COMPUTER ORGANISATION

LAB SESSION 7

FLOATING POINT ARITHMETIC

Introduction

In this session we work with the floating point arithmetic in MIPS R2000. The tool we will use is the simulator PCSpim. Another version of the simulator named **PCSpimVISTA** can also be used for this session.

Goals

- To understand the fundamentals of processing floating point numbers in a computer.
- To handle real numbers encoded according to the IEEE 754 standard in simple and double precision
- To know how to read real numbers from main memory.
- To understand basic assembler programs that process real numbers.

Material

- The material for the session can be found in the corresponding PoliformaT folder
- Simulator: PCSpim, PCSpimVISTA.
- Source files(**`formatos.s, promedio.s, pi-leibniz.s`**).

Floating Point arithmetic in MIPS R2000

The MIPS R2000 processor was designed to work with an external Float Point Unit (FPU) named MIPS R2010. Figure 1 shows the connection between both devices and main memory.

The Float Point Unit has a Register file with 32 registers of 32 bits each, named $f01, $f1, $f2,…, $f31. But, from the programmer point of view only 16 registers are visible (for 32 bits data or 64 bits data), i.e. only the even registers can be used ($f0, $f2, $f4,…, $f30).

Numbers encoded using the standard IEEE 754 for double precision are stored in a couple of registers, while simple precision numbers only need a single register. If we consider that register $f0 contains a double precision number, then its 32 most significant bits (according to the standard) are stored in register $f1, while the less significant 32 bits are stored in $f0. In summary, when programming only the even registers are used.

