Departamento de Informática
Escuela Politécnica
Universidad de Extremadura

GUIÓN DE PRÁCTICAS DE ENSAMBLADOR DE MIPS

Asignatura ESTRUCTURA DE COMPUTADORES

2º Ingeniería Informática
2º Ingeniería Técnica en Informática de Sistemas

Curso 2000 / 2001

Juan A. Gómez Pulido
2000

Normativa de estas prácticas.

- Las prácticas de ensamblador de MIPS se desarrollan durante el primer cuatrimestre de la asignatura "Estructura de Computadores". Todas las prácticas están descritas en este guión. Se recomienda seguirlo progresivamente de la forma propuesta, dedicando todo el tiempo que sea necesario a cada práctica (cada una requiere un tiempo distinto a las otras).

- No hay un tiempo fijo para cada práctica (por ejemplo, una práctica cada semana). Depende de la capacidad del alumno el ir más o menos rápido, para lo cual, además de su propio esfuerzo, cuenta con la tutorización y ayuda del profesor en el laboratorio. Se recomienda no pasar a la siguiente práctica sin haber entendido y efectuado correctamente la actual, puesto que la evaluación de las prácticas tendrá en cuenta aspectos estrechamente relacionados con las mismas. No es necesario presentar una memoria escrita de cada práctica, si bien, hacia el final del cuatrimestre, se propondrá un trabajo práctico del que se entregará una memoria.

- A principio de curso se organizarán los distintos grupos de prácticas y horarios (2 horas semanales seguidas por grupo), que serán comunicados en el momento oportuno. La prácticas se desarrollarán en la Sala 1 (2ª planta del pabellón de Informática).

- En estas clases de laboratorio, y debido a la disponibilidad de puestos y horarios, se organizarán los alumnos en equipos de 2 por cada PC. Durante las clases de laboratorio se contará con el asesoramiento y tutorización del profesor para cualquier cuestión relacionada con el desarrollo de las prácticas. Cualquier duda, pregunta o cuestión relacionada con estas prácticas se realizará en los horarios de laboratorio, en las horas de tutoría del profesor, o a través de correo electrónico en gacdl@unex.es

- El alumno respetará el material informático que le corresponda, no podrá cambiar de grupo durante el transcurso del cuatrimestre, y usará siempre el mismo puesto de trabajo.

- Para la realización de las prácticas, bastará tener el siguiente material:

Guión de las prácticas	Es este documento. Se puede encontrar en el Servicio de reprografía de la Escuela Politécnica, o en Internet, en la dirección http://atc.unex.es.
Tutorial del ensamblador MIPS	Documentación tutorizada del ensamblador MIPS. Se puede encontrar en el Servicio de reprografía de la Escuela Politécnica, o en Internet, en la dirección http://atc.unex.es.
Simulador SPIM	Software simulador del ensamblador MIPS, necesario para el desarrollo de las prácticas. Se puede solicitar al profesor de la asignatura durante las horas de laboratorio, o encontrar en Internet, en la dirección http://atc.unex.es.

- La superación de las prácticas es condición indispensable para aprobar la asignatura, y supone un porcentaje determinado en la calificación de ésta:

Nota final = 70% nota teoría + 30% nota prácticas	(A)

Asímismo, la nota de las prácticas de la asignatura será la media de ambos cuatrimestres:

Nota prácticas = (nota prácticas cuatrim. 1 + nota prácticas cuatrim. 2) / 2	(B)

Para poder aplicar la fórmula (A), la nota práctica ha de ser igual o superor a 5 (teoría y prácticas no compensables). Para poder aplicar la fórmula (B), las notas de prácticas de ambos cuatrimestres han de superar el 4,5 (notas compensables).

- La evaluación de estas prácticas consistirá en una prueba escrita de 1 hora de duración. En este examen se formularán cuestiones directamente relacionadas con las prácticas desarrolladas en este guión y con el tutorial del ensamblador MIPS, y no se permitirán apuntes ni calculadoras. El profesor facilitará en el examen los datos necesarios para que el alumno pueda realizar el examen sin necesidad de aprender de memoria información compleja, tal como números de códigos de llamadas al sistema, etc.

Normas de estilo para el lenguaje ensamblador de MIPS:

Código:

- Usar nombres nemónicos para etiquetas y variables:
  voltaje1 en lugar de v1
- Usar nemónicos para registros:
  $a0 en lugar de $4
- Usar indentación y espaciado apropiados para facilitar la lectura del programa.
- Usar convenios de llamadas a procedimientos (ver transparencias 20 y 21 del tutorial). Una violación razonable de estos convenios deberá documentarse claramente en los comentarios. Por ejemplo, todos los registros guardados por el invocador (caller-save) deberían ser guardados en la pila por el invocador (caller). Pero si alguno de esos registros no se usan en el invocado (callee) no hay porque guardarlo. SIN EMBARGO, hay que incluir un comentario diciendo que esos registros normalmente se guardan.

Comentarios:

- Usar comentarios significativos:

```
add  $t0,$t2,$t4   # temp ← voltaje1 + voltaje2
```

en lugar de:

```
add  $t0,$t2,$t4   # sumar los registros $t2 y $t4 y poner resultado en $t0
```

Uso del laboratorio.

Cada equipo (2 personas) utilizará **siempre el mismo puesto**, responsabilizándose de:

- No grabar en el disco duro **ningún fichero** (utilizar diskettes).
- Dejar el puesto tal como se encontró, con monitor y CPU **apagadas**.
- No modificar, bajo ningún concepto, la apariencia o estado del escritorio de Windows, y limitarse a la ejecución del simulador SPIM.

Los códigos fuente han de almacenarse en un **disquette** o, temporalmente, en el subdirectorio **c:\temp** del disco duro.

Práctica 1. Introducción al simulador SPIM.

Objetivos:

- Instalación y manejo del entorno de SPIM.
- Operaciones básicas con la memoria, para comprender la forma de localización de datos y direcciones.

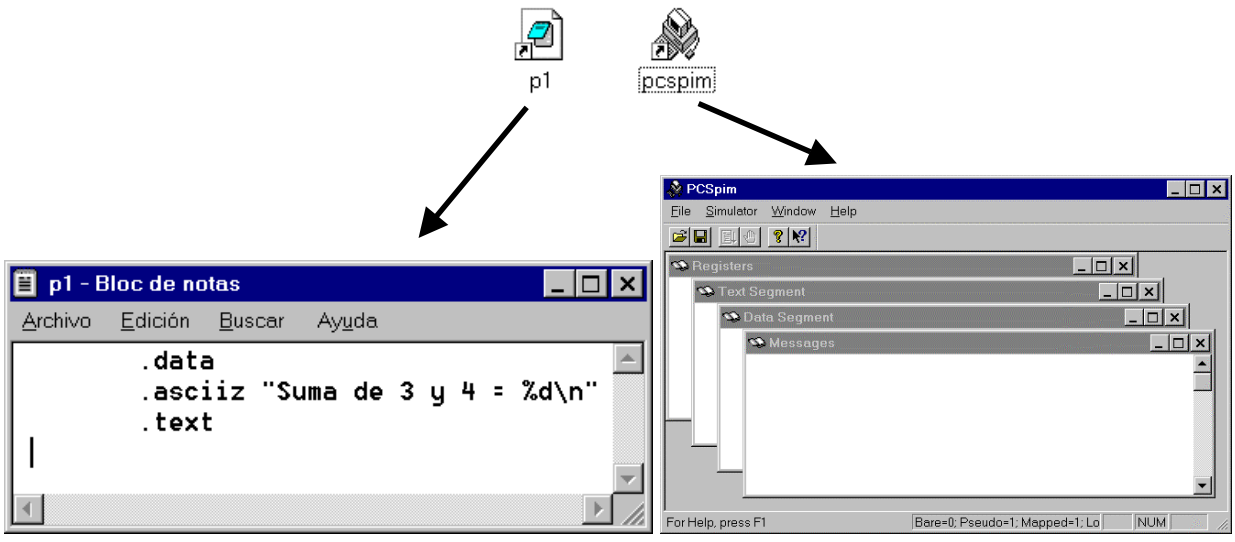
Fundamentos teóricos.

