Lab 2 Report

February 11, 2024

Instructor: Majid Sarrafzadeh, Chenda Duan

Myra Yamazakin (805618899), Haoxuan Li (705738656)

Table of Contents:

I. Introduction and Technical Specification

This lab focuses on the design and application of combinational circuits that convert 12-bit linear encoding of an analog signal into a 8-bit Floating Point representation. It is given that the 12-bit linear encoding is in Two's Complement representation, so it'll represent any integer ranging from -128 to 127.

The input of the program is a 12-bit linear encoding represented in Two's Complement.a
The output of the program is a 8-bit simplified Floating Point representation. The Floating Point representation consists of 3 parts:

1. S: Sign (1 bit)
    - 0 indicates positive and 1 indicates negative
2. E: Exponent (3 bits)
    - ranges from 000 (0) to 111 (7)
3. F: Significand (4 bits)
    - ranges from 0000 (0) to 1111 (15)

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

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

Each components of inputs/outputs can be summarized as the table below:

FPCVT Pin Descriptions	
D [11 : 0]	Input data in Two's Complement Representation. D0 is the Least Significant Bit (LSB). D11 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 [3 : 0]	4-Bit Significand of the Floating Point Representation.

Graph 1: FPCVT inputs and outputs

One of the challenges presented in this lab is that we are trying to implement combinational circuits for inverse operation - compression. Since the input bits are more than the output bits, we need to map input values by rounding.

The specs give some guideline on how to find the outputs:

Sign

The specs don't provide any guideline on how to determine the sign, but since the input is given in Two's Complement representation, the most significant bit should determine the sign.
If the most significant bit is 1, S = 1 (since it's negative) and if it's 0, S = 0 (since it's negative).

Exponent

The exponent is determined from the number of leading zeros. If the input is a negative number, it must be converted to the corresponding positive value first (negate and add 1).
The table below shows the appropriate exponent according to the number of leading zeros:

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

Graph 2: Value of exponent based on leading zeros

Significand

The significand is found after the leading zeros. Ensure that the significand is 4-bit long. If the bits after leading zeros are less than 4-bit long, take the last 4 bits (4 least significant bits) from the input.

Rounding process must be taken care of in this step. The 5th bit after the leading zero tells whether or not to round; if the 5th bit is 0 do not change the significand, and if it's 1 add 1 to the significand. Table below shows examples to demonstrate:

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

Graph 3: Rounding Examples

Also note that when rounding up, there's a possibility that it can make the significand overflow.

For example:

000001111101→ | 0 | 3 | 10000 | OOPS! → | 0 | 4 | 1000 |

As shown above, if the analog encoding was 000001111101, the sign is S = 0, the exponent is E = 3 = 101 (since there are 5 leading zeros), and significand is S = 1111 = 15. Because the 5th bit is 1, we need to add 1 which would result in S = 1111 + 1 = 10000 = 16. As a result there is an

overflow, so we must truncate the least significant bit and increase the exponent by 1.

Lastly, it is needed to consider that the largest number possible to represent in a 8-bit Floating Point is 1920, and the smallest is -1920. If an input value goes out of bound, the program should return the largest/smallest value that the 8-bit Floating Point can represent.

Although the specs seems relatively straightforward, there are several edge cases that needs to be carefully considered: rounding up/down, overflowing bits in the significand,

II. Design Description

The module *fpcvt(D, S, E, F)* takes in a 12-bit input ‘D’ and outputs a 1-bit sign bit ‘S’, 3-bit exponent bit ‘E’, and a 4-bit significand bit ‘F’. The concatenation of these three outputs respectively will produce an 8-bit floating point representation of the input D or the closest representation to it.

```
    // check negativity of D most sig bit
        if (D[11] == 1'b1) begin
            in_D = ~in_D + 1;
            S = 1'b1;
        end else begin
            in_D = D;
            S = 0;
        end
```

Image 1: code with conditions described on the left

In order to regulate negative and positive two's complement numbers, the most significant bit (MSB) of the 12-bit input ‘D’, D[12], must be checked for high, indicating negativity. In such a case, inverse the input and add 1.

