# COL216 Assignment 11

Vishal Bindal
2018CS50425

## **To Run**:

1. To print clock cycles along with the output, and total no of cycles, and input no., set `show_clock = true` in main.cpp. Else set it to false, and only the floating number outputs will be printed.
2. Run ‘ `make` ’ to create an executable ‘main’
3. Run ‘ `./main` ’ if the input is from ‘input.txt’ in the same directory, or else ‘ `./main <input_file_path>` ’ . Input file can also contain comments in the line after the 2 numbers, as shown in the sample input.txt submitted.

## **Design:**

The 32-bit floating point number **a** is parsed to yield the following:

1. **bool s**: set to a[0]. It denotes the sign of the number
2. **int exp**: set to integer value of a[1..8]. It denotes actual exponent of number + bias, in this case, actual exponent + 127. 0<=exp<=255. It can also refer to a special case as explained below
3. **vector<bool> f** : length 26, set as following:

-----------------------------------------------------------------------------------------------------------------------
f[0] | f[1] | f[2] , f[3] , f[4], ………………………………… f[24] | f[25]
-----------------------------------------------------------------------------------------------------------------------

- Here f[1] is the implicit bit: set as
  - 1 for exp != 0
  - 0 for exp == 0 : this is for the special case of Zero, and denormal representation
- f[2 ….. 24] is set equal to a[ 9…..31]. These are equal to the fraction bits given in a.
- The decimal no is assumed to be between f[1] and f[2].
- f[0] and f[25] are extra bits, kept to facilitate step 3 and 4.

- f[0] is kept in case an addition leads to an extra bit at the start, e.g. 1.0 + 1.0 = 10.0. Then it is normalised back during step 3
- f[25] is kept in case there is a bit at that position due to right shift in step 2, which if equal to 1, will lead to rounding up of the number in step 4.

# Algorithm:

1. Step 1: Equating exponents
    a. The exponents are compared, and the number having a smaller exponent will be shifted right to equate its exponent to the bigger one.
    b. Only one extra bit, f[25] will be stored during right shift and other data will be lost, as **only one bit is sufficient to round correctly in step 4**
2. Step 2: Adding bits
    a.
        i. If the sign of both numbers is same (s_a == s_b), then the f vector is added, by adding bits from right to left, storing the carry. (f_ans = add(f_a, f_b)).
        ii. The sign s of the ans is equal to either of the signs of input (s_ans = s_a)
    b.
        i. If sign of both numbers is opposite (s_a == not(s_b)), then the f vector is obtained by subtracting the smaller one (in magnitude) from the greater one (in magnitude). f_a and f_b are compared bitwise for this.
        ii. f_ans = sub(greater, smaller), where greater = max(f_a, f_b), smaller = min(f_a, f_b)
        iii. s_ans = s_a if greater == f_a, s_b otherwise
3. Step 3: Normalisation
    a. Case 1: f[0] is non-zero. Then addition in step 2 must have caused this. To restore the decimalonly one bit is sufficient to round correctly in step 4 point to be just after the first 1 in f, f is shifted right by 1 bit. (and exp incremented by 1)
    b. Case 2: f[0] and f[1] are both non-zero. Then the first 1 lies after the decimal point. The no of shifts are calculated so as to shift the first 1 before the decimal points. f is left shifted by those many shifts (and exp decreased)
4. Check overflow: overflow if exp==255 (except special cases like NaN,inf)
5. Step 4: Rounding
    a. Case 1: f[25] is zero. Then there’s no need to do anything.
    b. Case 2: f[25] is 1. Then we have to round up the number. (last bit == 1 is equivalent to say the last digit >= 5 in a 10-base system). For rounding up, add 00000000000000000000000010 (26-bit) to f.
6. Check normalisation: In case normalisation ends after step 4 (due to rounding up), go back to step 3.
7. Convert the no back to floating point representation as:

s + exp(converted to 8-bit) + f[2….24]

## Special cases:

1. If any number is NaN, the answer is bound to be NaN
2. Else if one number is inf and one is -inf, then answer is bound to be NaN
3. Else if any one number is inf, ans is inf
4. Else if any one number is -inf, ans is -inf
5. Normalisation step for denormal no, or zero answer yields sets exp = 0

All special cases are handled appropriately in various steps of the algorithm.

