CS 152A Lab 2: Floating Point Conversion

Gee Won (Jennifer) Jo, Andrew Lin, Daxuan (Albert) Shu
704171725, 204455444, 204853061
CS 152A Lab 5
May 9th, 2017

Introduction

In Lab 2, we implemented a module to convert from a 12-bit linear encoding of an analog signal to a 8-bit Floating Point value shown in figure 1. This involved not only converting between the encodings, but also handling maximum values, rounding and other abnormal behavior.

Design Description

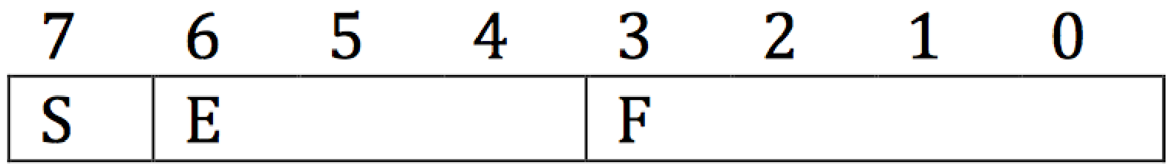


Figure 1: Floating Point Representation

The floating point representation consists of a 1-Bit Sign Representation, a 3-Bit Exponent, and a 4-Bit Significand. The representation was calculated using the formula:

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

Our floating point converter was implemented in a single module, but the module had multiple well defined portions including converting 2's complement to sign-magnitude, counting leading zeros, extracting the significand, and rounding. The module finally outputs the floating point representation as described in Figure 1.

Overall block diagram shown as below:

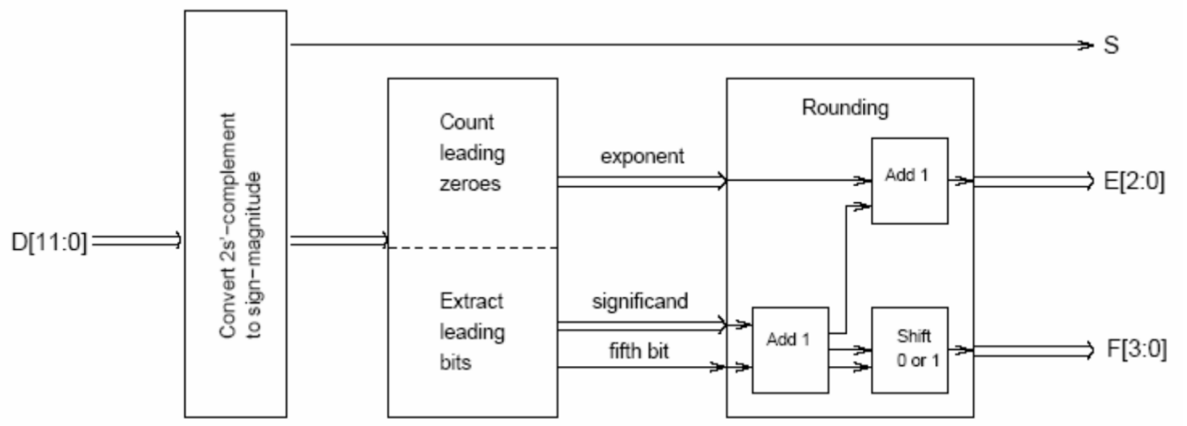
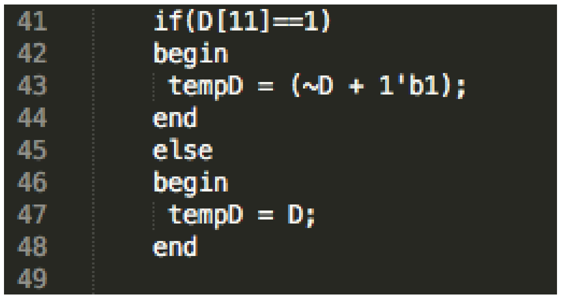


Figure 2: the overall design block diagram

Our first portion gets the sign bit from the linear encoding, and creates a temporary version of the encoding for parsing purposes. This meant that, if the encoding was for a positive number, the temporary version was just the same, but if it were a negative number, then the encoding had to be translated by flipping all the bits and adding one. Shown as figure 2



```
if(D[11]==1)
begin
 tempD = (~D + 1'b1);
end
else
begin
 tempD = D;
end
```

