Lab 2 Report
Sari, Drake, Adarsh

Introduction and Requirement:

In this lab, we focus on designing a combinational circuit to convert a 12 bit linear encoding of a signal into an 8 bit floating point representation. Analog signals on hardware are usually encoded into a digital representation to be used in software or to be transmitted. In this lab, the linear encoding of the signal is used as input to our circuit and is then converted to a floating point value which is used as output. Specifically, we implement the compression operation of a compounder by doing this; the expansion operation produces a linear encoding given a floating point representation, while compression produces a floating point representation given a linear encoding. A linear encoding is a method of arranging the bits such that the number of leading zeros indicates the exponent and the four bits after the last leading zero indicate the significand. Since we input 12 bits and output 8 bits, not all linear encoding inputs will lead to a unique output. Neither do all linear encodings have an exact floating point representation in our design. Therefore, values are also sometimes rounded, where the input is converted to the closest floating point representation. Furthermore, instead of counting the leading ones for negative numbers, we take the absolute value of negative numbers instead by taking their complement and adding one. The fifth bit after the last leading zero is used to determine whether we need to round up or down; if the bit is one, we round up, and if it's zero, we round down. We output 8 bits that are broken into 1 bit for the sign, 3 for the exponent, and 4 for the significand.

2's Complement:

> Two's Complement represents a number with a 'sign bit' in the most significant bit. If the sign bit is zero, then the number is simply the addition of two to the power of each location with a 1 (i.e. 0101 = 2^0 + 2^2 = 5). If the sign bit is 1 then the number is negative two to the power of the most significant bit plus 2 to the power of all of the other locations, making the number less negative (i.e. 1001 = -(2^3) + 2^0 = -7). Therefore, the most negative number represented by two's complement is 10000….000 as nothing is added to the negative. Some examples of two's complement numbers specifically in the 12 bit format of this lab:

Decimal:	Linear Encoding:
0	000000000000
-1	111111111111
-2048	100000000000
841	001101001001

Signed Magnitude From 2's Complement:

We convert from the input 2's complement into a signed magnitude representation which contains both the absolute value of the 2's complement representation and a signed bit that we will carry into our floating point representation. To switch from 2's complement to signed magnitude we do not need to do anything for positive numbers (most significant bit 0). However, for negative numbers, we convert to an absolute value representation by flipping all bits and adding 1 (i.e. 1001 = -7 -> 0110 + 1 = 0111 = 7). Additionally we will set the signed bit to 1 to indicate a negative value. We run into a special case in this transformation process where our most negative number 100000000000 when transformed overflows back into the exact same number and looks negative again.

Decimal:	2's Complement:	Signed Magnitude:
0	000000000000	000000000000 sign:0
-1	111111111111	000000000001 sign:1
-2048	100000000000	** 011111111111 sign:1
841	001101001001	001101001001 sign:0

**special case

Floating Point:

Our final output will be a modified floating point representation:

7	6	5	4	3	2	1	0
S	E			F			

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

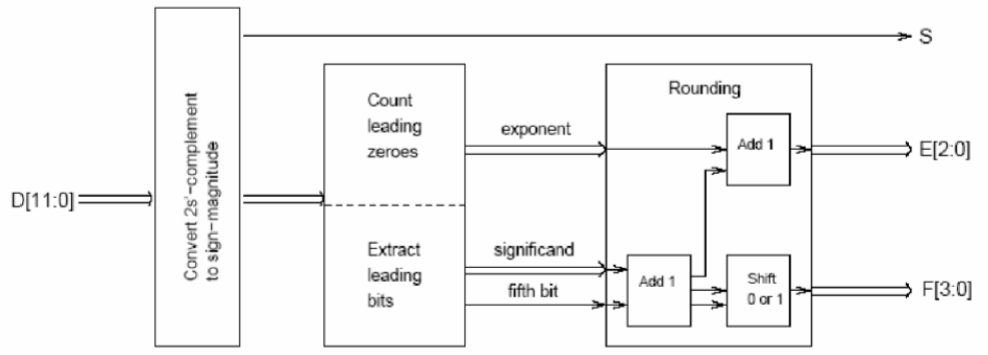
$$V = (-1)^S \times F \times 2^E$$

The **S**-Bit signifies the **Sign** of the number. The 4-Bit **Significand**, **F**, ranges from [0000] = 0 to [1111] = 15, 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.