# Test cases

(A) Test for corner cases with NaN, inf, -inf

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| `01111111100000000000000000000000`<br>`01100101010010000000000011111111` | `01111111100000000000000000000000` | 4 | inf + any no = inf |
| `01111111111111111111111111111000`<br>`01101010101001000000000111110011` | `01111111111111111111111111111111` | 4 | NaN + any no = NaN |
| `01111111111111111111111111111100`<br>`11111111100000000000000000000000` | `01111111111111111111111111111111` | 4 | NaN + (-inf) = NaN |
| `11111111100000000000000000000000`<br>`11111111100000000000000000000000` | `11111111100000000000000000000000` | 4 | (-inf) + (-inf) = -inf |
| `01111111100000000000000000000000`<br>`01111111100000000000000000000000` | `01111111100000000000000000000000` | 4 | inf + inf = inf |
| `11111111100000000000000000000000`<br>`01111111100000000000000000000000` | `01111111111111111111111111111111` | 4 | inf + (-inf) = NaN |

(B) Correctness for all sign combinations

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| 10111111100000000000000000000000<br>00111111000000000000000000000000 | 10111111000000000000000000000000 | 4 | -1.0 + 0.5 = -0.5 |
| 00111111100000000000000000000000<br>10111111000000000000000000000000 | 00111111000000000000000000000000 | 4 | 1.0 - 0.5 = 0.5 |
| 10111111100000000000000000000000<br>10111111000000000000000000000000 | 10111111110000000000000000000000 | 4 | -1.0 - 0.5 = -1.5 |
| 00111111100000000000000000000000<br>00111111000000000000000000000000 | 00111111110000000000000000000000 | 4 | 1.0 + 0.5 = 1.5 |

(C) Overflow

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| 01111111011111111111111111111111<br>01111111011111111111111111111111 | Overflow | 3 | 1.111...1 * 2^127 + 1.11111 * 2^127 |
| 01111111011111111111111111111111<br>01111111011111111111111111111111 | Overflow | 3 | -1.111...1 * 2^127 + -1.11111 * 2^127 |
| 01111111011111111111111111111111<br>01110011000000000000000000000000 | Overflow | **5** | 1.111...1 * 2^127 + 1.0 * 2^103<br><br>**Overflows after rounding up** |
| 01111111010000001010101000010000<br>01111110100000000000000000000000 | Overflow | 3 | 1.10000001010 10100001 * 2^127 + 1.0 * 2^126 |

(D) 6 clock cycles due to rounding up

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| 00111111111111111111111111111111<br>00110011100000000000000000000000 | 01000000000000000000000000000000 | 6 | 1.111...1 * 2^0 + 1.0 * 2^-24 - 6 |
| 00111111111111111111111111111111<br>00110011111111111111111111111111 | 01000000000000000000000000000000 | 6 | 1.111...1 * 2^0 + 1.11...1 * 2^-24 |

(E) Handling denormal numbers

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| 00000000000000000000000000000010<br>10000000000000000000000000000001 | 00000000000000000000000000000001 | 4 | Subtraction |
| 00000000000000000000000000000010<br>00000000000000000000000000000001 | 00000000000000000000000000000011 | 4 | Addition |
| 00000000010000000000000000000000<br>00000000011000000000100000000001 | 00000000101000000000100000000001 | 4 | Sum of denormal numbers yielding a normal number |
| 00000000100000000000000000000010<br>10000000100000000000000000000001 | 00000000000000000000000000000001 | 4 | Difference of normal numbers yielding a denormal number |

(F) Cases involving zero

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| 00010111111111111100000000111000<br>00000000000000000000000000000000 | 00010111111111111100000000111000 | 4 | Adding zero to a non-zero |

| | | | |
|---|---|---|---|
| | | | number |
| 10111111111111011111111111111111 00111111111111011111111111111111 | 00000000000000000000000000000000 | 4 | Adding number to its negation |
| 00000000000000000000000000000000 00000000000000000000000000000000 | 00000000000000000000000000000000 | 4 | 0.0 + 0.0 |
| 10000000000000000000000000000000 00111111100000000000000000000000 | 00111111100000000000000000000000 | 4 | Adding (-0.0) to a number |
| 10000000000000000000000000000000 10000000000000000000000000000000 | 10000000000000000000000000000000 | 4 | (-0.0) + (-0.0) |

(E) Miscellaneous

| Input | Output | Clock cycles | Comments |
|---|---|---|---|
| 01111111011111111111111111111111 01110010100000000000000000000000 | 01111111011111111111111111111111 | 4 | 1.111...1 * 2^127 + 1.0 * 2^102, no overflow |
| 00111111100000000000000000000000 00111111100000000000000000000000 | 01000000000000000000000000000000 | 4 | Normalisation by right shift |
| 10111111101000000000000000000000 00111111100000000000000000000000 | 10111110100000000000000000000000 | 4 | Normalisation by left shift |
| 00101001010100101000100101001011 01010101010010101111111110100100 | 01010101010010101111111110100100 | 4 | Random |
| 10101010100100101011111011110101 01010010000000010101010101000000 | 01010010000000010101010101000000 | 4 | Random |
| 11111000000000001101000001010101 00000011101000101010111110100000 | 11111000000000001101000001010101 | 4 | Random |
| 01111000000000011111110000000111 11110000000011111111111100000000 | 01111000000000011111101101111000 | 4 | Random |