Figure 3: convert 2's complement to sign-magnitude

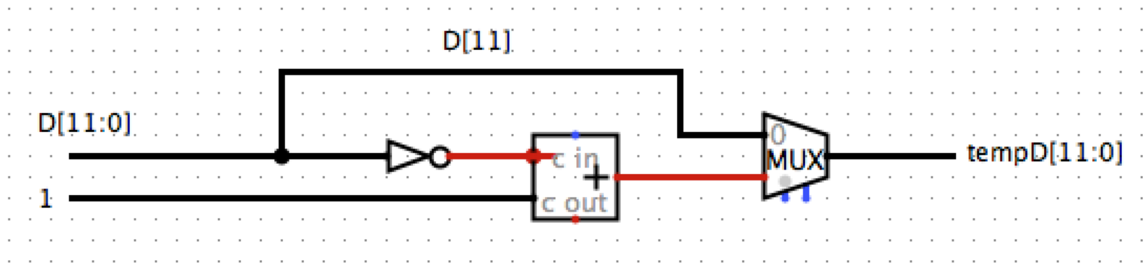
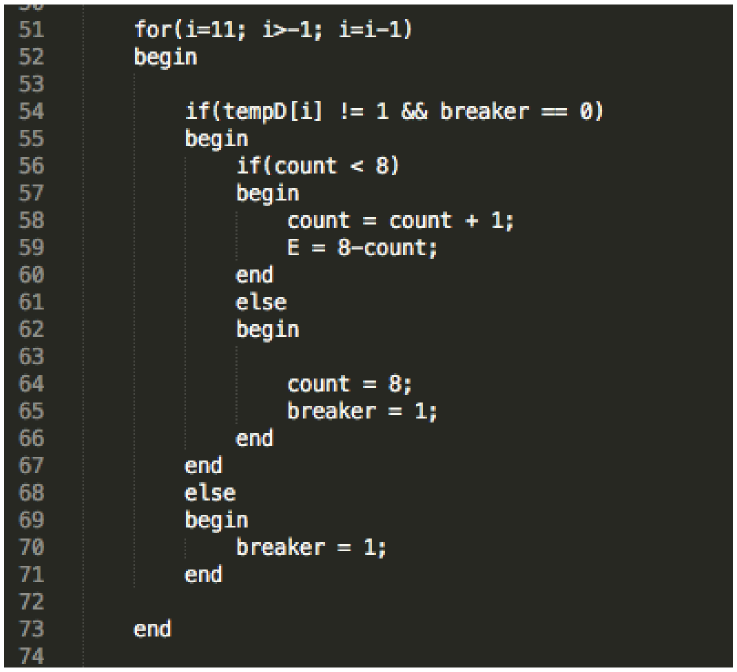


Figure 4: Our design for this portion based on our code.

Notice that we did not consider negative overflow at here. Instead, we would deal with that case at the Rounding portion.

Our next part(Figure 3) counted the number of leading zeroes that the parsable linear encoding has, in order to get the value of the floating point's exponent.



```
for(i=11; i>-1; i=i-1)
begin

    if(tempD[i] != 1 && breaker == 0)
    begin
        if(count < 8)
        begin
            count = count + 1;
            E = 8-count;
        end
        else
        begin

            count = 8;
            breaker = 1;
        end
    end
    else
    begin
        breaker = 1;
    end

end
```

