Team Kublinh
Kubilay Agi 304784519
Baolinh Nguyen 104732121
Lab 2
February 12, 2018

Design Description

The overall design of this project was based off of the diagram given in the specification. We broke up the project into three major submodules.

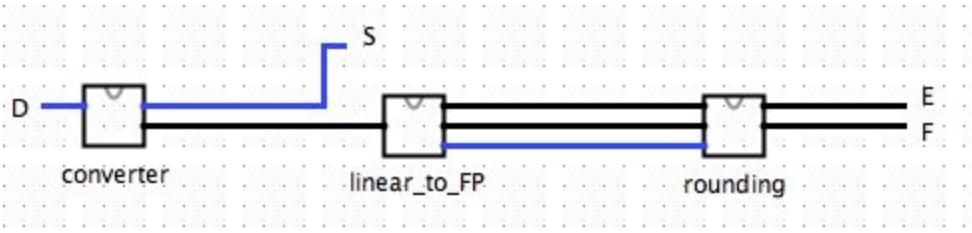

Figure 1: The above screenshot shows a high level diagram of our compounder design that is broken into three major submodules: converter, linear_to_FP, and rounding.

The first module converts two's complement numbers to signed magnitude. This way, we are able to obtain the bits for the magnitude of the number that we want to convert into floating point form. After the conversions, the positive numbers stay the same, but the negative numbers are set to their absolute value.

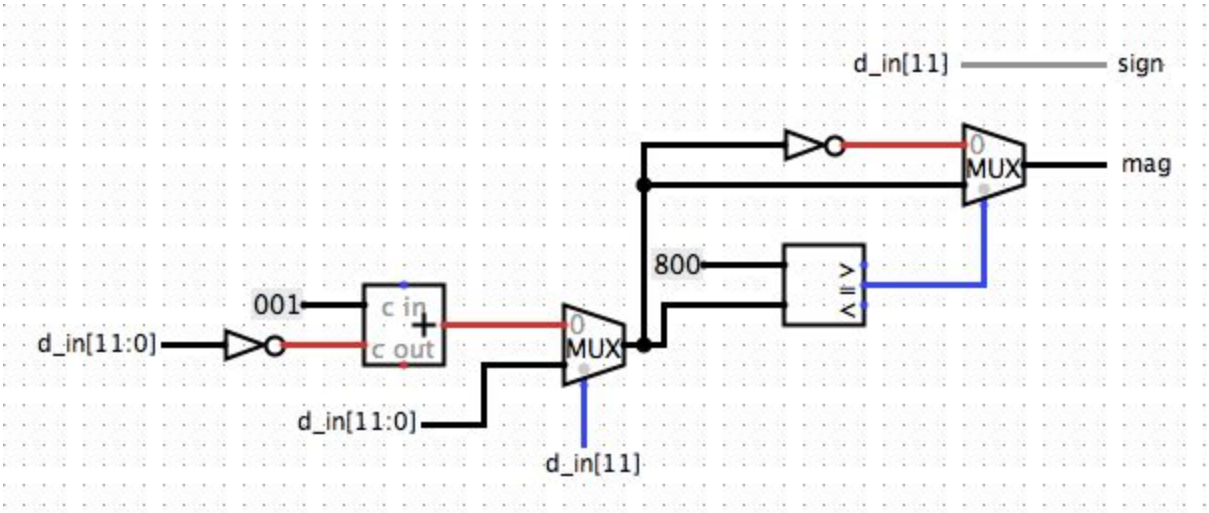

Figure 2: The above screenshot shows the schematic for our converter submodule, which converts our two's complement number into signed magnitude.

To perform this conversion, we simply check the most significant bit of the input wire ‘d’. If d[11] has value one, this means that the inputted number is negative, and we will flip this number by inverting all of the bits and adding one. However, with the case of INT_MIN (which is -2048 in this case), this method of conversion does not work correctly. If we flip all bits of -2048 and add one, we will end up with -2048 again. In order to solve this, we added a special case to our code that would check specifically for -2048. A simple if statement checking for whether or not the bit representation of the input matched INT_MIN suffices for this check.

```
33  //CHECK FOR INT MIN
34  assign temp_temp_mag = (temp_mag[11:0] == 12'b100000000000) ? ~temp_mag : temp_mag;
```

Figure 3: The check that is performed in the case of INT_MIN in the converter submodule.

