

Universidad de Buenos Aires - FIUBA
66.20 Organización de Computadoras
Trabajo Práctico 1: Assembly Mips

Joaquin Segui, *Padrón Nro. 91.451*
`segui.joaquin@gmail.com`

Pernin Alejandro, *Padrón Nro. 92.216*
`ale.pernin@gmail.com`

Menniti Sebastián Ezequiel, *Padrón Nro. 93.445*
`mennitise@gmail.com`

1. Introducción

Se implementó un programa, en lenguaje C, que se encarga de multiplicar matrices de números reales, representados en punto flotante de doble precisión. A diferencia del trabajo anterior, la función encargada de la multiplicación de las matrices se encuentra escrita en el lenguaje Assembly Mips.

2. Diseño e Implementación

Dada las limitaciones del assembly Mips, fue necesario realizar algunos cambios respecto de la entrega anterior

- Representar las matrices como un unico arreglo de doubles.
- Diseñar el manejo de los argumentos mediante el stack.
- Modificar el algoritmo de multiplicación para contemplar errores de MIPS¹.

Para manejar diversas variables dentro del programa, se utilizo un Stack Frame de 32 bytes.

52	columnas 2
48	columnas 1
44	a3 (filas 1)
40	a2 (double* src1)
36	a1 (double* src2)
32	a0 (double* dest)
28	ra
24	fp
20	gp
16	i
12	j
8	k
4	(not used)
0	accum

Cuadro 1: Stack Frame

Alli se alojan los valores empleados para los controles de los ciclos y el acumulador auxiliar para la multiplicación. Si bien la posición 4 no se utiliza, el frame debe ser multiplo de 8 bytes y por ello se mantienen los 32 bytes.

3. Comandos para compilar el programa

3.1. Copiar a VM

Dado que el programa se corre dentro de la maquina virtual gxemul, se facilita un script en bash que copia el contenido de la carpeta en el guest (considerando la configuración default vista en clase). Para ello invocar desde el host

```
copiar_tp.sh
```

el mismo se copiará en `/root/tp1`.

3.2. Compilacion y Ejecucion

Para facilitar la compilacion se utiliza un *Makefile*, para invocar la compilación del programa ejecutar desde una consola, dentro del mismo directorio que el código fuente:

```
make
```

¹Ver sección 4.3

Asimismo se proporciona un script que realiza pruebas con un set de datos preexistente, para invocarlo:

```
bash pruebas_mips.sh
```

4. Pruebas

En esta sección se detallarán las pruebas realizadas. Los archivos utilizados se encuentran en el directorio *test_files*.

4.1. Casos Exitosos

Los casos exitosos están comprendidos por los set de datos *test1* y *textittest2*. En el caso del primero:

```
1x2 1 2
2x3 1 0 4 5 1 3
```

Representa la operación

$$\begin{pmatrix} 1 & 2 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 4 \\ 5 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 11 & 2 & 10 \end{pmatrix}$$

cuya salida por consola mediante el script de pruebas es

```
1x3 11 2 10
```

como es esperable.

El segundo set de datos es:

```
3x1 1.000 2.00 3.00
1x3 0.0 3.000 1.000
```

```
1x2 1 3
2x3 1 0 4 5 1 0
```

representando las operaciones

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} * \begin{pmatrix} 0 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 3 & 1 \\ 0 & 6 & 2 \\ 0 & 9 & 3 \end{pmatrix}$$
$$\begin{pmatrix} 1 & 3 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 4 \\ 5 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 16 & 3 & 4 \end{pmatrix}$$

cuya salida se obtuvo correctamente

```
3x3 2.22507e-308 3 1 3.33761e-308 6 2 4.45015e-308 9 3
1x3 16 3 4
```

4.2. Casos de error

Como casos de error del programa probamos matrices incompatibles para su multiplicación y matrices mal definidas.

