# CS M152A - Introduction to Digital Design Lab

## Project 2: Floating Point Conversion

Arnav Garg - 304911796

## Introduction and background:

This lab is aimed towards implementing a combination circuit to simulate the conversion of a 13-bit two's complement analog input signal into a 9-bit floating-point "digital" output representation, which is represented as three separate output signals - Signed Bit (1-bit), Exponent (3-bit) and Mantissa (5-bit). This is often done to convert the analog signals into some sort of digital form for storage or transmission. Together, these three signals are related to the input 13-bit signal value (V) in base-10 using the relationship,

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

where S represents the Signed Bit, E represents the Exponent, and F represents the Mantissa.

This lab only focuses on implementing a top-level module called *FPCVT* in Verilog and creating a testbench to test the module using the simulation waveforms. It did not require editing a ucf file or programming the FPGA itself.

Two's Complement Representation (2C): Two's complement is used to represent signed integers. An n-bit two's complement binary representation can represent numbers within the range $[-2^{N-1}, 2^{N-1} - 1]$, where the most significant bit (MSB) is usually used to indicate the sign. If the MSB is 1, then the integer being represented is negative, otherwise, the integer being represented is positive. The MSB also usually has a weight of $(-2)^{N-1}$, and subsequent bits in the n-bit representation have weights $2^{N-p}$, where p represents the $p^{th}$ index from the left in the bit-array starting right after the MSB. In our case, given the 13-bit input signal *input*, the base-10 equivalent can be calculated using:

$$V(base-10) = input[12] * (-2)^{12} + \sum_{k=0}^{11} input[k] * 2^k$$

where input[0] is the LSB and input[12] is the MSB.

Floating Point Representation (FPR): The floating-point representation can be used to compress a two's complement representation by using fewer bits. In particular, floating-point representations usually consist of 3 parts: A signed bit, indicating the sign of the two's complement integer, an exponent, and a mantissa. However, like with any form of compression, not all 2's complement representations have a unique representation in the floating-point form, therefore multiple two's complement integers can inadvertently map to the same floating-point representation.

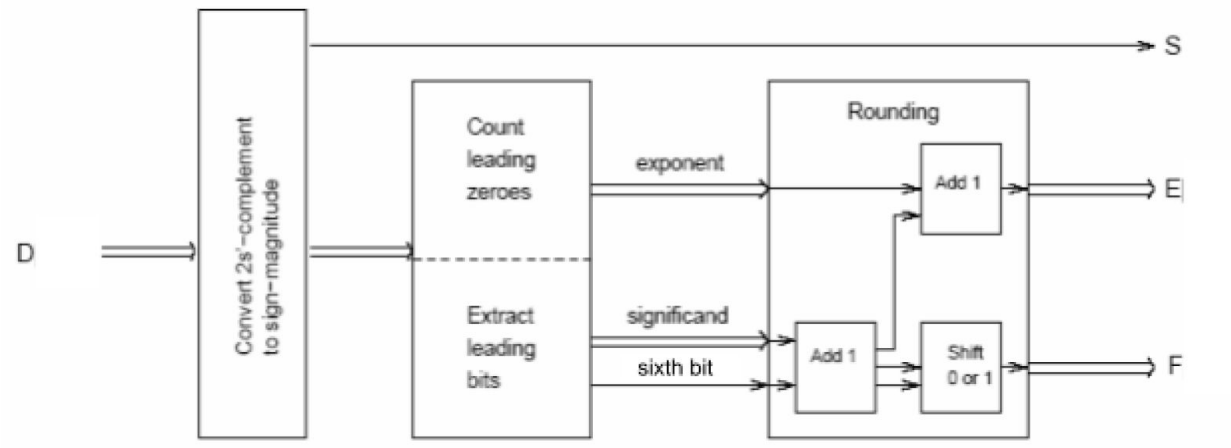
## Design Description:

For the *FPCVT* module that needs to be designed, we are required to convert a 13-bit 2C input to a 9-bit FPR output. For this module, the table below lists the input and output pins.

| FPCVT Pin Descriptions | |
|---|---|
| D [12:0] | Input: Signed 2C representation. 13 bits. |
| S | Output: Sign bit of the FPR |
| E [2:0] | Output: 3-bit Exponent of the FPR |
| F [4:0] | Output: 5-bit Mantissa of the FPR |

For the input pin D, D0 is the LSB, and D12 is MSB.