- **Leer las transparencias 1, 2, 4 y 5 del tutorial.**
- El contenido de las transparencias 3, 6 y 7 resume algunos aspectos del formato del ensamblador MIPS, aspectos que también se ven en la parte teórica de la asignatura.
- **Conviene leer y tener presentes las transparencias 22 a la 28**, en las que se muestran distintas características del simulador SPIM, y que iremos viendo a lo largo de este guión.

Desarrollo.

- Instalación de SPIM. Si vamos a utilizar un PC que no tiene SPIM instalado, ejecutamos el fichero de instalación del programa. Una vez instalado, ejecutamos el programa. Lo primero que observamos es un entorno de ventanas con menús y sub-menús. Es conveniente tener todas las ventanas abiertas (*messages, registers, text segment* y *data segment*). Observar que las ventanas no contienen nada aún.

- Conviene tener abierto un editor de texto ASCII en Windows. Se recomienda el "block de notas", o "notepad". En este editor se escribirá el código ensamblador MIPS, y lo almacenaremos con un fichero con extensión **.s**, por ejemplo **p1.s**, en una camino de datos temporal (como c:\temp ó a:\). Se recomienda encarecidamente el uso de diskettes. Escribimos el código ensamblador indicado en la figura.

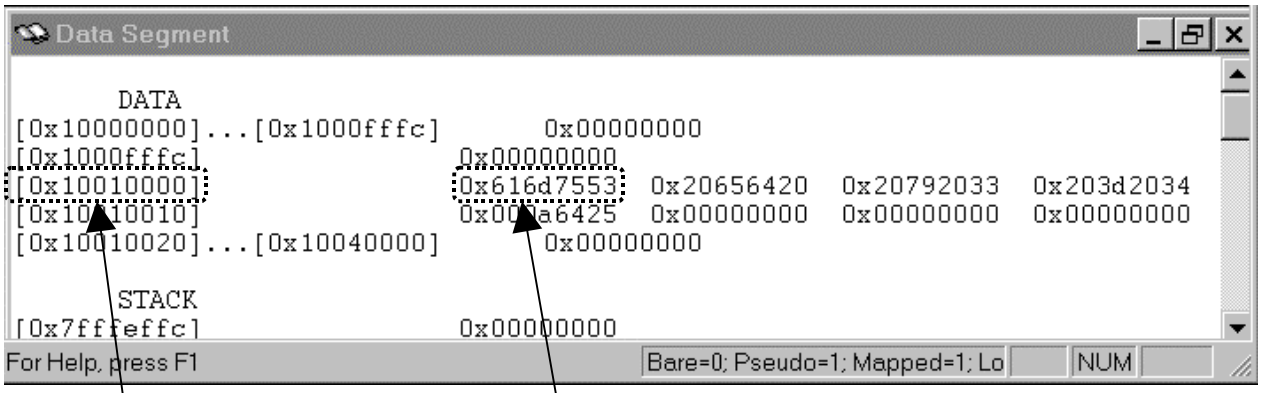


- Algunas versiones de SPIM tienen un "bug", que se evita haciendo que los códigos ensamblador que almacenamos en los ficheros **.s** terminen siempre con una línea en blanco (vacía).

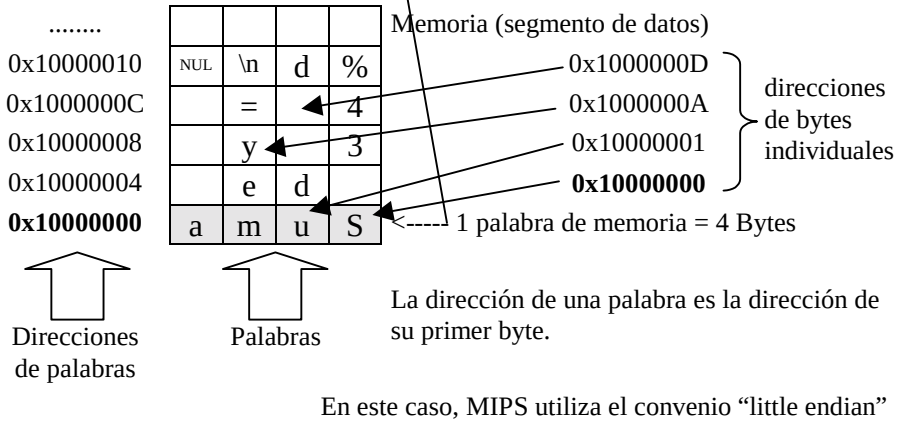
- El código **p0.s** de esta práctica no contiene instrucciones, luego no es necesario ejecutarlo. Consta de la declaración de un segmento de datos estático, es decir, mediante la directiva **.data** indicamos al ensamblador que vamos a utilizar una porción de memoria en la que vamos a declarar unos contenidos para unas direcciones dadas (ver transparencia 16). También, es necesario declarar el segmento de texto (mediante la directiva **.text**), que en este caso estará vacío, es decir, que no tendrá insturcciones.

- Para cargar el programa, vamos al menú FILE -> OPEN, y seleccionamos p0.s en el directorio adecuado. Si no hay error en el código, en la ventana *messages* debe aparecer un texto que concluya con la frase *".....\p1.s has been successfully loaded*".

- Al cargarlo en SPIM, se cargarán en el segmento de datos los bytes que se han definido mediante la directiva **.asciiz**. Esto no es una ejecución del programa: la ejecución del programa sería la ejecución de las instrucciones presentes en el segmento de texto. Abrir la ventana del segmento de datos, e interpretar las direcciones de memoria y el contenido de las mismas. Las siguientes figuras ayudan a comprender cómo es la estructuración de la memoria.



Ascii	S	u	m	a		d	e		3		y		4		=		%	d	\n	NULL
Dec	83	117	109	97	32	100	101	32	51	32	121	32	52	32	61	32	37	100	10	0
Hex	53	75	6D	61	20	64	65	20	33	20	79	20	34	20	3D	20	25	64	0A	00

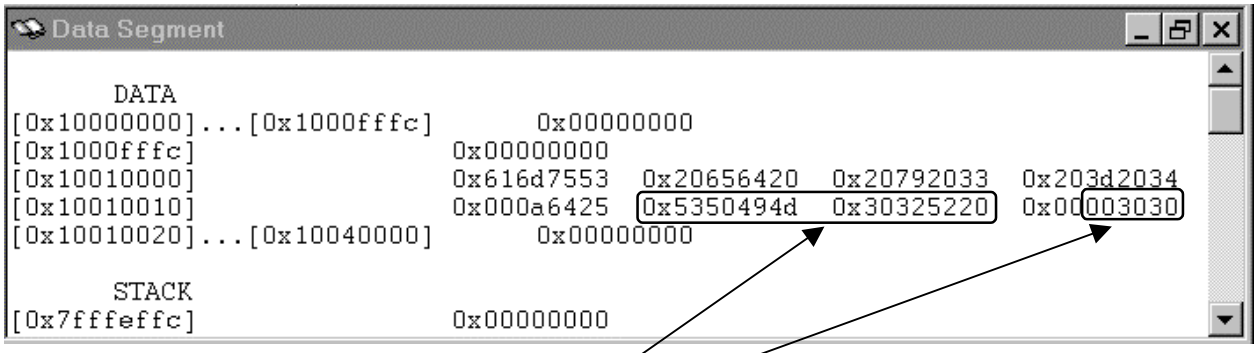


En este caso, MIPS utiliza el convenio "little endian"

- Vamos a colocar, a continuación de la directiva **.asciiz**, otra directiva, llamada **.byte** (ver transparencia nº 8) que carga datos en memoria en forma de bytes, de forma que éstos se colocarán inmediatamente después de los últimos cargados en memoria:

```
        .data
        .asciiz "Suma de 3 y 4 = %d\n"
        .byte 77,73,80,83,32,82,50,48,48,48,0   #Cadena “MIPS R2000”
        .text
```

- Grabar el fichero p1.s modificado, y en el menú Simulator, hacer sucesivamente *Clear Registers*, *Reinizialite* y *Reload*.... El contenido de la memoria ha de ser el siguiente:



Nuevos datos colocados en la memoria:

Práctica 2: Introducción al ensamblador MIPS.

