Computer Architecture - CS2323. Autumn 2024
Lab-6 (Double Floating Point Arithmetic in RV64I Assembly)

Name : Ahmik Virani
Roll Number : ES22BTECH11001

Note : I have implemented extra credit

Approach:
I have tried to replicate what the textbook is covering for addition and multiplication.

In order to accommodate the mantissa and exponent bits, I am creating mask, and for checking sign, I am just check if the input number is negative/positive itself.

Along with that, for the rounding purposes, whenever I am shifting right, I am ensuring to store the bit that I have shifted into a register to ensure that I am taking care of rounding, as rounding only depends on the the Most significant bit that has been removed.

Now, for reporting the error, I have used the following github repository as reference to write my error message when the number of bits of mantissa + exponent > 63.
Credits: https://gist.github.com/argoc/77ffe85738c12d4bf178ccdb935d1aae

Note that after my program finishes execution, I run an infinite loop as taught in class so we do not access other memory/instructions which we should not use.

Addition Specifications:

Step 1: Converting the given numbers into floating point representation

Step 2 : Checking for edge cases numbers

a. X + 0 = X
b. NaN + X = NaN
c. Inf + - inf = 0
d. Inf + X = inf
e. -inf + x = -inf

Step 3: Checking which number has the smaller exponent

Step 4: Shifting right this number till exponent becomes equal

Step 5: Add the mantissa

Step 6: Normalize ensuring that any right shifting I do, I will track the bit to round, also check that you are not going out of range (if so make +/- inf)

Step 7: Add the sign bit, result mantissa and result exponent and return

Multiplication Specifications:

Step 1: Converting the given numbers into floating point representation

Step 2 : Checking for denormal numbers

a. NaN * x = NaN

b. Inf * (+/- ve number) = +/- Inf
c. -Inf * (+/- ve number) = -/+ Inf
d. X * 0 = 0
e. Int * 0 = NaN

Step 3: Adding the exponent
Step 4: Calculate the sign bit, by checking sign of both numbers
Step 5: Multiplying the mantissa
Step 6: Normalize ensuring that any right shifting I do, I will track the bit to round, and check overflow
Step 7: Add the sign bit, result mantissa and result exponent and return

Test Cases:

The first 3 test cases I tested were from the lab problem statement itself.

```
.data
.dword 11, 52
.dword 3
.dword 0x3fd2c00000000000, 0x40a22676f31205e7
.dword 0x409fa0cf5c28f5c3, 0x4193aa8fc4f0a3d7
.dword 0x40e699d2003eea21, 0x420e674bcb5a1877
```

0x0000000010000…	0x43057929844f64ac
0x0000000010000…	0x420e675171ce9887
0x0000000010000…	0x4243700f85975d74
0x0000000010000…	0x4193aaaf65c00000
0x0000000010000…	0x4085451364d91eeb
0x0000000010000…	0x40a2270cf31205e7

Then I started testing some other cases:

1. Since all the number in the problem statement were positive, I tested 3 other cases
    a. positive , negative
    b. Negative, positive,
    c. Negative, negative

```
.data
.dword 11, 52
.dword 3
.dword 0xbfd2c00000000000, 0x40a22676f31205e7
.dword 0x409fa0cf5c28f5c3, 0xc193aa8fc4f0a3d7
.dword 0xc0e699d2003eea21, 0xc20e674bcb5a1877
```

0x0000000010000…	0x0000000000000000
0x0000000010000…	0x43057929844f64ac
0x0000000010000…	0xc20e675171ce9887
0x0000000010000…	0xc243700f85975d74
0x0000000010000…	0xc193aa70242147b0
0x0000000010000…	0xc085451364d91eeb
0x0000000010000…	0x40a225e0f31205e7

2. Next we need to check arithmetic with 0 (both positive and negative)

```
.data
.dword 11, 52
.dword 4
.dword 0x0, 0xc193aa8fc4f0a3d7
.dword 0xc0e699d2003eea21, 0x0
.dword 0x8000000000000000, 0x40a22676f31205e7
.dword 0x40a22676f31205e7, 0x8000000000000000
```

0x0000000010000…	0x8000000000000000
0x0000000010000…	0x40a22676f31205e7
0x0000000010000…	0x8000000000000000
0x0000000010000…	0x40a22676f31205e7
0x0000000010000…	0x8000000000000000
0x0000000010000…	0xc0e699d2003eea21
0x0000000010000…	0x8000000000000000
0x0000000010000…	0xc193aa8fc4f0a3d7

3. Next lets do arithmetic with NaN : Any arithmetic with NaN should give a NaN

```
.data
.dword 11, 52
.dword 5
.dword 0xffffffffffffffff, 0xfff94beef5fee33f
.dword 0x0, 0xfff94beef5fee33f
.dword 0xfff94beef5fee33f, 0x40a22676f31205e7
.dword 0xfff94beef5fee33f, 0x8000000000000000
.dword 0xfff0000000000000, 0x7ff94beef5fee33f
```

