ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Sesión de laboratorio número 7

ARITMÉTICA DE COMA FLOTANTE

Introducción

En esta práctica se trabaja con la aritmética de coma flotante del MIPS R2000. La herramienta de trabajo es el simulador del procesador MIPS R2000 denominado **PCSpim**.

Objetivos

- Entender los fundamentos del procesamiento de número reales en un computador.
- Manipular números reales codificados mediante el estándar IEEE 754 de simple y de doble precisión.
- Conocer cómo leer de la memoria principal los números reales.
- Entender el funcionamiento de programas en ensamblador que procesan números reales.

Material

El material se puede obtener de la carpeta de recursos de PoliformaT.

- Simulador PCSpim del MIPS R2000.
- Archivos fuente (`formatos.s`, `promedio.s`, `pi-leibniz.s`).

La aritmética real en el procesador MIPS R2000

El MIPS R2000 está diseñado para trabajar con una unidad de coma flotante (FPU, *floating point unit*) externa denominada MIPS R2010. La Ilustración 1 muestra gráficamente la conexión de ambos dispositivos así como su relación con la memoria.

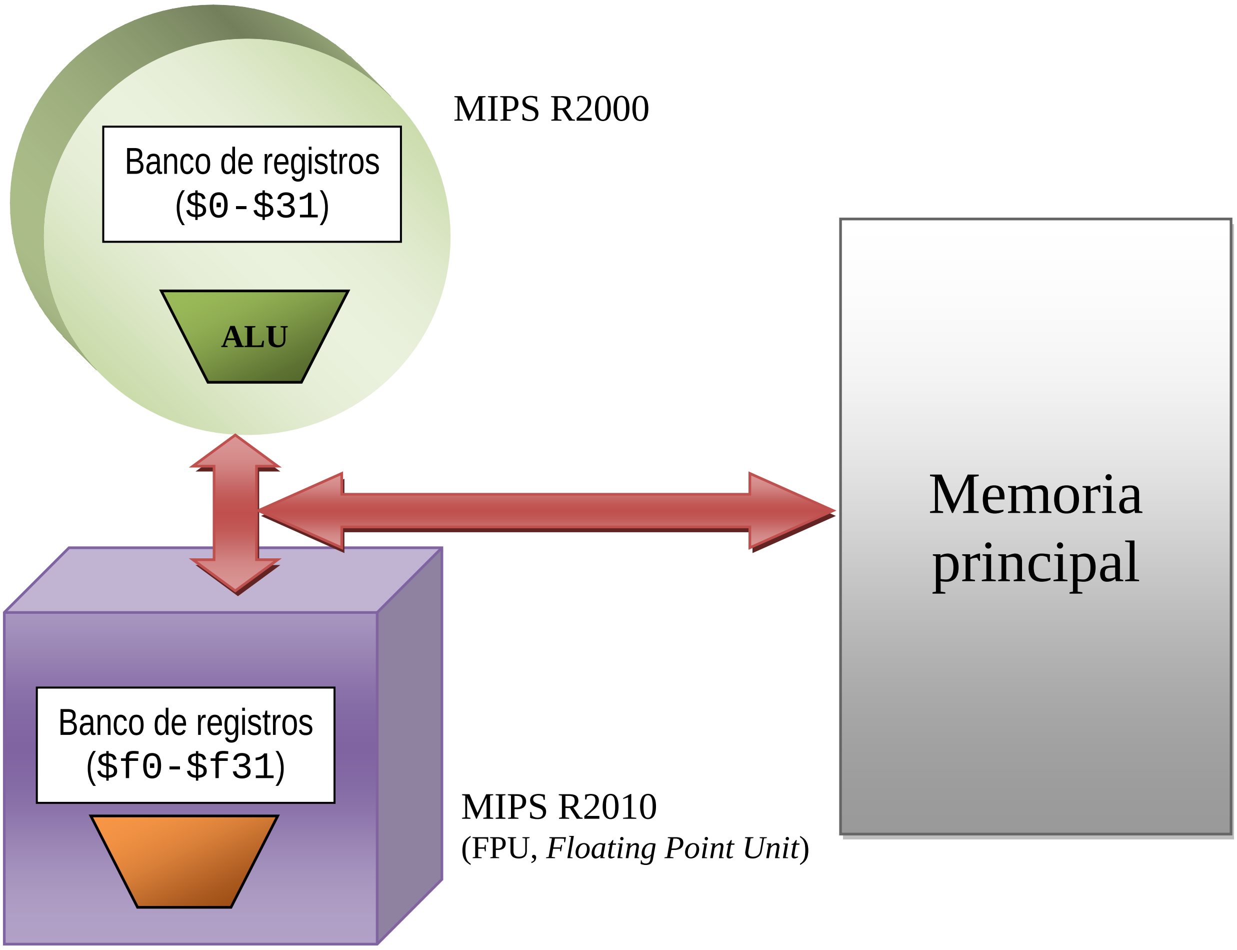


Ilustración 1 La unidad de coma flotante del procesador MIPS R2000

La unidad de coma flotante tiene un banco de 32 registros de 32 bits denominados `$f01`, `$f1`, `$f2`,…, `$f31`. Sin embargo, desde el punto de vista del programador, estos registros se ven solamente como 16 registros, bien de 64 o de 32 bits; en cualquier caso, se hace uso exclusivamente de los registros pares (`$f0`, `$f2`, `$f4`,…, `$f30`).

Los valores codificados en el formato IEEE 754 de doble precisión (DP) se almacenan en una pareja de registros, mientras que los valores de simple precisión (SP) se ubican en un único registro del banco. Si, por ejemplo, decimos que el registro `$f0` contiene un valor real de doble precisión, entonces sus 32 bits de mayor peso se almacenan en `$f1` y los 32 de menor peso en `$f0`. En definitiva y, como se ha dicho, cuando se diseñan programas solamente se hace uso explícito de los registros pares.

Como ocurre con el banco de registros de la unidad aritmético-lógica, el patrón de utilización de los registros de coma flotante por parte del programador no es arbitrario y viene establecido en la tabla siguiente:

