Prácticas de Introducción a la Arquitectura de Computadores

con el simulador SPIM

Sergio Barrachina Mir
Maribel Castillo Catalán
Jose M. Claver Iborra
Juan C. Fernández Fdez

PRÓLOGO

Este libro de prácticas está dirigido a estudiantes de primeros cursos de Ingenierías Técnicas Informáticas e Ingenierías Informáticas que cursen asignaturas de introducción a la Arquitectura de Computadores, y en general, a aquellos lectores que deseen aprender Arquitectura de Computadores por medio de la realización de ejercicios con lenguaje ensamblador.

Pese a que el hilo conductor del libro es la programación en lenguaje ensamblador, el lector no debe confundirlo con un curso de programación en ensamblador. El objetivo de este libro no es el de enseñar a programar en ensamblador, sino el de ayudar a que el lector asimile y comprenda, por medio de la programación en ensamblador, los conceptos fundamentales de Arquitectura de Computadores. En particular, se abordan los siguientes temas: representación de la información, tipos de datos, el juego de instrucciones, registros, organización de la memoria, programación de computadores a bajo nivel, relación entre la programación en un lenguaje de alto nivel y el funcionamiento del procesador y tratamiento de interrupciones y excepciones.

Puesto que se pretende favorecer la comprensión de dichos conceptos por medio de la realización de prácticas en ensamblador, cobra especial importancia la elección del procesador que será objeto de estudio. Analizando qué procesadores han sido escogidos, a lo largo de los años, por las asignaturas de introducción a la Arquitectura de Computadores de distintas Universidades, se puede ver que han sido varios los elegidos y que la elección de uno u otro ha estado fuertemente condicionado por las modas.

En cualquier caso, es posible constatar que aquellas elecciones que han contado con mayor aceptación, y que han permanecido por más tiempo, han estado siempre relacionadas con la aparición de procesadores que, por su concepción y diseño, han revolucionado el campo de los computadores. Cabe destacar, entre estos procesadores, los siguientes: el PDP 11 de DEC, el MC68000 de Motorola, el 8088/86 de Intel y el R2000/3000 de MIPS.

El procesador MC68000 de Motorola ha sido ampliamente utilizado, y lo es aún hoy en día, en las prácticas de las asignaturas de Arquitectura de Computadores de muchas Universidades. Lo mismo se puede decir del procesador

8086 de Intel. Aunque este último en menor medida. El procesador de Motorola tiene a su favor la ortogonalidad de su juego de instrucciones y la existencia de diferentes modos prioritarios de funcionamiento. En el caso de Intel, el motivo determinante para su adopción ha sido sin duda, la amplia difusión de los computadores personales (PCs) de IBM y compatibles.

Sin embargo, el procesador más extendido en la actualidad en el ámbito de la enseñanza de Arquitectura de Computadores, y el que se propone en este libro, es el MIPS32[1]. Dicho procesador, por un lado, mantiene la simplicidad de los primeros procesadores RISC y, por otro, constituye la semilla de muchos de los diseños de procesadores superescalares actuales.

Debido a la simplicidad del juego de instrucciones del MIPS32, es relativamente fácil desarrollar pequeños programas en ensamblador y observar el efecto de su ejecución. Por otro lado, al ser su arquitectura la semilla de muchos de los diseños de procesadores superescalares actuales, es fácil extender los conocimientos adquiridos en su estudio a arquitecturas más avanzadas.

Dicho de otra forma, lo que hace idóneo al MIPS32 para utilizarlo en la enseñanza de Arquitectura de Computadores es que se trata de un procesador real pero lo suficientemente sencillo como para estudiarlo sin demasiadas complicaciones, y a la vez, puede servir de base para el estudio de arquitecturas más avanzadas.

Otro punto a favor de la utilización del MIPS32, es la existencia de simuladores que permiten comprobar cómodamente su funcionamiento, sin la necesidad de tener acceso a un computador real basado en dicho procesador. En este libro se propone el simulador SPIM de James Larus [Lar]. Puede obtenerse de forma gratuita desde la página web del autor y está disponible tanto para Windows como para GNU/Linux.

En cuanto a la organización del presente libro, simplemente decir que se ha hecho un esfuerzo importante para que sea, en la medida de lo posible, autocontenido. Así, muchos de los conceptos teóricos necesarios para realizar los ejercicios que se proponen, se introducen conforme son requeridos.

En cualquier caso, no hay que perder de vista que el libro se ofrece como complemento a una formación teórica en Arquitectura de Computadores y, por tanto, gran parte de los conceptos generales o más específicos se han dejado forzosamente fuera. Nuestra intención no va más allá de la de ofrecer un complemento eminentemente práctico a una formación teórica en Arquitectura de Computadores.

[1] En su versión no segmentada y sin instrucciones retardadas de carga, almacenamiento y salto.

En cuanto a la formación teórica en Arquitectura de Computadores, consideramos que como libro de referencia se podría utilizar el libro «*Estructura y diseño de computadores: interficie circuitería/programación*», de David A. Patterson y John L. Hennesy [Pat00], en el que también se estudia el procesador MIPS32.

De esta forma, un curso introductorio a la Arquitectura de Computadores podría utilizar el libro «*Estructura y diseño de computadores: interficie circuitería/programación*», de David A. Patterson y John L. Hennesy [Pat00] como referencia de la parte de teórica de la asignatura y el presente libro para la realización de las prácticas de laboratorio de dicha asignatura.

Convenios tipográficos

Se presentan a continuación los convenios tipográficos seguidos en la redacción de este libro.

Para diferenciar un número del texto que lo rodea, el número se representa utilizando una fuente mono espaciada, como por ejemplo en: `1024`.

Para expresar un número en hexadecimal, éste se precede del prefijo «`0x`», p.e. `0x4A`. Si el número expresado en hexadecimal corresponde a una palabra de 32 bits, éste aparece de la siguiente forma: `0x0040 0024`. Es decir, con un pequeño espacio en medio, para facilitar su lectura.

Para expresar un número en binario se utiliza el subíndice «$_2$», p.e. 0010_2. Si el número expresado en binario corresponde a una palabra de 32 bits, se representa con un pequeño espacio cada 8 bits, para facilitar su lectura, p.e.: $00000001\ 00100011\ 01000101\ 01100111_2$.

Para diferenciar una sentencia en ensamblador del texto que la rodea, se utiliza el siguiente el siguiente formato: «**la** $a0, texto». Cuando se hace referencia a un registro, éste se marca de la siguiente forma: `$s0`.

Los códigos o fragmentos de código se representan de la siguiente forma:

hola-mundo.s

```
1           .data                   # Zona de datos
2  texto:   .asciiz "¡Hola mundo!"
3
4           .text                   # Zona de instrucciones
5  main:    li $v0, 4               # Llamada al sistema para print_str
6           la $a0, texto           # Dirección de la cadena
7           syscall                 # Muestra la cadena en pantalla
```

En la parte superior derecha se muestra el nombre del fichero asociado a dicho código. En el margen izquierdo se muestra la numeración correspondiente a cada una de las líneas del código representado. Esta numeración tiene por objeto facilitar la referencia a líneas concretas del listado y, naturalmente,

no forma parte del código en ensamblador. Con el fin de facilitar la lectura del código, se utilizan diferentes formatos para representar las palabras reservadas y los comentarios. Por último, y para evitar confusiones, los espacios en blanco en las cadenas de texto se representan mediante el carácter «␣».

En cuanto a los bloques de ejercicios, éstos aparecen delimitados entre dos líneas horizontales punteadas. Cada uno de los ejercicios del libro posee un identificador único. De esta forma, puede utilizarse dicho número para referenciar un ejercicio en particular. A continuación se muestra un ejemplo de bloque de ejercicios:

EJERCICIOS

▶ **1** Localiza la cadena «"Hola␣mundo"» en el programa anterior.

