Prácticas guiadas para el Ensamblador del MIPS R2000

Autores:
Maribel Castillo Catalán
José Manuel Claver Iborra

INTRODUCCIÓN

Las prácticas de lenguaje ensamblador son cruciales en las asignaturas introductorias de arquitectura de computadores. Estas se centran en el conocimiento de la arquitectura del juego de instrucciones de un procesador y el modo en que éstas se relacionan con la programación del computador y los lenguajes de alto nivel.

La elección del procesador es muy importante para entender adecuadamente, tanto la programación a bajo nivel como el funcionamiento global del computador a partir de éste. La elección del primer procesador debe cumplir varias condiciones entre las que se deben combinar la simplicidad y sencillez, en un primer momento, con la potencia del juego de instrucciones y la generalidad del modelo arquitectural que permita fácilmente explicar arquitecturas más avanzadas o diferentes.

Las razones para la elección del procesador han estado condicionada a b largo de los años por las modas. Aquellas elecciones más difundidas, y las que han permanecido más tiempo, siempre han estado relacionadas con la aparición de procesadores que por su concepción y diseño han revolucionado el campo de los computadores. Entre ellos cabe destacar el PDP 11 de DEC, el 8085/8086 de Intel, el MC68000 de Motorola y el R2000/30000 de MIPS.

El procesador MC68000 de Motorola ha sido muy utilizado, y lo es aún hoy en día, en las prácticas de las asignaturas de arquitectura de computadores de muchas universidades, junto con el procesador 8086 de Intel, éste en menor medida. En el caso del procesador de Motorola destaca la ortogonalidad del juego de instrucciones y la existencia de diferentes modos prioritarios de funcionamiento, mientras que en el caso de Intel ha sido determinante su difusión a través de los ordenadores personales de IBM (PCs) y sus clónicos.

Pero si hay en la actualidad un procesador que sea el más extendido en el ámbito de la enseñanza de la arquitectura de los computadores, ese es el MIPS R2000. Esto es debido a que su arquitectura es la semilla de muchos de los diseños de los procesadores superescalares actuales, a la par que mantiene la simplicidad de los primeros procesadores RISC, lo que hace de éste un instrumento ideal para introducir el estudio de la arquitectura de los computadores. Con este objetivo se han elaborado unas prácticas en las que el estudiante utilizará el lenguaje ensamblador de este procesador y conocerá su funcionamiento mediante el simulador SPIM.

Este manual de prácticas se ha diseñado para ser seguido de forma casi autónoma, requiriendo únicamente puntuales aclaraciones por parte del profesor. Con ello se pretende que el estudiante pueda repasar fuera del horario de prácticas de la asignatura en la que esté matriculado, aquellas partes que no hayan quedado lo suficientemente claras.

Las prácticas propuestas en esta publicación se han dividido en los siguientes capítulos:

Capítulo 1. El simulador SPIM

Capítulo 2. Los datos en memoria

Capítulo 3. Carga y almacenado de los datos

Capítulo 4. Las operaciones aritméticas y lógicas

Capítulo 5. Operadores booleanos

Capítulo 6. Estructuras de control: condicionales y bucles

Capítulo 7. Interfaz con el programa

Capítulo 8. Gestión de subrutinas

Capítulo 9. Gestión de la E/S mediante consulta de estado

Capítulo 10. Gestión de la E/S mediante interrupciones

Para que no exista necesidad de una asistencia obligatoria a las sesiones prácticas de laboratorio, existen versiones del simulador que se va a utilizar (XSPIM) tanto para Linux como para Windows 95/98/ME, que se pueden encontrar en la dirección web de la asignatura correspondiente o a través de Internet. También es muy importante que se complemente este manual con un listado del juego completo de las instrucciones del MIPS R2000 que incluya su uso. Para ello, recomendamos el libro de Patterson/Hennessy titulado "Organización y diseño de computadores: la interficie circuitería/programación", en particular su tomo III.

ÍNDICE

INTRODUCCIÓN **1**

CAPÍTULO 1. EL SIMULADOR SPIM **5**

1.1 DESCRIPCIÓN DEL SIMULADOR XSPIM 5
1.2. SINTAXIS DEL LENGUAJE ENSAMBLADOR DEL MIPS R2000 19

CAPÍTULO 2. LOS DATOS EN MEMORIA **23**

2.1. DECLARACIÓN DE PALABRAS EN MEMORIA 23
2.2. DECLARACIÓN DE BYTES EN MEMORIA 24
2.3. DECLARACIÓN DE CADENAS DE CARACTERES 25
2.4. RESERVA DE ESPACIO EN MEMORIA 25
2.5. ALINEACIÓN DE DATOS EN MEMORIA 26

CAPÍTULO 3. CARGA Y ALMACENADO DE LOS DATOS **29**

3.1. CARGA DE DATOS INMEDIATOS (CONSTANTES) 29
3.2. CARGA DE PALABRAS (PALABRAS DE MEMORIA A REGISTRO) 30
3.3. CARGA DE BYTES (BYTES DE MEMORIA A REGISTRO) 31
3.4. ALMACENADO DE PALABRAS (PALABRAS DE REGISTRO A MEMORIA) 32
3.5. ALMACENADO DE BYTES (BYTES DE REGISTRO A MEMORIA) 33

CAPÍTULO 4. LAS OPERACIONES ARITMÉTICAS Y LÓGICAS **35**

4.1. OPERACIONES ARITMÉTICAS CON DATOS INMEDIATOS (CONSTANTES) 35
4.2. OPERACIONES ARITMÉTICAS CON DATOS EN MEMORIA 36
4.3. MULTIPLICACIÓN Y DIVISIÓN CON DATOS EN MEMORIA 36
4.4. OPERACIONES LÓGICAS 37
4.5. OPERACIONES DE DESPLAZAMIENTO 38

CAPÍTULO 5. EVALUACIÓN DE LAS CONDICIONES. VARIABLES Y OPERADORES BOOLEANOS **41**

5.1. EVALUACIÓN DE CONDICIONES SIMPLES: MENOR, MAYOR, MENOR O IGUAL, MAYOR O IGUAL 43
5.2. EVALUACIÓN DE CONDICIONES COMPUESTAS POR OPERADORES LÓGICOS “AND” Y “OR” 45

CAPÍTULO 6. ESTRUCTURAS DE CONTROL CONDICIONAL **49**

6.1. ESTRUCTURA DE CONTROL *SI-ENTONCES* CON CONDICIÓN SIMPLE 49
6.2. ESTRUCTURA DE CONTROL *SI-ENTONCES* CON CONDICIÓN COMPUESTA 51
6.3. ESTRUCTURA DE CONTROL *SI-ENTONCES-SINO* CON CONDICIÓN SIMPLE 53
6.4. ESTRUCTURA DE CONTROL *SI-ENTONCES-SINO* CON CONDICIÓN COMPUESTA 55
6.5. ESTRUCTURA DE CONTROL REPETITIVA *MIENTRAS* 56
6.6. ESTRUCTURA DE CONTROL REPETITIVA *PARA* 58

CAPÍTULO 7. INTERFAZ CON EL PROGRAMA **61**

7.1. IMPRESIÓN DE UNA CADENAS DE CARACTERES 61
7.2. IMPRESIÓN DE ENTEROS 62
7.3. LECTURA DE ENTEROS 63
7.4. LECTURA DE CADENAS DE CARACTERES 63

CAPÍTULO 8. GESTIÓN DE SUBRUTINAS **65**

8.1. GESTIÓN DE LA PILA 65
8.2. LLAMADA Y RETORNO DE UNA SUBRUTINA 67
8.3. ANIDADO DE SUBRUTINAS 68
8.4. PASO DE PARÁMETROS 71
8.5. BLOQUE DE ACTIVACIÓN DE LA SUBRUTINA 75

CAPÍTULO 9. GESTIÓN DE LA E/S MEDIANTE CONSULTA DE ESTADO 83

9.1. ENTRADA DE DATOS DESDE EL TECLADO 84
9.2. SALIDA DE DATOS POR LA PANTALLA 86
9.3. ENTRADA/SALIDA DE DATOS 87

CAPÍTULO 10. GESTIÓN DE E/S MEDIANTE INTERRUPCIONES 91

10.1 PROCESAMIENTO DE LAS EXCEPCIONES EN EL SIMULADOR SPIM DEL MIPS.. 91
10.2 DESCRIPCIÓN DEL CONTROLADOR DEL TECLADO SIMULADO POR SPIM 94
10.3 CONTROL DE ENTRADA DE DATOS MEDIANTE INTERRUPCIONES 94

CAPÍTULO 1. EL SIMULADOR SPIM

En esta primera parte se describe, en primer lugar, el simulador SPIM utilizado para ejecutar los programas desarrollados en lenguaje ensamblador para los computadores basados en el procesador MIPS R2000. A continuación, se describe un conjunto de conceptos básicos para la programación en lenguaje ensamblador que se irá ampliando a medida que se avance en los capítulos de este manual. Así pues, este capítulo se divide en dos apartados que contienen:

- La descripción del simulador SPIM en sus dos versiones para Linux y para Windows.
- La descripción de los recursos de programación básicos que permite la programación en ensamblador.

1.1 Descripción del Simulador XSPIM

El SPIM (MIPS al revés) es un simulador que ejecuta programas en lenguaje ensamblador de los computadores basados en los procesadores MIPS R2000/R3000. La arquitectura de este tipo de procesadores es RISC, por lo tanto simple y regular, y en consecuencia fácil de aprender y entender.

La pregunta obvia en estos casos es por qué se va a utilizar un simulador y no una máquina real. Las razones son diversas: entre ellas cabe destacar la facilidad de poder trabajar con una versión simplificada y estable del procesador real. Los procesadores actuales ejecutan varias instrucciones al mismo tiempo y en muchos casos de forma desordenada, esto hace que sean más difíciles de comprender y programar.

El simulador a utilizar en prácticas es una versión “X” del SPIM, es decir, una versión gráfica con ventanas, denominada XSPIM.

La instalación del XSPIM es sencilla:

Windows: Ejecuta el programa “spimwin.exe”. Se realizará la instalación y sólo habrá que ejecutar el icono “PCSpim for Windows” para arrancar el programa.

Linux: Ejecuta “rpm –i spim.rpm”. Una vez realizada la instalación, el programa se ejecuta mediante “xspim”.

A continuación se pasa a describir de forma más detallada el simulador en cada una de estas plataformas.

Versión para Linux

```
xspim
PC     = 00400000    EPC    = 00000000    Cause  = 00000000    BadVAddr= 00000000
Status = 00000000    HI     = 00000000    LO     = 00000000
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
R6  (a2) = 00000000  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
                         Double Floating Point Registers
FP0  = 0            FP8  = 0            FP16 = 0            FP24 = 0
FP2  = 0            FP10 = 0            FP18 = 0            FP26 = 0
FP4  = 0            FP12 = 0            FP20 = 0            FP28 = 0
FP6  = 0            FP14 = 0            FP22 = 0            FP30 = 0
                         Single Floating Point Registers

 quit    load    reload    run    step    clear
 set value    print    breakpoints    help    terminal    mode

                              Text Segments
[0x00400000]  0x8fa40000  lw $4, 0($29)              ; 102: lw $a0, 0($sp)
[0x00400004]  0x27a50004  addiu $5, $29, 4           ; 103: addiu $a1, $sp, 4 #
[0x00400008]  0x24a60004  addiu $6, $5, 4            ; 104: addiu $a2, $a1, 4 #
[0x0040000c]  0x00041080  sll $2, $4, 2              ; 105: sll $v0, $a0, 2
[0x00400010]  0x00c23021  addu $6, $6, $2            ; 106: addu $a2, $a2, $v0
[0x00400014]  0x0c000000  jal 0x00000000 [main]      ; 107: jal main
[0x00400018]  0x3402000a  ori $2, $0, 10             ; 108: li $v0 10
[0x0040001c]  0x0000000c  syscall                    ; 109: syscall

        KERNEL
                              Data Segments

        DATA
[0x10000000]...[0x10020000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA
[0x90000000]                    0x78452020  0x74706563  0x206e6f69  0x636f2000
[0x90000010]                    0x72727563  0x61206465  0x6920646e  0x726f6e67
SPIM Version 6.0 of July 21, 1997
Copyright 1990-1997 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/trap.handler
```


El XSPIM, en la versión de Linux, posee una interfaz gráfica, dividida en cinco ventanas:

a) La parte superior se llama *register display* (ventana de registros). Muestra los valores de todos los registros de la CPU y la FPU (unidad de coma flotante) del MIPS.

b) La siguiente parte llamada *control buttoms* (ventana de botones) contiene los botones de control que permiten gestionar el funcionamiento del simulador,

c) El siguiente apartado llamado *text segments* (segmentos de texto), muestra las instrucciones del programa de usuario y del núcleo del sistema (*Kernel*) que se carga automáticamente cuando *xspim* empieza su ejecución.

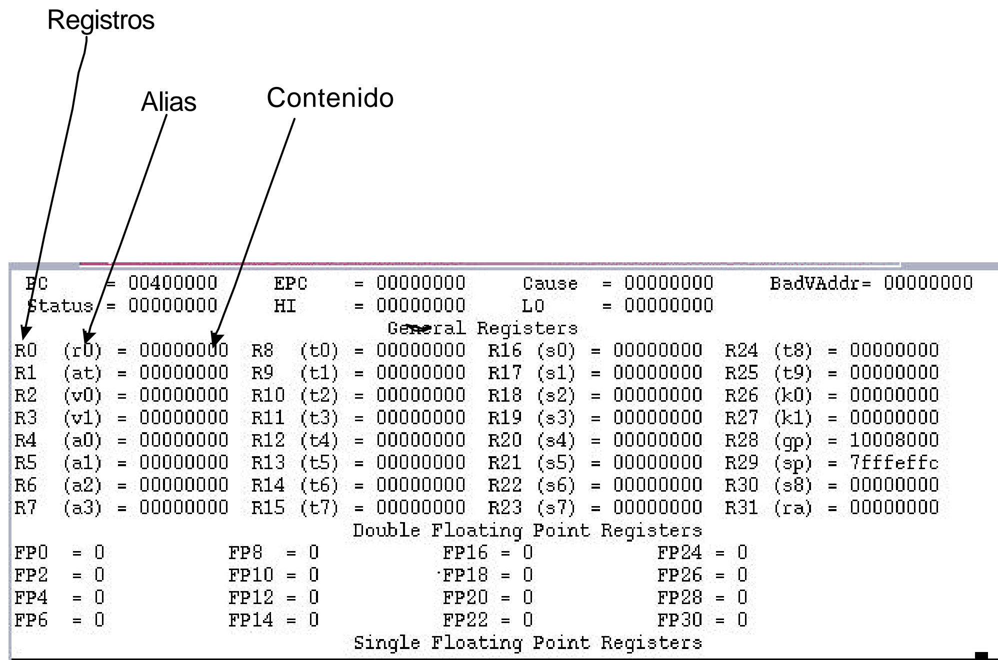
d) La siguiente parte, llamada *data segments* (segmento de datos y pila) muestra los datos de la memoria y de la pila del programa de usuario cargado en el simulador.

e) El apartado inferior es el de mensajes de XSPIM (ventana de mensajes), que *xspim* usa para escribir todo tipo de información generada durante la ejecución del mismo.

Cada una de estas ventanas contiene la siguiente información:

***a)* Ventana de registros.**

En este panel se muestran el nombre y el contenido de los registros enteros de la CPU y de la FPU (unidad de coma flotante). Estos son: los registros **R0** a **R31**, con sus correspondientes alias entre paréntesis, los registros de coma flotante, **FP0** a **FP31**, los registros de control de la CPU (**BadVAddr**, **Cause**, **Status**, **EPC**) y los registros especiales para la multiplicación y división entera, **HI** y **LO**. La longitud de cada uno de estos registros es de 32 bits (4 bytes).



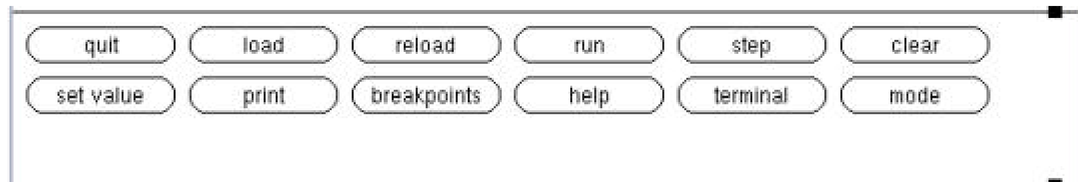
Las características de la información contenida en esta ventana son las siguientes:

- El contenido de los registros siempre se mostrará en hexadecimal.
- Cuando se ejecuta *xspim*, o se inicialicen los registros, el contenido de estos será cero, excepto el del registro que apunta a la pila (R29) que contiene el valor 0x7fffeffc.

- Esta ventana se actualiza siempre que el simulador detiene la ejecución de un programa con los resultados obtenidos durante la misma.

b) Ventana de botones de control

En este panel se pueden observar los botones que permiten controlar el funcionamiento del simulador. Son los siguientes:



- **quit**: Termina la sesión del simulador.
- **load**: Lee un fichero en lenguaje ensamblador, lo ensambla y lo carga en la memoria. Cuando se pulsa este botón se pide el nombre del fichero a cargar.
- **run:** Ejecuta el programa ensamblado y cargado en memoria. Esta opción pide como parámetro la dirección de memoria a partir de la cual se quiere que empiece la ejecución del programa simulado. Por defecto esta ejecución empieza en la dirección 0x00400000. A partir de esta dirección se encuentran almacenadas las instrucciones que se cargan automáticamente cuando se ejecuta *xspim*, y que constituyen el código de inicio que sirve para invocar el programa de usuario que se pretende ejecutar.
- **step:** Esta opción es útil cuando se quiere depurar programas, puesto que permite realizar una ejecución de las instrucciones una a una (ejecución paso a paso), y comprobar el resultado de la ejecución de cada una de las instrucciones. Esta opción pide como parámetro el número de instrucciones que ejecutará el simulador de forma continuada. Si éste es 1 la ejecución se realizará instrucción a instrucción.
- **clear:** Reinicializa el contenido de los registros y/o la memoria, poniéndolo todo al valor 0, excepto el registro sp (R29).
- **set value:** Fuerza a un valor el contenido de un registro o posición de memoria.
- **print:** Imprime el valor de registros o memoria.
- **breackpoint:** Introduce o borra puntos de ruptura (parada) en la ejecución de un programa. Esta opción permite detener la ejecución de un programa justo antes de ejecutar una instrucción particular. Habrá que pasarle como parámetro la dirección de la instrucción donde se quiere detener la ejecución. Se pueden añadir tantos puntos de ruptura como se necesiten.

- **help:** Imprime un mensaje de ayuda.

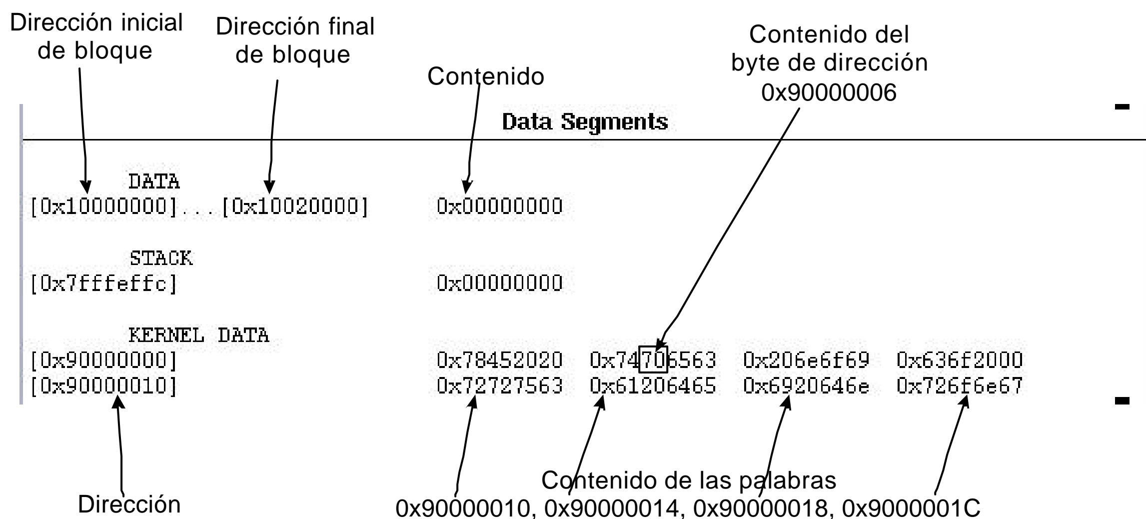
- **terminal:** Visualiza o esconde la ventana de consola (llamada terminal). Si el programa de usuario lee o escribe del terminal, *xspim* crea otra ventana, todo los caracteres que escriba el programa aparecen en ella y cualquier cosa que se quiera introducir al programa se debe teclear en esta ventana.

- **mode:** Modifica los modos de funcionamiento del XSPIM. Estos son: quiet y bare. El primero hace que el simulador no escriba un mensaje en la ventana de mensajes cuando se produzca una excepción. El segundo simula una máquina MIPS pura sin pseudoinstrucciones o modos de direccionamiento adicionales ofrecidos por el ensamblador. Ambas opciones deben estar desactivadas para simular nuestras ejecuciones.

c) Ventana del segmento de datos y pila.