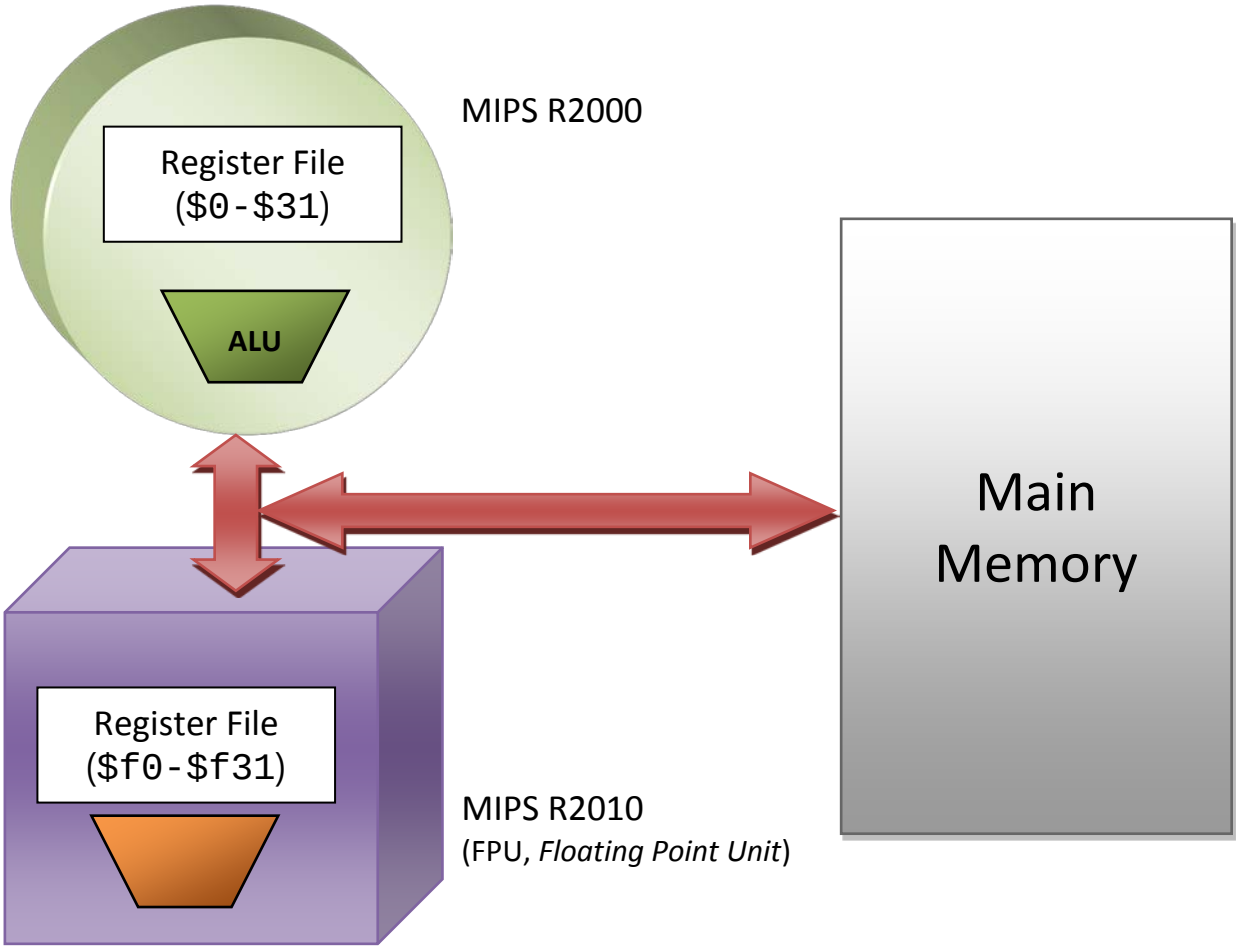


Figure 1. MIPS R2000 and the FPU R2010

As it happens with integer registers there is a convention for the use of floating point registers. This convention can be seen in the next table:

Name of register	Conventional use
$f0	Function return (real part)
$f2	Function return (imaginary part)
$f4,$f6,$f8,$f10	Temporary registers
$f12,$f14	Function parameters
$f16,$f18	Temporary registers
$f20,$f22,$f24,$f26,$f28,$f30	Registers to preserve among function calls

MIPS R2000 processor has specific instructions for data interchange between memory and floating point registers:

- **`lwc1 FPdst, Despl(Rsrc)`**
- **`swc1 FPsrc, Despl(Rsrc)`**

Where FPsrc and FPdst are registers of the FPU ($f0..$f31) and Rsrc is a register of the basic MIPS R2000 processor ($0..$31).

For example, instruction `lwc1 $f4, 0($t0)` reads the contents of memory address [$t0 + 0] and stores it in $f4, while instruction swc1 $f8, 0($t0) writes the content of $f8 in the position of main memory pointed by address [$t0 + 0]. When using double precision data two consecutive instructions `lwc1` or `swc1` are needed respectively.

The assembler language permits the use of *pseudo-instructions* to facilitate the programmer writing the code. The following *pseudo-instruction* permit to store in the FPU registers real numbers directly from main memory:

- **`li.s FPdst, No_float`** **`# Load inmediate`**
- **`li.d FPdst, No_double`**

For example, the *pseudo-instruction* **`li $f4, 2.7539`** stores in $f4 the real number 2.7539 encoded in simple precision (32 bits).

Other *pseudo-instructions* permit to directly read/write from main memory:

- **`l.s FPdst, Address`** # Load float from memory Address to FPdst
- **`l.d FPdst, Address`** # Load double from memory Address to FPsrc|FPsrc+1
- **`s.s FPsrc, Address`** # Store float (FPsrc) to memory Address
- **`s.d FPsrc, Address`** # Store double (FPsrc|FPsrc+1) to memory Address

For example, the *pseudo-instruction* **`l.d $f4, A`** reads a double precision floating point number from the memory position ´A´ and stores it in the pair of registers $f4|$f5. Variable ´A´ must be declared as:

```
A:    .double  2753.9E-3     # or another initial value
```

Or also,

```
A:    .space 8
```

Note that in this last case the variable has to be correctly aligned in an address multiple of 8.

Remember that the compiler translates the pseudo-instructions in the corresponding instructions machine.

PCSpim simulator settings

The PCSpim simulator allows the execution of MIPS R2000 assembler programs using real numbers encoded according the IEEE 754 standard.