▶ **2** Localiza el comentario «*# Zona de datos*» en el programa anterior.

Agradecimientos

Este texto no es obra únicamente de sus autores. Es fruto de la experiencia docente del profesorado involucrado en las asignaturas de «Introducción a los Computadores», «Arquitectura de Computadores» y «Estructura y Tecnología de Computadores» de las titulaciones de Ingeniería Informática, Ingeniería Técnica de Sistemas e Ingeniería Técnica de Gestión de la Universidad Jaume I de Castellón. En particular, se ha enriquecido especialmente con las aportaciones, comentarios y correcciones de los siguientes profesores del departamento de Ingeniería y Ciencia de los Computadores de la Universidad Jaume I: Eduardo Calpe Marzá, Germán León Navarro, Rafael Mayo Gual, Fernando Ochera Bagan y Ximo Torres Sospedra.

También nos gustaría agradecer a los siguientes estudiantes de la Universidad Jaume I el que revisaran con impagable interés una de las primeras versiones del manuscrito: Nicolás Arias Sidro, José Cerisuelo Vidal, Inés de Jesús Rodríguez, Joaquín Delfín Bachero Sánchez, Mariam Faus Villarrubia, Roberto García Porcellá, Daniel Gimeno Solís y Sergio López Hernández.

Para todos ellos, nuestro más sincero agradecimiento.

Nos gustaría, además, agradecer de antemano la colaboración de cuantos nos hagan llegar aquellas erratas que detecten o las sugerencias que estimen oportunas sobre el contenido de este libro. De esta forma, esperamos poder mejorarlo en futuras ediciones.

ÍNDICE GENERAL

Índice general v

1 Introducción al simulador SPIM **1**
1.1. Descripción del simulador SPIM 2
1.2. Sintaxis del lenguaje ensamblador del MIPS32 13
1.3. Problemas del capítulo 15

2 Datos en memoria **17**
2.1. Declaración de palabras en memoria 17
2.2. Declaración de bytes en memoria 19
2.3. Declaración de cadenas de caracteres 20
2.4. Reserva de espacio en memoria 21
2.5. Alineación de datos en memoria 21
2.6. Problemas del capítulo 22

3 Carga y almacenamiento **25**
3.1. Carga de datos inmediatos (constantes) 26
3.2. Carga de palabras (de memoria a registro) 29
3.3. Carga de bytes (de memoria a registro) 30
3.4. Almacenamiento de palabras (de registro a memoria) 31
3.5. Almacenamiento de bytes (bytes de registro a memoria) 32
3.6. Problemas del capítulo 33

4 Operaciones aritméticas, lógicas y de desplazamiento **35**
4.1. Operaciones aritméticas 36
4.2. Operaciones lógicas 39
4.3. Operaciones de desplazamiento 41
4.4. Problemas del capítulo 43

5 Operaciones de salto condicional y de comparación **45**
5.1. Operaciones de salto condicional 46

5.2. Operaciones de comparación 48
5.3. Evaluación de condiciones 48
5.4. Problemas del capítulo 53

6 Estructuras de control condicionales y de repetición 55
6.1. Estructuras de control condicionales 55
6.2. Estructuras de control repetitivas 60
6.3. Problemas del capítulo 63

7 Introducción a la gestión de subrutinas 65
7.1. Llamada y retorno de una subrutina 66
7.2. Paso de parámetros 70
7.3. Problemas del capítulo 77

8 Gestión de subrutinas 79
8.1. La pila 80
8.2. Bloque de activación de la subrutina 84
8.3. Problemas del capítulo 98

9 Gestión de la entrada/salida mediante consulta de estado 101
9.1. Dispositivos periféricos en el simulador SPIM 104
9.2. Entrada de datos desde el teclado 106
9.3. Salida de datos por pantalla 107
9.4. Entrada/Salida de datos 109
9.5. Problemas del capítulo 110

10 Gestión de la entrada/salida mediante interrupciones 111
10.1. Registros para el tratamiento de las excepciones 113
10.2. Tratamiento de las excepciones 117
10.3. Opciones del programa SPIM para la simulación de interrupciones 124
10.4. Excepciones software 124
10.5. Interrupciones 127
10.6. Problemas del capítulo 141

Bibliografía 143

A Manual de uso del comando xspim 145

B Llamadas al Sistema Operativo 149
B.1. Introducción 149
B.2. Finalización del programa en curso: `exit` 150

B.3. Impresión de un entero: `print_int` 151
B.4. Impresión de una cadena: `print_string` 152
B.5. Lectura de un entero: `read_int` 152
B.6. Lectura de una cadena: `read_string` 153

C Registros del Coprocesador 0 **155**

D Cuadro resumen del juego de instrucciones del MIPS **157**

CAPÍTULO 1

INTRODUCCIÓN AL SIMULADOR SPIM

Índice

1.1. Descripción del simulador SPIM **2**
1.2. Sintaxis del lenguaje ensamblador del MIPS32 . . . . . **13**
1.3. Problemas del capítulo **15**

El objetivo de este capítulo es el de describir el funcionamiento básico del simulador SPIM y la sintaxis de los programas en ensamblador que acepta dicho simulador.

El simulador SPIM está disponible de forma gratuita tanto para GNU/Linux como para Windows. Aunque en este capítulo se va a describir únicamente la versión para GNU/Linux, gran parte de la descripción que se haga del funcionamiento de dicha versión es directamente aplicable a la versión para Windows. Naturalmente, las prácticas propuestas a lo largo del libro se pueden realizar con cualquiera de las versiones.

El capítulo comienza con una breve descripción de la interfaz gráfica del simulador y de la información que éste proporciona. Una vez realizada esta descripción, se presta atención a las operaciones que, de forma más habitual, se realizarán durante el seguimiento de las prácticas propuestas: cargar el código fuente de un programa en el simulador y depurar errores de programación mediante la ejecución fraccionada del programa. El capítulo continúa con una introducción al ensamblador del MIPS32 en la que se comenta la sintaxis básí-

ca de este lenguaje (que se irá ampliando, conforme se necesite, en los siguientes capítulos). Finalmente, presenta un ejemplo sencillo que permitirá al lector practicar y familiarizarse con el funcionamiento del simulador.

Es importante estudiar con detalle el contenido de este capítulo ya que de su comprensión depende en gran medida el mejor aprovechamiento de las prácticas propuestas en lo que resta de libro.

1.1. Descripción del simulador SPIM

El simulador SPIM [Lar] (MIPS, escrito al revés) es un simulador desarrollado por James Larus, capaz de ensamblar y ejecutar programas escritos en lenguaje ensamblador para computadores basados en los procesadores MIPS32. Puede descargarse gratuitamente desde la siguiente página web:
http://www.cs.wisc.edu/~larus/spim.html

En dicha página web también se pueden encontrar las instrucciones de instalación para Windows y GNU/Linux. La instalación de la versión para Windows es bastante sencilla: basta con ejecutar el fichero descargado. La instalación para GNU/Linux es un poco más compleja ya que en la página web sólo está disponible el código fuente del simulador. Si se utiliza una distribución GNU/Linux RedHat, SuSE o similar, puede que sea más sencillo buscar en Internet un paquete `RPM` actualizado del simulador e instalarlo (p.e. con «`rpm -i spim.rpm`»); si, en cambio, se utiliza la distribución Debian o Gentoo, se puede instalar ejecutando «`apt-get install spim`» o «`emerge spim`», respectivamente.

En las siguientes secciones se describe el funcionamiento de la versión GNU/Linux del simulador.