En esta ventana se muestran las direcciones y datos almacenados en las zonas de memoria de datos de usuario (a partir de la dirección 0x10000000 hasta la dirección 0x10020000), del núcleo del simulador (a partir de la dirección 0x90000000) y de la pila (referenciada mediante el registro sp, y creciendo hacia direcciones decrecientes de memoria a partir de la dirección 0x7fffeffc).

Cuando un bloque de memoria contiene la misma información, se muestra la dirección inicial (columna de la izquierda), final (siguiente columna) y el contenido del bloque (columna de la derecha). En otro caso, se muestra en una misma fila el contenido de 4 palabras de memoria, indicando en la columna de más a la izquierda la dirección de la primera palabra y en las cuatro columnas siguientes el contenido de cada una de las cuatro palabras. La longitud de una palabra de memoria es de 4 bytes.



Las características de la información contenida en esta ventana son las siguientes:

- El contenido de las posiciones de memoria se muestra en hexadecimal.

➢ Cuando se ejecuta el simulador sin cargar ningún programa o al pulsar el botón *clear* con la opción *register & memory* el contenido del segmento de datos de usuario y de la pila se inicializan a cero.

d) **Ventana del segmento de texto**

Se muestran cada una de las direcciones, el código máquina, las instrucciones ensambladas y el código fuente del programa del usuario (a partir de la dirección 0x00400000) y del núcleo del simulador (a partir de la dirección 0x80000000).

A partir de la dirección 0x004000000, cuando se ejecuta el *xspim* se cargan una serie de instrucciones (código de inicio, `start-up`), que ayudan a la ejecución y depuración del código del usuario.

Dirección

Instrucción máquina

Instrucción en ensamblador

Instrucción o pseudoinstrucción del código fuente

```
                         Text Segments
[0x00400000]    0x8fa40000  lw $4, 0($29)          ; 102: lw $a0, 0($sp)
[0x00400004]    0x27a50004  addiu $5, $29, 4       ; 103: addiu $a1, $sp, 4 #
[0x00400008]    0x24a60004  addiu $6, $5, 4        ; 104: addiu $a2, $a1, 4 #
[0x0040000c]    0x00041080  sll $2, $4, 2          ; 105: sll $v0, $a0, 2
[0x00400010]    0x00c23021  addu $6, $6, $2        ; 106: addu $a2, $a2, $v0
[0x00400014]    0x0c000000  jal 0x00000000 [main]  ; 107: jal main
[0x00400018]    0x3402000a  ori $2, $0, 10         ; 108: li $v0 10
[0x0040001c]    0x0000000c  syscall                ; 109: syscall

        KERNEL
```

En esta ventana se muestra la información organizada en columnas, la de más a la derecha indica la instrucción o pseudoinstrucción del código fuente, la siguiente muestra la instrucción o instrucciones en ensamblador por las que el simulador ha traducido la instrucción de la columna anterior, a continuación se muestra la instrucción máquina, y finalmente la dirección de memoria donde está almacenada esta instrucción máquina.

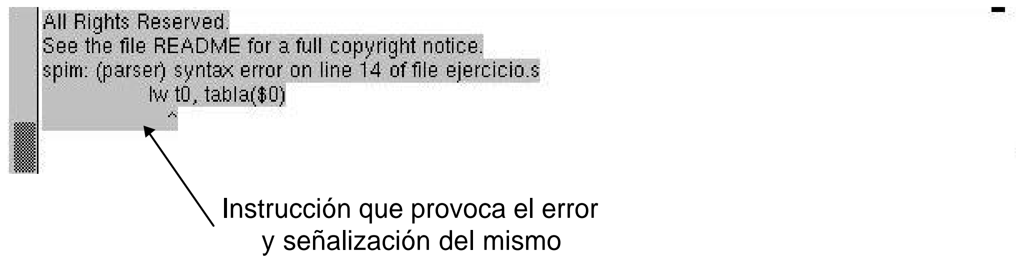
e) **Mensajes del simulador**

En este panel se observan los diversos mensajes que comunica el simulador, que nos tendrán informados del resultado y evolución de las acciones que éste lleva a cabo. En esta ventana aparecerán los mensajes de error que se generan. Estos pueden darse durante el ensamblado y carga del programa o durante la ejecución de éste en el simulador.

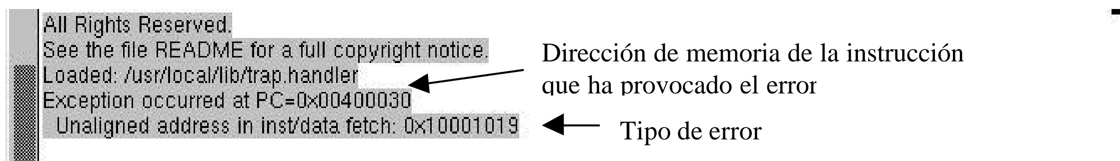
Si el programa se carga en memoria sin ningún tipo de error, en esta ventana se muestra el siguiente mensaje:

```
SPIM Version 6.2 of January 11, 1999
Copyright 1990-1998 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/lib/trap.handler
```

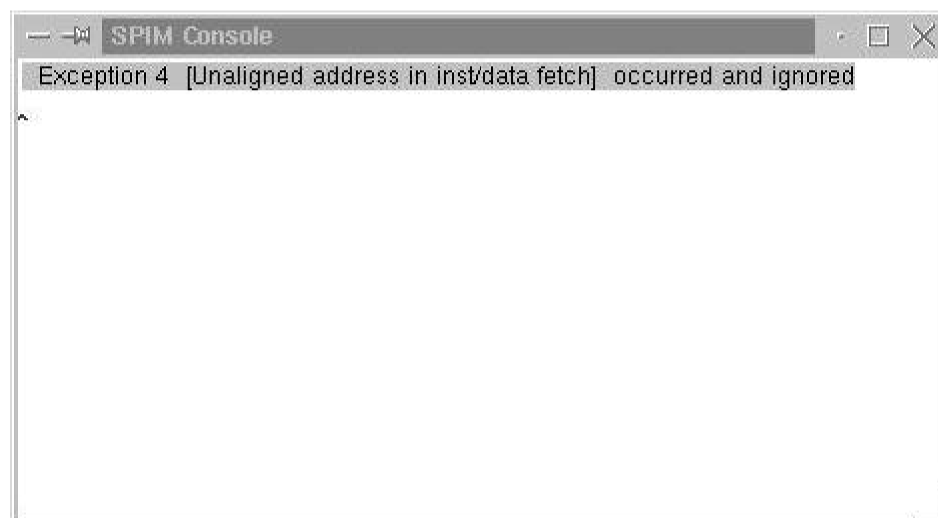
En cambio si durante la traducción del programa ensamblador a lenguaje máquina se encuentra algún tipo de error, en esta ventana se muestra la instrucción que ha provocado el error y una señalización del mismo. La aparición de este error indica que el resto del programa no se ha traducido a lenguaje máquina y por lo tanto tampoco se ha cargado en memoria.



Si el error se produce durante la ejecución del programa, en esta ventana se muestra un mensaje indicando el error producido,



y se abre otra ventana indicando el tipo de excepción que genera este error.



Carga y ejecución de programas

Los ficheros de entrada a *xspim* son de tipo texto ASCII, que incluyen las instrucciones en ensamblador del programa que se quiere simular su ejecución.

Para cargar un programa se selecciona en la ventana de botones el botón **load**, con lo que aparecerá un cuadro de diálogo donde se especifica el nombre del fichero que contiene el código del programa a ejecutar.

Al cargar cualquier programa de usuario en memoria, éste se puede cargar solo o junto con el manejador de excepciones. En el primer caso (-notrap) el programa de usuario se almacenará a partir de la dirección 0x00400000.

En el segundo caso, opción por defecto (-trap), antes de las instrucciones del programa de usuario, se carga un conjunto de instrucciones, llamado código de inicio (start-up), que permite lanzar a ejecución el programa de usuario y gestionar las excepciones asociadas a los errores producidos durante la ejecución del código. En este caso, al cargar el programa en memoria, el código del programa de usuario se cargará a partir de la dirección 0x00400020 y el código de inicio se cargará a partir de la dirección 0x00400000.

Para ejecutar el programa de forma continuada, hasta su finalización, se selecciona el botón **run** en la ventana de botones, lo que hará que *xspim* empiece a simular la ejecución del programa cargado. Previamente pedirá que se le indique la dirección de comienzo del programa (en hexadecimal). En nuestro caso será normalmente la dirección 0x00400000 (donde comienza el segmento de texto con el código de inicio). Si el código de usuario no tiene definida la etiqueta `main`, el código de inicio no lanza correctamente su ejecución y se muestra un mensaje que alerta de esta situación.

Si el programa incluye operaciones de lectura o escritura desde el terminal, *xspim* abre una ventana independiente, llamada *terminal*, a través de la cual se realiza la entrada/salida (se simula un terminal de la máquina MIPS).

Depuración de programas

Si un programa no hace lo que se esperaba, hay algunas herramientas del simulador que ayudarán a depurar el programa. Con la opción **Step** (en la ventana de botones) es posible ejecutar las instrucciones del programa una a una (paso a paso). Esta opción permite seleccionar cuantas instrucciones se desean ejecutar de forma continuada antes de que el simulador detenga la ejecución del programa y actualice las ventanas de registros y datos con los resultados obtenidos durante la misma. Esto permite verificar el contenido de los registros, la pila, los datos, etc., en cada paso.

El simulador *xspim* también permite ejecutar todas las instrucciones de un programa hasta llegar a un determinado punto, denominado *breakpoint* (punto de ruptura), a partir del cual se puede recuperar el control del programa y, por ejemplo, continuar paso a paso. Para ello, se selecciona la opción **Breakpoints** (en la ventana de botones). Una vez seleccionada esa opción, *xspim* muestra una ventana en la que pide la(s) dirección(es) en la(s) que se quiere que el programa se detenga, para recuperar el control sobre el mismo. Se debe mirar cuál es la dirección en que interesa parar el programa, en la ventana del segmento de texto, e introducirla (en hexadecimal) en la ventana, pulsando a continuación la tecla **Add**, para añadir dicho breakpoint. Se pueden introducir tantos puntos de ruptura como se desee.

Una vez encontrado el error y corregido, se vuelve a cargar el programa con la opción **Reload**. Con **Clear Registers** se pone el contenido de los registros a cero (excepto sp), mientras que **Set Value** permite cambiar el valor actual de un registro o

de una posición de memoria por un valor arbitrario. Todas estas opciones están en la ventana de botones.

A lo largo de cada una de las partes que constituyen estas prácticas se irá ampliando y aclarando el funcionamiento del simulador y aprendiendo cómo se programa en el ensamblador del R2000.

Aunque las prácticas se van a desarrollar utilizando la versión del Linux de este simulador, en el apartado siguiente se describe brevemente la versión de Windows, que el alumno puede instalar también fácilmente.

Opciones de XSPIM en la línea de comandos

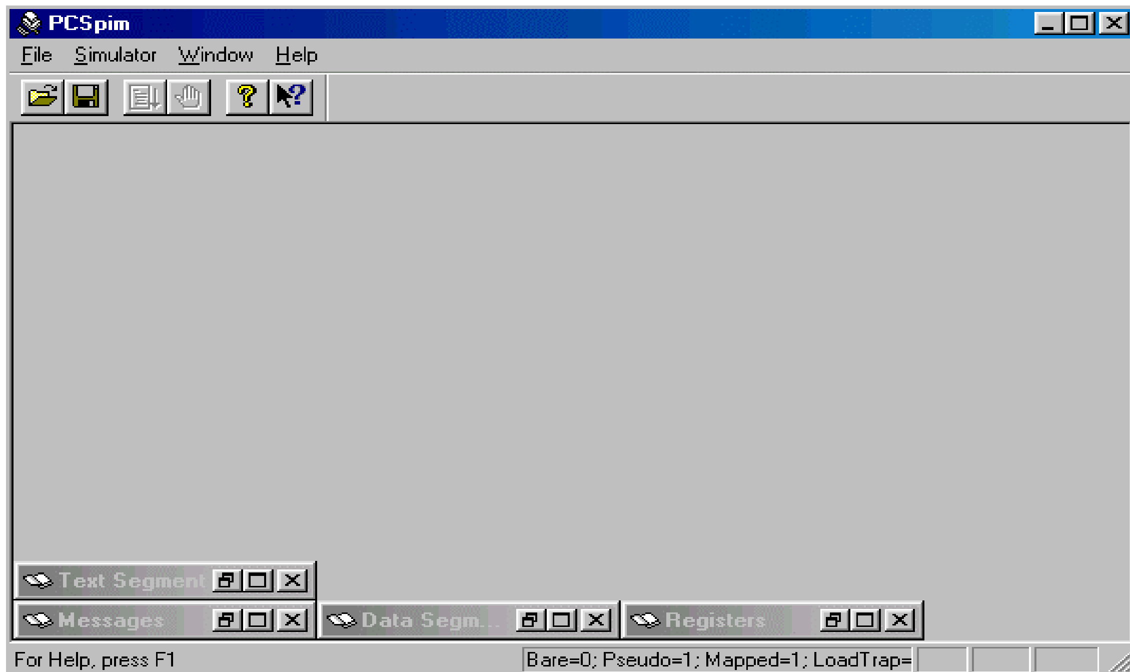
El simulador XSPIM acepta las siguientes opciones en la línea de comandos:

- -bare: Simula una máquina pura, sin pseudoinstrucciones o modos de direccionamiento adicionales ofrecidos por el programa ensamblador. La opción opuesta es -asm, que simula la máquina virtual MIPS que ofrece el programa ensamblador.
- -notrap: No carga la rutina de atención a las excepciones ni el código de inicio. Cuando se produce una excepción, XSPIM salta a la posición 0x80000080, que debe contener código para servir la excepción. La opción opuesta es -trap que si que carga la rutina de atención a las excepciones y el código de inicio.
- -nomapped_io: Deshabilita la utilidad de E/S ubicada en memoria (opción por defecto). En este caso el manejo de las operaciones de E/S serán gestionadas por el sistema a través de la llamada *syscall*. La opción opuesta es -mapped_io, que deberá estar activada cuando la gestión de la E/S la realice al usuario a través de los puertos asociados a cada dispositivo.

Versión para windows

En este apartado se presenta el uso del simulador para Windows, como variante de los programas *spim* y *xspim* de Unix. Para instalar la versión bajo Windows del simulador de MIPS, se ejecuta el fichero de instalación *spimwin.exe* que se puede encontrar en la dirección web de la asignatura.

Al ejecutar *PCSpim* aparece la interfaz que se muestra a continuación. A diferencia de la versión de Linux, no presenta desplegadas cada una de las ventanas que la componen, segmento de texto, segmento de datos, ventana de registros y ventana de mensajes. Para ver cada una de ellas hay que abrirlas explícitamente.



Esta ventana se divide en cuatro partes:

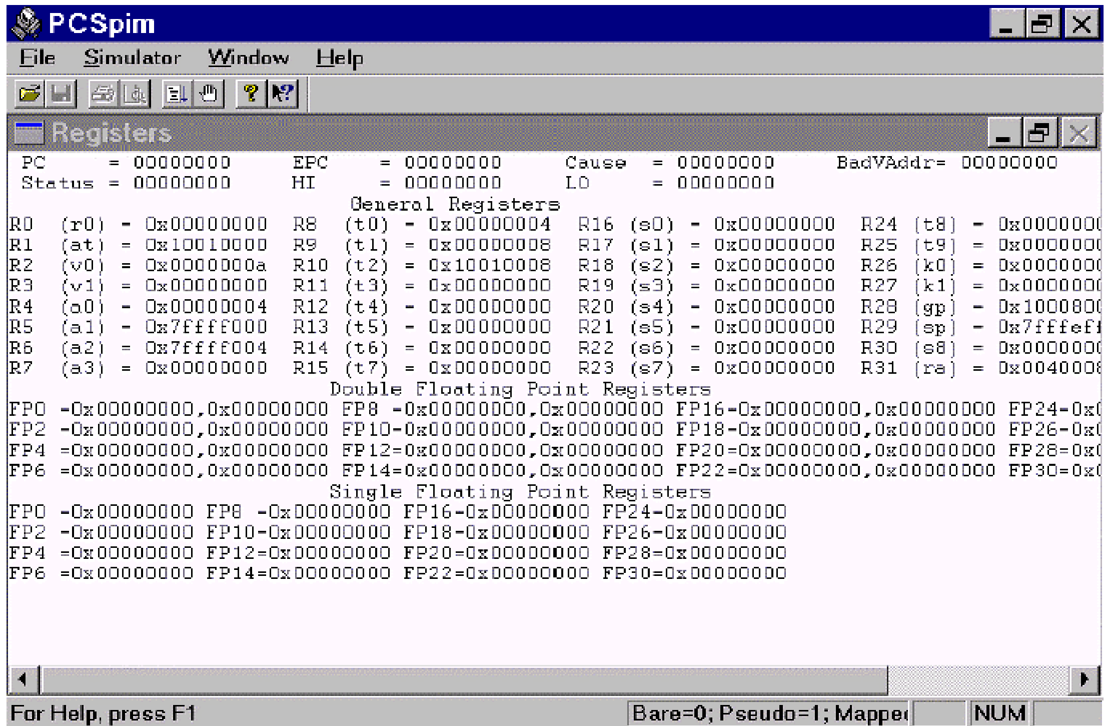
1. La parte superior es la **barra de menús** que permite:

   - acceder a las operaciones con ficheros (*menú File*),
   - especificar las opciones del simulador (*menú Simulator*),
   - seleccionar la forma de visualización de las ventanas incluidas en la ventana principal (*menú Window*), y
   - obtener información de ayuda (*menú help*)

2. Debajo de la barra de menús se encuentra la **barra de herramientas** que incluye en forma de botones algunas de las acciones más comunes en PCSpim.

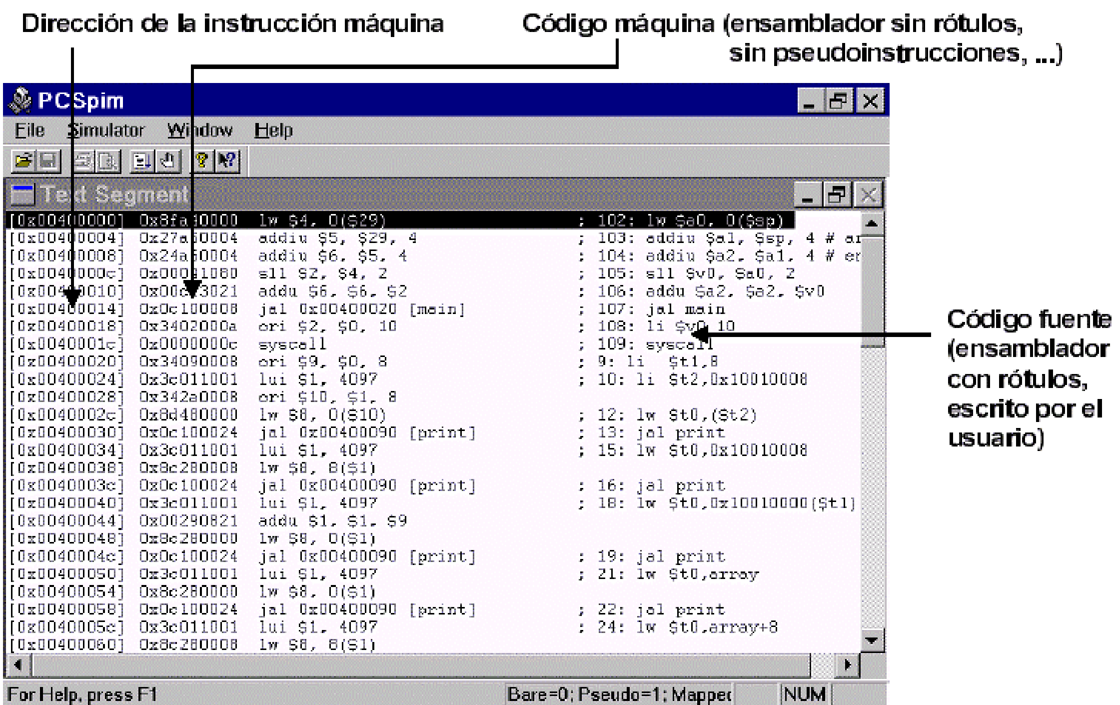
3. La parte central de la ventana de la aplicación sirve para visualizar las cuatro ventanas: **Registros** (*Registers*), **Segmento de Texto** (*Text Segment*), **Segmento de datos** (*Data Segment*) y **Mensajes** (*Messages*). Estas cuatro ventanas son similares a las que se han descrito en el apartado anterior (versión de Linux del simulador). La única diferencia con la versión Linux consiste en que en la de Windows las ventanas pueden estar abiertas o cerradas de forma independiente. A continuación se muestran cada una de ellas:

   - **Ventana de Registros** (*Registers*)

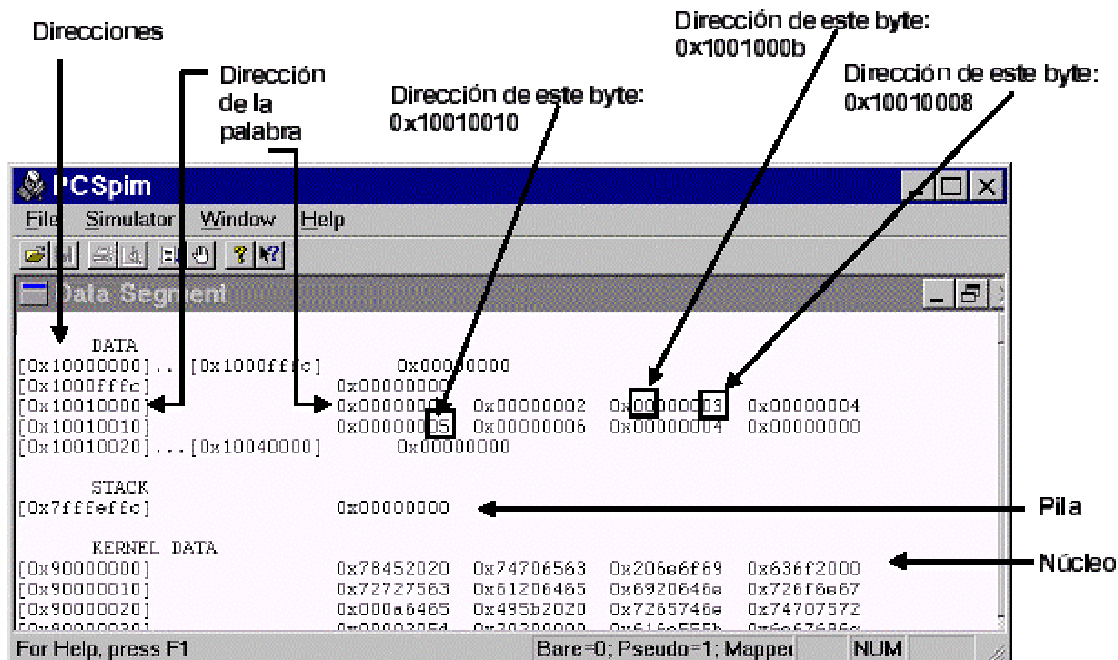


Por defecto la información contenida en los registros se visualiza en decimal. Mediante las opciones del simulador se puede cambiar a hexadecimal.