Objetivos:

- Carga y ejecución de un código ensamblador MIPS en el simulador SPIM.
- Localización del segmento de datos y el segmento de código.
- Arrays y constantes en memoria.
- Operaciones de carga de memoria a registros.

Desarrollo:

- SPIM no lleva editor de texto asociado, por lo que el código ha de manipularse con algún procesador de textos (ASCII), como ya se indicó en la anterior práctica. Escribir el código de abajo en un fichero con extensión .S (no olvidar de incluir una línea en blanco al final).

```
# Práctica introducción al MIPS
# Juan A. Gómez Pulido. UEX. 1999

          .data                    # *** SEGMENTO DE DATOS ***
valor:    .word  8,9,10,11         # Defino array 4 palabras (decimal).
                                   # valores[2] es 10 = 0xa
          .byte  0x1a,0x0b,10      # Defino los 3 primeros bytes de la
                                   # siguiente palabra (hex. y dec.).
          .align 2                 # Para que el siguiente dato en mem.
                                   # esté alineado en palabra
          .ascii "Hola"            # Cadena de caracteres
          .asciiz",MIPS"           # Cadena de caracteres terminada
                                   # con el caracter nulo.
          .align 2                 # Las dos cadenas anteriores, junto
                                   # con el NULL final, ocupan 10 bytes,
                                   # por lo que alineo el siguiente dato
                                   # para que empieze en una palabra
id:       .word  1                 # Usado como índice del array "valor"
#-----------------------------------------------------------------------------
          .text                    # *** SEGMENTO DE TEXTO ***
          .globl main              # Etiqueta main visible a otros ficheros
main:                              # Rutina principal
          lw     $s0,valor($zero) # Carga en $s0 valor[0]
                                   # También vale: lw $s0,valor
          lw     $s4,id            # En $s4 ponemos el índice del array
          mul    $s5,$s4,4         # En $s5 ponemos id*4
          lw     $s1,valor($s5)    # Carga en $s1 valor[1]
          add    $s4,$s4,1         # Incrementamos el índice del array
          mul    $s5,$s4,4
          lw     $s2,valor($s5)    # Carga en $s2 valor[2]
          add    $s4,$s4,1         # Incrementamos el índice del array
          mul    $s5,$s4,4
          lw     $s3,valor($s5)    # Carga en $s3 valor[3]
```

- Ejecutar el simulador SPIM. Cargar el código ensamblador. Comprobar en la ventana de mensajes que la carga ha sido correcta.

⇒ Ver la ventana de **registros**. Observar los que hay, y que todos están inicializados a cero (excepto los que el S.O. ha dejado inicializados a ciertos valores). Podemos ver un listado de los registros y sus significados en la transparencia nº 9 del tutorial.

⇒ Ver la ventana del **segmento de datos**. Comprobar que todos los bytes que aparecen en las palabras de memoria corresponden con las definiciones de datos que aparecen en las directivas del código ensamblador.

⇒ Ver la ventana del **segmento de texto**. Observar que hay tres columnas. En la tercera aparece un código algo diferente al que hemos escrito. El código que hemos cargado en un fichero es un ensamblador con rótulos, escrito por el programador. El ensamblador lo convierte en un código ensamblador sin rótulos, apto para ser directamente convertido en código máquina (binario), que está descrito en la segunda columna. La primera columna informa de las direcciones de memoria donde se encuentra cada una de las instrucciones. En la versión SPIMWIN se puede observar cómo las primeras líneas de código no han sido escritas por el programador: son las instrucciones que el ensamblador introduce para una correcta carga y manipulación del programa en memoria. Las ignoraremos por ahora, ya que no son las que realizan las operaciones indicadas en nuestro código.

- Ejecutar el programa cargado en SPIM. La dirección que aparece en una ventana de confirmación corresponde a la dirección de inicio del programa que se va a ejecutar (inicialización del registro contador del programa, PC), y que corresponde con la promera de las instrucciones que el ensamblador introdujo para la adecuada ejecución de nuestro programa.

- Abrir la ventana de registros, y observar los valores que han tomado algunos de ellos. Para explicar esos valores, observar detenidamente las operaciones descritas por las instrucciones de nuestro código.

- Estudiar cómo se cargan en registros las distintas posiciones de un array de datos en memoria, y qué operaciones aritméticas es necesario realizar (y porqué).

- Estudiar las directivas e instrucciones que aparecen en el código (ver transparencia nº 8 del tutorial).

Práctica 3: Uso de la consola.

Objetivos:

- Manejo de las llamadas al sistema.
- Cómo imprimir y leer a / desde consola.
- Cómo terminar la ejecución de un programa.
- Cómo manipular elementos de un array de datos en memoria.
- Cómo cargar datos, no alineados en la memoria.

Desarrollo:

- Leer la transparencia nº 13 del tutorial. Introducir el siguiente código.

```
# PRAC_3A.S

          .data
string:   .asciiz  "Esto es una cadena\n"
item:     .word    99
array:    .word    11,22,33,44,55,66,77,88,99
          #-----------------------#
          .text
          .globl   main
main:
          #-----------------------#(1)
          la       $a0,string
          li       $v0,4
          syscall
          #-----------------------#(2)
          lw       $a0,item
          li       $v0,1
          syscall
          #-----------------------#(3)
          li       $v0,5
          syscall
          #-----------------------#(4)
          li       $v0,8
          la       $a0,string
          li       $a1,9
          syscall
          li       $v0,4
          syscall
          #-----------------------#(5)
          li       $t0,3
          li       $t1,4
          mul      $t2,$t1,$t0
          lw       $a0,array($t2)
          li       $v0,1
          syscall
          #-----------------------#(6)
          li       $v0,10
          syscall
```

- (1). Imprime una cadena en la consola. El método es cargar la dirección de la cadena en $a0, y entonces usar una llamada al sistema para imprimir la cadena. La instrucción “la” carga la

dirección base de la cadena en $a0. La instrucción “li” pone $v0 a 4, lo cual le indica a la llamada al sistema que va a imprimir la cadena de texto especificada por $a0. Finalmente, “syscall” hace la llamada al sistema, imprimiendo en la consola todos los caracteres que encuentra desde la dirección “string” hasta donde encuentre el byte “00” (caracter NULL, o fin de cadena).

- (2). Carga un entero desde el segmento de datos, y lo imprime en la consola.
- (3). Lee un entero desde la consola (espera a que ilo introduzcamos por teclado, imprimiéndose en la consola).
- (4). Lee una cadena de la consola. Mediante “li $v0,8” seleccionamos la función de syscall. Con “la $a0,string” seleccionamos la dirección base del buffer donde se va a escribir la cadena que introduciremos por consola (teclado). Por tanto, vamos a “machacar” lo que haya a partir de esa dirección (lo que había antes era “Esto es una cadena\n”). Con “li $a1,9” limitamos a 9 caracteres el tamaño del buffer (porción de memoria) donde irá la cadena. Después, con “syscall” se hace la llamada al sistema. Nótese que, al terminar de escribir la cadena, al accionar la tecla “Enter” estamos poniendo el caracter nulo, indicando el final de la cadena. . Comprobar qué pasa si nos pasamos de caracteres Para sacar en consola la cadena introducida, usamos “li $v0,4” y “syscall”.
- (5). Imprime un elemento desde un array de enteros que está en el segmento de datos.

El siguiente código es útil para ver cómo podemos acceder a datos que pertenecen a arrays de datos en memoria.

```
# PRAC_3B.S

     .data
X:   .word    5
Y:   .byte    3
Z:   .word    6
s:   .asciiz  "Hola\n"
a:   .word    10,3,5,2,6
     #-----------------------#
     .text
     .globl main
main:
     #-----------------------# Voy a leer en un registro un elemento
                             # del array "a", el elemento a[2]

     li   $s0,2              # i=2 : Índice del elemento del array "a"
     la   $s1,a              # p=&a[0]=a, dirección base del array "a"

     mul  $t0,$s0,4          # $t0 tiene el desplazamiento (en bytes)
                             # del elemento a[i] del array

     add  $t1,$s1,$t0        # $t1=p+4*i  $t1 tiene la dirección del
                             # elemento a[i] del array

     lw   $t2,0($t1)         # $t2 tiene el elemento a[i]

     lw   $s0,X              # Los datos etiquetados con X,Y,Z no están
     lb   $s1,Y              # alineados. Para poder cargarlos en los
     lw   $s2,Z              # registros, usamos la instrucción "lw"
                             # cuando es una palabra, y "lb" si es un byte

     #-----------------------# Fin del programa
     li   $v0,10
     syscall
```

Práctica 4: Bucles y condiciones.

