CSM152A
Team 4 (Justin and Krish)

Lab Report 2

Introduction:

For this project, our primary objective was designing and implementing a combinational circuit that performs 8 bit floating bit conversion of a 12 bit linear 2's complement number. This conversion process involved breaking down the input data, represented in 12 distinct segments in the D[11:0] wire. The floating point conversion is carried out in the following manner: 1 bit for the sign, 3 bits for the exponent portion,, and 4 bits for the significand portion. Our code delves into the extraction of this sign bit in S and then computationally derives the exponent and significand according to the lab instructions. We employ the following equation: (-1)^S * F * 2^E, where S is the sign, F is the significand, and E is the exponent.

For certain decimal values, there can be multiple representations, such as the decimal value 56 which can be expressed as 00110111 and 00101110 (7 * 2^3 = 56 vs. 14 * 2^2). Our preferred representation follows normalization principles, emphasizing the significance of having the most significant bit (MSB) of the significand set to 1.

A crucial part of this experiment was dealing with rounding. Since some numbers in the 12-bit 2's complement format don't fit neatly into the 8-bit floating-point format, we had to round them. Our approach involved checking the fifth bit after the significand. If it's 0, we leave the significand as it is. But if it's 1, we increase the significand by 1.

We also had to handle two special situations to avoid errors. First, if rounding up the significand would make it too big, we reset it to 1000 and add 1 to the exponent. Second, if the exponent becomes too large, we set both the significand and exponent to their maximum values, represented by all 1's. These steps ensured our calculations stayed within the boundaries of the floating-point representation.

Design:

Our design is made up of a single module called fpcvt which takes a 12 bit two's complement integer (D) as its input and outputs a sign bit (S), a 3 bit exponent (E), and a 4 bit significand (F).

Sign Bit

The module starts by storing the most significant bit of D into S which gives us the sign bit. We then take the absolute value of D by inverting it and adding 1 and store this in the variable abs.

Leading Zeros

Afterwards, we count the number of 0 bits abs starts with by checking if the most significant bit is 0 and left shifting until we reach a 1 bit or we have left shifted 7 times. The number of times we left shift here is stored as an integer i and will later on be used to determine our exponent.

```
assign temp = abs;
	while(temp[11] != 1 && i < 8)
	begin
	   i = i + 1;
	   assign temp = temp << 1;
	end
```

Code used for counting number of leading 0s of abs

Significand Rounding

Next, using the left shifted value we obtained, we store the 4 most significant bits in a variable called tempF which we will use to determine our significand. We then check if we need to round up the significand by checking if the fifth most significant bit of our left shifted value is a 1, if not we do nothing to tempF. Otherwise we do two things. First, we check if tempF has the value 1111 and if this is true we decrement i by 1 and right shift tempF one time (this will have the effect of incrementing our exponent by 1 and dividing our significand by 2 since otherwise the significand would overflow). And second, we add 1 to tempF.

```
assign tempF = temp[11:8];
if (temp[7] == 1)
	begin
	   if (tempF == 4'b1111)
	   begin
	      i = i - 1;
	      assign tempF = tempF >> 1;
	   end
	   assign tempF = tempF + 1;
	end
```

Code for significand rounding

Overflow Check
As a final check, if the value of i is 0, we increment i by 1 and set tempF to 1111, this will set the output to the max value if the input is too large.

Exponent and Significand Output
Finally, we set the value of E = 8 - i and F = tempF, giving us our exponent, and significand.