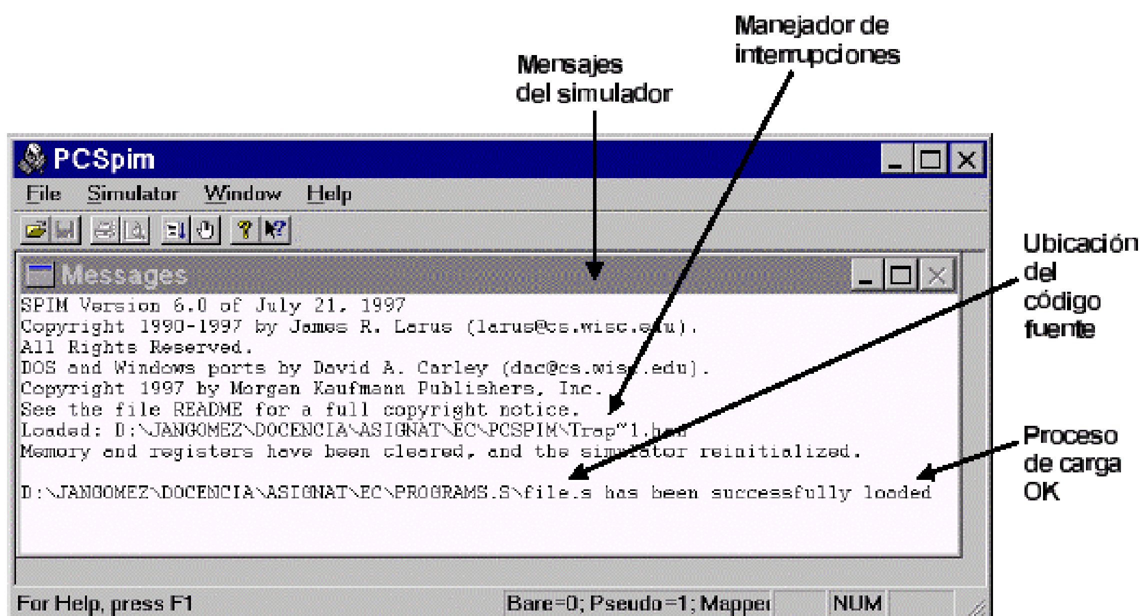
- **Segmento de texto** (*Text Segment*)



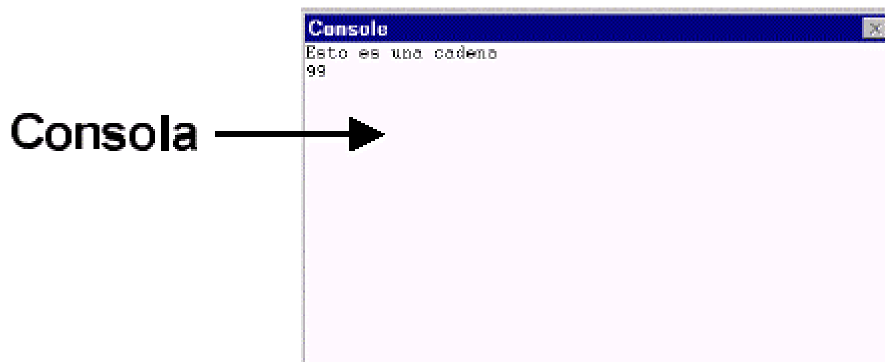
- **Segmento de datos** (*Data Segment*)



- **Mensajes** (*Messages*)



Existe una quinta ventana llamada Consola (terminal en la versión linux), independiente, a la que se accede con la opción **Window → Console**, y que sirve para realizar la entrada/salida del programa simulado.



Carga y ejecución de programas

Los ficheros de entrada a *PCSpim* también son de tipo texto ASCII, e incluyen las instrucciones en ensamblador del programa que se desea ejecutar.

Para cargar un programa se selecciona **File → Go** (o el botón **Open** de la barra de herramientas, con el icono de la carpeta abriéndose) con lo que aparecerá un cuadro de diálogo donde se puede seleccionar el fichero que se quiere abrir. Si en el código de usuario no aparece la etiqueta `main` y el simulador tiene activa la opción "Load trap file", el programa no será cargado y aparecerá un mensaje de error.

Para ejecutar el programa se selecciona, **Simulator → Go** (o el botón **Go** de la barra de herramientas, con el icono de un programa con una flecha que indica ejecución), hará que *PCSpim* comience a simularlo. Previamente pedirá que se le indique la dirección de comienzo del programa (en hexadecimal). En nuestro caso este valor será normalmente 0x00400000 (donde comienza nuestro segmento de texto). Si se desea detener la ejecución del programa se selecciona **Simulator → Break** (Ctrl-C). Para continuar con la ejecución, **Simulator → Continue**.

Si el programa incluye operaciones de lectura o escritura desde el terminal, *PCSpim* despliega una ventana independiente llamada *Console*, a través de la cual se realiza la entrada/salida (se simula un terminal de la máquina MIPS).

Depuración de programas

Si un programa no hace lo que se esperaba, hay algunas herramientas del simulador que ayudarán a depurar el programa. Con la opción **Simulator → Single Step** (o bien pulsando la tecla F10) es posible ejecutar las instrucciones del programa una a una (paso a paso). Esto permite verificar el contenido de los registros, la pila, los datos, etc., tras la ejecución de cada instrucción. Utilizando **Simulator → Multiple Step** se consigue ejecutar el programa un número determinado de instrucciones.

El programa *PCSpim* también permite ejecutar todas las instrucciones de un programa hasta llegar a un determinado punto, denominado *breakpoint* (punto de ruptura), a partir del cual se puede recuperar el control del programa y, por ejemplo, continuar paso a paso. Para ello, se selecciona **Simulator → Breakpoints** (o el botón **Breakpoints** de la barra de herramientas, con el icono de una mano indicando

detención). Una vez seleccionada esa opción, PCSpim muestra una ventana en la que pide la(s) dirección(es) en la(s) que se quiere que el programa se detenga, para recuperar el control sobre el mismo. Se debe mirar cuál es la dirección en que interesa parar el programa, en la ventana del segmento de texto, e introducirla (en hexadecimal) en la ventana, pulsando a continuación la tecla **Add**, para añadir dicho breakpoint. Se pueden introducir tantos puntos de ruptura como se desee.

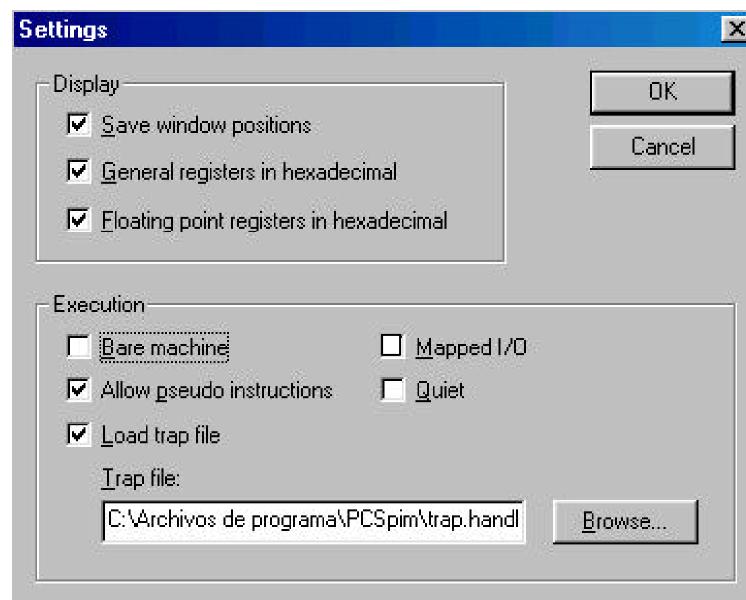
Una vez encontrado el error y corregido, se vuelve a cargar el programa con **Simulator → Reload** <nombre_fichero>. Con **Simulator → Clear Registers** se pone el contenido de los registros a cero (excepto sp), mientras que **Simulator → Set Value** permite cambiar el valor actual de un registro o de una posición de memoria por un valor arbitrario.

Otras opciones útiles disponibles en el menú principal son las contenidas en el menú de ventana (*Window*). Con ellas se pueden mostrar y ocultar las barras de herramientas y de estado, así como las distintas ventanas del simulador, organizar visualmente las mismas y limpiar la consola de entrada/salida.

El *PCSpim* también proporciona ayuda en línea (Opción **Help → Help Topics**), que muestra un documento con ayuda sobre el programa, y el icono de la barra de herramientas con una flecha y una interrogación, que sirve para pedir ayuda sobre una opción concreta de los menús.

Opciones del simulador

Al elegir la opción **Simulator → Setting** se muestran las diversas opciones que ofrece el simulador. El *PCSpim* utiliza estas opciones para determinar cómo cargar y ejecutar los programas. Una vez escogida esta opción aparece la siguiente ventana



La parte superior de la ventana (*Display*) permite:

- Salvar la posición de las ventanas al salir, y recuperarla al ejecutar de nuevo PCSpim (opción ***Save window positions*** activa)

- Fijar el formato de los registros en hexadecimal (opciones ***General registers/Floating point registers in hexadecimal*** activas). En caso de no estar seleccionada esta opción, el contenido se muestra en decimal (con signo, es decir, interpretado en complemento a 2).

La parte inferior (*Execution*) permite fijar distintos modos de funcionamiento del simulador. Para que la simulación de nuestros programas sea correcta, estas opciones deben estar activadas o desactivadas como se muestra en la figura. La opción **Bare machine** simula el ensamblador sin pseudoinstrucciones o modos de direccionamiento suministrados por el simulador. La opción **Allow pseudo instrucctions** determina si se admiten las pseudoinstrucciones. **Load trap file** indica si se carga el manejador de intrerrupciones (fichero *trap.handler*). En tal caso, cuando se produce una excepción, *PCSpim* salta a la dirección 0x80000080, que contiene el código necesario para tratar la excepción. La opción **Mapped I/O** permite seleccionar si se activa la entrada/salida mapeada en memoria. Los programas que utilizan llamadas al sistema (`syscall`), para leer o escribir en el terminal, deben desactivar esta opción. Aquellos otros programas que vayan a hacer entrada/salida mediante mapeado en memoria deben tenerla activada. Finalmente la opción **Quiet** permite seleccionar que *PCSpim* no imprima mensaje alguno cuando se producen las excepciones.

1.2. Sintaxis del lenguaje ensamblador del MIPS R2000

El *lenguaje ensamblador* es la representación simbólica de la codificación binaria de un computador, *el lenguaje máquina*. El lenguaje máquina está constituido por *instrucciones máquina* que indican al computador lo que tiene que hacer, es decir, son las órdenes que el computador es capaz de comprender. Cada instrucción máquina está constituida por un conjunto ordenado de unos y ceros, organizados en diferentes campos. Cada uno de estos campos contiene información que se complementa para indicar al procesador la acción a realizar.

El lenguaje ensamblador ofrece una representación más próxima al programador y simplifica la lectura y escritura de los programas. Las principales herramientas que proporciona la programación en ensamblador son: la utilización de nombres simbólicos para operaciones y posiciones de memoria, y unos determinados recursos de programación que permiten aumentar la claridad (legibilidad) de los programas. Entre estos recursos cabe destacar: la utilización de directivas, pseudoinstrucciones y macros que permiten al programador ampliar el lenguaje ensamblador básico definiendo nuevas operaciones.

Una herramienta denominada *programa ensamblador* traduce un programa fuente escrito de forma simbólica, es decir, en lenguaje ensamblador, a instrucciones máquina. El programa ensamblador lee un fichero fuente escrito en lenguaje ensamblador y genera un fichero objeto. Este fichero contiene instrucciones en

lenguaje máquina e información que ayudará a combinar varios ficheros objeto para crear un fichero ejecutable (puede ser interpretado por el procesador).

Otra herramienta denominada *montador*, combina el conjunto de ficheros objeto para crear un único fichero que puede ser ejecutado por el computador.

En el simulador SPIM, para cargar un programa en memoria se utiliza la opción *load*, que lleva a cabo todos los pasos necesarios, desde la traducción del programa ensamblador a lenguaje máquina hasta la carga del mismo en memoria. En caso de encontrar errores sintácticos de ensamblado aparece un mensaje de error en la ventana de mensajes y, a partir de esa instrucción, no se traduce ni se carga en memoria ninguna otra.

La sintaxis del lenguaje ensamblador se descubrirá poco a poco a lo largo de los diferentes capítulos que componen este manual, pero es interesante introducir ya algunos conceptos básicos. Los primeros que se verán están referidos a los recursos de programación que permite utilizar la programación en ensamblador y que facilitan la programación:

- ***Comentarios***. Estos son muy importantes en los lenguajes de bajo nivel ya que ayudan a seguir el desarrollo del programa y, por tanto, se usan con profusión. Comienzan con un carácter de almohadilla "#" y desde este carácter hasta el final de la línea es ignorado por el ensamblador.

- ***Identificadores.*** Son secuencias de caracteres alfanuméricos, guiones bajos (_) y puntos (.), que no comienzan con un número. Los códigos de operación son palabras reservadas que no pueden ser utilizadas como identificadores.

- ***Etiquetas.*** Son identificadores que se sitúan al principio de una línea y seguidos de dos puntos. Sirven para hacer referencia a la posición o dirección de memoria del elemento definido en ésta. A lo largo del programa se puede hacer referencia a ellas en los modos de direccionamiento de las instrucciones.

- ***Pseudoinstrucciones.*** Son instrucciones que no tienen traducción directa al lenguaje máquina que entiende el procesador, pero el ensamblador las interpreta y las convierte en una o más instrucciones máquina reales. Permiten una programación más clara y comprensible. A lo largo del desarrollo de las prácticas se irán introduciendo diferentes pseudoinstrucciones que permite utilizar este ensamblador.

- ***Directivas***. Tampoco son instrucciones que tienen traducción directa al lenguaje máquina que entiende el procesador, pero el ensamblador las interpreta y le informan a éste de cómo tiene que traducir el programa. Son identificadores reservados, que el ensamblador reconoce y que van precedidos por un punto. A lo largo del desarrollo de las prácticas se irán introduciendo las distintas directivas que permite utilizar este ensamblador.

Por otro lado, los números se escriben, por defecto, en base 10. Si van precedidos de `0x`, se interpretan en hexadecimal. Las cadenas de caracteres se

encierran entre comillas dobles (“). Los caracteres especiales en las cadenas siguen la convención del lenguaje de programación C:

- Salto de línea: \n
- Tabulador: \t
- Comilla: \”

Problemas propuestos

1. Dado el siguiente ejemplo de programa ensamblador:

```
      .data
dato: .byte 3  #inicializo una posición de memoria a 3
      .text
      .globl main    # debe ser global
main: lw $t0,dato($0)
```

Indica las etiquetas, directivas y comentarios que aparecen en el mismo.

CAPÍTULO 2. LOS DATOS EN MEMORIA

El primer paso para el desarrollo de un programa en ensamblador es definir los datos en memoria. La programación en ensamblador permite utilizar directivas que facilitan reservar espacio de memoria para datos e inicializarlos a un valor. En este capítulo se trata el uso de estas directivas.

En primer lugar recordar que, aunque la unidad base de direccionamiento es el *byte*, las memorias de estos computadores tienen un ancho de 4 bytes o 32 bits, que se llamará palabra o *word*, el mismo ancho que el del bus de datos. Así pues, cualquier acceso a una palabra de memoria supondrá leer cuatro bytes (el byte con la dirección especificada y los tres almacenados en las siguientes posiciones). Las direcciones de palabra deben estar alineadas en posiciones múltiplos de cuatro. Otra posible unidad de acceso a memoria es transferir media palabra (*half-word*).

2.1. Declaración de palabras en memoria

En este apartado se verá el uso de las directivas `.data` y `.word`. Para ello, en el directorio de trabajo se crea un fichero con la extensión '.s' y con el siguiente contenido:

```
          .data            # comienza zona de datos
palabra1: .word 15         # decimal
palabra2: .word 0x15       # hexadecimal
```

Descripción:

Este programa en ensamblador incluye diversos elementos que se describen a continuación: la directiva `.data [dir]` indica que los elementos que se definen a continuación se almacenarán en la zona de datos y, al no aparecer ninguna dirección como argumento de dicha directiva, la dirección de almacenamiento será la que hay por defecto (`0x10010000`). Las dos sentencias que aparecen a continuación reservan dos números enteros, de tamaño *word*, en dos direcciones de memoria, una a continuación de la otra, con los contenidos especificados.

Ejecuta el programa XSPIM desde el directorio de trabajo y cárgalo mediante el botón *load*.

Cuestión 2.1: Encuentra los datos almacenados en memoria en el programa anterior. Localiza dichos datos en el panel de datos e indica su valor en hexadecimal.

Cuestión 2.2: ¿En qué direcciones se han almacenado dichos datos? ¿Por qué?

Cuestión 2.3: ¿Qué valores toman las etiquetas `palabra1` y `palabra2`?

Crea otro fichero o modifica el anterior con el siguiente código:

```
          .data 0x10010000   # comienza zona de datos
palabras: .word     15,0x15  # decimal/hexadecimal
```

Borra los valores de la memoria mediante el botón *clear* y carga el fichero.

Cuestión 2.4: Comprueba si hay diferencias respecto al programa anterior.

Cuestión 2.5: Crea un fichero con un código que defina un vector de cinco palabras (word), que esté asociado a la etiqueta `vector`, que comience en la dirección 0x10000000 y con los valores 0x10, 30, 0x34, 0x20 y 60. Comprueba que se almacena de forma correcta en memoria.

Cuestión 2.6: ¿Qué ocurre si se quiere que el vector comience en la dirección 0x10000002? ¿En qué dirección comienza realmente? ¿Por qué?

2.2. Declaración de bytes en memoria

La directiva `.byte valor` inicializa una posición de memoria, de tamaño byte, con el contenido `valor`.

Crea un fichero con el siguiente código:

```
          .data        # comienza zona de datos
octeto:   .byte    0x10 # hexadecimal
```

Borra los valores de la memoria mediante el botón *clear* y carga el fichero.

Cuestión 2.7: ¿Qué dirección de memoria se ha inicializado con el contenido especificado?

Cuestión 2.8: ¿Qué valor se almacena en la palabra que contiene el byte?

Crea otro fichero o modifica el anterior con el siguiente código:

```
          .data
palabra1: .byte    0x10,0x20,0x30,0x40# hexadecimal
palabra2: .word    0x10203040         # hexadecimal
```

Borra los valores de la memoria y carga el fichero.

Cuestión 2.9: ¿Cuáles son los valores almacenados en memoria?

Cuestión 2.10: ¿Qué tipo de alineamiento y organización de los datos (Big-endian o Little-endian) utiliza el simulador? ¿Por qué?

Cuestión 2.11: ¿Qué valores toman las etiquetas `palabra1` y `palabra2`?

2.3. Declaración de cadenas de caracteres

La directiva `.ascii "tira"` permite cargar en posiciones de memoria consecutivas, cada una de tamaño byte, el código ASCII de cada uno de los caracteres que componen `"tira"`.

Crea un fichero con el siguiente código:

```
          .data
cadena:   .ascii    "abcde"   # defino string
octeto:   .byte     0xff
```

Borra los valores de la memoria y carga el fichero.

Cuestión 2.12: Localiza la cadena anterior en memoria.

Cuestión 2.13: ¿Qué ocurre si en vez de `.ascii` se emplea la directiva `.asciiz`? Describe lo que hace esta última directiva.

Cuestión 2.14: Crea otro fichero cargando la misma tira de caracteres a la que apunta la etiqueta `cadena` en memoria, pero ahora utilizando la directiva `.byte`.

2.4. Reserva de espacio en memoria

La directiva `.space n` sirve para reservar espacio para una variable en memoria, inicializándola a valor 0.

Crea un fichero con el siguiente código:

```
          .data
palabra1: .word     0x20
espacio:  .space    8    # reservo espacio
palabra2: .word     0x30
```

Borra los valores de la memoria y carga el fichero.