1.1.1. Versión para GNU/Linux: XSPIM

XSPIM, que así se llama la versión gráfica para GNU/Linux, presenta una ventana dividida en cinco paneles (véase la Figura 1.1) que, de arriba a abajo, son:

1. Panel de visualización de registros (*registers' display*): muestra los valores de los registros del procesador MIPS.
2. Panel de botones (*control buttons*): contiene los botones desde los que se gestiona el funcionamiento del simulador.
3. Panel de visualización de código (*text segments*): muestra las instrucciones del programa de usuario y del núcleo del sistema (*kernel*) que se carga automáticamente cuando se inicia XSPIM.

4. Panel de visualización de datos (*data segments*): muestra el contenido de la memoria.

5. Panel de visualización de mensajes: lo utiliza el simulador para informar qué está haciendo y avisar de los errores que ocurran durante el ensamblado o ejecución de un programa.

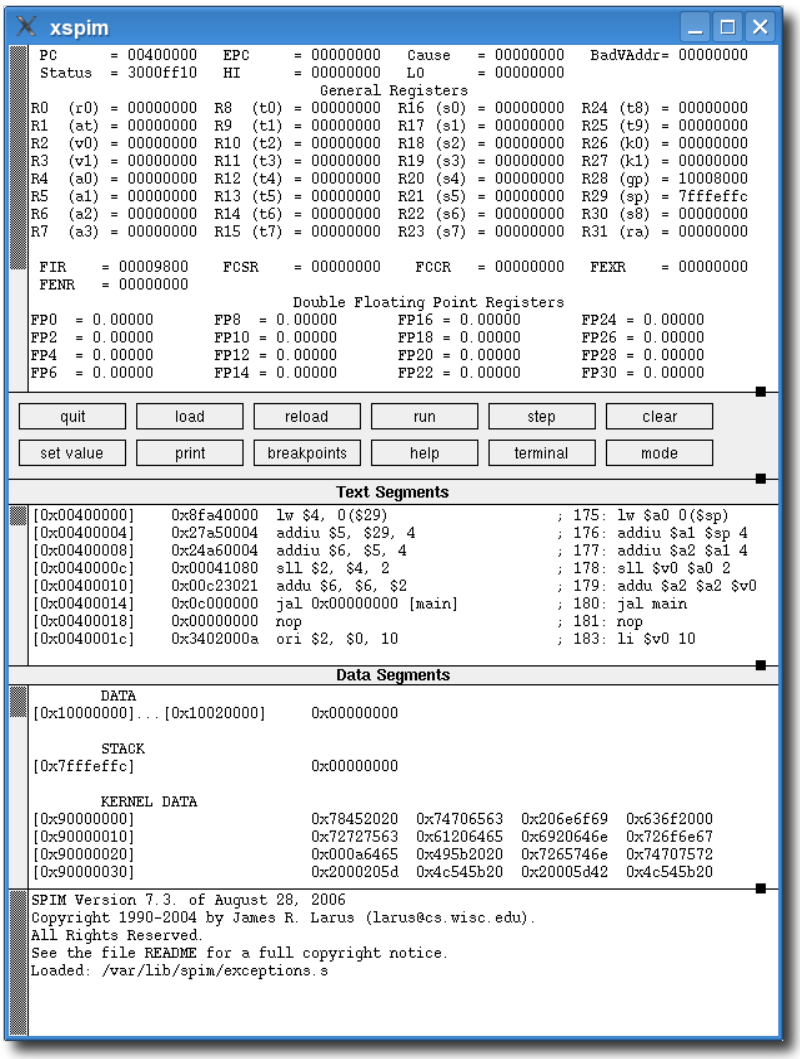


Figura 1.1: Ventana principal del simulador XSPIM

La información presentada en cada uno de estos paneles se describe en las siguientes secciones.

Panel de visualización de registros

En este panel (véase la Figura 1.2) se muestran los diversos registros del MIPS32. Para cada registro se muestra su nombre, su alias entre paréntesis (si es que lo tiene) y su contenido. En concreto, los registros mostrados son:

- Del banco de enteros:
  - Los registros de enteros del `$0` al `$31` con sus respectivos alias entre paréntesis. El simulador los etiqueta con los nombres `R0` al `R31`. Al lado de cada uno de ellos aparece entre paréntesis su alias; que no es más que otro nombre con el que puede referirse al mismo registro. Así, el registro `$29`, el puntero de pila (*stack pointer*), que se identifica mediante la etiqueta `R29` seguida por la etiqueta `sp` entre paréntesis, podrá utilizarse indistintamente como `$29` o `$sp`.
  - El contador de programa: `PC` (*program counter*).
  - Los registros especiales: `HI` (*HIgh*) y `LO` (*LOw*).
- Del banco de reales en coma flotante:
  - Los registros del `$f0` al `$f31` etiquetados con los nombres `FP0` al `FP31`. De éstos, se muestra el valor del número real que almacenan. Puesto que se puede representar un número real en el MIPS32 utilizando los formatos IEEE 754 de simple y doble precisión, el simulador muestra la interpretación del contenido de dichos registros según sea el formato utilizado bajo las leyendas *Single Floating Point Registers* y *Double Floating Point Registers*, respectivamente.
- Del banco para el manejo de excepciones:
  - Los registros `Status`, `EPC`, `Cause` y `BadVAddr`.

Como se puede observar, el contenido de todos los registros, salvo los de coma flotante, se muestra en hexadecimal. Puesto que los registros del MIPS32 son de 4 bytes (32 bytes), se utilizan 8 dígitos hexadecimales para representar su contenido. Recordar que cada dígito hexadecimal corresponde a 4 bits, por tanto, 2 dígitos hexadecimales, constituyen un byte.

Panel de botones

En este panel (véase la Figura 1.3) se encuentran los botones que permiten controlar el funcionamiento del simulador. Pese a que están representados con

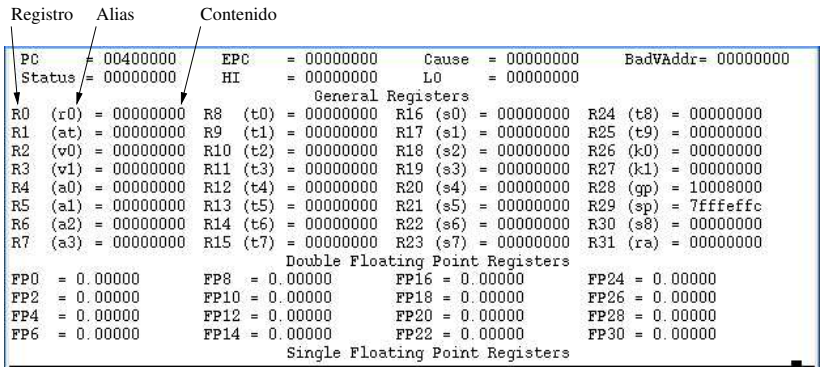


Figura 1.2: Panel de visualización de registros de XSPIM

forma de botones, su funcionamiento, en algunos casos, es más similar al de los elementos de un barra de menús. Son los siguientes:

- `quit` Se utiliza para terminar la sesión del simulador. Cuando se pulsa, aparece un cuadro de diálogo que pregunta si realmente se quiere salir.

- `load` Permite especificar el nombre del fichero que debe ser ensamblado y cargado en memoria.

- `reload` Habiendo especificado previamente el nombre del fichero que debe ensamblarse por medio del botón anterior, se puede utilizar este botón para ensamblar de nuevo dicho fichero. De esta forma, si el programa que se está probando no ensambla o no funciona, puede editarse y recargarse de nuevo sin tener que volver a teclear su nombre.

- `run` Sirve para ejecutar el programa cargado en memoria. Antes de comenzar la ejecución se muestra un cuadro de diálogo en el que se puede especificar la dirección de comienzo de la ejecución. Por defecto, esta dirección es la `0x0040 0000`.

