CS M152A Lab1

Floating Point Conversion

In this lab, you will learn how to use the Xilinx ISE program to design and test a floating point converter.

Introduction

For this lab, you will use the Xilinx ISE software to design and test a **combinational circuit** that converts a 13-bit linear encoding of an analog signal into a compounded 9-bit Floating Point (FP) Representation.

This lab will be based on simulation only; but you still need to complete the implementation steps (except editing ucf file and programming the FPGA) to generate the design summary report. At the end of the lab, you are expected to present a design project with source code and test bench, and the design will be tested against a test bench that runs through all possible input patterns.

Overview

The module for the floating-point conversion (called FPCVT). The inputs and outputs of the FPCVT logic block should in the following table:

FPCVT Pin Descriptions	
D [12 : 0]	Input data in Two's Complement Representation. D0 is the Least Significant Bit (LSB). D12 is the Most Significant Bit (MSB).
S	Sign bit of the Floating Point Representation.
E [2 : 0]	3-Bit Exponent of the Floating Point Representation.
F [4 : 0]	5-Bit Significand of the Floating Point Representation.

Background

Analog signals are often converted to digital form for storage or transmission. A linear encoding using 8 bits can represent the unsigned number within the range 0 – 255 or a signed number within the range -128 to +127 using Two's Complement representation. Seven or eight bits of precision is adequate for intelligible speech or music almost good enough to listen. However, seven or eight bits do not provide sufficient dynamic range to capture both loud and soft sounds. Therefore, nonlinear encodings are used in most commercial systems.

Output Format:

For this laboratory assignment, we will use a simplified Floating Point Representation consisting of a 1-Bit Sign Representation, a 3-Bit Exponent, and a 5-Bit Significand (the significand is sometimes called the *mantissa*).

8	7	6	5	4	3	2	1	0
S	E			F				

The value represented by an 8-Bit Byte in this format is:

$$\mathbf{V = (-1)^S \times F \times 2^E}$$

The **S**-Bit signifies the **Sign** of the number. The 5-Bit **Significand**, **F**, ranges from [00000] = 0 to [11111] = 31, and the 3-Bit **Exponent**, **E**, ranges from [000] = 0 to [111] = 7. The following table shows the values corresponding to several Floating Point Representations.

Floating Point Representation Examples

Floating Point Representation	**Formula**	**Value**
[0 000 00001]	1×2^0	1
[1 010 11010]	-26×2^2	-104
[0 010 11110]	30×2^2	120

The last two rows of the above table demonstrate that some numbers have multiple Floating Point Representations. The preferred representations are the ones in which the Most Significant Bit of the Significand is 1. This representation is said to be the **normalized** representation. It is quite straightforward to produce the linear encoding corresponding to a floating-point representation; this operation is called *expansion*.

The goal of this laboratory assignment is to build a combinational circuit for the inverse operation, called *compression*. A device that performs both expansion and compression is called a compounder. The compression half of a compounder is more challenging because there are more input bits than output bits. As a result, many different linear encodings must be mapped to the same Floating Point Representation. Values that do not have Floating Point Representations should be mapped to the closest Floating Point encoding; this process is called *rounding*.

Input Format:

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

The Significand consists of the 5 bits immediately following the last leading 0. When the exponent is 0, the Significand is the Least Significant 5 bits. For example, 422 = [0_0001_1010_0110] has 4 leading zeroes, including the sign bit, thus its Exponent is 4 (according to the table above), and its Significand is [11010]. In case of a negative number which is in the 2's complement format, you should first negate it, and then count the number of leading zeroes. For example, -422 = [1_1110_0101_1010], after negation it will become 422 = [0_0001_1010_0110], thus it also has 4 leading zeroes.

This FP representation expands to 26 x 2^4 = 416. The number 422 cannot be represented exactly, so it is represented with an error of about 1.5%.

Rounding

The procedure presented above produces the correct Floating Point Representation for about half the possible linear encodings. However, it does not guarantee the most accurate representation. The circuit that you will design is required to round the linear encoding to the nearest Floating Point encoding. You should use the simple rounding rule that depends only on the 6th bit following the last leading 0. Recall that the first 5 bits following the last leading 0 make up the Significand. The next (6th) bit then tells us whether to round up or down. If that bit is 0, the nearest number is obtained by truncation – simply using the first 5 bits. If, on the other hand, the 6th bit is 1, the representation is obtained by rounding the first 5 bits up – by adding 1.