Cuestión 2.15: ¿Qué rango de posiciones se han reservado en memoria para la variable `espacio`?

Cuestión 2.16: ¿Cuántos bytes se han reservado en total? ¿Y cuántas palabras?

2.5. Alineación de datos en memoria

La directiva `.align n` alinea el siguiente dato a una dirección múltiplo de 2^n.

Crea un fichero con el siguiente código:

```
          .data
byte1:    .byte 0x10
espacio:  .space 4
byte2:    .byte 0x20
palabra:  .word 10
```

Cuestión 2.17: ¿Qué rango de posiciones se han reservado en memoria para la variable `espacio`?

Cuestión 2.18: ¿Estos cuatro bytes podrían constituir los bytes de una palabra? ¿Por qué?

Cuestión 2.19: ¿A partir de que dirección se ha inicializado `byte1`? ¿y `byte2`?

Cuestión 2.20: ¿A partir de que dirección se ha inicializado `palabra`? ¿Por qué?

Crea un fichero con el siguiente código:

```
          .data
byte1:    .byte 0x10
          .align 2
espacio:  .space 4
byte2:    .byte 0x20
palabra:  .word 10
```

Cuestión 2.21: ¿Qué rango de posiciones se ha reservado en memoria para la variable `espacio`?

Cuestión 2.22:¿Estos cuatro bytes podrían constituir los bytes de una palabra? ¿Por qué? ¿Qué ha hecho la directiva `.align`?

Problemas propuestos:

1. Diseña un programa ensamblador que reserva espacio para dos vectores A y B de 20 palabras cada uno a partir de la dirección `0x10000000`.

2. Diseña un programa ensamblador que realiza la siguiente reserva de espacio en memoria a partir de la dirección 0x10001000: una palabra, un byte y otra palabra alineada en una dirección múltiplo de 4.
3. Diseña un programa ensamblador que realice la siguiente reserva de espacio e inicialización en memoria a partir de la dirección por defecto: 3 (palabra), 0x10 (byte), reserve 4 bytes a partir de una dirección múltiplo de 4, y 20 (byte).
4. Diseña un programa ensamblador que defina, en el espacio de datos, la siguiente cadena de caracteres: "Esto es un problema" utilizando
   a) .ascii
   b) .byte
   c) .word
5. Sabiendo que un entero se almacena en un word, diseña un programa ensamblador que defina en la memoria de datos la matriz A de enteros definida como

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

   a partir de la dirección `0x10010000` suponiendo que:
   a) La matriz A se almacena por filas.
   b) La matriz A se almacena por columnas.

CAPÍTULO 3. CARGA Y ALMACENADO DE LOS DATOS

En este tercer capítulo se estudia la carga y almacenamiento de datos entre memoria y los registros del procesador. Dado que la arquitectura del R2000 es RISC, utiliza un subconjunto concreto de instrucciones que permiten las acciones de carga y almacenamiento de los datos entre los registros del procesador y la memoria. Generalmente, las instrucciones de carga de un dato de memoria a registro comienzan con la letra "l" (de *load* en inglés) y las de almacenamiento de registro en memoria con "s" (de *store* en inglés), seguidos por la letra inicial correspondiente al tamaño del dato que se va a mover, *b* para byte, *h* para media palabra (*half word*) y *w* para palabra (*word*).

3.1. Carga de datos inmediatos (constantes)

Crea un fichero con el siguiente código:

```
        .text       #zona de instrucciones
main:   lui $s0, 0x8690
```

Descripción:

La directiva `.text` sirve para indicar el comienzo de la zona de memoria dedicada a las instrucciones. Por defecto esta zona comienza en la dirección `0x00400000` y en ella se pueden ver las instrucciones que ha introducido el simulador para ejecutar, de forma adecuada, nuestro programa. La primera instrucción se referencia con la etiqueta `main` y le indica al simulador dónde está el principio del programa que debe ejecutar. Por defecto hace referencia a la dirección `0x00400020`. A partir de esta dirección, el simulador cargará el código de nuestro programa en el segmento de memoria de instrucciones.

La instrucción `lui` es la única instrucción de carga inmediata real, y almacena la media palabra que indica el dato inmediato de 16 bits en la parte alta del registro especificado, en este caso `$s0`. La parte baja del registro, correspondiente a los bits de menor peso de éste, se pone a 0.

Borra los valores de la memoria del simulador y carga el fichero anterior.

Cuestión 3.1: Localiza la instrucción en memoria de instrucciones e indica:

- La dirección donde se encuentra.
- El tamaño que ocupa.

- La instrucción máquina, analizando cada campo de ésta e indicando qué tipo de formato tiene.

Ejecuta el programa mediante el botón *run* del *xspim*.

Cuestión 3.2: Comprueba el efecto de la ejecución del programa en el registro.

El ensamblador del MIPS ofrece la posibilidad de cargar una constante de 32 bits en un registro utilizando una pseudoinstrucción. Ésta es la pseudoinstrucción `li`.

Crea un fichero con el siguiente código:

```
          .text     #zona de instrucciones
main:     li $s0,0x12345678
```

Borra los valores de la memoria del *xspim* y carga este fichero.

Ejecuta el programa mediante el botón *run* del *xspim*.

Cuestión 3.3: Comprueba el efecto de la ejecución del programa en el registro.

Cuestión 3.4: Comprueba qué conjunto de instrucciones reales implementan esta pseudoinstrucción.

3.2. Carga de palabras (palabras de memoria a registro)

Crea un fichero con el siguiente código:

```
          .data
palabra:  .word 0x10203040
          .text              #zona de instrucciones
main:     lw $s0,palabra($0)
```

Descripción:

La instrucción `lw` carga la palabra contenida en una posición de memoria, cuya dirección se especifica en la instrucción, en un registro. Dicha posición de memoria se obtiene sumando el contenido del registro (en este caso `$0`, que siempre vale cero) y el identificador `palabra`.

Borra los valores de la memoria del SPIM y carga el fichero anterior.

Cuestión 3.5: Localiza la instrucción en memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Cuestión 3.6: Explica cómo se obtiene a partir de esas instrucciones la dirección de `palabra`. ¿Por qué crees que el simulador traduce de esta forma la instrucción original?

Cuestión 3.7: Analiza cada uno de los campos que componen estas instrucciones e indica el tipo de formato que tienen.

Ejecuta el programa mediante el botón *run*.

Cuestión 3.8: Comprueba el efecto de la ejecución del programa.

Cuestión 3.9: ¿Qué pseudoinstrucción permite cargar la dirección de un dato en un registro? Modifica el programa original para que utilice esta pseudoinstrucción, de forma que el programa haga la misma tarea. Comprueba qué conjunto de instrucciones sustituyen a la pseudoinstrucción utilizada una vez el programa se carga en la memoria del simulador.

Cuestión 3.10: Modifica el código para que en lugar de transferir la palabra contenida en la dirección de memoria referenciada por la etiqueta palabra, se transfiera la palabra que está contenida en la dirección referenciada por `palabra+1`. Explica qué ocurre y por qué.

Cuestión 3.11: Modifica el programa anterior para que guarde en el registro `$s0` los dos bytes de mayor peso de `palabra`. Nota: Utiliza la instrucción `lh` que permite cargar medias palabras (16 bits) desde memoria a un registro (en los 16 bits de menor peso del mismo).

3.3. Carga de bytes (bytes de memoria a registro)

Crea un fichero con el siguiente código:

```
            .data
octeto:     .byte 0xf3
siguiente:  .byte 0x20

            .text    #zona de instrucciones
main:       lb $s0, octeto($0)
```

Descripción:

La instrucción `lb` carga el byte de una dirección de memoria en un registro. Al igual que antes, la dirección del dato se obtiene sumando el contenido del registro `$0` (siempre vale cero) y el identificador `octeto`.

Borra los valores de la memoria y carga el fichero.

Cuestión 3.12: Localiza la instrucción en memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Ejecuta el programa.

Cuestión 3.13: Comprueba el efecto de la ejecución del programa.

Cuestión 3.14: Cambia en el programa la instrucción `lb` por `lbu`. ¿Qué sucede al ejecutar el programa? ¿Qué significa esto?

Cuestión 3.15: Si `octeto` se define como:

"`octeto:  .byte    0x30`", ¿existe diferencia entre el uso de la instrucción `lb` y `lbu`?¿ Por qué?

Cuestión 3.16: ¿Cuál es el valor del registro `$s0` si `octeto` se define como:

"`octeto:  .word    0x10203040`"? ¿Por qué?

Cuestión 3.17: ¿Cuál es el valor del registro `$s0` si se cambia en `main` la instrucción existente por la siguiente:

```
main:     lb $s0,octeto+1($0)?
```

¿Por qué? ¿Por qué en este caso no se produce un error de ejecución (excepción de error de direccionamiento)?

3.4. Almacenado de palabras (palabras de registro a memoria)

Crea un fichero con el siguiente código:

```
               .data
palabra1: .word 0x10203040
palabra2: .space 4
palabra3: .word 0xffffffff

               .text     #zona de instrucciones
main:     lw $s0, palabra1($0)
          sw $s0, palabra2($0)
          sw $s0, palabra3($0)
```

Descripción:

La instrucción `sw` almacena la palabra contenida en un registro en una dirección de memoria. Esta dirección se obtiene sumando el contenido de un registro más un desplazamiento especificado en la instrucción (identificador).

Borra los valores de la memoria del SPIM y carga el fichero.

Cuestión 3.18: Localiza la primera instrucción de este tipo en la memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Ejecuta el programa.

Cuestión 3.19: Comprueba el efecto de la ejecución del programa.

3.5. Almacenado de bytes (bytes de registro a memoria)

Crea un fichero con el siguiente código:

```
               .data
palabra:  .word 0x10203040
octeto:        .space 2

               .text    #zona de instrucciones
main:          lw $s0, palabra($0)
               sb $s0, octeto($0)
```

Descripción:

La instrucción "`sb`" almacena el byte de menor peso de un registro en una dirección de memoria. La dirección se obtiene sumando el desplazamiento indicado por el identificador y el contenido de un registro.

Borra los valores de la memoria y carga el fichero.

Cuestión 3.20: Localiza la instrucción en memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Ejecuta el programa.

Cuestión 3.21: Comprueba el efecto de la ejecución del programa.

Cuestión 3.22: Modifica el programa para que el byte se almacene en la dirección "`octeto+1`". Comprueba y describe el resultado de este cambio.

Cuestión 3.23: Modifica el programa anterior para transferir a la dirección de `octeto` el contenido de la posición `palabra+3`.

Problemas propuestos

1. Diseña un programa ensamblador que defina el vector de enteros V=(10, 20, 25, 500, 3) en la memoria de datos a partir de la dirección 0x10000000 y cargue todos sus componentes en los registros $s0 - $s4.

2. Diseña un programa ensamblador que copie el vector definido en el problema anterior a partir de la dirección 0x10010000.

3. Diseña un programa ensamblador que, dada la palabra 0x10203040 almacenada en una posición de memoria, la reorganice en otra posición, invirtiendo el orden de sus bytes.

4. Diseña un programa ensamblador que, dada la palabra 0x10203040 definida en memoria la reorganice en la misma posición, intercambiando el orden se sus medias palabras. Nota: utiliza la instrucción lh y sh.

5. Diseña un programa en ensamblador que inicialice cuatro bytes a partir de la posición 0x10010002 a los siguientes valores 0x10, 0x20, 0x30, 0x40, y reserve espacio para una palabra a partir de la dirección 0x1001010. El programa transferirá los cuatro bytes contenidos a partir de la posición 0x10010002 a la dirección 0x1001010.

CAPÍTULO 4. LAS OPERACIONES ARITMÉTICAS Y LÓGICAS

En este capítulo se presentan las instrucciones que permiten realizar operaciones aritméticas, lógicas y de desplazamiento de datos. Las instrucciones aritméticas están constituidas por las operaciones de suma y resta (add, addu, addi, addiu, sub, subu) y las operaciones de multiplicación y división (mult, multu, div, divu). La diferencia entre las instrucciones acabadas o no acabadas en u (por ejemplo entre add y addu) está en que las primeras provocan una excepción de desbordamiento si el resultado de la operación no es representable en 32 bits y las segundas no tienen en cuenta el desbordamiento. Recordar que el rango de representación de los números enteros con signo de 32 bits en complemento a 2 va desde –2.147.483.648 a 2.147.483.647 (0x80000000 a 0x7fffffff).

Dentro del grupo de instrucciones que permiten realizar operaciones lógicas están: suma lógica (or y ori), producto lógico (and y andi) y la or exclusiva (xor y xori).

Finalmente, se presentan las instrucciones de desplazamiento aritmético y lógico (sra, sll, srl).

4.1. Operaciones aritméticas con datos inmediatos (constantes)

Crea un fichero con el siguiente código:

```
        .data           #zona de datos
      #Máx. Positivo representable 0x7FFFFFFF
numero: .word 2147483647
        .text       #zona de instrucciones
main:   lw  $t0,numero($0)
        addiu  $t1,$t0,1
```

Descripción:

La instrucción “addiu” es una instrucción de suma con un dato inmediato y sin detección de desbordamiento.

Borra los valores de la memoria, carga el fichero y ejecútalo paso a paso.

Cuestión 4.1: Localiza el resultado de la suma efectuada. Comprueba el resultado.

Cuestión 4.2: Cambia la instrucción "addiu" por la instrucción "addi".

Borra los valores de la memoria, carga el fichero y ejecútalo paso a paso.

Cuestión 4.3: ¿Qué ha ocurrido al efectuar el cambio? ¿Por qué?

4.2. Operaciones aritméticas con datos en memoria

Crea un fichero con el siguiente código:

```
               .data
numero1:  .word 0x80000000 #max. Negativo represent.
numero2:  .word 1
numero3:  .word 1
          .text
main:
          lw   $t0,numero1($0)
          lw   $t1,numero2($0)
          subu $t0,$t0,$t1
          lw   t1,numero3($0)
          subu $t0,$t0,$t1
          sw   $t0,numero1($0)
```

Borra los valores de la memoria, carga el fichero y ejecútalo paso a paso.

Cuestión 4.4: ¿Qué hace el programa anterior? ¿Qué resultado se almacena en `numero3`? ¿Es correcto?

Cuestión 4.5: ¿Se producirá algún cambio si las instrucciones "subu" son sustituidas por instrucciones "sub"? ¿Por qué?

4.3. Multiplicación y división con datos en memoria

Crea un fichero con el siguiente código:

```
          .data
numero1:  .word 0x7FFFFFFF #Máx. Positivo represent.
numero2:  .word    16
          .space   8
```

```
            .text
main:       lw $t0,numero1($0)
        lw $t1,numero2($0)
        mult $t0,$t1 # multiplica los dos números
        mfhi $t0
        mflo $t1
        sw $t0,numero2+4($0) #32 bits más peso
        sw $t1,numero2+8($0) #32 bits menos peso
```

Descripción:

El código realiza la multiplicación de dos números, almacenando el resultado de la multiplicación a continuación de los dos multiplicandos.

Borra los valores de la memoria, carga el fichero y ejecútalo.

Cuestion 46: ¿Qué resultado se obtiene después de realizar la operación? ¿Por qué se almacena en dos palabras de memoria?

Cuestión 4.7: Modifica los datos anteriores para que `numero1` y `numero2` sean 10 y 3 respectivamente. Escribe el código que divida `numero1` entre `numero2` (dividendo y divisor respectivamente) y coloque el cociente y el resto a continuación de dichos números.

4.4. Operaciones lógicas

Crea un fichero con el siguiente código:

```
            .data
numero:     .word     0x3ff41
            .space    4
            .text
main:       lw $t0,numero($0)
            andi $t1,$t0,0xfffe
            # 0xffe en binario es 0...01111111111111110
            sw $t1,numero+4($0)
```

Descripción:

El código anterior pone los 16 bits más significativos y el bit 0 de `numero` a 0, y almacena el resultado en la siguiente palabra (`numero+4`). Esto se consigue utilizando la instrucción “andi”, que realiza el producto lógico, bit a bit, entre el dato contenido en un registro y un dato inmediato. El resultado obtenido es que aquellos

bits que en el dato inmediato están a 1 no se modifican con respecto al contenido original del registro, es decir, los 16 bits de menor peso excepto el bit 0. Por otro lado, todos aquellos bits que en el dato inmediato están a 0, en el resultado también están a 0, es decir, los 16 bits de mayor peso y el bit 0. Los 16 bits de mayor peso del resultado se ponen a 0 puesto que, aunque el dato inmediato que se almacena en la instrucción es de 16 bits, a la hora de realizar la operación el procesador trabaja con un dato de 32 bits, poniendo los 16 de mayor peso a 0, fijando por tanto los bits del resultado también a 0.

Borra los valores de la memoria, carga el fichero y ejecútalo. Comprueba el resultado.

Cuestión 4.8: Modifica el código para obtener, en la siguiente palabra, que los 16 bits más significativos de `numero` permanezcan tal cual, y que los 16 bits menos significativos queden a cero, excepto el bit cero que también debe quedar como estaba.

Cuestión 4.9: Modifica el código para obtener, en la siguiente palabra, que los 16 bits más significativos de `numero` permanezcan tal cual, y que los 16 bits menos significativos queden a uno, excepto el bit cero que debe quedar como estaba.

4.5. Operaciones de desplazamiento

Crea un fichero con el siguiente código:

```
         .data
numero:  .word    0xffffff41
         .text
main:    lw $t0,numero($0)
         sra $t1,$t0,4
```

Descripción:

El código anterior desplaza el valor de `numero` cuatro bits a la derecha, rellenando el hueco generado a su izquierda con el bit del signo.

Borra los valores de la memoria, carga el fichero y ejecútalo.

Cuestión 4.10: ¿Para qué se ha rellenado `numero` con el bit del signo?

Cuestión 4.11: ¿Qué ocurre si se sustituye la instrucción “sra” por “srl”?

Cuestión 4.12: Modifica el código para desplazar el contenido de `numero` 2 bits a la izquierda.

Cuestión 4.13: ¿Qué operación aritmética acabamos de realizar?

Cuestión 4.14: Volviendo al código que iniciaba este apartado, indica qué operación aritmética tenia como consecuencia la ejecución de dicho código.

Problemas propuestos

1. Diseña un programa ensamblador que defina el vector de enteros de dos elementos V=(10,20) en la memoria de datos a partir de la dirección 0x10000000 y almacene su suma a partir de la dirección donde acaba el vector.

2. Diseña un programa ensamblador que divida los enteros 18,-1215 almacenados a partir de la dirección 0x10000000 entre el número 5 y que a partir de la dirección 0x10010000 almacene el cociente de dichas divisiones.

3. Pon a cero los bits 3,7,9 del entero 0xabcd12bd almacenado en memoria a partir de la dirección 0x10000000, sin modificar el resto.

4. Cambia el valor de los bits 3,7,9 del entero 0xff0f1235 almacenado en memoria a partir de la dirección 0x10000000, sin modificar el resto.

5. Multiplica el número 0x1237, almacenado en memoria a partir de la dirección 0x10000000, por 32 (2^5) sin utilizar las instrucciones de multiplicación ni las pseudoinstrucciones de multiplicación.

CAPÍTULO 5. EVALUACIÓN DE LAS CONDICIONES. VARIABLES Y OPERADORES BOOLEANOS

El lenguaje ensamblador no dispone de estructuras de control de flujo de programa definidas, que permitan decidir entre dos (o varios) caminos de ejecución de instrucciones distintos (por ejemplo la sentencia *if* de otros lenguajes de programación como PASCAL, C, etc.). Normalmente para implementar cualquier estructura de este tipo es necesario evaluar previamente una condición, simple o compuesta. El camino que seguirá la ejecución del programa dependerá del resultado de esta evaluación.

En éste capítulo se describe como se evalúan las condiciones en ensamblador del MIPS R2000. En primer lugar se realiza un breve repaso al conjunto de instrucciones y pseudoinstrucciones que tiene el MIPS R2000 para realizar comparaciones y control del flujo de programa. A continuación, se proponen varios ejemplos que implementan fragmentos de código para evaluar diversas condiciones simples y compuestas.

Instrucciones de salto condicional del MIPS R2000

El ensamblador del MIPS incluye dos instrucciones básicas de toma de decisiones `beq` y `bne`, que permiten implementar estructuras de control muy sencillas. La sintaxis de estas instrucciones es la siguiente:

```
beq rs,rt,etiqueta

bne rs,rt,etiqueta
```

Ambas comparan el contenido de los registros `rs` y `rt` y según el resultado de esta comparación (cierta o falsa), saltan a la dirección de la instrucción que referencia `etiqueta` o no. El resultado de la evaluación es cierto si el contenido del registro `rs` es igual al del `rt` (instrucción `beq`), o distinto (instrucción `bne`), falsa en caso contrario para cada una de las instrucciones.

También dispone de instrucciones de salto condicional para realizar comparaciones con cero. Estas instrucciones son: `bgez, bgtz, blez, bltz`, y tienen la siguiente sintaxis:

```
bgez rs,etiqueta

bgtz rs,etiqueta

blez rs,etiqueta
```

```
bltz rs,etiqueta
```

Todas ellas comparan el contenido del registro rs con 0 y saltan a la dirección de la instrucción referenciada por `etiqueta` si rs>=0 (`bgez`), rs>0 (`bgtz`), rs<=0 (`blez`) y rs<0 (`bltz`).