The overall block diagram for the floating-point conversion circuit then looks like the following,




where D represents the input net, the first block converts the 2C representation a sign-magnitude representation (converts D to absolute value). Two operations are performed on this intermediate result: the first is counting the number of leading zeros that will help calculate the value for the output net exponent E, and the leading bits will help keep track of the Mantissa as well as the sixth bit to used for rounding. Finally, once we extract the Mantissa, Signed Bit, and Exponent, we round our representation up or down based on the value of the signed bit. This may result in a variety of different edge cases due to bit overflows, which must be carefully handled.

My module, *FPCVT,* therefore largely does this overall operation in 3 steps as well:

1. Find the sign-magnitude form of the 2C representation. If the sign is negative, invert the bits and add 1 to ensure we can use the absolute value. Extract the signed bit.

2. Match the bit pattern of the magnitude to detect the number of leading 0s. This can be used to calculate the exponent. Use this to determine which section of the bit-array, D, we need to extract to get the Mantissa, and also extract the sixth bit for rounding.
3. Check for the need for rounding based on the value of the 6th bit. If it is 0, there is no need to round sound our representation rounds down by default. If the sixth bit is 1, divide the Mantissa and increment the Exponent. If either overflows, handle them carefully [Further descriptions of what causes these edge cases and how they were handled is explained below].

Therefore, the top-level module *FPCVT* could have been split into three modules - one for inversion, one for extraction of the Mantissa and Exponent, and one for rounding. However, I found it easier to model these into one compact module, with clearly demarcated segments. This helped me streamline all aspects of the design requirements, and be able to assess the overall flow required for this combinational circuit to work.

The above diagram is further explained here, along with other major considerations that need to be made for this Project:

1. **Sign Detection and Inversion**
   To detect the sign bit, we need to look at the MSB of the input net D. Therefore, we can just index D[12] and assign the value of this bit to the output register S.

2. **Leading Zeros and Exponent**
   To calculate the exponent segment of the FPR, we need to count the number of leading zeros in the 2C representation according to the algorithm provided in the spec.

| Leading Zeros | Exponent |
|---|---|
| 1 | 7 |
| 2 | 6 |
| 3 | 5 |
| 4 | 4 |
| 5 | 3 |
| 6 | 2 |
| 7 | 1 |
| >=8 | 0 |

However, before we can continue with counting the number of leading zeros, we must use the absolute value of the input net D. To get the absolute value (or the positive) equivalent of the 2C representation of an integer, we simply need to follow the algorithm below:

*if (D[12] does not equal 0):*
*Abs = ~D + 1;*

where Abs represents the absolute/positive value of the data input. Then, by simply using some sort of pattern matching, either via regex or using chunks of the bit-array from the data input, we can determine the number of leading zeros and assign the exponent 3-bit output register accordingly.

**3. Mantissa**

To calculate the Mantissa, we simply extract the 5 bits following the last leading zero after "truncation" of the leading zeros.