Nombre del registro	Utilización
`$f0`	Retorno de función (parte real)
`$f2`	Retorno de función (parte imaginaria)
`$f4,$f6,$f8,$f10`	Registros temporales
`$f12,$f14`	Paso de parámetros a funciones
`$f16,$f18`	Registros temporales
`$f20,$f22,$f24,$f26,$f28,$f30`	Registros a preservar entre llamadas

El procesador MIPS R2000 dispone de las siguientes instrucciones para leer o escribir números reales en la memoria principal:

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

Donde `FPsrc` y `FPdst` son registros del coprocesador de coma flotante ($f0..$f31) y `Rsrc` es un registro del procesador base ($0..$31).

Por ejemplo, la instrucción `lwc1 $f4, 0($t0)` lee el contenido de la dirección de memoria [$t0 + 0] y lo deja en el registro `$f4`, mientras que `swc1 $f8, 0($t0)` escribe el contenido de `$f8` en memoria. Cuando las variables son de doble precisión las operaciones de lectura o escritura necesitan utilizar dos instrucciones `lwc1` o `swc1`, respectivamente.

El lenguaje ensamblador también permite usar *pseudoinstrucciones* que facilitan la escritura de los programas. Algunas de esas pseudoinstrucciones permiten introducir número reales directamente en los registros de la FPU:

- `li.s FPdst, Num_float      # Load inmediate`
- `li.d FPdst, Num_double`

Por ejemplo, la pseudoinstrucción `li $f4,` 2.7539 cargará en el registro $f4 el valor 2.7539 codificado en simple precisión (32 bits). Otras pseudoinstrucciones permiten leer o escribir de la memoria principal:

- `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`

Por ejemplo, la presudoinstrucción `l.d $f4, A` lee un número de doble precisión (8 bytes) de la dirección de la variable en memoria 'A' y lo almacena en el par `$f4|$f5`. La variable 'A' deberá estar declarada como:

A: .double 2753.9E-3   # o cualquier otro valor inicial

O también: A: `.space 8`

Pero en este caso hay que asegurarse de que la variable está correctamente alineada en una dirección múltiplo de 8.

Recuerde que las pseudoinstrucciones son traducidas por el programa ensamblador a instrucciones ejecutables por el procesador.

