CS M152A Lab 2 Report

Hanna Co

November 1, 2021

1 Introduction and Requirement

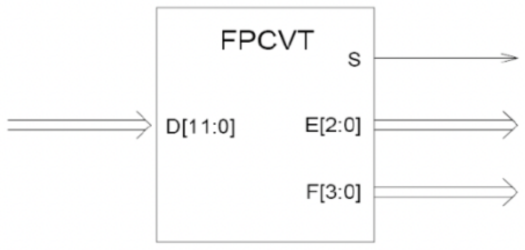
In computing, signals are often converted into integers. These integers can typically be stored in binary form using 8 bits. However, this can only store unsigned integers from 0-255, or signed integers ranging from -128-127. Thus, sometimes a different convention is used to store extremely small or large numbers. This is called floating point representation.

Floating point representation also uses 8 bits: 1 sign bit, 3 bit exponent, and a 4 bit significand, also known as a mantissa.

7	6	5	4	3	2	1	0
S	E			F			

The value represented by this is calculated as V = $(-1)^S$ x F x 2^E. The value of E, the exponent, ranges from 0 to 7, and the value of F, the significand, ranges from 0 to 15. Although there can be multiple floating point representations of the same number, the preferred representation is one where the most significant bit of the significand is 1, also known as the normalized representation.

We were tasked with implementing the following module to convert binary numbers to their floating point representation.



We first look at how many leading zeros there are, and use that to get the exponent. This is calculated according to the following chart:

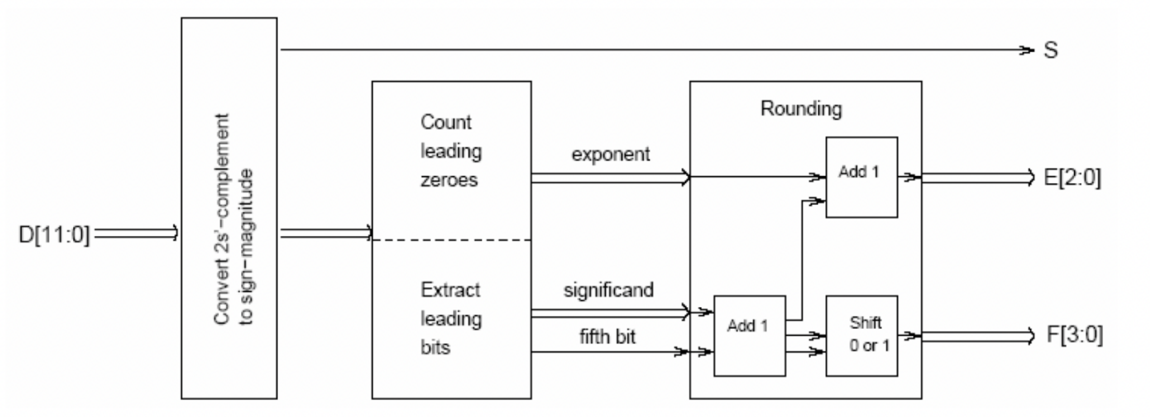
Leading Zeroes	Exponent
1	7
2	6
3	5
4	4
5	3
6	2
7	1
≥ 8	0

For example, 422 in binary is 000110100110, which has three leading zeros. Thus, its exponent is 5, represented as 101. Its significand is the four bits following the leading zeros, 1101. For negative numbers, we set the sign bit, S, equal to 1. We then negate the number and add 1.

We also need to account for rounding when converting to floating point. The fifth bit following the last leading zero tells us if we should round up or round down. If it is a 1, we add 1. Another thing to consider is if we have 1111, and we have to round up. In that case, we add one to the exponent, and change our significand to 1000. In the case that the number we are trying to encode is too large for our 8-bit floating point, we simply use the largest floating point representation.

2. Design Requirement

Our implementation takes in an 11-bit input D, and outputs a 1-bit sign bit S, 3-bit exponent, and 4-bit significand. A high-level diagram of this is shown below.



