COL216 – Computer Architecture
Assignment 11
Floating Point Addition (32-bit)

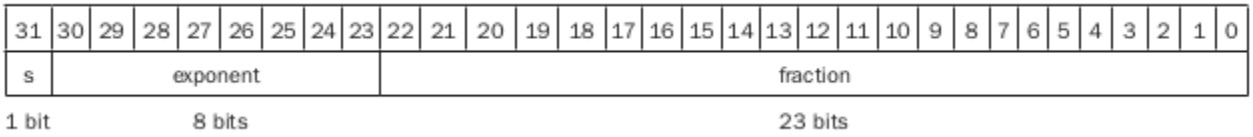
(Manoj Kumar, cs5180411@cse.iitd.ac.in)

AIM :

In this assignment, we implemented a Floating Point Adder using the encodings of IEEE 754 floating point numbers.

Representation of Floating Point Numbers :

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), *exponent* is the value of the 8-bit exponent field (including the sign of the *exponent*), and *fraction* is the 23-bit number.



In general, floating-point numbers are of the form :

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

Where

F (significand) = (1+*Fraction*)

E = (*Exponent – bias*)

For IEE 754 single precision, *bias* is 127

Floating Point Addition :

1. Compare the exponents of the two numbers,; shift the smaller number to the right until its exponent would match the larger exponent.

2. Now exponents of both are same as the larger *exponent*. Do bitwise-addition of both significands.

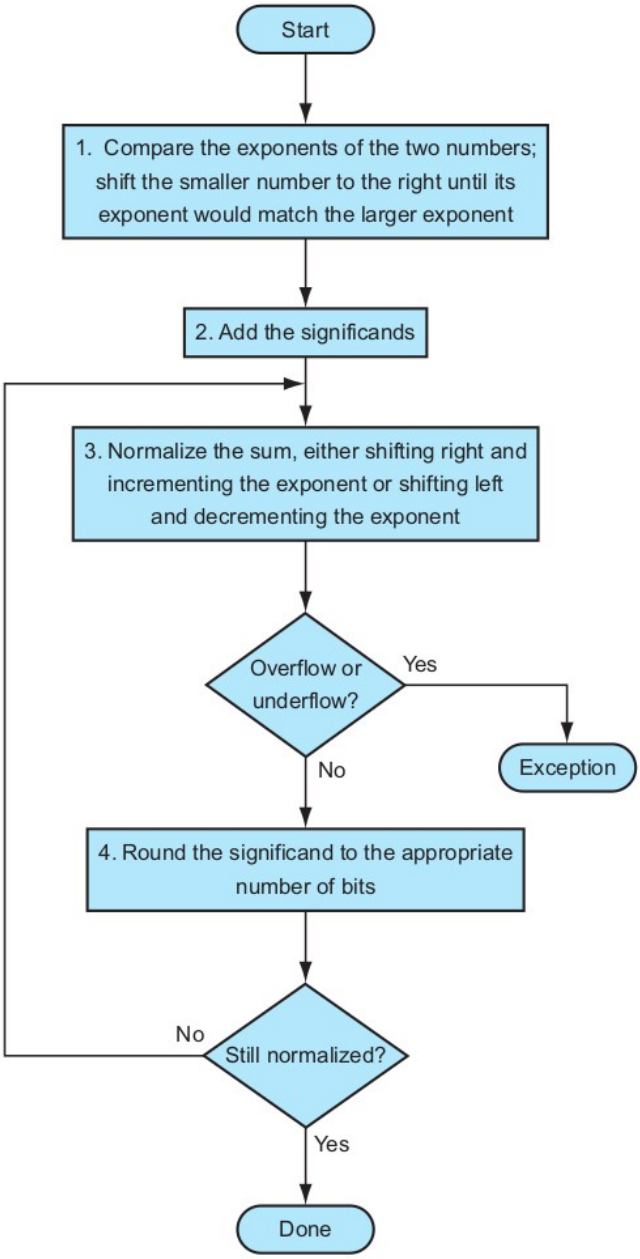
3. Basically our sum has been calculated, but we need to normalize it and round off the extra digits to fit it in 32 bits. So normalize the sum by either shifting one digit to right and incrementing the exponent or to left and decrementing the *exponent*.

4. Now check for Overflow or underflow, if any of both occurs, raise exception.

5. If overflow or underflow doesn’t occur, then Round the significand to the appropriate number of bits.

6. If it is still not normalized, go to the step 3, and if it is, addition of floating points is done.

FLOW-CHART OF ADDITION PROCESS IS SHOWN BELOW :



Underflow :

If the *exponent* has minimum value (all zero), the underflow situation happens. i.e. if the value of the exponent is less than 1, we will consider it as an underflow.

Overflow :

If the exponent has maximum value(all one), the overflow situation happends. i.e. the value of the exponent is greater than 254, we will consider it as an overflow.