Figure 5: Count leading zeros

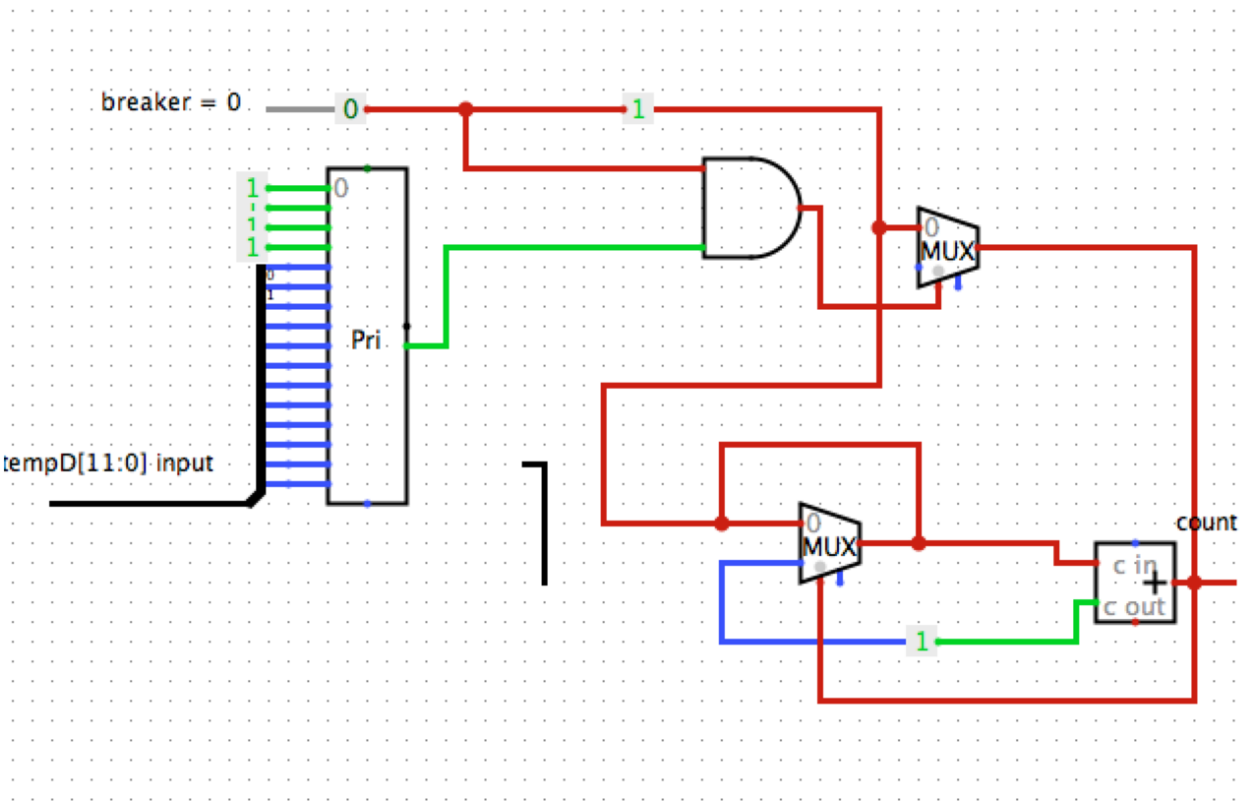
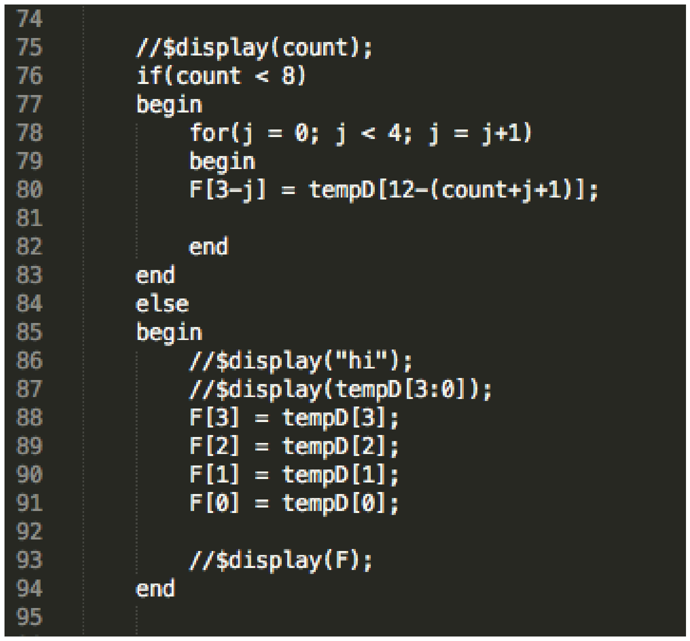


