Lab 2 Report

UCLA Samueli School of Engineering
CS M152A Spring ‘24
Lab 5 TA Chenda Duan

Group 6
Tae Hwan Kim 506043010
Brad Lowe 905581956

1 Introduction and requirement

It is important to handle floating-point for many digital applications and systems. This lab project introduces how to develop a module that converts 12-bit binary numbers into an 8-bit floating-point representation. This representation includes a 1-bit sign to show if numbers are positive or negative, a 3-bit exponent to adjust the range, and a 4-bit significand to keep the numbers precise. We focused on breaking down the 12-bit input into these key parts using three separate logic blocks to precisely adjust the size and precision based on the input value. Xilinx Vivado software and Verilog HDL are used to design these conversion modules, focusing on improving performance and using fewer resources. This project was conducted through simulations only to test and refine the modules in a controlled setting.

2 Design Requirements

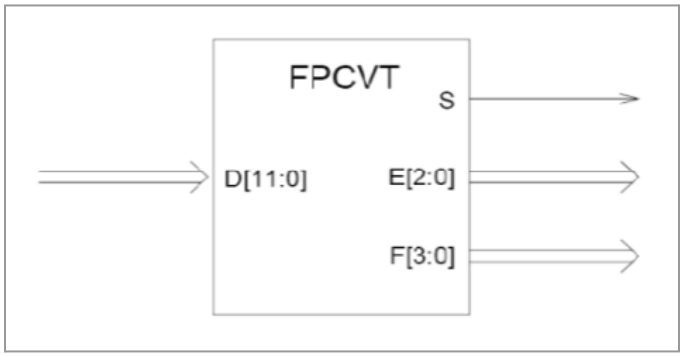

Figure 2.1: FPCVT module

The input data D[11:0] is represented in two's complement format, where D0 is the least significant bit (LSB) and D11 is the most significant bit (MSB). As the output in the floating-point representation, S represents the sign bit. The exponent is a 3-bit value E[2:0], and the significand comprises 4 bits, F[3:0].

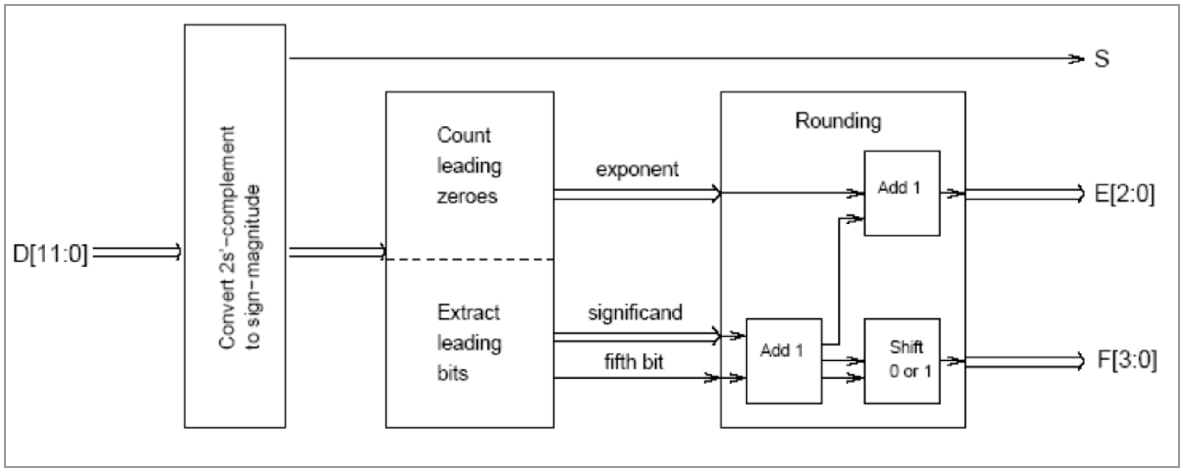

Figure 2.2: Block diagram for the floating-point conversion circuit

The FPCVT module contains three separate blocks: a converter of two's complement to sign-magnitude, a primary encoder, and a rounding module.

