CA P1 Lab Week 3

I) Floating-Point Instructions:

The floating-point instructions are similar to the integer instructions, however, the floating-point register must be used with the floating-point instructions. Specifically, this means the architecture does not support the use of integer registers for any floating- point arithmetic operations.

When single-precision (32-bit) floating-point operation is performed, the specified 32-bit floating point register is used. When a double-precision (64-bit) floating-point operation is performed, two 32-bit floating-point registers are used; the specified 32-bit floating-point register and the next numerically sequential register is used by the instruction. For example, a double-precision operation using $f12 will use automatically $f12 and $f13.

II) Floating-Point Data Movement:

To support the loading of data from memory into floating-point registers and storing of data in floating-point registers to memory, there are a series of specialized load and store instructions. The basic format is the same as the integer operations, however the type is either ".s" for single precision 32-bit IEEE floating-point representation or ".d" for double-precision 64-bit IEEE floating-point representation.

Instruction	Description
`l.<type>  FRdest, mem`	Load value from memory location memory into destination register.
`s.<type>  FRsrc, mem`	Store contents of source register into memory location.
`mov.<type>  Frdest, FRsrc`	Copy the contents of source register into the destination register.

Exercise 1: Scan and Print Float.

```
.data
val1:
.float 0.001
.text
main:
l.s $f12, val1
li $v0,2
syscall
li $v0, 6
syscall
mov.s $f12, $f0
li $v0,2
syscall
li $v0,10
syscall
```

Exercise 2 (Part 1) : Use double precision using l.d and mov.d instead of l.s and mov.s.

```
.data
val1:
.double 0.001
.text
main:
l.d $f12, val1
li $v0,3
syscall
li $v0, 7
syscall
mov.d $f12, $f0
li $v0,3
syscall
li $v0,10
syscall
```

III) Integer / Floating-Point Conversion Instructions:

When data is moved between integer and floating-point registers, the data representation must be addressed. For example, when moving an integer value from an integer register into a floating-point register, the data is still represented as an integer value in two's complement. Floating-point operations require an appropriate floating-point representation (32-bit or 64-bit). When data is moved between integer and floating-point registers, a data conversion would typically be required. The general format for the conversion instructions is as follows:

Instruction	Description
`cvt.d.s FRdest, FRsrc`	Convert the 32-bit floating-point value in register FRsrc into a double precision value and put it in register FRdest.

Instruction	Description
`cvt.d.w FRdest, FRsrc`	Convert the 32-bit integer in register FRsrc into a double precision value and put it in register FRdest.
`cvt.s.d FRdest, FRsrc`	Convert the 64-bit floating-point value in register FRsrc into a 32-bit floating-point value and put it in register FRdest.
`cvt.s.w FRdest, FRsrc`	Convert the 32-bit integer in register FRsrc into a 32-bit floating-point value and put it in register FRdest.
`cvt.w.d FRdest, FRsrc`	Convert the 64-bit floating-point value in register FRsrc into a 32-bit integer value and put it in register FRdest.
`cvt.w.s FRdest, FRsrc`	Convert the 32-bit floating-point value in register FRsrc into a 32-bit integer value and put it in register FRdest.

Exercise 2 (Part 2): Also use cvt.d.s and cvt.s.d to convert across precession types.

```
.data
val1:
.double 7.10
val2:
 .float 2.53
.text
main:
#Display double 7.10
l.d $f12, val1
li $v0,3
syscall

#Display float 2.53
l.s $f12, val2
li $v0,2
syscall

#Convert float to double
#cvt.d.s $f12,$f14
#li $v0,2
#syscall

#Convert double to float
#cvt.s.d $f12,$f14
#li $v0,3
#syscall

li $v0,10
syscall
```

IV) Floating-Point Arithmetic Operations:

The arithmetic operations include addition, subtraction, multiplication, division, remainder (remainder after division), logical AND, and logical OR. The general format for these basic instructions is as follows:

Instruction	Description
`add.<type> FRdest, FRsrc, FRsrc`	FRdest = FRsrc + FRsrc
`sub.<type> FRdest, FRsrc, FRsrc`	FRdest = FRsrc - FRsrc
`mul.<type> FRdest, FRsrc, FRsrc`	FRdest = FRsrc * FRsrc
`div.<type> FRdest, FRsrc, FRsrc`	FRdest = FRsrc / FRsrc
`rem.<type> FRdest, FRsrc, FRsrc`	FRdest = FRsrc % FRsrc

Exercise 3: Add/Sub of float: use instructions like add.d and add.s; sub.s, neg.s etc. Use addition along with negation to implement subtraction.

