University of California, Los Angeles

M152A Lab2

Floating Point Conversion

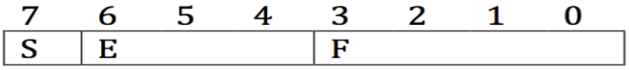
Group Members:
Chengyu Wang
Simeng Pang
Yaowei Guo

TA:
Jia Guo

July 11, 2017

Introduction

This lab is about converting a 12-bit twos-complement integer to an 8-bit floating point number. The converted floating point number consists of three parts: a sign bit, three exponent bits and four significand bits.



The value represented by this 8-bit number is:

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

Design

There is only one module called FPCVT in this project. In this module, there are 4 steps:

- Determine the sign bit
- Determine the exponent field
- Determine the significand filed with rounding
- Most negative number case

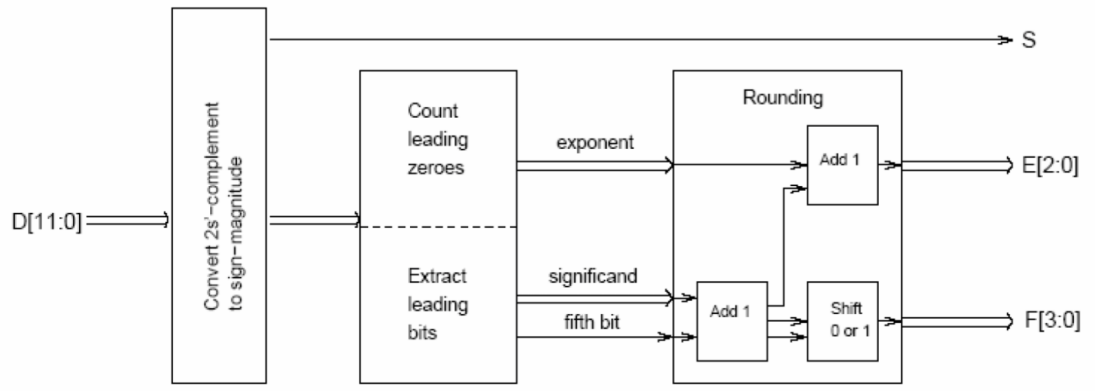


Figure 1: Overall design diagram from lab manual

Sign bit

Sign bit is the most significant bit of the 12-bit input number. The code goes as follows:

$$S = D[11]$$

Exponent bit

To count the leading zeroes of the input, the first step is to convert the twos-complement number to the number of its absolute value. The code goes as follows:

$$tmp = \sim D + 1$$

We have a table of the relationship between leading zeroes of tmp and the exponent filed.

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

According to this table, we can get the exponent field of 3 bits.

Significand and Rounding

There are two cases to consider here: less than 8 leading zeroes or not.
When there are 8 or more leading zeroes, the exponent will be 0. So we simply need to take the 4 least significant bits.

$$F = newtmp[11 : 8]$$

When there are less than 8 leading zeroes, we first take the four bits following the last leading zero as the significand. The code goes as follows:

$$newtmp = tmp << lead\_zero;$$

$$F = newtmp[11 : 8];$$

Then we use the fifth bit following the last leading zero to judge whether there is rounding. If the fifth bit ($newtmp[7]$) is 0, the nearest number can be obtained by truncating-simply using the first four bits. If the fifth bit is 1, the nearest number is obtained by rounding up-adding 1 to the first four bits.
When rounding up happens, there are three cases:

- The four bits are not 1111. After adding one to the four bits, no overflow happens. Then we use the result after addition as the significand.
- The four bits are 1111 and the exponent bits are not 111. After adding one to the four bits, there is overflow.

  $$1111 + 1 = 10000$$

  This problem is solved by right shift the significand by 1 bit and get 1000. Meanwhile, exponent will increase by 1 to compensate. The code goes as follows(Decimal value corresponding to the bit string is used in the code):

  $if(F == 15)$
  $begin$
  $F = 8;$
  $if(E! = 7)$
  $E = E + 1;$
- The four bits are 1111 and exponent bits are 111. If exponent increases by 1, there will be exponent overflow. So, we change nothing in this case.