Decimal:	Signed Magnitude:	Floating Point:
0	000000000000	00000000
-1	000000000001	10000001
-2048	011111111111	11111111 (-1920 is closest)
56	000000111000	00101110

Design Description:

We have three modules that together complete the compression conversion from 2's complement to the floating point representation.



Block Diagram of Final Circuit

2's Complement to Unsigned Magnitude (twoCompToSigned.v):

> This module takes in our initial input D[11:0] and outputs the absolute value of the input as well as a sign bit. The sign bit comes directly from D[11] and the absolute value is computed from the complement of the input plus one. We also account for the special case of -2048 with a hardcoded fix. This module is called in our FPCVT.v file and has no dependents itself. It is a leaf node in our code architecture for dependencies even though it is the first module in the series of computation to convert from raw input to our final output.

Unsigned Magnitude to Floating Point (linearToExponent.v):

> This module takes in the absolute value from twoCompToSigned and outputs the exponent based on the number of leading zeroes (inversely correlated from 1-8 leading zeroes to 7-0 exponent value) and the significand and rounding bit from the remaining bits in the absolute value (Note if there are 8 leading zeroes the significand gets the next 4 bits and the rounding it is automatically 0). This module is called in our FPCVT.v file and is also a leaf node in our dependency architecture. It is the second module needed in the series of computations for the conversion in the circuit block diagram.

```
module linearToExponent(
    input [11:0] absVal,
    output reg [2:0] exponent,
    //output reg [7:0] shifted_magnitude
            output reg [3:0] significand,
            output reg round_bit //fifth_bit
    );

reg [2:0] ctr = 0;
reg [3:0] shifted_magnitude;
always@* begin
            if (absVal[11] == 1'b0) begin
                        exponent = 3'b111;
                        shifted_magnitude = absVal[10:7];
                        round_bit = absVal[6]; //set to the fifth bit after the leading 0s
                        if (absVal[10] == 1'b0) begin
                                    exponent = 3'b110;
                                    shifted_magnitude = absVal[9:6];
                                    round_bit = absVal[5]; //set to the fifth bit after the leading 0s
                                    if (absVal[9] == 1'b0) begin
                                                exponent = 3'b101;
                                                shifted_magnitude = absVal[8:5];
                                                round_bit = absVal[4]; //set to the fifth bit after the leading 0s
                                                if (absVal[8] == 1'b0) begin
                                                            exponent = 3'b100;
                                                            shifted_magnitude = absVal[7:4];
                                                            round_bit = absVal[3]; //set to the fifth bit after the leading 0s
                                                            if (absVal[7] == 1'b0) begin
                                                                        exponent = 3'b011;
                                                                        shifted_magnitude = absVal[6:3];
                                                                        round_bit = absVal[2]; //set to the fifth bit after the leading 0s
                                                                        if (absVal[6] == 1'b0) begin
                                                                                    exponent = 3'b010;
                                                                                    shifted_magnitude = absVal[5:2];
                                                                                    round_bit = absVal[1]; //set to the fifth bit after the leading 0s
                                                                                    if (absVal[5] == 1'b0) begin
                                                                                                exponent = 3'b001;
                                                                                                shifted_magnitude = absVal[4:1];
                                                                                                round_bit = absVal[0]; //set to the fifth bit after the leading 0s

                                                                                                if (absVal[4] == 1'b0) begin
                                                                                                            exponent = 3'b000;
                                                                                                            shifted_magnitude = absVal[3:0];
                                                                                                            round_bit = 1'b0; //set to the fifth bit after the leading 0s
                                                                                                end
                                                                                    end
                                                                        end
                                                            end
                                                end
                                    end
                        end
            end
            assign significand = shifted_magnitude;

end


/*while(temp == 1'b0)
            exponent = exponent + 1'b1;
            temp = absVal[ctr];
            ctr = ctr - 1'b1;
            if (exponent == 1'd7)
                        temp = 1'b1;
*/

//shifted_magnitude = absVal[ctr:ctr-1'd4];
```