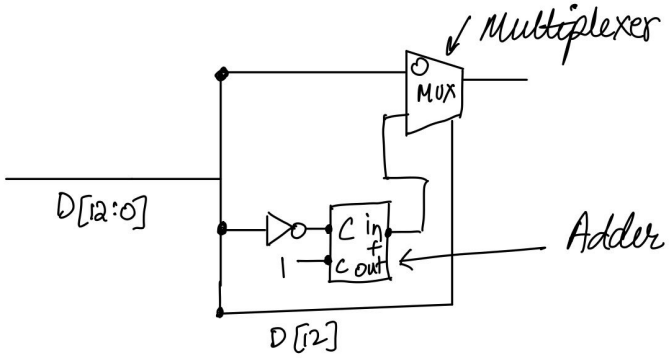
**4. Rounding**

Since we're compressing 13-bits into 9-bits, there's bound to be collisions in mappings, that is, there will be cases where two or more 13-bit 2C representations map to the same 9-bit representation. Therefore, this will result in close "approximations", and in some cases, a deviation of up to 1.5% from the actual decimal representation of the 13-bit 2C integer. Therefore, to calculate the closest FPR representation for D, we need to round the representation *up* or *down*. In particular, based on the algorithm described in the specification for this project, we look at the 6th bit following the last leading 0 (or the bit immediately following the Mantissa). If this bit has a value of 0, then we round *down*, which is the equivalent of not changing our current Exponent or Mantissa. However, in the instance that this bit is a 1, we need to round *up*, which involves incrementing the Mantissa by 1.

While this is not a problem in most cases, incrementing the Mantissa by 1 can cause an overflow of bits, particularly in the case where the Mantisssa, F, is 11111 (31), where an increment would cause an overflow to 100000. To make up for this, we effectively divide the Mantissa by 2 so that F = 10000 and increment the Exponent by 1. This could still result in a potential overflow in the Exponent, which would happen if the Exponent is III (7), resulting in an Exponent equal to 1000 (8). To combat this, we can represent the output as the largest possible FPR representation. This makes sense, since the only case where we would have both a Mantissa and Exponent overflow is for very large numbers, and almost particularly where there are no leading zeros in the absolute value-form of D, which happens when the input is D = 1_0000_0000_0000 (-4096) since its absolute value is equal to the original 2C value using the algorithm described above, or in the case of D = 0_1111_1111_1111 (4095), where there is only one leading zero.

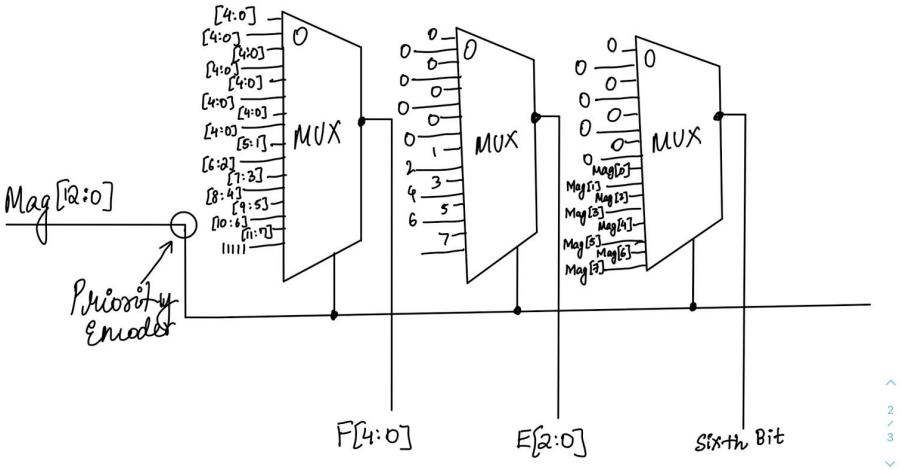
**Module Breakdown**:
As mentioned above, my module does the required conversion in 3 steps.

1. Find the sign-magnitude form of the 2C representation. If the sign is negative, invert the bits and add 1 to ensure we can use the absolute value. Extract the signed bit.




This part of the module can be represented as a 1bit multiplexer, where D[12] serves as the select-in and based on whether it is positive or negative, we process D[12:0] accordingly. In the case that D[12] is 1, we negate D[12:0] and pass it through an added with 1, and then send this to the channel in the multiplexer corresponding to D[12] = 1.

2. Match the bit pattern of the magnitude to detect the number of leading 0s. This can be used to calculate the exponent. Use this to determine which section of the bit-array, D, we need to extract to get the Mantissa, and also extract the sixth bit for rounding.

