CS M152A - Lab 2 Report

[Section 2, Team 2]
Edgar Ayala (UID: 604619231)
Nathaniel Thomas (UID: 506164479)

1. Introduction and Design Requirements

What is a floating-point converter, and why is it useful?

A floating-point converter is a circuit that converts data from one encoding (e.g. a linear encoding) into a floating-point (FP) encoding. The purpose of this tool is to preserve the values of numbers across different formats so that data is represented and interpreted correctly across different computer systems. In a simple floating-point converter, such as the one implemented in this lab, not all linear encodings may have exact FP representations, while other linear encodings may have multiple FP representations.

What is special about the design of this lab project in particular (what are the inputs and outputs and special behavior)?

The design of this lab is special in that we use a 12-bit encoding when most computers use 32 or 64-bit encodings. This results in much lower precision and an inability to represent a lot of the two's-complement numbers. The input to our floating point encoder is a 12-bit two's-complement number (D) and the outputs are the Sign-bit (S), the 3-bit Exponent (E), and the 4-bit Significand (F). This results in an 8-bit byte with the structure [S EEE FFFF], which can be used to generate a value using the formula:

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

Some special behavior for this design resides in the handling of especially large numbers or especially negative numbers. Since we have a limited 12 bits to represent all the two's-complement numbers, there are some numbers that all get mapped to the same floating point representation even though they may be significantly different since they are either too large or too negative. When the input is -2048, for example, the traditional method of inverting the bits and adding 1 cannot be used as this leads to a wrong sign-magnitude representation. Instead, this number must be manually converted.

2. Design Description

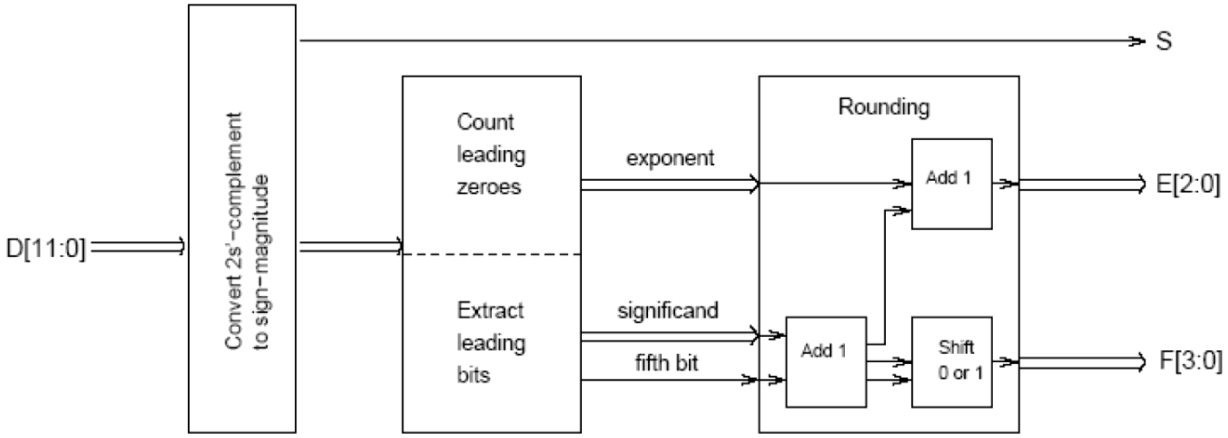


Figure 1: Floating-Point Converter Circuit Block Diagram (Note: Our implementation splits the second block into two modules).

First Block: sign_magnitude_converter

The purpose of this module is to convert a number from two's-complement form to sign-magnitude where the sign is separately extracted into the S output and the sign magnitude form in absolute value form is extracted into the D output. This module achieves this by taking the first bit of the two's-complement number, D11, as the Sign bit because that is what determines the sign of the number. Then, depending on if the sign was positive or negative, it will either keep the rest of the number the same or invert the bits and add 1, respectively. The last case it will check for is the maximum negative number, which is 1 followed by all 0s or [1000 000 0000], which it will specially convert to 0 followed by all 1s or [0000 0000 0001] since the inversion-increment doesn't work in this case.

```
always @(*) begin
    temp_sign = signed_num;
    if (complement_num == 12'b100000000000) begin
        temp_sign = 12'b011111111111;
    end else if (signed_bit == 1) begin
        temp_sign = ~complement_num + 1'b1;
    end else begin
        temp_sign = complement_num;
    end
end
```

Figure 2: Logic in the sign_magnitude_converter performs inversion-increment if input is not the maximum negative number.

Second block: count_leading_zeros

The purpose of this module is to take a number in absolute value sign-magnitude form and count the number of leading zeros in this number and derive an exponent for the floating point representation of this number from that. It is done by a series of if-else statements which determine the position of the first 1 in the number and assign the exponent to 7 all the way down to 0 depending on how far along the first 1 appears with a maximum of 8 leading 0s counted.

```
always @(*) begin
    if (signed_num[10]) begin
        temp_exp = 3'b111;
    end
    else if (signed_num[9]) begin
        temp_exp = 3'b110;
    end
    else if (signed_num[8]) begin
        temp_exp = 3'b101;
    end
```

Figure 3: Example of the if-else statements found in count_leading_zeros module.

