CS M152A

Lab 2 Report

Puvali Chatterjee
Lab Partner: Vivian Ha
TA: Samuel Lee

Introduction and Requirements

In this lab, we used the Xilinx ISE software to design and test a combinational circuit that converts a 12-bit linear encoding of an analog signal into a compounded 8-bit floating point representation, with no FPGA use. We named the floating point conversion module "FPCVT" as per the spec and added the following input and output ports:

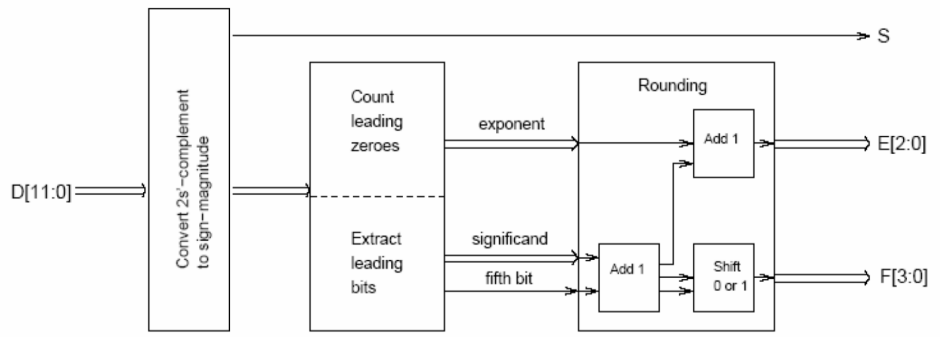
1. D[11:0]: This is the input data (12-bit linear encoding of analog signal) in two's complement representation. Here, D[11] is the most significant bit and D[0] is the least significant bit.
2. S: This is the sign bit of the floating point representation
3. E[2:0]: This is the 3-bit exponent of the floating point representation
4. F[3:0]: This is the 4-bit mantissa/significand of the floating point representation

FPCVT module

The simplified floating point representation of a 12-bit input linear encoding consists of a sign bit (S), 3-bit exponent (E) and 4-bit significand (F) and the value represented by a byte in this format is:

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

To map linear encodings to floating point representations, we built a combinational circuit that performs compression. Here, the preferred floating point representation of a linear encoding is one where the MSB of the significand is 1. To map values that do not have exact floating point representations, we perform rounding (up or down) to the nearest floating point encoding.



Block diagram of FPCVT circuit

The block diagram above shows the different blocks in the FPCVT circuit. The first block converts the 12-bit two's complement input data into sign-magnitude representation, where positive inputs (sign bit 0) remain unchanged and negative inputs (sign bit 1) are negated. The second block

performs basic linear to floating point conversion by counting the number of leading zeros and leading bits. The third block performs rounding to ensure the closest possible floating point representation of the input linear encoding.

Design Description

The main *FPCVT* module first assigns the most significant bit of D to S as the sign bit. We implemented three sub-modules to perform the functions needed to support the design:

1. module sign_mag(D, D_abs)

The *sign_mag* module takes in the 12-bit linear encoding in two's complement representation and returns its absolute value. For positive input data, D_abs is the same as D. For negative input data, we performed two's complement negation by first inverting all the bits of D and then adding 1 to get the absolute value of D.

```
module sign_mag(D, D_abs);

input [11:0] D;
output reg [11:0] D_abs;

always @* begin
   //if largest negative number, set magnitude to largest positive number
   if (D == -12'd2048)
      D_abs = 12'd2047;
   else if(D[11] == 1'b1)
      //2's complement inversion
      D_abs = ~D + 1;
   else
      D_abs = D;
end
endmodule
```

The *sign_mag* module also tackles the case of D being -2048 or [1000 0000 0000]. Since the largest possible 12-bit two's complement number is 2047, we manually set D_abs to 2047 when D is -2048. This is because complement-incrementing -2048 to get its absolute value will not work because the result will just be -2048.

2. module leading_0s_bits(D_abs, lz, raw_E, raw_F, rndg_bit)

The *leading_0s_bits* module first counts the number of leading zeros in D_abs as follows:

```
always @* begin

   i = 4'd11;
   while (D_abs[i] != 1) begin
      i = i - 1;
   end
   lz = 4'd11 - i;

   //set Exponent depending on # of leading zeroes
   case(lz)
      4'd1: raw_E = 3'd7;
      4'd2: raw_E = 3'd6;
      4'd3: raw_E = 3'd5;
      4'd4: raw_E = 3'd4;
      4'd5: raw_E = 3'd3;
      4'd6: raw_E = 3'd2;
      4'd7: raw_E = 3'd1;
      default: raw_E = 0;
   endcase
```