After converting the number to signed magnitude, we needed to break up the number into the exponent and significand. We do this using the linear_to_FP submodule. One functionality of this submodule was to find the number of leading zeros in the number. This allows us to determine the exponent required for the number to be represented as floating point. We created a priority encoder for this step by chaining together ternary operators. We also used priority encoders to find the next four bits after the last leading zero and for the fifth bit after the last leading zero. These steps were straightforward as all we needed to do was check each bit in order from most-significant to least-significant to see if it was one. If it was indeed equal to one, then we assigned the significand of our new floating point number to that bit as well as the following three bits. For example, if d_in[9] was the most-significant digit to be one in our input signal, then we assigned the significand to d_in[9:6]. Following the same example, the fifth bit would be assigned to the one immediately follows the least-significant bit of the significand, which would be d_in[5] in this case.

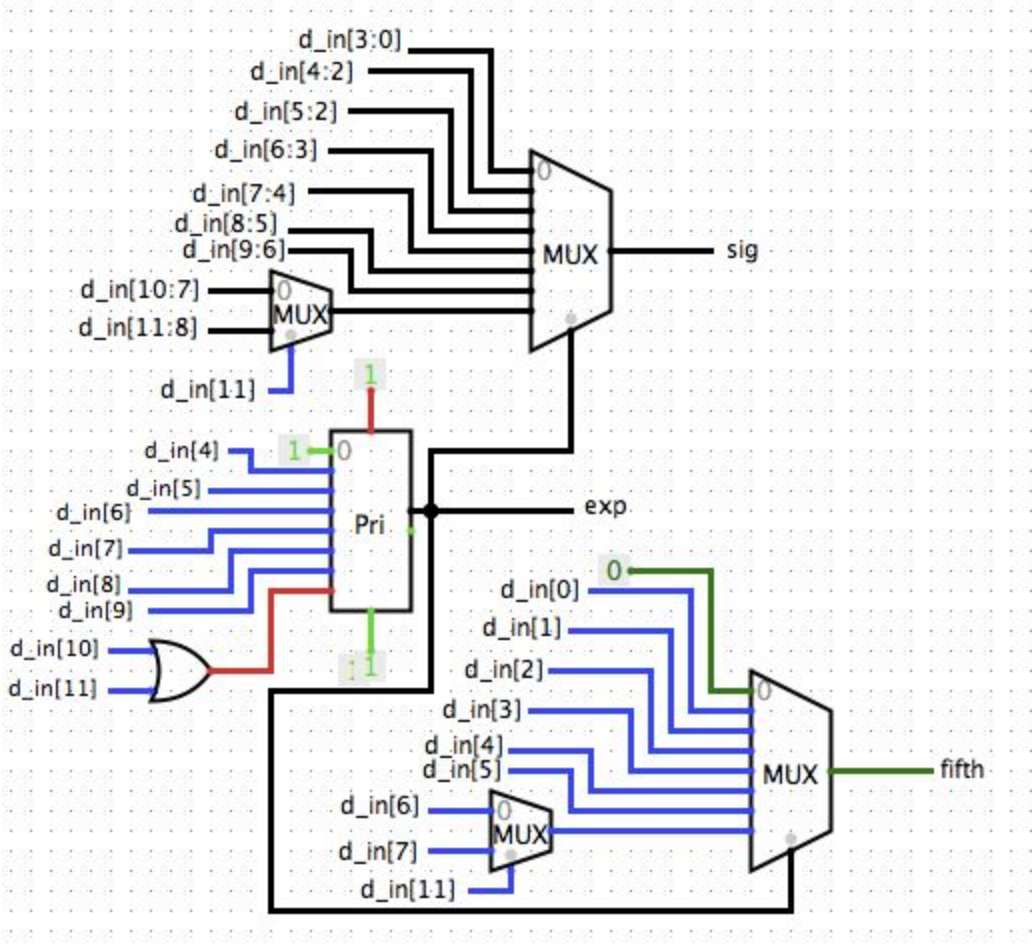

Figure 4: The above screenshot shows the schematic diagram for our linear_to_FP submodule that determines the number of leading zeros and the exponent value and obtains the fifth bit.