Third block: extract_leading_bits

The purpose of this module is to determine the significand of the floating point representation of the given number in absolute value sign magnitude form as well as the exponent previously derived. This module will also provide the fifth bit following the extracted 4 bits from the significand to determine rounding. This is done by using the exponent previously calculated to to shift the D input and take the 4 bits that follow the leading 0s. Then, using that same exponent we subtract 1 and get a new shift value to extract the 5th bit from the D input. In the case where exponent is 0 the subtraction of 1 causes the shift value to overflow to 7 which will ultimately lead to a 0 being extracted as the 5th bit because of the number of leading 0s on this D input which is the desired output since if the exponent is 0 there should be no 5th bit which is the same as 0.

```
always @(*) begin
    last_bit_extractor = signed_num;
    last_bit_extractor = last_bit_extractor >> (exp-1);
end

assign fifth = last_bit_extractor[0];
assign sig = (signed_num >> exp);
```

Figure 4: The fifth bit and the significand are extracted using shift logic in the extract_leading_bits module.

Fourth block: rounding

The purpose of this block is to take the exponent, significand, and fifth bit to round the exponent and significand for a final floating point representation of the input. This module accomplishes this by checking the 5th bit and only rounding the significand if it is 1. The rounding is done by adding 1 to the significand which in the case that it overflows the significand is set to 1000 and the exponent is incremented by 1. However, if the exponent also overflows from this we simply keep both the exponent and significand as all 1s since this was the max representable value in FP form. The result from this module is the final E and F.

```
always @(*) begin
    if (fifth == 0) begin
        temp_rs = sig;
        temp_re = exp;
    end
    else begin
        if (sig == 4'b1111) begin
            if (exp == 3'b111) begin
                temp_re = exp;
                temp_rs = sig;
            end
            else begin
                temp_re = exp + 1;
                temp_rs = 4'b1000;
            end
        end
        else begin
            temp_rs = sig + 1;
            temp_re = exp;
        end
    end
end
```

Figure 5: The edge case when the significand and exponent are maxed out is considered when performing the rounding procedure in the rounding module.

Interfaces (inputs and outputs, wire sizes)

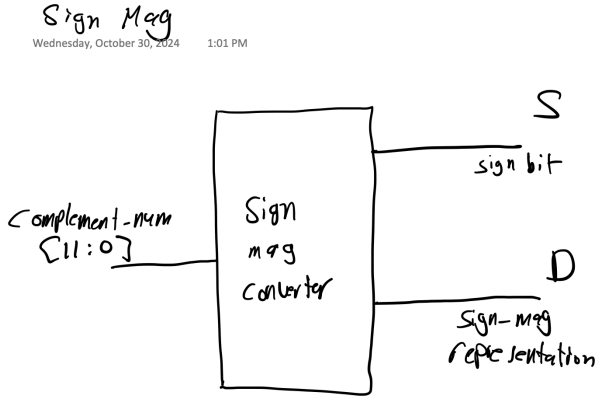
First block (sign_magnitude_converter): The input is the 2s complement number with a wire size of [11:0] (12 bits). The outputs are the sign bit (S) with a wire size of 1 bit and the absolute value sign magnitude value with a wire size of [11:0] (12 bits).

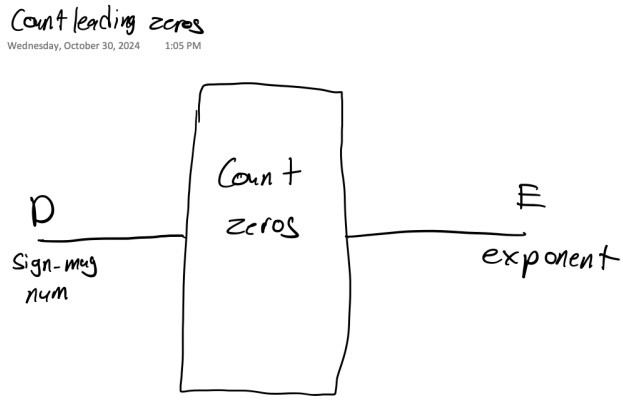
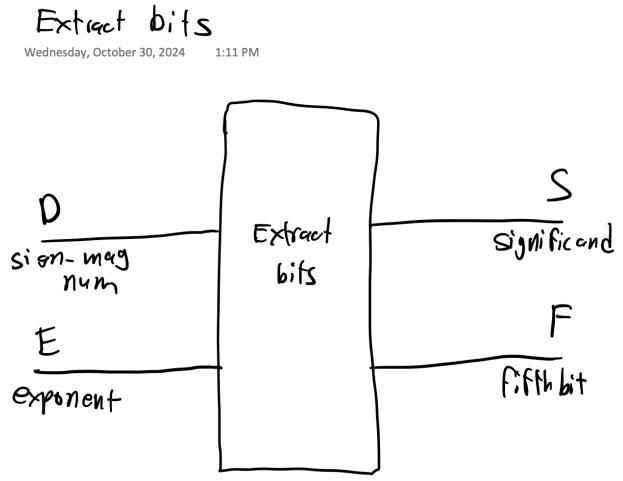
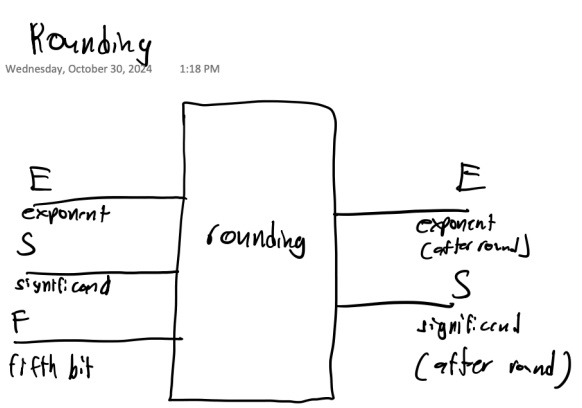
Second block (count_leading_zeros): The input is the absolute value sign magnitude number (D) with a wire size of [11:0] (12 bits) and the output is the exponent value unrounded with a wire size of [2:0] (3 bits).