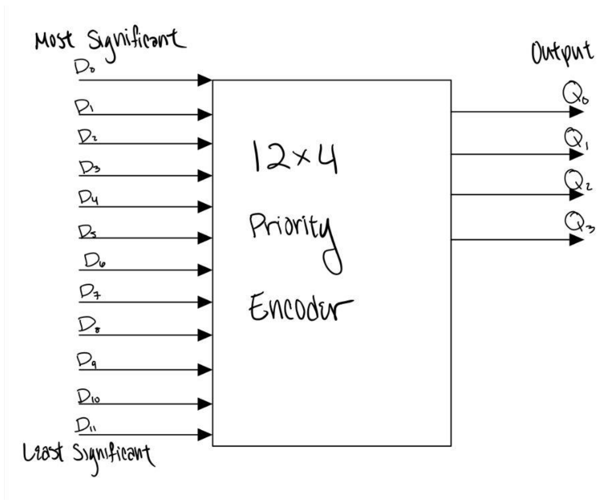
We create another 12-bit register, abs, that we use to store the value of D, or its negation.

```
if (S == 1) begin
```

```
            abs = ~abs;
            abs = abs + 1;
            if (abs[11] == 1) begin
                abs = 12'b011111111111;
            end
        end
```

In the above segment of code, we look at the value of S, which is set to the most significant bit of D. If S is 1, that means the number we are trying to represent is negative-- in that case, we use the negation of D and increment it by 1. Otherwise, we simply store D into abs. We also check for the special case that after incrementing abs, it overflows into a negative number again. In that case, we simply set abs to the largest positive integer.

Next, we use a priority encoder to determine how many zeros there are, and thus determine the exponent. The following diagrams show an overview of the priority encoder's inputs and outputs.



Input												Output			
D_0	**D_1**	**D_2**	**D_3**	**D_4**	**D_5**	**D_6**	**D_7**	**D_8**	**D_9**	**D_{10}**	**D_{11}**	**Q_0**	**Q_1**	**Q_2**	**Q_3**
0	1	X	X	X	X	X	X	X	X	X	X	0	0	0	1
0	0	1	X	X	X	X	X	X	X	X	X	0	0	1	0
0	0	0	1	X	X	X	X	X	X	X	X	0	0	1	1
0	0	0	0	1	X	X	X	X	X	X	X	0	1	0	0
0	0	0	0	0	1	X	X	X	X	X	X	0	1	0	1
0	0	0	0	0	0	1	X	X	X	X	X	0	1	1	0

0	0	0	0	0	0	0	1	X	X	X	X	0	1	1	1
Default												1	0	0	0

We then shift abs to the left by however many zeros, so we end up with the significand as the most significant bits in abs. From the number of zeros, we subtract that from 8 to get our exponent E.

Our next and final bit of code determines our significand, F.

```
if (abs[7] == 1 && abs[11:8] == 4'b1111) begin
    if (E != 7) begin
        F = 4'b1000;
        E = E + 1;
    end
    else begin
        F = abs[11:8];
    end
end
else begin
    F = abs[11:8] + abs[7];
end
```

We first check to see if rounding will overflow. If it does, then we make sure our exponent is not the maximum value 7. This way, we can increment the exponent without worrying about overflow. If it does overflow, then we simply set F to 1111, the four most significant bits of abs. In the case that rounding does not overflow, we simply set `F = abs[11:8] + abs[7]`. This works because if `abs[7]` is 0, `abs[11:8]` does not change, and if `abs[7]` is 1, we need to increment `abs[11:8]` by 1 anyway.

3 Simulation Documentation

To test our implementation and ensure accuracy, we created a testbench with test cases. We tested our implementation with the test cases outlined in the project specification, as well as edge cases, such as when D was the smallest possible value, the largest possible value, zero, or one. We then tested for positive and negative number overflow. Finally, we had one large for loop that iterates through a range of numbers, and checked to see that the rounding war accurate, and that they produced the expected values.

4 Conclusion

In conclusion, we implemented our floating point convertor using a priority encoder and simple digital logic. We accounted for rounding and edge cases by checking the values of abs at certain points, and adjusting accordingly. Writing our testbench was fairly straightforward, as we simply had to check and see that the output value was correct.

There were a couple challenges we encountered when writing our implementation. For one, we weren't sure on how to get the number of leading zeros. Our first approach involved iterating through abs with a while loop and counting the zeros, but it felt inefficient. We eventually changed our approach and used a priority encoder instead. Another challenge we faced was handling the edge cases. We struggled with understanding where our code was incorrect. However, once we figured this out, it only took a few lines of code before we were able to pass the edge cases.

Overall, I don't have many suggestions on how to improve this lab. I felt that it was fairly straightforward, yet was challenging enough that it took us all designated lab sections to complete.