In the final step of the conversion, we need to round the number because we do not have enough bits in our output signal for the number to be entirely accurate. We do this using the rounding submodule. The specification tells us that we need to check if the fifth bit following the last leading 0 is 1 and if so, the significant should be incremented by one. We also check that if the significant overflows, the exponent should also be incremented by one and the significant should be shifted right by one bit.

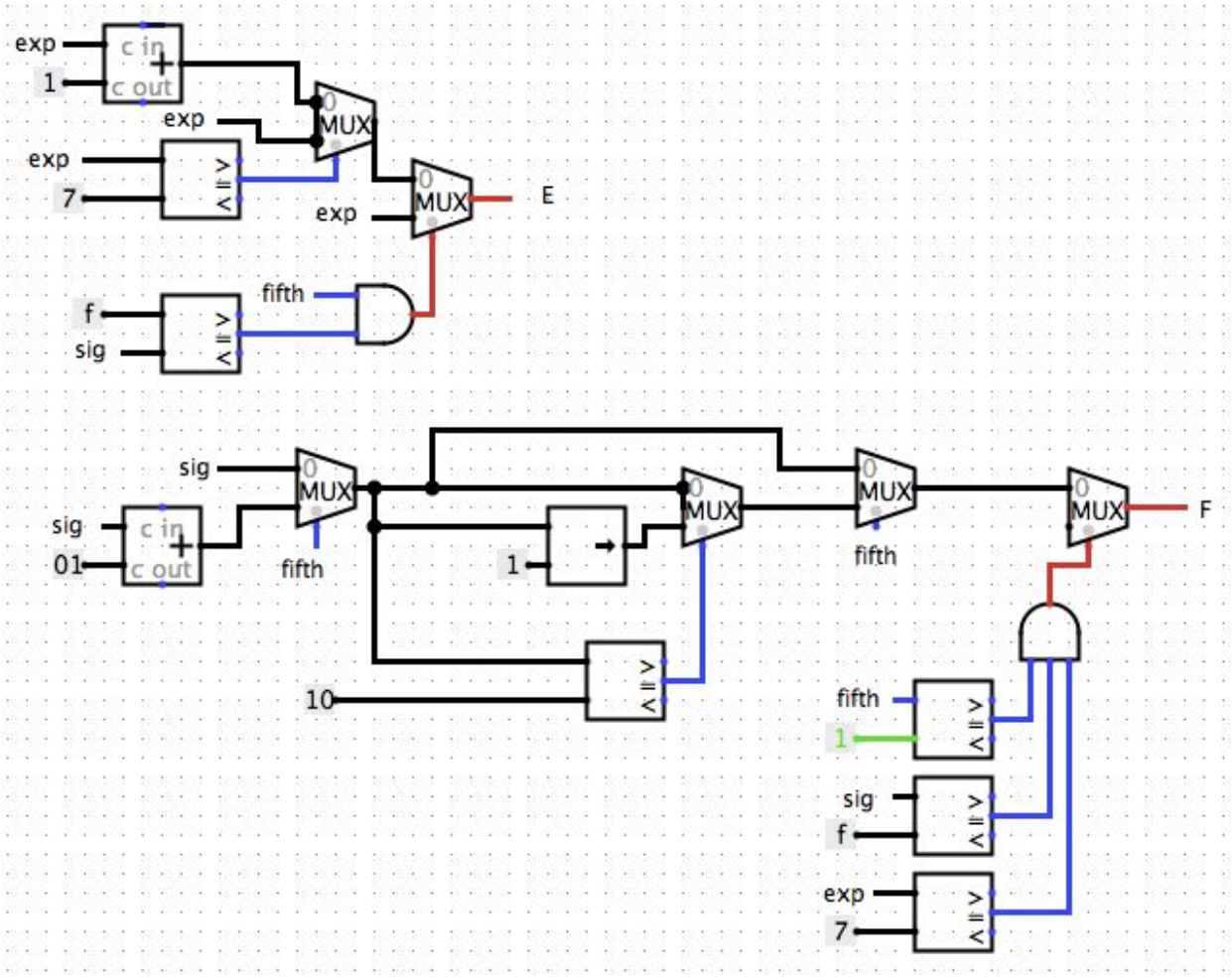

Figure 5: The above screenshot is the schematic diagram of the rounding submodule that ensures that our number is as accurate as possible, considering the number of bits we are allowed.