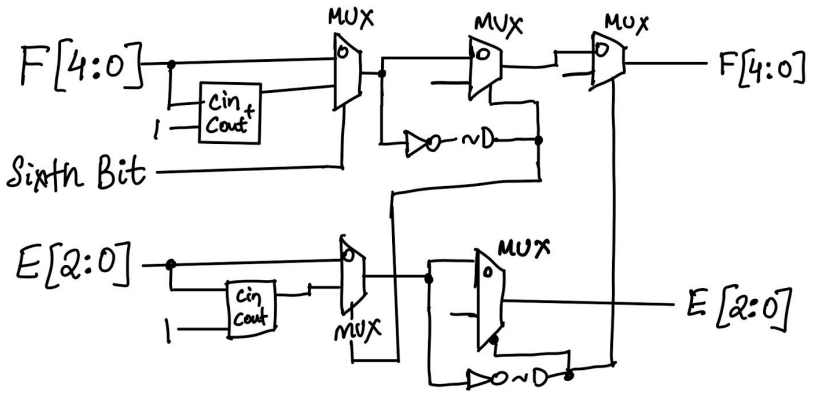



This part of the module can be represented as a collection of MUXes that act as selectors for the Mantissa, Exponent, and Sixth Bit.

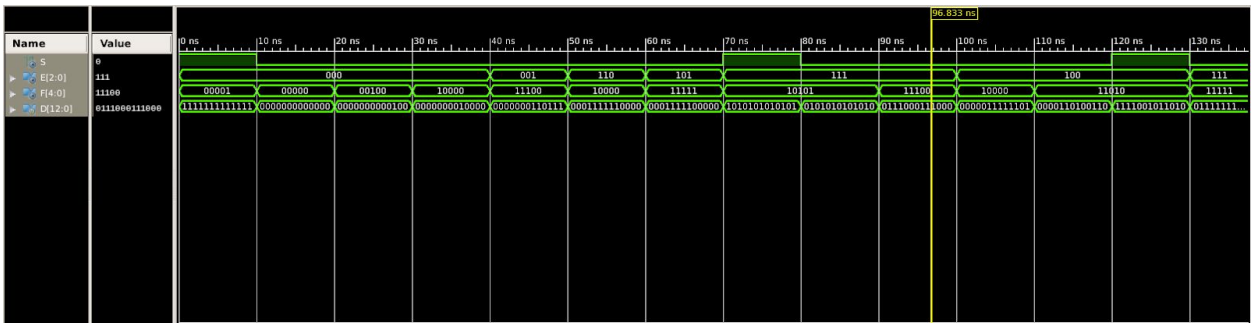
3. Check for the need for rounding based on the value of the 6th bit. If it is 0, there is no need to round sound our representation rounds down by default. If the sixth bit is 1,

divide the Mantissa and increment the Exponent. If either overflow, handle them carefully.

The rounder uses the 6th bit to determine whether or not the Mantissa or Exponent will overflow, or whether they can simply be incremented. It handles all overflows and exponent overflow exceptions as required.




**Simulation:**




For the testbench, I tested the following test cases and these were the outputs:

```
D=1  1111  1111  1111 [-1], S=1, E=000, F=00001, V=-1
D=0  0000  0000  0000 [0], S=0, E=000, F=00000, V=0
D=0  0000  0000  0100 [4], S=0, E=000, F=00100, V=4
D=0  0000  0001  0000 [16], S=0, E=000, F=10000, V=16
D=0  0000  0011  0111 [55], S=0, E=001, F=11100, V=56
D=0  0011  1111  0000 [1008], S=0, E=110, F=10000, V=1024
D=0  0011  1110  0000 [992], S=0, E=101, F=11111, V=992
D=1  0101  0101  0101 [-2730], S=1, E=111, F=10101, V=-2688
D=0  1010  1010  1010 [2730], S=0, E=111, F=10101, V=2688
D=0  1110  0011  1000 [3640], S=0, E=111, F=11100, V=3584
D=0  0000  1111  1101 [253], S=0, E=100, F=10000, V=256
D=0  0001  1010  0110 [422], S=0, E=100, F=11010, V=416
D=1  1110  0101  1010 [-422], S=1, E=100, F=11010, V=-416
D=0  1111  1111  1111 [4095], S=0, E=111, F=11111, V=3968
```

| |
|---|
| D=1 0000 0000 0001 [-4095], S=1, E=111, F=11111, V= -3968<br>D=1 0000 0000 0000 [-4096], S=1, E=111, F=11111, V=-3968 |