Uno de estos casos es el de tener dos matrices cuyas dimensiones hacen incompatibles la multiplicación entre sí. Este es el caso del set *test3*.

```
1x2 1 2
1x3 1 0 4
```

$$\begin{pmatrix} 1 & 2 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 4 \end{pmatrix}$$

Al ejecutar dicha prueba, el programa termina con el siguiente mensaje:

```
Dimensiones no compatibles para multiplicar
```

Otra prueba es tener una cantidad impar de matrices, por lo cuál una no podrá ser multiplicada. Por ejemplo *test4*.

```
3x1 1.000 2.00 3.00
1x3 0.0 3.000 1.000
```

```
1x2 1 3
```

Como resultado arroja

```
3x3 2.22507e-308 3 1 3.33761e-308 6 2 4.45015e-308 9 3
Fallo al leer dimensiones
```

Otros casos de prueba, consisten en definir dimensiones de matrices inconsistentes con la cantidad de elementos leídos. Al ver *test5*

```
3x1 1.000 2.00
1x3 0.0 3.000 1.000
```

```
1x2 1 3
2x3 1 0 4 5 1 0
```

$$\begin{pmatrix} 1 \\ 2 \\ X \end{pmatrix} * \begin{pmatrix} 0 & 3 & 1 \end{pmatrix}$$

si bien las dimensiones declaradas son compatibles para su multiplicación, los elementos provistos son inconsistentes. Dicha prueba arroja:

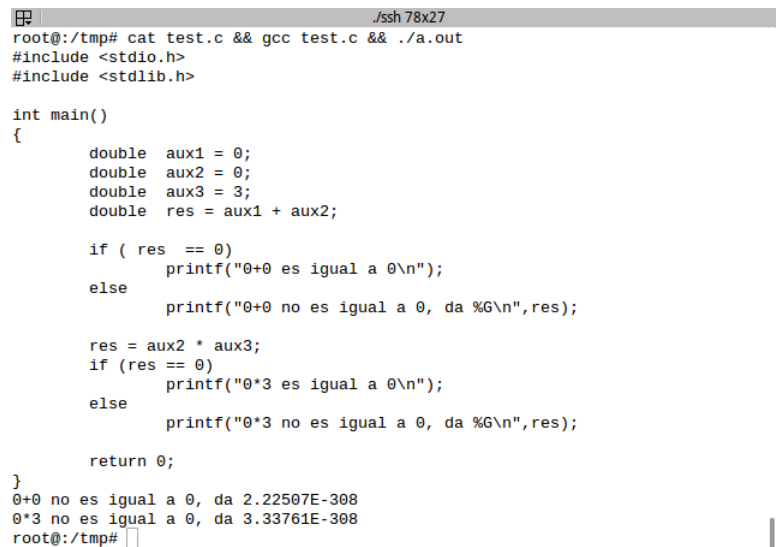
Cantidad elementos distinta a dimensiones de matriz

4.3. Comportamiento en MIPS

En la anterior implementación de este algoritmo en MIPS, observamos que los resultados que debieran dar 0, dan números extremadamente chicos expresados en notación científica. Observando este comportamiento sólo en el *guest* y no el *host* lo que nos llevó a efectuar un posterior análisis. Por ejemplo el caso de prueba exitoso 2 (*test2.txt*) arrojaba como primer resultado:

```
3x3 2.22507e-308 3 1 3.33761e-308 6 2 4.45015e-308 9 3
```

Para rastrear el origen de dicho problema se hicieron diversos prints y luego se empleó una prueba unitaria para ejemplificar dicho inconveniente (*comportamiento_mips.c*).



```

./ssh 78x27
root@:/tmp# cat test.c && gcc test.c && ./a.out
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double aux1 = 0;
    double aux2 = 0;
    double aux3 = 3;
    double res = aux1 + aux2;

    if ( res == 0)
        printf("0+0 es igual a 0\n");
    else
        printf("0+0 no es igual a 0, da %G\n",res);

    res = aux2 * aux3;
    if (res == 0)
        printf("0*3 es igual a 0\n");
    else
        printf("0*3 no es igual a 0, da %G\n",res);

    return 0;
}
0+0 no es igual a 0, da 2.22507E-308
0*3 no es igual a 0, da 3.33761E-308
root@:/tmp#

```

Figura 1: Prueba Comportamiento Mips

Como es observable, el problema radica en las operaciones de suma y multiplicación cuando uno de sus operandos es el cero.