Rounding Errors :

Not every decimal number can be expressed exactly as a floating point number. This can be seen when entering “0.1” and examining its binary representation which is either slightly smaller or larger, depending on the last bit.

Rules for Rounding off digits:

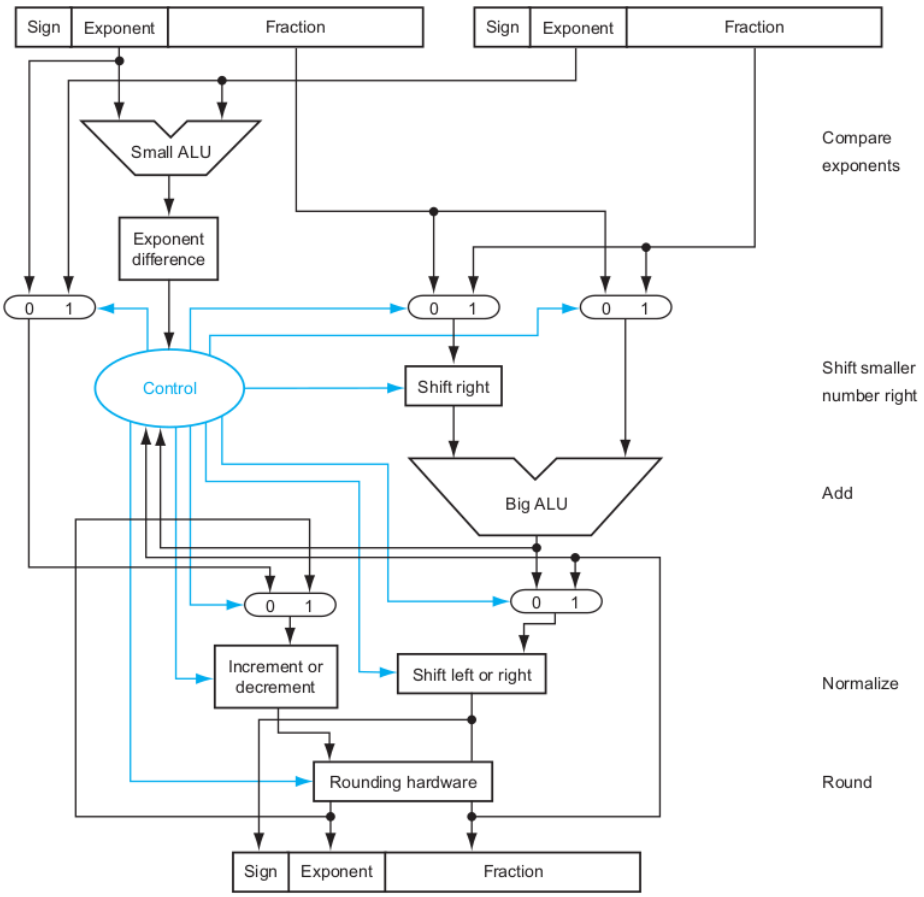
We are all familiar with rounding numbers in decimal system. Rounding in binary system is similar, but it still may cause some confusion. The biggest challenge is rounding fractions. For example, it may not be obvious right away why the fraction 0.11101 when rounded to 2 places after the decimal point results in integer 1.
There are five rounding methods, as defined by IEEE-754 standard, and most of them are pretty straightforward. The first two round to a nearest value (ties to even and ties away from zero); the others are called directed roundings: towards zero, towards positive infinity and towards negative infinity.

The general rule when rounding binary fractions to the n-th place prescribes to check the digit following the n-th place in the number. If it’s 0, then the number should always be rounded down. If, instead, the digit is 1 and any of the following digits are also 1, then the number should be rounded up. If, however, all of the following digits are 0’s, then a tie breaking rule must be applied and usually it’s the ‘ties to even’. This rule says that we should round to the number that has 0 at the n-th place.

- **0.11001** — rounds down to **0.11**, because the digit at the **3**-rd place is **0.**
- **0.11101** —rounds up to **1.00**, because the digit at the **3**-rd place is **1** and there are following digits of **1** (**5**-th place).
- **0.11100** — apply the *‘ties to even’* tie breaker rule and round up because the digit at **3**-rd place is **1** and the following digits are all **0**'s.

Block Diagram of an arithmetic unit dedicated to floating-point addition :

The steps of the Figure below correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.



TEST CASES (input1.txt):

Test case 1 :
number 1 :
Floating point representation : 11001100110011001111110011001100
Decimal Representation : -107472480.0
Binary Representation : -1.10011001111110011001100 x 2^(26)
number 2 :
Decimal Representation : -107472480.0
Floating point representation : 11001100110011001111110011001100
Binary Representation : -1.10011001111110011001100 x 2^(26)

PROGRAM OUTPUT :
11001101010011001111110011001100
total cycles consumed : 4

ACTUAL ADDITION (64-bit calculator) :
Decimal representation : -214944960.0

REMARKS :
Program output is same as actual addition.

