ASSIGNMENT 11 REPORT

Name - Diwakar Prajapati
Entry - 2018CS10330

The order of the test cases here are the same as given in the input file to avoid any confusion.
*Give input in a file "**input.txt**".*
To run the program, type in terminal:

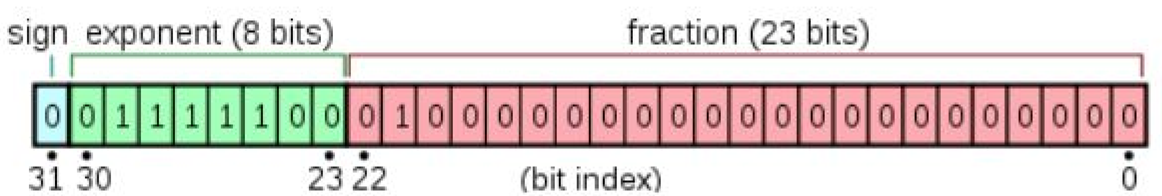
> ***g++ Adder.cpp***
> ***./a.out***

*The output is in the form of located in the same directory as a file "**output.txt**".*

AIM: Software implementation of Floating Point Adder.

General Conventions:

In general the single precision floating point numbers are represented in the hardware using 32 bits. Although for higher precision like doubles, they are represented using two registers having 32 bits each, so a total of 64 bits.
The format of single precision number is:



where a number is given as: **$(-1)^s \times (1.f)_2 \times 2^{exponent - 127}$**

Special numbers are also representable, like

Since it is signed representation, we have +0 & -0, +∞ & -∞

+0 = 0 00000000 00000000000000000000000

-0 = 0 00000000 00000000000000000000000

+∞ = 0 11111111 00000000000000000000000

-∞ = 1 11111111 00000000000000000000000

If we have at least one non zero in the fraction part then, it becomes NaN (Not a Number)

Example of NaN: X 11111111 00110000010000000000000

Design detail:

The simulation assumes that we have two signed floating point numbers in two 32 bit registers. Now two add the number we are sending the data to ALU block, where it uses another two registers to add the significands. So the addition of numbers is 32-bits.

*"Rather than using all the digits of the number, floating-point hardware normally operates on a fixed number of digits. Suppose that the number of digits kept is **p**, and that when the smaller operand is shifted right, digits are simply discarded (as opposed to rounding)."*

This statement is taken from:

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#693

In my implementation **p = 32.**

Implementation Details:

The implementation is quite simple.

Step 1: Take input from the file and parse the two 32-bit floating point numbers.

Step 2: Shifting the exponents based on their decimal values. Decimal values calculated using the below function:

```
long long int todecimal(string s){ ... }
```

Step 3: append extra zeros at the end(if needed) to make the number 32 bits long.

Note- I have used a decimal also so the length of the number in string form is 32.

```
long long int todecimal(string s){ ... }
```

Step 4: Add the significands.

```
string addsignificand(string frac1,string frac2,string
sign1,string sign2,string &sign){ ... }
//frac1:fraction part of first number
//frac2:fraction part of second number.
//sign1:sign of first number.
//sign2:sign of second number.
//sign: sign of resulting number.
```

Step 5: Normalizing the result.

```
void normalize(string &s,long long int &dec_exp){ ... }
```

.Step 6: Rounding the result.

```
void round(string &s){ ... }
```

Test Cases

//Simple test case

Test Case 1: 00001111110000000000000010010000 00001111110000000000000000010000

Test Case 2: 00001111110000000000000010010000 10001111110000000000000000010000

//OVERFLOW

Test Case 3: 01111111011111111111111111111111 01111110111111111111111000000000

//Simple test case, having one number ZERO

Test Case 4: 00000000000000000000000000000000 01111110111111111111111000000000

//UNDERFLOW

Test Case 5: 00000000100000000001000100000000 10000000100000000001000000000000

//Simple test case

Test Case 6: 00001100100000000001000100000000 10001100100000000001000100000000

//Corner cases, i.e. including INF and NaN.

Test Case 7: 01111111100000000000000000000000 01010101001101000000000000000000

Test Case 8: 01111111100000011000000000000000 01010101001101000000000000000000

Test Case 9: 01111111100000000000000000000000 01111111100000011000000000000000

Test Case 10: 01111111100000000000000000000000 11111111100000000000000000000000

//Miscellaneous cases where exponent shifting, normalisation and rounding occurs.