As Figure 2 shows, the content of the floating point registers are shown in the upper part of the window. They can be interpreted as double precision numbers (64 bits) or simple precision numbers (32 bits). Note that in each case what really changes is the registers “interpretation”.

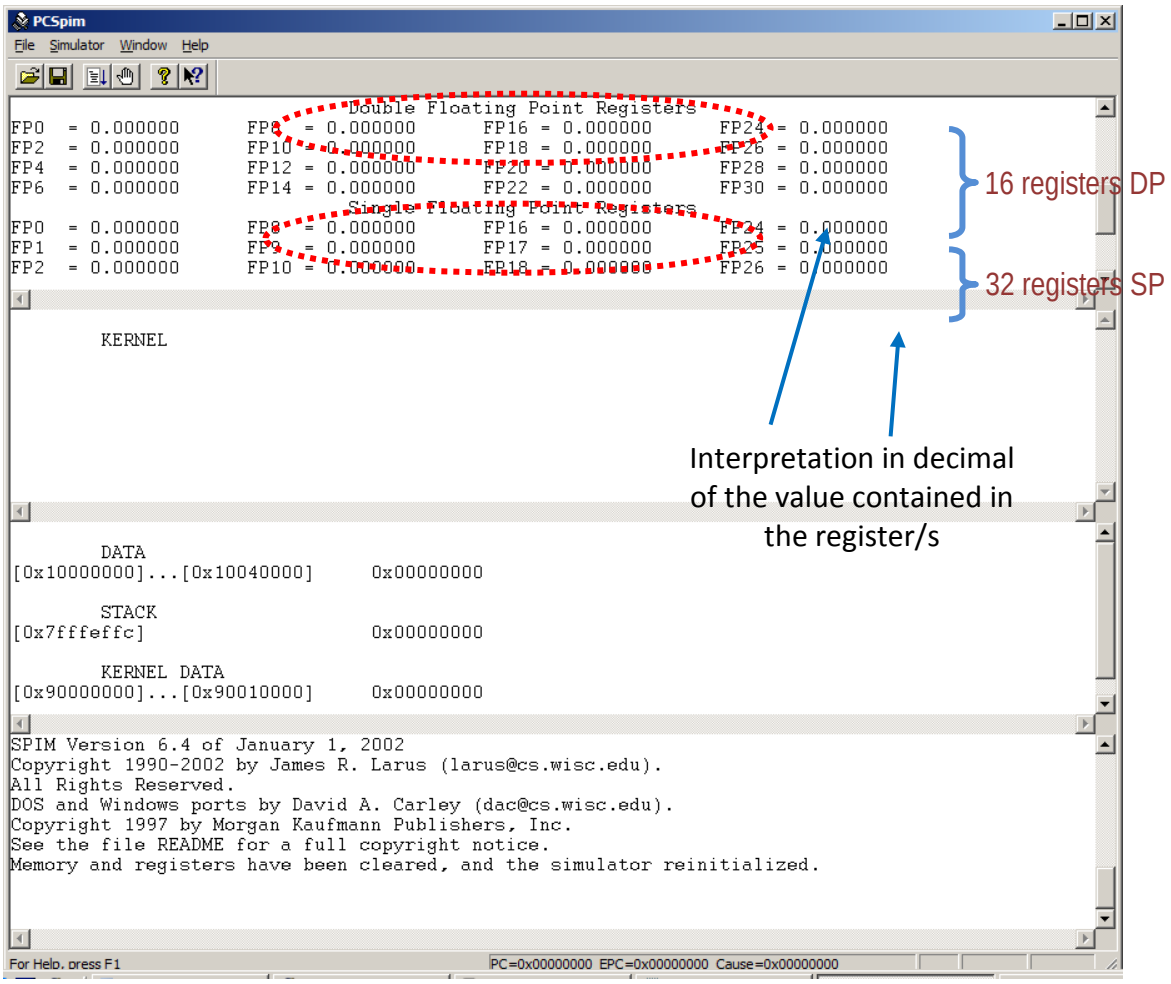


Figure 1. Registers of the floating point unit

The way how the contents of registers are shown can be selected through the menu *Simulator|Settings.* Figure 3 shows how to select the option hexadecimal to see those contents. If you don´t tick this check box, the value is shown according to the bits interpretation for the standard IEEE 754.