The following table gives examples of rounding:

Rounding Examples		
Linear Encoding	**Floating Point Encoding**	**Rounding**
0000001101100	[0 010 11011]	Down
0000001101101	[0 010 11011]	Down
0000001101110	[0 010 11100]	Up
0000001101111	[0 010 11100]	Up

The rounding stage of Floating Point conversion can lead to a complication. When the maximum Significand [11111] is rounded up, adding one causes an overflow. The result, 100000, does not fit in the 5-Bit Significand field. This problem is solved by dividing the Significand by 2, or shifting right, to obtain 10000, and increasing the Exponent by 1 to compensate.

For example:

0000011111101	0	3	100000	OOPS!	0	4	10000

In this example, 253 is converted to 16 x 2^4 = 256, which is indeed the closest Floating Point number. Note that the overflow possibility can be detected either before or after the addition of the rounding bit.

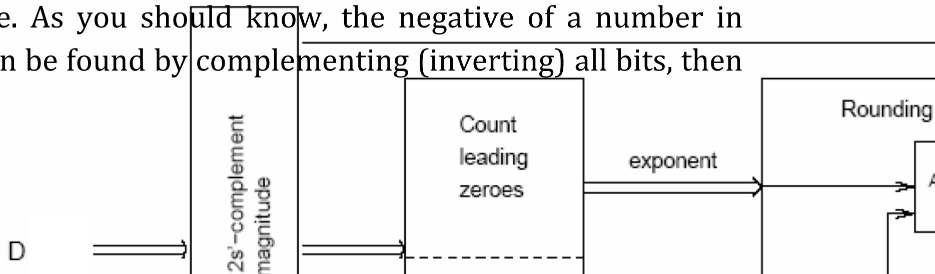
When rounding very large linear encodings, such as 4095 = [0_1111_1111_1111], the exponent may be incremented beyond 7, to 8, which cannot be stored in the exponent field. Our solution to this problem is to **use the largest possible Floating Point Representation**.

Overall Design

Note: the diagram below is just a recommendation, your actual design may follow a different architecture of your choice.

An overall block diagram for the floating-point conversion circuit is shown below:

The first block converts the 13-bit two's-complement input to sign-magnitude representation. Nonnegative numbers (sign bit 0) are unchanged, while negative numbers are replaced by their absolute value. As you should know, the negative of a number in twos'-complement representation can be found by complementing (inverting) all bits, then



incrementing (adding 1) to this intermediate result. One problem case is the most negative number, -4096 = (1_0000_0000_0000); when complement-increment is applied, the result is 1_0000_0000_0000, which looks like -4096 instead of +4096. Make sure you handle it well.

The second block performs the basic linear to Floating Point conversion. The Exponent output encodes the number of leading zeroes of the linear encoding, as shown in table A above. To count the leading zeroes, we recommend you implement a **priority encoder**. The Significand output is obtained by right shifting the most significant input bits from bit positions 0 to 7. What this means is that each bit of the Significand comes from one of 8 possible magnitude bits.

The third block performs rounding of the Floating Point Representation. If the 6th bit following the last leading 0 of the intermediate Floating Point Representation is 1, the Significand is incremented by 1. If the Significand overflows, then we shift the Significand right one bit and increase the Exponent by 1.

Deliverables

When you finish, the following should be submitted for this lab:

1. **Verilog source code** for the "FPCVT" module. The file should be named exactly as "FPCVT.v" and the module and port names should exactly match names defined in the Overview section (see the table on the first page). It is very important as your code is automatically evaluated. Also note that, this code should be completely synthesizable and define all submodules within the same verilog file (i.e., FPCVT.v contains the definition of your top module and all submodules used).
2. **Verilog testbench** you used to evaluate your design. Note that your testbench is graded based on the cases it covers and also ideas used to make sure all important corner cases are being covered. Please name the file "testbench_UID.v" where UID is your UCLA ID.
3. **Lab Report** should be consisting of explanations about your module and testbench design as well as ISE Design Overview Summary Report. Explain ideas you used to implement different blocks and how you test your design (e.g., which corner cases you consider?). Make sure to include schematics of your design that are drawn by hand, not using the ISE tool . Please name your report "UID.pdf" where UID is your UCLA ID. Simulation outputs should be included in the report as well.
4. **Video Demo** a 10 minute video showing your screen while you are explaining your design logic as well as simulation outputs

Note: please upload items 1-4 above on CCLE submission form as separate files (do NOT compress everything in a single file). CCLE limits the total submission size to 100MB.