CprE 381 – Computer Organization and Assembly Level Programming, Fall 2018

Lab – 7, Part B

In this part, you will learn to use the HI and LO special purpose registers and then experiment with floating point registers, instructions and finally complete a programming challenge. For simulation purposes, you may use SPIM or MARS (the java executable is provided to you on BBL).

To run SPIM, please refer to Lab 4.

To run MARS, do the following:
1. Open up a Command Prompt from the Start menu
2. Navigate to the directory containing the MARS jar file
3. Type **`java -jar Mars<version>.jar`** and hit the return key.

5. HI and LO registers:

These are *special purpose registers* used to store result of multiplication and division. They are not directly addressable. Their contents are accessed with special instruction **mfhi** and **mflo**, given below is their usage.

```
 mfhi $rd

mflo  $rd
```

where rd is your destination register name.

Why do we need these registers? Suppose we want to multiply two 3-digit numbers. What's the maximum number of digits needed? Think about multiplying the two largest numbers possible, which is 999 times 999. This result is smaller than 1000 times 1000 which is 1,000,000 and uses 7 digits as a result. Thus, 999 times 999 must have 6 digits at most.

Thus we can say that multiplying an **m** digit number by an **n** digit number results in a **m + n** digit number, at most. Thus, when we multiply two 32 bit binary numbers, the result can be, at most, 32 + 32 = 64 bits. Thus the result of a multiplication or division operation cannot be stored in a regular 32 bit MIPS register. The lower 32 bits of the result is stored in **LO** and the upper 32 bits in **HI**. For division, **LO** stores the result of the integer division (i.e., the quotient), while **HI** stores the remainder. The operation is undefined if the divisor is 0.

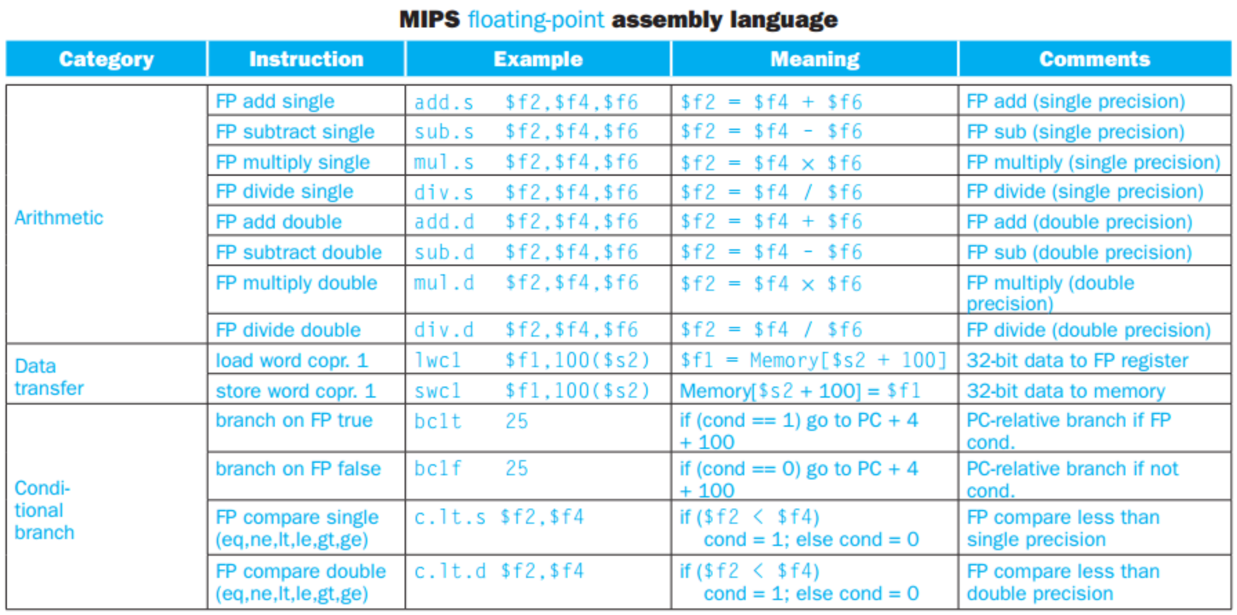
Write an interactive program that inputs 3 integers (say A,B and C) and does the following:
(i) computes their product(A*B*C) and displays the result (refer to Table 2 for system call codes)

(ii) computes their sum (A+B+C), divides the sum by the first input (SUM / A) and displays the quotient and the remainder.
Submit your code and screenshot(s) of your simulation containing the Register values.