Test Case 11: 00000010110000000000010000000000 10000110110000000000110000000000

Test Case 12: 10000010110000000000010000000000 10000110110000001000110000000000

Test Case 13: 00010110011100011100011100011100 00011000111100011100011100011100

Test Case 14: 10100101011001100110011001100110 11001010011001100110011001100110

Test Case 15: 11000101011001100110011001100110 00010101001100110011001100110011

Test Case 16: 00001111011000110001100011000110 10010101001100110011001100110011

Test Case 17: 10001111011000110001100011000110 11111111001100110011001100110011

//Cases where renormalization occurs after rounding.

Test Case 18: 00000001001010101010101010101010 00000000101010101010101010101011

Test Case 19: 00100100010011001100110011001111 00100101010011001100110011001100

Test Case 20: 10100100010011001100110011001111 10100101111001100110011001100110

The details of each test case are given below.

Test Case - 1:

00001111110000000000000010010000 & **00001111110000000000000000010000**

0_00011111_10000000000000010010000 & 0_00011111_10000000000000000010000

A = + 1.10000000000000010010000 x $2^{31 - 127}$

B = + 1.10000000000000000010000 x $2^{31 - 127}$

A = + 1.10000000000000010010000 | 00000000 x $2^{31 - 127}$

B = + 1.10000000000000000010000 | 00000000 x $2^{31 - 127}$

A+B = + 11.00000000000000010100000 | 00000000 x $2^{31 - 127}$

Normalising = 1.10000000000000001010000 | 00000000 x $2^{32 - 127}$

Rounding = 1.10000000000000001010000 x $2^{32 - 127}$

Answer

Sign bit	= (+)	0
Exponent	= (32)	00100000
Fraction	=	10000000000000001010000
Result	=	**00010000010000000000000001010000**
Cycle	=	**4**

Test Case - 2:

00001111110000000000000010010000 & **10001111110000000000000000010000**

0_00011111_10000000000000010010000 & 1_00011111_10000000000000000010000

A = + 1.10000000000000010010000 x $2^{31 - 127}$
B = - 1.10000000000000000010000 x $2^{31 - 127}$

A = + 1.10000000000000010010000 | 00000000 x $2^{31 - 127}$
B = - 1.10000000000000000010000 | 00000000 x $2^{31 - 127}$
A+B = + 0.00000000000000010000000 | 00000000 x $2^{31 - 127}$

Normalising = 1.00000000000000000000000 | 00000000 x $2^{15 - 127}$
Rounding = 1.00000000000000000000000 x $2^{15 - 127}$

Answer

Sign bit = (+) 0
Exponent = (15) 00001111
Fraction = 00000000000000000000000
Result = **00000111100000000000000000000000**
Cycle = **4**

Test Case - 3:

01111111011111111111111111111111 & **01111110111111111111111000000000**

0_11111110_11111111111111111111111 & 0_11111101_11111111111111000000000

A	=	+	1.11111111111111111111111		x	$2^{254 - 127}$
B	=	+	1.11111111111111000000000		x	$2^{253 - 127}$

A	=	+	1.11111111111111111111111	| 00000000	x	$2^{254 - 127}$
B	=	+	0.111111111111111000000000	| 00000000	x	$2^{254 - 127}$
A+B	=	+	10.111111111111110111111110	| 00000000	x	$2^{254 - 127}$

Normalising = 1.0111111111111110111111110 | 00000000 x $2^{255 - 127}$

OVERFLOW

Answer: **OVERFLOW**

Test Case - 4:

00000000000000000000000000000000 & 01111110111111111111111000000000

0_00000000_00000000000000000000000 & 0_11111101_11111111111111000000000

A	=	+	0.00000000000000000000000	x	$2^{0 - 127}$
B	=	+	1.11111111111111000000000	x	$2^{253 - 127}$

A	=	+	0.00000000000000000000000	| 00000000	x	$2^{253 - 127}$
B	=	+	1.11111111111111000000000	| 00000000	x	$2^{253 - 127}$
A+B	=	+	1.11111111111111000000000	| 00000000	x	$2^{253 - 127}$

Normalising =	1.11111111111111000000000	| 00000000	x	$2^{253 - 127}$
Rounding =	1.11111111111111000000000		x	$2^{253 - 127}$

Answer

Sign bit	= (+)	0
Exponent	= (253)	11111101
Fraction	=	11111111111111000000000
Result	=	**01111110111111111111111000000000**
Cycle	=	**4**

Note: One of the numbers is ZERO so the result is the same as the other number.

