Lab 1 Report: Floating Point Conversion

This report is composed by

Aaron Yoo, 004745379
Chayanis Techalertumpai, 404798097
Christian Rodriguez, 804789345

Introduction and Requirements

In Lab 1, we had to convert a 12-bit linear encoding of an analog signal into a compounded 8-bit Floating Point Representation. For the lab, we merely used a simulation to test our design and we did not use the FPGA board. The design for converting the 12-bit linear encoding to an 8-bit floating point representation was broken up into three modules.

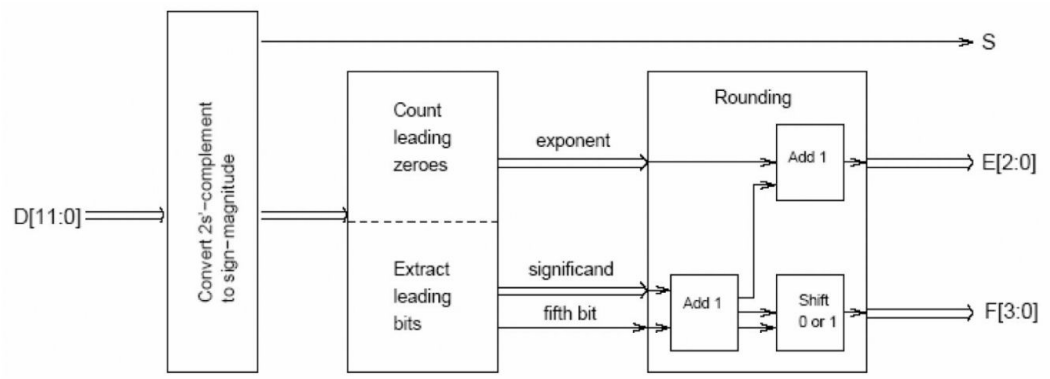
The first block converts the 12-bit two's complement input to a sign-magnitude representation. We had to be careful of certain cases: nonnegative numbers are unchanged while negative numbers are replaced by their absolute value and the most negative number (-2048) should be replaced with -1024 in signed-magnitude representation because if we handled this case normally, it would be replaced with -0.

The second block is responsible for converting sign-magnitude to floating point conversion. We had to keep track of the sign bit, exponent, and significand. Specifically for the exponent bit, we followed the rule: exponent = 8 - i, where i is the number of leading zeroes.

The third block is responsible for rounding our floating point representation that we obtained from the second block. We had to closely monitor the fifth bit following the last leading 0. If this fifth bit is a 1, then we round our result up, and if it is a 0, then we round down. We also handled the special case of the significand overflowing by adding one to it, shifting it to the right once and increasing the exponent bit by 1.

Design description

The floating-point conversion circuit takes in a 12-bit binary string as input and outputs the floating point representation of that number in three components: sign (S), exponent (E), and mantissa (F). The circuit is broken down into three sequential modules. Our modules are named M1, M2 and M3. Below is the schematic diagram for our system.



M1: 2's-complement to Sign-Magnitude Converter

Our first module takes in a 12-bit binary string and outputs a 12-bit signed-magnitude value and a 1-bit sign of the input string. The module first checks for the sign of the input and returns the original value for a nonnegative input or the absolute value for a negative input. One exception is when the input is the largest negative number (-2048), it should be replaced with -1024 in signed magnitude representation.

```
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21 module M1(
22    D,  // two's complement input
23    Z,  // signed magnitude
24    S
25     );
26
27 input [11:0] D;
28 output reg [11:0] Z;
29 output reg S;
30
31 always @*
32 if (D[11])
33 begin
34    if (D == 'b100000000000)
35    begin
36       // special case
37       S = 1'b1;
38       Z = 'b111111111111;
39    end
40    else
41    begin
42       // X is negative
43       S = D[11];
44       Z = (~D[11:0] + 1'b1);
45    end
46 end
47 else
48 begin
49    // D is positive
50    S = D[11];
51    Z = D[11:0];
52 end
53
54 endmodule
55
```

