Lab 2 Report

● Graded

Group

EMANUEL ZAVALZA
KAI WANG
View or edit group

Total Points

39 / 40 pts

Question 1

Introduction and requirement

10 / 10 pts

✓ **− 0 pts** Correct

Question 2

Design description

14 / 15 pts

✓ **− 0 pts** Correct

− 1 pt Missing details

✓ **− 1 pt** Schematic/figures missing

Question 3

Simulation documentation

10 / 10 pts

✓ **− 0 pts** Correct

Question 4

Conclusion

5 / 5 pts

✓ **− 0 pts** Correct

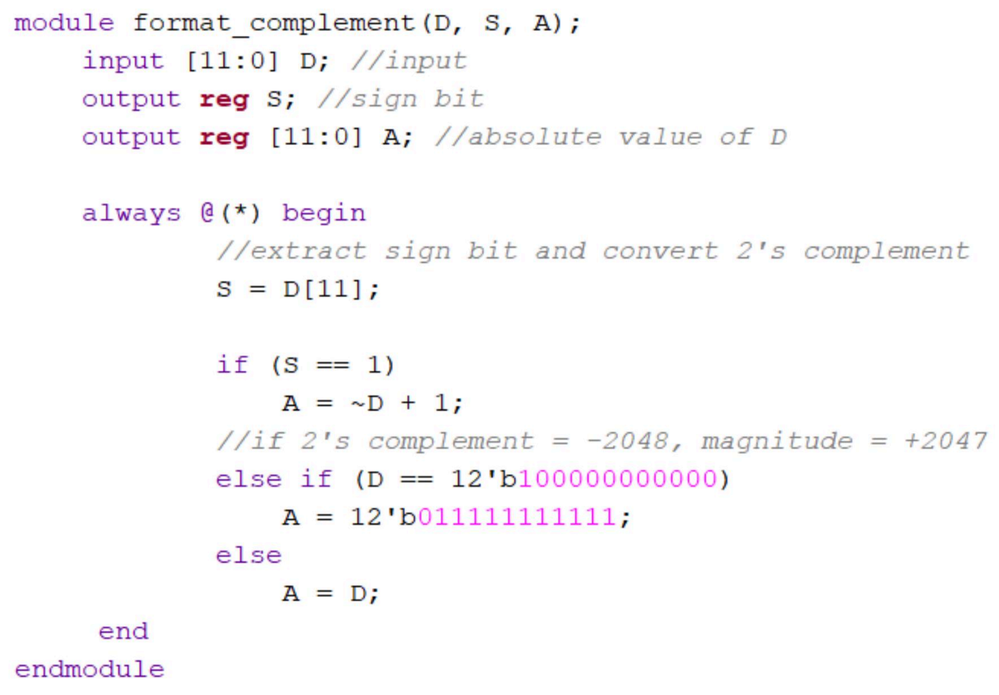
Questions assigned to the following page: 1 and 2

1. Introduction and Requirement

In this lab we are implementing a combinational circuit that converts a twelve bit linear encoding of an analog signal into a compounded eight bit floating point representation. This is useful as it enables us to represent more real numbers, like fractions that aren't possible with two's complement representations for integers. The conversion outputs the closest floating point number by extracting 3 components that make one up: sign bit, the Significand, and the exponent.

