COL216 – ASSIGNMENT 11
FLOATING POINT ADDITION

DIPEN KUMAR
2018CS50098

DESIGN –

I read two numbers from input file line by line for addition.
I follow the algorithm that was covered in the class and is outlined in figure 3.14 (page 205 of the textbook). I first read the numbers and stores their exponents and significands with signs in decimal.
I have actually stored their significand times 2^23.

-	-	-	-	-	-	-	-	-	1	0	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1

Here fraction part is –

1	0	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1

And Hence significant is 1.10011000111111111100101
But I have stored 110011000111111111100101 in decimal.

STEP 1 is to compare the exponents of the two number – (and cycle ++)
First I do is what I raise Invalid Input exception when any one has exponent of 128. I had already considered bias in exponent calculation.

Now, If I have been given correct input, Algorithm say shift the smaller number (number with small exponent) to the right until its exponent would match the larger exponent.
That is same as-
Divide the smaller exponent number by 2^(difference in exponent) and raise the smaller exponent by difference in exponent which will make both their exponents same to the larger exponent.

NOTE: I have stored significands in decimal representation in a data type double of C++ and hence when it is divided by 2^(something) which may be large and we may get very small number but it will always fall in the range of double of C++ because double uses double precision to store floating point i.e. 32 + 32 = 64 bits. Our data may go from [2^(-254) to 2^(24)) with max 23 bit precision which is an easy game for double which can take [2^(-1022) to 2^(1024)) with 52 bits of precision and also denormalized numbers.

STEP 2 is to add the significands – (and cycle ++)
As I mentioned earlier I have stored significands in decimal form in double data type. I will add both significands by basic addition operator provided to us by C++.

STEP 3 is to normalize the sum, either shifting right and increasing the exponent or shifting left and decrementing the exponent – (and cycle ++)

Again to mention I have stored significand times 2^(23) hence for normalization I will try to bring the addition result from previous step in the range between [2^(23) to 2^(24)) by either multiplying or dividing by 2 to the power "something" and decreasing or increasing the result exponent by "something" which was till now equal to larger given exponent.

NOTE: If not I have stored significands times 2^(23), I would be bringing significands in [1 to 2) range. Also note that I said I will try because in case when significand is zero, No matter what you multiply or divide it will be zero. Hence I was handled separately and made exponent of result equal to -127

After Normalization check for OVERFLOW and UNDERFLOW if exponent is greater than 127 and less than -126 respectively. Note zero is handled separately. And will not give UNDERFLOW though its exponent was assigned negative 127.

STEP 4 is to round the result significand to the appropriate number of bits – (and cycle ++)

For round we have to see 24th bit after decimal in result. If it is 1 then increase the number by 2^(-23) else do nothing.

Since my significand is in decimal representation times 2^(23) which means its fraction part is equal to fraction part of 2^(23) times actual significand which is equal to 2^(-1)*(24th bit after decimal) + 2^(-2)*(25th bit after decimal) and continue so on.
Now see if 24th bit after decimal in result is 1 then fraction part is greater than equal to 0.5 else less than 0.5

Keep this in mind if fraction part of my significand which is in decimal form which is 2^(23) times actual significand is greater than 0.5 then add 2^(-23)*2^(23) that is equal to 1 else do nothing. This is how I rounded by significant number.

I stored the integer part and checked fraction part, if it was greater than equal to 5 I increased my integer part else do nothing I have result.

Now rounding may make the significand again not in normalized form. Hence we will check, if not normalized we will follow again from step 3 else we have our answer.

I changed by number from decimal form to binary whose lower 23 bit is my fraction part times 2^(23) because my significand is from [2^(23) to 2^(24)) hence actual significant is my significand / 2^(23). Fraction part is (my significand / 2^(23)) -1. That is(my significand – 2^(23))/2^(23) that is (lower 23 bits)/2^(23)
Hence fraction part times 2^(23) is lower 23 bits.
fraction part times 2^(23) makes 22 down to 0 vector in 32 bit single precision register. Hence we will fill it with lower 23 bits of our significand converted in binary from decimal.
Also convert exponent adding bias to binary and fill it from 30 down to 23.
Fill sign bit 1 if significant was less than 0 else fill it with 0.