M2: Sign-Magnitude to Float Converter

Our second module takes in a 12-bit signed magnitude value and returns a 3-bit exponent, a 4-bit significand, and a single fifth bit. For the exponent, we were given a handy table to follow so we came up with a formula: exponent = 8 - leading zeroes. We counted the leading zeroes by getting the position of the last non-zero number and subtracting 1 to it. The formula for leading zeros is: 12 - position (of last non-zero number, right to left) - 1. In terms of edge cases, we take into account when there are no leading zeros in our input (111111111111), and we set the exponent to the highest values (111). Furthermore we handle another edge case when the exponent is 0 by making sure that our significand is the least significant 4 bits of the input. If we do not hit any of the edge cases, we simply store the 4 bits immediately following the last leading zero as our significand and the bit after that (fifth bit) as a register named T.

```
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module M2(
22     Z,  // 11 bit signed magnitude
23     E,  // exponent
24     M,  // mantissa
25     T   // fifth bit
26      );
27
28  input [11:0] Z;
29
30  output reg [2:0] E;
31  output reg [3:0] M;
32  output reg T;
33
34  reg [4:0] pos;  // position of the highest one
35  reg [4:0] leading_zeroes;
36  reg [11:0] TZ;  // temporary Z for shifting
37  integer i;
38
39  always@* begin
40     for (i=0; i <= 12; i=i+1) begin
41        if (Z[i]) pos = i;
42     end
43
44     leading_zeroes = 12 - pos - 1;
45     if (leading_zeroes <= 8)
46        E = 8 - leading_zeroes;
47     else
48        E = 0;
49
50     if (leading_zeroes == 0) begin
51        E = 3'b111;
52     end
53
54     if (E == 0) begin
55        // Edge case, exponent is zero
56        TZ = Z;
57        M = TZ[3:0];
58        T = 1'b0;
59     end
60     else begin
61        // Normal case
62        TZ = Z << leading_zeroes;
63        M = TZ[11:8];
64        T = TZ[7];
65     end
66  end
67
68
69  endmodule
70
```

M3: Rounding Machine

Our third module implements a simple rounding rule that depends solely on the fifth bit that we gathered from M2. This bit tells us whether to round up or down. If the bit is 0, then we round up, if the bit is 1, then we round up. There are a couple of edge cases we had to take care of. The first edge case is when exp = 111, significand = 1111, and when the fifth bit = 1 -- max overflow. If this happens, we simply use the biggest possible floating point number. Another edge case is when the fifth bit is a 1 and the significand = 1111. By following the spec, we handle this case by adding one to our exponent and making our significand 1000. If the significand is not 1111, then we simply add one to it and keep our exponent the same. If our fifth bit is 0, then there is no rounding done so we simply use the same exponent and mantissa from M2.

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module M3(
   exp,  // 3 bit exponent
   mant, // 4 bit mantissa
   fifth, // 1 bit input fifth
   E, // 3 bit output exponent
   F
);

input [2:0] exp;
input [3:0] mant;
input     fifth;

output reg [2:0] E;
output reg [3:0] F;

always @(*)
begin
   if (exp == 3'b111 && mant == 4'b1111 && fifth == 1'b1)
   begin
      // This is a max overflow so just use the biggest possible
      // floating point number.
      E = exp;
      F = mant;
   end
   else if (fifth == 1)
   begin
      if (mant == 4'b1111)  // We want to round up
      begin
         // Special case described in spec, shift 10000 right
         // and increase exponent by 1
             F = 4'b1000;
             E = exp[2:0] + 1'b1;
         // exp cannot overflow here or we would be in case 1
      end
      else
      begin
         // Just add one to the mantissa
         E = exp[2:0];
         F = mant[3:0] + 1'b1;
      end
   end
   else
   begin
      // No rounding
      E = exp[2:0];
      F = mant[3:0];
   end
end

endmodule
```

FPCVT: Overall High Level Design

Our high level module is called FPCTV. Here, we simply hook up M1, M2, and M3 together using wires. Our high level input is D[11:0], which is a 12-bit two's complement number and outputs a sign bit S, a 3-bit exponent E[2:0], and a 4-bit significand F[3:0]. We use wire w_z (12-bit signed-magnitude value) to connect M1 and M2 together, we use wire w_exp (3-bit exponent from M2), w_mant (4-bit mantissa/significand from M2), and w_fifth (fifth bit from M2) to connect M2 and M3 together. By connecting all of our modules together, we get our correct output!