Pseudoinstrucciones de salto condicional

El lenguaje ensamblador del MIPS R2000 dispone de un conjunto de pseudoinstrucciones de salto condicional que permiten comparar dos variables almacenadas en registros (a nivel de mayor, mayor o igual, menor, menor o igual) y, según el resultado de esa comparación, saltan o no, a la instrucción que referencia etiqueta. Estas pseudoinstrucciones son: `bge` (branch if greater or equal, saltar si mayor o igual), `bgt` (branch if greater, saltar si mayor que), `ble` (branch if less or equal, saltar si menor o igual), `blt` (branch if less, saltar si menor que). El formato es el mismo para todas ellas y coincide con el de las instrucciones de salto descritas, `beq` y `bne`:

```
bxx rs,rt,etiqueta
```

donde xx es `ge` (saltar si rs es mayor o igual que rt), `gt` (saltar si rs es mayor que rt), `le` (saltar si rs es menor o igual que), `lt` (saltar si rs es menor que).

Instrucciones de comparación

El lenguaje máquina y ensamblador del MIPS dispone de una instrucción que compara dos registros y carga un 1 ó un 0 en un tercer registro dependiendo del resultado de la comparación. Si el contenido del primer registro es menor que el segundo carga un 1, en caso contrario carga un 0. Ésta es la instrucción `slt` y tiene la siguiente sintaxis:

```
slt rd,rs,rt
```

La nomenclatura que se va a seguir a lo largo de este capítulo para describir el resultado que proporciona la ejecución de una instrucción de este tipo (evaluación de una condición) es el siguiente:

```
rd(1) ← (rs < rt),
```

indicando que el registro rd se pondrá a 1 si el contenido del registro rs es menor que rt y se pondrá a 0 en caso contrario.

Pseudoinstrucciones de comparación

El ensamblador del MIPS permite utilizar un conjunto de pseudoinstrucciones que facilitan la evaluación de condiciones. Estas pseudoinstrucciones realizan comparaciones de dos variables a otros niveles (como mayor, mayor o igual, menor o igual, igual, distinto) contenidas en sendos registros y almacenan el resultado de la comparación en un tercer registro (poniendo un 1 si la condición es cierta, y un 0 en caso contrario). Estas pseudoinstrucciones tienen la misma sintaxis que la instrucción

`slt` descrita. Estas son: `sge` (poner 1 si mayor o igual), `sgt` (poner 1 si mayor), `sle` (poner 1 si menor o igual), `sne` (poner a 1 si distinto), `seq` (poner a 1 si igual).

A continuación se estudia como se evalúan condiciones simples y compuestas por los operadores "and" y "or".

5.1. Evaluación de condiciones simples: menor, mayor, menor o igual, mayor o igual

Crea un fichero con el siguiente código que compara las variables `dato1` y `dato2` y deja el resultado en la variable booleana `res`:

```
            .data
dato1:      .word    30
dato2:      .word    40
res:        .space   1
            .text
main:       lw $t0,dato1($0) # cargar dato1 en t0
            lw $t1,dato2($0) # cargar dato2 en t1
            slt $t2,$t0, $t1 # poner a 1 $t2 si t0<t1
            sb $t2,res($0)   # almacenar $t2 en res
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo. El diagrama de flujo asociado a este fragmento de código, que puede ayudar a analizarlo es el siguiente:

$t0=dato1 $t1=dato2 $t2(1) ← ($t0<$t1)

Cuestión 5.1: ¿Qué valor se carga en la posición de memoria res?

Inicializa las posiciones de memoria `dato1` y `dato2` con los valores 50 y 20 respectivamente. Ejecuta de nuevo el programa

Cuestión 5.2: ¿Qué valor se carga ahora en la posición de memoria `res`?

Cuestión 5.3: ¿Qué comparación se ha evaluado entre `dato1` y `dato2`?

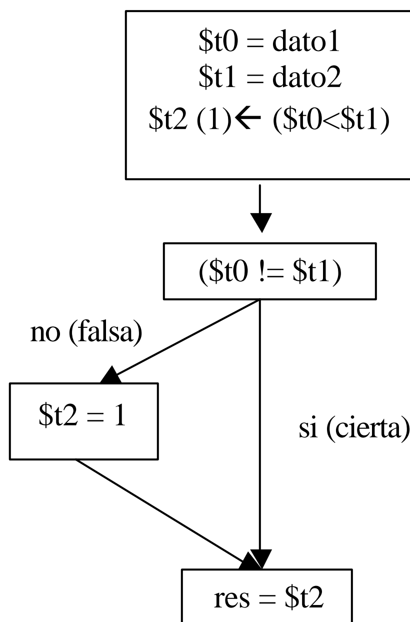
Cuestión 5.4: Modifica el código anterior para evaluar la siguiente condición `res`(1)←(`dato1 = dato2`).

Cuestión 5.5: Resuelve la cuestión anterior utilizando la pseudoinstrucción `sge`.

Crea un fichero con el siguiente código que compara las variables `dato1` y `dato2` y deja el resultado en la variable booleana `res`:

```
        .data
dato1:    .word    30
dato2:    .word    40
res:      .space   1
        .text
main: lw $t0,dato1($0)   # cargar dato1 en t0
      lw $t1, dato2($0)  # cargar dato2 en t1
      slt $t2, $t0, $t1  # poner a 1 $t2 si t0<t1
      bne $t0,$t1,fineval # si t0<>t1 salta a fineval
      ori $t2,$0,1       # poner a 1 t3 si t0=t1
fineval: sb $t2,res($0)  # almacenar $t2 en res
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo. El diagrama de flujo asociado al fragmento de código anterior, que puede ayudar a analizarlo, es el que se muestra en la siguiente figura:



Cuestión 5.6: ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.7: Inicializa las posiciones de memoria `dato1` y `dato2` con los valores 50 y 20, respectivamente. Ejecuta de nuevo el programa, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.8: Inicializa las posiciones de memoria `dato1` y `dato2` con los valores 20 y 20, respectivamente. Ejecuta de nuevo el programa, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.9: ¿Qué comparación se ha evaluado entre `dato1` y `dato2`?

Cuestión 5.10: Evalúa esta comparación utilizando pseudoinstrucciones.

Cuestión 5.11: Modifica el código anterior para que la condición evaluada sea `res(1) ← (dato1 >= dato2)`. No utilices pseudoinstrucciones

Cuestión 5.12: Modifica el código anterior utilizando pseudoinstrucciones.

5.2. Evaluación de condiciones compuestas por operadores lógicos "and" y "or".

Crea un fichero con el siguiente código que compara las variables `dato1` y `dato2` y deja el resultado de la comparación en la variable `res`:

```
          .data
dato1:    .word     40
dato2:    .word     -50
res       .space    1
          .text
main:     lw $t8,dato1($0)
          lw $t9,dato2($0)
          and $t0,$t0,$0
          and $t1,$t1,$0
          beq $t8,$0,igual
          ori $t0,$0,1
igual:    beq $t9,$0,fineval
          ori $t1,$0,1
fineval:  and $t0,$t0,$t1
          sb $t0,res($0)
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo.

Cuestión 5.13: ¿Qué valor se carga en la posición de memoria `res`? Para ayudarte en el análisis del código, dibuja el diagrama de flujo de éste.

Cuestión 5.14: Inicializa `dato1` y `dato2` con los valores 0 y 20, respectivamente. ¿Qué valor se carga en la posición de memoria res?

Cuestión 5.15: Inicializa `dato1` y `dato2` con los valores 20 y 0, respectivamente. ¿Qué valor se carga en la posición de memoria res?

Cuestión 5.16: Inicializa `dato1` y `dato2` con los valores 0 y 0, respectivamente. ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.17: ¿Qué comparación compuesta se ha evaluado entre `dato1` y `dato2`?

Cuestión 5.18: Modifica el código anterior para que la condición evaluada sea: `res` (1) ← ((`dato1` <> 0) and (`dato1` <> `dato2`)).

Crea el siguiente fichero con la extensión .s:

```
         .data
dato1    .word     30
dato2    .word     -50
res      .space    1
         .text
main:    lw $t8,dato1($0)
         lw $t9,dato2($0)
         and $t1,$t1,$0
         and $t0,$t0,$0
         beq $t8,$0,igual
         ori $t0,$0,1
igual:   slt $t1,$t9,$t8
fineval: and $t0,$t0,$t1
         sb $t0,res($0)
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecutalo.

Cuestión 5.19: ¿Qué valor se carga en la posición de memoria res? Si te sirve de ayuda dibuja el diagrama de flujo asociado al código que se quiere analizar.

Cuestión 5.20: Inicializa `dato1` y `dato2` con los valores 10 y 20, ejecútalo ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.21: Inicializa `dato1` y `dato2` con los valores 0 y -20, ejecútalo ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.22: ¿Qué comparación compuesta se ha evaluado?

Cuestión 5.23: Modifica el código anterior para que la condición evaluada sea `res` (1) ← ((`dato1` <> `dato2`) and (`dato1` <= `dato2`)).

Cuestión 5.24: Modifica el código anterior utilizando la pseudoinstrucción `sle`.

Crea el siguiente fichero con la extensión .s que compara las variables `dato1` y `dato2` y deja el resultado de la comparación en la variable `res`:

```
          .data
dato1:    .word     30
dato2:    .word     -20
res       .space    1
          .text
main:     lw $t8,dato1($0)
          lw $t9,dato2($0)
          and $t0,$t0,$0
          and $t1,$t1,$0
          slt $t0,$t8,$t9
          bne $t9,$0,fineval
          ori $t1,$0,1
fineval:  or $t0,$t0,$t1
          sb $t0,res($0)
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo.

Cuestión 5.25: ¿Qué valor se carga en la posición de memoria `res`? Si te sirve de ayuda dibuja el diagrama de flujo asociado al código que se quiere analizar.

Cuestión 5.26: Inicializa `dato1` y `dato2` con los valores -20 y 10, respectivamente, ejecuta de nuevo el código, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.27: Inicializa `dato1` y `dato2` con los valores 10 y 0, respectivamente, ejecuta de nuevo el código, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.28: Inicializa `dato1` y `dato2` conlos valores 20 y 10, respectivamente, ejecuta de nuevo el código, ¿Qué valor se carga en la posición de memoria `res`?

Cuestión 5.29: ¿Qué comparación compuesta se ha realizado?

Cuestión 5.30: Modifica el código anterior para que la condición evaluada sea `res`(1)← ((`dato1` <= `dato2`) or (`dato1` <= 0)).

Cuestión 5.31: Modifica el código anterior utilizando la pseudoinstrucción `sle`.

Problemas propuestos

1. Diseña un programa en ensamblador que defina un vector de booleanos, `V`. Este se implementa a partir de un vector de bytes donde cada byte sólo puede tomar dos valores, 0 ó 1, para el valor cierto o falso, respectivamente. V se inicializará con los siguientes valores V=[0,1,1,1,0]. El programa obtendrá otro vector de booleanos, `res`, de tres elementos tal que

   `res`[1]= (V[1] and V[5]),

   `res`[2]=(V[2] or V[4]) ,

   `res`[3]=((V[1] or V[2]) and V[3]).

2. Diseña un programa en ensamblador que defina un vector de enteros, `V`, inicializado según los siguientes valores (V=[2, -4, -6]). Y obtenga un vector de booleanos, tal que cada elemento será 1 si el correspondiente elemento en el vector de enteros es mayor o igual que cero y 0 en caso contrario.

3. Diseña un programa en ensamblador que defina un vector de enteros, `V`, inicializado a los siguientes valores `V`=[1, -4, -5, 2] y obtenga como resultado una variable booleana que será 1 si todos los elementos de este vector son menores que cero.

CAPÍTULO 6. ESTRUCTURAS DE CONTROL CONDICIONAL

Una vez introducidas, en el capítulo anterior, el conjunto de instrucciones y pseudoinstrucciones que permiten implementar cualquier estructura de control de flujo de programa, se va a describir cómo se implementan las estructuras más típicas de un lenguaje de alto nivel, estructuras condicionales como *Si-entonces, Si-entonces-sino,* o estructuras repetitivas como *Mientras* y *Para*, si se habla en lenguaje algorítmico, o las sentencias *if-then, if-then-else, while*, y *for* del lenguaje Pascal. Estas estructuras dependen, implícita o explícitamente, de la verificación de una o varias condiciones para determinar el camino que seguirá la ejecución del código en curso. La evaluación de esta condición (o condiciones) vendrá asociada a una o varias instrucciones de salto condicional e incondicional o pseudoinstrucciones.

6.1. Estructura de control *Si-entonces* con condición simple

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control condicional *Si-entonces*:

```
          .data
dato1:    .word     40
dato2:    .word     30
res:      .space    4
          .text
main:     lw  $t0,dato1($0)  #cargar dato1 en $t0
          lw  $t1,dato2($0)  #cargar dato2 en $t1
          and $t2,$t2,$0     #t2=0
Si:       beq $t1,$0,finsi   #si $t1 = 0 finsi
entonces: div $t0,$t1        #t0/$t1
          mflo $t2           #almacenar LO en $t2
finsi:    add $t3,$t0,$t1    #$t3=$t0+$t1
          add $t2,$t3,$t2    #$t2=$t3+$t2
          sw  $t2,res($0)    #almacenar en res $t2
```

La descripción algorítmica de este programa en ensamblador se muestra a continuación. Esta descripción es muy cercana al código anterior, utilizando registros para almacenar temporalmente el contenido de las variables en memoria:

```
PROGRAMA SI-ENTONCES-SIMPLE-1
VARIABLES
     ENTEROS: dato1=40; dato2=30; res;
INICIO
     $t0=dato1;
     $t1=dato2;
     $t2=0;
     Si ($t1!=0) entonces
          $t2=$t0/$t1;
     FinSi
     $t3=$t0+$t1;
     $t2=$t2+$t3;
     res=$t2;
FIN
```

Esta transcripción casi directa del programa en ensamblador, donde se tienen que utilizar registros del procesador para almacenar temporalmente las variables en memoria es debida a que se trata de un procesador basado en una arquitectura de carga y almacenamiento.

Una descripción algorítmica de más alto nivel pasaría por no utilizar los registros y trabajar directamente sobre las variables almacenadas en memoria:

```
PROGRAMA SI-ENTONCES-SIMPLE-2
VARIABLES
     ENTERO: dato1=40; dato2=30; res;
INICIO
     Si (dato2!=0) entonces
          res=dato1/dato2;
     FinSi
     res=res+dato1+dato2;
FIN
```

Borra los valores de la memoria y carga el fichero que contiene el programa en ensamblador en el simulador.

Cuestión 6.1: Identifica la instrucción que evalúa la condición y controla el flujo de programa. Compárala con la condición del programa descrito en lenguaje algorítmico.

Cuestión 6.2: Identifica el conjunto de instrucciones que implementan la estructura condicional *Si-entonces*.

Cuestión 6.3: ¿Qué valor se almacena en la variable `res` después de ejecutar el programa?

Cuestión 6.4: Si `dato2` es igual 0 ¿Qué valor se almacena en la variable `res` después de ejecutar el programa? Dibuja el diagrama de flujo asociado a la estructura de control implementada en el fragmento de código anterior.

Cuestión 6.5: Implementar el siguiente programa descrito en lenguaje algorítmico:

```
PROGRAMA SI-ENTONCES-SIMPLE-3

VARIABLES

     ENTERO: dato1=40; dato2=30; res;

INICIO

     Si (dato2>0) entonces

          res=dato1/dato2;

     FinSi

     res=res+dato1+dato2;

FIN
```

6.2. Estructura de control *Si-entonces* con condición compuesta

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control condicional *Si-entonces*:

```
          .data

dato1:    .word     40

dato2:    .word     30

res:      .space    4
```

```
          .text
main:     lw  $t0,dato1($0)  #cargar dato1 en t0
          lw  $t1,dato2($0)  #cargar dato2 en $t1
          and $t2,$t2,$0     #pone a 0 $t2
Si:       beq $t1,$0,finsi   #si $t1=0 saltar finsi
          beq $t0,$0,finsi #si $t0 =0 saltar finsi
entonces: div $t0,$t1        #t0/$t1
          mflo $t2           #almacenar LO en t2
finsi:    add $t3,$t0,$t1    #t3=t0+$t1
          add $t2,$t3,$t2    #$t2=t3+$t2
          sw  $t2,res($0)    #almacenar en res $t2
```

La descripción algorítmica, utilizando registros para almacenar temporalmente las variables que están en memoria de este código en ensamblador se muestra a continuación:

```
PROGRAMA SI-ENTONCES-COMPUESTA-1
VARIABLES
     ENTERO: dato1=40; dato2=30; res;
INICIO
     $t0=dato1;
     $t1=dato2;
     Si ((t0!=0) and (t1!=0)) entonces
          $t2=t0/t1;
     FinSi
     $t2=$t2+$t0+$t1;
     res=$t2;
FIN
```

Borra los valores de la memoria y carga el fichero en el simulador.

Cuestión 6.6: Describe en lenguaje algorítmico, sin utilizar registros para almacenar temporalmente las variables que están en memoria, el programa ensamblador anterior.

Cuestión 6.7: Identifica la (las) instrucción(es) que evalúa(n) la condición y controla(n) el flujo de programa y compárala(s) con la condición del programa descrito en lenguaje algorítmico.

Cuestión 6.8: Identifica el conjunto de instrucciones que implementan la estructura condicional *Si-entonces*. Dibuja el diagrama de flujo asociado con esta estructura de control.

Cuestión 6.9: Al ajecutar el programa ¿Que se almacena en la variable `res`?

Cuestión 6.10: Si `dato1=0`, ¿Que valor se almacena en la variable `res` después de ejecutar el programa? Si `dato2=0` ¿Que valor se almacena en la variable `res`?

Cuestión 6.11: Implementa el siguiente programa descrito en lenguaje algorítmico:

```
PROGRAMA SI-ENTONCES-COMPUESTA-2

VARIABLES

     ENTERO dato1=40; dato2=30; res;

INICIO

     Si ((dato1>0) and (dato2>=0)) entonces

          res=dato1/dato2;

     FinSi

     res=res+dato1+dato2;

FIN
```

6.3. Estructura de control *Si-entonces-sino* con condición simple.

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control *Si-entonces-sino*.

```
          .data

dato1:    .word    30

dato2:    .word    40
```

```
res:      .space   4
          .text
main:     lw  $t0,dato1($0)  #cargar dato1 en $t0
          lw  $t1,dato2($0)  #cargar dato2 en $t1
Si:       bge $t0,$t1, sino  #si $t0>=$t1 ir a sino
entonces: sw  $t0,res($0)    #almacenar $t0 en res
          j finsi            #ir a finsi
sino:     sw $t1,res($0)#almacenar $t1 en res
finsi:
```

Borra los valores de la memoria, carga el fichero en el simulador y ejecútalo.

Cuestión 6.12: Describe en lenguaje algorítmico el equivalente a este programa en ensamblador.

Cuestión 6.13: ¿Qué valor se almacena en `res` después de ejecutar el programa? Si `dato1=35`, ¿qué valor se almacena en `res` después de ejecutar el programa?

Cuestión 6.14: Identifica en el lenguaje máquina generado por el simulador el conjunto de instrucciones que implementan la pseudoinstrucción `bge`.

Cuestión 6.15: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico:

```
PROGRAMA SI-ENTONCES-SINO-SIMPLE-1
VARIABLES
    ENTERO: dato1=30; dato2=40; res;
INICIO
    Si ((dato1>=dato2)) entonces
        res=dato1-dato2;
    Sino
        res=dato2-dato1;
    FinSi
FIN
```

6.4. Estructura de control *Si-entonces-sino* con condición compuesta

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control *Si-entonces-sino*.

```
            .data
dato1:      .word     30
dato2:      .word      40
dato3:      .word      -1
res:        .space    4
            .text

main:       lw $t1,dato1($0)   #cargar dato1 en $t1
            lw $t2,dato2($0)   #cargar dato2 en $t2
            lw $t3,dato3($0)   #cargar dato3 en $t3

Si:         blt $t3,$t1, entonces #si $t3<$t1 ir entonces
            ble $t3,$t2, sino      #si $t3<=$t2 ir a sino
entonces:   addi $t4,$0,1          #$t4=1
            j finsi                #ir a finsi
sino:       and  $t4,$0,$0         #$t4=0
finsi:      sw   $t4,res($0)       #almacenar res
```

Borra los valores de la memoria, carga el fichero en el simulador.

Cuestión 6.16: Describe en lenguaje algorítmico el equivalente a este programa en ensamblador.

Cuestión 6.17: ¿Qué valor se almacena en `res` después de ejecutar el programa? Si `dato1=40` y `dato2=30`, ¿qué valor se almacena en `res` después de ejecutar el programa?

Cuestión 6.18: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico:

```
PROGRAMA SI-ENTONCES-SINO-COMPUESTA-1
VARIABLES
     ENTERO: dato1=30; dato2=40; dato3=-1; res;
INICIO
     Si ((dato3>=dato1) AND (dato3<=dato2)) entonces
          res=1;
     Sino
          res=0;
     FinSi
FIN
```

6.5. Estructura de control repetitiva *Mientras*

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control repetitiva *Mientras*:

```
          .data
cadena:   .asciiz   "hola"
          .align    2
n:        .space    4
          .text
main:     la $t0,cadena #carga dir. cadena en $t0
          andi $t2,$t2, 0    #$t2=0
mientras: lb   $t1,0($t0)    #almacenar byte en $t1
          beq  $t1,$0,finmientras #si $t1=0 saltar a
                                  #finmientras
          addi $t2,$t2, 1    #$t2=$t2+1
          addi $t0,$t0, 1    #$t0=$t0+1
          j    mientras      #saltar a mientras
finmientras:   sw $t2,n($0)  #almacenar $t2 en n
```

La descripción algorítmica de este programa en ensamblador se muestra a continuación:

```
PROGRAMA MIENTRAS-1
VARIABLES
     VECTOR DE CARACTERES: cadena(4)='hola';
     ENTERO: n;
INICIO
     i=0;
     n=0;
     Mientras (cadena[i]!=0) hacer
          n=n+1;
          i=i+1;
     FinMientras
FIN
```

Borra los valores de la memoria y carga el fichero que contiene el programa en ensamblador en el simulador.

Cuestión 6.19: Ejecuta paso a paso el programa anterior y comprueba detenidamente la función de cada una de las instrucciones que constituyen el programa ensamblador.

Cuestión 6.20: ¿Qué valor se almacena en `n` después de ejecutar el programa?

Cuestión 6.21: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico

```
PROGRAMA MIENTRAS-2
VARIABLES
     VECTOR DE CARACTERES:
          tira1(4)="hola"; tira2(5)="adios";
     ENTERO: n;
INICIO
     i=0;
     n=0;
Mientras ((tira1[i])!=0) and (tira2[i]!=0)) hacer
          n=n+1;
          i=i+1;
     FinMientras
FIN
```

6.6. Estructura de control repetitiva *Para*

Crea un fichero con el siguiente fragmento de código que implementa una estructura de control repetitiva *Para*. Ésta es una estructura de control *Mientras* donde el bucle se repite un número de veces conocido a priori. Para implementarla se necesita pues un contador (inicializado a 0 ó al máximo número de veces menos uno que queremos que se repita el bluce) que llevará la cuenta de las veces que éste se repite. Este contador se deberá incrementar o decrementar dentro del bucle. La condición de salida del bucle siempre será comprobar si este contador alcanza la cota superior o inferior (0) (según el contador cuente de forma ascendente o descendente).