In order to round this number, we first looked to check for the case that both the fifth bit and the significand both contain only 1's for all of their digits. If this is the case, then we know that exponent will overflow when we add one to it as the instructions in the specification say to do. Therefore, we simply keep the value of the exponent as is for this particular case; otherwise, the resulting conversion does not resemble the original number that we had as input.

```
33   assign temp_exp = (fifth == 1 && sig == 4'b1111) ? ((exp == 3'b111) ? exp : exp + 1) : exp;
```

Figure 6: We preemptively check if we will increment the exponent and if overflow wil occur in the exponent to account for it.

For the next step, we then check if the fifth bit is 1, indicating that we need to add one to our significand value. We store this in a register that is 5 bits in order to account for overflow. Once

we calculate that intermediate value (temp_sig), we can then check if overflow occurred. If so, we need to shift the intermediate value by one. Otherwise, we can proceed with the value previously calculated.

```
36  assign temp_sig = (fifth == 1) ?  sig + 1 : sig;
37  assign temp_temp_sig = (fifth == 1) ? ((temp_sig == 5'b10000) ? temp_sig >> 1 : temp_sig) : temp_sig;
```

Figure 7: We test if overflow occurred in the case that we need to increment the significand.

Once the overflow has been accounted for, we can now assign our final values of E and F. However, though assigning E is straightforward, we need to account for the case where the value input into this module is the maximum possible value of the floating point representation. In that case, we need to have a special case for assigning F, which we account for in a ternary operator statement.

```
39  assign E = temp_exp;
40  assign F = (sig == 4'b1111 && exp == 3'b111 && fifth == 1'b1) ? 4'b1111 : temp_temp_sig[3:0];
```

Figure 8: Assigning the final values for E and F, wherein for F, we account for another edge case.

Simulation Documentation

In this lab, we were required to test a module that would convert a number from two's complement to a simple floating point representation.To test our program, we simulated a test bench that tested several different conversions of varying types, including some special edge cases.

We first tested arbitrary numbers and their corresponding negative values. In this initial test, we also included two clear edge cases, INT_MAX (2047) and INT_MIN (-2048).

```
52      //Testing Arbitrary Numbers & Their Negative Values
53      D = 69;
54      #100;
55      D = 100;
56      #100;
57      D = 2047;   //INT MAX
58      #100;
59      D = -69;
60      #100;
61      D = -100;
62      #100;
63      D = -2048;  //INT MIN
```

Figure 9: The varying numbers we tested, including INT_MIN and INT_MAX

Upon our first test, we saw that there were some issues so we remedied those by adding special cases in the submodules that are detailed above. Including these cases corrected the issues and yielded the correct results.

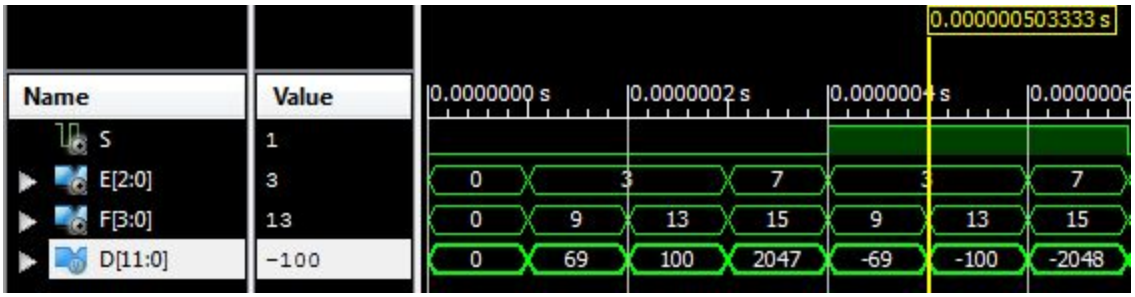

Figure 10: The waveform results for the test cases that were initially tested, showing the correct values for INT_MAX and INT_MIN**.**

We then began to test other cases that perhaps would fail, including common edge cases such as 1, -1, and also testing a number larger possible than the largest possible number using out simple and limited floating point representation: 1920. We also tested varying powers of two and multiples of powers of two.

```
66      //Testing Other Special Cases and Powers of Two
67      D = 1921;
68      #100;
69      D = -1921;
70      #100;
71      D = 1;
72      #100;
73      D = -1;
74      #100;
75      D = 8;
76      #100;
77      D = -8;
78      #100;
79      D = 24;
80      #100;
81      D = -24;
82      #100;
83      D = 16;
84      #100;
85      D = -16;
86      #100;
```

Figure 11: The various numbers that were tested, including a number larger than the maximum value of the simplified floating point representation and various powers of two and multiples of powers of two.