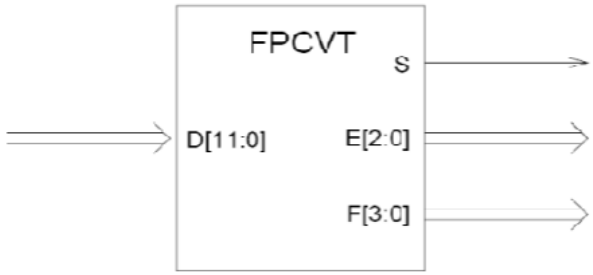
Rounding (rounding.v):

This module takes in the exponent [2:0], significand [3:0], and rounding bit (fifth bit) as input and deals with rounding to output the final exponent and significand. If the rounding bit is 0 we just output the original exponent and significand. If the rounding bit is 1 we will attempt to add 1 to the significand. If that causes overflow in the significand, we will increase the exponent 1 and shift the significand to the right. However, if the significand and exponent are both maxed out (all 1's) and will overflow if added to then we will just leave them as the closest approximation to the number we are attempting to represent even if the rounding bit is 1. This is the final module in the series of computations in the block diagram. It has no dependencies, but is called by the FPCVT.v main module and is a leaf node in our dependency tree.

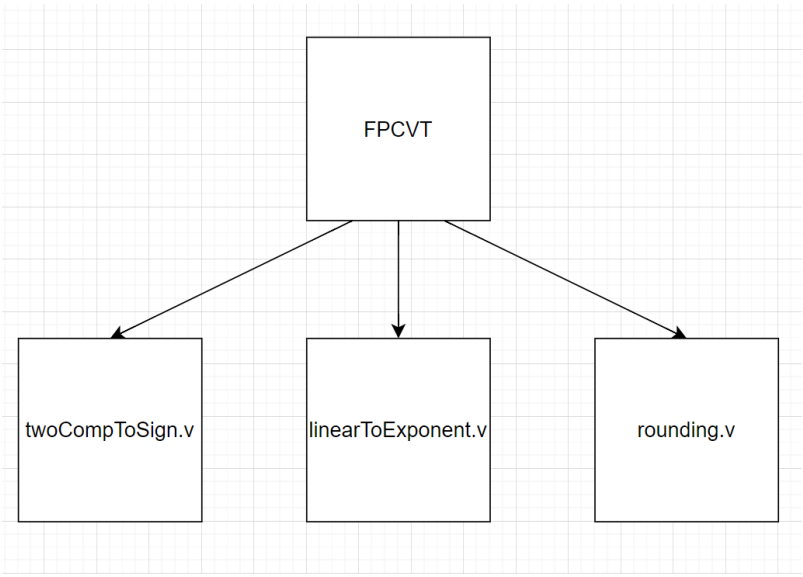
```
module rounding(
    input [2:0] exponent,
          input [3:0] significand,
          input round_bit,
          output wire [2:0] E,
          output wire [3:0] F
    );

//just add 1 to the significand if rounding && no overflow potential
//assign significand = (round_bit && !(significand == 4'b1111)) ? (significand[3:0] + 4'b0001) : significand[3:0];
//edge case: if rounding && significand is 1111
//shift significand right one bit and increase exponent by 1
//assign significand = (round_bit && (significand == 4'b1111)) ? (4'b0111) : (4'b1111);

//assign E = (round_bit && (significand == 4'b1111)) ? (exponent[2:0] + 3'b001) : exponent[2:0];
//assign F = significand;
reg [2:0] tempexponent;
reg [3:0] tempsignificand;
always @(*) begin
tempexponent = exponent;
tempsignificand = significand;
if(round_bit) begin
        if (significand == 4'b1111) begin
                if( exponent != 3'b111) begin
                        tempsignificand = (tempsignificand >> 1) + 1;
                        tempexponent = tempexponent + 3'b001;
                       end
                else begin
                        tempsignificand = 4'b1111;
                        tempexponent = 3'b111;
                end
        end
        else begin
                tempsignificand = tempsignificand + 4'b0001;
                end
        end
end
assign E = tempexponent;
assign F = tempsignificand;
endmodule
```