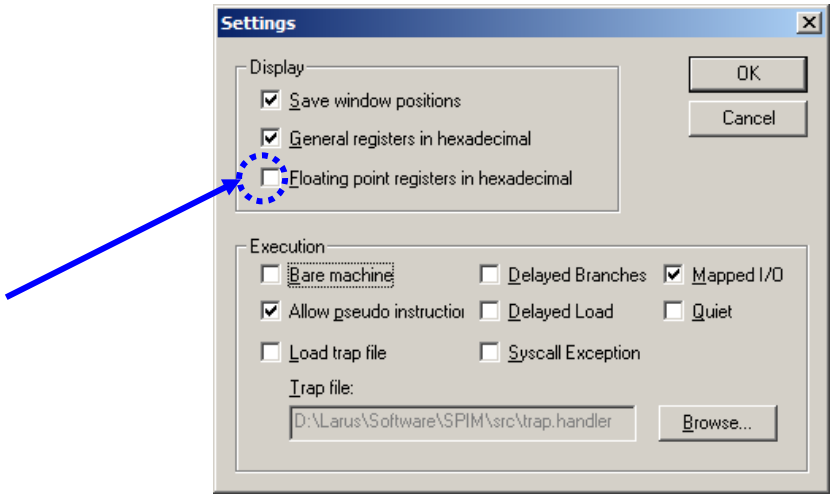


Figure 3. Settings for the interpretation of floating point registers

Floating Point representation

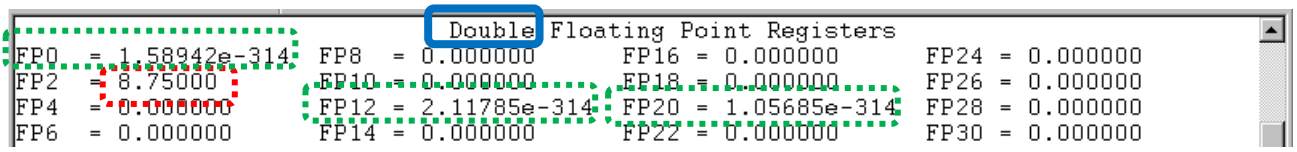
Let´s start with an example to understand how the simulator shows floating point numbers. Consider the following chunk of code:

```
                .globl __start
              .text 0x00400000

__start:        li.s $f0, -1.5          # Constant -1.5
                li.d $f2, 8.75          # Constant 8.75
                li $t0, 0xFF800000      # -∞
                mtc1 $t0, $f12          # move to $f12
                li $t1, 0x7F8003A0      # NaN
                mtc1 $t1, $f20          # move to $f20
```

The program initializes four floating point constants. The two first constants are set using two *pseudo-instruction* (*load immediate* for *simple* and *double precision*) and they are encoded in simple and double precision respectively. The number of bits used for each case is very important to understand what the simulator displays and how to interpret the contents of the registers.

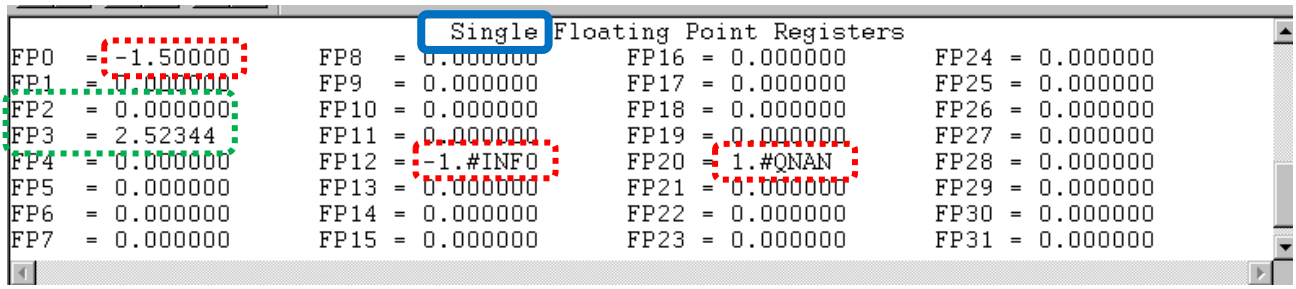
The last two constants are also set by means of a pseudo-instruction, but in this case a load immediate instruction for integer operands. We consider directly their encoding according to the standard IEEE 754 to represent negative infinity (–∞) and not a number (NaN), respectively.