Figure 6 : Count Leading Zero

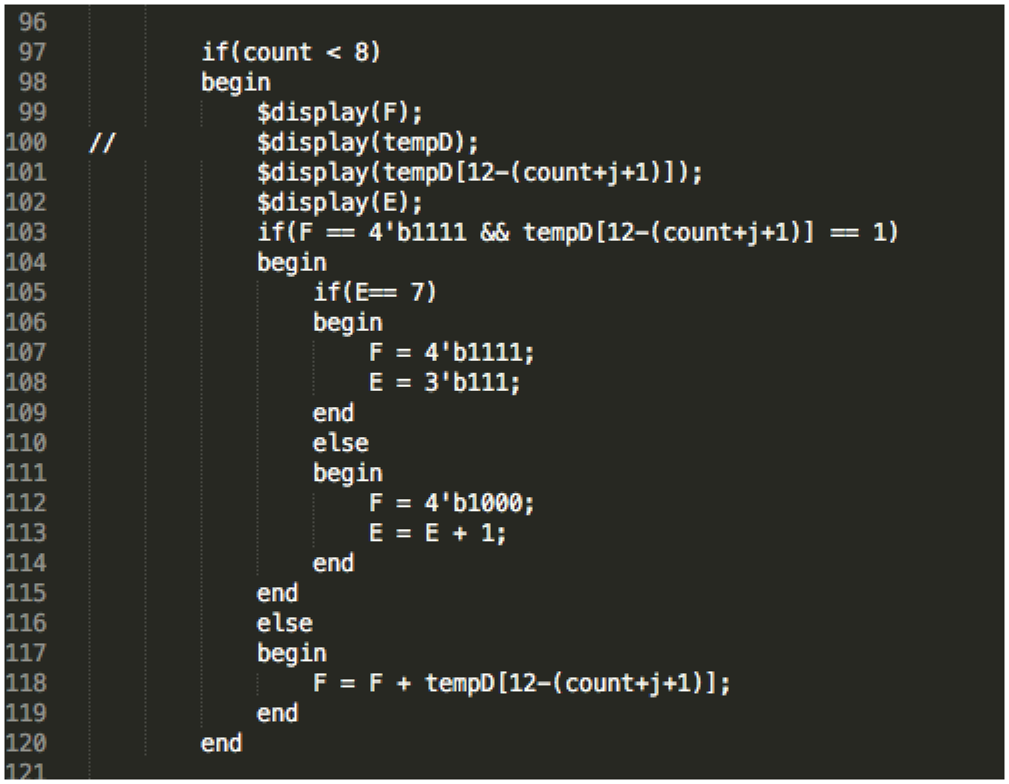
Next, we use the count from the previous part, and use the next four bits to get our floating point's significand. We also start error handling in this case. If there were too many leading zeroes in the 12-bit linear encoding and there aren't four bits after to use, then the significand is just set to the last 4 bits of the linear encoding.



```

        //$display(count);
        if(count < 8)
        begin
            for(j = 0; j < 4; j = j+1)
            begin
            F[3-j] = tempD[12-(count+j+1)];

            end
        end
        else
        begin
            //$display("hi");
            //$display(tempD[3:0]);
            F[3] = tempD[3];
            F[2] = tempD[2];
            F[1] = tempD[1];
            F[0] = tempD[0];

            //$display(F);
        end

```

Figure 7: Extract Significand

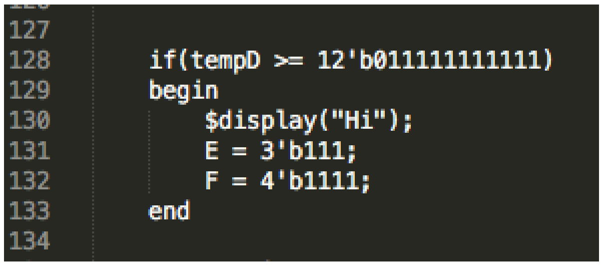
Next, our module handles rounding. This meant that if the 5th bit after the four significand bits was on, then the 4 bits had to be rounded up. It was possible that this would overflow the significand, in which case the significand was halved and the exponent was increased. If this made the exponent overflow, then the exponent and significand were just set to the largest possible values.



```

        if(count < 8)
        begin
            $display(F);
//          $display(tempD);
            $display(tempD[12-(count+j+1)]);
            $display(E);
            if(F == 4'b1111 && tempD[12-(count+j+1)] == 1)
            begin
                if(E== 7)
                begin
                    F = 4'b1111;
                    E = 3'b111;
                end
                else
                begin
                    F = 4'b1000;
                    E = E + 1;
                end
            end
            else
            begin
                F = F + tempD[12-(count+j+1)];
            end
        end