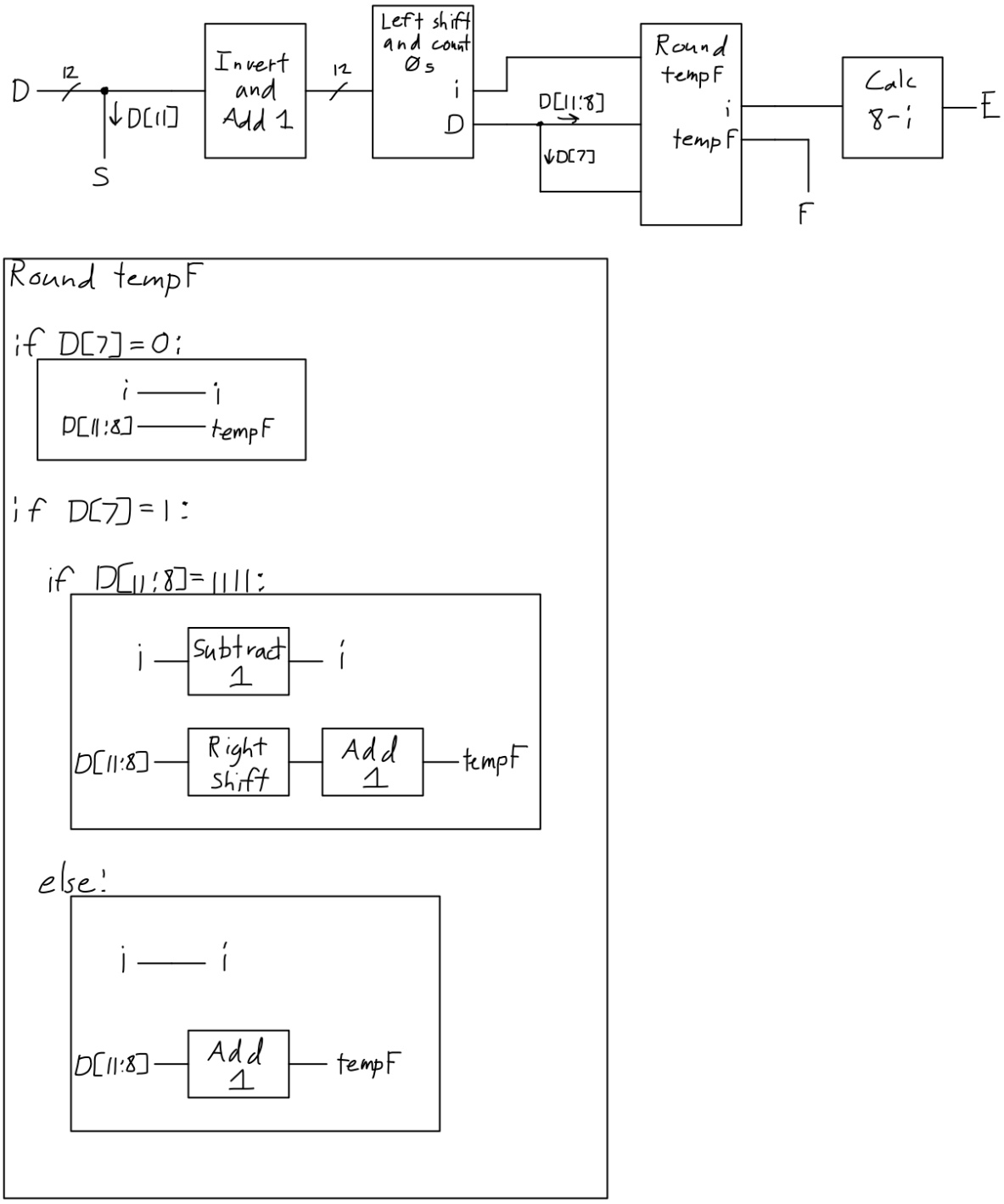

Figure 1. Logical design diagram for the floating point conversion program

Simulation and test cases:

To ensure our float-point conversion circuit works correctly, we ran several test cases that we designed ourselves, and checked them by manually assigning D to those test cases. carefully checked the outputs against what we expected for each test.
Test Cases:

1. **Positive Number:**
    - Input: 0 (000000000000 in binary)
    - Expected Output: Sign bit (s) = 0, Exponent (e) = 0, Significand (f) = 0
    - Explanation: We expected the sign bit to be positive, and the exponent and significand values to match the input values, and we checked it with the final S, E , and F values.
2. **Negative Number:**
    - Input: -422 (111001011010 in binary)
    - Expected Output: Sign bit (s) = 1, Exponent (e) = 5, Significand (f) = d
    - Explanation: With a negative input, the sign bit should be 1. The exponent and significand values should be equal to the approximation done by the program.
3. **Smallest Negative Input:**
    - Input: -2048 (100000000000 in binary)
    - Expected Output: Sign bit (S) = 1, Exponent (E) = 7, Significand (F) = f
    - Explanation: When the input the smallest possible 12 bit number, then the exponent and the significand would have the largest values and the sign should be 1 indicating a negative number.
4. **Largest Positive Number**
    - Input: 2047 (011111111111 in binary)
    - Expected Output: Sign bit (s) = 0, Exponent (e) = 7, Significand (f) = f
    - Explanation: This input represents a case where the value D is the largest 12 bit number that can be represented. Similar to the explanation above, the sign bit should be zero, and the F and F values should be the maximum values
5. **Overflow Correction and rounding to the next exponent.**
    - Input: 1928(0111 1000 1000 in binary)
    - Expected Output: Sign bit (s) = 0, Exponent (e) = 7, Significand (F) =f
    - Explanation: This is another case where the value is very close to the largest value that can be represented in 8 bit floating point representation. We experimented with values around that and found that the exponent increments from and e to an f when D is around 1728 in decimal (due to rounding being carried out).

Issues and Bugs Encountered:
During testing, we encountered a problem with handling the exponent, where we were doing a check to see whether I was equal to zero, in which case we could increment I to

1(as the exponent would be 8-i and E was a 3 bit binary number, and then we assign F to be the largest 4 bit binary number, however our if statement wasn't working due to if operators not working outside the always block. Thus instead of checking whether E was greater than 7, we put this condition near the end of the always block:

```
if(i==0)
dd      begin
        i= i+1;
        assign tempF = 4'b1111;
        end
```

Given below are some of the inputs with their respective floating point conversions:

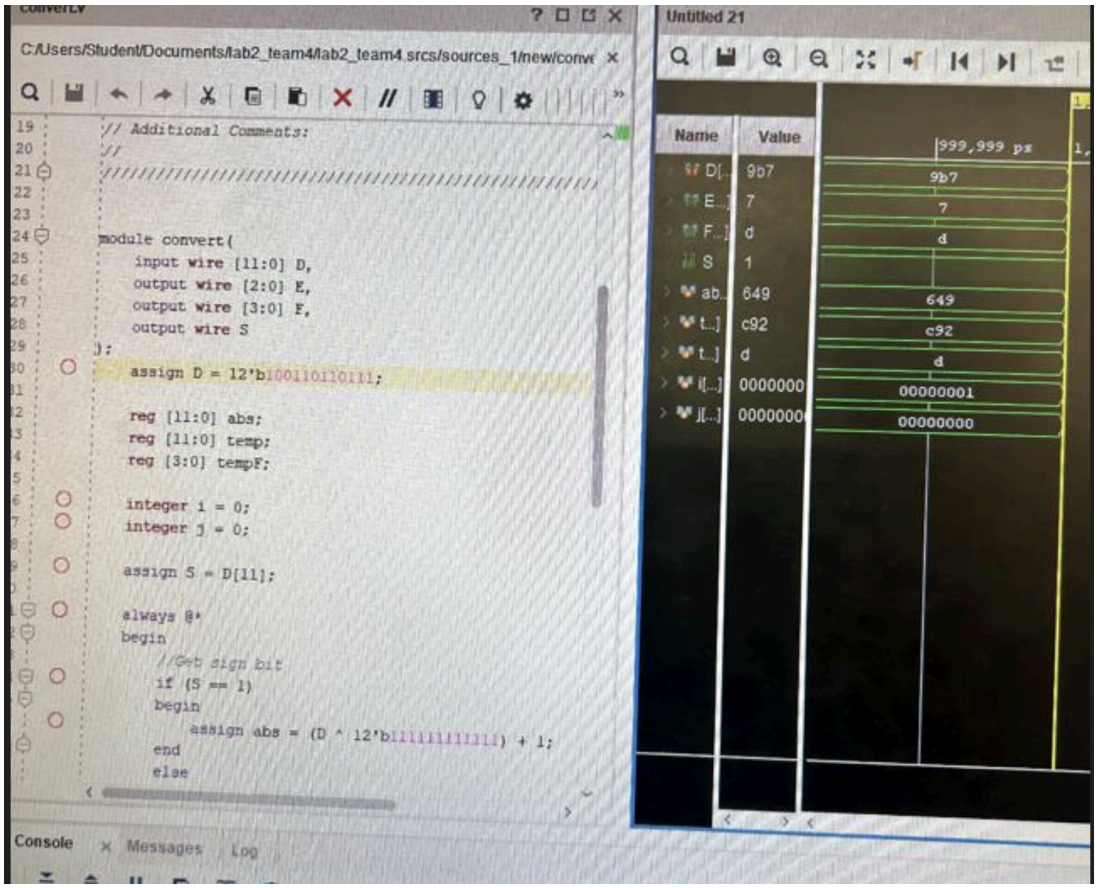

Figure 2: S, E, and F values for when D = 100110110111(random negative number)

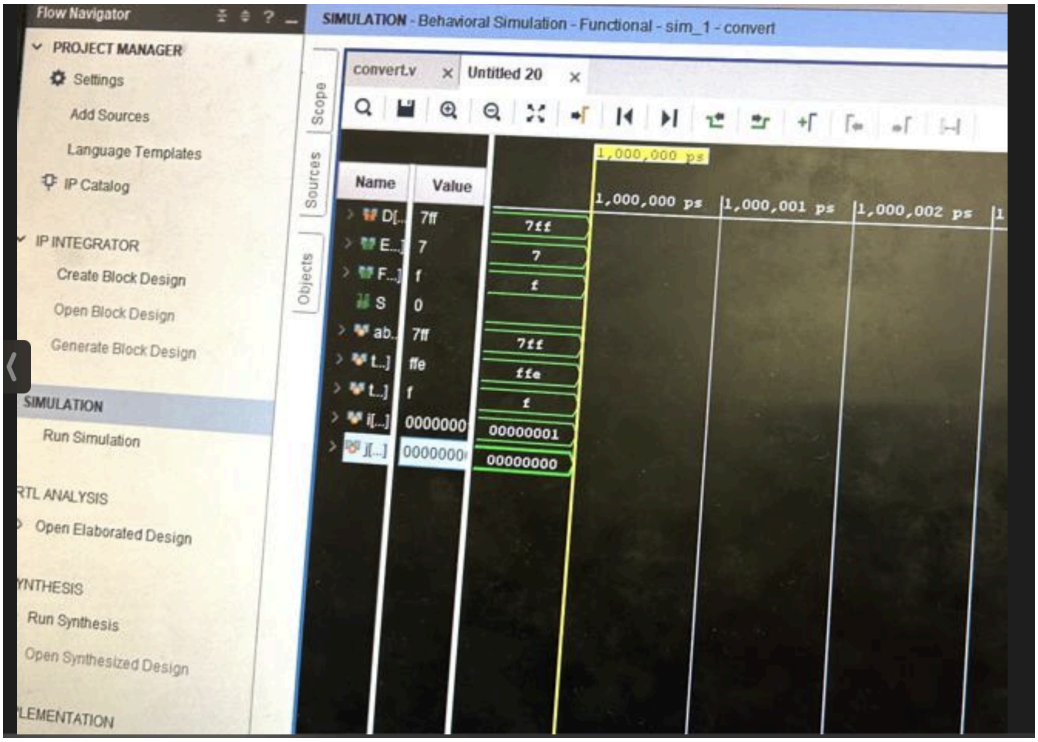

Figure 2. S, E, and F values for the value D = 1928(Decimal representation)

