Lab 2

12pm-2pm

Calvin Liu

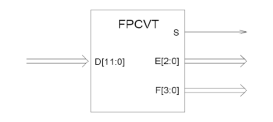
804182525

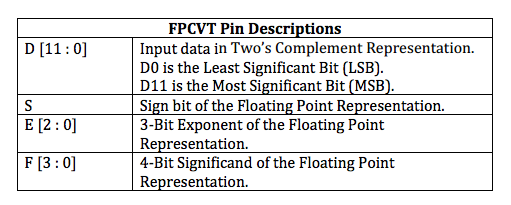
Darin Minamoto

704140102

**Introduction:**

The purpose of the lab was to create a module that converts a 12-bit linear encoding of an analog signal into a compounded 8-bit floating-point representation. A 12-bit number in binary would be fed into the module and a 8-bit floating-point number is outputted with 3 different parts: the significant F, the exponent magnitude E, and the sign bit S. This conversion should take into account all possibilities including certain overflows and rounding.









The number the floating-point number represents can be calculated with the above equation.

**Design:**

We started off with finding a way of parsing the value of the 12-bit input into the corresponding 8-bits needed. The most significant sign bit was just taken since the first bit in the 12 bit input is the same as the first bit in the 8-bit floating-point output.

For the exponent we had to first count the number of leading zeros in the 12-bit input. This was done with a brute force method since we only needed to count a maximum of 8 zeros.

if (flipRegister[11] == 0) begin

count = 1;

if (flipRegister[10] == 0) begin

count = 2;

if (flipRegister[9] == 0) begin

count = 3;

if (flipRegister[8] == 0) begin

count = 4;

if (flipRegister[7] == 0) begin

count = 5;

if (flipRegister[6] == 0) begin

count = 6;

if (flipRegister[5] == 0) begin

count = 7;

if (flipRegister[4] == 0) begin

count = 8;

end

end

end

end

end

end

end

end

After registering the number of zeroes we calculated the exponent by subtracting the number of zeroes from 8. If there were 8 or more zeroes then we set the number of zeroes to be 8 at which the exponent would be 0.

E = 8 - count;

For the significant, we just registered the next immediate 4 bits that came after the number of zeroes.

F = (flipRegister >> (8 - count));

We then had to handle the case of overflow where we divided the significant and increase the value of the exponent. If the exponent overflowed then we left the sign bit the way it was based on the specifications in the lab.

if (F == 4'b1111) begin //4b'1111

F = 4'b0111;

E = E+add1;

end

F = F+add1;

We then had to handle the case of negatives in which we flipped it in 2’s complement to a positive number and then processed the number the same way as the other positive numbers we had.

if(D[11] == 1) begin

flipRegister = ~D;

flipRegister = flipRegister + add1;

end

else begin

flipRegister = D;

end

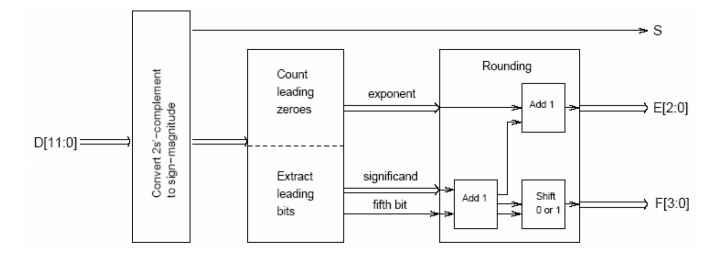
We then had to handle the case of rounding in which we observed the immediate digit after the significant and saw if we needed to round or not. If the digit was 1 then we rounded, and then if the digit was 0, we didn’t round.

temp = flipRegister[lastindex];

if (temp == 1) begin

….

F = F+add1;



The code can be compared to the module shown above that shows the break down of what each step of the module is supposed to do.

**Which method is easier? Checking for overflow after or before adding rounding bit?**

It is easier to check for overflow before the addition of the rounding bit because if we add the rounding bit then check for overflow it might have already overflowed which would make it difficult to detect it. Also if we add it before the overflow, then we can modify the register containing 1111 and shift it to 1000 (essentially dividing it by 2) and then raise the exponent.