The constant value -1.5 is stored in $f0, but the constant 8.75 is stored in two registers $f3|$f2. The simulator offers two different views for the floating point register file. First, it shows the interpretation supposing that the variables are encoded in double precision thus using two registers, consequently only 16 values are shown:



In this figure we can see that the value 8.75 is stored in two registers and is correctly interpreted but, the rest of constants are not correctly interpreted because they are single precision values (see registers $f0, $f12 y $f20).

The next figure shows the contents of registers for single precision numbers. Now, we can see 32 registers.



In this case, the interpretation of registers $f0, $f12 y $f20 is correct: $f0 has the value −1.5, $f12 has −∞ and $f20 stores NaN (*Not a Number*).

► Load the program *formatos.s* and run it. Check the obtained results.

► Why does the value 2.52344 appear as the contents of the $ f3 register? For an easier understanding you can visualize the contents of the registers in hexadecimal.

► How many possible representations are there for the real value 0.0 in the IEEE 754 standard for simple precision? What are those representations? Code them in hexadecimal.

► How many possible representations are there for the real value infinity (∞) in the IEEE 754 standard of simple precision? What are those representations? Encode them in hexadecimal.

► In which instructions the assembler program has translated the *pseudo-instruction* **`li.d $f2, 8.75`**. Interpret the generated code.

► Show in hexadecimal the encoding in single and double precision of the constant 78.325. Use the simulator to get both representations.

► How many different words are there in the IEEE 754 simple-precision format to represent the NaN value?

► Why is there not an instruction for immediate addition for real numbers similar to *addi*?

Arithmetic mean calculation

Below is an assembler program that calculates the arithmetic mean of a set of real values. Given *n* numbers a_0, a_1, ..., a_{n-1}, its mean value is defined by the formula:

$$\frac{1}{n}\sum_{i=0}^{n-1} a_i$$

Real values are encoded in simple precisión (*float*) and are stored in main memory from the address given by the label *valores*. The mean value is calculated both in single (media_s) and double precision (media_d) and stored in data segment according to the declaration.