- `step` Este botón permite ejecutar el programa paso a paso. Esta forma de ejecutar los programas es extremadamente útil para la depuración de errores ya que permite observar el efecto de la ejecución de cada una de las instrucciones que forman el programa y de esta forma detectar si el programa está realizando realmente lo que se piensa que debería hacer. Cuando se pulsa el botón aparece un cuadro de diálogo que permite especificar el número de instrucciones que se deben ejecutar en cada paso (por defecto, una), así como interrumpir la ejecución paso a

paso y ejecutar lo que quede de programa de forma continua. Es conveniente, sobre todo al principio, ejecutar los programas instrucción a instrucción para comprender el funcionamiento de cada una de ellas y su contribución al programa. Si uno se limita a pulsar el botón `run` tan sólo verá si el programa ha hecho lo que se esperaba de él o no, pero no comprenderá el proceso seguido para hacerlo.

- `clear` Sirve para restaurar a su valor inicial el contenido de los registros y de la memoria o para limpiar el contenido de la consola. Cuando se pulsa, se despliega un menú que permite indicar si se quiere restaurar sólo los registros, los registros y la memoria, o limpiar el contenido la consola. Restaurar el valor inicial de los registros o de la memoria es útil cuando se ejecuta repetidas veces un mismo programa, ya que en caso contrario, al ejecutar el programa por segunda vez, su funcionamiento podría verse afectado por la modificación durante la ejecución anterior del contenido de ciertos registros o posiciones de memoria.

- `set value` Permite cambiar el contenido de un registro o una posición de memoria.

- `print` Permite mostrar en el panel de mensajes el contenido de un rango de memoria o el valor asociado a las etiquetas globales (*global symbols*).

- `breakpoints` Sirve para introducir o borrar puntos de ruptura o parada (*breakpoints*) en la ejecución de un programa. Cuando se ejecuta un programa y se alcanza un punto de ruptura, la ejecución del programa se detiene y el usuario puede inspeccionar el contenido de los registros y la memoria. Cuando se pulsa este botón aparece un cuadro de diálogo que permite añadir las direcciones de memoria en las que se desea detener la ejecución del programa.

- `help` Imprime en el panel de mensajes una escueta ayuda.

- `terminal` Muestra o esconde la ventana de consola (también llamada terminal). Si el programa de usuario escribe o lee de la consola, todos los caracteres que escriba el programa aparecerán en esta ventana y aquello que se quiera introducir deberá ser tecleado en ella.

- `mode` Permite modificar el modo de funcionamiento de XSPIM. Los modificadores disponibles son: *quiet* y *bare*. El primero de ellos, inhibe la escritura de mensajes en la ventana de mensajes cuando se produce

una excepción. El segundo, *bare*, simula una máquina MIPS sin pseudo-instrucciones ni modos de direccionamiento adicionales. Para la correcta realización de los ejercicios propuestos en este libro ambos modificadores deberán estar desactivados (es la opción por defecto).

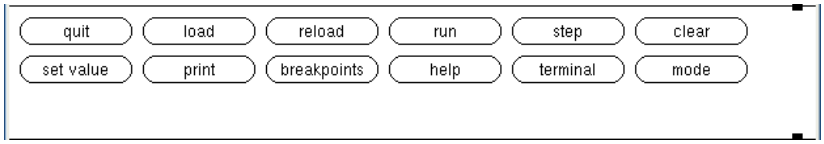


Figura 1.3: Panel de botones de XSPIM

Panel de visualización de código

Este panel (véase la Figura 1.4) muestra el código máquina presente en el simulador. Se puede visualizar el código máquina correspondiente al código de usuario (que comienza en la dirección `0x0040 0000`) y el correspondiente al núcleo (*kernel*) del simulador (que comienza en la dirección `0x8000 0000`).

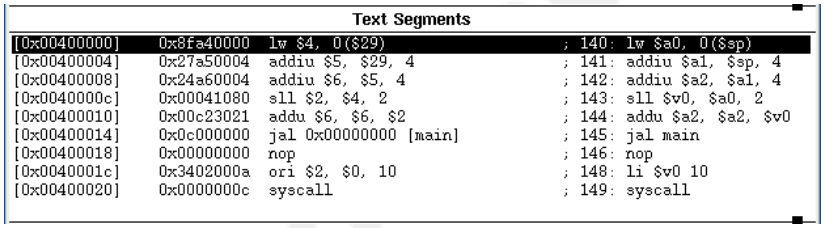


Figura 1.4: Panel de visualización de código de XSPIM

Cada una de las líneas de texto mostradas en este panel se corresponde con una instrucción en lenguaje máquina. La información presentada está organizada en columnas que, de izquierda a derecha, indican:

- la dirección de memoria en la que está almacenada dicha instrucción máquina,
- el contenido en hexadecimal de dicha posición de memoria (o lo que es lo mismo, la representación en ceros y unos de la instrucción máquina),
- la instrucción máquina (en ensamblador), y
- el código fuente en ensamblador desde el que se ha generado dicha instrucción máquina (una línea de ensamblador puede generar más de una instrucción máquina).

Cuando se cargue un programa en el simulador, se verá en la cuarta columna las instrucciones en ensamblador del programa cargado. Cada instrucción estará precedida por un número seguido de «:»; este número indica el número de línea del fichero fuente en la que se encuentra dicha instrucción. Cuando una instrucción en ensamblador produzca más de una instrucción máquina, esta columna estará vacía en las siguientes filas hasta la primera instrucción máquina generada por la siguiente instrucción en ensamblador.

Panel de visualización de datos

En este panel (véase la Figura 1.5) se puede observar el contenido de las siguientes zonas de memoria:

- Datos de usuario (*DATA*): van desde la dirección `0x1000 0000` hasta la `0x1002 0000`.
- Pila (*STACK*): se referencia mediante el registro `$sp`. La pila crece hacia direcciones bajas de memoria comenzando en la dirección de memoria `0x7fff effc`.
- Núcleo del simulador (*KERNEL*): a partir de la dirección `0x9000 0000`.

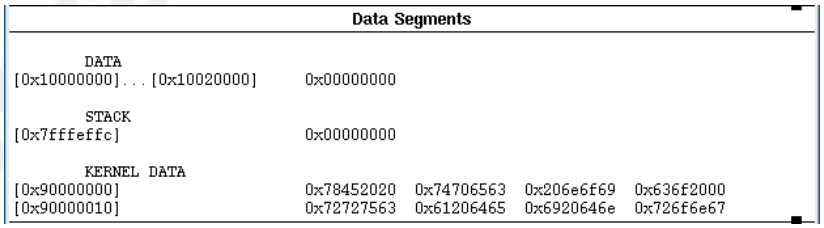


Figura 1.5: Panel de visualización de datos de XSPIM

El contenido de la memoria se muestra de una de las dos formas siguientes:

- Por medio de una única dirección de memoria (entre corchetes) al comienzo de la línea seguida por el contenido de cuatro palabras de memoria: el valor que hay en la posición de memoria indicada y el que hay en las tres posiciones siguientes. Una línea de este tipo presentaría, por ejemplo, la siguiente información:

  ```
  [0x10000000]  0x0000 0000 0x0010 0000 0x0020 0000 0x0030 0000 ,
  ```

  donde «`[0x1000 0000]`» indica la dirección de memoria en la que está almacenada la palabra `0x0000 0000`; las siguientes tres palabras, que en el ejemplo contienen los valores: `0x0010 0000`, `0x0020 0000`

y 0x0030 0000, están en las posiciones de memoria consecutivas a la 0x1000 0000, esto es, en las direcciones de memoria 0x1000 0004, 0x1000 0008 y 0x1000 000c, respectivamente.

- Por medio de un rango de direcciones de memoria (dos direcciones de memoria entre corchetes con «. . . » entre ellas) seguida por el contenido que se repite en cada una de las palabras de dicho rango. Se utiliza este tipo de representación para hacer el volcado de memoria lo más compacto posible. Una línea de este tipo tendría, por ejemplo, la siguiente información:

  ```
  [0x1000 0010]...[0x1002 0000] 0x0000 0000 ,
  ```

  que indica que todas las palabras de memoria desde la dirección de memoria 0x1000 0010 hasta la dirección 0x1002 0000 contienen el valor 0x0000 0000.