Third block (extract_leading_bits): The inputs are D (sign magnitude form of the input in absolute value) and E (the exponent value calculated so far) with a wire size of [11:0] (12 bits) and [2:0] (3 bits) respectively. The outputs are S(the significand for the time being) and F(the fifth bit aka the bit right after the significand) with a wire size of [3:0] (4 bits) and 1 bit, respectively.

Fourth block (rounding): The inputs are S, E, and F which we have calculated so far which are all unrounded with wire sizes of [3:0] (4 bits), [2:0] (3 bits), and 1 bit respectively. The outputs are S and E which are the final rounded values of the significand and exponent for the floating point representation of the input number with wire sizes of [3:0] (4 bits) and [2:0] (3 bits), respectively.

Schematics

First block: sign_magnitude_converter	

Second block: count_leading_zeros	
Third block: extract_leading_bits	
Fourth block: rounding	

Considerations

We took into consideration the edge cases such as all 0s, all 1s, maximum negative number, and maximum positive number when constructing these blocks to ensure that our modules worked for any input.

3. Simulation Documentation

Testing and Simulation

The design of the FPCVT was divided into 3 main blocks (as shown in Figure 1 above) so we tested each component individually.

To test the first block (converting 12-bit two's-complement input into sign-magnitude representation), we knew that if an input contained a '1' in the MSB position, then it was a negative number and a conversion-increment would need to be performed. This conversion generally yielded the correct result, but the one specific edge case where it didn't was with the most negative number: -2048 or [1000 0000 0000]. For this reason, our code would first check if the input matched this edge case; if it did, then we would not increment by 1 like we normally would to avoid generating the wrong sign-magnitude representation.

To test the next block (count leading zeroes+extract leading bits), we implemented two separate modules to handle each function. After converting the two's-complement input into sign-magnitude representation, the bit at the MSB position (position 11) was guaranteed to be a '0' so we checked if the following bit (at position 10) was a '1'. If it was, then we knew that the number had exactly one leading '0' and assigned it its corresponding exponent value (which was 7 in this case). If this bit was not a '1', then a series of if-else statements would check the subsequent bits one-by-one for a '1' to determine the value of the exponent corresponding to the number of leading zeroes. The exponent was set to 0 when there were at least eight leading zeroes (this was also the base case) and 1-7 otherwise. To test this behavior, we used input values with varying amounts of leading zeroes. The leading bit extractor module served two purposes: to extract the 4-bit significand and the fifth bit for rounding. Since this module heavily relied on the value of the exponent found from the previous module, testing both modules was done in conjunction. The logic was straightforward: the 12-bit input was shifted to the right by the amount specified in the exponent, and the last four bits made up the significand. For example, when the input was [0000 0010 1110], the count_leading_zeroes module found that the number of leading zeroes was 6 and set the exponent to 2. The extract_leading_bits module then shifted the input to the right by 2: [0000 0010 1110] → [000000 0010 11]. This resulted in [1011] as

the significand, which was the desired result. To extract the fifth bit, the same shifting procedure was used by copying the input into a 12-bit register, but we would shift right by the amount specified in the exponent minus 1 to obtain the LSB (fifth bit). Since this bit determined the rounding procedure that takes place at the last block, we used input values with the same significand, but different fifth bits, such as [0000 0010 1100] and [0000 0010 1110].

Lastly, to test the last block (rounding), we knew that if the fifth bit was a '0', then we would round down (i.e. truncate the result). If the fifth bit was a '1', then we would round up, by adding one to the significand and leaving the exponent unchanged. If the significand was maxed out at [1111] when rounding up, then there were two edge cases to consider: a maxed out exponent [111] and a non-maxed out exponent. If both the significand and exponent were maxed out, then we had reached the largest floating point representation (e.g. 2047 = [0111111111]) so no rounding would take place. If the significand was maxed out and the exponent was not maxed out, then the rounding up would overflow the significand (as incrementing 1111 would lead to 10000) so the significand would be shifted right to obtain [1000] and the exponent would be incremented.

```
initial begin
    D = 12'b000000000000;
    #100
    D = 12'b111111111111;
    #100
    D = 12'b000000101100;
    #100
    D = 12'b000000101101;
    #100
    D = 12'b000000101110;
    #100
    D = 12'b000000101111;
    #100
    D = 12'b000001111101;
    #100
    D = 12'b111111011000;
    #100
    D = 12'b000000111000;
    #100
    D = 12'b100000000000;
end
```

Figure 6: Test bench showing variation in inputs for D.