.float reserva memoria a partir de direcciones múltiplos de cuatro, 0, 4, 8, C (igual que .word)

.double reserva memoria a partir de direcciones múltiplos de ocho 0,8

Configuración del simulador PCSpim

El simulador PCSpim permite la ejecución de programas escritos en ensamblador del procesador MIPS R2000 para el tratamiento de números reales. Como se puede ver en la Ilustración 2, en la parte superior de la pantalla se muestra el contenido de los registros de la unidad de coma flotante. Se pueden visualizar como números de doble precisión (64 bits) o números de simple precisión (32 bits). Nótese que lo que cambia de un caso a otro es la interpretación del contenido de los registros (doble o simple precisión).

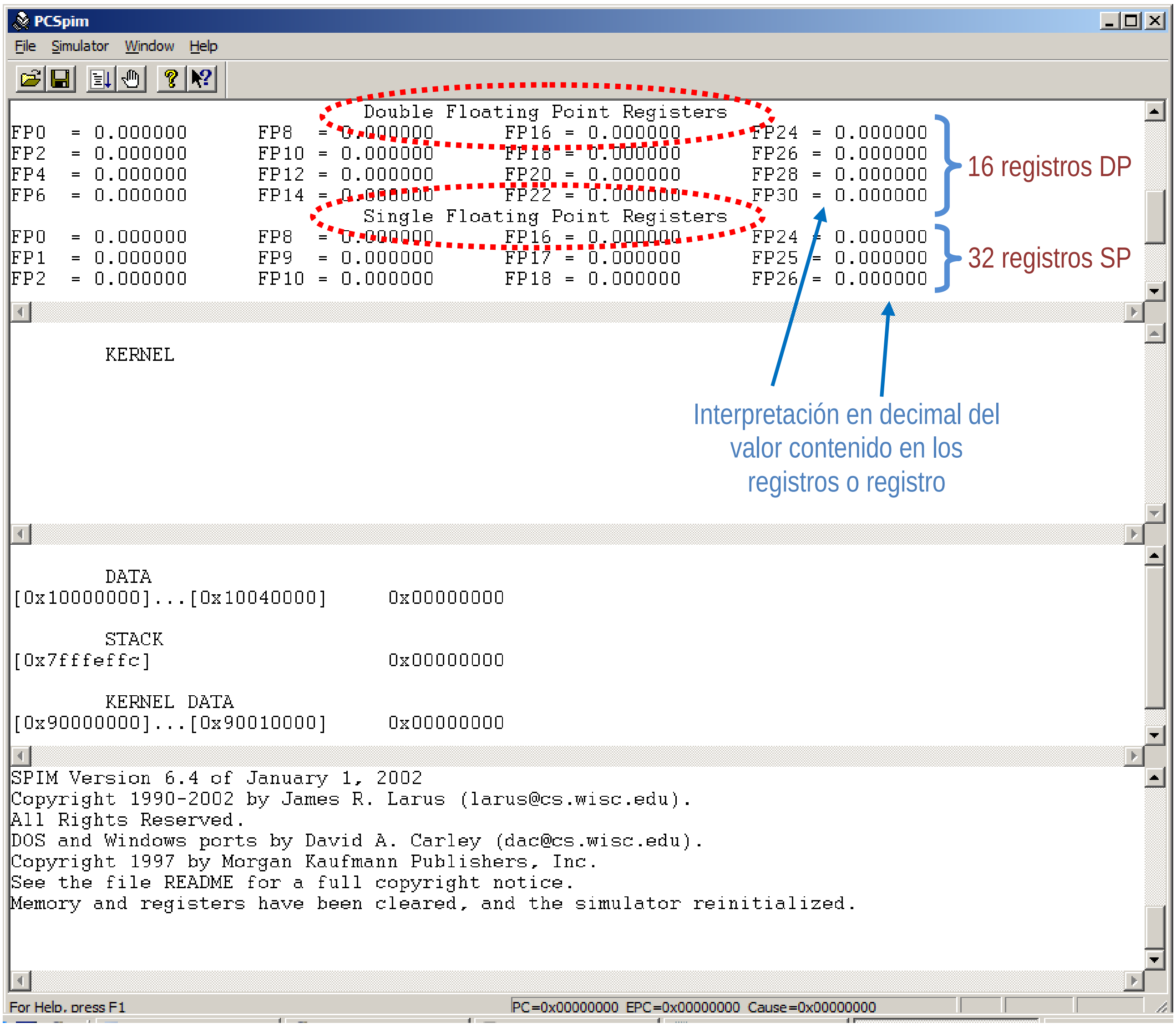


Ilustración 2 Los registros de la unidad de coma flotante

La forma en que se visualiza el contenido de cada registro se puede seleccionar en el menú *Simulator|Settings*. En este caso, según se aprecia en la Ilustración 3, si se marca la casilla señalada el contenido de los registros se muestra en hexadecimal. Si no se marca, como ocurre en nuestro caso, el contenido que se muestra es su valor de acuerdo con la interpretación de los bits según el estándar IEEE 754.