```

Figure 8: Rounding based on the fifth bit

Finally, the module handles errors if the encoded number is larger than the largest possible floating point value, or smaller than the smallest possible floating point value. In this case, the output is just capped as be the largest or smallest possible values.



```

    if(tempD >= 12'b011111111111)
    begin
        $display("Hi");
        E = 3'b111;
        F = 4'b1111;
    end

```

Figure 9: Rounding very large numbers (overflow)

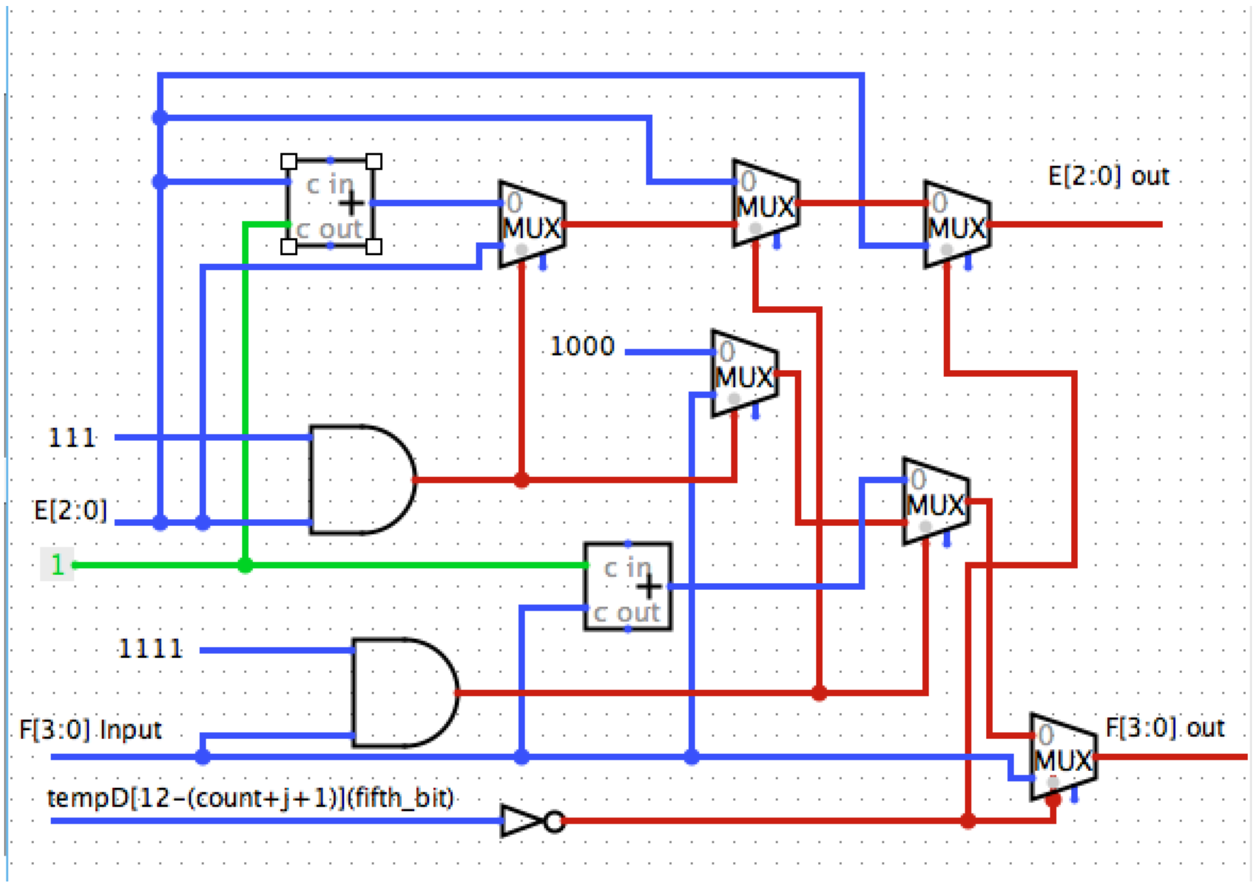


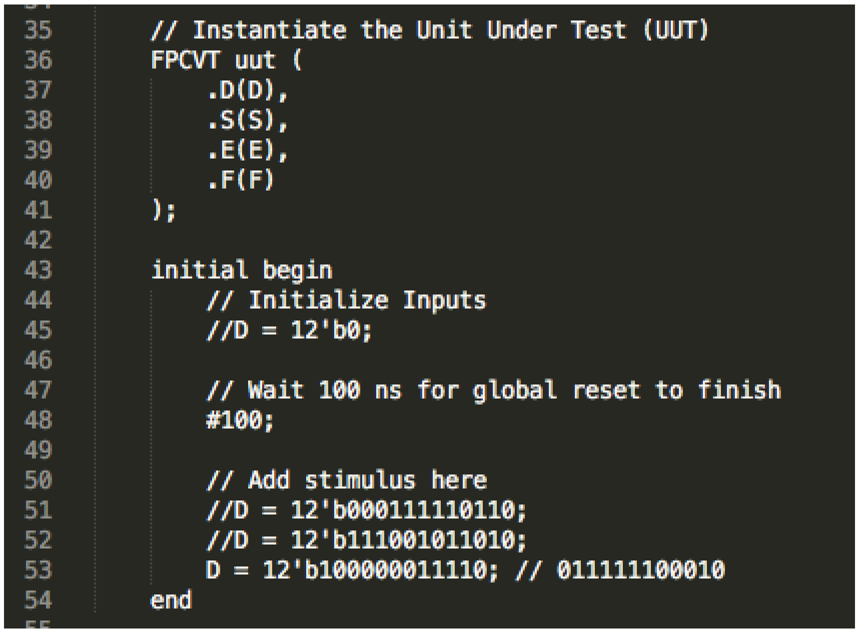
Figure 10: Using multiplexers as if else statements.
If the fifth bit is zero, then we simply output the input. However, if the fifth bit is 1, then we increase the significand by 1 and if the result is overflow, then add 1 to the exponent and shift the significand to 1000. If both overflow, it should output all 1's.

Simulation Documentation

Our primary means of testing was the TA's demo test script, which included over 5000 test cases that also covered multiple edge cases that our implementation failed at first.

An example of an edge case we originally failed was that values that were too small would be not be capped correctly. Additionally, when rounding, it was possible for the exponent to go above the maximum value (7), but we did not catch this possibility. Of course, all of these problems were eventually fixed.

In order to test individual cases, we also created a test bench file. This file was really simple, and just fed various values for the 12 bit linear encoding (stored in a register 'D') to our translator file, and received the translated sign bit ('S'), exponent ('E'), and significand ('F'). Here, we can see three linear encodings that we used to test our module. Changing the value for D feeds a different test case to our module.