Objetivos:

- Cómo implementar un bucle **for** usando la instrucción **bge**.
- Cómo implementar **switch** usando la instrucción **bne**.
- Llamada a una rutina no recursiva, pasando 3 argumentos y devolviendo un solo valor.
- Ejemplo de cómo pasar de un código C a un código ensamblador equivalente.

Desarrollo:

A. Sea el siguiente código C:

```
#include <iostream.h>

int ip(int *x,int *y, int n)
        {
        int i,sum;
        for(i=0,sum=0; i<n; i++,x++,y++)
                sum += (*x)*(*y);
        return sum;
        }

main()
        {
        int x[] = {2,4,3,5,1};
        int y[] = {3,3,2,5,4};
        int n = 5;

        cout << ip(x,y,n);
        }
```

Este programa hace una llamada a la rutina **ip**, pasándole tres argumentos, que son:

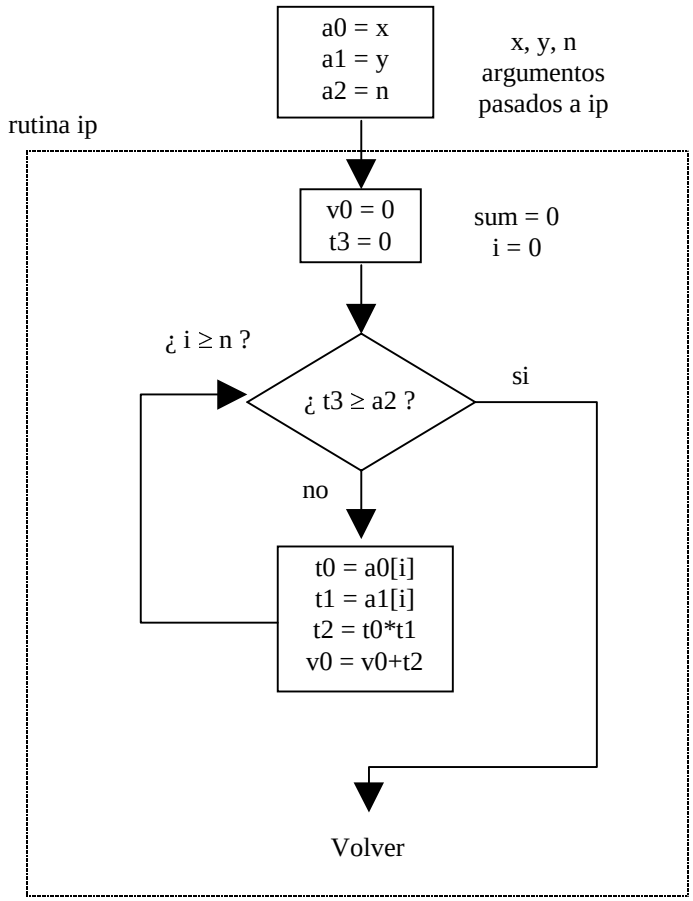
- Las direcciones base de los arrays **x** e **y**, que son arrays de enteros en memoria.
- El número entero 5

La rutina **ip** calcula el producto interior de esos arrays, es decir, la suma de los productos de los elementos de los dos arrays. Estudiar el código ensamblador equivalente (que sería, por ejemplo, el resultado de compilar el código C), observando cómo trabajan las rutinas de saltos condicionales para implementar bucles.

```
# PRAC_4A.S

        .text
        .globl  main
main:
        la      $a0,x           # $a0 = dirección base del array x
        la      $a1,y           # $a1 = dirección base del array y
        lw      $a2,size        # $a2 = tamaño del array = n
        jal     ip              # Invoca rutina ip, guarda en $ra la dir. de vuelta
        move    $a0,$v0         # Imprime el número entero que hay en $a0, y que
        li      $v0,1           # es el resultado que ip dejó en $v0
        syscall
        li      $v0,10          # Termina la ejecución del programa
        syscall

ip:     li      $v0,0           # Inicializa sum a 0. Usamos $v0 para guardar el
                                # resultado que devuelve la rutina.
        li      $t3,0           # Pone i en $t3. Usamos $t3 como índice de
```

```
                                # elementos de los arrays.
ipl:    bge     $t3,$a2,ipx     # Sale cuando i >= n (n está en $a2)
        lw      $t0,0($a0)      # Captura a0[i]
        lw      $t1,0($a1)      # Captura a1[i]
        mul     $t2,$t0,$t1
        add     $v0,$v0,$t2     # Suma a0[i]*a1[i] a sum
        addi    $a0,$a0,4       # Incrementa los punteros
        addi    $a1,$a1,4
        addi    $t3,$t3,1       # i++
        b       ipl             # Contin·a. Va a ipl incondicionalmente.
ipx:    jr      $ra             # Retorna al invocador


        .data
size:   .word   5
x:      .word   2,4,3,5,1
y:      .word   3,3,2,5,4
```

- Cuando invocamos mediante **jal** la rutina **ip**, en $ra se guarda la dirección de retorno, que es la de la siguiente instrucción a la instrucción en ejecución (jal), o sea, PC+4. Comprobarlo en el simulador cuando ejecutamos el programa paso a paso, mirando la ventana de registros (para ver el contenido de $ra y PC), y la ventana del segmento de texto (para ver las direcciones de las instrucciones).
- Observar cómo, mediante la instrucción etiquetada por **ipx:**, estamos retornando a la siguiente instrucción a la que hizo la llamada (mirar las ventanas del simulador para ver el efecto de esta instrucción en el contador de programa).
- Observar cómo el organigrama o diagrama de flujo del dibujo responde a las operaciones descritas en el código ensamblador.



B. Sea el siguiente código C:

```
#include <iostream.h>

main()
        {
        int c = 3;
        char *s;

        switch(c)
                {
                case 0: s = “negro”; break;
                case 1: s = “azul”; break;
                case 2: s = “amarillo”; break;
                case 3: s = “verde”; break;
                case 4: s = “rojo”; break;
                case 5: s = “gris”; break;
                case 6: s = “naranja”; break;
                case 7: s = “blanco”; break;
                default: s = “otros”;
                }
        cout << s << endl;
        }
```

Este código lleva la instrucción C **switch**, la cual, dependiendo del valor del entero **c**, otorga a la cadena **s** un nombre determinado de un color. Concretamente, como c=3, s=”green”. El código ensamblador equivalente será:

```
# PRAC_4B.S

                .text
                .globl  main
main:
                li      $s0, 3          # Selecciono el color

                bne     $s0, 0, c1
                la      $a0, negro
                b       cx
c1:             bne     $s0, 1, c2
                la      $a0, azul
                b       cx
c2:             bne     $s0, 2, c3
                la      $a0, amarillo
                b       cx
c3:             bne     $s0, 3, c4
                la      $a0, verde
                b       cx
c4:             bne     $s0, 4, c5
                la      $a0, rojo
                b       cx
c5:             bne     $s0, 5, c6
                la      $a0, gris
                b       cx
c6:             bne     $s0, 6, c7
                la      $a0, naranja
                b       cx
c7:             bne     $s0, 7, c8
                la      $a0, blanco
                b       cx
c8:             la      $a0, otros

cx:             li      $v0, 4          # Imprime el color
                syscall

                li      $v0, 10
                syscall
```

```
                .data
negro:          .asciiz "negro\n"
azul:           .asciiz "azul\n"
amarillo:       .asciiz "amarillo\n"
verde:          .asciiz "verde\n"
rojo:           .asciiz "rojo\n"
gris:           .asciiz "gris\n"
naranja:        .asciiz "naranja\n"
blanco:         .asciiz "blanco\n"
otros:          .asciiz "otros\n"
```

- Dibujar un organigrama o diagrama de flujo que responda a las operaciones descritas en el código ensamblador.