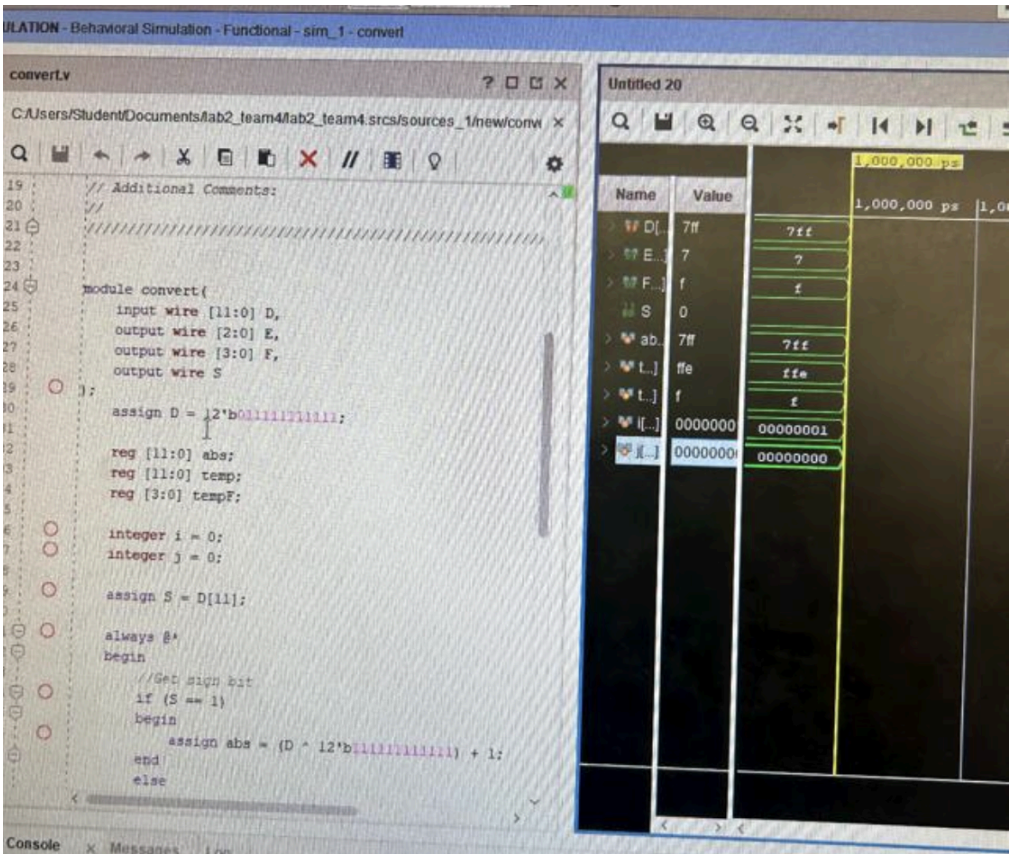

Figure 3. S, E, and F values for the largest positive number(D = 011111111111, 2047)

Conclusion:

In conclusion, our project aimed at designing and implementing a combinational circuit for converting a 12-bit linear 2's complement number into an 8-bit Floating Bit representation was successful, following the lab instructions to ensure accuracy, handling rounding and extreme cases, etc in our implementation.

Throughout the project, we encountered challenges, notably in handling rounding and overflow scenarios. However through debugging and problem-solving, we were able to address these issues effectively. We fixed a critical problem related to handling the exponent, ensuring our circuit produces accurate results under all conditions.

Our testing strategy involved creating diverse test cases, covering various input scenarios, both positive and negative numbers, as well as edge cases such as the smallest and largest representable numbers. By simulating these test cases and comparing the outputs with expected results, we verified the correctness and reliability of our float-point conversion circuit.

The simulation results showed the effectiveness of our circuit design, producing accurate floating-point representations for a wide range of input values. Moreover, the manual verification process provided additional confidence in the functionality of our circuit.

In conclusion, our project not only fulfilled the lab requirements but also provided valuable insights into floating-point conversion and digital circuit design. Through collaboration efforts and testing using diverse cases, we successfully implemented our float-point conversion circuit in Verilog, meeting the objectives of the assignment.