IV) Integer Mul and div, and others: ($hi, $lo, move, mfhi, mflo, mthi, mtlo)

1) mul Rdest, Rsrc, Src Rdest = Rsrc * Src or Imm

2) div Rdest, Rsrc, Src Rdest = Rsrc / Src or Imm

3) div Rsrc1, RSrc2 — $lo = Rsrc1 / Rsrc2
$hi = Rsrc1 % Rsrc2

4) rem Rdest, Rsrc, Src — Rdest = Rsrc % Src or Imm

5) neg Rdest, Rsrc — Rdest = - Rsrc

move Rdest, Rsrc	Move the contents of Rsrc to Rdest. - The multiply and divide unit produces its result in two additional registers, **$hi** and **$lo**. These instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudo-instructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.
mfhi Rdest	Move from **$hi** - Move the contents of the hi register to register Rdest
mflo Rdest	Move from **$lo** - Move the contents of the lo register to register Rdest
mthi Rdest	Move to **$hi** - Move the contents register Rdest to the hi register. - *Note*, Co-processors have their own register sets. This instruction move values between these registers and the CPU's registers.
mtlo Rdest	Move to **$lo** - Move the contents register Rdest to the lo register. - *Note*, Co-processors have their own register sets. This instruction move values between these registers and the CPU's registers.

Exercise 4: Use examples from Lab week 2 for Integer Multiply / Divide. Sample code is given below. Extend it to print the remainder as well.

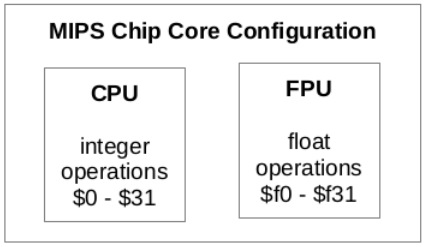
```
.data
str0:
.asciiz "\nMul:"
str1:
.asciiz "\nDiv:"

w1:
.word 300
w2:
.word 7
.text
main:
```

```
lw $t0, w1
lw $t1, w2
la $a0, str0
li $v0, 4
syscall
mul $a0, $t0, $t1
li $v0,1
# print from $a0
syscall
la $a0, str1
li $v0, 4
syscall
div $a0, $t0, $t1
li $v0,1
# print from $a0
syscall
# Modify to also print the modulo / remainder
li $v0,10
syscall
```

V) Integer / Floating-Point Register Data Movement:

The arithmetic instructions require either floating-point registers or integer registers and will not allow a combination. In order to move data between integer and floating-point registers, special instructions are required.

The following diagram shows a basic configuration of the MIPS processor internal architecture.



The FPU (floating-point unit) is also referred to as the FPU co-processor or simply co-processor 1.

Instruction	Description
`mfc1 Rdest, FRsrc`	Copy the contents from co-processor 1 (FPU) float register FRsrc into Rdest integer register.
`mfc1.d Rdest, FRsrc`	Copy the contents from co-processor 1 (FPU) float registers FRsrc and FRsrc+1 into integer registers Rdest and Rdest+1.
`mtc1 Rsrc, FRdest`	Copy the contents from integer Rsrc register to co-processor 1 (FPU) float register FRdest.
`mtc1.d Rsrc, FRdest`	Copy the contents from integer registers Rdest and Rdest+1 to co-processor 1 (FPU) float registers FRsrc and FRsrc+1.

Exercise 5: Convert Integer to Single precession float and vice versa. Also compute the conversion errors for mathematical operations – e.g. rounding off during divide, precision loss during convert to float and multiple etc. Some of the new instructions are cvt.s.w, cvt.w.s, mtc1/mtcz, mfc1/mfcz ...
Note that cvt.x.w or cvt.w.x (where we convert from / to integers) can only be done with registers in CP1. So if the value is in the main registers you first need to use mtc1. Similarly after converting to integer, you need to use mfc1 to get the values in regular registers.

Skeletal algorithm:
//A. Define two integers with relatively large values – but ensure that no overflow in multiplication
//B. multiply and print output – save integer output to a spare register
//C. Convert both integers to single precision float; multiply and print output using float [make sure to use mtc1 first and then cvt.s.w or similar
//D. Convert the output to integer and compare with saved values from step B – first convert to integer in co-processor 1 register and then move the value to a0 register of main processor.
//E. Convert both integers to double precision float; multiply and print using double precision float
//F. Convert the output to integer and compare with saved values from step B
//G. Print difference in output of D and E – (may happen only if large integer values are used)