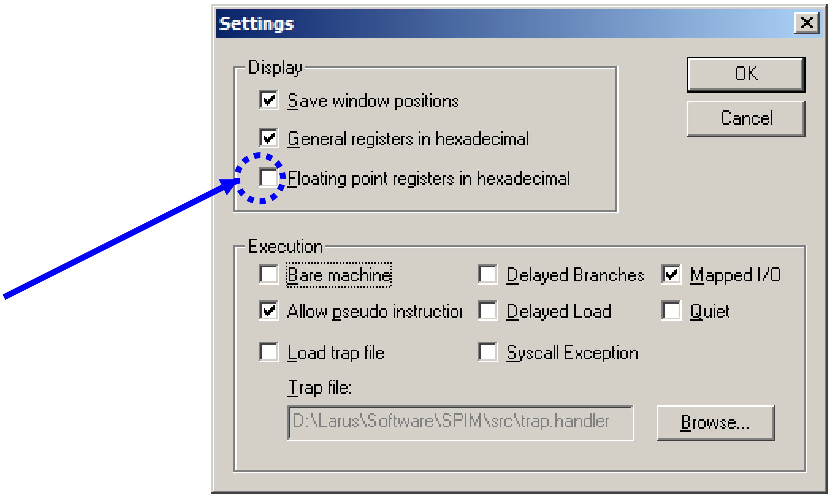


Ilustración 3 Elección de la visualización del contenido de los registros de coma flotante

Representación de los números en coma flotante

Vamos a empezar esta sesión práctica con un pequeño ejemplo para ilustrar la manera en que el simulador PCSpim visualiza los números de coma flotante. Considere el código siguiente:

```
                .globl __start
                .text 0x00400000

__start:        li.s $f0, -1.5          # Constante -1.5
                li.d $f2, 8.75          # Constante 8.75

                li $t0, 0xFF800000      # Menos infinito (-∞)
                mtc1 $t0, $f12          # Envío a $f12
                li $t1, 0x7F8003A0      # Not a Number (NaN)
                mtc1 $t1, $f20          # Envío a $f20
```

El programa crea cuatro constantes reales. Las dos primeras constantes, –1.5 y 8.75, se especifican mediante dos pseudoinstrucciones (*load immediate* para *simple* y para *double*) y se codifican en simple y doble precisión, respectivamente. El tamaño de cada variable es importante para interpretar bien lo que el simulador nos muestra en la pantalla. En cualquier caso, no olvidemos que el contenido de los registros es único: lo que cambiará será la interpretación que hagamos de su contenido.

Las dos últimas constantes se especifican directamente, también mediante pseudoinstrucciones (*load immediate* para enteros), en su codificación directa en el estándar IEEE 754 y sirven, respectivamente, para representar el valor infinito con signo negativo (–∞) y un NaN (*Not a Number*) empleado para poner de manifiesto situaciones anómalas de cálculo (por ejemplo, indeterminaciones del tipo ±0/±0, ±0× ±∞, etc.).

La constante –1.5 se guarda en `$f0`, pero la constante 8.75 utiliza los registros `$f3|$f2`. El simulador nos ofrece dos vistas complementarias del banco de registros de la unidad de coma flotante. En primer lugar nos muestra la interpretación que hace suponiendo que las variables son de doble precisión y cada una de ellas ocupa dos registros; en consecuencia, solamente veremos 16 valores:

```
                        Double Floating Point Registers
FP0  = 1.58942e-314   FP8  = 0.000000       FP16 = 0.000000       FP24 = 0.000000
FP2  = 8.75000        FP10 = 0.000000       FP18 = 0.000000       FP26 = 0.000000
FP4  = 0.000000       FP12 = 2.11785e-314   FP20 = 1.05685e-314   FP28 = 0.000000
FP6  = 0.000000       FP14 = 0.000000       FP22 = 0.000000       FP30 = 0.000000
```

(Ocupa los registros $2 y $3)

En la figura vemos que, en efecto, 8.75 ocupa dos registros. Sin embargo, el resto de valores almacenados como valores de simple precisión no se interpretan correctamente (véanse los registros `$f0`, `$f12` y `$f20`).