```
     .data
vector   .word    6,7,8,9,10,1
res      .space   4
     .text
main: la $t2,vector      #$t2=dirección de vector
      and $t3,$0,$t3     #$t3=0
      li $t0,0           #$t0=0
      li $t1,6           #$t1=5
para: bgt $t0,$t1,finpara #si $t0>$t1 saltar finpara
      lw $t4,0($t2) #carga elemento vector en $t4
      add $t3,$t4, $t3 #suma los elementos del vector
      addi $t2,$t2, 4    #$t2=$t2+4
      addi $t0,$t0, 1    #$t0=$t0+1
      j para             #saltar a bucle
finpara:  sw $t3, res($0) #almacenar $t3 en res
```

La descripción algorítmica de este programa en ensamblador se puede transcribir como un bucle repetitivo mientras se muestra a continuación:

```
PROGRAMA PARA-1
VARIABLES
     VECTOR DE ENTEROS: v(6)=(6,7,8,9,10,1);
     ENTERO: res;
INICIO
     res=0;
     i=0;
          Mientras (i<=5) hacer
               res=res+v[i];
               i=i+1;
```

```
        FinMientras
FIN
```

Una descripción algorítmica equivalente es utilizar una estructura de control Para que incluyen la mayoría de los lenguajes de alto nivel, por ejemplo *For* de Pascal, de C etc. Así utilizando esta estructura la descripción algorítmica del programa ensamblador anterior resulta más sencilla:

```
PROGRAMA PARA-1
VARIABLES
    VECTOR DE ENTEROS: v(6)=(6,7,8,9,10,1);
    ENTERO: res;
CODIGO
    res=0;
    Para i=0 hasta 5 hacer
        res=res+v[i];
    FinPara
FIN
```

Borra los valores de la memoria, carga el fichero que contiene el programa en ensamblador en el simulador.

Cuestión 6.22: Ejecuta paso a paso el programa y comprueba detenidamente la función que tiene cada una de las instrucciones que componen el programa en ensamblador.

Cuestión 6.23: ¿Qué valor se almacena en `res` después de ejecutar el código anterior?

Cuestión 6.24: Implementa en ensamblador el siguiente programa descrito en lenguaje algorítmico:

```
PROGRAMA PARA-2
VARIABLES
    VECTOR DE ENTEROS: v1(8)=(6,7,8,9,10,-1,34,23);
                       v2(8);
INICIO
    Para i=0 hasta 7 hacer
        v2[i]=v1[i]+1;
    FinPara
FIN
```

Problemas propuestos

1. Diseña un programa en ensamblador que almacene en memoria los 5 enteros siguientes (dato1=2, dato2=10, dato3=50, dato4=70, dato5=34) y que reserve 1 palabra para almacenar el resultado, (variable res). Implementa en ensamblador del R2000 un programa que almacene en la variable res un 1 si dato5 está en alguno de los intervalos formados por dato1 y dato2 o dato3 y dato4. Se almacenará un cero en caso contrario.

2. Diseña un programa en ensamblador que dado un vector de enteros, obtenga como resultado cuántos elementos son iguales a cero. Este resultado se debe almacenar sobre la variable "total". El programa deberá inicializar los elementos del vector en memoria, así como una variable que almacenará el número de elementos que tiene el vector y reservará espacio para la variable resultado.

3. Diseña un programa en ensamblador que dado un vector de enteros "V" obtenga cuántos elementos de este vector están dentro del rango determinado por dos variables "rango1" y "rango2". El programa deberá inicializar los elementos del vector en memoria, una variable que almacenará el número de elementos que tiene ese vector y dos variables donde se almacenarán los rangos. También deberá reservar espacio para la variable resultante.

4. Diseña un programa en ensamblador que dado un vector de caracteres, contabilice cuántas veces se repite un determinado carácter en el mismo. El programa deberá inicializar la cadena en memoria, y ésta deberá finalizar con el carácter nulo. También deberá reservar espacio para la variable resultado.

CAPÍTULO 7. INTERFAZ CON EL PROGRAMA

El SPIM ofrece un pequeño conjunto de servicios tipo Sistema Operativo a través de la instrucción de llamada al sistema (*syscall*). Estos servicios permiten introducir datos y visualizar resultados, de forma cómoda, en los programas que se desarrollan. Para pedir un servicio, el programa debe cargar el número identificador del tipo de llamada al sistema en el registro $v0 y el/los argumento/s en los registros $a0-$a3 (o f12 para valores en coma flotante). Las llamadas al sistema que retornan valores ponen sus resultados en el registro $v0 ($f0 en coma flotante). La interfaz con nuestro programa se hará a través de una ventana o consola que puede visualizarse u ocultarse mediante el botón "terminal". A continuación, se muestra una tabla resumen del número de identificación de cada una de las funciones disponibles y de los parámetros que son necesarios en cada una de ellas:

Servicio	Número de identificación (en $v0)	Argumentos	Resultado
Imprimir_entero	1	$a0=entero	
Imprimir_cadena	4	$a0=dir_cadena	
Leer_entero	5		Entero (en $v0)
Leer_string	8	$a0=dir_cadena, $a1=longitud	

7.1. Impresión de una cadenas de caracteres

Crea un fichero con el siguiente código:

```
          .data
dir:      .asciiz "Hola. Ha funcionado."
          .text
main:     li $v0,4        # código de imprimir cadena
          la $a0,dir      # dirección de la cadena
          syscall         # Llamada al sistema
```

Descripción:

La pseudoinstrucción “li” carga de forma inmediata el dato que contiene la instrucción en el registro. La pseudoinstrucción “la” carga de forma inmediata la dirección que contiene la instrucción en el registro especificado.

Borra los valores de la memoria, carga el fichero y ejecútalo.

Cuestión 7.1: Comprueba que se imprime la cadena de caracteres “Hola. Ha funcionado.” Vuélvelo a ejecutar. Acuérdate que antes de volver a ejecutar el programa se debe borrar el contenido de los registros. ¿Dónde se imprime ahora el mensaje anterior?

Cuestión 7.2: Introduce al final de la cadena los caracteres “\n”. Ejecútalo dos veces y comprueba cuál es la diferencia con respecto al código original.

Cuestión 7.3: Modifica la línea de la instrucción “la ...” para que sólo se imprima “Ha funcionado.”.

7.2. Impresión de enteros

Crea un fichero con el siguiente código:

```
         .data
dir:     .asciiz "Se va a imprimir el entero:  "
entero:  .word    7
         .text
main:    li $v0, 4      # código de imprimir cadena
         la $a0, dir    # dirección de la cadena
         syscall        # Llamada al sistema
         li $v0, 1      # código de imprimir entero
         li $a0, 5      # entero a imprimir
         syscall        # Llamada al sistema
```

Borra los valores de la memoria, carga el fichero y ejecútalo.

Cuestión 7.4: Comprueba la impresión del entero “5”.

Cuestión 7.5: Modifica el programa para que se imprima el entero almacenado en la dirección “entero”. Comprueba su ejecución.

7.3. Lectura de enteros

Crea un fichero con el siguiente código:

```
        .data
dir:    .asciiz "Introduce el entero: "
        .align 2
entero: .space   4
        .text
main:   li $v0, 4       # código de imprimir cadena
        la $a0, dir     # dirección de la cadena
        syscall         # Llamada al sistema
        li $v0, 5       # código de leer entero
        syscall         # Llamada al sistema
```

Borra los valores de la memoria, carga el fichero y ejecútalo. Introduce, a través de la consola, el entero que te parezca.

Cuestión 7.6: Comprueba el almacenamiento del entero introducido en el registro $v0.

Cuestión 7.7: Modifica el programa para que el entero leído se almacene en la dirección de memoria `entero`.

Cuestión 7.8: Elimina la línea `align 2` del programa anterior y ejecútalo. ¿Qué ocurre? ¿Por qué?

7.4. Lectura de cadenas de caracteres

Crea un fichero con el siguiente código:

```
        .data
dir:    .asciiz "Introduce los caracteres: \n"
buffer: .space 10 # zona para los caracteres
        .text
main:   li $v0,4  # código de imprimir cadena
```

```
la $a0,dir     # dirección de la cadena
syscall        # Llamada al sistema
li $v0,8       # código de leer el string
la $a0,buffer # dirección lectura cadena
li $a1,10 # espacio máximo cadena
syscall        # Llamada al sistema
```

Borra los valores de la memoria, carga el fichero y ejecútalo. Introduce, a través de la consola, la cadena de caracteres que te parezca.

Cuestión 7.9: Comprueba el almacenamiento de la cadena de caracteres en `buffer`.

Cuestión 7.10: ¿Qué ocurre si la cadena de caracteres que se escribe es más larga que el tamaño reservado en `buffer`?

Problemas propuestos

1. Diseña un programa ensamblador que pida por la consola dos enteros "A" y "B" dando como resultado su suma "A+B".

2. Diseña un programa que pida los datos de un usuario por la consola y los introduzca en memoria, reservando espacio para ello. La estructura de la información es la siguiente:

Nombre	cadena de 10 bytes
Apellidos	cadena de 15 bytes
DNI	entero

A continuación se deberá imprimir los datos introducidos para comprobar el correcto funcionamiento del programa.

CAPÍTULO 8. GESTIÓN DE SUBRUTINAS

El diseño de un programa que resuelve un determinado problema puede simplificarse si se plantea adecuadamente la utilización de subrutinas. Éstas permiten dividir un problema largo y complejo en subproblemas más sencillos o módulos, más fáciles de escribir, depurar y probar que si se aborda directamente el programa completo. De esta forma se puede comprobar el funcionamiento individual de cada rutina y, a continuación, integrar todas en el programa que constituye el problema global de partida. Otra ventaja que aporta la utilización de subrutinas es que en ocasiones una tarea aparece varias veces en el mismo programa; si se utilizan subrutinas, en lugar de repetir el código que implementa esa tarea en los diferentes puntos, bastará con incluirlo en una subrutina que será invocada en el programa tantas veces como sea requerida. Yendo más lejos, subproblemas de uso frecuente pueden ser implementados como rutinas de utilidad (librerías), que podrán ser invocadas desde diversos módulos.

Con el fin de dotar de generalidad a una subrutina, ésta ha de ser capaz de resolver un problema ante diferentes datos que se le proporcionan como parámetros de entrada cuando se la llama. Por tanto, para una correcta y eficaz utilización de las subrutinas es necesario tener en cuenta, por un lado, la forma en que se realizan las *llamadas* y, por otro, el *paso de parámetros*.

Este capítulo comenzará con la presentación de algunos ejercicios sencillos que permitan la familiarización con el manejo de la pila, estructura imprescindible para una buena gestión de las subrutinas. Posteriormente se introducirán las técnicas de llamada, paso de parámetros y gestión de las variables.

8.1. Gestión de la pila

Una pila o cola LIFO (Last Input First Output) es una estructura de datos caracterizada por que el último dato que se almacena es el primero que se obtiene después. Para gestionar la pila se necesita un puntero a la última posición ocupada de la misma, con el fin de conocer dónde se tiene que dejar el siguiente dato a almacenar, o para saber dónde están situados los últimos datos almacenados en ella. Para evitar problemas, el puntero de pila siempre debe estar apuntando a una palabra de memoria. Por precedentes históricos, el segmento de pila siempre "crece" de direcciones superiores a direcciones inferiores. Las dos operaciones más típicas con esta estructura son:

- Transferir datos hacia la pila (*push* o apilar). Antes de añadir un dato a la pila se tienen que restar 4 unidades al puntero de pila.

- Transferir datos desde la pila (*pop* o desapilar). Para eliminar datos de la pila, después de extraído el dato se tienen que sumar 4 unidades al puntero de pila.

El ensamblador del MIPS tiene reservado un registro, $sp, como puntero de pila (stack pointer). Para realizar una buena gestión de la pila será necesario que este puntero sea actualizado correctamente cada vez que se realiza una operación sobre la misma.

El siguiente fragmento de código muestra cómo se puede realizar el apilado de los registros $t0 y $t1 en la pila:

```
         .text
main:    li $t0,10
         li $t1, 13      #inicializar reg. t0,$t1
         addi $sp, $sp, -4  #actualizar el sp
         sw   $t0, 0($sp)   #apilar t0
         addi $sp, $sp, -4  #actualizar el sp
         sw   $t1, 0($sp)   #apilar t1
```

Edita el programa, reinicializa el simulador y carga el programa.

Cuestión 8.1: Ejecuta el programa paso a paso y comprueba en qué posiciones de memoria, pertenecientes al segmento de pila se almacena el contenido de los registros $t0 y $t1.

Cuestión 8.2: Modifica el programa anterior para que en lugar de actualizar el puntero de pila cada vez que se pretende apilar un registro en la misma, se realice una sola vez al principio y después se apilen los registros en el mismo orden.

Cuestión 8.3: Añade el siguiente código al programa original después de la última instrucción:

- Modifica el contenido de los registros $t0 y $t1, realizando algunas operaciones sobre ellos.
- Recupera el contenido inicial de estos registros desapilándolos de la pila, y actualiza el puntero de pila correctamente (limpiar la pila).

Ejecuta el programa resultante y comprueba si finalmente $t0 y $t1 contienen los datos iniciales que se habían cargado. Comprueba también que el contenido del registro $sp es el mismo antes y después de la ejecución del programa.

Cuestión 8.4: Implementa el siguiente programa:

- Almacena en memoria una tira de caracteres de máximo 10 elementos.

- Lee la cadena de caracteres desde el teclado.
- Invierte esta cadena, almacenando la cadena resultante en las mismas posiciones de memoria que la original. (Sugerencia para realizar el ejercicio: Se apilan los elementos de la tira en la pila y, a continuación, se desapilan y se almacenan en el mismo orden que se extraen).

8.2. Llamada y retorno de una subrutina

El juego de instrucciones del MIPS R2000 dispone de una instrucción específica para realizar la llamada a una subrutina, `jal etiqueta`. La ejecución de esta instrucción conlleva dos acciones:

- Almacenar la dirección de memoria de la siguiente palabra a la que contiene la instrucción `jal` en el registro $ra.
- Llevar el control de flujo de programa a la dirección `etiqueta`.

Por otra parte, el MIPS R2000 dispone de una instrucción que facilita el retorno de una subrutina. Esta instrucción es `jr $ra`, instrucción de salto incondicional que salta a la dirección almacenada en el registro $ra, que justamente es el registro dónde la instrucción `jal` ha almacenado la dirección de retorno cuando se ha hecho el salto a la subrutina.

Para ejecutar cualquier programa de usuario el simulador XSPIM hace una llamada a la rutina `main` mediante la instrucción `jal main`. Esta instrucción forma parte del código que añade éste para lanzar a ejecución un programa de usuario. Si la etiqueta `main` no está declarada en el programa se genera un error. Esta etiqueta deberá siempre referenciar la primera instrucción ejecutable de un programa de usuario. Para que cualquier programa de usuario termine de ejecutarse correctamente, la ultima instrucción ejecutada en éste debe ser `jr $ra` (salto a la dirección almacenada en el registro $ra), que devuelve el control a la siguiente instrucción desde donde se lanzó la ejecución del programa de usuario. A partir de este punto se hace una llamada a una función del sistema que termina la ejecución correctamente.

El siguiente código es un programa que realiza la suma de dos datos contenidos en los registros $a0 y $a1.

```
          .text
main:     li $a0,10
          li $a1,20
          add $v1,$a0,$a1
          jr $ra
```

Reinicializa el simulador, carga el programa y ejecútalo paso a paso, contestando a las siguientes cuestiones:

Cuestión 8.5: ¿Cuál es el contenido del PC y del registro $ra antes y después de ejecutar la instrucción `jal main`?

Cuestión 8.6: ¿Cuál es el contenido de los registros PC y $ra antes y después de ejecutar la instrucción `jr $ra`?

Cuestión 8.7: Reinicializa el simulador, carga el programa de nuevo y ejecútalo todo completo. Comprueba que la ejecución termina correctamente sin que salga el mensaje de error que salía en ejecuciones anteriores cuando no se incluía la instrucción `jr $ra`.

8.3. Anidado de subrutinas

El anidado de subrutinas se produce cuando se encadenan dos o más llamadas a subrutinas desde otras subrutinas. Esta circunstancia es muy habitual cuando se trabaja con librerías y se están diseñando programas complejos y bien estructurados.

Cuando una subrutina llama a otra utilizando la instrucción `jal` se modifica automáticamente el contenido del registro $ra con la dirección de retorno (dirección de memoria de la siguiente palabra a la que contiene la instrucción que ha efectuado el salto, es decir, la dirección de la instrucción posterior al salto). Esto hace que se pierda cualquier contenido anterior que pudiera tener este registro, que podría ser a su vez otra dirección de retorno suponiendo que se llevan efectuadas varias llamadas anidadas. Así pues, en cada llamada a una subrutina se modifica el contenido del registro $ra, y sólo se mantiene en este registro la dirección de retorno asociada a la última llamada, ¿Qué ocurre entonces con todas las direcciones de retorno que deben guardarse en las llamadas anidadas?

Una de las soluciones a este problema es que antes de ejecutar una llamada desde una subrutina a otra se salve el contenido del registro $ra en la pila. Como la pila crece dinámicamente, las direcciones de retorno de las distintas llamadas anidadas quedarán almacenadas a medida que éstas se van produciendo. La última dirección de retorno apilada estará en el tope de la pila, y ésta es justamente la primera que se necesita recuperar. Así pues, una pila es la estructura adecuada para almacenar las direcciones de retorno en llamadas anidadas a subrutinas.

El siguiente código implementa una llamada anidada a dos subrutinas, rutina `main` y `subr`. A la primera se le llamará desde el fragmento de código que tiene el simulador para lanzar la ejecución de un programa de usuario y a la subrutina `subr` se la llamará desde `main`. La primera necesita apilar la dirección de retorno (contenido de $ra) para que sea posible la vuelta a la instrucción siguiente desde donde se hizo la llamada a la rutina `main`. La segunda como no hace ninguna llamada a otra subrutina no necesita apilar el contenido de $ra.

La descripción algorítmica se muestra a continuación:

```
PROGRAMA EJEMPLO-1
VARIABLES
     ENTERO: dato1=10; dato2=20; suma;
INICIO
     suma=subr(dato1,dato2);
FIN

FUNCION subr(ENTERO: para1, para2):ENTERO;
INICIO
     subr=para1+para2;
FIN
```

Y el programa en ensamblador quedaría como sigue:

```
          .data
suma:     .space    4
dato1:    .word     10
dato2     .word     20
          .text
main:     addi $sp,$sp,-4    #apilar dir. ret.
          sw   $ra,0($sp)
          lw   $a0,dato1($0)
          lw   $a1,dato2($0)
          jal  subr
          sw   $v0,suma($0)
          lw   $ra,0($sp)    #desapilar dir. ret.
          addi $sp,$sp,4
          jr   $ra
subr:     add  $v0,$a0,$a1
          jr   $ra
```

Reinicializa el simulador, carga el programa y ejecútalo paso a paso respondiendo a las siguientes cuestiones:

Cuestión 8.8: Comprueba qué contiene el registro $ra y el PC antes y después de ejecutar la instrucción `jal main`.

Cuestión 8.9: Comprueba qué hay almacenado en el tope de la pila después de ejecutar las dos primeras instrucciones del programa .

Cuestión 8.10: Comprueba el contenido de los registros $ra y PC antes y después de ejecutar la instrucción `jal subr`.

Cuestión 8.11: Comprueba el contenido de los registros $ra y PC antes y después de ejecutar la instrucción `jr $ra` que está en la subrutina subr.

Cuestión 8.12: Comprueba el contenido de los registros $ra y PC antes y después de ejecutar la instrucción `jr $ra` que está en la subrutina main. ¿Qué hubiera ocurrido si antes de ejecutar la instrucción `jr $ra` de esta subrutina `main` no se hubiese desapilado el contenido del registro $ra?

Cuestión 8.13: Añade el código necesario para realizar las siguientes modificaciones sobre el código original:

La subrutina `subr` debería calcular la siguiente operación dato1/dato2. Esta división sólo se realizará si ambos operandos son mayores que 0. La comprobación se realizará en una segunda subrutina llamada `comp` que devolverá un 1, si ambos datos, `dato1` y `dato2` son mayores que 0, y un 0 en caso contrario. La descripción algorítmica del programa propuesto se muestra a continuación:

```
PROGRAMA EJEMPLO-2
VARIABLES
     ENTERO: dato1=10; dato2=20; division;
INICIO
     division=subr(dato1, dato2);
FIN

FUNCION comp(ENTERO: para1, para2):ENTERO;
INICIO
     Si ((para1>0) and (para2>0)) ENTONCES
          comp=1;
     Sino
          comp=0;
     FinSi
FIN
```

```
FUNCION subr(ENTERO: para1, para2):ENTERO;
VARIABLES
     ENTERO: mayor;
INICIO
     Si (comp(para1,para2)=1)
          subr=para1/para2;
     Sino
          subr=-1;
     FinSi
FIN
```

8.4. Paso de parámetros

A la hora de implementar una subrutina hay que decidir:

- Parámetros a pasar a la rutina.
- Tipo de parámetros:

  por valor:

  - se pasa el valor del parámetro,

  por referencia:

  - se pasa la dirección de memoria donde está almacenado el parámetro.

- Lugar donde se van a pasar los parámetros:
  - registros,
  - pila

El ensamblador del MIPS establece el siguiente convenio para realizar el paso de parámetros:

- Los 4 primeros parámetros de entrada se pasarán a través de los registros \$a0 - \$a3. A partir del quinto parámetro se pasaría a través de la pila.
- Los dos primeros parámetros de salida se devuelven a través de los registros \$v0 - \$v1, el resto a través de la pila.

El siguiente programa implementa la llamada a una subrutina y el código de la misma, que devuelve una variable booleana que vale 1 si una determinada variable está dentro de un rango y 0, en caso contrario. La descripción algorítmica del mismo se muestra a continuación:

```
PROGRAMA EJEMPLO-3

VARIABLES

     ENTERO: rango1=10; rango2=50; dato=12; res;

INICIO

     res=subr(rango1,rango2,dato);

FIN


FUNCION subr(ENTERO: para1, para2, para3):ENTERO;

INICIO

     Si ((para3>=para1)and(para3<=para2)) ENTONCES

          subr=1;

     Sino

          subr=0;

     FinSi

FIN
```

La subrutina tendrá, pues, los siguientes parámetros:

- Parámetros de entrada:
  - Las dos variables que determinan el rango, pasados por valor, a través de \$a0 y \$a1.
  - La variable que se tiene que estudiar si está dentro del rango, pasada por valor, a través de \$a2.
- Parámetros de salida:
  - Variable booleana que indica si la variable estudiada está o no dentro del rango, por valor, devuelto a través de \$v0.

El listado del programa en ensamblador se muestra a continuación:

```
            .data
rango1:     .word     10
rango2:     .word     50
dato:       .word     12
res:        .space    1
            .text
main:       addi $sp,$sp,-4
            sw $ra,0($sp)       #apilar ra
            lw $a0,rango1($0)   #a0=rango1
            lw $a1,rango2($0)   #a1=rango2
            lw $a2,dato($0)     #a2=dato
            jal subr       #saltar a subr
            sb $v0,res($0)      #res=v0
            lw $ra,0($sp)
            add $sp,$sp,4  #desapilar ra
            jr $ra         #terminar ejecucion programa
subr:       blt $a2,$a0,sino    #Si a2<a0 saltar a sino
            bgt $a2,$a1,sino    #si a2>a1 saltar a sino
entonces:   addi $v0, $0,1      #v0=1
            j finsi             #saltar a finsi
sino:       add $v0,$0,$0       #v0=0
finsi:      jr $ra              #retornar
```

Reinicializa el simulador, carga el programa, ejecútalo y comprueba el resultado almacenado en la posición de memoria `res`. Contesta a las siguientes cuestiones:

Cuestión 8.14: Identifica las instrucciones que se necesitan en:

- El programa que hace la llamada para:

  a) La carga de parámetros en los registros.

b) La llamada a la subrutina.

c) El almacenamiento del resultado.

- La subrutina para:

a) La lectura y procesamiento de parámetros.

b) La carga del resultado en $v0

c) El retorno al programa que ha hecho la llamada.

Cuestión 8.15: Modifica el código anterior para que los parámetros que se pasan a la subrutina `subr`, tanto los de entrada como el de salida, se pasen por referencia. La descripción algorítmica del programa se muestra a continuación:

```
PROGRAMA EJEMPLO-4

VARIABLES

     ENTERO: rango1=10; rango2=50; dato=12; res;

INICIO

     subr(rango1,rango2,dato,res);

FIN


PROCEDIMIENTO subr(ENTERO: para1,para2;para3;

                   VAR ENTERO: para4);

INICIO

     Si (para3>=para1)or(para3<=para2)) ENTONCES

          para4=1;

     Sino

          para4=0;

     FinSi

FIN
```

Los pasos a realizar tanto por el programa que hace la llamada como por la subrutina son los siguientes:

- En el programa que hace la llamada:

  a) Cargar la dirección de los parámetros en los registros correspondientes (entrada y salida).

  b) Llamada a la subrutina.

- En la subrutina:

  a) Lectura de los parámetros de entrada a partir de las direcciones pasada como parámetro.

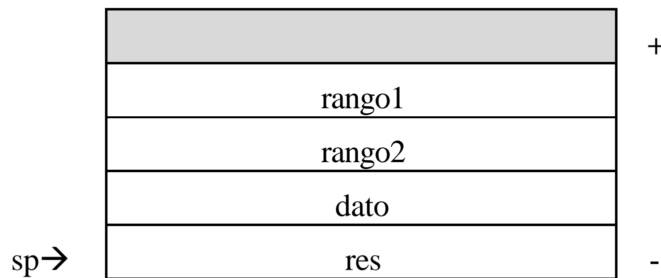
  b) Procesamiento de los parámetros de entrada y generación del resultado.

  c) Almacenamiento del resultado en la dirección de memoria pasada a través del parámetro de salida

  d) Retorno al programa que hizo la llamada.

Ejecuta el programa obtenido y comprueba que el resultado obtenido es el mismo que el del programa original.

Cuestión 8.16: Modifica el código anterior para que los parámetros de entrada a la subrutina se pasen por valor mediante la pila y el de salida se pase también a través de la pila pero por referencia. A continuación se muestra un esquema de cómo debe ser la situación de la pila en el momento que empieza a ejecutarse la subrutina:



8.5. Bloque de activación de la subrutina

El bloque de activación de la subrutina es el segmento de pila que contiene toda la información referente a la llamada a una subrutina (parámetros pasados a través de la pila, registros que modifica la subrutina y variables locales). Un bloque típico abarca la memoria entre el puntero de bloque, normalmente llamado registro $fp (en el ensamblador del MIPS se trata del registro r30), que apunta a la primera palabra almacenada en el bloque, y el puntero de pila ($sp), que apunta a la última palabra del bloque. El puntero de pila puede variar su contenido durante la ejecución de la subrutina y, por lo tanto, las referencias a una variable local o a un parámetro pasado a través de la pila podrían tener diferentes desplazamientos relativos al puntero de pila

dependiendo de dónde estuviera éste en cada momento. De forma alternativa, el puntero de bloque apunta a una dirección fija dentro del bloque. Así pues, la subrutina en ejecución usa el puntero de bloque de activación para acceder mediante desplazamientos relativos a éste a cualquier elemento almacenado en el bloque de activación independientemente de dónde se encuentre el puntero de pila.

Los bloques de activación se pueden construir de diferentes formas. No obstante, lo que realmente importa es que el programa que hace la llamada y la subrutina deben estar de acuerdo en la secuencia de pasos a seguir. Los pasos que se enumeran a continuación describen la convención de llamada y retorno de una subrutina que se adopta. Esta convención interviene en tres puntos durante una llamada y retorno a una subrutina: Inmediatamente antes de llamar a la subrutina e inmediatamente después del retorno de la misma, en el programa que hace la llamada, en el momento justo en el que la subrutina empieza su ejecución e inmediatamente antes de realizar el retorno en la subrutina:

- En el programa que hace la llamada:

  - Inmediatamente antes de hacer la llamada a la subrutina:

    Se deben cargar los parámetros de entrada (y los de salida si se pasan por referencia) en los lugares establecidos, los cuatro primeros en registros y el resto en la pila.

  - Inmediatamente después del retorno de la subrutina:

    Se debe limpiar la pila de los parámetros almacenados en ella, actualizando el puntero de pila.

- En la subrutina:

  - En el momento que empieza la ejecución:

    - Reservar espacio en la pila para apilar todos los registros que la subrutina vaya a modificar y para las variables locales que se almacenarán en la pila.

    - Puesto que el registro $fp es uno de los registros modificados por la subrutina, va a ser utilizado como puntero al bloque de activación, deberá apilarse. Para que la posterior limpieza del bloque de activación sea más sencilla, conviene apilarlo como primera palabra detrás de los parámetros que ha apilado el programa que ha hecho la llamada (si los hubiese).

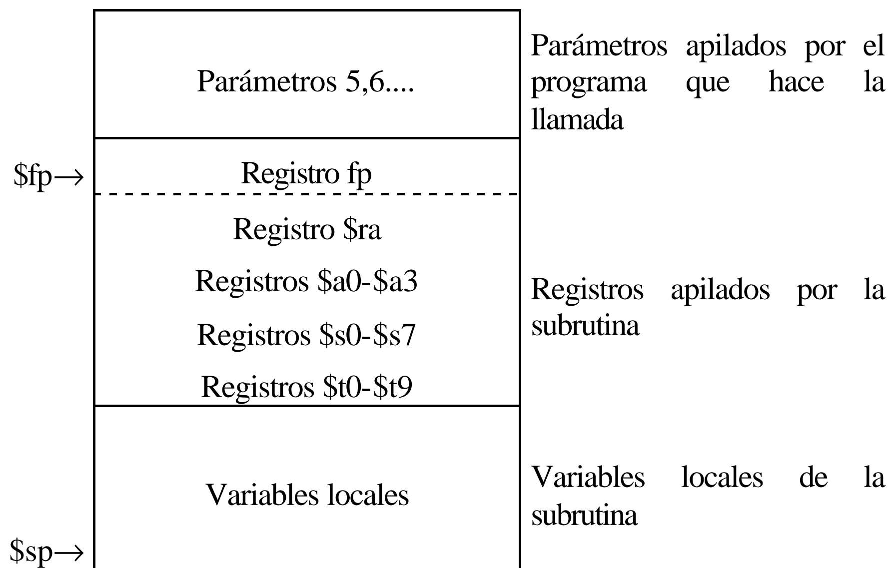
    - Actualizar el contenido del registro fp, para que apunte a la posición de la pila donde acabamos de almacenar el contenido del fp.

    - Apilar el contenido del resto de registros que se van a modificar, que pueden ser: $t0 - $t9, $s0 - $s7, $a0 - $a3, $ra.

- Inmediatamente antes del retorno:

  - Recuperar el contenido de los registros apilados y actualizar el puntero de pila para limpiar la pila.

El Bloque de activación de una subrutina tendría, por lo tanto, la siguiente estructura:



El siguiente programa implementa una subrutina que calcula los elementos nulos de un vector.

```
PROGRAMA EJEMPLO-5

VARIABLES

     ENTERO: n1=4; vec1(4)=(1,0,0,2), nul1;

INICIO

     nul1=subr(vec1,n1);

FIN

FUNCION subr(VAR ENTERO: para1(4);

               ENTERO: para2):ENTERO

INICIO

     subr=0;

     Para i=0 hasta para2-1 hacer
```

```
            Si (para1[i]=0) entonces
                 subr=subr+1;
            FinSi
        FinPara
    FIN
```

A partir de esta descripción algorítmica se puede realizar la implementación del programa en ensamblador. Para ello en primer lugar se va a decidir los parámetros que se le pasan a la subrutina:

- Parámetros de entrada:
  - La dimensión del vector (pasado por valor).
  - La dirección del primer elemento del vector (pasado por referencia). Siempre que se pase a una subrutina un dato de este tipo, un vector o una matriz, siempre se pasará por referencia, es decir, la dirección del primer elemento del vector o de la matriz.

Estos parámetros se pasarán a través de la pila ya que el objetivo que se pretende remarcar con este programa es la gestión del bloque de activación de una subrutina. Según el convenio establecido en este capítulo, estos parámetros se pasarían a través de los registros a0 y a1.

- Parámetros de salida:
  - Contador de elementos nulos del vector. Este parámetro se devolverá a través del registro v0.

En la subrutina se termina de crear el bloque de activación de la subrutina, que estará formado por los parámetros que se han pasado a través de la pila (parámetros de entrada) y por los registros que vaya a modificar la subrutina, que apilará al principio de la ejecución de la subrutina. No se reserva espacio para variables locales puesto que se utilizan registros para almacenarlas.

Por otro lado, para que se pueda apreciar mejor la evolución del bloque de activación de la subrutina en distintos puntos del programa, en el programa `main` lo primero que se hace es inicializar los registros \$s0, \$s1, \$s2 y \$fp a unos valores. Al finalizar la ejecución del programa el contenido de estos registros debería ser el mismo.

```
        .data
n1:     .word   4
vec1:   .word   1,0,0,2
```

```
nul1:  .space  4
       .text
# primero se inicializan los registros s0,s1,s2 y fp
main:  li $s0,1         #iniciliza $s0
       li $s1,2         #iniciliza $s1
       li $s2,3         #iniciliza $s2
       li $fp,4         #iniciliza $fp
       addi $sp,$sp,-4
       sw $ra,0($sp)    #apilar $ra
       addi $sp, $sp,-8
       lw $t0, n1($0)
       sw $t0,4($sp)
       la $t0, vec1
       sw $t0, 0($sp)
       jal subr         #llamada a la subrutina
ret:   addi $sp,$sp,8
       sw $v0,nul1($0)
       lw $ra,0($sp)
       addi $sp,$sp,4
       jr $ra
subr:  addi $sp,$sp,-16
       sw $fp, 12($sp)
       addi $fp,$sp,12
       sw $s0,-4($fp)
       sw $s1,-8($fp)
       sw $s2,-12($fp)
       lw $s0,4($fp)
```

```
        lw $s1,8($fp)
        and $v0, $v0,$0
#bucle de cuenta de elementos nulos
bucle:  beq $s1,$0,finb  # si s1 = 0 saltar a finb
        lw  $s2, 0($S0)  # cargar s2=Mem(s0)
        bne $s2, $0, finsi #si s3<>0 saltar a finsi
        addi $v0,$v0,1   #v0=s2
finsi:  addi $s0, $s0,4  # s0 = s0+4
        addi $s1, $s1,-1 # s1=s1-1
        j bucle          #saltar a bucle
finb:   lw $s0,-4($fp)
        lw $s1,-8($fp)
        lw $s2,-12($fp)
        addi $sp,$fp,0
        lw $fp,0($sp)
        addi $sp,$sp,4
        jr $ra
```

Cuestión 8.17: Identifica la instrucción o conjunto de instrucciones que realizan las siguientes acciones en programa que hace la llamada a la subrutina subr y en la subrutina:

- En el programa que hace la llamada:
  a) Carga de parámetros en la pila.
  b) Llamada a la subrutina.
  c) Limpieza de la pila de los parámetros pasados a través de ella.
  d) Almacenamiento del resultado.
- En la subrutina:
  a) Actualización del bloque de activación de la subrutina.
  b) Lectura de los parámetros de la pila y procesamiento de los mismos.

c) Carga del resultado en $v0.

d) Desapilar bloque de activación los registros apilados y eliminar las variables locales. Actualizar el puntero de pila.

e) Retorno al programa que ha hecho la llamada.

Cuestión 8.18: Ejecuta paso a paso el programa anterior y dibuja la situación del bloque de activación y de los registros $s0, $s1, $s2, $fp, $sp, $ra y PC en cada una de las siguientes situaciones:

- Programa que hace la llamada a la subrutina `subr`, hasta el momento que hace la llamada:

a) Antes y después de ejecutar la instrucción `addi $sp,$sp,-8` (situación uno).

b) Antes y después de ejecutar la instrucción `jal subr`.(situación justo antes de llamar a la subrutina)

- Subrutina:

c) Después de ejecutar la instrucción `addi $sp, $sp, -16`.

d) Después de ejecutar la instrucción `sw $fp,12($sp)`.

e) Después de ejecutar la instrucción `addi $fp,$sp,12`.

f) Después de ejecutar la instrucción `sw $s2,-12($fp)`.

g) Después de ejecutar la instrucción `lw $s0,4($fp)`.

h) Después de ejecutar la instrucción `lw $s1,4($fp)`.

i) Después de ejecutar el bucle (denominado `bucle`).

j) Antes y después de ejecutar `addi $sp,$fp,0`.

k) Después de ejecutar `lw $fp,0($sp)`.

l) Después de ejecutar `addi $sp,$sp,4`. Comprobar qué la situación de la pila y de los registros $sp y $fp es la misma que justo antes de llamar a la subrutina).

m) Después de ejecutar `jr $ra`

- Programa que hace la llamada después de retornar de la subrutina:

n) Antes y después de ejecutar `addi $sp,$sp,8`. Comprobar qué el contenido de los registros y de la pila se corresponde con la situación uno.

Cuestión 8.19: Modifica el código para que los parámetros se pasen a través de los registros $a0 - $a1 y el de salida a través de $v0. Supón que el programa que hace la llamada pretende que después de ejecutar la subrutina estos registros contengan el mismo contenido que el que se ha cargado inicialmente en el programa.

Problemas propuestos

1. Implementa una subrutina en ensamblador que calcule cuántos elementos de un vector de enteros de dimensión n son iguales a un elemento dado:

   - ❑ Parámetros de entrada a la subrutina:
     - dirección del primer elemento del vector,
     - total de elementos del vector (dimensión),
     - elemento a comparar.
   - ❑ Parámetro de salida de la subrutina:
     - contador calculado.

   Realiza las siguientes implementaciones:

   a) Implementa la subrutina de forma que todos los parámetros que se le pasan sean por valor excepto aquéllos que obligatoriamente se deban pasar por referencia, y a través de registros.

   b) Implementa la subrutina de forma que todos los parámetros se pasen por referencia y a través de registros.

   c) Implementa la subrutina donde los parámetros que se pasan sean del mismo tipo que en el caso a) pero utilizando la pila como lugar para realizar el paso de parámetros.

2. Implementa una subrutina en ensamblador, tal que dado un vector de enteros de dimensión *n* obtenga el elemento (i) de dicho vector. La subrutina tendrá como parámetros de entrada: la dirección del vector, la dimensión del mismo y el índice del elemento a devolver. La subrutina devolverá el elemento i-ésimo. Realiza la llamada y el retorno a la subrutina según el convenio establecido.

3. Implementa una subrutina en ensamblador, tal que dada una matriz de enteros de dimensión n x m, almacenada por filas, obtenga el elemento (i,j) de dicha matriz. La subrutina tendrá como parámetros de entrada: la dirección de la matriz, las dimensiones de la misma y los índices del elemento a devolver. La subrutina devolverá el elemento (i,j). Realiza la llamada y el retorno a la subrutina según el convenio establecido.

CAPÍTULO 9. GESTIÓN DE LA E/S MEDIANTE CONSULTA DE ESTADO

Los computadores no funcionan aisladamente, tienen que relacionarse habitualmente con el entorno que les rodea y con el cual interactúan: usuarios, otros computadores, herramientas electromecánicas en la industria, etc. Para que esto sea posible se han diseñado sistemas capaces de realizar este cometido denominados *periféricos*: teclado, monitor, impresora, scanner, etc. Los periféricos se comunican con el computador mediante el intercambio de información o datos, y al sistema que engloba estos mecanismos de intercambio de datos se le denomina *sistema de Entrada/Salida* (E/S). Al elemento del computador que permite comunicarse con un periférico se le llama *interfaz* (o *interface*) y está formado principalmente por un circuito, más o menos complicado, denominado *controlador* o dispositivo de E/S.

Estas interfaces y controladores son específicos de cada periférico y el computador accede a ellos, para programarlos y/o comunicarse con los periféricos, leyendo y escribiendo en unos elementos direccionables denominados *puertos*. Cuando las direcciones de estos puertos se corresponden con direcciones del espacio de memoria del computador, para leer o escribir en estos puertos se utilizan las mismas instrucciones que para acceder a la memoria y se dice que el sistema de entrada/salida está *mapeado en memoria* (el utilizado por MIPS). Si por el contrario se utilizan instrucciones especiales, se dice que el computador tiene un *mapa de direcciones de E/S independiente* (es el caso de Intel).

Los periféricos pueden interactuar de forma muy distinta con el computador dependiendo de cuales sean sus características y va ha ser crítico, para el buen funcionamiento del computador, que el flujo de información entre el computador y los periféricos se adecuen a las características de estos últimos.

En este capítulo se aborda uno de los métodos más sencillos para la gestión de la E/S denominado *consulta de estado*. Para ello utilizaremos como periféricos el teclado (entrada) y la pantalla (la consola) del ordenador personal (salida).

La técnica de consulta de estado presupone que el controlador del dispositivo posee un *puerto de lectura/escritura de los datos* y al menos un *puerto de control*. La lectura de este puerto de control por parte del procesador, y en particular de un bit especial llamado *bit de estado*, indica si el controlador está preparado para leer o escribir los datos a través del periférico mediante el puerto de lectura/escritura de datos.

En el simulador XSPIM, para la entrada de datos desde el teclado, el puerto de datos se ubica en la dirección 0xffff0004 y el puerto de control en la dirección 0xffff0000 (siendo el bit 0 el bit de estado, e indica que el controlador posee un

carácter pulsado desde el teclado). Para la salida de datos por la consola el puerto de datos se ubica en la dirección 0xffff000c y el puerto de control en la dirección 0xffff0008 (siendo el bit 0 el bit de estado e indica que puede escribirse un carácter para ser mostrado por la consola)

9.1. Entrada de datos desde el teclado

En este apartado vamos a estudiar el control de la entrada de datos desde el teclado del ordenador.

Crea un fichero con el siguiente código:

```
         .data 0x10010000
long:    .word     7 # tamaño del buffer
buffer:  .space    7 #buffer donde se almacenan los
                     #caracteres pulsados
         .data 0xffff0000
cin:     .space 1  #Puerto de control del teclado
         .data 0xffff0004
in:      .space 1  # Puerto de lectura del teclado
         .text
         .globl main
main:    la $a0, buffer     #carga dir buffer
         lw $v0,long($0)    #control longitud del buffer
         addi $v0,$v0,-1
         li $v1,0x0a   # return o cambio línea (\n)
         la $a1, in    #carga dir in
         la $a2, cin   #carga dir control teclado
ctr:          # por ahora no ponemos nada
         lb $s0, 0($a1)     # lee del puerto del teclado
         sb $s0,0($a0) # almacena el dato en el buffer
         addi $a0,$a0,1#incremento puntero del buffer
```

```
        addi $v0,$v0,-1 # decr. tamaño restante buffer
        bne $v0,$0, ctr # control de fin de cadena
fin:    li $s0, 0x00
        sb $s0,0($a0)  #almacena fin cadena en buffer
        jr $ra         #vuelve a main
```

Descripción:

El código anterior es un programa que lee los caracteres introducidos desde el teclado pero sin realizar correctamente el control de la entrada de datos mediante consulta de estado, eliminado del código a propósito.

Con simulador XSPIM ejecutado con la opción "–mapped_io", carga y ejecuta el programa editado en el simulador.

Como puedes comprobar, el programa se ejecuta de un tirón y no da oportunidad para que puedas introducir nada desde el teclado.

Cuestión 9.1: ¿Qué carácter se almacena en el espacio reservado en `buffer`?

Cuestión 9.2: ¿Por qué se almacena siempre el mismo valor en cada byte?

Ahora haremos las cosas bien y vamos a leer del puerto de la entrada de datos cuando se haya pulsado una tecla. Para ello hay que inserta el siguiente código en la etiqueta `ctr`:

```
############## consulta de estado #############
ctr:     lb $t0, 0($a2)
         andi $t0,$t0,1
         beq $t0,$0, ctr
################################################
```

Cuestión 9.3: Describe con detalle qué está haciendo esta parte del código.

Comprueba el correcto comportamiento del programa con el código de consulta de estado introducido.

Cuestión 9.4: Modifica el código del programa completo añadiendo el bucle de consulta de estado para que cuando se pulse la tecla "return", el programa almacene este carácter y el carácter de fin cadena (0), y termine.

9.2. Salida de datos por la pantalla

Ahora vamos a estudiar el control de la salida de datos por la pantalla del ordenador a través de la consola del simulador XSPIM.

Crea un fichero con el siguiente código:

```
          .data 0x10010000
cadena:   .asciiz  "¡Lo ves como ahora si que sale bien!"
          .data 0xffff0008
cout:     .space 1      # Puerto de control de la consola
          .data 0xffff000c
out:      .space 1  # Puerto de escritura en la consola
          .text
          .globl main
main:     la $a0,cadena       #carga dir cadena
          la $a1,out          #carga dir salida
          lb $s0,0($a0)       #lee el dato de la cadena
          la $a2,cout         #carga dir control de salida
     ############# bucle de consulta de estado ##########
ctr1:
     #####################################################
          sb $s0, 0($a1)# escribe consola
          addi $a0,$a0,1#incremento puntero de cadena
          lb $s0, 0($a0)#lee el siguiente dato
          bne $s0,$0,ctr1 # control de fin de cadena (0)
     ############ bucle de consulta de estado ##########
ctr2:
     #####################################################
          sb $0, 0($a1) #escribe en consola fin cadena (0)
```

```
        jr $ra    # vuelve a la rutina principal
```

Descripción:

El código anterior lee los caracteres almacenados en la memoria a partir de la dirección `cadena` y los escribe en el puerto de la salida de datos por la consola del simulador `out`. Sin embargo, no se realiza correctamente el control de la salida de datos mediante consulta de estado que, al igual que en apartado anterior, se ha quitado a propósito.

Ejecuta el programa editado en el simulador.

Como puedes comprobar no aparece en la consola la frase almacenada en memoria.

Introduce ahora en las etiquetas `ctr1` y `ctr2` el siguiente código:

```
############### bucle de retardo ##############

      li $t0,n       # n debe sustituirse por un valor

cont: addi $t0,$t0,-1

      bnez $t0,cont

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

Sustituye inicialmente el valor de "n" por 10 y ejecuta el programa en el simulador.

Cuestión 9.5: Ahora es posible que aparezcan más caracteres de la cadena en la pantalla ¿Los caracteres que aparecen siguen algún patrón? Fíjate en su posición en la cadena.

Cuestión 9.6: Incrementa el valor de "n" de forma progresiva y comprueba que cada vez aparecen más caracteres de la cadena en la consola ¿A partir de qué valor aparece la cadena completa?

Cuestión 9.7: ¿Qué estamos haciendo al incrementar el valor de "n"?

Cuestión 9.8: ¿El valor de "n" obtenido tendría que ser el mismo si ejecutáramos el programa en otro ordenador? ¿Cuál es la razón?

Cuestión 9.9: Siguiendo el ejemplo del apartado anterior realiza el control de la salida de datos por la consola de este programa mediante consulta de estado.

9.3. Entrada/Salida de datos

El siguiente código permite leer los caracteres pulsados desde el teclado (hasta un total de 10) y mostrarlos por la consola:

```
        .data
long:   .word   10  #tamaño del buffer
        .data 0xffff0000
cin:    .space 1
        .data 0xffff0004
in:     .space 1
        .data 0xffff0008
cout:   .space 1
        .data 0xffff000c
out:    .space 1
        .text
main:   addi $sp,$sp,-4
        sw $ra,0($sp)
        lw $s0,long
        addi $s0,$s0,-1
        li $t6,0x0a   # return
        la $t1, in    #carga dir in
        la $t2, cin   #carga dir control teclado
        la $t3, out   #carga dir out
        la $t4, cout  #carga dir control de salida
ctri:   jal wi
        lb $t7, 0($t1)# leo de teclado
        jal wo
        sb $t7, 0($t3)# escribo consola
        addi $s0,$s0,-1
        beq $t7,$t6, fin #control cambio de línea
        bne $s0,$0, ctri # control de fin de cadena
```

```
            j cero
fin:        li $t7, 0x0a
            jal wo
            sb $t7, 0($t3)# escribo consola
cero:       andi $t7,$t7,0
            jal wo
            sb $t7, 0($t3)# escribo consola
            lw $ra, 0($sp)
            addi $sp,$sp,4
            jr $ra
############ Consulta de estado de entrada #########
wi:
####################################################
############ Consulta de estado de salida ##########
wo:
####################################################
```

Sin embargo, se han omitido, a propósito, las rutinas `wi` y `wo` que realizan el control de consulta de estado de los periféricos.

Cuestión 9.10: Basándote en los ejemplos anteriores, completa el programa anterior con las rutinas de control de los periféricos (wi y wo) y comprueba su funcionamiento en el simulador.

Cuestión 9.11: Modifica el código del programa anterior para utilizar sólo una rutina de consulta de estado y que, pasándole el parámetro del puerto de control del periférico, sirva para realizar tanto la consulta de estado de la entrada como de la salida de datos. Comprueba su funcionamiento en el simulador.

Problemas propuestos

Nota: para estos ejercicios no se debe utilizar la llamada "syscall".

1. Implementa una subrutina en ensamblador que, dado un entero en memoria, muestre su valor por la consola:

a) En decimal.

b) En hexadecimal.

2. Implementa una subrutina que pida un entero por el teclado y lo guarde en memoria.

3. Implementa un programa que, dada una dirección de memoria, muestre por la consola los diez bytes siguientes en hexadecimal y separados por guiones

CAPÍTULO 10. GESTIÓN DE E/S MEDIANTE INTERRUPCIONES

En este capítulo vamos a estudiar la gestión del subsistema de E/S de un computador utilizando interrupciones. Esta técnica, que se usa en la práctica totalidad de los computadores para gestionar algunos periféricos, emplea las interrupciones para indicar al procesador que el dispositivo de E/S necesita ser atendido. Cuando un dispositivo quiere notificar al procesador que ha terminado una operación de E/S, o que necesita su atención, solicita una interrupción. Se trata de otra forma de adaptar, como en el caso de consulta de estado, la velocidad del procesador a la del periférico, sin que sea necesario consumir ciclos de reloj de la CPU durante la espera. Utilizando las interrupciones se consigue un mayor nivel de paralelismo entre el trabajo del procesador y el dispositivo de E/S.

10.1 Procesamiento de las excepciones en el simulador SPIM del MIPS

En esta práctica se describe el procesamiento de interrupciones y excepciones que realiza el simulador SPIM, que es una parte de lo que hace el procesador MIPS. En los procesadores MIPS, una parte de la CPU llamada coprocesador 0 mantiene la información que necesitan los programas para tratar excepciones e interrupciones. El SPIM proporciona los siguientes registros del coprocesador 0:

Nombre del Registro	Número de registro	Utilización
Estado	12	Máscara de interrupciones y bits de autorización
Causa	13	Tipo de excepción y bits de interrupción pendientes
EPC	14	Registro que contiene la dirección de la instrucción que ha causado la excepción

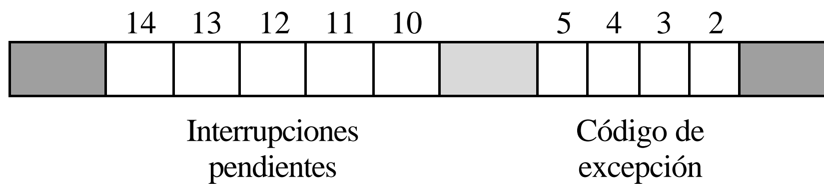
Estos tres registros son parte de los registros del coprocesador 0 y se acceden mediante las instrucciones `mfc0` y `mtc0`, que mueven la información desde estos registros a otros del procesador (o viceversa). Veamos la información que contiene cada uno de ellos.

- Registro EPC (Exception Program Counter)

En este registro el procesador almacena la dirección de la instrucción que se estaba ejecutando en el momento de producirse la excepción.

- Registro Causa

La siguiente figura enumera los bits que se utilizan en este registro.



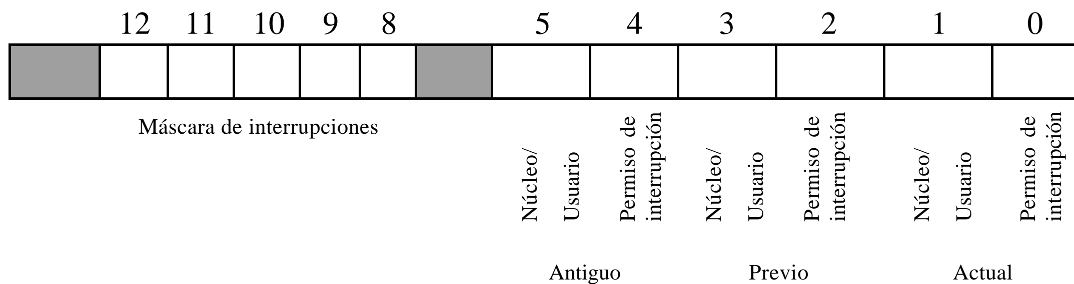
En el campo *código de excepción* el procesador almacena la causa de la excepción con los siguientes códigos:

Número	**Nombre**	**Descripción**
0	INT	Interrupción externa
4	ADDRL	Excepción de dirección errónea (load o fectch de una instrucción)
5	ADDRS	Excepción de dirección errónea (store)
6	IBUS	Error de bus en fetch de instrucción
7	DBUS	Error de bus en load o store de datos
8	SYSCALL	Excepción de llamada al sistema
9	BKPT	Excepción de punto de parada
10	RI	Excepción de instrucción reservada
12	OVF	Excepción de desbordamiento aritmético

Si la causa que ha provocado la excepción es una interrupción, en el campo *interrupciones pendientes* del registro de estado se almacena el nivel de interrupción que ha provocado dicha petición. En total hay 5 niveles de interrupción, cada uno tiene asociado un bit, que pasará a 1 cuando se ha producido una petición en su nivel y no ha sido atendida.

- Registro Estado

Esta figura enumera los campos del registro Estado que ofrece el simulador SPIM de MIPS.



El campo *Máscara de interrupciones* contiene un bit para cada uno de los niveles de interrupción externas posibles (bit 8 para el nivel 0, bit 9 para el nivel 1, etc). Un 1 en este bit permite interrupción a este nivel. A 0 la enmascara. Los 6 bits de menor peso del registro Estado forman una pila de profundidad tres para los bits *Núcleo/Usuario* y *Permiso de interrupción*. El bit *Núcleo/Usuario* vale 0 si el programa estaba en el núcleo del Sistema Operativo cuando se ha producido la excepción y 1 si estaba ejecutando en modo usuario. Si el bit de *Permiso de interrupción* está a 1 las interrupciones externas están habilitadas y si está a 0 están enmascaradas. Cuando se produce una excepción estos 6 bits se desplazan 2 posiciones hacia la izquierda, de modo que los bits *Actuales* pasan a almacenarse en la posición de los bits *previos* y estos a la posición de los *Antiguos*. Los bits *antiguos* se pierden. Los bits actuales se ponen ambos a 0, de modo que la excepción se ejecuta en modo *Kernel* con las interrupciones enmascaradas.

Cuando se produce una excepción, el procesador lleva a cabo las siguientes acciones (acciones hardware):

1. Salvar la dirección de la instrucción causante de la excepción en el registro EPC.

2. Almacenar el código de la excepción producida en el registro Causa. Cuando es una interrupción se activa el bit que indica el nivel de la petición.

3. Salvar el registro de Estado en el propio registro de estado, desplazando la información de los 6 primeros bits 2 posiciones hacia la izquierda, y poner los dos primeros bits a 0, que indican que se pasa a modo *kernel* y que las interrupciones están enmascaradas.

4. Cargar en el contador de programa una dirección de memoria fija (800000080 en el espacio de direcciones del núcleo). Esta es la dirección de memoria a partir de la que está almacenada la rutina de servicio general de atención a la excepción (*interrupt handler*, que está en el fichero **/usr/lib/spim/trap.handler**). Esta rutina determina la causa de la excepción y salta al lugar adecuado del Sistema Operativo para tratar la excepción. La rutina de servicio de atención a la excepción responde a una excepción o bien terminando el proceso que la ha causado o bien realizando alguna acción.

10.2 Descripción del controlador del teclado simulado por SPIM

El controlador del teclado que simula SPIM dispone de dos registro control y datos. El registro de control está en la posición 0xffff0000 y sólo se usan dos de sus bits. El bit 0 es el *bit de estado* y ya se vio su funcionamiento en la práctica anterior. El bit 1 es el *bit de permiso de interrupción*. A 1 indica que la interrupción de este dispositivo está habilitada. A 0 está enmascarada. Este bit se puede modificar por programa. Si este bit está a 1, este dispositivo solicitará interrupción de nivel 0 al procesador cuando el *bit de estado* también esté a 1. De todos modos, para que el procesador atienda esta interrupción, las interrupciones deben estar habilitadas también en el registro de estado del procesador (bit 0 y 8 a 1). El registro de datos se localiza en la posición 0xffff0004.

10.3 Control de Entrada de datos mediante interrupciones

```
            .data 0x10010000
perm:       .word 0x00000101
caracter:   .asciiz     "a"
            .data 0xffff0000
cin:        .space 1          #control de lectura
            .data 0xffff0004
in:         .space 1          # puerto de lectura
            .data 0xffff0008
cout:       .space 1          #control de escritura
            .data 0xffff000c
out:        .space 1          #puerto de escritura
#######zona de datos del núcleo del sistema operativo###########
            .kdata
inter:      .asciiz     " int "
##mensaje que aparece cuando el procesador atiende la interrupción##
__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] "
            .align 2
__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
       .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)

##### código de inicialización necesario para que la entrada de datos
##### se gestione mediante interrupciones
      .globl main
main: addi $sp,$sp,-4
      sw $ra,0($sp)
   ######## habilitación de las interrupciones del teclado ########
      lw $t0,cin($0)
      ori $t0,$t0,2
      sw $t0,cin($0)
######### habilitación de interrupciones de nivel cero ########
      mfc0 $t0,$12
      lw $t1,perm($0)
      or $t0,$t0,$t1
```

```
      mtc0 $t0,$12
    ############ bucle infinito que imprime el carácter a #########
      addi $s0,$0,100
haz:  la $a0,caracter
      li $v0,4
      syscall
      addi $t1,$0,10000
wat:  addi $t1,$t1,-1
      bnez $t1,wat
      addi $s0,$s0,-1
      bnez $s0, haz

      lw $ra, 0($sp)
      addi $sp,$sp,4
      jr $ra
###### rutina de servicio general de excepciones almacenada en el
###### núcleo del sistema operativo
      .ktext 0x80000080
      .set noat
       ########### Utilización de k0 y k1 sin salvarlo ########
      move $k1 $at      # Salva $at
      .set at
## Salvar v0 y a0 en dos posiciones de memoria reservadas para ello
##### No se puede utilizar la pila, por si la excepción lo hubiese
##### provocado algún acceso a memoria referente a la pila
      sw $v0 s1   # No reentrant. No podemos confiar en el $sp
      sw $a0 s2
########### Comprobación de la causa de la excepción ##############
      mfc0 $k0 $13      # lee reg. Cause y lo guarda en $k0
###Si es una interrupción salta a la rutina de servicio de int. ####
      sgt $v0 $k0 0x44 # comprobación si es una interrupción externa
      bgtz $v0 ext
##### Comprobación del resto de causas de excepción y saltar a su
##### tratamiento
      addu $0 $0 0
      i $v0 4     # syscall 4 (print_str)
      la $a0 __m1_
      syscall
      li $v0 1    # syscall 1 (print_int)
      srl $a0 $k0 2     # desplaza el registro Cause
      syscall
```

```
        li $v0 4    # syscall 4 (print_str)
        lw $a0 __excp($k0)
        syscall
        bne $k0 0x18 ok_pc # PC erróneo requiere de chequeo especial
        mfc0 $a0, $14     # EPC
        and $a0, $a0, 0x3 # Está la dir. de EPC alineada?
        beq $a0, 0, ok_pc
        li $v0 10   # Salir si el PC es erroneo(fuera de text)
        syscall
ok_pc:
        li $v0 4    # syscall 4 (print_str)
        la $a0 __m2_
        syscall
        mtc0 $0, $13      # Limpiar el registro Cause
        bnez $v0, ret
###################rutina de servicio de interrupción########3
ext:    li $v0 4    # interrupción externa
        la $a0 inter
        syscall
        mtc0 $0, $13      # Limpiar el registro Cause
###retorno del tratamiento de una excepción a la siguiente
###instrucción interrumpida
ret:    lw $v0 s1
        lw $a0 s2
        mfc0 $k0 $14      # EPC
        .set noat
        move $at $k1      # Restaurar el valor de $at
        .set at
        rfe         # Recuperar estado
        addiu $k0 $k0 4 # Volver a la próxima instrucción
        jr $k0
```

Descripción

El programa anterior incluye, por un lado, el código necesario para que la entrada de datos se pueda gestionar mediante interrupciones y, por otro, las acciones que se deben tomar para cada una de las excepciones que se producen en el sistema. Fíjate en las intrucciones marcadas en negrita.

Escribe en un fichero el programa anterior a partir del fichero que contiene el código estándar para el manejo de las interrupciones (*trap.handler*). Ejecútalo en el simulador XSPIM lanzado con las opciones **–mapped_io** y **–notrap**.

Cuestión 10.1: Describe qué ocurre durante la ejecución del programa cuando se pulsa una tecla.

Cuestión 10.2: Indica qué acciones deben llevarse a cabo tanto en el procesador como en el controlador del teclado para que la entrada de datos se pueda gestionar mediante interrupciones.

Cuestión 10.3: Indica qué acciones ejecuta la rutina de servicio de atención a la interrupción

Cuestión 10.4: Describe qué ocurre si se mantiene la misma tecla pulsada durante un tiempo continuado.

Problemas propuestos

1. Modifica la rutina de servicio de la interrupción del apartado 10.3 para que además también se imprima el número interrupciones atendidas hasta este momento.

2. Modifica la rutina de servicio de la interrupción del apartado 10.3 para que se imprima la tecla que se ha pulsado.

3. Modifica la rutina de servicio de la interrupción del apartado 10.3 para que cuando se pulse una determinada tecla (por ejemplo 'f') termine la ejecución del programa.