```
// Company:
// Engineer:
//
// Create Date:    12:36:38 10/09/2019
// Design Name:
// Module Name:    FPCVT
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module FPCVT(
   D,  // Two's complement representation D0(LSB) D11(MSB)
   S,   // Sign bit of floating point representation
   E,  // 3-bit exponent
   F  // 4-bit mantissa
);

input [11:0] D;

output       S;
output [2:0] E;
output [3:0] F;

// M1 -- converts 2's complemented to signed magnitude
// M2 -- counts leading zeroes
// M3 -- rounding

wire [11:0] w_z;  // connects M1 to M2
wire [2:0] w_exp;    // connects M2 to M3
wire [3:0] w_mant;   // connects M2 to M3
wire w_fifth;     // connects M2 to M3

M1 A1(.D(D), .S(S), .Z(w_z));
M2 A2(.Z(w_z), .E(w_exp), .M(w_mant), .T(w_fifth));
M3 A3(.exp(w_exp), .mant(w_mant), .fifth(w_fifth), .E(E), .F(F));

endmodule
```

Simulation documentation

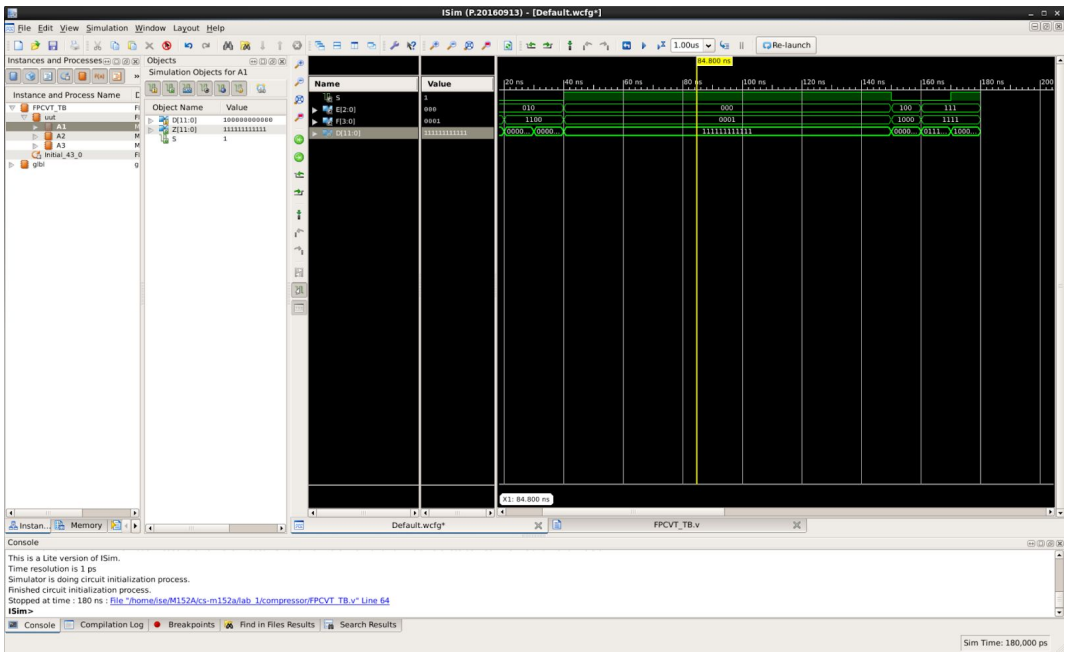