Práctica 5: Modos de direccionamiento.

Objetivos:

- Aprender los modos de direccionamiento de MIPS.

Desarrollo:

- MIPS es una arquitectura de carga/almacenamiento. Sólo las instrucciones de carga y almacenamiento pueden acceder a memoria, según los cálculos efectuados sobre los valores en los registros.

- Casi todas las instrucciones de carga y almacenamiento operan sólo sobre datos alineados. La alineación de una cantidad significa que su dirección de memoria es un múltiplo de su tamaño en bytes. Por ejemplo, una palabra de 32 bits debe almacenarse en direcciones múltiplo de 4. Las instrucciones para alinear datos no alineados son: lwl, lwr, swl y swr.

- Hay varias formas de direccionar un dato en memoria (estudiar la transparencia nº 10 del tutorial). La forma en que se calcula esta dirección de memoria depende del modo de direccionamiento contemplado por la instrucción de acceso a memoria. A continuación se listan los modos de direccionamiento existentes:

Formato	**Cálculo de la dirección**	**Ejemplo**
(registro)	contenido del registro (cr)	lw $t0,($t2)
valor	valor inmediato (vin)	lw $t0,0x10010008
valor (registro)	vin + cr	lw $t0,0x10010000($t1)
identificador	dirección del identificador (did)	lw $t0,array
identificador +/- valor	did +/- vin	lw $t0,array+8
identificador (registro)	did + cr	lw $t0,array($t1)
identificador +/- valor (registro)	did +/- vin + cr	lw $t0,array+4($t1)

- Hacer un programa que carge el contenido de una posición de memoria en un registro, y luego imprima el valor del registro en consola. Repetir esta operación para cada uno de los 7 modos de direccionamiento listados anteriormente, y comprobar que en la consola aparece siempre el mismo valor (pues son modos distintos de acceder a un mismo dato).

- Se recomienda enseñar al profesor el código desarrollado, con el objeto de identificar correctamente cada modo de direccionamiento.

Práctica 6: Condiciones if-then-else.

Objetivos:

- Cómo implementar la condición **if-then**.
- Cómo implementar la condición **if-then-else**.

Desarrollo:

A. Sea el siguiente código C:

```
#include <iostream.h>

main() {
  int a[] = {36, 20, 27, 15, 1, 62, 41};
  int n = 7;
  int max, i;

  for (max = i = 0; i<n; i++)
    if (a[i] > max)
      max = a[i];

  cout << max << endl;
}
```

Este programa calcula el máximo de los elementos del array **a**, almacenándolo en la variable **max**. Estudiar el código ensamblador equivalente:

```
# PRAC_6A.S

        .text
        .globl  main
main:
        li      $t0, 0          # i en $t0
        li      $s0, 0          # max en $s0
        lw      $s1, n          # n en $s1

m1:     bge     $t0, $s1, m3
        mul     $t1, $t0, 4     # Dá el valor adecuado para i
        lw      $t2, a($t1)     # Carga a[i] en $t2
        ble     $t2, $s0, m2    # Salta el "then" si a[i] <= max
        move    $s0, $t2        # "then": max = a[i]
m2:     addi    $t0, $t0, 1     # i++
        b       m1
m3:     move    $a0, $s0        # Fin del bucle
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall

        .data
a:      .word   36, 20, 27, 15, 1, 62, 41
n:      .word   7
max:    .word   0
```

- Estudiar en profundidad cómo trabaja el código. El resultado debe aparecer en consola: **62**.
- Dibujar un organigrama o diagrama de flujo que responda a las operaciones descritas.

B. Sea el siguiente código C:

```
#include <iostream.h>

main() {
  int a[] = {-36, 20, -27, 15, 1, -62, -41};
  int n = 7;
  int i;
  int npos, nneg;

  for (i = npos = nneg = 0; i<n; i++)
    if (a[i] > 0)
      npos++;
    else
      nneg++;

  cout << "+: " << npos << "; -: " << nneg << endl;
}
```

Este código cuenta, en las variable **npos** y **nneg**, cuántos números positivos y negativos, respectivamente, aparecen en el array **a**:

```
# PRAC_6B.S

        .text
        .globl  main

main:
        li      $t0, 0                  # i en $t0
        li      $t1, 0                  # npos en $t1
        li      $t2, 0                  # nneg en $t2
        lw      $s1, n                  # n en $s1

m1:     bge     $t0, $s1, m4
        mul     $t3, $t0, 4             # da el valor adecuado para i
        lw      $t4, a($t3)             # carga a[i] en $t4
        bgez    $t4, m2                 # if (a[i] > max)...
        addi    $t2, $t2, 1             # "else": incrementa nneg
        b       m3                      # salta sobre "then"
m2:     addi    $t1, $t1, 1             # "then": incrementa npos
m3:     addi    $t0, $t0, 1             # i++
        b       m1
m4:     move    $a0, $t1                # imprime npos
        li      $v0, 1
        syscall
        la      $a0, endl
        li      $v0, 4
        syscall
        move    $a0, $t2                # imprime nneg
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall

        .data
a:      .word   -36, 20, -27, 15, 1, -62, -41
n:      .word   7
max:    .word   0
endl:   .asciiz "\n"
```

- Estudiar en profundidad cómo trabaja el código. Los resultados que deben aparecer en consola son npos=3 y nneg=4.
- Dibujar un organigrama o diagrama de flujo que responda a las operaciones descritas.

Práctica 7: Otros bucles condicionales.

Objetivos:

- Cómo implementar en ensamblador las funciones C **strlen** y **strcpy** (ejemplos de cómo hacer en ensamblador los bucles condicionales **while** y **do-while**, respectivamente).

Desarrollo:

Sea el siguiente código C:

```
#include <iostream.h>

void strcpy(char *dest, char *src)
   {
   do
      *dest++ = *src++;
   while (*src);
   }

int strlen(char *s)
   {
   int n = 0;
   while (*s)
      {
      n++;
      s++;
      }
   return n;
   }

main()
   {
   char si[] = “texto”;
   char so[80];

   cout << strlen(si);
   strcpy(so,si);
   cout << so;
   }
```

- La función strcpy copia una cadena en otra posición de memoria (que es lo mismo que crear una cadena idéntica). La función strlen cuenta el número de caracteres que tiene una cadena.

- La función main declara la cadena “texto”, y reserva un buffer (memoria) de 80 bytes (80 caracteres) para que podamos copiar a él la cadena “texto”. Después, cuenta los caracteres de la cadena “texto” para, a continuación, copiarla sobre el buffer.

- El código ensamblador que hace las tareas del código C es:

```
# PRAC_7.S

        .text
        .globl  main

main:   la              $a1,si          # $a1 tiene si = dirección base
                                        # de la cadena "texto"
        jal             strlen          # Devuelve en $v0 la longitud de
                                        # la cadena que hay en $a1
        move            $t0,$v0         # Imprimo información de la longitud
        la              $a0,nota1
        li              $v0,4
        syscall
        la              $a0,si
        li              $v0,4
        syscall
        la              $a0,nota2
        li              $v0,4
        syscall
        move            $a0,$t0
        li              $v0,1
        syscall
        la              $a0,nota3
        li              $v0,4
        syscall

        la              $a1, si         # $a1 tiene si = dirección base
                                        # de la cadena "texto"
        la              $a0, so         # $a0 tiene so = dirección base
                                        # para la futura cadena
        jal             strcpy          # Copio en so la cadena que hay en si
        la              $a0, so         # Consola muestra la cadena
        li              $v0, 4          # que hay en so
        syscall

        li              $v0, 10         # Fin del programa
        syscall

        #-----------------------#
strcpy: lb              $t0,0($a1)      # strcpy -- copia la cadena
        addi            $a1,$a1,1       #       apuntada por $a1 a la
        sb              $t0,0($a0)      #       localidad apuntada por $a0
        addi            $a0,$a0,1
        bnez            $t0,strcpy
        jr              $ra
        #-----------------------#
strlen: li              $v0,0           # strlen -- calcula la longitud
str0:   lb              $t0,0($a1)      #       de la cadena apuntada
        beqz            $t0,str1        #       por $a1
        addi            $v0,$v0,1
        addi            $a1,$a1,1
        b               str0
str1:   jr              $ra
        #-----------------------#

        .data
nota1:  .asciiz         "Longitud de ("
nota2:  .asciiz         ") = "
nota3:  .asciiz         "\n"
si:     .asciiz         "texto"
so:     .space          80
```

- Estudiar el uso de la directiva *.space*
- Estudiar en profundidad cómo se implementan strcpy y strlen en ensamblador, observando cómo se construyen los bucles condicionales while y do-while.
- Observar que, para imprimir los resultados en consola, y dado que la cadena “texto” no termina con un retorno de carro, hemos puesto unas cadenas adicionales para que la presentación por consola sea más ordenada.

