Assignment - 11 Floating Point Adder

The directory contains the C++ implementation of the Floating-Point Adder. The file A11.cpp contains the source code whereas all the .txt files are the test cases that are used for testing the program. The report contains an overview of the method used and the example test cases.

The implementation is consistent with the suggested block diagram, however, a particular methodology regarding the rounding scheme is used. The source of which is mentioned in the references. The implementation is divided into 4 main stages namely:

- Exponent Comparison and Equalization.
- Addition of Operands.
- Normalization
- Rounding

The detailed descriptions of the aforementioned stages are as follows:

1. **Exponent Comparison and Equalization:** The numbers are read in this stage and the different fields such as the sign, exponent, and significand are extracted from the inputs. Three extra bits are introduced at the end of the inputs which are Ground (G), Round (R), and Sticky (S). The bits G and R simply ordinary bits and increase the precision and the bit S represents the presence of any 1 to the right of R bit which is now truncated due to the right shift. S = 1 denotes the presence of bit 1 and S = 0, denotes the absence of 1. Right shifts are performed to the number having smaller exponent, and the presence of bit 1 is tracked by calculating the bitwise OR of the bits that are truncated. In the end, a 27-bit number is formed with 3 extra utility bits. The implicit bit 1 is appended in this stage as well.

2. **Arithmetic Addition:** The 27-bit numbers obtained from the first stage are then converted to decimal numbers and stored in the long long format in C++. Depending on their respective sign bits, they are multiplied by -1. Their addition is performed, and the result is converted to a binary string of size 28, as the max size of the result after the addition of 2 numbers of size 27 would be 28.

3. **Normalization:** In this step, the result computed in the previous step is normalized. The normalization procedure would involve left as well as the right shift of the exponent and thus, they are handled separately. For the right shift, if the carry bit (MSB) at the end of the addition process is ‘1’, then a right shift needs to be performed and the value of the exponent would be increased by 1. 27 bits are extracted from the resultant 28 bits with the value of the S bit set as per its specification earlier. The **overflow** exception is checked as well in this subcase.

   If the carry bit isn’t 1, then left shifts are performed until the first non-zero bit is received. The exponent is reduced and the **underflow** condition is checked for the reduced exponent. Again the 27-bit number is extracted from the left-shifted result, and the last three bits remain consistent with their specifications.

4. **Rounding:** The result is now normalized and contains 27 bits. The rounding procedure is carried out depending on the values of the last 3 bits as follows-

   GRS - Action
   0xx - round down = do nothing (x means any bit value, 0 or 1)
   100 - this is a **tie**: round up if the mantissa's bit just before **G** is 1, else round down=do nothing
   101 - round-up
   110 - round-up
   111 - round-up

For the tie condition, rounding to the nearest even is followed, i.e if the bit preceding **G** is 0, then the number is rounded down, whereas if the bit preceding **G** is 1, 1 is added to the number and again the procedure goes to the stage 3.

Test Cases:
The number beside the output denotes the number of clock cycles required for the execution of the instructions.

1. 0 10000001 10000000000000000000000 = 6.0
   0 10000101 10000000000000000000000 = 96.0

Output: 0 10000101 1011000000000000000000000 = 102.0 4

2. 1 10000001 10000000000000000000000 = -6.0
   0 10000101 10000000000000000000000 = 96.0

Output: 0 10000101 011010000000000000000000 = 90.0 4

3. 0 10000001 10000000000000000000000 = 6.0
   1 10000101 10000000000000000000000 = -96.0

Output: 1 10000101 011010000000000000000000 = -90.0 4

4. 1 10000001 10000000000000000000000 = -6.0
   1 10000101 10000000000000000000000 = -96.0

Output: 1 10000101 100110000000000000000000 = -102.0 4

The above test cases were also tested after interchanging the places of the operands and yield the corresponding results. (No. 5-8)

```
     9.  0 10000001 10000000000000000000000  =  6.0
         0 10000001 10000000000000000000000  =  6.0

Output:  0 10000010 10000000000000000000000  =  12.0          4

    10. 0 10000001 10000000000000000000000  =  6.0
         1 10000001 10000000000000000000000  =  -6.0

Output:  0 00000000 00000000000000000000000  =  0.0           4

    11.  0 10000001 10100000000000000000000  =  -6.0
         1 10000001 10000000000000000000000  =  6.5

Output:  0 0111111 00000000000000000000000  =  0.5            4

    12. 0 10000011 10100000000000000000000  =  26
         0 10000000 10000000000000000000111  =  3.00000166893

Output:  0 10000011 11010000000000000000001  =  29.00000000      6

   13.  0 11111110 10000000000000000000000  =  2.55211775191e+38
         0 11111110 10000000000000000000000  =  2.55211775191e+38

Output:  Overflow Exception                                       3

   14.  0 11111111 10000000000000000000000  =  NaN
         0 11111111 10000000000000000000000  = NaN

Output: NaN                                                      1
```

15. 0 11111111 00000000000000000000000 = Infinity
    0 11111111 00000000000000000000000 = Infinity

Output: Infinity 1

16. 0 00000011 11100000000000000000000 = 8.81620763117e-38
    1 00000011 11000000000000000000000 = -8.22846045576e-38

Output: UnderFlow Exception 3

17. 0 10000011 01010101010101010101010 = 21.3333320618
    0 10000000 01010101010101010101010 = 2.66666650772

Output: 0 10000011 01111111111111111111111 = 23.9999980927 4

18. 0 10000000 10000000000000000001101 = 3.00000309944
    0 10000011 10000000000000000000000 = 24.0

Output: 0 10000011 10110000000000000000010 = 27.0000038147 6

19. 0 10000000 10000000000000000001111 = 3.00000357628
    0 10000101 10000000000000000000000 = 96.0

Output: 0 10000101 10001100000000000000000 = 99.0 4

20. 0 10000000 10000000000000000111111 = 3.00001502037
    0 10000101 10000000000000000000000 = 96.0

Output: 0 10000101 10001100000000000000010 = 99.0 6

21. 0 10000000 10000000000000000111111 = 3.01032853127
0 10000101 10000000000000000000000 = 96.0

Output: 0 10000101 10001100000000000000010 = 99.0103302002 6

22. 0 10000000 11100000101010010011100 = 3.75516414642
0 10000000 11100000000000000000000 = 3.75

Output: 0 1000000 111100000101010010011100 = 7.51032829285 4

23. 0 10000000 11100001010100100111001 = 3.76032853127
0 10000011 11101010101010101010101 = 30.6666660309

Output: 0 10000100 00010011011010100111110 = 34.4269943237 4

24. 0 10000000 11100001010100100111111 = 3.76032996178
0 10000111 11101010101010101010101 = 490.666656494

Output: 0 10000111 11101110011011010100111 = 494.426971436 4

25. 0 00000000 10000000000000000111111 = Denormalized number
0 10000101 10000000000000000000000 = 96.0

Output: Denormalized Number 1

For running the test case eg. tc1.txt, use **./a.out < tc1.txt**. The serial number of test cases correspond to the .txt files in which they are put.

Param Khakhar
2018CS10362

Reference for Rounding Scheme:

1. Floating Point Arithmetic Lecture by Prof. Anshul Kumar (NPTEL)
   https://www.youtube.com/watch?v=03fhijH6e2w&t=2384s