I/O Format

Input: D[11:0]

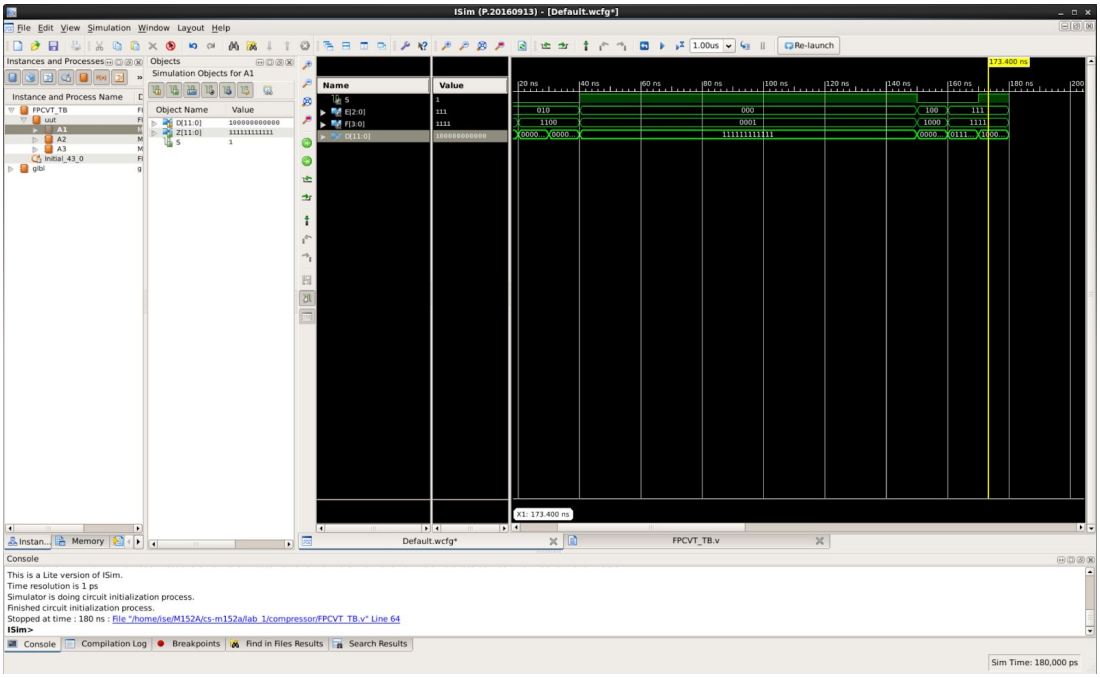
Output: S + E[2:0] + F[3:0]

Note: "+" means concatenate, not addition

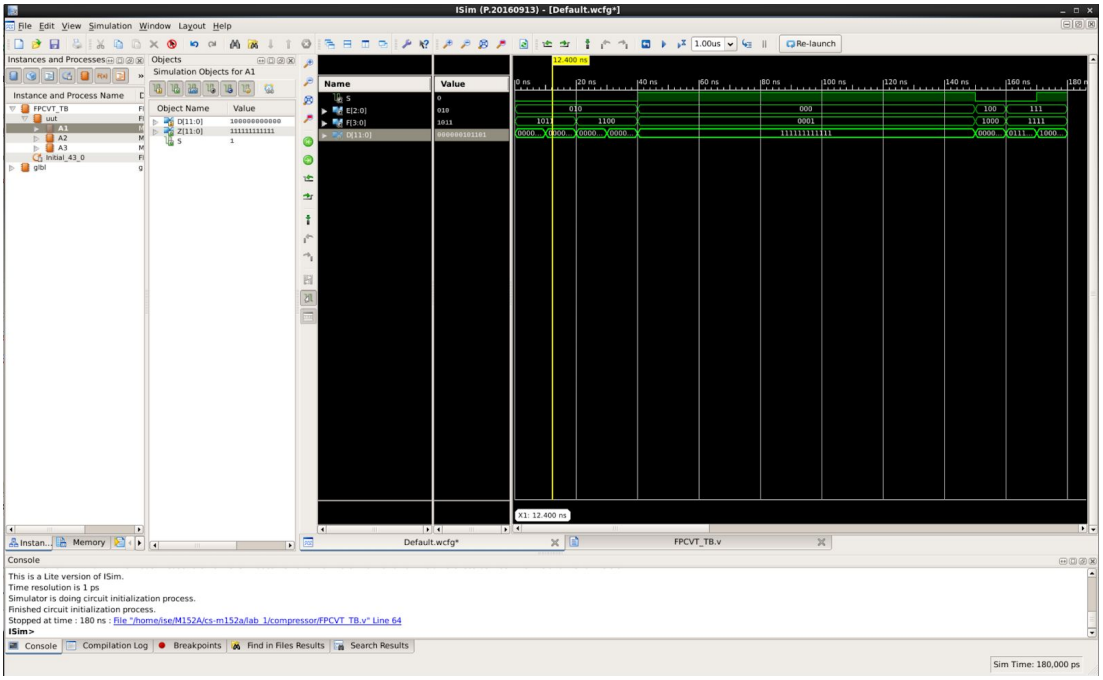
Test Case: Regular Two's Complement Number
Input: 111111111111
Output: 1 000 0001



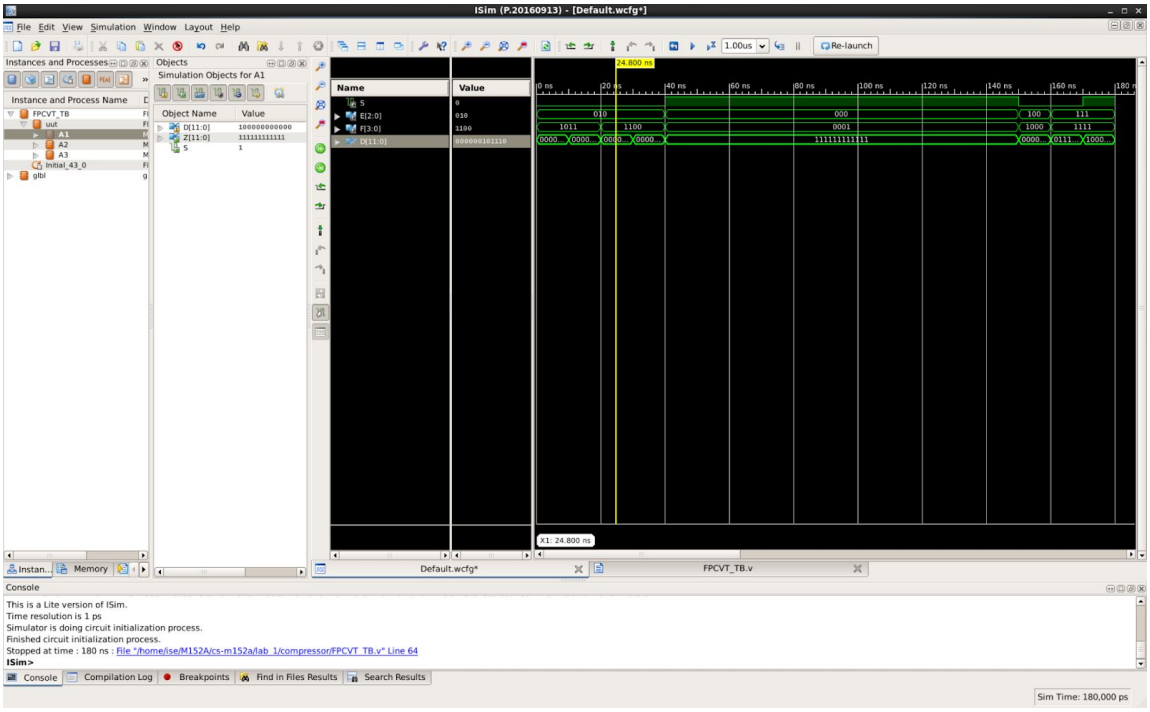
Test Case: Most Negative Two's Complement
Input: 100000000000
Output: 1 111 1111



Test Case: Round Down
Input: 000000101101
Output: 0 010 1011



Test Case: Round Up
Input: 000000101110
Output: 0 010 1100



Conclusion

Lab 1 consisted of converting a 12-bit linear encoding of an analog signal into a compounded 8-bit floating representation. We broke our implementation into 3 modules: M1, M2, and M3. M1 converts the 12-bit two's complement input to a sign-magnitude representation. M2 takes in the sign-magnitude representation and converts it into a floating point number. The last module, M3, takes in the floating point number and rounds it up or down.

We faced many difficulties regarding how Verilog worked. We were not familiar with the syntax of Verilog, such as assign statements, for loops, and concatenation procedures. We learned how to deal with these problems by searching through verilog documentation and online resources. With regards to problems during the implementation, we had difficulty in resolving edge cases in our code. There were a handful of max overflow problems that we had to consider and resolve by reading the spec carefully.

The lab spec was really thorough and provided great explanations on what we needed to do but missed some edge cases such as having 0 leading zeros in the sign-magnitude to floating point module, or how to handle max overflow in the rounding module of the design. Overall, the lab was still very friendly to new users of Verilog.