Then, we use a case statement to temporarily set the exponent (this may change later during rounding) as:

Leading Zeroes	Exponent
1	7
2	6
3	5
4	4
5	3
6	2
7	1
≥ 8	0

After counting the leading bits, we set the significand (temporarily, as it may also change during rounding) and rounding bit as follows:

- ➔ If there are 1-7 leading zeros, the significand is the 4 leading bits (after the leading zeros) and the rounding bit is the 5th leading bit
- ➔ If there are 8 leading zeros, the significand is the 4 leading bits and the rounding bit is 0
- ➔ If there are more than 8 leading zeros, the significand is the 4 least significant bits and the rounding bit is 0

3. module rounding(raw_E, raw_F, rndg_bit, E, F)

The rounding bit from the previous stage (*leading_0s_bits*) decides how rounding will be done: if the rounding bit is 0, truncation is performed, meaning that either the number stays the same (if the 4 bits in the significand are sufficient to represent it) or it is rounded down. If the rounding bit is 1, the number is rounded up. There are a few cases when rounding:

- Rounding down

  Here, the exponent and significand determined in the previous block (*leading_0s_bits*) remain unchanged.

- Rounding up, overflow, E < 7

  When rounding up, the significand is incremented by 1. However, if the significand is 1111, there is an overflow when trying to add 1. Our solution for this situation was to check for overflow *before* adding 1 to the significand as follows:

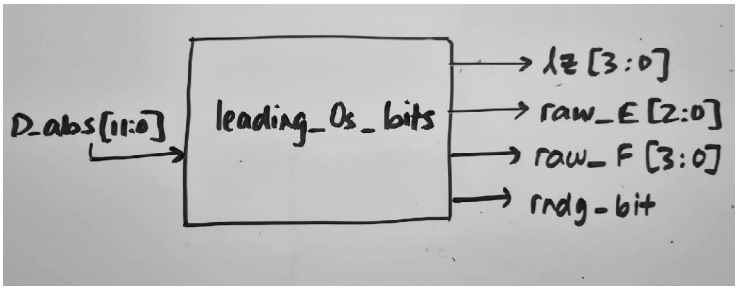
```
//case 2: overflow when rounding up
if(raw_F[0] & raw_F[1] & raw_F[2] & raw_F[3]) begin    //when raw_F = 1111

    //case 2a: if E < 7 --> increment E
    if(raw_E < 3'd7) begin
        F = 4'b1000;
        E = raw_E + 1;
    end
```

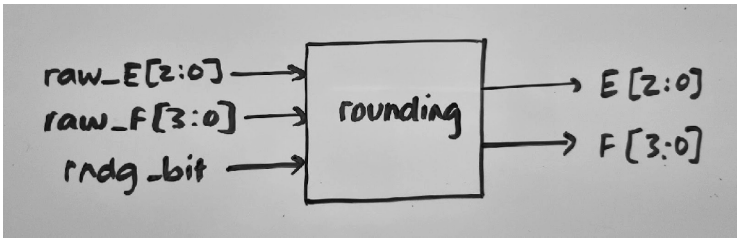
  Instead of adding 1 to the significand, we manually set the significand to 1000 (which is the same as right-shifting the overflowed sum 10000 by 1 bit) and increment the exponent by 1.

- Rounding up, overflow, E = 7
  E is a 3-bit output so the maximum value it can have is [111] or 7. This becomes a problem when there is overflow when rounding up because the exponent cannot be incremented further. In this case, we set the significand and exponent to that of the largest floating point representation possible, namely [S 111 1111]. The value of this representation is $(-1)^S \times 15 \times 2^7$ or ± 1920.

- Rounding up, no overflow
  If there is no overflow when rounding up, the significand can simply be incremented by 1 while the exponent remains unchanged.

The top-level module *FPCVT* instantiates the *sign_mag*, *leading_0s_bits* and *rounding* modules to perform floating point conversion. The following are representations of each sub-module showing all the input and output reg's:

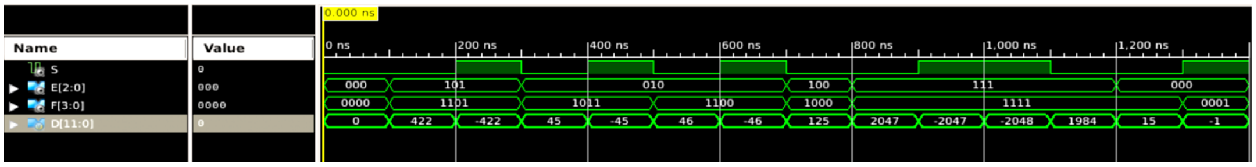




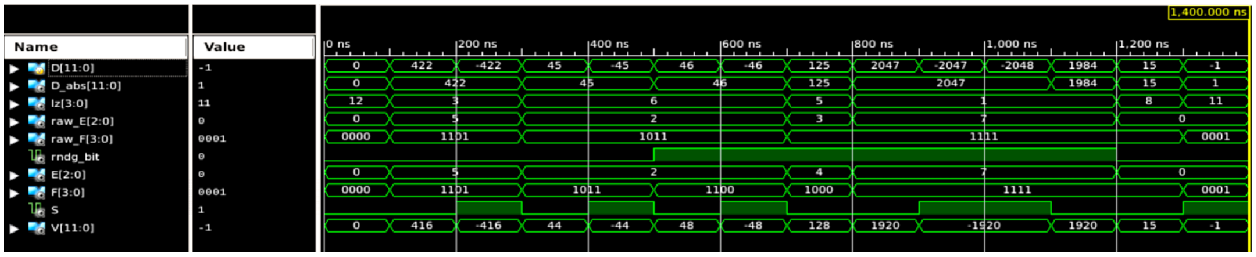
The intermediate variables of lz (# of leading zeros), D_abs (magnitude of D), raw_E (E before rounding), raw_F (F before rounding) and rndg_bit (rounding bit) are declared as wires in the top-level module *FPCVT* so that they can be connected to the output ports of the instantiated sub-modules.

Simulation Documentations

The following screenshot shows FPCVT testbench simulation for a few general as well as special cases (with the radix of D changed from binary to signed decimal for readability):



A more detailed screenshot of the simulation showing the intermediary variables as well as the value V of the output floating point representation is shown below:



All these cases demonstrate two's complement to sign-magnitude, counting of leading zeros, and rounding. Some of the general and edge cases demonstrated here are:

- D = [0000_0000_0000] or 0 → S = 0, E = 000, F = 0000
  D = [0111_1111_1111] or -1 → S = 1, E = 000, F = 0001
  D = [0000_0000_1111] or 15 → S = 0, E = 000, F = 1111
  There are >=8 leading zeros so E is 0. Subsequently, F consists of the 4 least significant bits.

- D = [0001_1010_0110] or 422 → S = 0, E = 101, F = 1101
  D = [1110_0101_1010] or -422 → S = 1, E = 101, F = 1101
  These cases demonstrate rounding down. The number of leading bits is 3 so E is set to 5. The 5th leading bit in D_abs is 0 so the number is rounded down. The value of the output FP representation is $\pm$ 416.

- D = [0000_0010_1110] or 46 → S = 0, E = 2, F = 1100
  D = [1111_1101_0010] or -46 → S = 1, E = 2, F = 1100
  These cases demonstrate rounding up with no overflow. The number of leading bits is 6 so E is set to 2 and F is initially 1011. The 5th leading bit is 1 so the number is rounded up by incrementing F by 1: F becomes 1100. The value of the FP representation is $\pm$ 48.

- D = [0000_0111_1101] or 125 → S = 0, E = 4, F = 1000
  This test case demonstrates rounding up with overflow and E < 7. Here, the number of leading bits is 5 so E is initially set to 3. F is initially set to 1111, which would lead to overflow upon adding 1, so it is replaced by 1000 and E is instead incremented by 1 → E = 4. The value of the FP representation is 128.

- D = [0111_1111_1111] or 2047 → S = 0, E = 111, F = 1111
  D = [1000_0000_0001] or -2047 → S = 1, E = 111, F = 1111
  These test cases demonstrate rounding up with overflow and E = 7. These very large linear encodings cannot be rounded up since F is 1111 and E = 111 cannot be incremented further, so instead the FP representation of these inputs is set to the largest possible FP values: ± 1920.

- D = [1000_0000_0000] or -2048 → S = 0, E = 111, F = 1111
  Here, the magnitude of D i.e. D_abs is set to 2047 since 2048 cannot be represented in 12-bit two's complement. After this, E and F are determined in the same way as when D = -2047.

Conclusion

The top-level module *FPCVT* and its sub-modules *sign_mag, leadings_0s_bits*, and *rounding* form the different blocks of the floating point conversion circuit and perform the functions of two's complement to sign-magnitude separation, leading zeros counting, leading bits extraction and rounding down (truncation) or rounding up. Some of the difficulties I encountered while doing this lab were while writing code for the *leading_0s_bits* sub-module. I initially used complicated nested for loops and kept getting type errors in the assignment of the appropriate bits to the significand F. I was able to fix these issues by simplifying my code to use a while statement to count the leading zeros and by making sure that everything was in its correctly defined always block.