0x0000000010000…	0xfff94beef5fee33f
0x0000000010000…	0x7ff94beef5fee33f
0x0000000010000…	0x7ff94beef5fee33f
0x0000000010000…	0xfff94beef5fee33f
0x0000000010000…	0xfff94beef5fee33f
0x0000000010000…	0xfff94beef5fee33f
0x0000000010000…	0xfff94beef5fee33f
0x0000000010000…	0xfff94beef5fee33f
0x0000000010000…	0x7fffffffffffffff
0x0000000010000…	0xffffffffffffffff

4. Lets check infinity

```
.data
.dword 11, 52
.dword 5
.dword 0xFFF0000000000000, 0x7FF0000000000000
.dword 0x7FF0000000000000, 0x10
.dword 0x7FF0000000000000, 0x90F0000850000000
.dword 0xFFF0000000000000, 0xFFF0000000000000
.dword 0x7FF0000000000000, 0x7FF0000000000000
```

0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0xfff0000000000000
0x0000000010000…	0xfff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0xfff0000000000000
0x0000000010000…	0x0000000000000000

5. Now check x+ (-x) = 0 case

```
.data
.dword 11, 52
.dword 1
.dword 0x3fd2c00000000000, 0xbfd2c00000000000
```

0x0000000010000…	0xbfb5f90000000000
0x0000000010000…	0x0000000000000000

6. Lets check overflow:

```
.data
.dword 11, 52
.dword 4
.dword 0xffe0000000000020, 0xffe0000000000020
.dword 0x7fe0000000000010, 0x7fe0000000000020
.dword 0xffe0000000000020, 0x7fe0000000000020
.dword 0x7f70000000000020, 0x7f70000000000010
```

0x0000000010000…	0x0000000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7f80000000000018
0x0000000010000…	0xfff0000000000000
0x0000000010000…	0x0000000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0xfff0000000000000

7. Check inf * 0 cases

```
.data
.dword 11, 52
.dword 4
.dword 0xfff0000000000000, 0x0
.dword 0x0, 0x7ff0000000000000
.dword 0xfff0000000000000, 0x8000000000000000
.dword 0x8000000000000000, 0x7ff0000000000000
```

0x0000000010000…	0x0000000000000000
0x0000000010000…	0xfff0000000000001
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0x7ff0000000000001
0x0000000010000…	0xfff0000000000000
0x0000000010000…	0x7ff0000000000001
0x0000000010000…	0x7ff0000000000000
0x0000000010000…	0xfff0000000000001
0x0000000010000…	0xfff0000000000000

Extra Credit

8. Suppose error

```
.data
.dword 11, 53
.dword 5
.dword 0x7ff0000000000000, 0xfff0000000000000
.dword 0x7ff0000000000000, 0x10
.dword 0x7ff0000000000000, 0x9079878098fff656
.dword 0xfff0000000000000, 0xfff0000000000000
.dword 0x7ff0000000000000, 0x7ff0000000000000
```

You can see the console:

```
Error! bits for fraction + mantissa > 63
```

9. Lets try single precision

```
.data
.dword 8, 23
.dword 1
.dword 0x42AA4000, 0xE2AA4000
```

0x0000000010000…	0x00000000e5e27220
0x0000000010000…	0x00000000e2aa4000

10. Finally lets play with denormal numbers:

```
.data
.dword 11, 52
.dword 4
.dword 0x8000000000000020, 0x8000000000000020
.dword 0x0000000000000010, 0x8000000000000020
.dword 0x8000000000000020, 0x0000000000000020
.dword 0x0000000000000020, 0x0000000000000010
```

0x0000000010000…	0x0000000000000000
0x0000000010000…	0x0000000000000030
0x0000000010000…	0x8000000000000000
0x0000000010000…	0x8000000000000000
0x0000000010000…	0x8000000000000000
0x0000000010000…	0x8000000000000010
0x0000000010000…	0x0000000000000000
0x0000000010000…	0x8000000000000040

Difficulties and what I have failed to implement and some optimizations (summary):
Whereas I have tried accommodating for denormal numbers, I could not do arithmetic for denormal and normal numbers combination, but only normal +/* normal or denormal +/* denormal but not a combination of both.

Some other difficulties I faced were to ensure that if we gave less bits, then how to handle them but ensuring that now, unlike 64 bits, I cannot directly use values like left shift by 63, or just do a negative 1 for mask, thus i ensured to store the number of bits as a global variable in the stack and always call it whenever i needed to ensure specific shifting on the number of bits for exponent/mantissa.

Also to ensure a little speed of arithmetic, I ensured that multiplying by zero is another branch which ensures that we can directly jump to the result instead of going through the big loop for a trivial answer.

How I am implementing rounding : Just checking the bit which i have just shifted is 0 or 1, if 1 then I add that to the mantissa, else I dont add it.