2. Design Description

This floating point converter utilizes 3 modules arranged sequentially: format_complement to handle negative numbers by setting the sign bit, priority_encoder to determine the significand, exponent, and overflow bit, then finally rounding_logic that utilizes the overflow bit to determine how the intermediate value should be rounded to determine the final result. See the pictures below for the interface and comments that describe each module. Lastly, module fpcvt chains together the 3 modules to take in input number D and output final floating point number in 3 parts: sign bit S, 3 bit exponent number E, and 4 bit Significand.



```
module format_complement(D, S, A);
    input [11:0] D; //input
    output reg S; //sign bit
    output reg [11:0] A; //absolute value of D

    always @(*) begin
            //extract sign bit and convert 2's complement
            S = D[11];

            if (S == 1)
                A = ~D + 1;
            //if 2's complement = -2048, magnitude = +2047
            else if (D == 12'b100000000000)
                A = 12'b011111111111;
            else
                A = D;
     end
endmodule
```

```verilog
module priority_encoder(A, B, R, O);
    input [11:0] A; //absolute value of input
    output reg [2:0] B; //unoverflowed exponent
    output reg [3:0] R; //unrounded significand
    output reg O; //overflow rounding bit
    integer i;
    integer break = 0;

    always @(*) begin

        casez (A) // casez allows us to use 'z' for don't care
            12'b1??????????? : begin B = 3'd7; R = 4'b1111; O = 1; end //-2048
            12'b01?????????? : begin B = 3'd7; R = A[10:7]; O = A[6]; end // 1 leading zero
            12'b001????????? : begin B = 3'd6; R = A[9:6]; O = A[5]; end // 2 leading zeroes
            12'b0001???????? : begin B = 3'd5; R = A[8:5]; O = A[4]; end // 3 leading zeroes
            12'b00001??????? : begin B = 3'd4; R = A[7:4]; O = A[3]; end // 4 leading zeroes
            12'b000001?????? : begin B = 3'd3; R = A[6:3]; O = A[2]; end // 5 leading zeroes
            12'b0000001????? : begin B = 3'd2; R = A[5:2]; O = A[1]; end // 6 leading zeroes
            12'b00000001???? : begin B = 3'd1; R = A[4:1]; O = A[0]; end // 7 leading zeroes
            12'b00000000???? : begin B = 0; R = A[3:0]; O = 0; end // 8 or more leading zeroes
            default: begin B = 0; R = 0; O = 0; end
        endcase
    end


endmodule
```

```verilog
module rounding_logic(R, F, O, B, E);
    input [3:0] R; //unrounded significand
    input [2:0] B; //unoverflowed exponent
    input O; //overflow rounding bit
    output reg [3:0] F; //final rounded significand/mantissa
    output reg [2:0] E;  //final exponent

    reg [4:0] overflow_F;
    reg [3:0] overflow_E;

    always @(*) begin
        overflow_F = R + O;
        overflow_E = B + overflow_F[4]; //increase exponent by one if overflow

        //catch exponent overflow edge case
        if (overflow_E[3]) begin
            F = 4'b1111;
            E = 3'b111;
        end
        //check for F overflow
        else begin
            F = overflow_F >> overflow_F [4]; //shift if overflowed
            E = overflow_E[2:0];
        end
    end
endmodule
```

Question assigned to the following page: 2

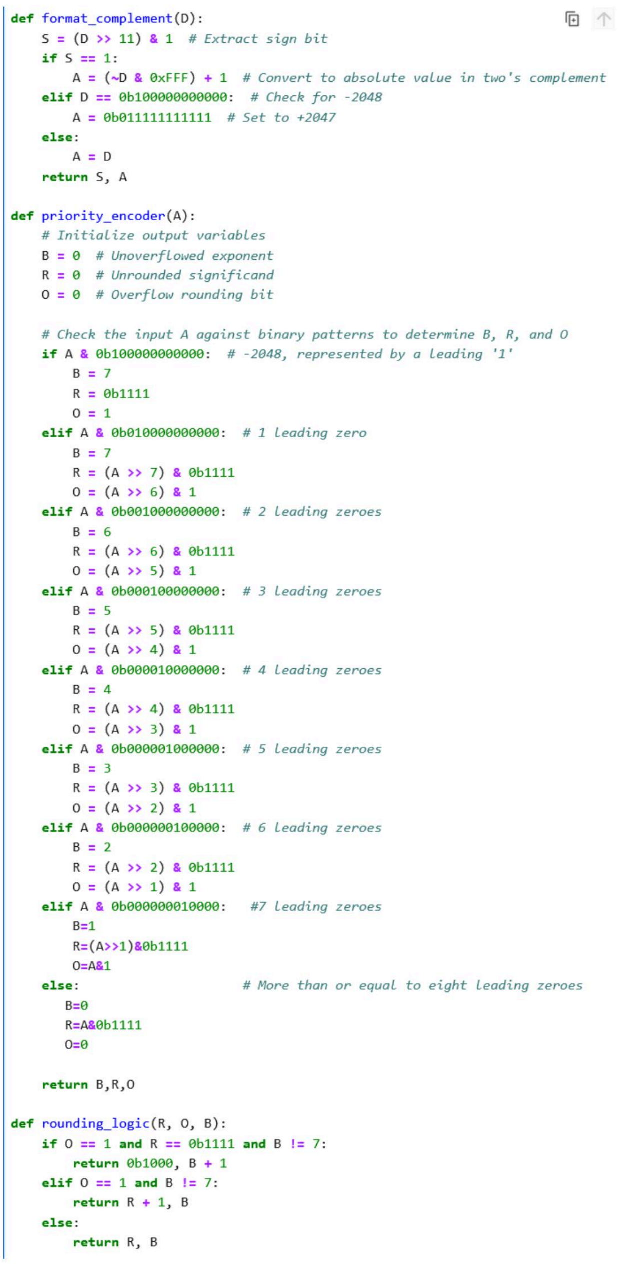
```
module fpcvt(D,S,E,F);
    input [11:0] D; //input 12 bit
    output S; //final sign bit
    output [2:0] E; //final exponent
    output [3:0] F; //final
    wire [3:0] R;
    wire [2:0] B;
    wire O;
    wire [11:0] A;

    format_complement complement(
        .D(D),
        .S(S),
        .A(A)
    );

    priority_encoder encoder (
        .A(A),
        .B(B),
        .R(R),
        .O(O)
    );

    rounding_logic rounder (
        .R(R),
        .F(F),
        .O(O),
        .B(B),
        .E(E)
    );

endmodule
```