Test case 2 :

number 1 :

Floating Point Representation : 01001110110011001111110011001101

Decimal Representation : 1719559808.0

number 2 :

Floating point Representation : 11001100110111001101110011001110

Decimal Representation : -115795568.0

PROGRAM OUTPUT :

Floating Point Representation : 01001110101111110010111100000000

total cycles : 4

ACTUAL ADDITION (64-bit calculator):

Decimal : 1603764240.0

REMARKS (Rounding Errors) :

decimal representation of the program output is different from actual addition (64-bit calculator output) becuase of rounding off some non-zero LSB digits of the significand in cycle 4. Baiscally we cannot represent the actual addition value in 32 bits because fraction part is large enough to fit in fraction 23 bits. So an error of -16 due to conversion by rounding off LSBs appears in the output.

Test case 3 :

number 1 :

Floating Point Representation : 01111111010000000000000000000000

Decimal Representation : 1.1 x 2^(127)

number 2 :

Floating Point Representation : 01111111010000000000000000000000

Decimal Representation : 1.1 x 2^(127)

PROGRAM OUTPUT :

OVERFLOW

total cycles : 3

REMARKS (Overflow) :

if we use 64-bit calculator, we get the result:

11.0 x 2^(127)

Normalized form : 1.1 x 2^(128)

Here as we see the exponent part of the floating point is 255 that is a situation of overflow. So we can’t store this number in 32 bit IEEE 754 convention.

Test case 4 :

number 1 :

Floating Point Representation : 00000001010100000000000000000000

Binary Representation : 1.101 x 2^(-125)

number 2 :

Floating Point Representation : 10000001010000000000000000000000

Binary Representation : -1.1 x 2^(-125)

PROGRAM OUTPUT :

UNDERFLOW

total cycles : 5

REMARKS (Underflow) :

If we do manual calculation we will get

0.001 x 2^(-125)

Normalized form : 1.0 x 2^(-127)

Here as we see the *exponent* part of the floating point is 0 that is a situation of underflow. So we can't store this number in 32-bit IEEE 754 convention.

Test case 5 :

number 1 :

Floating Point Representation : 01111111111111111111111111111111

Binary Form : 1.11111111111111111111 x 2^(128)

number 2 :

Floating Point Representation : 10011111010000000000000000000000

Binary Form : -1.01 x 2^(-64)

PROGRAM OUTPUT :

Value of number 1 is NaN

Total cycles : 1

REMARKS (Input is NaN):

during the first cycle, when we compare the *exponents* of the given floating points, we observe that exponent of number 1 is greater than 254 (255) that is the condition of overflow. So the given input is Not a valid Number (NaN).

Test case 6 :

number 1 :

Floating Point Representation : 00011111010000000000000000000000

Binary Form : 1.01 x 2^(-64)

number 2 :

Floating Point Representation : 11111111111111111111111111111111

Binary Form : -1.11111111111111111111 x 2^(128)

PROGRAM OUTPUT :

Value of number 2 is NaN

Total cycles : 1

REMARKS (Input is NaN) :

during the first cycle, when we compare the *exponents* of the given floating points, we observe that exponent of number 2 is greater than 254 (255) that is the condition of overflow. So the given input is Not a valid Number (NaN).

Test case 7 :

number 1 :

Floating Point Representation : 00001111001000000000000000000000

Binary Form : 1.01 x 2^(-97)

number 2 :

Floating Point Representation : 10101111001000000000000000000000

Binary Form : -1.01 x 2^(-33)

PROGRAM OUTPUT :

10101111001000000000000000000000

TOTAL cycles : 4

REMARKS (Rounding Errors):

Since, magnitude of second number is too large as compared to number 1. So after rounding off LSBs of significands we got the same output as input 2.

Test case 8 :

number 1 :

Floating Point Representation : 00001000001000000000000000000000

Binary Form : 1.01 x 2^(-111)

number 2 :

Floating Point Representation : 10001000000000000100000000000000

Binary Form : =1.000000001 x 2^(-111)

PROGRAM OUTPUT :

00000110111111100000000000000000

Total cycles : 8

REMARKS (total cycles increments) :

When we add both floating points, we first compare the *exponent* part of both and add the significands. We got the following de-normalized significand after addition:

0.00111111100000000000000

Here observe that it is not normalized, so to normalize it, we need to shift it to the left for 3 decimal places. By the above explained method, we know that each shift take one loop (2 cycles) so total cycles = 1+1+3*(1+1)

To run input1.txt : make run1
To run input2.txt : make run2
To run input3.txt : make run3

Submitted By :
Manoj Kumar
2018CS50411