Panel de visualización de los mensajes del simulador

Este panel (véase la Figura 1.6) es utilizado por el simulador para mostrar una serie de mensajes que tienen por objeto informar de la evolución y el resultado de las acciones que se estén llevando a cabo en un momento dado.

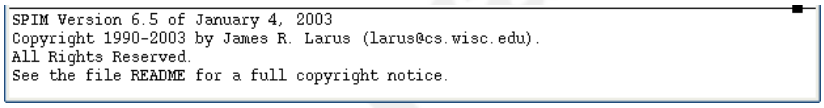


Figura 1.6: Panel de visualización de mensajes de XSPIM

A continuación se muestran algunas situaciones y los mensajes que generan.

Si se carga un programa y durante el proceso de ensamblado se detecta un error, la carga en memoria del programa se interrumpirá y se mostrará un mensaje similar al mostrado en la Figura 1.7.

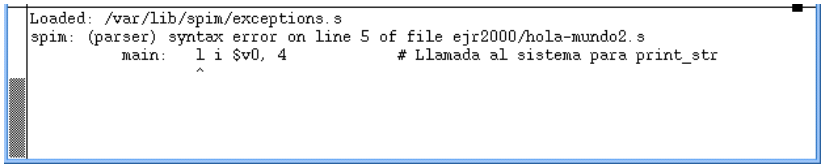


Figura 1.7: Mensaje de error al ensamblar un programa

En el mensaje anterior se puede ver que el simulador informa de que el error ha sido detectado en la línea 5 del fichero «hola-mundo2.s» y que está a

la altura de la letra «l» (gracias al carácter «^» que aparece en la tercera línea). En efecto, en este caso, en lugar de escribir « **li** » habíamos tecleado «l i» por error.

Además, si cuando se está ejecutando un programa se produce un error, se interrumpirá la ejecución del programa y se indicará en este panel la causa del error. Un mensaje de error típico durante la ejecución de un programa sería similar al mostrado en la Figura 1.8. El mensaje de error indica la dirección de memoria de la instrucción máquina que ha provocado el error y el motivo que lo ha provocado. En el ejemplo, la dirección de memoria de la instrucción es la `0x0040 0028` y el error es `Unaligned address in inst/data fetch: 0x10010001`.

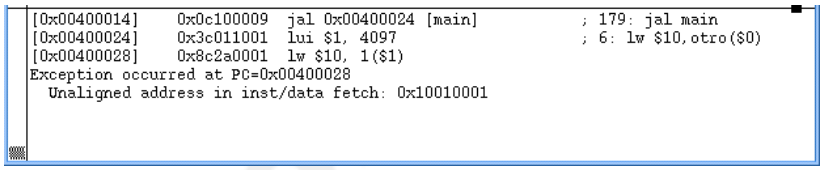


Figura 1.8: Mensaje de error durante la ejecución de un programa

De momento el lector no debe preocuparse si no entiende el significado de lo expuesto en esta sección. Lo realmente importante es que cuando algo falle, ya sea en el ensamblado o en la ejecución, recuerde que debe consultar la información mostrada en este panel para averiguar qué es lo que debe corregir.

1.1.2. Opciones de la línea de comandos de XSPIM

Cuando se ejecuta el simulador XSPIM desde la línea de comandos, es posible pasarle una serie de opciones. De éstas, las más interesantes son:

-bare: Sirve para que la simulación sea la de una máquina pura, es decir, sin que se disponga de las pseudo-instrucciones y modos de direccionamiento aportados por el ensamblador.

-notrap: Se utiliza para evitar que se cargue de forma automática la rutina de captura de interrupciones. Esta rutina tiene dos funciones que deberán ser asumidas por el programa de usuario. En primer lugar, capturar excepciones. Cuando se produce una excepción, XSPIM salta a la posición `0x8000 0180`, donde debería encontrarse el código de servicio de la excepción. En segundo lugar, añadir un código de inicio que llame a la rutina `main`. (Cuando se utiliza esta opción la ejecución comienza en la instrucción etiquetada como « __start » y no en «main» como es habitual).

-mapped_io: Sirve para habilitar los dispositivos de E/S mapeados en memoria. Deberá utilizarse cuando se desee simular la gestión de dispositivos de E/S.

Para cada una de las opciones anteriores existen otras que sirven justamente para lo contrario. Puesto que estas otras son las que están activadas por defecto, se ha considerado oportuno no detallarlas. De todas formas, se puede utilizar el comando `man xspim` para consultar el resto de opciones (en el Apéndice A se muestra la salida de dicho comando).

1.1.3. Carga y ejecución de programas

Los ficheros de entrada de XSPIM son ficheros de texto. Es decir, si se quiere realizar un programa en ensamblador, basta con crear un fichero de texto con un editor de textos cualquiera.

Una vez creado, para cargarlo en el simulador debemos pulsar el botón `load` y, en el cuadro de diálogo que aparezca, especificar su nombre.

Cuando cargamos un programa en el simulador, éste realiza dos acciones: ensambla el código fuente generando código máquina y carga en memoria el código máquina generado. Por regla general, el código máquina generado se cargará en memoria a partir de la dirección `0x0040 0024`. Esto es así ya que, por defecto, el simulador carga de forma automática una rutina de captura de excepciones (ver Sección 1.1.2). Parte de dicha rutina la constituye un código de inicio (*startup code*) encargado de llamar a nuestro programa. Este código de inicio comienza en la dirección `0x0040 0000` y justo detrás de él, en la dirección `0x0040 0024`, se cargará el código máquina correspondiente a nuestro programa.

Si se ha ejecutado XSPIM con la opción **-notrap** (ver Sección 1.1.2) no se cargará el código de inicio comentado anteriormente y el código máquina correspondiente a nuestro programa estará almacenado a partir de la dirección `0x0040 0000`.

Una vez el programa ha sido cargado en el simulador, está listo para su ejecución. Para ejecutar el programa se deberá pulsar el botón `run`. En el cuadro de diálogo que aparece después de pulsar este botón se puede cambiar la dirección de comienzo de la ejecución (en hexadecimal). Normalmente no se deberá cambiar la que aparece por defecto (`0x0040 0000`).

1.1.4. Depuración de programas

Cuando se desarrolla un programa se pueden cometer distintos tipos de errores. El más común de éstos es el cometido al teclear de forma incorrec-

ta alguna parte del código. La mayoría de estos errores son detectados por el ensamblador en el proceso de carga del fichero fuente y el ensamblador avisa de estos errores en el panel de mensajes tal y como se describió en la Sección 1.1.1. Por tanto, son fáciles de corregir. Este tipo de errores reciben el nombre de *errores en tiempo de ensamblado*.

Sin embargo, el que un código sea sintácticamente correcto no garantiza que esté libre de errores. En este caso, los errores no se producirán durante el ensamblado sino durante la ejecución del código. Este tipo de errores reciben el nombre de *errores en tiempo de ejecución*.

En este caso, se tienen dos opciones. Puede que el código máquina realice alguna acción no permitida, como, por ejemplo, acceder a una dirección de memoria no válida. Si es así, esta acción generará una excepción y el simulador avisará de qué instrucción ha generado la excepción y será fácil revisar el código fuente y corregir el fallo. En la Sección 1.1.1 se puede ver la información mostrada en este caso en el panel de mensajes.

Como segunda opción, puede ocurrir que aparentemente no haya errores (porque el fichero fuente se ensambla correctamente y la ejecución no genera ningún problema) y sin embargo el programa no haga lo que se espera de él. Es en estos casos cuando es útil disponer de herramientas que ayuden a depurar el código.