Test Case - 5:

00000000100000000001000100000000 & **10000000100000000001000000000000**

0_00000001_00000000001000100000000 & 1_00000001_00000000001000000000000

A	=	+	1.00000000001000100000000	x	$2^{1 - 127}$
B	=	-	1.00000000001000000000000	x	$2^{1 - 127}$

A	=	+	1.00000000001000100000000	| 00000000	x	$2^{1 - 127}$
B	=	-	1.00000000001000000000000	| 00000000	x	$2^{1 - 127}$
A+B	=	+	0.00000000000000100000000	| 00000000	x	$2^{1 - 127}$

Normalising =	1.00000000000000000000000	| 00000000	x	$2^{-14 - 127}$
	UNDERFLOW			

Answer: **UNDERFLOW**

<u>Test Case - 6:</u>

00001100100000000001000100000000 & **10001100100000000001000100000000**

0_00011001_00000000001000100000000 & 1_00011001_00000000001000100000000

A	=	+	1.00000000001000100000000	x	$2^{25 - 127}$
B	=	-	1.00000000001000100000000	x	$2^{25 - 127}$

A	=	+	1.00000000001000100000000	| 00000000	x	$2^{25 - 127}$
B	=	-	1.00000000001000100000000	| 00000000	x	$2^{25 - 127}$
A+B	=	+	0.00000000000000000000000	| 00000000	x	$2^{25 - 127}$

Normalising =	0.00000000000000000000000	| 00000000	x	$2^{0 - 127}$
Rounding =	0.00000000000000000000000		x	$2^{0 - 127}$

Answer

Sign bit	= (+)	0
Exponent	= (0)	00000000
Fraction	=	00000000000000000000000
Result	=	**00000000000000000000000000000000**
Cycle	=	**4**

We can also see that the given numbers have the same magnitude but opposite sign.

<u>Test Case - 7:</u>

01111111100000000000000000000000 & **01010101001101000000000000000000**

0_11111111_00000000000000000000000 & 0_10101010_01101000000000000000000

A	=	+	1.00000000000000000000000	x	$2^{255 - 127}$	(INF)
B	=	+	1.01101000000000000000000	x	$2^{170 - 127}$	
			INFINITY			

Answer

INF

Test Case - 8:

01111111100000011000000000000000 & **01010101001101000000000000000000**

0_11111111_00000011000000000000000 & 0_10101010_01101000000000000000000

A	=	+	1.00000011000000000000000	x	$2^{255 - 127}$	(NaN)
B	=	+	1.01101000000000000000000	x	$2^{170 - 127}$	
			Not a Number			

Answer

NaN

Test Case - 9:

01111111100000000000000000000000 & **01111111100000011000000000000000**

0_11111111_00000000000000000000000 & 0_11111111_00000011000000000000000

A	=	+	1.00000000000000000000000	x	$2^{255 - 127}$	(INF)
B	=	+	1.00000011000000000000000	x	$2^{255 - 127}$	(NaN)
			Not a Number			

Answer

NaN

Test Case - 10:

01111111100000000000000000000000 & 11111111100000000000000000000000

0_11111111_00000000000000000000000 & 1_11111111_00000000000000000000000

A = + 1.00000000000000000000000 x $2^{255 - 127}$ (INF)

B = - 1.00000000000000000000000 x $2^{255 - 127}$ (INF)

Not a Number

Answer

NaN

Test Case - 11:

00000010110000000000010000000000 & 10000110110000000000110000000000

0_00000101_10000000000010000000000 & 1_00001101_10000000000110000000000

A	=	+	1.10000000000010000000000	x	$2^{5 - 127}$
B	=	-	1.10000000000110000000000	x	$2^{13 - 127}$

A	=	+	0.00000001100000000000100	| 00000000	x	$2^{13 - 127}$
B	=	-	1.10000000000110000000000	| 00000000	x	$2^{13 - 127}$
A+B	=	-	1.01111110100101111111100	| 00000000	x	$2^{13 - 127}$

Normalising =	1.01111110100101111111100	| 00000000	x	$2^{13 - 127}$
Rounding =	1.01111110100101111111100		x	$2^{13 - 127}$

Answer

Sign bit	= (-)	1
Exponent	= (13)	00001101
Fraction	=	01111110100101111111100
Result	=	**10000110101111110100101111111100**
Cycle	=	**4**

Test Case - 12:

10000010110000000000010000000000 & **10000110110000001000110000000000**

1_00000101_10000000000010000000000 &
1_00001101_10000001000110000000000

A = - 1.10000000000010000000000 x $2^{5 - 127}$
B = - 1.10000001000110000000000 x $2^{13 - 127}$

A = - 0.00000001100000000000100 | 00000000 x $2^{13 - 127}$
B = - 1.10000001000110000000000 | 00000000 x $2^{13 - 127}$
A+B = - 1.10000001010011000000100 | 00000000 x $2^{13 - 127}$

Normalising = 1.10000001010011000000100 | 00000000 x $2^{13 - 127}$
Rounding = 1.10000001010011000000100 x $2^{13 - 127}$

Answer

Sign bit = (-) 1
Exponent = (13) 00001101
Fraction = 10000001010011000000100
Result = **10000110110000001010011000000100**
Cycle = **4**

Test Case - 13:

00010110011100011100011100011100 & **00011000111100011100011100011100**

0_00101100_11100011100011100011100 & 0_00110001_11100011100011100011100

A	=	+	1.11100011100011100011100	x	$2^{44 - 127}$
B	=	+	1.11100011100011100011100	x	$2^{49 - 127}$

A	=	+	0.00001111000111000111000	| 11100000	x	$2^{49 - 127}$
B	=	+	1.11100011100011100011100	| 00000000	x	$2^{49 - 127}$
A+B	=	+	1.11110010101010101010100	| 11100000	x	$2^{49 - 127}$

Normalising =	1.11110010101010101010100	| 11100000	x	$2^{49 - 127}$
Rounding =	1.11110010101010101010101		x	$2^{49 - 127}$

Answer

Sign bit	= (+)	0
Exponent	= (49)	00110001
Fraction	=	11110010101010101010101
Result	=	**00011000111110010101010101010101**
Cycle	=	**4**

Test Case - 14:

10100101011001100110011001100110 & **11001010011001100110011001100110**

1_01001010_11001100110011001100110 &
1_10010100_11001100110011001100110

A	=	-	1.11001100110011001100110		x	$2^{74 - 127}$
B	=	-	1.11001100110011001100110		x	$2^{148 - 127}$
A	=	-	0.00000000000000000000000	| 00000000	x	$2^{148 - 127}$
B	=	-	1.11001100110011001100110	| 00000000	x	$2^{148 - 127}$
A+B	=	-	1.11001100110011001100110	| 00000000	x	$2^{148 - 127}$
Normalising	=		1.11001100110011001100110	| 00000000	x	$2^{148 - 127}$
Rounding	=		1.11001100110011001100110		x	$2^{148 - 127}$

Answer

Sign bit	= (-)	1
Exponent	= (148)	10010100
Fraction	=	11001100110011001100110
Result	=	**11001010011001100110011001100110**
Cycle	=	**4**

Test Case - 15:

11000101011001100110011001100110 & **00010101001100110011001100110011**

1_10001010_11001100110011001100110 &
0_00101010_01100110011001100110011

A	=	-	1.11001100110011001100110		x	$2^{138 - 127}$
B	=	+	1.01100110011001100110011		x	$2^{42 - 127}$

A	=	-	1.11001100110011001100110	| 00000000	x	$2^{138 - 127}$
B	=	+	0.00000000000000000000000	| 00000000	x	$2^{138 - 127}$
A+B	=	-	1.11001100110011001100110	| 00000000	x	$2^{138 - 127}$

Normalising =	1.11001100110011001100110	| 00000000	x	$2^{138 - 127}$
Rounding =	1.11001100110011001100110		x	$2^{148 - 127}$

Answer

Sign bit	= (-)	1
Exponent	= (148)	10010100
Fraction	=	11001100110011001100110
Result	=	**11001010011001100110011001100110**
Cycle	=	**4**

Test Case - 16:

00001111011000110001100011000110 & **10010101001100110011001100110011**

0_00011110_11000110001100011000110 & 1_00101010_01100110011001100110011

A	=	+	1.11000110001100011000110	x	2^{30-127}
B	=	-	1.01100110011001100110011	x	2^{42-127}

A	=	+	0.00000000000111000110001	| 10001100	x	2^{42-127}
B	=	-	1.01100110011001100110011	| 00000000	x	2^{42-127}
A+B	=	-	1.01100110010010100000001	| 01110100	x	2^{42-127}