Question assigned to the following page: 3

3. Simulation Documentation

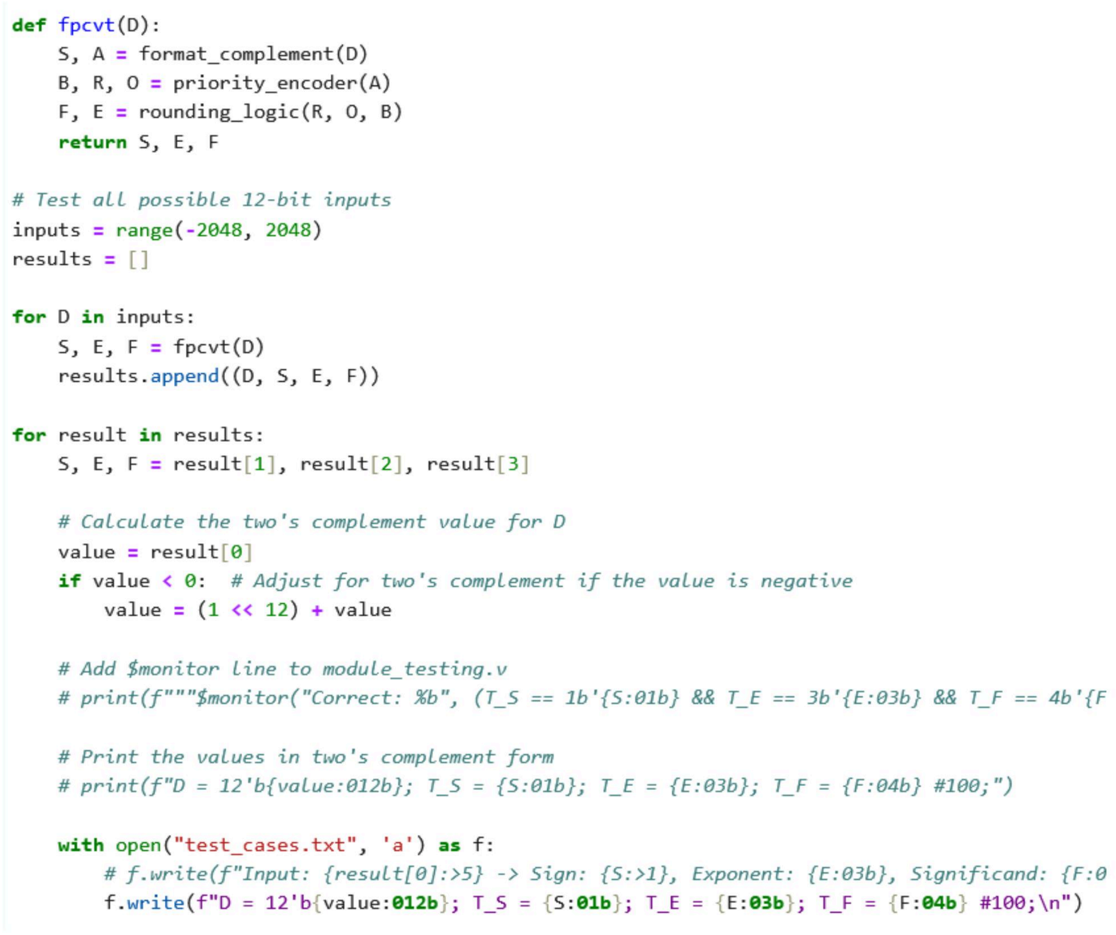
For simulation as we wrote the 4 modules we first individually tested each module using test cases based upon the lab 2 specification examples and hand calculated examples. Once format_complement, priority_encoder, and rounding_logic modules all seemed to work according to our test cases, we wrote test cases for the fpcvt module similarly to the previous 3, and validated using the examples given in the specification and the handout. These seemed to work, so before turning it in we decided to generate test cases for every single possible input value D using a python script.

