Computer Architecture (COL 216)

Assignment 11: Floating Point Addition

Soham Gaikwad (2018CS10394)

Usage

To build the program, use: **g++ floating_point_addition.cpp**

Then to run, use: **./a.out {input file} {output file}**

Additional information is printed to console.

Implementation

Handling inputs

All types of inputs are handled:

- **Zero inputs:** If any of the numbers is a zero (all bits zero) then result can be calculated in just 1 clock cycle.
- **NaN:** If any of the numbers in nan, then output is nan. Clock cycles required is 1.
- **+/-INF:** If there is addition of positive and negative inf numbers then result is nan, otherwise it is +/- inf. Clock cycles required is 1.
- **Denormalized:** Any denormalized number is handled as a normalized number after making its exponent to -126.
- **Normalized:** Results for normalized numbers is calculated by following the algorithm given in the book.

The steps for adding normalized numbers are:

1. Compare exponents
2. Add significands
3. Normalize
4. Rounding (Go back to step 3 if result is not normalized after rounding)

Rounding

The rounding scheme uses Guard bit, Round bit, and Sticky bit as described in the book. Guard and round are two extra bits in significand which store values which are right-shifted. Sticky bit is 1 if atleast 1 bit after GR bits is one. Round up is done by adding 1 to the significand and nothing is required to be done for round down. This follows the IEEE standard for rounding.

GRS	**Rounding scheme**
0xx	round down
100	Tie breaker: round up if the mantissa's bit just before G is 1, else round down
101	round up
110	round up
111	round up

Code

Since size of the numbers is known (32 bits), implementation was done using C++’s bitset for ease of readability and faster computation. Macros are defined for ease of understanding the code. This also enables to implement **double-precision or 64 bit adder with a few changes**.

```
#define SIGN_BIT 1
#define EXPONENT_BIT 8
#define FRACTION_BIT 23
#define GUARD_BIT 1
#define ROUND_BIT 1
#define STICKY_BIT 1

bitset<SIGN_BIT+EXPONENT_BIT+FRACTION_BIT> num1_bitset;
```

Significands are represented as:

```
<1.><fraction bit><guard bit><round bit><sticky bit>
1 bit    23 bit      1 bit      1 bit       1 bit
```

Result significand has an extra bit at start for carry.

```
bitset<2+FRACTION_BIT+GUARD_BIT+ROUND_BIT+STICKY_BIT> result_significand;
```

Testing

The correctness of the results was checked by matching with the results computed by the C++’s inbuilt adder. For all normalized inputs which don’t result in overflow or underflow, *"(Matched with C++'s adder!)"* or *"(Oops! Didn't match with C++'s adder.)"* is printed.

inp.txt **contains many test cases covering all edge-cases and special cases. See** ***input_description.txt*** **for description of each test case. All the steps of calculation and intermediate values can be seen by *setting the DEBUG macro to 1*.**

The program was also tested for **over 1 Lakh inputs** consisting of random and pre-defined custom test cases, generated using ***test_case_generater.cpp*** and gave correct outputs for all the test-cases.

test_case_generater.cpp can be customized to include more pre-defined edge-cases or random cases as well as different combinations of fractions and exponents.

```
#define RANDOM_FRACTIONS 100
#define RANDOM_EXPONENTS 20
```

*Total test cases generated = RANDOM_FRACTIONS * (RANDOM_EXPONENTS + #custom exponents) * (RANDOM_EXPONENTS + #custom exponents) * 2 + #custom test cases.*

inp_large.txt contains over 12138 test cases.

rand_inp_sp.txt contains many cases which need to be normalized again (clock cycles > 4).