2.1 Two's Complement to Sign-magnitude Block

The first module, called the converter of 2's complement to sign-magnitude, takes in a 12-bit input and outputs the sign bit along with the 12 bits converted to sign-magnitude. If the sign bit is 0, it outputs the same value as the input. If the sign bit is 1, it outputs the absolute value of the negative number.

2.2 Primary Encoding Block

The second module, called the primary encoder, takes in a 12-bit sign-magnitude input and outputs a 3-bit exponent, a 4-bit significand, and a 1-bit fifth bit. The exponent is a factor of floating-point representation, so the fewer the number of leading zeros in the value, the higher the exponent. Additionally, the significand is the 4 bits immediately following the end of the leading zeros. If there are more than 8 leading zeros, meaning no 4 bits remain right after the leading zeros, the significand should be the last 4 bits of the value. The fifth bit is the bit following the 4-bit significand. If there are more than 7 leading zeros and thus no fifth bit, it should output 0.

2.3 Rounding Block

For the rounding module, it checks whether the fifth bit is 1 or 0. If the bit is 1, it adds 1 to the 4-bit significand. If the bit is 0, the 4-bit significand remains unchanged. In the case of 4-bit significand overflow, it sets the significand to 0 and adds 1 to the exponent. There is another special case when the exponent is 111 and the significand is also 1111, where two overflows happen. In this case, the value should not be changed, keeping the same value as is.

3 Design and Implementation

3.1 Design Two's Complement to Sign-magnitude Module

```
23      module twos_complement_to_sign_mag(
24          input wire [11:0] d,
25          output wire [11:0] m,
26          output wire sign
27      );
28
29          assign sign = d[11];
30          assign m = sign ? (~d + 1'b1) : d;
31
32      endmodule
```

Figure 3.1: Verilog for two's complement to sign-magnitude converter module

We have implemented the first module, the two's complement to sign-magnitude converter, primarily using if statements and a NOT operation followed by adding 1 if the sign bit is 1, which indicates a negative number. If the sign bit is 0, it returns the same value as the input.

3.2 Design Primary Encoder Module

```
`timescale 1ns / 1ps

module primary_encoder(
    input wire [11:0] m,
    output reg [2:0] exp,
    output reg [3:0] significand,
    output reg fifthbit
);

    always @(*) begin
        exp = 3'b000;
        significand = 4'b0000;
        fifthbit = 0;

        if (m[10]) begin
            exp = 3'b111;
            significand = m[10:7];
            fifthbit = m[6];
        end else if (m[9]) begin
            exp = 3'b110;
            significand = m[9:6];
            fifthbit = m[5];
        end else if (m[8]) begin
            exp = 3'b101;
            significand = m[8:5];
            fifthbit = m[4];
        end else if (m[7]) begin
            exp = 3'b100;
            significand = m[7:4];
            fifthbit = m[3];
        end else if (m[6]) begin
            exp = 3'b011;
            significand = m[6:3];
            fifthbit = m[2];
        end else if (m[5]) begin
            exp = 3'b010;
            significand = m[5:2];
            fifthbit = m[1];
        end else if (m[4]) begin
            exp = 3'b001;
            significand = m[4:1];
            fifthbit = m[0];
        end else begin
            exp = 3'b000;
            significand = m[3:0];
            fifthbit = 0;
        end
    end
endmodule
```

Figure 3.2: Verilog for primary encoder module

For the second module, the primary encoder, we calculated the exponent by using if statements to detect the first non-zero bit in the 12-bit input. Once a non-zero bit is found, the exponent is determined based on the number of leading zeros, according to the table provided in the lab 2 specification. Concurrently, the 4-bit significand is extracted starting from this non-zero bit and extends through the next four bits. The fifth bit, following the significand, is determined using the same logic. In cases where there are so many leading zeros that neither a complete 4-bit significand nor a fifth bit are present, we added code to accurately handle and validate these scenarios.