Question assigned to the following page: 3



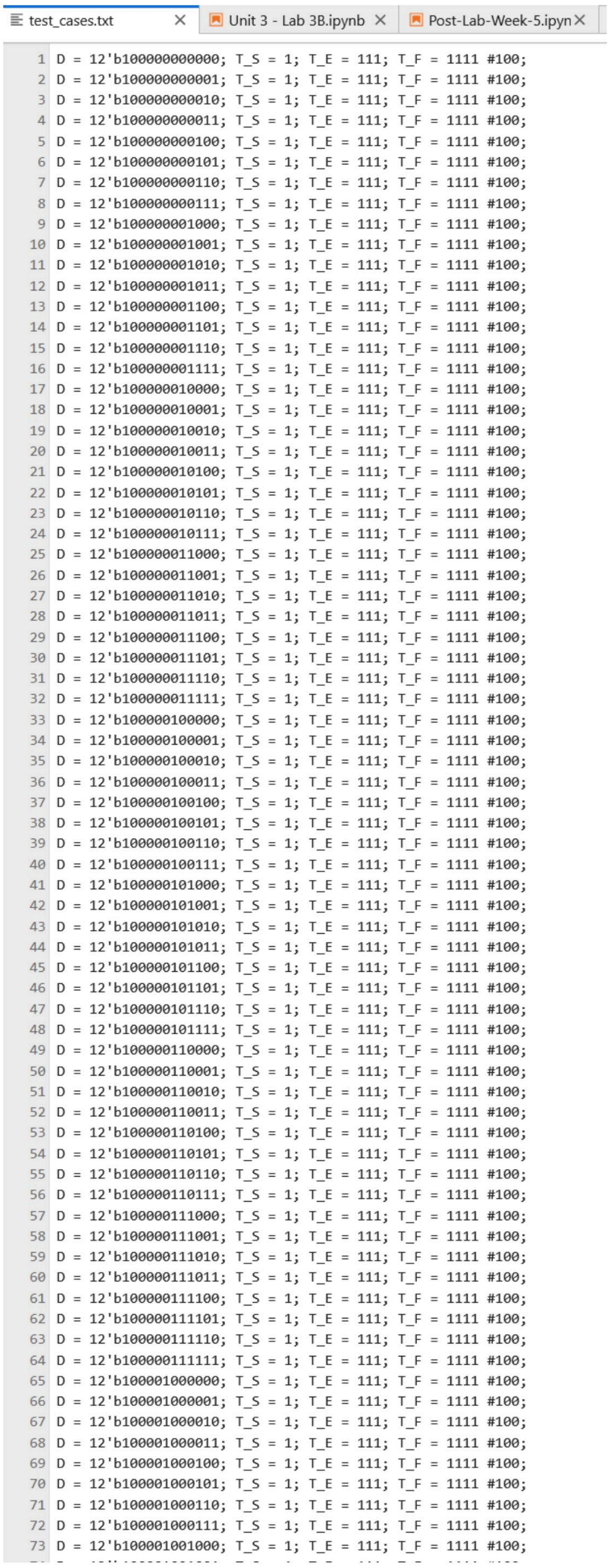
```python
def format_complement(D):
    S = (D >> 11) & 1  # Extract sign bit
    if S == 1:
        A = (~D & 0xFFF) + 1  # Convert to absolute value in two's complement
    elif D == 0b100000000000:  # Check for -2048
        A = 0b011111111111  # Set to +2047
    else:
        A = D
    return S, A

def priority_encoder(A):
    # Initialize output variables
    B = 0  # Unoverflowed exponent
    R = 0  # Unrounded significand
    O = 0  # Overflow rounding bit

    # Check the input A against binary patterns to determine B, R, and O
    if A & 0b100000000000:  # -2048, represented by a leading '1'
        B = 7
        R = 0b1111
        O = 1
    elif A & 0b010000000000:  # 1 leading zero
        B = 7
        R = (A >> 7) & 0b1111
        O = (A >> 6) & 1
    elif A & 0b001000000000:  # 2 leading zeroes
        B = 6
        R = (A >> 6) & 0b1111
        O = (A >> 5) & 1
    elif A & 0b000100000000:  # 3 leading zeroes
        B = 5
        R = (A >> 5) & 0b1111
        O = (A >> 4) & 1
    elif A & 0b000010000000:  # 4 leading zeroes
        B = 4
        R = (A >> 4) & 0b1111
        O = (A >> 3) & 1
    elif A & 0b000001000000:  # 5 leading zeroes
        B = 3
        R = (A >> 3) & 0b1111
        O = (A >> 2) & 1
    elif A & 0b000000100000:  # 6 leading zeroes
        B = 2
        R = (A >> 2) & 0b1111
        O = (A >> 1) & 1
    elif A & 0b000000010000:   #7 leading zeroes
        B=1
        R=(A>>1)&0b1111
        O=A&1
    else:                      # More than or equal to eight leading zeroes
       B=0
       R=A&0b1111
       O=0

    return B,R,O

def rounding_logic(R, O, B):
    if O == 1 and R == 0b1111 and B != 7:
        return 0b1000, B + 1
    elif O == 1 and B != 7:
        return R + 1, B
    else:
        return R, B
```