Práctica 8: Pilas y rutinas recursivas.

Objetivos:

- Llamada a una rutina recursiva.
- Creación y uso de marcos de pila.
- Cómo guardar y restaurar registros.
- Cómo usar el convenio **guardar-invocador**.

Desarrollo:

- Fundamentos teóricos: Estudiar detalladamente las transparencias nº 14, 15, 16, 17, 18, 19 , 20 y 21 del tutorial. Parte de la información ahí descrita ya la conocemos, tanto por la teoría de la asignatura como por el desarrollo de anteriores prácticas (por ejemplo, palabras de memoria, segmentos de datos y texto, etc.). Ahora vemos aspectos más detallados como el uso de pilas de datos, formas de introducir y sacar datos de pila, los convenios de llamadas a procedimientos, etc.

- Sea el siguiente código ensamblador:

```
# PRAC_8.S

        .text
        .globl  main
main:   #-----------------------#
        # (0)
        #-----------------------#
        subu    $sp,$sp,32              # La rutina main crea su marco de pila
        sw      $ra,20($sp)             # Mem[$sp+20]=$ra. Guardo direcc. de vuelta
        sw      $fp,16($sp)             # Mem[$sp+16]=$fp. Guardo $fp antiguo.
                                        # Estas posiciones de memoria van separadas
                                        # por 4 Bytes (estamos almacenando palabras)
        addu    $fp,$sp,32              # $fp=$sp+32. $fp apunta al comienzo del
                                        # marco de pila (donde estaba $sp antes)
        #-----------------------#
        # (1)
        #-----------------------#
        li      $a0,3                   # Pongo argumento (n=3) en $a0
        jal     fact                    # Llama fact, almacena en $ra dir. sig. inst.
        #-----------------------#
        move    $a0,$v0                 # En $v0 está el resultado. Lo imprimo
        li      $v0,1                   # en la consola.
        syscall
        #-----------------------#
        lw      $ra,20($sp)             # Restauro registros
        lw      $fp,16($sp)
        addu    $sp,$sp,32
        #-----------------------#
        # (10)
        #-----------------------#
        j       $ra
```

```
fact:   #======================#       # Rutina 'fact'
        subu    $sp,$sp,32              # Crea marco de pila
        sw      $ra,20($sp)
        sw      $fp,16($sp)
        addu    $fp,$sp,32
        sw      $a0,0($fp)              # Guardo argumento $a0 en marco de pila (n)
        #----------------------#
        # (2),(3),(4),(5)
        #----------------------#
        lw      $v0,0($fp)
        bgtz    $v0,$L2
        li      $v0,1
        j       $L1
$L2:    lw      $v1,0($fp)
        subu    $v0,$v1,1
        move    $a0,$v0
        jal     fact
        lw      $v1,0($fp)
        mul     $v0,$v0,$v1
$L1:    lw      $ra,20($sp)
        lw      $fp,16($sp)
        addu    $sp,$sp,32
        #----------------------#
        # (6),(7),(8),(9)
        #----------------------#
        j       $ra
        #----------------------#
```

Este programa calcula, mediante la rutina **fact**, el factorial de un número **n**. Tomaremos n=3, de forma que el resultado (que imprimimos por consola), será 6.

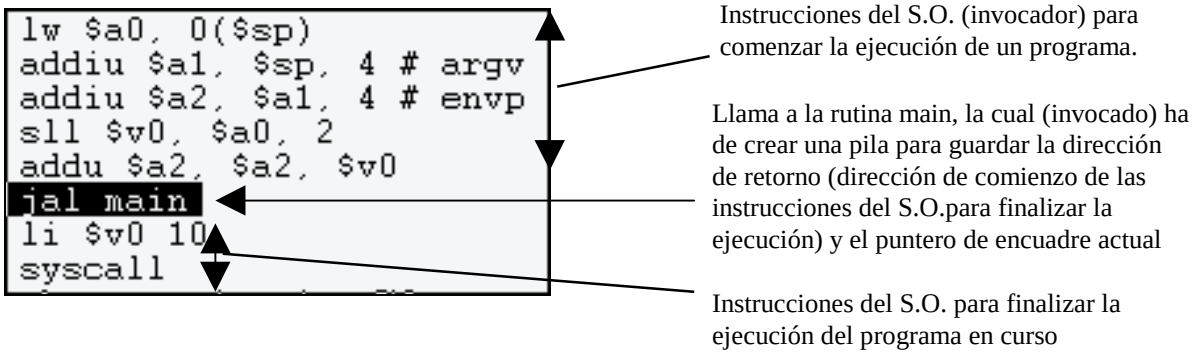
La rutina fact está implementada de forma recursiva, o sea, es una rutina que se llama a sí misma. Por tanto, ha de guardar en memoria (en una pila) sus valores actuales puesto que, al llamarse a sí misma, esos valores serán alterados, y los necesitaremos a la vuelta.

Para explicar las operaciones de este programa, nos ayudaremos del dibujo en el que se presenta la memoria y algunos registros. En él (y en el código ensamblador), rotulamos con los indicadores (0), (1), ..., (10) ciertos momentos en la ejecución del programa, con el objeto de comprobar la situación de la memoria y de los registros justo cuando se ha alcanzado este punto.

La ejecución transcurre de la siguiente forma:

- Justo antes de comenzar la ejecución de la función **main**, la situación de la memoria y los registros es la descrita en (0). Para que el programa “arranque”, hay una serie de instrucciones suministradas por el sistema operativo (invocador), que se ejecutan, y que producen que en (0) la situación de los registros sea la descrita.

- La rutina **main** (invocado) comienza creando un marco de pila para guardar la dirección de vuelta y el puntero de encuadre, que, al ser recuperados al final del programa, nos llevarán de nuevo al invocador, es decir, a las instrucciones de fin del programa:



- El tamaño mínimo de una estrcutura de pila debe ser de 32 bytes (que es bastante más grande de lo que necesitamos para nuestros propósitos).

- Después, **main** llama a la rutina **fact** (habiendo puesto en $a0, previamente, el argumento a pasarle a **fact**, y que es n=3). Cuando **fact** termine su ejecución, devolverá (en $v0) el resultado, que **main** va a imprimir por consola.

- Una vez imprimido el resultado, hay que restaurar los valores que main guardó en pila (dirección de vuelta y puntero del marco de pila), y debe hacer que el puntero de pila apunte a la posición primitiva (lo que equivale a "borrar" la pila de main). Así, y mediante la instrucción j $ra, podemos volver a la posición donde se llamó a **main**.

- En esta posición están las instrucciones necesarias para terminar la ejecución del programa, devolviendo el control al S.O.

  Vamos ahora a estudiar cómo funciona la rutina **fact**.

- Como **fact** es una rutina que se llama a sí misma, actúa como invocador e invocado al mismo tiempo, por lo que puede que nos interese guardar en pila el valor de algún registro (en nuestro caso, $a0) durante las sucesivas llamadas. Usaremos, entonces, el convenio de "guardar-invocador". :

⇒ Ejecutamos por primera vez fact, al ser llamada por main.

⇒ Al principio de fact hay que crear un marco de pila, en el que guardaremos la dirección de retorno y el puntero del marco de pila del invocador.