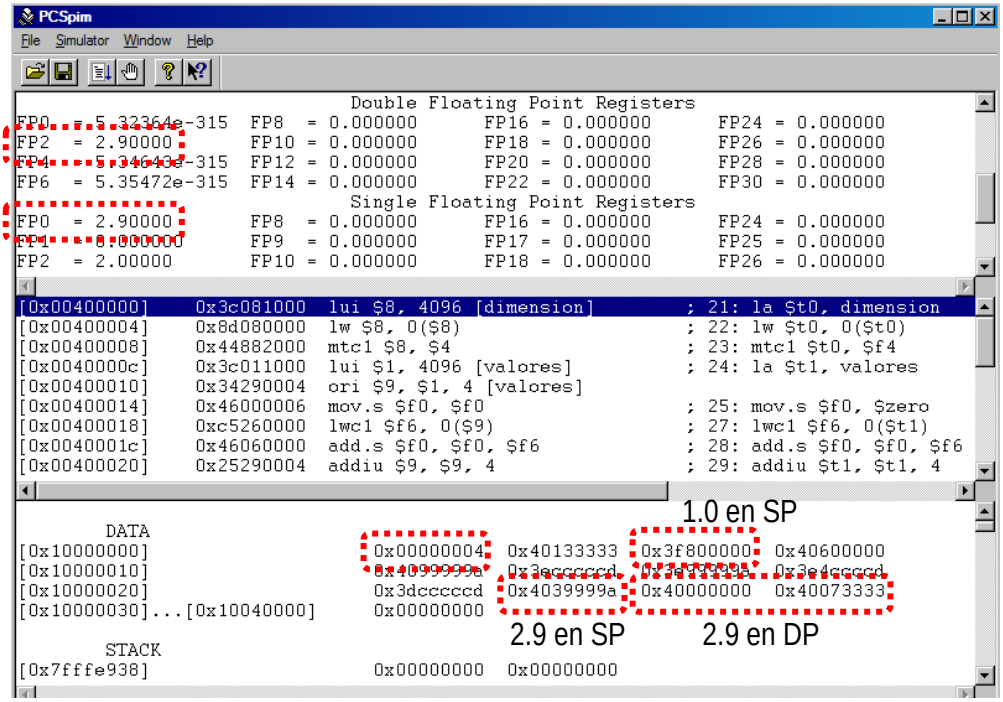
```
        ############################################################
       # Data segment
       ############################################################

                .data 0x10000000
dimension:      .word 4
valores:        .float 2.3, 1.0, 3.5, 4.8
pesos:          .float 0.4, 0.3, 0.2, 0.1
media_s:        .float  0.0
media_d:        .double 0.0

       ############################################################
        # Code segment
       ############################################################

          .globl __start
          .text 0x00400000

__start:        la $t0, dimension       # Pointer to dimension
                lw $t0, 0($t0)          # Reads dimension
                mtc1 $t0, $f4           # Moves dimension to $f4
                la $t1, valores         # Pointer to values
                mtc1 $zero, $f0         # Moves 0.0 to $f0

     bucle:     lwc1 $f6, 0($t1)        # Reads valor[i]
                add.s $f0, $f0, $f6     # Adds the value
                addiu $t1, $t1, 4       # Pointer to valor[i+1]
                addiu $t0, $t0, -1      # Decreases counter
                bgtz $t0, bucle
                cvt.s.w $f4, $f4        # Converts dimension to fp
                div.s $f0, $f0, $f4     # calculates arithmetic mean
                cvt.d.s $f2, $f0        # Converts to double
                la $t0, media_s         # Pointer to media_s
                swc1 $f0, 0($t0)        # Stores the result sp
```

```
la $t0, media_d          # Pointer to media_d
swc1 $f2, 0($t0)         # Stores low part result dp
swc1 $f3, 4($t0)         # Stores high part result dp
.end
```

► That program is stored in the file *promedio.s*. Load and execute it in the simulator. Check that the obtained results are equal to those shown in the next figure:

SP Result in `$f0`

DP result in `$f3|$f2`



Check again the code analyzing how the program works. Pay special attention to data declaration, their encoding and, the writing in memory of the results obtained.

► Explain the purpose of the **`cvt.s.w`** instruction that appears in the code.

► Complete the next table with the results obtained for single and double precision. Recall that the low part of the variable encoded in double precision is stored in the lowest memory address and the high part in the highest address.

Variable	Value in decimal	Encode using IEEE 754
`media_s`		
`media_d`		

► How many arithmetic floating-point operations and what class (addition, subtraction, type conversion, etc.) are executed in the program.

► If the runtime of this program on a real processor is 0.5 microseconds, calculate the number of floating point operations per second achieved by the processor (FLOPS, floating point operations per second). Show the result in millions of operations per second (MFLOPS).

Calculation of number π

Number π (pi) is the relationship between the length of a circle and its diameter. It is an irrational number and one of the most important mathematical constants. It is often used in mathematics, physics and engineering. The numerical value of π, truncated to its first digits, is as follows:

$$\pi \approx 3{,}14159265358979323846...$$

The value of π has been obtained using various approximations throughout history, being one of the mathematical constants that most appears in the equations of physics, together with the number *e*.