Most negative number case

When the input is 100000000000, the most negative number that can be represented within 12 bits, the leading zeroes counted will be 0.

$$\sim 100000000000 + 1 = 100000000000$$

In this case we simply give the exponent and the significand the largest possible values(111 and 1111, in decimal value 7 and 15).

$if(D == 12'b100000000000)$

begin

$E = 7;$

$F = 15;$

end

Simulation Document

In this section we cover many possible 12-bit input cases, D, and show the outputs S, E and F. We consider the normal cases and edges cases and check that if they satisfy the rounding requirement.

Test Cases:
D = 12'b100000000000;
S:1, E:111, F:1111 //this is the smallest number we can test, since negating this number gives the same number, we have to treat this as a special case.

D = 12'b000000101100;
S:0, E:010 F:1011 //this is the normal case

D = 12'b000110110000;
S:0, E:101, F:1110 //this is the case when we have to round up F because there is a 1 following the truncated number, so we change 1101 to 1110 for F.

D = 12'b000001111101;
S:0, E:100, F:1000 //this is the special case for rounding. Since we can only take four bits for F, we have to consider the fifth significant number. Since it's 1, we have to round up. At the same time E should also add one.

D = 12'b011111000000;
S:0, E:111, F:1111 //this is the special case when F needs to round up but E has already reached its biggest number, so we can just keep both E and F with their largest representation.

D = 12'b111111111111;

S:1, E:000, F:0001 //this is the negative number conversion. We first converted it to positive representation using two's complement, then we just set S to 1 and do the normal conversion.

D = 12'b000000000000;
S:0, E:000, F:0000 //this is representation of 0, so S,E and F are all zero.

D = 12'b111111110000;
S:1, E:001, F:1000 //this is also a negative number conversion.

D = 12'b011111001010;
S:0, E:111 F:1111 //this is also the special case that the number needs to be round up but E and F both reach the biggest number, so we just keep E and F as before.

D = 12'b100000011111;
S:0, E:111, F:1111 //this is the negative number case that after converting it using two's complement, both E and F reach the largest number and we don't need to round up then.

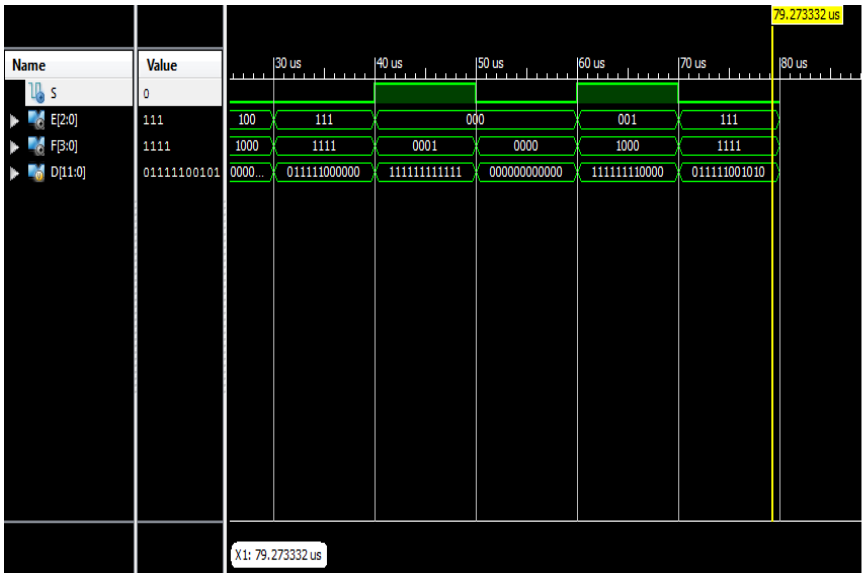


Figure 2: Output simulation waveforms of some testcases