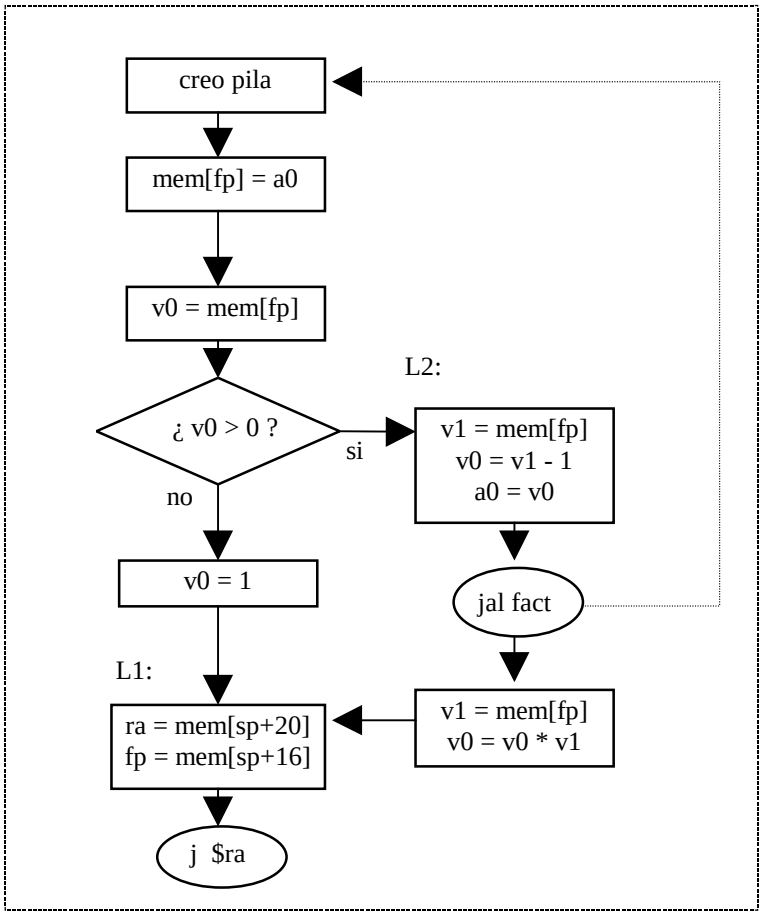
⇒ Enseguida, fact va a actuar como invocador de fact. Como usamos el convenio "guardar invocador", hemos de guardar en esta pila, antes de llamar al invocado mediante "jal fact", el valor de $a0 (argumento n). En este momento estamos en el rótulo (2).

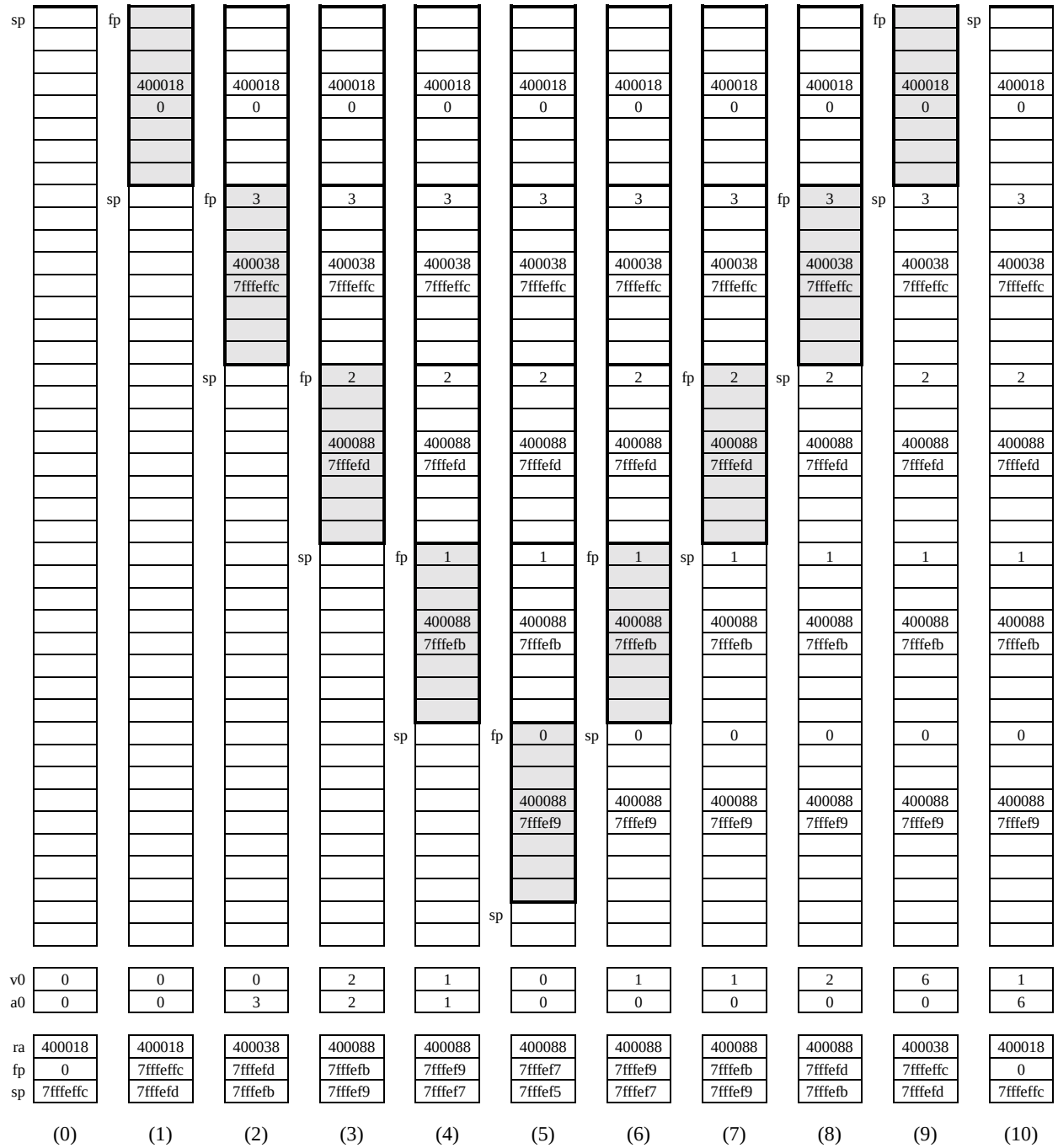
⇒ Tras unas operaciones condicionales, llamamos al invocado.

⇒ El proceso se repite pasando por los rótulos (3), (4) y (5). Como ya no vamos a volver a llamar a fact, tenemos que ir retornando a los sucesivos invocadores. Para ello, primero realizaremos las operaciones con los valores que hemos ido guardando en las pilas para calcular el factorial; después, restauraremos los registros con las direcciones de vuelta y los punteros de encuadre, y "borraremos" las pilas mediante la suma del puntero de pila. Esto está reflejado en los rótulos (6), (7), (8) y (9).

  En el siguiente esquema se puede ver cómo trabaja la rutina fact :

Rutina **fact**


creo pila
mem[fp] = a0
v0 = mem[fp]
L2:
¿ v0 > 0 ?
si
v1 = mem[fp]
v0 = v1 - 1
a0 = v0
no
v0 = 1
jal fact
L1:
ra = mem[sp+20]
fp = mem[sp+16]
v1 = mem[fp]
v0 = v0 * v1
j $ra



Propuesta. Se ha calculado el factorial de un número de esta forma (un tanto complicada) para ilustrar el uso de pilas y, sobre todo, las funciones recursivas. Hay formas mucho más sencillas de calcular el factorial de un número sin necesidad de llamadas a funciones recursivas, pilas, etc. Elaborar un código ensamblador sencillo que lo realize.

Práctica 9: Arranque de un programa.

Objetivos:

- Estudio del código de arranque de un programa, y del manipulador de instrucciones.

Desarrollo:

- El simulador SPIM nos ofrece la posibilidad de incorporar un código de arranque y un manipulador de instrucciones. Estudiar la transparencia nº 29 del tutorial.