Un poco más abajo nos muestra la información suponiendo que los registros contienen variables de simple precisión. Así pues, veremos 32 valores:

```
                        Single Floating Point Registers
FP0  = -1.50000       FP8  = 0.000000       FP16 = 0.000000       FP24 = 0.000000
FP1  = 0.000000       FP9  = 0.000000       FP17 = 0.000000       FP25 = 0.000000
FP2  = 0.000000       FP10 = 0.000000       FP18 = 0.000000       FP26 = 0.000000
FP3  = 2.52344        FP11 = 0.000000       FP19 = 0.000000       FP27 = 0.000000
FP4  = 0.000000       FP12 = -1.#INF0       FP20 = 1.#QNAN        FP28 = 0.000000
FP5  = 0.000000       FP13 = 0.000000       FP21 = 0.000000       FP29 = 0.000000
FP6  = 0.000000       FP14 = 0.000000       FP22 = 0.000000       FP30 = 0.000000
FP7  = 0.000000       FP15 = 0.000000       FP23 = 0.000000       FP31 = 0.000000
```

En este caso la interpretación del contenido de los registros `$f0`, `$f12` y `$f20` es correcta: vemos que el primero contiene −1.5, el segundo −∞ y el tercero NaN (*Not a Number*).

► Cargue el programa anterior (fichero `formatos.s`) y ejecútelo en el simulador. Compruebe que los resultados obtenidos coinciden con los mostrados en las figuras anteriores.

► ¿Por qué aparece el valor 2.52344 como contenido del registro `$f3`? Puede ayudarse con el simulador visualizando el contenido de los registros en hexadecimal.

Si visualizamos los registros en hexadecimal vemos que FP3 (simple precisión) = segunda mitad de FP2 (doble precisión). Por lo tanto, ese valor representa los 32 bits de menor peso del número 8.75.

► ¿Cuántas representaciones posibles hay para el valor real 0.0 en el estándar IEEE 754 de simple precisión? ¿Cuáles son esas representaciones? Exprésela en hexadecimal.

Hay +0 y -0.

+0 -> 0x00000000. -0 -> 0x80000000

► ¿Cuántas representaciones hay para el valor infinito (∞) en el estándar IEEE 754 de simple precisión? ¿Cuáles son esas representaciones? Exprésela en hexadecimal.

+ y - + = 0x7F800000 - = 0xFF800000

► Indique en qué instrucciones ha traducido el programa ensamblador la pseudoinstrucción del programa `li.d $f2, 8.75`. Interprete el código generado.

```
0x34010000  ori $1, $0, 0
0x44811000  mtc1 $1, $f2
0x3c014021  lui $1, 16417
0x34218000  ori $1, $1, -32768
0x44811800  mtc1 $1, $f3
```

► Indique en hexadecimal la representación en simple y doble precisión de la constante 78.325. Ayúdese del simulador para obtener las dos representaciones.

Doble precisión -> li.d $f2, 78.325 ->cccccccd 405394cc

Simple precisión -> li.s $f4, 78.325 -> 0x429CA666

► ¿Cuántas palabras diferentes existen en el formato del estándar IEEE 754 de simple precisión para representar el valor NaN?

Según la norma IEEE 754 los casos de NaN se pueden representar rellenando el campo de exponente con unos y la mantisa con algunos número diferentes de cero. Un ejemplo en simple precisión sería:

x1111111axxxxxxxxxxxxxxxxxxxxxxx, siendo x cualquier número.

► ¿Por qué no existe una instrucción de suma de números reales similar a `addi`?

