

$$V = (-1)^S * F * 2^E$$

The design consisted of five modules. One module was the framework for the design and instantiated three modules. These three modules converted the 12-bit two's complement to a 12-bit signed magnitude representation, converted the 12-bit signed magnitude representation to a floating-point representation, and performed rounding. The last module was a priority encoder that helped in converting the 12-bit signed magnitude representation to a floating-point representation.

There were a few requirements on how to determine the three outputs of the floating-point unit. The sign bit was extracted from the most significant bit of the 12-bit input signal. The significand was extracted from the four bits following the last leading zero in the signed magnitude representation; however, rounding was used based on the fifth bit following the last leading zero which could affect the significand and exponent and is described in the Design Description section. The exponent was determined by counting the number of leading zeroes in the signed magnitude representation and referencing Figure 3. Each of these requirements were implemented under the top-level design module FPCVT.

## 1.2: Design Description

The design has 4 major modules:

- FPCVT
- twos_complement_to_sign_magnitude
- linear_to_floating_point
- rounding

The main module is FPCVT, and it instantiates the other 3 modules. The logic flow within these three modules is shown in Figure 1.

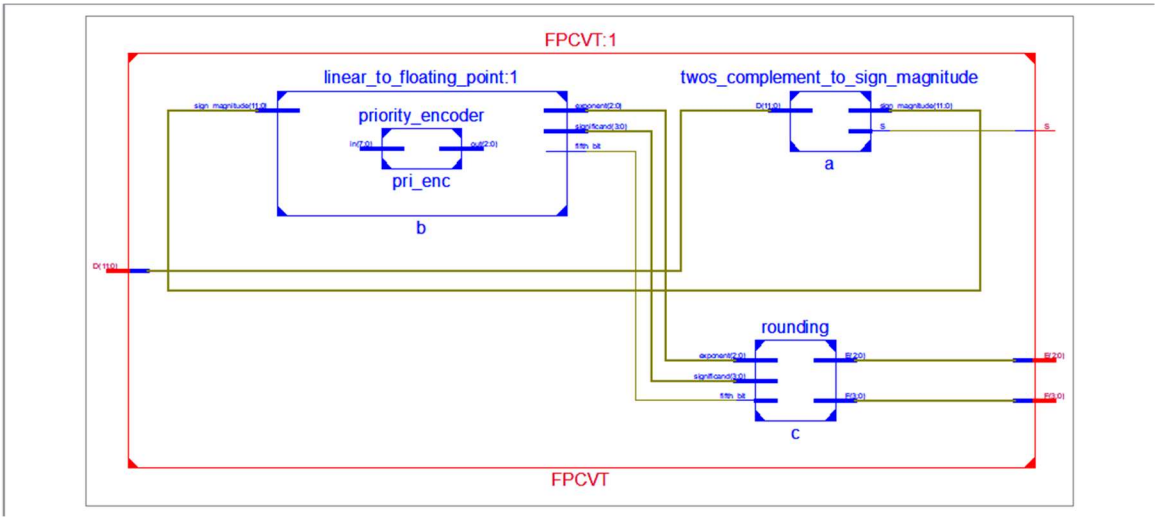


**Figure 1: Overall Design of FPCVT:** The above figure shows the general schematic of the Verilog code. In the middle left, the input D enters FPCVT and connects to the input of the twos_complement_to_sign_magnitude in the top right corner. One of the outputs of this module S is the direct output of FPCVT. The other output sign_magnitude gets sent as input to linear_to floating_point, which is the module on the top left. The three outputs of linear_to_floating_point are all provided as inputs to rounding. Finally, the two outputs of rounding, E and F, are outputs of the FPCVT module.

Each module has its own function as defined below. The first step in the process is to extract the sign bit and the magnitude of the two's complement number. The sign bit is outputted as is, and the magnitude is sent to the "linear_to_floating_point" module. This module converts it to an approximate floating-point number with some rounding errors. It outputs the approximate exponent, fraction and the fifth bit; all of which is fed to the rounding module. The rounding module corrects the fraction as described in the section below. The schematic in Figure 1 shows the modular architecture of the Verilog code.