1. Estudio del código de arranque.

- En el simulador **PCSpim** (Win95) y en el simulador **Spimsal** (Win 3.11) están los ficheros **trap.handler** y **traphand.sim**, respectivamente. En éstos está escrito el código de arranque (conjunto de instrucciones máquina) que el sistema operativo (S.O.) utiliza para arrancar el programa ensamblador escrito por el usuario. Este proceso de arranque consiste en invocar (mediante **jal**) la rutina **main**, sin argumentos. Para usar satisfactoriamente esta forma de ejecutar nuestros programas, éstos deben tener, forzosamente, la etiqueta main, como indicador de la rutina principal o cuerpo del programa de usuario.

- Visualizar (nunca modificar!!!) el fichero **trap.handler**.

- Localizar las instrucciones de arranque:

```
# Standard startup code.  Invoke the routine main with no arguments.

   .text
   .globl __start
__start:
   lw $a0, 0($sp) # argc
   addiu $a1, $sp, 4 # argv
   addiu $a2, $a1, 4 # envp
   sll $v0, $a0, 2
   addu $a2, $a2, $v0
   jal main
   li $v0 10
   syscall     # syscall 10 (exit)
```

- Si el simulador tiene desactivada la opción de cargar el fichero trap.handler (ver en el menú settings), significa que, para poder arrancar el programa, hemos de especificar como dirección de comienzo de la ejecución la dirección de la primera instrucción del programa (0x00400000).

```
      .text
      .......  # primera intr.

      ............... instrucciones
      ............... del código del
      ............... programador

   li $v0 10   # instr. para
   syscall     # terminar
```

Además, cuando el programa de usuario finalize, hemos de indicárselo mediante instrucciones de terminación de la ejecución.

- Si el simulador tiene activado el fichero trap.handler, automáticamente la ejecución comienza en la etiqueta __start (dirección 0x00400000). El código del usuario no tiene que indicar dicha etiqueta, y sí la etiqueta main:

```
      .text
      .globl main
main:
      ............... instrucciones
      ............... del código del
      ............... programador
   jal $ra  # instrucción para retornar al código de arranque
```

Además, cuando el programa de usuario ha terminado, mediante la instrucción de vuelta *jr $ra*, ejecutamos las dos últimas instrucciones del código de arranque, instrucciones que indican finalizar la ejecución, devolviendo así el control al programa del sistema operativo.
En este caso, main (invocado) debe guardar los $ra y $fp del código de arranque (invocador), y debe recuperarlos antes de volver al invocador.

2. Estudio del manipulador de excepciones.

- Ver "A.7: Excepciones e interrupciones" del libro "Organización y Diseño de Computadores, Patterson & Hennessy, para comprender el significado de los registros Cause, EPC y Status." Ver también las transparencias nº 11 y 12 del tutorial.

- Visualizar (no modificar!!!) el fichero trap.handler.

- Localizar las instrucciones del manipulador de excepciones:

```
# Define the exception handling code.  This must go first!

   .kdata
__m1_:   .asciiz "  Exception "
__m2_:   .asciiz " occurred and ignored\n"
__e0_:   .asciiz "  [Interrupt] "
__e1_:   .asciiz  ""
__e2_:   .asciiz  ""
__e3_:   .asciiz  ""
__e4_:   .asciiz  "  [Unaligned address in inst/data fetch] "
__e5_:   .asciiz  "  [Unaligned address in store] "
__e6_:   .asciiz  "  [Bad address in text read] "
__e7_:   .asciiz  "  [Bad address in data/stack read] "
__e8_:   .asciiz  "  [Error in syscall] "
__e9_:   .asciiz  "  [Breakpoint] "
__e10_:  .asciiz  "  [Reserved instruction] "
__e11_:  .asciiz  ""
__e12_:  .asciiz  "  [Arithmetic overflow] "
__e13_:  .asciiz  "  [Inexact floating point result] "
__e14_:  .asciiz  "  [Invalid floating point result] "
__e15_:  .asciiz  "  [Divide by 0] "
__e16_:  .asciiz  "  [Floating point overflow] "
__e17_:  .asciiz  "  [Floating point underflow] "
__excp:  .word    __e0_,__e1_,__e2_,__e3_,__e4_,__e5_,__e6_,__e7_,__e8_,__e9_
         .word    __e10_,__e11_,__e12_,__e13_,__e14_,__e15_,__e16_,__e17_
s1:      .word    0
s2:      .word    0

         .ktext   0x80000080
         .set     noat
                              # Because we are running in the kernel, we can use $k0/$k1
                              # without saving their old values.
         Move     $k1,$at     # Save $at
```

```
        .set    at
        sw      $v0,s1          # Not re-entrent and we can't trust $sp
        sw      $a0,s2
        mfc0    $k0,$13         # Cause
        sgt     $v0,$k0,0x44    # ignore interrupt exceptions
        bgtz    $v0 ret
        addu    $0 $0 0
        li      $v0 4           # syscall 4 (print_str)
        la      $a0 __m1_
        syscall
        li      $v0 1           # syscall 1 (print_int)
        srl     $a0 $k0 2       # shift Cause reg
        syscall
        li      $v0 4           # syscall 4 (print_str)
        lw      $a0 __excp($k0)
        syscall
        bne     $k0 0x18 ok_pc # Bad PC requires special checks
        mfc0    $a0, $14        # EPC
        and     $a0, $a0, 0x3  # Is EPC word-aligned?
        Beq     $a0, 0, ok_pc
        Li      $v0 10          # Exit on really bad PC (out of text)
        syscall

ok_pc:  li $v0 4                # syscall 4 (print_str)
        la $a0 __m2_
        syscall
        mtc0 $0, $13            # Clear Cause register
ret:    lw      $v0 s1
        lw      $a0 s2
        mfc0    $k0 $14         # EPC
        .set    noat
        move    $at $k1         # Restore $at
        .set    at
        rfe                     # Return from exception handler
        addiu   $k0 $k0 4       # Return to next instruction
        jr      $k0
```

Un manipulador de excepciones parecido, más sencillo, que nos ayudará a entender los mecanismos para tratar las excepciones, es el siguiente:

```
        .ktext 0x0000080      # (0)
        sw     $a0,save0      # (1)
        sw     $a1,save1
        mfc0   $k0,$13
        mfc0   $k1,$14
        sgt    $v0,$k0,0x44   # (3)
        bgtz   $v0,done
        mov    $a0,$k0        # (4)
        mov    $a1,$k1
        jal    print_excp
done:
        lw     $a0,save0      # (5)
        lw     $a1,save1
        addiu  $k1,$k1,4
        rfe                   # (6)
        jr     $k1            # (7)

        .kdata
save0:  .word  0
save1:  .word  1
```

Comentarios:

- (0). El manipulador de excepciones (ME) es una rutina del sistema que está almacenada en el segmento de texto del kernel (núcleo del sistema operativo, S.O.), a partir de la dirección 0x0000080.

- (1). El ME guarda $a0 y $a1, pues los usará más tarde para pasar argumentos a la función print_excp (son valores que ME va a alterar, y que habrá que restaurar justo antes de volver a la rutina de usuario que provocó la excepción). Estos registros los guarda en posiciones de memoria del segmento de datos del kernel.

- (2). Las instrucciones mfc0 transfieren registros del coprocesador 0 ($13=Cause, $14=EPC) a registros de la CPU ($k0 y $k1, que son registros reservados para el S.O.). Observar que el ME no guarda los registros $k0 y $k1, porque supone que los programas de usuario no utilizan estos registros.

- (3). Examinamos si la excepción fué provocada por una interrupción (examinando el registro Cause). Si la excepción fué una interrupción (ver diferencias entre ambas según lo que se explica en la asignatura), la ignoramos. Si la excepción no fué una interrupción, se invoca a una pequeña rutina para imprimir un mensaje.

- (4). Llama a la función de impresión de mensaje de excepción, pasando como argumentos los contenidos de los registros Cause y EPC.

- (5). Restauramos valores.

- (6). La instrucción rfe restaura los bits anteriores y la máscara de interrupción en el registro Status, cambiando a la CPU al estado en que se encontraba antes de la excepción, y preparándolo para reanudar la ejecución del programa.

- (7). Vuelve a la rutina que contiene la instrucción que provocó la excepción, bifurcando a la siguiente instrucción a aquella.