Cálculo de la media aritmética

A continuación se presenta un programa escrito en ensamblador que calcula la media aritmética de un conjunto de valores reales. Dados n números $a_0, a_1, \ldots, a_{n-1}$, su promedio se define mediante la fórmula:

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

Los números reales se codifican mediante variables de simple precisión (*float*) y se ubican en memoria a partir de la etiqueta `valores`. El valor del promedio se calcula tanto en simple precisión (`media_s`) como en doble precisión (`media_d`) y se almacena en el segmento de datos.

```
                ##########################################################
                # Segmento de datos
                ##########################################################

                .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

                ##########################################################
                # Segmento de código
                ##########################################################

                .globl __start
                .text 0x00400000

__start:        la $t0, dimension       # Dirección de la dimensión
                lw $t0, 0($t0)          # Lectura de la dimensión
                mtc1 $t0, $f4           # Lleva la dimensión a $f4
                la $t1, valores         # Dirección de los valores
                mtc1 $zero, $f0         # Lleva 0.0 a $f0
```

Se carga en los registros $t0 y $f4 el número 4 que es entre el que se dividirá la suma de los floats. En el registro $t1 se guarda la dirección de dichos valores y en el $f0 el valor 0.

```
bucle:          lwc1 $f6, 0($t1)        # Lee valor[i]
                add.s $f0, $f0, $f6     # Suma del valor
                addiu $t1, $t1, 4       # Dirección de valor[i+1]
                addiu $t0, $t0, -1      # Decrementa contador
                bgtz $t0, bucle

                cvt.s.w $f4, $f4        # Convierte dimensión a real
                div.s $f0, $f0, $f4     # Calcula media aritmética
                cvt.d.s $f2, $f0        # Convierte a doble precisión
                la $t0, media_s         # Dirección del resultado media_s
                swc1 $f0, 0($t0)        # Escribe resultado simple precisión
                la $t0, media_d         # Dirección del resultado media_d
                swc1 $f2, 0($t0)        # Escribe parte baja doble precisión
                swc1 $f3, 4($t0)        # Escribe parte alta doble precisión
                .end
```

Guarda la suma de todos los float en $f6

Como $t0 contiene una dirección parece que estamos escribiendo sobre un registro pero estamos escribiendo sobre memoria principal

$f0 = media en simple precisión

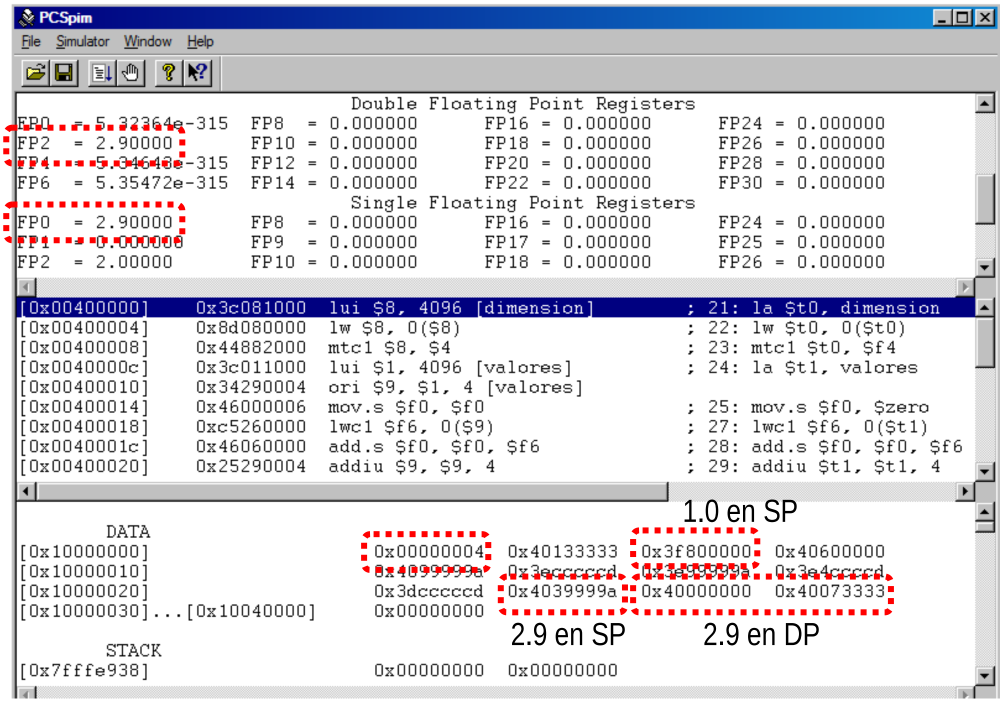
$f2 = media en doble precisión

(en $f2 se guardará la parte baja, y en $f3 la alta automáticamente)

► El programa anterior se halla en el fichero `promedio.s`. Cárguelo y ejecútelo en el simulador. La siguiente figura muestra el estado del simulador después de ejecutar el programa. Compruebe que los resultados que ha obtenido coinciden con los mostrados en la figura.