Normalising = 1.01100110010010100000001 | 01110100 x 2^{42-127}

Rounding = 1.01100110010010100000001 x 2^{42-127}

Answer

Sign bit	= (-)	1
Exponent	= (42)	00101010
Fraction	=	01100110010010100000001
Result	=	**10010101001100110010010100000001**
Cycle	=	**4**

Test Case - 17:

10001111011000110001100011000110 & **11111111001100110011001100110011**

1_00011110_11000110001100011000110 & 1_11111110_01100110011001100110011

A	=	-	1.11000110001100011000110		x	$2^{30 - 127}$
B	=	-	1.01100110011001100110011		x	$2^{254 - 127}$

A	=	-	0.00000000000000000000000	| 00000000	x	$2^{254 - 127}$
B	=	-	1.01100110011001100110011	| 00000000	x	$2^{254 - 127}$
A+B	=	-	1.01100110011001100110011	| 00000000	x	$2^{254 - 127}$

Normalising =	1.01100110011001100110011	| 00000000	x	$2^{254 - 127}$
Rounding =	1.01100110011001100110011		x	$2^{254- 127}$

Answer

Sign bit	= (-)	1
Exponent	= (254)	11111110
Fraction	=	01100110011001100110011
Result	=	**11111111001100110011001100110011**
Cycle	=	**4**

<u>Test Case - 18:</u>

00000001001010101010101010101010 & **00000000101010101010101010101011**

0_00000010_01010101010101010101010 & 0_00000001_01010101010101010101011

A = + 1.01010101010101010101010 x 2^{2-127}
B = + 1.01010101010101010101011 x 2^{1-127}

A = + 1.01010101010101010101010 | 00000000 x 2^{2-127}
B = + 0.10101010101010101010101 | 10000000 x 2^{2-127}
A+B = + 1.11111111111111111111111 | 10000000 x 2^{2-127}

Normalising = 1.11111111111111111111111 | 10000000 x 2^{2-127}
Rounding = 10.00000000000000000000000 | x 2^{2-127}
Normalising = 1.00000000000000000000000 | 0 x 2^{3-127}
Rounding = 1.00000000000000000000000 x 2^{3-127}

Answer

Sign bit = (+) 0
Exponent = (3) 00000011
Fraction = 00000000000000000000000
Result = **00000001100000000000000000000000**
Cycle = **6**

Test Case - 19:

00100100010011001100110011001111 & 00100101010011001100110011001100

0_01001000_10011001100110011001111 & 0_01001010_10011001100110011001100

A	=	+	1.10011001100110011001111	x	$2^{72 - 127}$
B	=	+	1.10011001100110011001100	x	$2^{74 - 127}$

A	=	+	0.01100110011001100110011	| 11000000	x	$2^{74 - 127}$
B	=	+	1.10011001100110011001100	| 00000000	x	$2^{74 - 127}$
A+B	=	+	1.11111111111111111111111	| 11000000	x	$2^{74 - 127}$

Normalising =	1.11111111111111111111111	| 11000000	x	$2^{74 - 127}$
Rounding =	10.00000000000000000000000		x	$2^{74 - 127}$
Normalising =	1.00000000000000000000000	| 0	x	$2^{75 - 127}$
Rounding =	1.00000000000000000000000		x	$2^{75 - 127}$

Answer

Sign bit	= (+)	0
Exponent	= (75)	01001011
Fraction	=	00000000000000000000000
Result	=	**00100101100000000000000000000000**
Cycle	=	**6**

Test Case - 20:

10100100010011001100110011001111 & **10100101111001100110011001100110**

1_01001000_10011001100110011001111 & 1_01001011_11001100110011001100110

A	=	-	1.10011001100110011001111		x	$2^{72 - 127}$
B	=	-	1.11001100110011001100110		x	$2^{75 - 127}$

A	=	-	0.00110011001100110011001	| 11100000	x	$2^{75 - 127}$
B	=	-	1.11001100110011001100110	| 00000000	x	$2^{75 - 127}$
A+B	=	-	1.11111111111111111111111	| 11100000	x	$2^{75 - 127}$

Normalising =	1.11111111111111111111111	| 11100000	x	$2^{75 - 127}$
Rounding =	10.00000000000000000000000		x	$2^{75 - 127}$
Normalising =	1.00000000000000000000000 | 0		x	$2^{76 - 127}$
Rounding =	1.00000000000000000000000		x	$2^{76 - 127}$

Answer

Sign bit	= (-)	1
Exponent	= (76)	01001100
Fraction	=	00000000000000000000000
Result	=	**10100110000000000000000000000000**
Cycle	=	**6**