Calling module (FPCVT.v):

This module calls the other three models and is therefore dependent on each of them. It has no logic of its own other than passing the input and output into each of the three modules. It takes in the initial D[11:0] as input and outputs the final sign bit (from twoCompToSigned.v), and the exponent and significand (from rounding.v).



FPCVT Block Design



Code Dependency Graph

Simulation Documentation:

We had 4 different simulations, one for each separate module/file in addition to one simulation combining all 3. Each simulation had specific test cases to the design to account for edge cases along with testing the logic of different paths overall. For instance, for the twoCompToSigned module testbench, we tested the conversion of both initially positive numbers along with initially negative numbers to see how the module would treat them and what it would get for their sign and absolute value. Specifically, we made sure that our treatment of the edge case of the largest negative number in our case (-2048 or 100000000000 in binary) functioned correctly and could be converted into the proper signed magnitude representation to represent the most positive number we could represent given the bits we have. As for the linearToExponent module testbench, we passed in the values of 12 bit numbers to represent the converted absVal that comes from the twoCompToSigned module. We tested the different cases in which there were no leading zeros, the 12 bit value was all 1's, the 12 bit value was all 0's, and the 12 bit value had a few leading zeros. In order to confirm our module performed correctly, we displayed the values of the output of the exponent, significand, and round_bit to see if they were captured effectively. Due to these test cases, we were able to find a bug in which we realized we were handling the exponents incorrectly; initially, the exponent increased in our code with every leading zero, but through testing, we found the bug and were able to reverse the order of how we assigned the exponents. The rounding module passes the exponent, significand, and round_bit in as input and returns the output E and F. We tested our logic with varying cases of input to the testbench with the round_bit input being 1 while the significand is all 1's and the exponent is all 1's or if the round_bit is 1 and the significand is all 1's while the exponent is not, in addition to other variations of these cases. Through this, we were able to clarify our logic for the edge cases of the rounding module and that helped us finalize our program. Once we completed all of these

modules and tested the individual parts, we used the FPCVT.v file and testbench to test how they worked all together, and since we had tested all of the individual modules beforehand, we had no bugs in doing so. We tested the values of all of the edge cases for each module and viewed how they functioned in the overall program, ensuring they gave the correct output.

Conclusion:

In conclusion, our design was modularized into 3 separate files and a main file that calls each function. Each file carries out a separate aspect of the conversion; the first file converts the input from two's complement to sign-magnitude representation, the second file separates the bits and determines the exponent and significand, and the third file deals with rounding. There were a couple difficulties we encountered during this lab. Our first difficulty was in structuring our code. We weren't sure whether to write and call our functions all in one file or break the functions up into multiple files. We finally decided to separate each main component into its own file and corrected our syntax with the help of the TA. Doing so made it easier to read and test our code. Another problem we faced was in deciding what format to create our inputs and outputs in. We initially tried registers, and then wires that we assigned values to, and finally wires that we used blocking assignments for so they could be written inside an always block. Our main issue here was matching the output format and the actual code. The final issue we faced was more trivial; we initially wrote the code that determines the exponent to increase the exponent with increasing numbers of leading zeros instead of with decreasing numbers of leading zeros, and this caused confusing errors in the output until we discovered this problem at the end of the lab.