Question assigned to the following page: 3



```python
def fpcvt(D):
    S, A = format_complement(D)
    B, R, O = priority_encoder(A)
    F, E = rounding_logic(R, O, B)
    return S, E, F

# Test all possible 12-bit inputs
inputs = range(-2048, 2048)
results = []

for D in inputs:
    S, E, F = fpcvt(D)
    results.append((D, S, E, F))

for result in results:
    S, E, F = result[1], result[2], result[3]

    # Calculate the two's complement value for D
    value = result[0]
    if value < 0:  # Adjust for two's complement if the value is negative
        value = (1 << 12) + value

    # Add $monitor line to module_testing.v
    # print(f"""$monitor("Correct: %b", (T_S == 1b'{S:01b} && T_E == 3b'{E:03b} && T_F == 4b'{F

    # Print the values in two's complement form
    # print(f"D = 12'b{value:012b}; T_S = {S:01b}; T_E = {E:03b}; T_F = {F:04b} #100;")

    with open("test_cases.txt", 'a') as f:
        # f.write(f"Input: {result[0]:>5} -> Sign: {S:>1}, Exponent: {E:03b}, Significand: {F:0
        f.write(f"D = 12'b{value:012b}; T_S = {S:01b}; T_E = {E:03b}; T_F = {F:04b} #100;\n")
```

In this script we re-implemented floating point conversion, created an array with each possible value 12-bit number D, then calculated the resulting values of S, E, and F for each D. This script outputs a file test_cases.txt defining one value of D and result S,E,F numbers per line, which we were able to copy paste into our testbench module_testing.v.

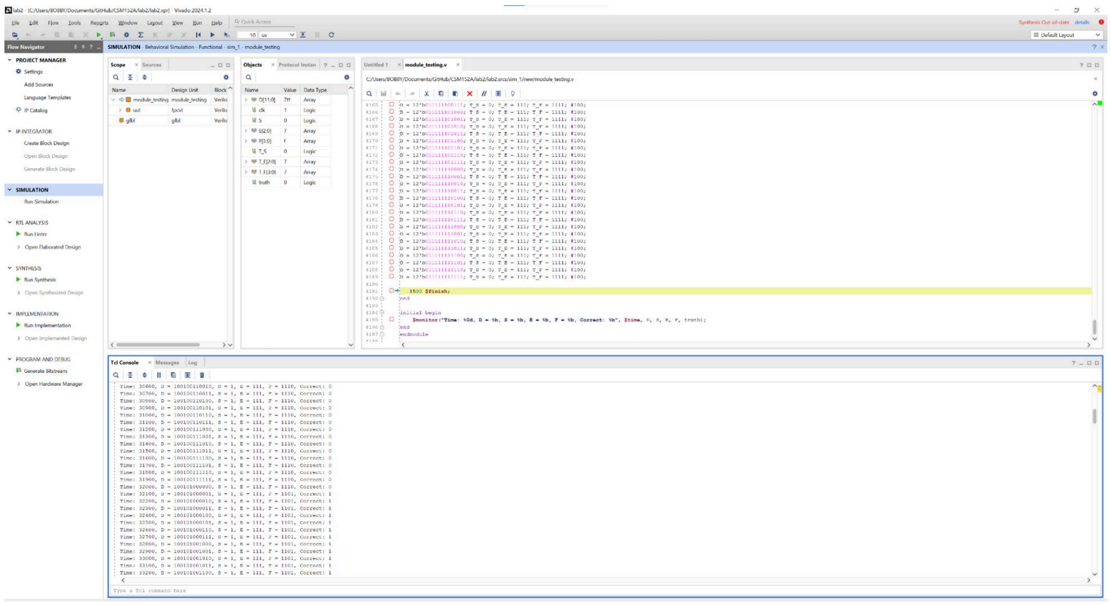


```
D = 12'b100000000000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000000111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000001111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000010111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000011111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000100111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000101111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000110111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100000111111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000000; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000001; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000010; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000011; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000100; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000101; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000110; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001000111; T_S = 1; T_E = 111; T_F = 1111 #100;
D = 12'b100001001000; T_S = 1; T_E = 111; T_F = 1111 #100;
```