**Simulation and Testbench:**

We made the test cases to check the module for the edge cases of rounding, overflow, negative numbers, and normalized numbers. The results of each test is printed out whenever D changes through the always block. We worked out the answers to these test cases that were not in the lab2 specification by hand and put them in the comments.

initial begin

// Initialize Inputs

D = 0;

// Wait 100 ns for global reset to finish

#100

//round up SOL: 0 010 1100

D = 12'b000000101110;

#100;

//round down SOL: 0 010 1011

D = 12'b000000101101;

#100;

//overflow SOL: 0110 1000

D = 12'b000111110011;

#100;

//negatives no rounding SOL: 1111 1110

D = 12'b100011101010;

#100;

//negative round up SOL: 1111 1110

D = 12'b100100011110;

#100;

//negative F overflow SOL: 1111 1000

D = 12'b110000000011;

#100;

//rounding with overflow SOL: 0100 1000

D = 12'b000001111100;

#100;

//Normalized respresentation SOL: 0010 1110

D = 12'b000000111000;

#100;

//More than 8 0s SOL: 0000 0111

D = 12'b00000000111;

#100;

//-40 SOL: 1010 1010

D = 12'b111111011000;

#100;

D = 0;

#100;

end

always @(D) begin

$display("S: %b E: %b F: %b", S, E, F);

end

Running the simulation produced the following output (without the parenthesis):

S: 0 E: 000 F: 0000 (initialize D)

S: 0 E: 010 F: 1100 (round up)

S: 0 E: 010 F: 1011 (round down)

S: 0 E: 110 F: 1000 (overflow)

S: 1 E: 111 F: 1110 (negative (no rounding))

S: 1 E: 111 F: 1110 (negative round up)

S: 1 E: 111 F: 1000 (negative overflow)

S: 0 E: 100 F: 1000 (rounding with overflow)

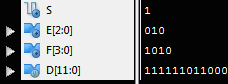
S: 0 E: 010 F: 1110 (normalized)

S: 0 E: 000 F: 0111 (more than 8 0s)

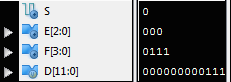
S: 1 E: 010 F: 1010 (-40)

Which verifies that the output of the module matches what we expected based on the specification. We also ran each test case separately and took screenshots of the waveform output which produces the same result.

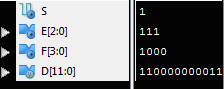
-40



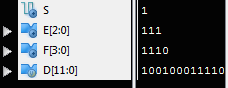
More Than 8 Zeros



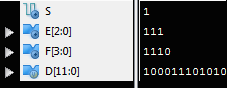
Negative F Overflow



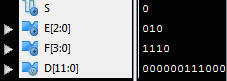
Negative Round Up



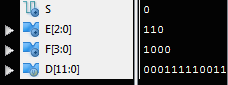
Negatives No Rounding



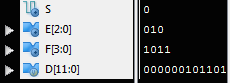
Normalized



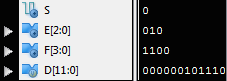
overflow



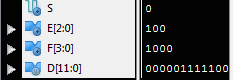
Rounding down



Rounding Up



Rounding with overflow



**Conclusion:**

The experiment in implementing the floating-point converter proved to be a success. By hand, we calculated what the expected output should be based on the floating-point conversion rules for many scenarios. These scenarios include negative binary input, overflows for the registers used for the significant, sign bit, and the exponent, and rounding. Many tests were made and compared to our calculated output by hand. The calculated output did match up with the output we got when we ran the testbench on the module. The inputs were tested one at a time and the output was observed through iSIM.

**Individual Efforts:**

We both shared the work equally. The coding was done in the lab as a collaboration for the module and for the test bench. The lab report was also shared. Out of the four different subcategories in the lab report for reference, we each took two different subcategories. I did the introduction and design while Darin did the testbench and the conclusion.