Para evitar este comportamiento, el algoritmo compara si alguno de los elementos de las matrices a ser multiplicado es nulo. De serlo, el acumulador permanece igual y se continua en el siguiente ciclo. De esta manera el acumulador se mantiene inalterado por dicho comportamiento.

5. Conclusiones

Mediante el estudio e implementación de algoritmos en assembly MIPS, pudimos observar cómo funciones sencillas en un lenguaje de alto nivel (desde este punto de vista) se traducen en múltiples instrucciones de bajo nivel en diversas ocasiones siendo éste mucho más complejo.

Asimismo entendimos la importancia de establecer convenciones a la hora de realizar llamadas entre funciones para que dichas no sólo puedan realizarse pasajes de argumentos y resultados, sino para que durante el funcionamiento de una de ellas no afecte de manera indebida la otra.

A lo largo del desarrollo de este TP, como guía observamos y estudiamos el código assembly obtenido mediante el compilador de diversos programas sencillos, resultando de gran utilidad.

6. Código fuente del programa

6.1. En lenguaje C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <getopt.h>
5 #include <string.h>
6
7 extern void myMultiplicar(double* matriz1, double* matriz2, double* matrizRes,int fila1 , int columnal,int
8
9 //Funcion que imprime el manual del TP0
10 void printManual(){
11     printf("Usage:\n\ntp1_-h\n");
12     printf("\ntp1_-V\n");
13     printf("\ntp1_<_in_file_>_out_file\n");
14     printf("Options:\n");
15     printf("\n-V,--version_ _Print_version_and_quit.\n");
16     printf("\n-h,--help_ _Print_this_information_and_quit.\n");
17     printf("Examples:\n");
18     printf("\ntp0_<_in.txt_>_out.txt\n");
19     printf("\ncat_in.txt_|_tp0_>_out.txt\n");
20 }
21
22 void parsearOpciones(int argc, char* argv[]) {
23     int next_option;
24     const char* const short_options = "hV";
25     const struct option long_options[] = {
26         { "help",      0, NULL, 'h' },
27         { "version",   0, NULL, 'V' },
28         { NULL,        0, NULL, 0 } //Necesario al final del array
29     };
30     //Procesamiento de los parametros de entrada.
31     do {
32         next_option = getopt_long(argc, argv, short_options, long_options, NULL);
33         switch (next_option){
34             case 'h': // -h, --help
35                 printManual();
36                 exit(EXIT_SUCCESS);
37                 break;
38             case 'V': // -V, --version
39                 printf("\nVersion_2.0_del_TP0\n");
40                 exit(EXIT_SUCCESS);
41                 break;
42             case -1: // Se terminaron las opciones
43                 break;
44             default: // Opcion incorrecta
45                 fprintf(stderr, "Error,_el_programa_se_cerrara.\n");
46                 printManual();
47                 exit(EXIT_FAILURE);
48         }
49     } while (next_option != -1);
50 }
51
52 double* alocarMatriz( int filas , int columnas) {
53     double* matriz;
54     matriz = (double*)malloc(filas*columnas*sizeof(double));
55     if (!matriz) {
56         return NULL;
57     }
58     return matriz;
59 }
60
61 size_t strLength(char* s){
62     size_t i;
63     for(i = 0; s[i] != 0; i++);
64     return i;
65 }
66 /*
67 fila: cantidad de filas
68 columna: cantidad de columnas
69 matriz: el puntero de la matriz
```