Test Cases + Console Output / Waveforms

The first test case was the most negative number: -2048 or [1000 0000 0000]. We chose this input as it is an important edge case that tests the correctness of our sign magnitude converter module (i.e. the first block).

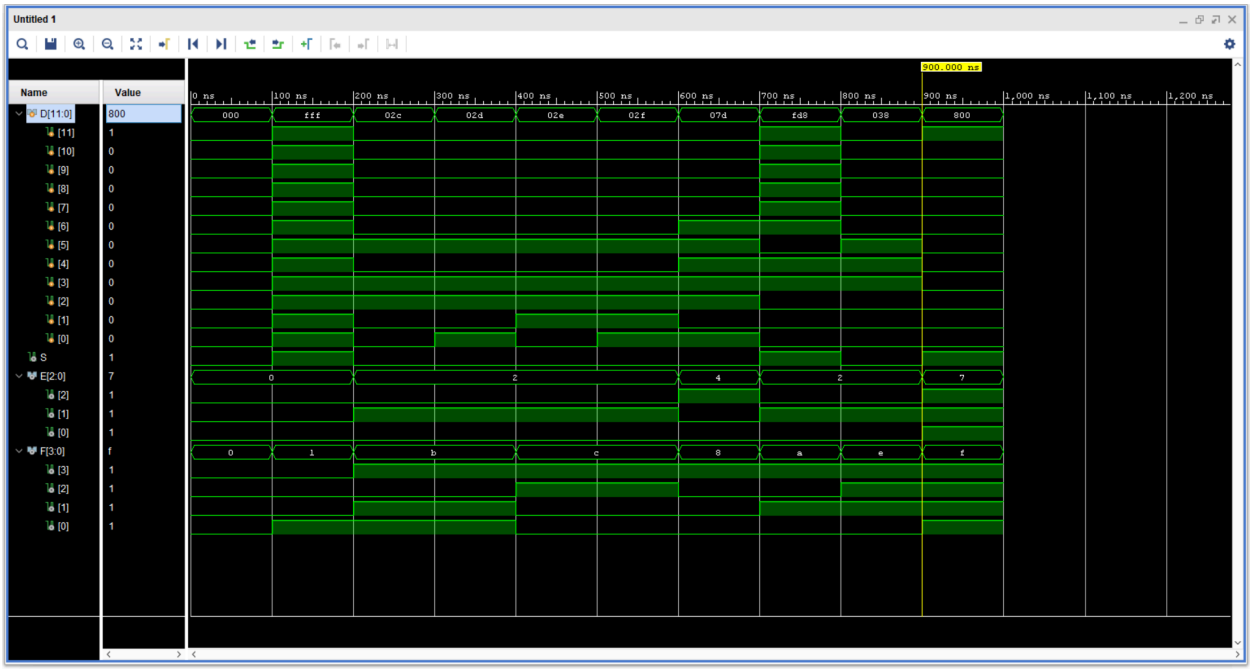


Figure 7: When D=-2048 or [1000 0000 0000], the FP encoding is [1 111 1111].

We also tested all 0s and all 1s because they are on the extremes of the binary representations of our input.