The 3-bit exponent ‘E’ can be values from 0-7, see graph 2. The amount of leading zeros are derived from the previous in_D, or a regularized version of D, through right shifting until the compiler sees the first one, indicating the end of the leading_zeros.

```
    // count leading zeros
        if (D != 12'b000000000000) begin
            while(in_D[11] == 1'b0)
                begin
                    leading_zeros = leading_zeros + 1;
                    in_D = in_D << 1;
                end
        end else begin
            leading_zeros = 0;
        end
        significand = in_D[11:8];
```

Image 2: code with conditions described on the left

A while loop continuously right shifts by one and counts the amount of shifts, indicating the amount of zeros, storing value in leading_zeros. The if-statement prevents infinite looping with specifically value 12’b000000000000.

A simple case statement is used in accordance to graph 2 following this piece of code to translate the amount of leading_zeros to the output 3-bit exponent.

The rounding rules, see graph 3, depends on the first bit following the 4. Since we right shifted to determine the amount of leading_zeros, the 7th bit of in_D[7] will always be the rounding bit following the 4-bit significand.

```
    // round for significand
        if (in_D[7] == 1'b1 && leading_zeros <= 8) begin

            if (significand == 4'b1111) begin
                if (exp == 7) begin
                    exp = 3'b111;
                    significand = 4'b1111;
                end else begin
                    exp = exp + 1;
                    significand = 4'b1000;
                end
            end else begin
                significand = significand + 1;
            end

        end

        if (leading_zeros >= 8) begin
            significand = stored_inD[3:0];
        end
```

Image 3: code with conditions described on the left

Inputs with leading_zeros greater than or equal to 8 will always have the significand be the 4 least significant bits with no need to account for rounding.

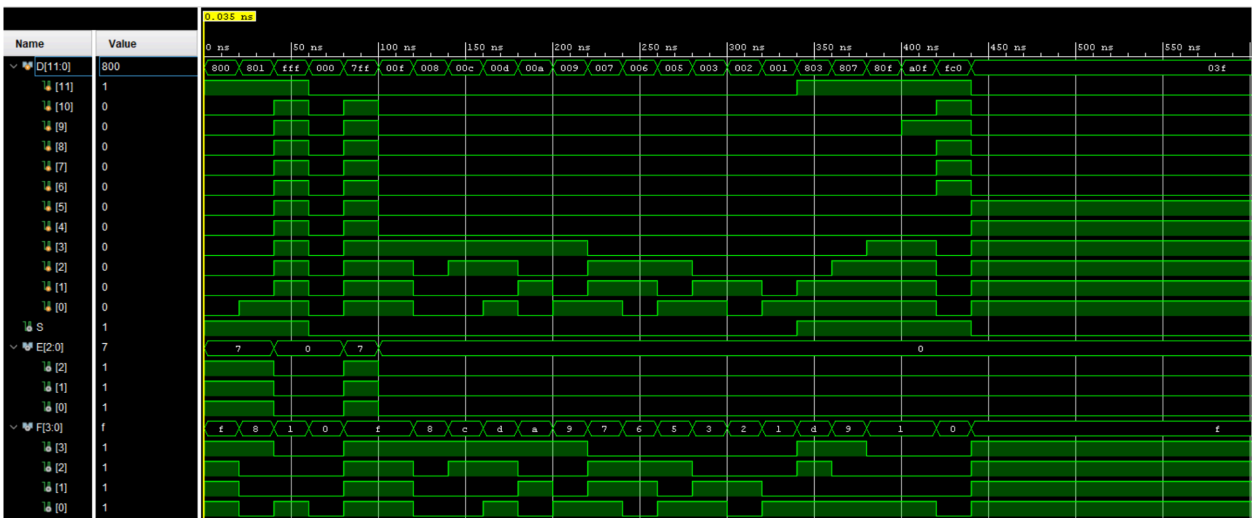
Inputs whose significands are 4’b1111 in addition to their round bit being 1 must be rounded appropriately depending if their exponent ‘E’ is 7.

Otherwise, rounding goes as normal and the significand adds one to itself.

Now that all cases have been accounted for, the compiler outputs ‘E’ and ‘F’ in addition to the ‘S’ bit determined earlier.