The German mathematician Gottfried Leibniz devised in 1682 a method for calculating the number π. This method performs an approximation to π / 4 through the following infinite series:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Next, you can see an assembler program that calculates number π using the series below. This code develops the series up to the value specified by the user for the program variable *n*. The main program uses the Leibniz subroutine for that calculation. Data entry and exit is carried out through system calls.

```
        ############################################################
        # Data segment
        ############################################################

                    .data 0x10000000
cad_entrada:        .asciiz "\n entry the number of iterations: "
cad_salida:         .asciiz "the value of pi is: "

        ############################################################
        # Code Segment
        ############################################################

        .globl __start
        .text 0x00400000
__start:
        ############################################################
```

```
      # Reading the number of iterations (n)
       ########################################################

                la $a0, cad_entrada # String to print
                li $v0, 4           # System call to print_string
                syscall
                li $v0, 5           # System call to read_int
                syscall
                move $a0, $v0       # Routine argument
                jal leibniz         # jump subroutine

       ########################################################
        # printing the result
       ########################################################

                la $a0, cad_salida     # String to print
                li $v0, 4              # System call to print_string
                syscall
                li $v0, 2              # System call to print_float
                mfc1 $t0, $f0          # Value to print
                mtc1 $t0, $f12
                syscall

       ########################################################
       #End program
        # Syscall "exit"
       ########################################################=

                li $v0, 10
                syscall

        ########################################################
         # Calculating  pi using Leibniz method
         # $a0 = Number of iteration
        ########################################################

leibniz:        li.s $f0, 0.0           # Constant 0.0
                li.s $f4, 1.0           # Constant 1.0
                li.s $f6, 2.0           # Constant 2.0
                move $t0, $a0           # Counter for iterations

bucle:          mtc1 $t0, $f8           # moves n to FPU
                cvt.s.w $f8, $f8        # Converts n in real number

                mul.s $f8, $f8, $f6     # Calculates 2.0*n
                add.s $f8, $f8, $f4     # Calculates 2.0*n + 1.0
                div.s $f8, $f4, $f8     # Calculates 1.0/(2.0*n + 1.0)
                andi $t1, $t0, 0x0001   # Select Less Sig. Bit of n
                bne $t1, $zero, resta   # Branch if odd (LSB==1)
                add.s $f0, $f0, $f8     # The term of the series is added
                j continua
resta:          sub.s $f0, $f0, $f8     # Subtract term
continua:       addi $t0, $t0, -1       # Decrease number of iterations
                bgez $t0, bucle         # loop control
```

```
li.s $f4, 4.0           # Constant 4.0
mul.s $f0, $f0, $f4     # Returns pi in $f0
jr $ra
.end
```

Analyze in detail the previous code. You can verify that the entire calculation of the series is carried out within the Leibniz subroutine.

► Explain how the program does to decide if the term of the series is added (n pair) or subtracted (n odd).

► How many floating-point operations are carried out in the previous program based on the number n of iterations?

► This program is in the file pi-leibniz.s. Load and run it for the different number of iterations shown below. Complete the table writing the values of π obtained for each case.

Iterations (n)	Value of π
10^3	3.1425914764
10^4	
10^5	
10^6	

MIPS R2000 architecture provides instructions for moving data between the banks of integer and floating point registers (mtc1, mfc1). There are also specific instructions to move data between floating-point registers: **`mov.s`** and **`mov.d`**. For example, **`mov.s $f4, $f2`** copies the contents of the record $ f2 to $ f4.

► Imagine for a moment that the instruction **`mov.s`** is not available in the processor architecture. What alternative instructions could be used to move the contents of the register $ f2 to $ f4?

► To move the contents from one integer register to another, the pseudo-instruction **`move`** can be used. For example, move $t0, $t1 copies the contents of $t1 to $t0. Why do you think that **`move`** has not been included in the processor as a machine instruction?

► Adapt the program to work with real numbers encoded in the IEEE 754 double precision standard (real variables of type double) and call the file **`pi-leibniz-d.s`**. Pay attention to the transfer of the final result located in the pair of registers $f1 | $f0 for printing the result. Among other things you will have to modify the system call that prints this type of variables (the index for **`print_double`** is 3). Briefly indicate the changes made with respect to the original version in simple precision.

► Run the program and complete the following Table:

Iterations (n)	Value of π
10^3	3.1425916543
10^4	
10^5	
10^6	