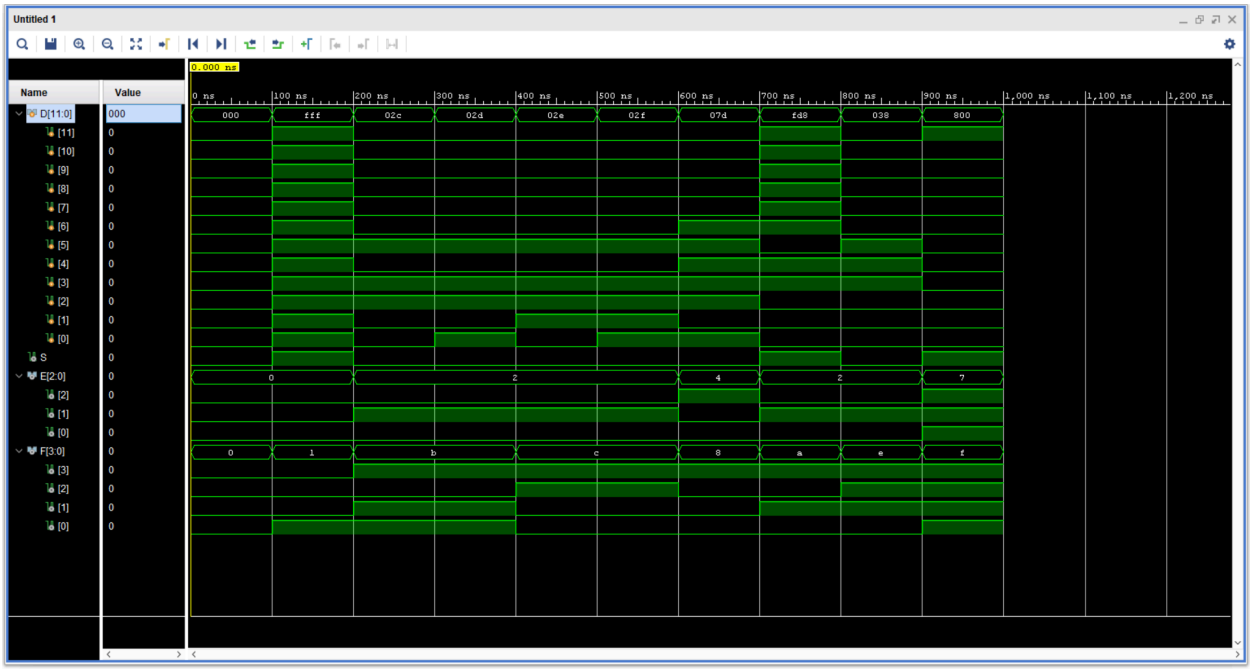


Figure 8: When D=0 or [0000 0000 0000], the FP encoding is [0 000 0000].

