Machine Problem 1

Floating Point Calculator

Implement a single precision floating point calculator using MIPS. The calculator can perform the four basic operations: addition, subtraction, multiplication, and division. The two 32-bit inputs are pre-stored at the addresses **0x10010000** and **0x10010004** using a combination of the .data and .float directives, as shown below:

```
.data           // the following lines are to be stored in the data memory
.float          // the following numbers are float
        a: 5.5  // 0x40b00000 (IEEE-754 format), stored in 0x10010000
```

The results should follow the IEEE-754 standard, and should be stored in the memory addresses **0x10010020** through **0x1001002C**. Multiplication and division operations should be performed strictly following the sequential multiplication and non-restoring division algorithms, which will be shown in the next few pages.

The calculator should be able to handle denormal numbers. Whenever an overflow occurs, the calculator should output an infinity. Moreover, NaNs should also be set as an output whenever an invalid operation is performed (e.g. $0/0$, $(+\infty)+(-\infty)$,etc). For simplicity, your calculator should simply truncate any excess bits. Shown below is a summary of special cases that need to be considered for the calculator:

Input	Result
$+\infty + \infty$ or $+\infty - (-\infty)$	$+\infty$
$-\infty + (-\infty)$ or $-\infty - \infty$	$-\infty$
$+\infty - \infty$ or $-\infty + \infty$	NaN
$+\infty \cdot +\infty$, $N \cdot +\infty$	$+\infty$
$-\infty \cdot +\infty$, $-N \cdot +\infty$	$-\infty$
$N \div 0$, $\infty \div 0$, $0 \div 0$, $\infty \cdot 0$	NaN

Table 1: Special Cases

Try to optimize your code by using subroutine calls. Subroutines are generally used for implementing procedures that are repetitive. Moreover, it is also better if the four operations have their own subroutines, which the main program will call.

Of course, you are not allowed to use any floating point instructions, as well as pseudo-instructions. As mentioned earlier, all multiplication and division operations are to be implemented using their respective algorithms. You may, however, use the floating point instructions for verification purposes. Results from these operations will be available in the *Coproc 1* tab beside the *Registers* tab. You may then store these results in the memory addresses **0x10010040** through **0x1001004C** for easy comparison.

Take note, however, that the MIPS floating point instructions perform rounding off, while your calculator simply truncates the excess bits. This may cause some discrepancies between your result and the one generated by the MIPS floating point instructions.

Test cases will be provided on the checking day itself. Grading will be based on how many test cases your calculator was able to solve correctly.

Sequential Multiplication Algorithm (4-bit example)

Operand A = 1 0 1 1 **Operand B =** 1 1 0 1

Product (with carry-out)	**Procedure**
0 0 0 0 0 0 0 0 0	Initialize
0 0 0 0 0 **1 1 0 1**	Load operand B to Product Low
0 0 0 0 0 1 1 0 **1**	Check last bit of product, if 1, add Operand A to
+	Product High; else add zero to Product High.
1 0 1 1	
---	
0 1 0 1 1 1 1 0 1	Shift right, shift in bit is 0. Discard shift out bit.
0 0 1 0 1 1 1 1 0	
0 0 1 0 1 1 1 1 **0**	Check last bit of product, if 1, add Operand A to
+	Product High; else add zero to Product High.
0 0 0 0	
---	
0 0 1 0 1 1 1 1 0	Shift right, shift in bit is 0. Discard shift out bit.
0 0 0 1 0 1 1 1 1	
0 0 0 1 0 1 1 1 **1**	Check last bit of product, if 1, add Operand A to
+	Product High; else add zero to Product High.
1 0 1 1	
---	
0 1 1 0 1 1 1 1 1	Shift right, shift in bit is 0. Discard shift out bit.
0 0 1 1 0 1 1 1 1	
0 0 1 1 0 1 1 1 **1**	Check last bit of product, if 1, add Operand A to
+	Product High; else add zero to Product High.
1 0 1 1	
---	
1 0 0 0 1 1 1 1 1	Shift right, shift in bit is 0. Discard shift out bit.
0 1 0 0 0 1 1 1 1	
0 **1 0 0 0 1 1 1 1**	Done. Discard the most significant bit (carry-out)

:: Operand A (1011 = 11) x Operand B(1101 = 13) = Product (10001111 = 143)

Non-Restoring Division Algorithm (4-bit example)

Operand A = 1 1 1 0 **Operand B =** 0 0 1 1

Remainder Quotient Array	**Procedure**
0 0 0 0 0 0 0 0 0	Initialize
0 0 0 0 0 **1 1 1 0**	Load operand A to the array
	Check sign bit of array (0).
0 0 0 0 1 1 1 0 _	Shift left. If sign bit earlier was 0, subtract
+	Operand B; else, add Operand B
1 1 1 0 1	
----------	
1 1 1 1 0 1 1 0 **0**	Shift in bit is 0 if sign bit of result is 1.
	Check sign bit of array (1).
1 1 1 0 1 1 0 **0** _	Shift left. If sign bit earlier was 0, subtract
+	Operand B; else, add Operand B
0 0 0 1 1	
----------	
0 0 0 0 0 1 0 **0 1**	Shift in bit is 1 if sign bit of result is 0.
	Check sign bit of array (0).
0 0 0 0 1 0 **0 1** _	Shift left. If sign bit earlier was 0, subtract
+	Operand B; else, add Operand B
1 1 1 0 1	
----------	
1 1 1 1 0 0 **0 1 0**	Shift in bit is 0 if sign bit of result is 1.
	Check sign bit of array (1).
1 1 1 0 0 **0 1 0** _	Shift left. If sign bit earlier was 0, subtract
+	Operand B; else, add Operand B
0 0 0 1 1	
----------	
1 1 1 1 1 **0 1 0 0**	Shift in bit is 0 if sign bit of result is 1.
	Check sign bit of array (1).
1 1 1 1 1 **0 1 0 0**	If sign bit earlier is 1, perform final addition to
+	Operand B; else, retain.
0 0 0 1 1	
----------	
0 **0 0 1 0** **0 1 0 0**	Done. Low half is Quotient. Upper is Remainder.

:: Op A (1110 = 14) / Op B(0011 = 3) = Quotient(0100 = 4), Remainder (0010 = 2)

Grading Breakdown

There will be ten (10) cases to be provided during checking period. the weight of each operation on your grade are as follows:

Operation	$^{\text{Points}}/_{\text{Case}}$	Total
Addition	0.5	5
Subtraction	0.5	5
Multiplication	2.5	25
Division	2.5	25

Points will be deducted depending on how far your output is from the expected output (which is **not** the result of the MIPS floating point instructions). For every erroneous bit in the final result of each, a 0.5-point deduction will be applied. However, the maximum deduction will be limited by the weight of each test case, i.e. 4 erroneus bits on an addition will result in a 0 for that test case, while the same number of erroneous bits on a multiplication will result in a 0.5.