In particular, the following cases are interesting:

| | |
|---|---|
| 253 | D=0000011111101, which results in E = 3 and F = 11111. The sixth bit is 1, so we round up, but since this causes a Mantissa overflow, we divide the Mantissa by 2 and increment Exponent by 1 -> E = 4, F = 10000. |
| -4096 | This is an exceptional case because even though the signed bit is 1, negative it and adding 1 to get the sign-magnitude representation results in the same representation. This has no leading zeros, therefore, to express this correctly as the largest possible negative FPR, we set F to 11111, E to 111 and S to 1, resulting in -3968. |

### ISE Design Overview Summary:

```
=================================================================================
*                            Design Summary                                     *
=================================================================================

Top Level Output File Name         : FPCVT.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 92
#      GND                         : 1
#      INV                         : 12
#      LUT1                        : 1
#      LUT3                        : 15
#      LUT5                        : 14
#      LUT6                        : 21
#      MUXCY                       : 12
#      MUXF7                       : 2
#      VCC                         : 1
#      XORCY                       : 13
# IO Buffers                       : 22
#      IBUF                        : 13
#      OBUF                        : 9

Device utilization summary:
---------------------------

Selected Device : 7a100tcsg324-3

Slice Logic Utilization:
```

```
 Number of Slice LUTs:                   63  out of  63400     0%
    Number used as Logic:                63  out of  63400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     63
   Number with an unused Flip Flop:      63  out of     63   100%
   Number with an unused LUT:             0  out of     63     0%
   Number of fully used LUT-FF pairs:     0  out of     63     0%
   Number of unique control sets:         0

IO Utilization:
 Number of IOs:                          22
 Number of bonded IOBs:                  22  out of    210    10%

Specific Feature Utilization:
```

From the design overview summary, we can see that there were no errors or warnings that were detected. Additionally, we can see that there's approximately 22 IO operations that are performed. Hardware resources on an FPGA are indicated by the number of slices that FPGA has, where a slice is comprised of look-up tables (LUTs) and flip flops. The number of LUTs and flip flops that Xilinx defines to make up a single slice is different based on the family of the chip. In our case, the total number of Slice LUTs are 63400, out of which we only use 63 =~ 0%. This module that was designed also 0 LUT flip-flop pairs, and uses approx 10% of the available input/output buffers (IOBs), indicating that we're using approximately 10% of the pins on the Nexys3 board/device. 22 is a result that comes from the total number of input and output pins being used: 13-bit input + 1 signed bit + 3 exponent bits + 5 Mantissa bits = 22 I/O buffers.

## Conclusion:

Therefore, through a 3 step systematic process, we're able to convert a 13-bit two's complement representation to a 9-bit floating-point representation. This is able to handle cases for rounding and bit overflows caused by it. It also successfully rounds numbers to within a 1.5% error range, which is a very acceptable signal loss given the 4-bit compression this conversion offers.

One of the challenges I faced with this project was trying to count the leading zeros. My first approach was to try using left shifts to eliminate zeros and count them incrementally, which seemed to difficult. I then tried creating a "counter" that gets incremented using a for a loop.

```
// 2. Match pattern to detect the number of leading 0s and mantissa
while (mantissa[i] != 1'b1)
begin
       leadingZeros = leadingZeros + 1;
       i = i - 1;
end
```

```
            ; Kept returning the wrong value
```

However, I learned that this looked too much like C code and makes no sense in terms of a hardware definition language. That's when I decided to change my approach and use pattern matching via if-else statements.

Another difficulty I faced was tracking all of the edge cases, especially -4096 because I had not considered that there could a case with no leading zeros. This took me a while to detect and required me to change one of the conditions in my if-else priority encoder.