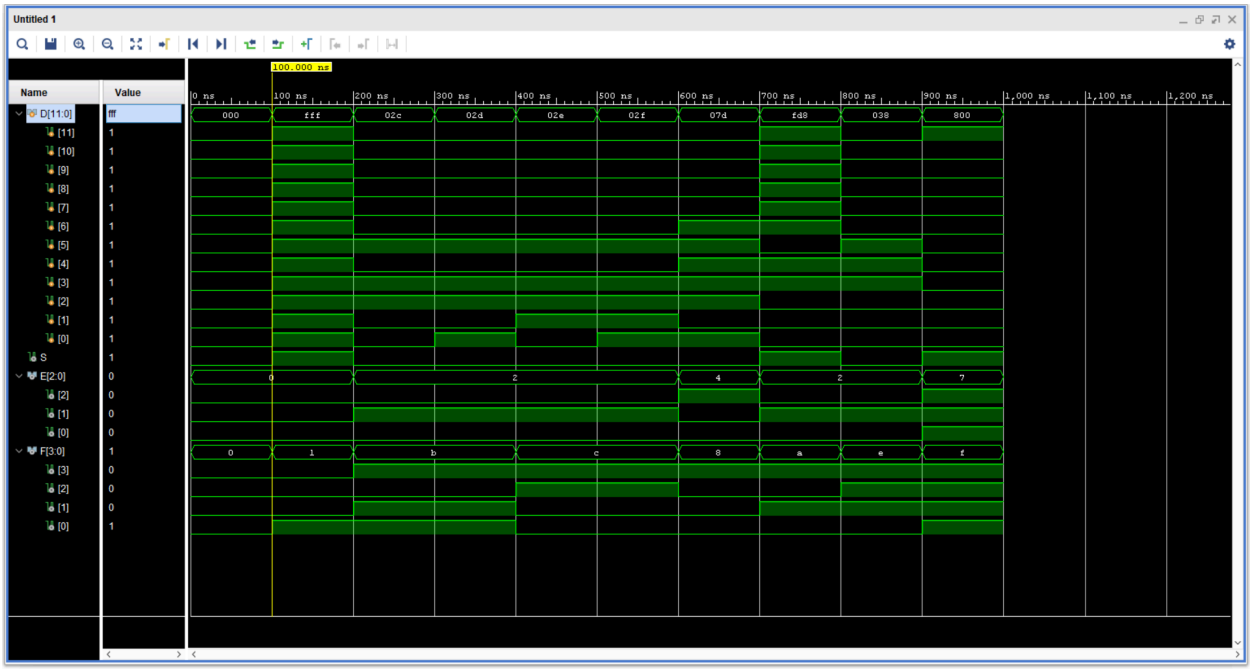
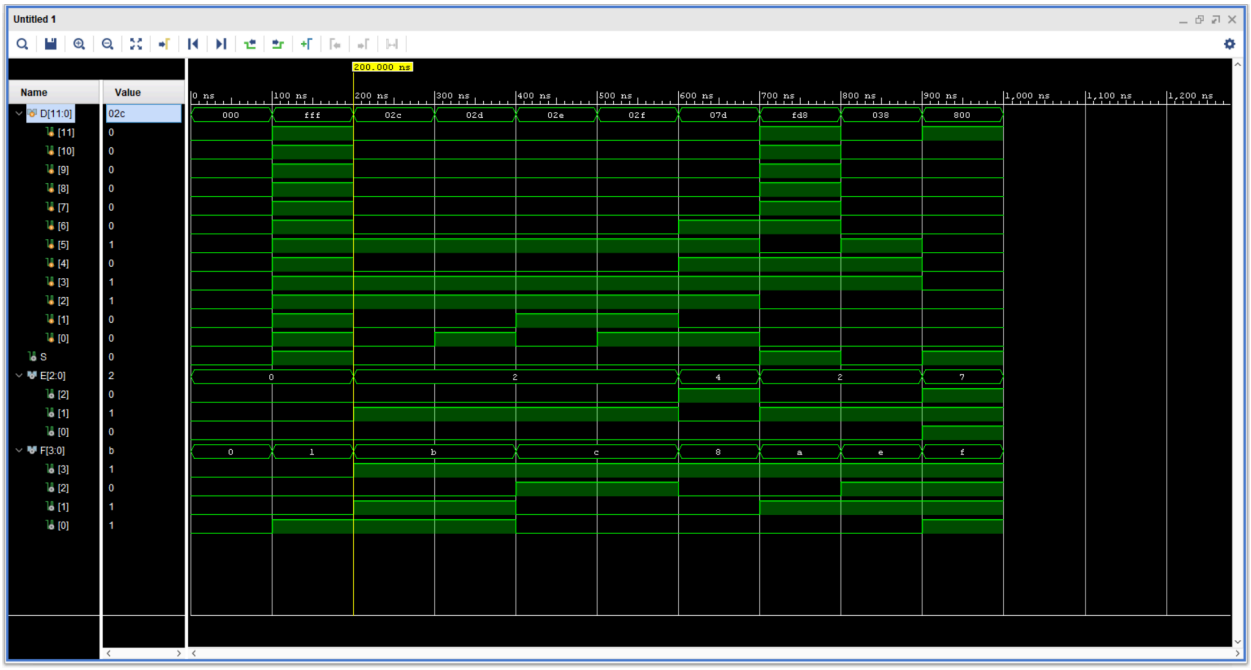
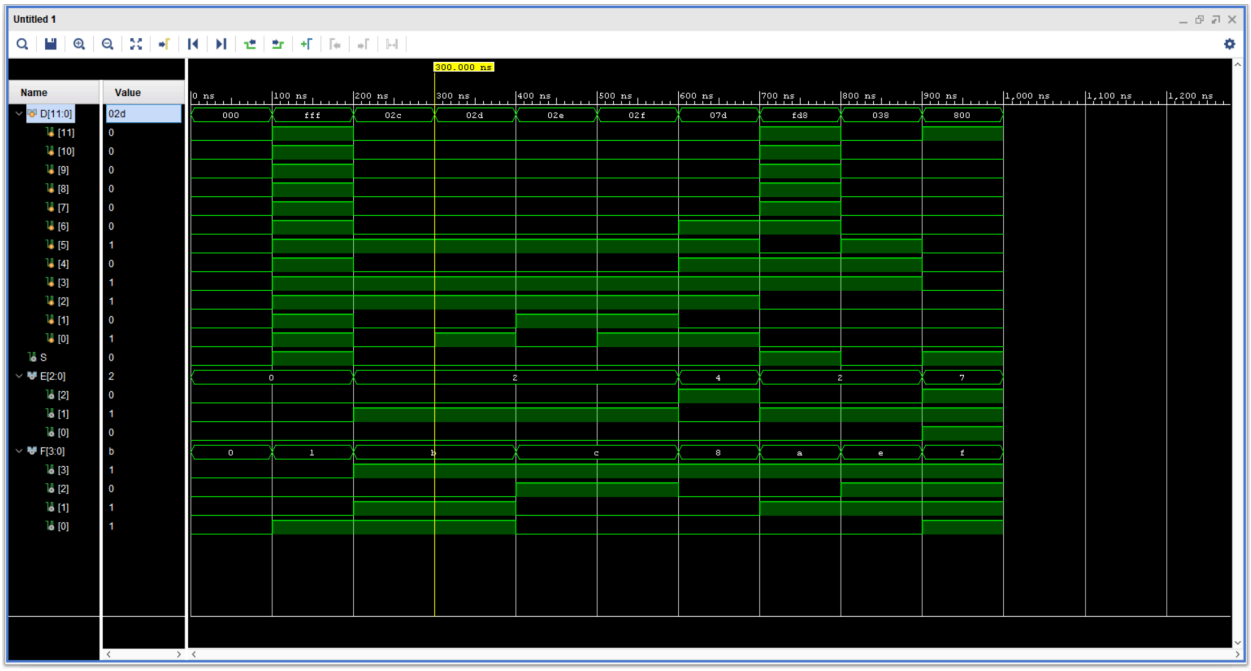