6. Floating Point Arithmetic

The floating-point unit (also known as coprocessor 1) has 32 floating-point registers. It operates on single precision (32-bit) and double precision (64-bit) floating point numbers. Numbered as $f0, $f1, …, $f31, these registers are 32 bits wide. Thus, each register can hold one single-precision floating-point number. Using the FPU for single precision floating point numbers is straightforward. What happens to double precision floating point arithmetic which require 64 bits? The 32 single-precision registers are grouped into 16 double-precision registers. The double-precision numbers are stored in an even-odd pair of registers ($f0-$f1, $f2-$f3 and so on), but we only refer to the even-numbered register. For example, if a double precision number is stored in $f2, it is actually stored in registers $f2 and $f3.

The following table (taken from your textbook) lists some commonly used MIPS floating point instructions:



MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[$s2 + 100] = $f1	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if ($f2 < $f4) cond = 1; else cond = 0	FP compare less than double precision

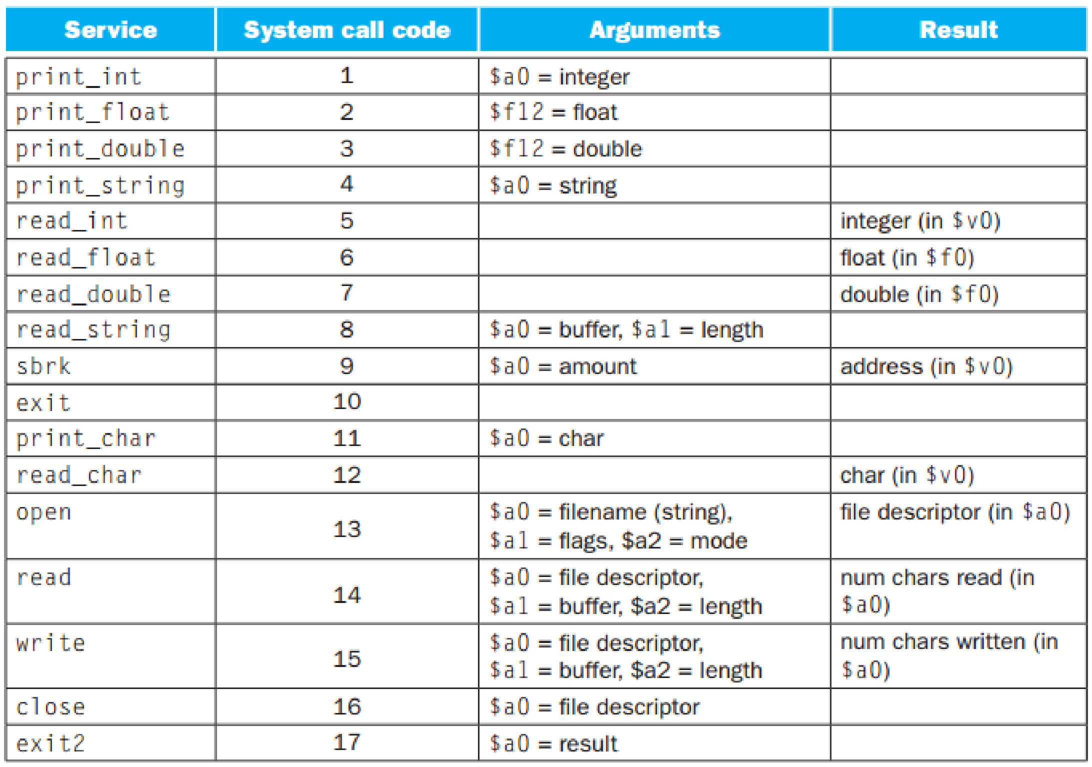
Table 1 MIPS FP instructions

Moving data from register to register:

We use the **mfc1** (move from coprocessor1) instruction to move data from a floating point register to integer register, the **mtc1** (move to coprpocessor1) instruction to move data to a floating point register and the **mov.s** to simply move data within single point floating point registers. The following examples illustrate their use.

```
mov.s $f0, $f2        #move between FP registers
mfc1 $t1, $f2         #move from FP registers
mtc1 $t1, $f2         #move to FP registers
```

Write an assembly code to multiply two (4x4) matrices A and B and store the result in C (also a matrix obviously). Populate A and B with **double precision floating point** values. The multiplication should happen in column-major order. Use **system calls** for printing your result to the console and for taking inputs from user. Submit your code and also a screenshot of your SPIM or MARS simulation (register values). Refer to page 215 of your textbook for an example. Table 2 below (taken from your textbook) lists system call codes for reference.



Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

FIGURE A.9.1 System services.

Table 2 System Call Codes

Submission

All submissions are through Canvas. You may edit this document with your answers or create a separate document. If you are submitting multiple files, their names must be self-explanatory. Put all files into a separate part b zip file before uploading.