III. Simulation Documentation

In order to promote robust results and in the design, it is important to validate the accuracy of edge cases that could potentially result in undefined behavior or create bugs in the code such as infinite loops. In creating these test cases, we made sure to include cases that would follow the guidelines of the specs and inputs of an average user as well as cases that deviated from the average input and would need special case statements in order to allow the output of the correct result.



Graph 4: Simulation of the output from the test cases

The simulation diagram above shows input D, a 12-bit value, and outputs S, the sign bit, E, the 3-bit exponent bit, and F, the 4-bit significand. When the outputs are concatenated respectively, the final result will be the closest floating point representation. The test cases used in this simulation are defined below with a manually calculated result compared to the computerized output produced by the program. The accuracy of this program seems to be robust if it passes the below cases which represent the most complex use of the program.

Manuel

D	Floating Point	Value
100000000000	1, 111, 1111	-1920
100000000001	1, 111, 1111	-1920
111111111111	1, 000, 0001	-1
000000000000	0, 000, 0000	0
011111111111	0, 111, 1111	1920
000000001111	0, 000, 1111	15
000000001000	0, 000, 1000	8
000000001100	0, 000, 1100	12
000000001101	0, 000, 1101	13
000000001010	0, 000, 1010	10
000000001001	0, 000, 1001	9
000000000111	0, 000, 0111	7
000000000110	0, 000, 0110	6
000000000101	0, 000, 0101	5
000000000011	0, 000, 0011	3
000000000010	0, 000, 0010	2
000000000001	0, 000, 0001	1
100000000011	1, 111, 1111	-1920
100000000111	1, 111, 1111	-1920
100000001111	1, 111, 1111	-1920
101000001111	1, 111, 1100	-1536
111111000000	1, 011, 1000	-64
000000111111	0, 011, 1000	64

Output

```
D value is 100000000000
Output is 1, 111, 1111
D value is 100000000001
Output is 1, 111, 1000
D value is 111111111111
Output is 1, 000, 0001
D value is 000000000000
Output is 0, 000, 0000
D value is 011111111111
Output is 0, 111, 1111
D value is 000000001111
Output is 0, 000, 1111
D value is 000000001000
Output is 0, 000, 1000
D value is 000000001100
Output is 0, 000, 1100
D value is 000000001101
Output is 0, 000, 1101
D value is 000000001010
Output is 0, 000, 1010
D value is 000000001001
Output is 0, 000, 1001
D value is 000000000111
Output is 0, 000, 0111
D value is 000000000110
Output is 0, 000, 0110
D value is 000000000101
Output is 0, 000, 0101
D value is 000000000011
Output is 0, 000, 0011
D value is 000000000010
Output is 0, 000, 0010
D value is 000000000001
Output is 0, 000, 0001
D value is 100000000011
Output is 1, 000, 1101
D value is 100000000111
Output is 1, 000, 1001
D value is 100000001111
Output is 1, 000, 0001
D value is 101000001111
Output is 1, 000, 0001
D value is 111111000000
Output is 1, 000, 0000
D value is 000000111111
Output is 0, 000, 1111
```

Graph 5: Human verified results compared to computerized results of floating point representation

(Note for the TA); the output values of D=100000000001, 100000000011, 100000000111, 100000001111, 101000001111, 111111000000, and 000000111111 are not displayed correctly in the output column but perform fine when ran individually without the presences of a for-loop and/or multiple Ds running in the same testbench

IV. Conclusion

The program first determines the sign (S) of the analog signal based on the most significant bit. Then, it determines the exponent (E) based on the number of leading zeros. Finally, significand (S) is adjusted based on the specs. Our program considers edge cases, rounding, and exponent shifting.

One of the challenges we faced was figuring out the logic for determining the significand. We needed to first ensure if we can take the succeeding 4 bits after the leading zeros, or if we need to take the last 4 bits from the original analog encoding. To deal with this, we did an if statement to see if we should take the first 4 bits after the leading zeros (if leading zero <= 8), or if we should take the last 4 bits of the analog encoding (if leading zero < 8).

Another challenge was ensuring that a very big integer would be mapped to the biggest possible number that an 8-bit floating point can represent, or a very small number (big negative) would be mapped to the most possible negative number that an 8-bit floating point can represent. If did checks to ensure that bit overflow didn't occur with if statements.