Resultado SP en `$f0`

Resultado DP en `$f3|$f2`



En la figura anterior se ha destacado la ubicación del resultado en el banco de registros de la unidad de coma flotante, así como su localización en el segmento de datos. Vuelva a examinar el código y analice su funcionamiento, en especial la ubicación de los datos de partida en el segmento de datos, su codificación, así como la localización de los resultados.

► Explique el cometido de la instrucción `cvt.s.w` que aparece en el código.

Convertir la variable 'dimension' de entero a real para poder usarla en una división con valores float

► Indique en la siguiente tabla la codificación IEEE 754 para el promedio calculado tanto en simple como en doble precisión. Tenga cuidado con la variable de doble precisión: la parte baja se almacena en la dirección de memoria más baja, y la parte alta en la dirección más alta.

Variable	Valor decimal	Codificación IEEE 754
media_s	2.9	0x4039999A
media_d	2.9	0x4007333340000000

► Indique cuántas operaciones aritméticas de coma flotante y de qué clase (suma, resta, conversión de tipo, etc.) se ejecutan en el programa.

4 sumas de simple precisión en el bucle (add.s)
1 división en simple precisión (div.s)
1 de conversión de word a simple precisión (cvt.s.w)
1 de conversión de simple a doble precisión (cvt.d.s)

7 operaciones

► Si el programa se ejecuta en un procesador real en 0.5 microsegundos, calcule el número de operaciones en coma flotante por segundo conseguidos por el procesador (FLOPS, *floating point operations per second*). Indique el resultado en millones de operaciones por segundo (MFLOPS).

7 operaciones —— 0.5 μs
x operaciones —— 1 s

$$x = \frac{7 \cdot 1s}{0.5 \cdot 10^{-6} s} = 14 \cdot 10^{6} \text{ operaciones} = 14 \text{ MFLOPS}$$

M

Cálculo del número π

El número π (pi) es la relación entre la longitud de una circunferencia y su diámetro. Es un número irracional y una de las constantes matemáticas más importantes. Se emplea frecuentemente en matemáticas, física e ingeniería. El valor numérico de π, truncado a sus primeras cifras, es el siguiente:

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

El valor de π se ha obtenido con diversas aproximaciones a lo largo de la historia, siendo una de las constantes matemáticas que más aparece en las ecuaciones de la física, junto con el número *e*.

El matemático alemán Gottfried Leibniz ideó en 1682 un método para el cálculo del número π. Dicho método realiza una aproximación a π/4 a través de la serie infinita siguiente:

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

A continuación se presenta un programa que calcula el valor de π mediante la serie anterior. El código que presentamos desarrolla la serie hasta el valor especificado por el usuario para la variable del programa `n`. El programa principal usa la subrutina `leibniz` para dicho cálculo. La entrada y salida de datos se lleva a cabo mediante llamadas al sistema.

```
                ############################################################
                # Segmento de datos
                ############################################################

                .data 0x10000000
cad_entrada:    .asciiz "\nDime el número de iteraciones: "
cad_salida:     .asciiz "El valor calculado de pi es: "

                ############################################################
```

```
                # Segmento de código
                ##########################################################

                .globl __start
                .text 0x00400000
__start:
                ########################################################
                # Lectura del número de iteraciones
                ########################################################

                la $a0, cad_entrada     # Cadena a imprimir
                li $v0, 4               # Función print_string
                syscall

                li $v0, 5               # Función read_int
                syscall
                move $a0, $v0           # Parámetro de la subrutina
                jal leibniz             # Salto a la subrutina

                ########################################################
                # Impresión del resultado
                ########################################################

                la $a0, cad_salida      # Cadena a imprimir
                li $v0, 4               # Función print_string
                syscall

                li $v0, 2               # Función print_float
                mfc1 $t0, $f0           # Valor a imprimir
                mtc1 $t0, $f12
                syscall

                ########################################################
                # Finalización del programa
                # Llamada al sistema denominada "exit"
                ########################################################=

                li $v0, 10
                syscall

                ########################################################
                # Cálculo de pi con el método de Leibniz
                # $a0 = Número de iteraciones de la serie
                ########################################################

leibniz:        li.s $f0, 0.0           # Constante 0.0
                li.s $f4, 1.0           # Constante 1.0
                li.s $f6, 2.0           # Constante 2.0
                move $t0, $a0           # Contador número de iteraciones
bucle:          mtc1 $t0, $f8           # Lleva n a la FPU
                cvt.s.w $f8, $f8        # Convierte n en número real
```