El simulador XSPIM proporciona dos herramientas de depuración: la ejecución paso a paso del código y la utilización de puntos de ruptura (*breakpoints*). Utilizando estas herramientas, se podrá ejecutar el programa por partes y comprobar si cada una de las partes hace lo que se espera de ellas.

La ejecución paso a paso se realiza utilizando el botón `step` (en lugar del botón `run` que provoca una ejecución completa del programa). Cuando se pulsa este botón, aparece un cuadro de diálogo que permite especificar el número de pasos que se deben ejecutar (cada paso corresponde a una instrucción máquina). Si se pulsa el botón `step` (dentro de este cuadro de diálogo) se ejecutarán tantas instrucciones como se hayan indicado.

La otra herramienta disponible permite indicar en qué posiciones de memoria se quiere que se detenga la ejecución de programa. Estas paradas reciben el nombre de puntos de ruptura. Por ejemplo, si se crea un punto de ruptura en la posición de memoria `0x0040 0024` y se pulsa el botón `run`, se ejecutarán las instrucciones máquina desde la dirección de comienzo de la ejecución hasta llegar a la dirección `0x0040 0024`. En ese momento se detendrá la ejecución y la instrucción máquina almacenada en la posición `0x0040 0024` no se ejecutará. Esto permite observar el contenido de la memoria y los registros y se podría comprobar si realmente es el esperado. Cuando se quiera reanudar la ejecución bastará con pulsar de nuevo el botón `run` (o se podría continuar

la ejecución paso a paso utilizando el botón [step]).

Por último, es conveniente que el lector sepa que cada vez que rectifique un fichero fuente tras haber detectado algún error, será necesario volver a cargarlo en el simulador. No basta con corregir el fichero fuente, se debe indicar al simulador que debe volver a cargar dicho fichero. Para hacerlo, basta con pulsar el botón [reload] y seleccionar la entrada «*assembly file*». De todas formas, antes de cargar de nuevo el fichero, es recomendable restaurar el valor inicial de los registros y de la memoria pulsando el botón [clear] y seleccionando la entrada «*memory & registers*». Es decir, una vez editado un fichero fuente en el que se han detectado errores se realizarán las siguientes acciones: [clear] → «*memory & registers*» seguido de [reload] → «*assembly file*».

1.2. Sintaxis del lenguaje ensamblador del MIPS32

Aunque se irán conociendo más detalles sobre la sintaxis del lenguaje ensamblador conforme se vaya siguiendo este libro, es conveniente familiarizarse con algunos conceptos básicos.

El código máquina es el lenguaje que entiende el procesador. Un programa en código máquina, como ya se sabe, no es más que una secuencia de instrucciones máquina, es decir, de instrucciones que forman parte del juego de instrucciones que el procesador es capaz de ejecutar. Cada una de estas instrucciones está representada por medio de ceros y unos en la memoria del computador. (Recuerda que en el caso del MIPS32 cada una de estas instrucciones ocupa exactamente 32 bits de memoria.)

Programar en código máquina, teniendo que codificar cada instrucción mediante su secuencia de ceros y unos correspondiente, es una tarea sumamente ardua y propensa a errores (a menos que el programa tenga tres instrucciones). No es de extrañar que surgieran rápidamente programas capaces de leer instrucciones escritas en un lenguaje más natural y que pudieran ser fácilmente convertidas en instrucciones máquina. Los programas que realizan esta función reciben el nombre de *ensambladores*, y los lenguajes de este tipo el de *lenguajes ensamblador*.

El lenguaje ensamblador ofrece por tanto, una representación más próxima al programador, aunque sin alejarse demasiado del código máquina. Simplifica la lectura y escritura de programas. Proporciona nemónicos (nombres fáciles de recordar para cada una de las instrucciones máquina) y ofrece una serie de recursos adicionales que aumentan la legibilidad de los programas. A continuación se muestran algunos de los recursos de este tipo proporcionados por el ensamblador del MIPS32 junto con su sintaxis:

Comentarios Sirven para dejar por escrito qué es lo que está haciendo alguna parte del programa y para mejorar su legibilidad remarcando las partes que lo forman. Si comentar un programa escrito en un lenguaje de alto nivel se considera una buena práctica de programación, cuando se programa en ensamblador, es obligado utilizar comentarios que permitan seguir el desarrollo del programa. El comienzo de un comentario se indica por medio del carácter almohadilla («#»): el ensamblador ignorará el resto de la línea a partir de la almohadilla.

Pseudo-instrucciones El lenguaje ensamblador proporciona instrucciones adicionales que no pertenecen al juego de instrucciones propias del procesador. El programa ensamblador se encarga de sustituirlas por aquella secuencia de instrucciones máquina que realizan su función. Permiten una programación más clara. Su sintaxis es similar a la de las instrucciones del procesador.

Identificadores Son secuencias de caracteres alfanuméricos, guiones bajos, «_», y puntos, «.», que no comienzan con un número. Las instrucciones y pseudo-instrucciones se consideran palabras reservadas y, por tanto, no pueden ser utilizadas como identificadores. Los identificadores pueden ser:

Etiquetas Se utilizan para posteriormente poder hacer referencia a la posición o dirección de memoria del elemento definido en la línea en la que se encuentran. Para declarar una etiqueta, ésta debe aparecer al comienzo de una línea y terminar con el carácter dos puntos («:»).

Directivas Sirven para informar al ensamblador sobre cómo debe interpretarse el código fuente. Son palabras reservadas, que el ensamblador reconoce. Se identifican fácilmente ya que comienzan con un punto («.»).

Entre los literales que pueden utilizarse en un programa en ensamblador están los números y las cadenas de caracteres. Los números en base 10 se escriben tal cual. Para expresar un valor en hexadecimal, éste deberá estar precedido por los caracteres «`0x`». Las cadenas de caracteres deben encerrarse entre comillas dobles (”). Es posible utilizar caracteres especiales en las cadenas siguiendo la convención empleada por el lenguaje de programación C:

- Salto de línea: `\n`
- Tabulador: `\t`

- Comilla doble: \"

Errores comunes

Cuando se escribe el código fuente correspondiente a un programa en ensamblador se debe prestar especial atención a los siguientes puntos:

- El fichero fuente habrá de terminar con una línea en blanco. No puede haber ninguna instrucción en la última línea del fichero ya que XSPIM no la ensamblará correctamente.
- El programa debe tener una etiqueta «main» que indique cuál es la primera instrucción que debe ser ejecutada. En caso contrario, cuando se pulse el botón `run`, y el código de inicio llegue a la instrucción «**jal** main», se detendrá la ejecución.

1.3. Problemas del capítulo

EJERCICIOS

▶ 1 Dado el siguiente ejemplo de programa ensamblador, identifica y señala las etiquetas, directivas y comentarios que aparecen en él.

introsim.s

```
1         .data
2 dato:   .word 3          # Inicializa una palabra con el valor 3
3
4         .text
5 main:   lw $t0, dato($0) # Carga el contenido de M[dato] en $t0
```

▶ 2 Crea un fichero con el programa anterior, cárgalo en el simulador y responde a las siguientes preguntas: ¿en qué dirección de memoria se ha almacenado el 3?, ¿en qué dirección de memoria se ha almacenado la instrucción «**lw** $t0, dato($0)»?, ¿qué registro, del `$0` al `$31`, es el registro `$t0`?

▶ 3 Ejecuta el programa anterior, ¿qué valor tiene el registro `$t0` después de ejecutar el programa?

CAPÍTULO 2

DATOS EN MEMORIA

Índice

2.1. Declaración de palabras en memoria **17**
2.2. Declaración de bytes en memoria **19**
2.3. Declaración de cadenas de caracteres **20**
2.4. Reserva de espacio en memoria **21**
2.5. Alineación de datos en memoria **21**
2.6. Problemas del capítulo **22**

Prácticamente cualquier programa de computador requiere de datos para llevar a cabo su tarea. Por regla general, estos datos son almacenados en la memoria del computador.

