# Lab 7

# Floating Point Operations and Macros

In this lab you will be introduced to operations using floating point, and discover the power of macros.

**Part 1.**

The MIPS CPU has 3 processors, the main processor takes care of 90% of all operations we wish to complete. The 2nd called Co-processor 1 is the floating point processor. In order to do any floating point operations, data must be moved to the floating point processor and converted to IEEE FPS standard before it can be used. This processor has 32 registers all used to hold floating point data. A double value is 64 bits, so these registers are paired, e.g. $f0 and $f1 will be used to hold a single double precision number, which can be referred to only as $f0. Let us take 2 doubles, add them together and print out the result. Here is some code to copy in:

line 2 defines a double precision number.

line 8-13 prompt for entry on a double, the result is placed in $f0.

line 15, We need to load Num1 into a register pair in co-processor 1.

line 16, add the 2 numbers together.

The syscall to print the result requires its argument in $f12, so we move $f2 to $f12.

**Part 2.**

Write a program to continuously prompt for a double, read it in, write it out, and stop if we enter a negative number. From the file menu create a new program, copy the code from part A, it is a good starting point.

Testing conditions with floating point is a 2 stage process. We first compare 2 values and set a condition flag, based on that test. Then we branch based on the condition flag. The code snippet assumes that $f4 is loaded with the value 0.0, you can do this much the same way Num1 was used to load $f2 in part 1.

Line 16, compare, sets condition flag 2 if $f0 < $f4. There are 7 condition flags which may be used, the example uses flag 2.

Line 17, branch if coporocessor 1 condition flag 2 is true.

Write the program.

**Part 3.**

Copy the above code into a new program. Modify the program to calculate the average of the numbers read in. In this case you will use an integer counter to keep track of the number of doubles read in, use $t1. In order to calculate the average we divide the sum of the numbers read in by $t1. $t1 is an integer and must be converted to a double before the divide. Thus we widen $t1. This again is a 2 step process, 1) copy $t1 to a float register. 2) change the contents of that register from a word (which is what an integer is) to IEEE standard 64 bit representation. Your algorithm should looks as follows:

Initialize $t1 to 0
Initialize $f4 to 0.0 #this will be the sum of numbers.

Read:

Read in a double
If < 0.0 then
 branch to end
inc $t1
add to $f4
branch to Read #Go get another number

end:
 convert $t1 to a double #assume to be $f6
 calc average $f4 / $f6
 print out result

**Part 4.**

It is possible to convert from a float to an integer. This is a 2 step process as well. Consider the following: This reverses the process by converting an IEEE double to a word and then storing it back to $f0. The move then allows the value to be brought back to $t1.

On the Website is the Letter Grade code. Copy it to a new program file. Modify it to prompt for a double then assign a letter grade. The modification is not hard.

**Part 5.**

Upto now the SysCalls have been a tedious endevour. Lets shorten them up by using the macro feature of MIPS. In short a macro is a symbolic substitution done prior to assembley. This allows for automation in coding and can shorten up code considerably. Consider the macro to the right: To use this we just write **done** to terminate the program in place of the usual syscall.

To the right is a macro to prompt for a double from standard input. It uses a variable called value (call it any name you want) which represents the *prompt* we normally would see. This would typically be used as:

***GetDouble(Prompt)*** where Prompt is well the prompt we defined in the code.

Modify Part 4 to include these macros. They are placed above the .data section.

**Part 6.**

These macros can be placed in a file and then imported into your program. Create a new file called ***io\_Macros.asm*** and copy the above 2 macros into that file. Modify Part 5 to use the .include statement.

You can add to the macro file as needed to print strings, integers etc. This should clean up your code considerably.

If you notice, the syntax looks remarkebly like “C” syntax, which is where macros were first made popular.