3.3 Design Rounding Module

```
23  module rounding(
24      input wire [2:0] exp,
25      input wire [3:0] significand,
26      input wire fifthbit,
27      output reg [2:0] E,
28      output reg [3:0] F
29  );
30      always @(*) begin
31          E = exp;
32          F = significand;
33
34          if (fifthbit) begin
35              if (significand == 4'b1111 && exp != 3'b111) begin
36                  F = 4'b0000;
37                  E[2:0] = exp[2:0] + 1'b1;
38              end else if (significand == 4'b1111 && exp == 3'b111) begin
39                  F = significand;
40                  E[2:0] = exp[2:0];
41              end else begin
42                  F = significand + 1;
43                  E[2:0] = exp[2:0];
44              end
45          end else begin
46              F = significand;
47              E[2:0] = exp[2:0];
48          end
49      end
50  endmodule
51
```

Figure 3.3: Verilog for rounding module

For the third module, rounding, we implemented logic using an if statement to assess the fifth bit. We add 1 to the significand if the fifth bit is 1; otherwise, the significand remains unchanged. We also address the special case of significand overflow by setting the significand to 0 and incrementing the exponent by 1. Another exceptional situation occurs when both the exponent is 111 and the significand is 1111, indicating dual overflows. In this case, we have programmed the system not to increment any values, thus maintaining the original value.

3.4 Design FPCVT Module

```
module FPCVT(D, S, E, F
    );

    input [11:0] D;
    wire [11:0] m;
    wire [2:0] exp;
    wire [3:0] significand;
    output wire S;
    output wire [2:0] E;
    output wire [3:0] F;

    twos_complement_to_sign_mag tcsm(
        .d(D),
        .m(m),
        .sign(S)
    );

    primary_encoder pe(
        .m(m),
        .exp(exp),
        .significand(significand),
        .fifthbit(fifthbit)
    );

    rounding rd(
        .exp(exp),
        .significand(significand),
        .fifthbit(fifthbit),
        .E(E),
        .F(F)
    );
endmodule
```

Figure 3.4: Verilog for FPCVT module

Combining all three separate modules above into the FPCVT module, it takes a 12-bit binary input and outputs a 1-bit sign, a 3-bit exponent, and a 4-bit significand after functioning as specified in the lab specifications. We can easily overview all inputs and outputs to ensure they are correctly designed for the separate blocks. By separating all designs into separate modules and integrating them into one module, we can simulate and test each function more efficiently and clearly during the testing period.

4 Simulation and Testing

For this lab, we used a similar testbench approach to what was used in Lab 1, using mocktb.v as a base. We made sure to test the output of each of the different modules, as well as the final output of the FPCVT for each test.

```
reg [11:0] Dfp;
wire Sfp;
wire [2:0] Efp;
wire [3:0] Ffp;

FPCVT fp(
   .D(Dfp),
   .S(Sfp),
   .E(Efp),
   .F(Ffp)
);

initial begin
    d = 12'b100000001010;
    Dfp = 12'b100000001010;
    #10;
    $display("<Sign-Mag> Input d: %b, Output m: %b, sign: %b", d, m, sign);
    #10;
    $display("<Primary-Encode> Exponents: %b, Significand: %b, Fifth-bit: %b", exp, significand, fifthbit);
    #10;
    $display("<Rounding> S: %b, E: %b, F: %b", sign, E, F);
    #10;
    $display("<FPCVT Output> S: %b, E: %b, F: %b", Sfp, Efp, Ffp);
    #10;

    $display("------------");

    d = 12'b000010110010;
    Dfp = 12'b000010110010;
     #10;
     $display("<Sign-Mag> Input d: %b, Output m: %b, sign: %b", d, m, sign);
     #10;
     $display("<Primary-Encode> Exponents: %b, Significand: %b, Fifth-bit: %b", exp, significand, fifthbit);
     #10;
     $display("<Rounding> S: %b, E: %b, F: %b", sign, E, F);
     #10;
     $display("<FPCVT Output> S: %b, E: %b, F: %b", Sfp, Efp, Ffp);
     #10;
```