```

70 */
71 int llenarMatriz(double* matriz, int fila, int columna) {
72     int i;
73     int j;
74     char c;
75     int cantidadElementos = 0;
76     i = 0;
77     j = 0;
78     bool exito = true;
79     double valor=0;
80     int flag,pos;
81     while (exito && i<fila) {
82         flag = scanf("%d%f%c",&valor,&c);
83         if (flag != EOF && flag == 2) {
84             //printf("Leo: %d y %c\n",valor,c);
85             pos = (i*columna)+j;
86             matriz[pos] = valor;
87             cantidadElementos++;
88             if (j==columna-1) {
89                 j=0;
90                 i++;
91             } else {
92                 j++;
93             }
94         } else {
95             exito = false;
96         }
97     }
98     if (cantidadElementos != ((fila)*(columna)) || (c!='\n')) {
99         return EXIT_FAILURE;
100     }
101     return EXIT_SUCCESS;
102 }
103
104 void mostrarMatriz(double* matriz, int fila, int col)
105 {
106     printf("%dx%d_",fila,col);
107     int i;
108     for(i=0; i<fila*col; i++)
109     {
110         printf("%g_",matriz[i]);
111     }
112     printf("\n");
113 }
114
115 void liberarMatriz(double* matriz, int fila) {
116
117     free(matriz);
118 }
119
120 void multiplicar(double* matriz1, double* matriz2, double* matrizRes,int fila1, int columna1,int columna2)
121 {
122     int i;
123     int j;
124     int k;
125     double accum;
126     printf("%x%g_",fila1,columna2);
127     for(i=0;i<fila1;i++) {
128         for(j=0;j<columna2;j++) {
129             accum = 0;
130             for(k=0;k<columna1;k++) {
131                 int pos1 = (i*columna1)+k;
132                 int pos2 = (k*columna2)+j;
133                 accum = accum + (matriz1[pos1] * matriz2[pos2]);
134             }
135             int pos3 = (i*columna2)+j;
136             matrizRes[pos3] = accum;
137             //printf("%g (en %d)",accum,pos3);
138         }
139     }
140     //printf("\n");
141 }
142
143 int main(int argc, char *argv[]) {
144     parsearOpciones(argc,argv);

```

```

144 //Construyo la primera matriz
145 double* matriz1;
146 int fila1;
147 int columna1;
148 int cant;
149 cant = scanf(" %d %c %d ",&fila1,&columna1);
150 do {
151     if (cant != 2) {
152         fprintf(stderr, "Fallo al leer dimensiones\n");
153         return EXIT_FAILURE;
154     }
155     matriz1 = alocarMatriz(fila1 ,columna1);
156     if (!matriz1) {
157         fprintf(stderr, "Fallo en malloc\n");
158         return EXIT_FAILURE;
159     }
160     int llenar;
161     llenar = llenarMatriz(matriz1 ,fila1 ,columna1);
162     if (llenar) {
163         liberarMatriz(matriz1 ,fila1 );
164         fprintf(stderr, "Cantidad de elementos distinta a dimensiones de matriz\n");
165         return EXIT_FAILURE;
166     }
167 //Repito para segunda matriz
168 double* matriz2;
169 int fila2;
170 int columna2;
171 cant = scanf(" %d %c %d ",&fila2,&columna2);
172 if (cant != 2) {
173     liberarMatriz(matriz1 ,fila1 );
174     fprintf(stderr, "Fallo al leer dimensiones\n");
175     return EXIT_FAILURE;
176 }
177 matriz2 = alocarMatriz(fila2 ,columna2);
178 if (!matriz2) {
179     liberarMatriz(matriz1 ,fila1 );
180     fprintf(stderr, "Fallo en malloc\n");
181     return EXIT_FAILURE;
182 }
183 llenar = llenarMatriz(matriz2 ,fila2 ,columna2);
184 if (llenar) {
185     liberarMatriz(matriz1 ,fila1 );
186     liberarMatriz(matriz2 ,fila2 );
187     fprintf(stderr, "Cantidad de elementos distinta a dimensiones de matriz\n");
188     return EXIT_FAILURE;
189 }
190 if(columna1 == fila2) {
191     double* matrizRes = alocarMatriz(fila1 ,columna2);
192     if (!matrizRes)
193     {
194         liberarMatriz(matriz1 ,fila1 );
195         liberarMatriz(matriz2 ,fila2 );
196         //liberarMatriz(matrizRes ,fila1 );
197         fprintf(stderr, "Fallo en malloc\n");
198         return EXIT_FAILURE;
199     }
200 //Multiplicar
201 myMultiplicar(matriz1 , matriz2 , matrizRes , fila1 , columna1 , columna2);
202 mostrarMatriz(matrizRes ,fila1 ,columna2);
203
204     liberarMatriz(matriz1 ,fila1 );
205     liberarMatriz(matriz2 ,fila2 );
206     liberarMatriz(matrizRes ,fila1 );
207 } else {
208     liberarMatriz(matriz1 ,fila1 );
209     liberarMatriz(matriz2 ,fila2 );
210     fprintf(stderr, "Dimensiones no compatibles para multiplicar\n");
211     return EXIT_FAILURE;
212 }
213 cant = scanf(" %d %c %d ",&fila1,&columna1);
214 } while(cant != EOF);
215 //Repetir
216 return EXIT_SUCCESS;
217 }

```


6.2. Funcion en MIPS32