```
	// Instantiate the Unit Under Test (UUT)
	FPCVT uut (
		.D(D),
		.S(S),
		.E(E),
		.F(F)
	);

	initial begin
		// Initialize Inputs
		//D = 12'b0;

		// Wait 100 ns for global reset to finish
		#100;

		// Add stimulus here
		//D = 12'b000111110110;
		//D = 12'b111001011010;
		D = 12'b100000011110; // 011111100010
	end
```

Figure 11: Test bench file

Conclusion

After a few iterations on our design, our module eventually worked and passed all of the TA's test cases. Since many of the test cases were clearly geared towards testing a specific type of edge case, it was relatively easy to identify which part of our module corresponded to the bug. Thus, most of the errors we encountered were easily fixed by slightly tweaking the code in the corresponding part of our module.

The most difficult bug we encountered was not actually related to the test cases. Verilog does not allow loops to be terminated early; all loops must run a specific, unvarying number of times. As such, we could not just trivially increment the count of leading zeroes, but we also needed a breaker value to keep track of whether the first one had been seen, in which case we would stop counting zeroes. What was surprising was that the value for this breaker as well as the value for the count of leading zeroes did not get reset if the module was called multiple times. Since Verilog's test bench actually calls the module with the default value of D, which is 0, we received very weird outputs at first where the count was always maxed out. This bug was fixed by resetting the values for the count and breaker at the end of the module's execution.

Our group members worked together on all parts of the project, helping each other design and implement our module, as well as helping to identify and fix bugs.