Al programar en un lenguaje de alto nivel se utilizan variables de diversos tipos. Es el compilador (o el intérprete según sea el caso) quien se encarga de decidir en qué posiciones de memoria se almacenarán las estructuras de datos requeridas por el programa.

En este capítulo se verá cómo indicar qué posiciones de memoria se deben utilizar para las variables de un programa y cómo se pueden inicializar dichas posiciones de memoria con un determinado valor.

2.1. Declaración de palabras en memoria

En este apartado se verán las directivas «**.data**» y «**.word**». Como punto de partida se considera el siguiente ejemplo:

Bytes, palabras y medias palabras

Los computadores basados en el procesador MIPS32 pueden acceder a la memoria a nivel de byte. Esto es, cada dirección de memoria indica la posición de memoria ocupada por un byte.

Algunos tipos de datos, por ejemplo los caracteres ASCII, no requieren más que un byte por dato. Sin embargo, la capacidad de expresión de un byte es bastante reducida (p.e. si se quisiera trabajar con números enteros habría que contentarse con los números del -128 al 127). Por ello, la mayoría de computadores trabajan de forma habitual con unidades superiores al byte. Esta unidad superior suele recibir el nombre de *palabra* (*word*).

En el caso del MIPS32, una *palabra* equivale a 4 bytes. Todo el diseño del procesador tiene en cuenta este tamaño de palabra: entre otros, los registros tienen un tamaño de 4 bytes y el bus de datos consta de 32 líneas.

Para aumentar el rendimiento del procesador facilitando la transferencia de información entre el procesador y la memoria, la arquitectura del MIPS32 impone una restricción sobre qué direcciones de memoria pueden ser utilizadas para acceder a una palabra: deben ser múltiplos de 4. Es decir, para poder leer o escribir una palabra en memoria, su dirección de memoria deberá ser múltiplo de 4.

Por último, además de acceder a bytes y a palabras, también es posible acceder a medias palabras (*half-words*), que están formadas por 2 bytes. De forma similar a lo comentado para las palabras, la dirección de memoria de una media palabra debe ser múltiplo de 2.

✎ datos-palabras.s

```
1           .data       # comienzo de la zona de datos
2 palabra1: .word 15    # representación decimal del dato
3 palabra2: .word 0x15 # representación hexadecimal del dato
```

El anterior ejemplo no acaba de ser realmente un programa ya que no contiene instrucciones en lenguaje ensamblador que deban ser ejecutadas por el procesador. Sin embargo, utiliza una serie de directivas que le indican al ensamblador (a SPIM) qué información debe almacenar en memoria y dónde.

La primera de las directivas utilizadas, «**.data**», se utiliza para avisar al ensamblador de que todo lo que aparezca debajo de ella (mientras no se diga lo contrario) debe ser almacenado en la zona de datos y la dirección en la que deben comenzar a almacenarse. Cuando se utiliza la directiva «**.data**» sin argumentos (tal y como está en el ejemplo) se utilizará como dirección de comienzo de los datos el valor por defecto `0x1001 0000`. Para indicar otra dirección de comienzo de los datos se debe utilizar la directiva en la forma

«**.data** DIR». Por ejemplo, si se quisiera que los datos comenzaran en la posición `0x1001 0020`, se debería utilizar la directiva «**.data** 0x10010020».

Volviendo al programa anterior, las dos siguientes líneas utilizan la directiva «**.word**». Esta directiva sirve para almacenar una palabra en memoria. La primera de las dos, la «**.word** 15», almacenará el número 15 en la posición `0x1001 0000` (por ser la primera después de la directiva «**.data**»). La siguiente, la «**.word** 0x15» almacenará el valor `0x15` en la siguiente posición de memoria no ocupada.

Crea el fichero anterior, cárgalo en el simulador y resuelve los siguientes ejercicios.

EJERCICIOS

▶ **4** Encuentra los datos almacenados en memoria por el programa anterior: localiza dichos datos en la zona de visualización de datos e indica su valor en hexadecimal.

▶ **5** ¿En qué direcciones se han almacenado las dos palabras? ¿Por qué?

▶ **6** ¿Qué valores toman las etiquetas «palabra1» y «palabra2»?

▶ **7** Crea ahora otro fichero con el siguiente código:

✎ datos-palabras2.s

```
1           .data 0x10010000 # comienzo de la zona de datos
2 palabras: .word 15, 0x15   # en decimal y en hexadecimal
```

Borra los valores de la memoria utilizando el botón `clear` y carga el nuevo fichero. ¿Se observa alguna diferencia en los valores almacenados en memoria respecto a los almacenados por el programa anterior? ¿Están en el mismo sitio?

▶ **8** Crea un programa en ensamblador que defina un vector de cinco palabras (*words*), asociado a la etiqueta vector, que comience en la dirección `0x1000 0000` y que tenga los siguientes valores `0x10`, `30`, `0x34`, `0x20` y `60`. Cárgalo en el simulador y comprueba que se ha almacenado de forma correcta en memoria.

▶ **9** Modifica el código anterior para intentar que el vector comience en la dirección de memoria `0x1000 0002` ¿En qué dirección comienza realmente? ¿Por qué? ¿Crees que tiene algo que ver la directiva «**.word**»?

2.2. Declaración de bytes en memoria

La directiva «**.byte** DATO» inicializa una posición de memoria, es decir, un byte, con el contenido DATO.

EJERCICIOS

Limpia el contenido de la memoria y carga el siguiente código en el simulador:

✎ datos-byte.s

```
1           .data               # comienzo de la zona de datos
2 octeto:   .byte 0x15          # DATO expresado en hexadecimal
```

▶ 10 ¿Qué dirección de memoria se ha inicializado con el valor `0x15`?

▶ 11 ¿Qué valor posee la palabra que contiene el byte?

Limpia el contenido de la memoria y carga el siguiente código en el simulador:

✎ datos-byte-palabra.s

```
1           .data                           # comienzo zona de datos
2 palabra1: .byte 0x10, 0x20, 0x30, 0x40 # datos en hexadecimal
3 palabra2: .word 0x10203040                # dato en hexadecimal
```

▶ 12 ¿Qué valores se han almacenado en memoria?

▶ 13 Viendo cómo se ha almacenado la secuencia de bytes y la palabra, ¿qué tipo de organización de los datos, *big-endian* o *little-endian*, utiliza el simulador? ¿Por qué?

▶ 14 ¿Qué valores toman las etiquetas «palabra1» y «palabra2»?

2.3. Declaración de cadenas de caracteres

La directiva « **.ascii** "cadena"» le indica al ensamblador que debe almacenar los códigos ASCII de los caracteres que componen la cadena entrecomillada. Dichos códigos se almacenan en posiciones consecutivas de memoria, de un byte cada una.

EJERCICIOS

Limpia el contenido de la memoria y carga el siguiente código en el simulador:

✎ datos-cadena.s

```
1           .data
2 cadena:   .ascii "abcde" # declaración de la cadena
3 octeto:   .byte 0xff
```

▶ 15 ¿Qué rango de posiciones de memoria se han reservado para la variable etiquetada con «cadena»?

▶ 16 ¿Cuál es el código ASCII de la letra «a»? ¿Y el de la «b»?

▶ 17 ¿A qué posición de memoria hace referencia la etiqueta « octeto »?

▶ 18 ¿Cuántos bytes se han reservado en total? ¿Y cuántas palabras?

▶ 19 La directiva « **.asciiz** "cadena"» también sirve para declarar cadenas. Modifica el programa anterior para que utilice « **.asciiz** » en lugar de « **.ascii** », ¿Hay alguna diferencia en el contenido de la memoria utilizada? ¿Cuál? Describe cuál es la función de esta directiva y qué utilidad tiene.