# FPCVT:

| Input | Output |
|---|---|
| • D | • S<br>• E<br>• F |

**Table 1: FPCVT Module Interface:** The above table shows the interface of the FPCVT module. This is the outermost module. It takes in a 12-bit two's complement integer as input. It compresses it to a 8-bit floating point number. It produces a 1-bit output that is the sign (S), a 3-bit exponent (E) and a 4-bit significand (F).

This module is the outermost module for our design. It takes as input a 12-bit 2's complement integer, and produces a compressed 8-bit floating-point representation of this number. FPCVT does this by instantiating the 3 modules: twos_complement_to_sign_magnitude, linear_to_floating_point, and rounding.

## twos_complement_to_sign_magnitude:

| Input | Output |
|---|---|
| • D | • S<br>• sign_magnitude |

**Table 2: Two's Complement to Sign Magnitude Interface**: The above table shows the interface of the twos_complement_to_sign_magnitude module. The module takes a 12-bit two's complement integer (D) as input. It then produces as output, a one bit number that is the sign of the two's complement number (S) and a 12-bit number which is the magnitude of the two's complement number (sign_magnitude).

This module works by extracting the MSB of the input D. If the MSB is a 0, then S is assigned a 0, because our input is positive, and so should our final floating point number. In this case, sign_magnitude remains unaffected and the output sign_magnitude is just D. If the MSB is a 1, then S is assigned a 1, because our input is negative, and our final output should also be negative. In this case, sign_magnitude is calculated by negating D, according to two's complement rules (flipping all the bits and adding 1).

The output sign_magnitude is then fed into the module linear_to_floating_point. S becomes the output of the outermost module, FPCVT.

## priority_encoder:

| Input | Output |
|---|---|
| • in | • out |

**Table 3: Priority Encoder Module Interface:** The above table shows the interface of the priority encoder module. It takes in as input an 8 bit number (in) and outputs a 3 bit number (out).

This module is a 8→3 priority encoder. It takes an 8 bit number as input, and outputs the position of the most-significant high bit. The truth table and logic for this is shown in Figure 2.

| Inputs | | | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ | $Y_2$ | $Y_1$ | $Y_0$ |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| × | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| × | × | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| × | × | × | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| × | × | × | × | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| × | × | × | × | × | 1 | 0 | 0 | 1 | 0 | 1 |
| × | × | × | × | × | × | 1 | 0 | 1 | 1 | 0 |
| × | × | × | × | × | × | × | 1 | 1 | 1 | 1 |

$$z_2 = x_7 + \bar{x}_7 x_6 + \bar{x}_7 \bar{x}_6 x_5 + \bar{x}_7 \bar{x}_6 \bar{x}_5 x_4$$
$$z_1 = x_7 + \bar{x}_7 x_6 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 x_3 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 x_2$$
$$z_0 = x_7 + \bar{x}_7 \bar{x}_6 x_5 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 x_3 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 \bar{x}_2 x_1$$

**Figure 2: Truth Table and Logic Functions for 8-3 Priority Encoder:** The image on the left is the truth table for an 8-3 priority encoder. [2] The image on the right contains the logic functions for the 8-3 priority encoder. [3]

# linear_to_floating_point:

| Input | Output |
|---|---|
| • sign_magnitude | • exponent<br>• significand<br>• fifth_bit |

**Table 4: Linear to Floating Point Interface:** The above table shows the interface of the module linear_to_floating_point. The module takes in a 12-bit unsigned number (sign_magnitude). It produces as output a 3-bit number (exponent), a 4-bit output (significand) and a 1 bit number (fifth_bit). The way these outputs is produced is described below.

The module works by first extracting the number of leading zeros in the sign_magnitude number. This is done using a priority encoder, which is described above. Once we know the number of leading zeros, we set the exponent according to the mapping given in Figure 3.

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

**Figure 3: Mapping from Leading Zeroes to Exponent:** The above figure describes the mapping between the number of leading zeros in the unsigned number and the value of exponent.[1]

Once we know the exponent, we extract the next 4 bits, and set that as our significand. If there is a 5$^{th}$ bit, we extract that and set that as fifth_bit, else we set fifth_bit as 0. In the special case that

our input is -2048 (in decimal), we set exponent and significand to the maximum possible value, that is 3'b111 for exponent and 4'b1111 for significand and set fifth_bit as 0.

The outputs exponent, significand and fifth_bit are pipelined to the rounding module.