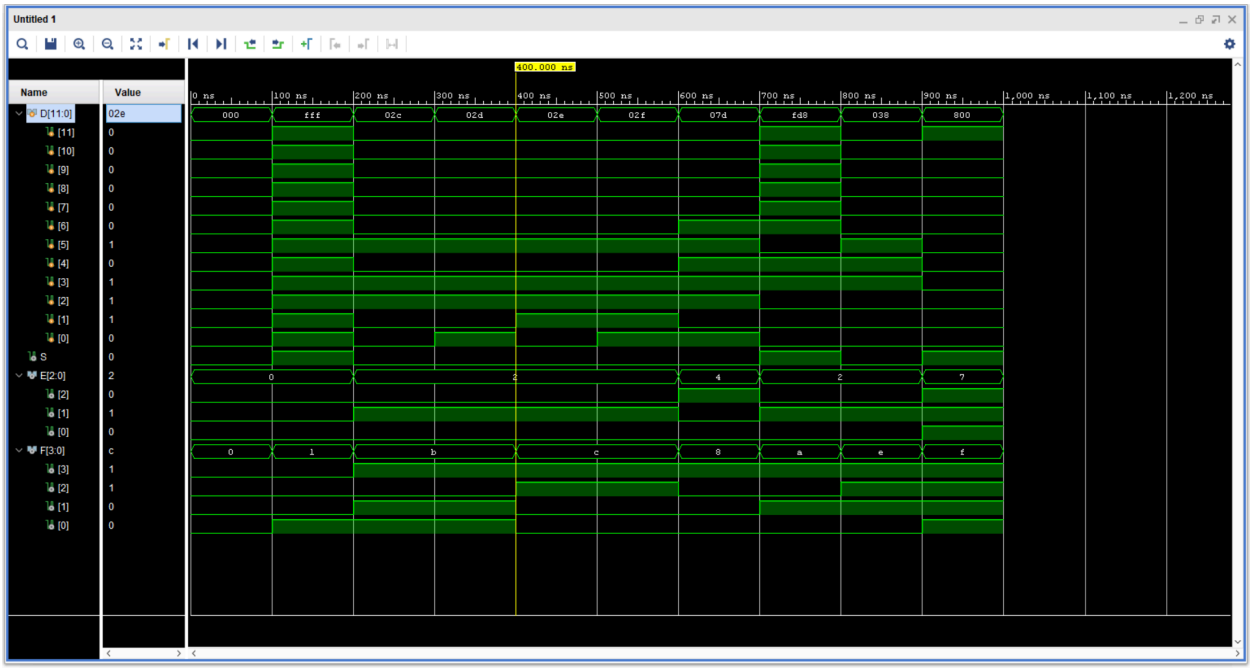
Figure 9: When D=-1 or [1111 1111 1111], the FP encoding is [1 000 0001].

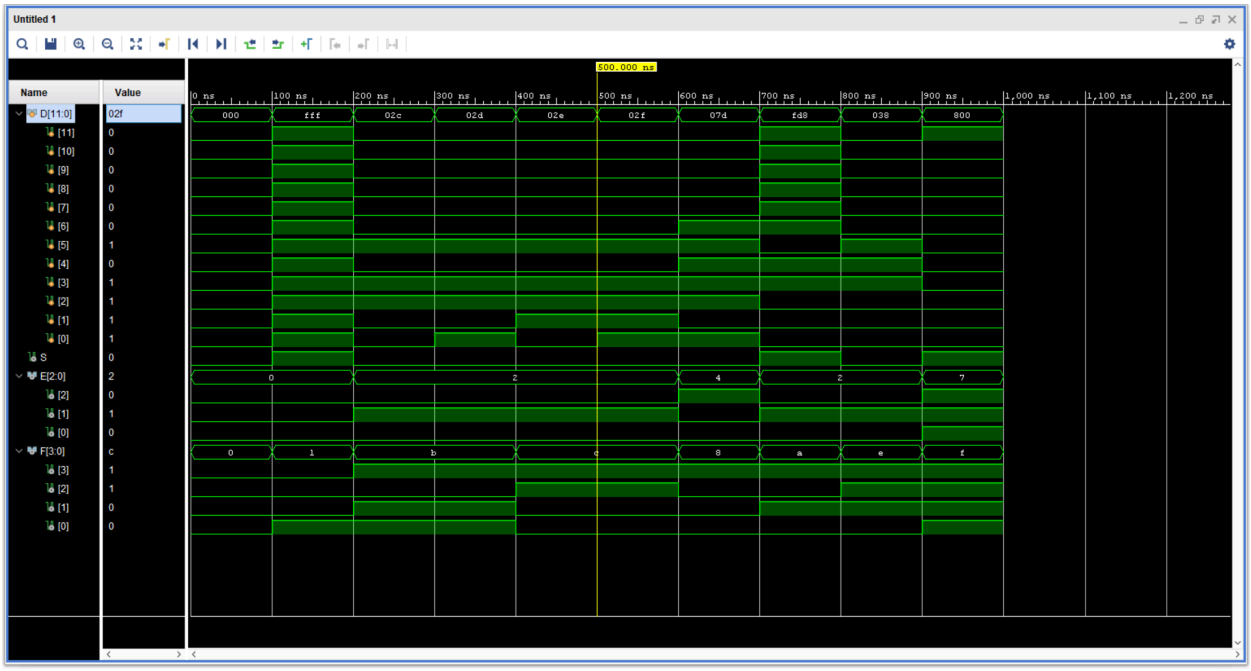
Lastly, we tested the random examples of input and output listed in the spec as well as the rounding examples listed in the spec. We chose these tests because we knew exactly what the input and output should look like and there wouldn't be any misleading test results in case we accidentally computed the floating point representation of a certain two's-complement number incorrectly. Also these numbers represent a good range of cases that our code should be able to handle as it tests all 4 rounding cases as well as positive and negative numbers as input.