Question assigned to the following page: 3

After some manual editing to modify syntax we were able to test our fpcvt code, using $monitor to automatically display if the results were correct or not. Unfortunately both our fpcvt module in Verilog and python code rounding logic had errors, so the resulting correctness values in the simulation were wrong, giving us false confidence in our code, and we submitted code with errors the first time around. After some debugging, we found that the code had an off by one error in the rounding_logic module, leaving us with a final significand that had the LSB of the significand incorrectly incremented by 1. The next class period we rewrote rounding_logic and that fixed the error, leaving us with a fully functional fpcvt module.

Simulation run example:



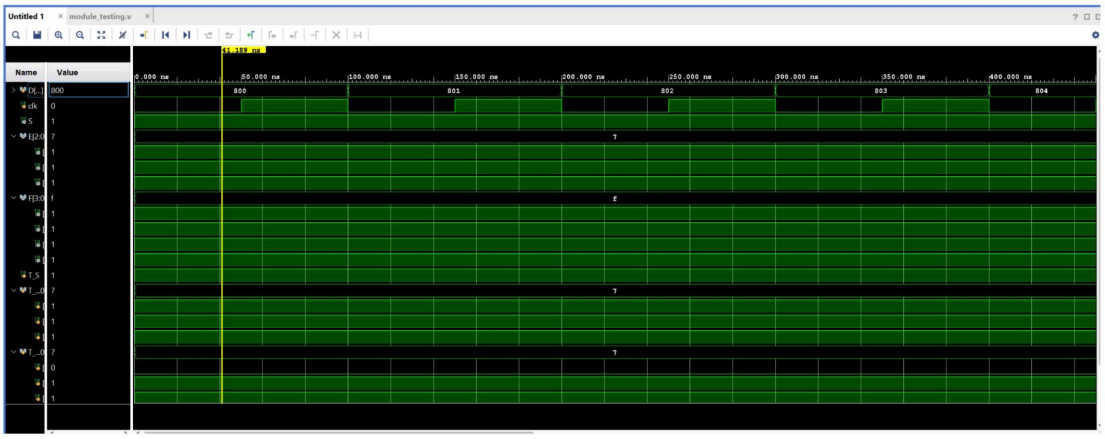
C:/Users/BOBBY/Documents/GitHub/CSM152A/lab2/lab2.srcs/sim_1/new/module_testing.v