## rounding:

| Input | Output |
|---|---|
| • exponent<br>• significand<br>• fifth_bit | • E<br>• F |

**Table 5: Rounding Interface:** The above table describes the interface of the rounding module. It takes as input a 3-bit exponent, a 4-bit significand and a 1-bit number that represents the fifth bit. It produces as output, the final exponent E (3 bits), and the final significand F (4 bits).

The rounding module performs rounding of the floating-point number as described in the Lab Manual. It takes as input the exponent, significand and the fifth bit as produced in linear_to_floating_point, and outputs the final exponent and floating point after performing rounding.

To perform rounding, we follow these steps. If the fifth bit is 0, we set E as exponent and F as significand, without making any changes. If the fifth bit is a 1, we add it to the significand. In the case that the addition doesn't overflow, we set F as this new significand value and set E as exponent. However, if it does overflow, we right shift the significand by 1, add 1 to the exponent, and assign E and F as the updated exponent and significand respectively. If adding 1 to the exponent, results in overflow, we set E and F as the maximum possible value. (E = 3'b111, F = 4'b1111).

## 1.3: Simulation and Documentation

The first module tested was twos_complement_to_sign_magnitude. The first case handled was the case of a positive number. In this case, the sign bit should be 0 and the number should be unchanged in the conversion. A for loop was used to iterate through thousands of positive numbers and they were checked to see if the correct result was given. The next case to check for was 0. For this case, the sign bit was 0 and the expression remained unchanged. The last case was the case of negative numbers. For this case, the output should be complemented bitwise and incremented. The special case of -2048 which is all 1s was handled in linear_to_floating_point.

The next module tested was linear_to_floating_point. For the input, the number of leading zeroes was varied for cases from 0 leading zeroes to 12 leading zeroes. The result of the exponent was checked with Figure 3. The significand was checked by displaying the four bits

following the last leading zeroes. The fifth bit was checked by looking at the bit after the four bits. A couple of special cases were tested. One was the case of all zeroes. The output was ensured to be all zeroes. Another special case tested was 1000_0000_0000. Initially, we did not know the correct output for this case; however, after consulting with the TA, we found that the correct output should be E=1111, F=111. After modifying some code, this special case was handled successfully.

To test the priority encoder module, nine test cases were used. The first test case was all zeroes. The following test cases varied the number of leading zeroes. The priority encoder would output the most significant bit that had a 1. Further variations of the bits after the first 1 bit were tested to ensure that they do not affect the output.

To test the rounding module, a for loop was used to test all possible cases of the inputs exponent, significand, and fifth_bit. The outputs of the edge cases were checked by hand with the outputs calculated. Moreover, random test cases in between were selected and checked with the outputs calculated by hand.

Finally, to test the FPCVT module, we chose around 10 test cases: positive numbers, negative numbers, 0, the largest negative number, and the largest positive number. The correct results were computed by hand and compared. One bug was found during the process. After simulating the test cases, each bit was in a state of high impedance. After tracing through the FPCVT module code, the outputs were declared as registers, which was giving us the error. After fixing this bug, the outputs were as expected.

## 1.4: Conclusion

The design described above successfully met the specifications and worked for all edge cases. Dividing up the code into separate modules enabled us to unit test and makes it possible to modify a module in the future without affecting any of the other modules.

We faced an obstacle in getting used to how Verilog simultaneously executes multiple statements that are not separated by time delays. We faced errors while testing the modules that included a display statement. The output printed to the console didn't match the waveform that the code created. We worked through the code multiple times to figure out the bug. Another challenge we faced was modifying the standard priority encoder to an encoder give us the leading number of 0s instead of the position of the most significant high bit. We dealt with this issue by working through a couple of examples on paper, seeing the output of a standard priority encoder and what the output of the modified priority encoder should be. This enabled us to come up with a conversion that worked well.

Overall, this lab was a good introduction to writing complex modules in Verilog by breaking it up into smaller modules.

Bibliography


1. Potkonjak, M. Computer Science M152A: Introductory Digital Design Laboratory Lab Manual. (Univ. California Los Angeles, Los Angeles, California).
2. www.electronicshub.org/wp-content/uploads/2015/06/Octal-to-Binary-Priority-Encoder.jpg.
3. "Encoders/Decoders." *Encoders/Decoders*, www.cs.umd.edu/class/sum2003/cmsc311/Notes/Comb/encoder.html.