Figures 10 & 11: When D=44 or [0000 0010 1100] and D=45 or [0000 0010 1101], both numbers are rounded DOWN to the nearest FP encoding of [0 010 1011].





Figures 12 & 13: When D=46 or [000000101110] and D=47 or [000000101111], both numbers are rounded UP to the nearest FP encoding of [0 010 1100].

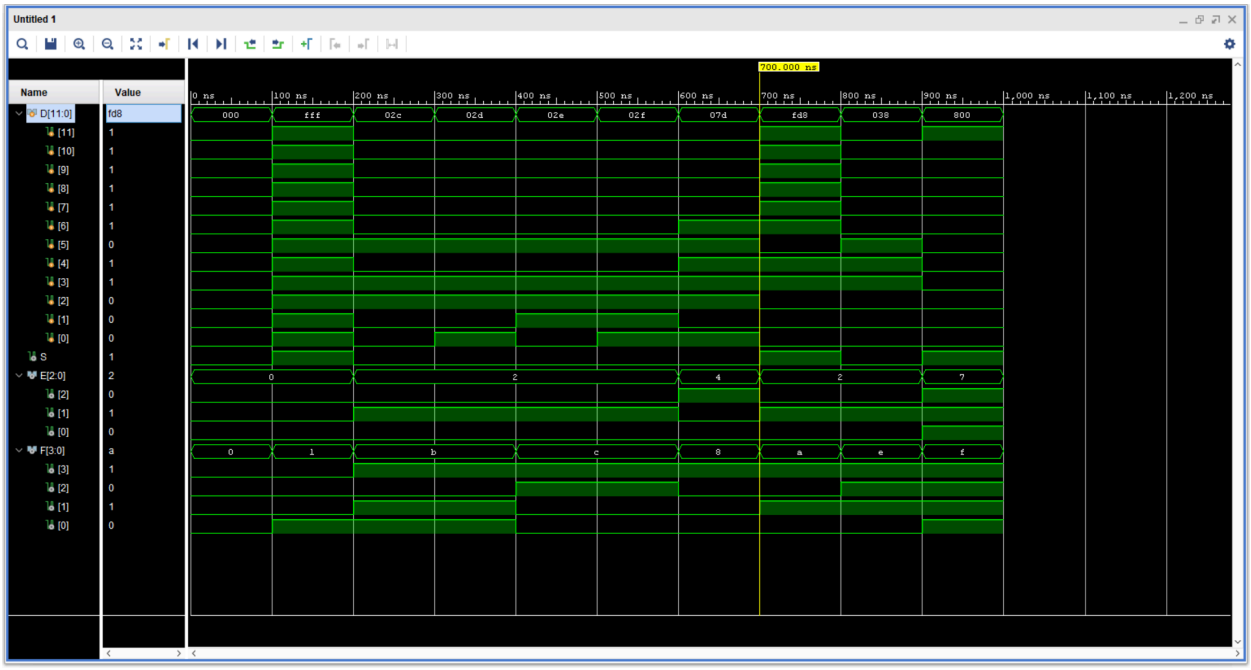


Figure 14: When D=-40 or [1111 1101 1000], the FP encoding is [1 010 1010].

Bugs

We didn't encounter any bugs specifically with the simulation, but we did find bugs with our module logic when running our simulations. We describe these bugs and our resolutions to them in the conclusion section.

4. Conclusion

Summary

Our design consists of four modules (instead of the suggested three) because we found it easier to conceptualize the calculation for the significand and the exponent as separate modules. The first module converts the two's-complement number to sign-magnitude form while also taking the absolute value of the number since the sign bit is separately extracted as well by this module. Next, we count the number of leading zeros in the sign-magnitude form of the number to get the exponent for the floating point form. Next, we take the four bits after the leading zeros (maxing out at 8 leading zeros) to form the significand and then we take the fifth bit after the leading zeros to get the decision on whether or not to round the number. Lastly, we perform rounding on the significand and if it overflows, we increment the exponent and shift the significand. If both the significand and exponent overflow (i.e. we hit the max or min representable number), we simply keep both the same.

Difficulties

We had some difficulties when doing the leading zero calculation and the rounding. We did not count the number of leading zeros correctly because of the order of our if-else statements and we also did not handle overflow properly when doing the rounding. We dealt with these issues by first identifying them using our test cases in the testbench and looking at the waveform output. Next, we changed the structure of the if-else statements and case handling for both of those modules to fix all issues.

Takeaways

We got a fundamental understanding of Verilog code and how Verilog modules can come together to form a larger program. Since we designed each of these modules from scratch, we got a better idea of how to design Verilog modules and how to provide input and output to them as well as the restrictions surrounding wires and registers. Lastly, since we wrote the testbench ourselves, we got a better understanding of how to design test cases and debug code in Verilog.

Suggestions

We would improve this lab by providing more assistance with how to structure the modules in Vivado because we struggled a lot at the start to instantiate the modules we created within the FPCVT.v file. We would also recommend the structure of the lab to suggest building four (instead of three) main modules since it really made it much easier for us to think about the converter as a whole.