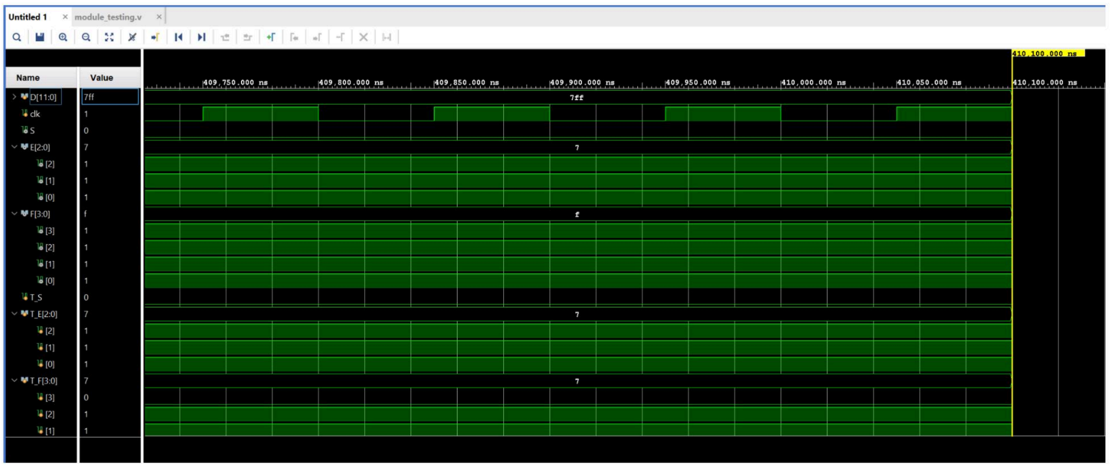
```
D = 12'b011111100111; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101000; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101001; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101010; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101011; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101100; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101101; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101110; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111101111; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110000; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110001; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110010; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110011; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110100; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110101; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110110; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111110111; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111000; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111001; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111010; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111011; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111100; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111101; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111110; T_S = 0; T_E = 111; T_F = 1111; #100;
D = 12'b011111111111; T_S = 0; T_E = 111; T_F = 1111; #100;

  #500 $finish;
end

initial begin
    $monitor("Time: %0d, D = %b, S = %b, E = %b, F = %b, Correct: %b", $time, D, S, E, F, truth);
end
endmodule
```

Tcl Console

```
Time: 30600, D = 100100110010, S = 1, E = 111, F = 1110, Correct: 0
Time: 30700, D = 100100110011, S = 1, E = 111, F = 1110, Correct: 0
Time: 30800, D = 100100110100, S = 1, E = 111, F = 1110, Correct: 0
Time: 30900, D = 100100110101, S = 1, E = 111, F = 1110, Correct: 0
Time: 31000, D = 100100110110, S = 1, E = 111, F = 1110, Correct: 0
Time: 31100, D = 100100110111, S = 1, E = 111, F = 1110, Correct: 0
Time: 31200, D = 100100111000, S = 1, E = 111, F = 1110, Correct: 0
Time: 31300, D = 100100111001, S = 1, E = 111, F = 1110, Correct: 0
Time: 31400, D = 100100111010, S = 1, E = 111, F = 1110, Correct: 0
Time: 31500, D = 100100111011, S = 1, E = 111, F = 1110, Correct: 0
Time: 31600, D = 100100111100, S = 1, E = 111, F = 1110, Correct: 0
Time: 31700, D = 100100111101, S = 1, E = 111, F = 1110, Correct: 0
Time: 31800, D = 100100111110, S = 1, E = 111, F = 1110, Correct: 0
Time: 31900, D = 100100111111, S = 1, E = 111, F = 1110, Correct: 0
Time: 32000, D = 100101000000, S = 1, E = 111, F = 1110, Correct: 0
Time: 32100, D = 100101000001, S = 1, E = 111, F = 1101, Correct: 1
Time: 32200, D = 100101000010, S = 1, E = 111, F = 1101, Correct: 1
Time: 32300, D = 100101000011, S = 1, E = 111, F = 1101, Correct: 1
Time: 32400, D = 100101000100, S = 1, E = 111, F = 1101, Correct: 1
Time: 32500, D = 100101000101, S = 1, E = 111, F = 1101, Correct: 1
Time: 32600, D = 100101000110, S = 1, E = 111, F = 1101, Correct: 1
Time: 32700, D = 100101000111, S = 1, E = 111, F = 1101, Correct: 1
Time: 32800, D = 100101001000, S = 1, E = 111, F = 1101, Correct: 1
Time: 32900, D = 100101001001, S = 1, E = 111, F = 1101, Correct: 1
Time: 33000, D = 100101001010, S = 1, E = 111, F = 1101, Correct: 1
Time: 33100, D = 100101001011, S = 1, E = 111, F = 1101, Correct: 1
Time: 33200, D = 100101001100, S = 1, E = 111, F = 1101, Correct: 1
```

Waveforms:





4. Conclusion

To summarize, we converted the 12-bit two's complement number into a floating point representation utilizing Verilog in Vivaldo. Unfortunately, our logic for testing these cases was not 100% correct as we needed to verify them visually by comparing the actual output vs the expected output. Upon further testing, we found a bug in the rounding module, so we rewrote the section. Once rewritten, we passed all the test cases!