TESTCASES –

11111111110110101001010110001011 (-2^(128)*something)
01010101011010010001001011101001 (some good value)
Hence output is INVALID INPUT detected in 1 clock cycle

10101011111000101010000111110101 (some good value)
01111111110100111110101000110101 (+2^(128)*something)
Hence output is INVALID INPUT detected in 1 clock cycle

11111111100000000000000000000000 (-2^(128)*0) (-infinity)
01111111100000000000000000000000 (+2^(128)*0) (+infinity)
Hence output is INVALID INPUT detected in 1 clock cycle

11111111100000000000000000000000 (-2^(128)*0) (-infinity)
00110101110101001110101000111101 (some good value)
Hence output is INVALID INPUT detected in 1 clock cycle

11010101001010101010101010101010
11010100101010101010101010101011
(-2^(y)*1. 01010101010101010101010)
(-2^(y)*0.101010101010101010101011)
Result = (-2^(y)*1.11111111111111111111111) where y = 10101010 -127
Hence we will do rounding –
Result = (-2^(y)*10.00000000000000000000000)
Again go back to step 3 to do rounding-
Result = (-2^(y+1)*1.00000000000000000000000)
Hence final result will be 1 for minus 10101011 for exponent and all zero for fraction. => 11010101100000000000000000000000 in 6 clock cycles because normalization required after rounding.

11111111010101010111010100100101
11111111011010111010010000100010
(-2^(127)*1.10101010111010100100101)
(-2^(127)*1.11010111010010000100010)
Adding gives = (-2^(127)*11.something)
After normalized get = (-2^(128)*1.1something)
Hence OVERFLOW occurred, result will be OVERFLOW in detected in 3 clock cycles.

11111111001010101010101010101010
11111110101010101010101010101011
(-2^(127)*1.01010101010101010101010)
(-2^(126)*1.01010101010101010101011)
=> (-2^(127)*1.01010101010101010101010)
    (-2^(127)*0.101010101010101010101011)
Adding (-2^(127)*1.11111111111111111111111)
Not overflow now round =>(-2^(127)*10.00000000000000000000000)
But we now need to normalize.
=>(-2^(128)*1.00000000000000000000000)
But OVERFLOW occurred in 6 clock cycles

10000000110101010110100101010011
00000000101100101001010010010010
(-2^(-126)* 1.10101010110100101010011)
(+2^(-126)*1.01100101001010010010010)
Result => (-2^(-126)*0.1something)
Not normalized Hence do normalization and we get (-2^(-127)*1.something)
But it is not zero. Hence UNDERFLOW with clock cycle = 3.
NOTE – UNDERFLOW cannot be in 6 clock cycle because after normalization value will increase not decrease.

10000001110101010110100101010011
00000001101100101001010010010010
(-2^(-126)*1.10101010110100101010011)
(+2^(-126)*1.01100101001010010010010)
Result => (-2^(-126)*0.something)
Normalize and get result, after rounding still same number
Hence result is 10000000100010110101001100000100 in 4 clock cycles.

10110100101001010010101010100111
00110100101001010010101010100111
Something – something = 0
Hence 00000000000000000000000000000000 in 4 clock cycle because significant addition will give zero and after normalization it will -127 exponent.
And not UNDERFLOW because of 0 and after rounding no change hence 4 clock cycles.

10000000011010101000011010101001
00000000010100101010101001011100
(-2^(-127)*0.11010101000011010101001)
(+2^(-127)*0.10100101010101001011100)
Addition and => (-2^(-127)*0.001something)
Normalized and get => (-2^(-130)*1.something)
Hence UNDERFLOW in 3 clock cycles.

11010010100100101010010100101010
00000000000101001010100101001010
Similarly, we get
Result => 11010010100100101010010100101010 in 4 clock cycles

00001111111111111111111111111111
10000011111111111111111111111111
Result => 00001111111111111111111111111110 4

10000000000100000000000000000001
00000000000100000000000000000001
Something – Something is 0, we represent it as + 0
Result => 00000000000000000000000000000000 4

THANK YOU