Figure 4.1: Verilog for simulation

The above image shows the majority of our testbench code, with more tests being run below this section, and the individual modules being defined earlier in the code. We used tests meant to capture multiple edge cases so we could see how each individual module would deal with most inputs we anticipate being tested. For each test, we define d as the input to the individual modules starting with sign-magnitude and Dfp as the input to the FPCVT module, and print out the outputs of each module after they finish running.

```
<Sign-Mag> Input d: 000010110010, Output m: 000010110010, sign: 0
<Primary-Encode> Exponents: 100, Significand: 1011, Fifth-bit: 0
<Rounding> S: 0, E: 100, F: 1011
<FPCVT Output> S: 0, E: 100, F: 1011
------------
<Sign-Mag> Input d: 101000010100, Output m: 010111101100, sign: 1
<Primary-Encode> Exponents: 111, Significand: 1011, Fifth-bit: 1
<Rounding> S: 1, E: 111, F: 1100
<FPCVT Output> S: 1, E: 111, F: 1100
------------
<Sign-Mag> Input d: 110000000101, Output m: 001111111011, sign: 1
<Primary-Encode> Exponents: 110, Significand: 1111, Fifth-bit: 1
<Rounding> S: 1, E: 111, F: 0000
<FPCVT Output> S: 1, E: 111, F: 0000
```

Figure 4.2: Console logs for testing modules

The above logs are the output of our testbench. The inputs are 000010110010, 101000010100, and 110000000101, which were chosen to test a range of different cases for our modules. These vary in exponents, sign bits, significand, and fifth bit, meaning the inputs are different enough to test a wide range of inputs. We used this testbench to debug our modules, but did not find any significant bugs in any modules other than the primary encoder on our second day of work. We also tested other inputs, although those are not shown in the final testbench as we switched to new tests later on.

```
<Sign-Mag> Input d: 011111111111, Output m: 011111111111, sign: 0
<Primary-Encode> Exponents: 111, Significand: 1111, Fifth-bit: 1
<Rounding> S: 0, E: 111, F: 1111
<FPCVT Output> S: 0, E: 111, F: 1111
------------
```

Figure 4.3: Console logs for the exceptional case that results in two overflows

In particular, we wanted to test the significand overflow + exponent overflow case, which has different behavior than the rest of the test cases. In this case, the magnitude of the input is large enough that all of the bits of the significand, exponent, and fifth bit are 1. Due to the rounding logic, the fifth bit being 1 means we add 1 to the significand, causing it to overflow, which also causes the exponent to overflow. This would end up with the final output being a significand and exponent of 0 for a very large number. To remedy this, we hard code a special case for large enough numbers such that this overflow does not occur, and the output is 111 and 1111 for the exponent and significand, respectively. The testbench helped us catch a few issues with this case, as we originally did not hard code it in and saw that the output was all zeroes, which seemed incorrect. After consulting the spec again, we were able to fix this bug.

5 Conclusion

5.1 Design Summary

In summary, our implementation of the floating point converter uses 3 sub-modules. The first is a two's complement to sign-magnitude block which takes in a two's complement input and outputs a sign bit and magnitude. This block is made up of a simple ternary statement. The second is a primary encoder block which takes in the magnitude and outputs an exponent, significand, and fifth bit. It is made up of sequential if statements to detect the number of leading zeroes and continues from there. The last is a rounding block which takes the outputs of the primary encoder as input and outputs the final values of the exponent and significand. This block checks the fifth bit and adds it to the significand, then updates the exponent if there is a significand overflow, while checking to make sure the number is not so big that the exponent will overflow as well.

5.2 Difficulties

During Lab 2, there were only a few difficulties we ran into. First, we had some trouble knowing where to start, mostly as a result of still being new to coding in Verilog and using the concept of modules rather than functions. This leaked through into some issues with figuring out how to run the testbench, but these were quickly resolved by checking in with Chenda.

Partway into working on the second module, the primary encoding block, we ran into an error when testing multiple different inputs in a row through our testbench. We noticed that the first run always had the correct output, but the rest were incorrect. We theorized that this was a problem with our looping logic, but nothing we tried would fix the issue. After consulting Chenda, we realized that we were incorrectly running our code in an initial block rather than an always block, and did not properly reset the registers between runs. After changing our code to an always block and resetting each register to the correct initial value, the problem cleared up. Other than these, there were no major issues with Lab 2.

5.3 Suggestions

We felt that this lab was mostly well-implemented, and was overall not too difficult. After reading the lab spec for the first time we were still unable to figure out exactly where to start with our code, but it did have detailed instructions and background info that made each module relatively painless to complete. Also, Chenda was very helpful for our group and answered any questions we had about the lab, meaning we did not have any difficulties that lasted for a long time, and we were able to finish the coding section earlier than expected.