Carga en el registro $f8 el valor 4, que será el número de iteraciones del bucle.

```
                mul.s $f8, $f8, $f6     # Calcula 2.0*n
                add.s $f8, $f8, $f4     # Calcula 2.0*n + 1.0
                div.s $f8, $f4, $f8     # Calcula 1.0/(2.0*n + 1.0)
                andi $t1, $t0, 0x0001   # Extrae bit LSB de n
                bne $t1, $zero, resta   # Salta si es impar (LSB==1)
                add.s $f0, $f0, $f8     # El término se suma
                j continua
resta:          sub.s $f0, $f0, $f8     # El término se resta
continua:       addi $t0, $t0, -1       # Decrementa número de iteraciones
                bgez $t0, bucle         # Vuelve si quedan iteraciones
```

Aplica fórmula para calcular pi/4

```
                li.s $f4, 4.0           # Constante 4.0
                mul.s $f0, $f0, $f4     # Devuelve en $f0 el cálculo de pi
                jr $ra
                .end
```

Multiplica pi/4 * 4 para obtener pi

Analice con detenimiento el código anterior. Podrá comprobar que todo el cálculo de la serie se lleva a cabo dentro de la subrutina `leibniz`.

► Indique cómo hace el programa para calcular si el término de la serie se suma (`n` par) o se resta (`n` impar).

Extrae el bit LSB de la cifra resultante del cálculo 1/(2*n + 1) y lo compara con 0. Si LSB == 0, el número es par. Si no, es impar.

► Exprese el número de operaciones de coma flotante que se llevan a cabo en el programa anterior en función del número *n* de iteraciones.

n * (una multiplicación, dos sumas y una división) + (n/2)*(una resta) + 4 li.s + 1 mult.s = 23 operaciones para n = 4

► Cargue en el simulador el programa anterior (fichero `pi-leibniz.s`) y ejecútelo para los diferentes desarrollos de la serie que se especifican más abajo. Complete la siguiente tabla indicando los diez primeros números decimales calculados del número π. Redondee el valor del décimo dígito.

Iteraciones (**n**)	Valor calculado de **π**
10^3	3.1425914764
10^4	
10^5	
10^6	

La arquitectura del MIPS R2000 nos ofrece instrucciones de movimiento de datos entre los bancos de registros enteros y de coma flotante (`mtc1`, `mfc1`). También existen instrucciones específicas de movimiento entre registros de coma flotante: `mov.s` y `mov.d`. Por ejemplo, `mov.s $f4,$f2` copia el contenido del registro `$f2` en `$f4`.

► Imagine por un momento que la instrucción `mov.s` no estuviese disponible en la arquitectura del procesador. ¿Qué instrucciones alternativas se podrían utilizar para mover el contenido del registro `$f2` a `$f4`?

Las de tipo lwc1 (?)

► Para mover el contenido de un registro entero a otro se puede utilizar la pseudoinstrucción `move`. Por ejemplo, `move $t0,$t1` lleva el contenido de `$t1` a `$t0`. ¿Por qué cree que `move` no se ha incluido en el procesador como una instrucción máquina?

Pues no lo se pero move $t0, $t1 hace lo mismo que add $t0, $zero, $t1

► Adapte el programa a números reales codificados en el estándar IEEE 754 de doble precisión (variables reales de tipo *double*) y llame al fichero `pi-leibniz-d.s`. Tenga cuidado con el traslado del resultado final ubicado en la pareja de registros `$f1|$f0` para su impresión en la consola. Entre otras cosas tendrá que modificar la llamada al sistema que imprime este tipo de variables (el índice de `print_double` es 3). Indique brevemente los cambios realizados respecto de la versión original en simple precisión.

► Ejecute el programa y complete la siguiente tabla:

Iteraciones (**n**)	Valor calculado de π
10^3	3.1425916543
10^4	
10^5	
10^6	