We saw that what we had since we created the edge cases for the INT_MIN and INT_MAX values still yielded the correct result once we checked the waveform.

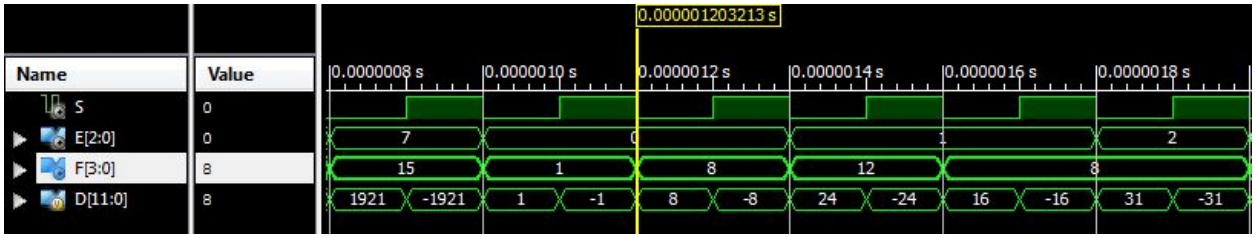

Figure 12: Waveform showing the correct values for the test cases that we had chosen.

Once we saw that what we had was working for cases of powers of two and large values, we decided to take a closer look at the different types of overflow that could occur, specifically in the values for E and F, more specifically, at the values from which E and F were derived: exp and sig. To test this, we looked at values that would cause either exp or sig to overflow specifically when rounding occured.

```
88      //Testing for Overflow of Exponent and/or Significance
89      D = 31;
90      #100;
91      D = -31;
92      #100;
93      D = 200;
94      #100;
95      D = -200;
96      #100;
```

Figure 13: Test cases that would cause exp and sig to overflow once rounding occured.

When we tested these values initially, we saw that we had not yet accounted for rounding overflow. Thus, when we remedied that issue, we were able to obtain the correct values.

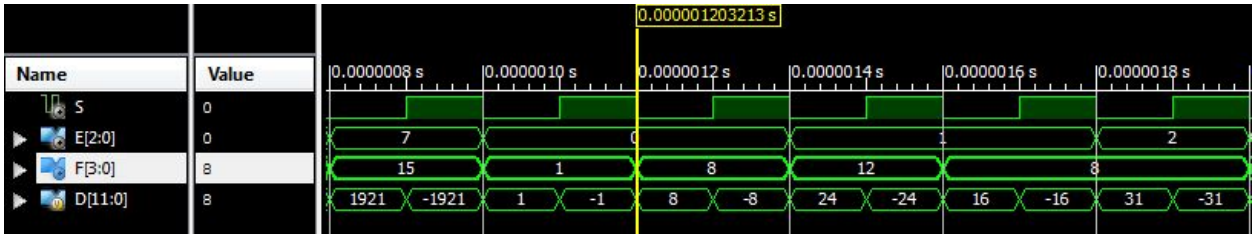

Figure 14: The last two values in the waveform show the success of the test case for 31, which causes specific overflow.

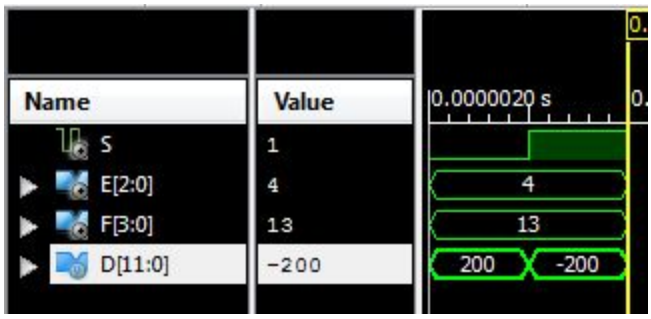

Figure 15: The value 200 causes a specific overflow.

By verifying all of these test cases, we were able to complete the demo.