```
1 #include <mips/regdef.h>
2 .text
3 .globl myMultiplicar
4 .ent myMultiplicar
5 #myMultiplicar(double* src1, double* src2, double* dest,int f1, int c1,int c2);
6 # a0 = src1
7 # a1 = src2
8 # a2 = src3
9 # a3 = f1
10 # 16(sp) = c1
11 # 20(sp) = c2
12
13 myMultiplicar:
14     .frame $fp,32,ra
15     subu sp, sp, 32      #Creo el Stack Frame
16     sw ra,28(sp)        #Guardo el Return Address
17     sw $fp,24(sp)
18     sw gp,20(sp)
19     move $fp, sp        #uso fp en vez de sp
20
21     #bloque para codigo PIC
22     .set noreorder
23     .cpload t9
24     .set reorder
25
26     #Cargo los args en el stack
27     sw a0,32($fp) #double* m1
28     sw a1,36($fp) #double* m2
29     sw a2,40($fp) #double* mres
30     sw a3,44($fp) #int filas 1
31     lw t0,48($fp) #int columnas 1
32     sw t0,48($fp) #Redundancia
33     lw t1,52($fp) #int columnas 2
34     sw t1,52($fp) #redundancia
35
36     move t0,zero #i = 0
37     sw t0,16($fp) #i -> 16($fp)
38
39
40 _loop_i:
41     lw t0,16($fp) #Redundante, t0=i
42
43     move t1, zero #j = 0
44     sw t1,12($fp) #j -> 12($fp)
45     b _loop_j
46
47 _end_loop_i:
48     lw t0,16($fp)
49     addu t0,t0,1
50     sw t0,16($fp) #i++
51
52     lw v0,44($fp) #v0 = filas 1
53     blt t0, v0, _loop_i #continuar si i < filas 1
54     j _exit #Salgo
55
56 _loop_j:
57     lw t1,12($fp) #Redundante, t1=j
58
59     move t2,zero #k = 0
60     sw t2,8($fp) #k -> 8($fp)
61
62     sub.d $f0,$f0,$f0 #acum = 0
63     s.d $f0,($fp) #PREGUNTAAR accum-> fp
64
65     b _loop_k
66
67 _end_loop_j:
68     lw t0,16($fp) #cargo i
69     lw t1,12($fp) #cargo j
70     lw v0,52($fp) #cargo columnas 2
71     mulou v1,v0,t0 # i * columnas 2
72     addu v1,v1,t1 # (i * columnas2)+j #offset
```

```

73  mulou v1,v1,8
74
75  l.d $f0,($fp) # accum <- fp
76  lw a2,40($fp)
77  la a2,0(a2)
78
79  addu a2,a2,v1 # mres[offset]
80  s.d $f0,(a2) #mres[offset]=acum
81
82  addu t1,t1,1
83  sw t1,12($fp) #j++
84
85  blt t1, v0, _loop_j #continuar si j < columnas 2
86  b _end_loop_i
87
88 _loop_k:
89  l.d $f0,($fp) #Cargo accum en $f0
90  lw t0,16($fp) #Cargo i
91  lw t1,12($fp) #Cargo j
92  lw t2,8($fp) #Cargo k
93
94  lw t3,44($fp) #Cargo Filas 1
95  lw t4,48($fp) #Cargo Columnas 1
96  lw t5,52($fp) #Cargo Columnas 2
97
98  mulou v0,t0,t4 # (i * columnas 1)
99  addu v0,v0,t2 #(i * columnas 1) + k = pos1
100 mulou v0,v0,8
101
102 mulou v1,t2,t5 #(k * columnas 2)
103 addu v1,v1,t1 #(k * columnas 2) + j = pos2
104 mulou v1,v1,8
105
106 lw a0,32($fp)
107 la a0,0(a0)
108 addu a0,a0,v0 #m1[pos1]
109 l.d $f2,(a0) # aux 1 = m1[pos1]
110
111 sub.d $f6,$f6 # $f6 = 0
112 c.eq.d $f2,$f6 # $f2 == 0?
113 bclt _end_loop_k #Si multiplico x 0, no sumo
114
115 lw a1,36($fp)
116 la a1,0(a1)
117 addu a1,a1,v1 #m2[pos2]
118 l.d $f4,(a1) #aux 2 = m2[pos2]
119
120 sub.d $f6,$f6 # $f6 = 0
121 c.eq.d $f4,$f6 # $f4 == 0?
122 bclt _end_loop_k #Si multiplico x 0, no sumo
123
124 #Si llego aca sumo algo
125 mul.d $f2,$f2,$f4 #m1[pos1] * m2[pos2]
126 add.d $f0,$f0,$f2 #acum = acum + (m1[pos1] * m2[pos2])
127
128 _end_loop_k:
129 s.d $f0,($fp) #Guardo el accum
130
131 addu t2,t2,1
132 sw t2,8($fp) #k++
133
134 blt t2, t4, _loop_k #Seguir si k < columnas 1
135 b _end_loop_j
136
137
138 _exit:
139 #Desarmo el Stack Frame
140
141 move sp,$fp
142 lw ra,28(sp)
143 lw $fp,24(sp)
144 lw gp,20(sp)
145 lw s0,(sp)
146 addu sp,sp,32

```

```
147      j ra
148
149 .end myMultiplicar
```

7. Enunciado

Universidad de Buenos Aires - FIUBA
66.20 Organización de Computadoras
Trabajo práctico 1: assembly MIPS
2^{do} cuatrimestre de 2015

\$Date: 2015/09/29 14:34:09 \$

1. Objetivos

Familiarizarse con el conjunto de instrucciones MIPS y el concepto de ABI, extendiendo un programa que resuelva el problema descrito en la sección 4.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 6, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

En este trabajo práctico implementaremos el programa descrito en el TP anterior, utilizando el conjunto de instrucciones MIPS, y aplicando la convención de llamadas a funciones explicada en clase [1].

4.1. Implementación

En su versión mas simple, cuando es invocado sin argumentos, el programa debe tomar datos provenientes de `stdin` e imprimir el resultado en `stdout`, implementando la funcionalidad descrita en el trabajo práctico inicial.

Componentes. La implementación deberá dividirse en tres secciones complementarias:

- **Arranque y configuración:** desde `main()` pasando por el procesamiento de las opciones y configuración del entorno de ejecución, incluyendo la apertura de los archivos de la línea de comando. Este componente deberá realizarse en lenguaje C.
- **Entrada/salida:** lectura y escritura de los *streams* de entrada (matrices), detección de errores asociados. Este componente también debe ser realizado en lenguaje C.
- **Procesamiento.** Recibe un entorno completamente configurado y las matrices a procesar (ya en memoria), calcula y retorna la matriz producto de acuerdo a la descripción del TP anterior. Esta parte del deberá escribirse íntegramente en assembly MIPS.

5. Pruebas

Es condición necesaria para la aprobación del trabajo práctico diseñar, implementar y documentar un conjunto completo de pruebas que permita validar el funcionamiento del programa. Asimismo deberán incluirse los casos de prueba correspondientes al TP anterior.

5.1. Interfaz

La interfaz de uso del programa coincide con la del primer TP.

6. Informe

El informe deberá incluir:

- Documentación relevante al diseño e implementación del programa.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C.
- Este enunciado.

7. Fechas

Fecha de vencimiento: martes 20/10/2015.

Referencias

- [1] MIPS ABI: Function Calling Convention, Organización de computadoras - 66.20 (archivo "func_call_conv.pdf". <http://groups.yahoo.com/groups/orga-comp/Material/>).