2.4. Reserva de espacio en memoria

La directiva «**.space** N» sirve para reservar N bytes de memoria e inicializarlos a 0.

EJERCICIOS

Dado el siguiente código:

✎ datos-space.s

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

▶ 20 ¿Qué posiciones de memoria se han reservado para almacenar la variable «espacio»?

▶ 21 ¿Los cuatro bytes utilizados por la variable «espacio» podrían ser leídos o escritos como si fueran una palabra? ¿Por qué?

▶ 22 ¿A partir de qué dirección se ha inicializado «byte1»? ¿Y «byte2»?

▶ 23 ¿A partir de qué dirección se ha inicializado « palabra »? ¿Por qué ha hecho esto el ensamblador? ¿Por qué no ha utilizado la siguiente posición de memoria sin más?

2.5. Alineación de datos en memoria

La directiva « **.align** N» le indica al ensamblador que el siguiente dato debe ser almacenado en una dirección de memoria que sea múltiplo de 2^n.

EJERCICIOS

Dado el siguiente código:

datos-align.s

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

▶ 24 ¿Qué posiciones de memoria se han reservado para almacenar la variable «espacio»? Compara la respuesta con la obtenida en el ejercicio 20.

▶ 25 ¿Los cuatro bytes utilizados por la variable «espacio» podrían ser leídos o escritos como si fueran una palabra? ¿Por qué? ¿Qué es lo que ha hecho la directiva « **.align** 2»?

2.6. Problemas del capítulo

EJERCICIOS

▶ 26 Desarrolla un programa ensamblador que reserve espacio para dos vectores consecutivos, A y B, de 20 palabras cada uno a partir de la dirección `0x1000 0000`.

▶ 27 Desarrolla un programa ensamblador que realice la siguiente reserva de espacio en memoria a partir de la dirección `0x1000 1000`: una palabra, un byte y otra palabra alineada en una dirección múltiplo de 4.

▶ 28 Desarrolla un programa ensamblador que realice la siguiente reserva de espacio e inicialización de memoria: una palabra con el valor `3`, un byte con el valor `0x10`, una reserva de 4 bytes que comience en una dirección múltiplo de 4, y un byte con el valor `20`.

▶ 29 Desarrolla un programa ensamblador que inicialice, en el espacio de datos, la cadena de caracteres «Esto es un problema», utilizando:

a) La directiva « **.ascii** »

b) La directiva «**.byte**»

c) La directiva «**.word**»

*(Pista: Comienza utilizando sólo la directiva « **.ascii** » y visualiza cómo se almacena en memoria la cadena para obtener la secuencia de bytes.)*

▶ **30** Sabiendo que un entero ocupa una palabra, desarrolla un programa ensamblador que inicialice en la memoria, a partir de la dirección `0x1001 0000`, la matriz A de enteros definida como:

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

suponiendo que:

a) La matriz A se almacena por filas (los elementos de una misma fila se almacenan de forma contigua en memoria).

b) La matriz A se almacena por columnas (los elementos de una misma columna se almacenan de forma contigua en memoria).

. . . . . .

CAPÍTULO 3

CARGA Y ALMACENAMIENTO

Índice

3.1. Carga de datos inmediatos (constantes) **26**
3.2. Carga de palabras (de memoria a registro) **29**
3.3. Carga de bytes (de memoria a registro) **30**
3.4. Almacenamiento de palabras (de registro a memoria) . . . **31**
3.5. Almacenamiento de bytes (bytes de registro a memoria) . **32**
3.6. Problemas del capítulo **33**

El estilo del juego de instrucciones del MIPS32 es del tipo carga/almacenamiento (*load/store*). Esto es, los operandos deben estar en registros para poder operar con ellos ya que el juego de instrucciones no incluye instrucciones que puedan operar directamente con operandos en memoria. Por tanto, para realizar una operación con datos en memoria se deben cargar dichos datos en registros, realizar la operación y finalmente almacenar el resultado, si fuera el caso, en memoria.

Para la carga de datos en registros se pueden utilizar los modos de direccionamiento inmediato y directo a memoria. Para el almacenamiento de datos en memoria únicamente el modo de direccionamiento directo a memoria.

En la primera sección se verán las instrucciones de carga de datos inmediatos. Las restantes secciones están dedicadas a las instrucciones de carga y almacenamiento en memoria.

3.1. Carga de datos inmediatos (constantes)

En esta sección se verá la instrucción «**lui**» que carga un dato inmediato en los 16 bits de mayor peso de un registro y las pseudo-instrucciones «**li**» y «**la**» que cargan un dato inmediato de 32 bits y una dirección de memoria, respectivamente.

Se comienza viendo un programa de ejemplo con la instrucción «**lui**»:

✎ carga-lui.s

```
1          .text                # Zona de instrucciones
2  main:   lui $s0, 0x8690
```

Directiva ««.text [DIR]»»
Como se ha visto en el capítulo anterior, la directiva «**.data** [DIR]» se utiliza para indicar el comienzo de una zona de datos. De igual forma, la directiva «**.text** [DIR]» se tiene que utilizar para indicar el comienzo de la zona de memoria dedicada a instrucciones. Si no se especifica el parámetro opcional «DIR», la dirección de comienzo será la `0x0040 0024` (que es la primera posición de memoria libre a continuación del programa cargador que comienza en la dirección `0x0040 0000`).

La instrucción «**lui**» (del inglés ***l**oad **u**pper **i**nmediate*) almacena la media palabra indicada por el dato inmediato de 16 bits, en el ejemplo `0x8690`, en la parte alta del registro especificado, en este caso `$s0`; y, además, escribe el valor `0` en la media palabra de menor peso de dicho registro. Es decir, después de la ejecución del programa anterior, el contenido del registro `$s0` será `0x8690 0000`.

EJERCICIOS

▶ **31** Carga el anterior programa en el simulador, localiza en la zona de visualización de código la instrucción «**lui** $s0, 0x8690» e indica:

a) La dirección de memoria en la que se encuentra.

b) El tamaño que ocupa.

c) La representación de la instrucción en código máquina.

d) El formato de instrucción empleado.

e) El valor de cada uno de los campos de dicha instrucción.

▶ **32** Ejecuta el programa y comprueba que realmente realiza lo que se espera de él.

En el caso de que se quisiera cargar un dato inmediato de 32 bits en un registro, no se podría utilizar la instrucción «**lui**». De hecho, no se podría utilizar ninguna de las instrucciones del juego de instrucciones del MIPS32 ya que todas ellas son de 32 bits y no se podría ocupar toda la instrucción con el dato que se desea cargar.

La solución consiste en utilizar dos instrucciones: la primera de ellas sería la instrucción «**lui**» en la que se especificarían los 16 bits de mayor peso del dato de 32 bits; la segunda de las instrucciones sería una instrucción «**ori**» que serviría para cargar los 16 bits de menor peso respetando los 16 bits de mayor peso ya cargados. La instrucción «**ori**» se verá con más detalle en el siguiente capítulo. Por el momento, sólo interesa saber que se puede utilizar en conjunción con «**lui**» para cargar un dato inmediato de 32 bits en un registro.

Se supone que se quiere cargar el dato inmediato `0x86901234` en el registro `$s0`. Un programa que haría esto sería el siguiente:

✎ carga-lui-ori.s

```
1           .text                   # Zona de instrucciones
2 main:     lui $s0, 0x8690
3           ori $s0, $s0, 0x1234
```

EJERCICIOS

▶ **33** Carga el programa anterior en el simulador, ejecuta paso a paso el programa y responde a las siguientes preguntas:

a) ¿Qué valor contiene `$s0` después de ejecutar «**lui** $s0, 0x8690»?

b) ¿Qué valor contiene `$s0` después de ejecutar «**ori** $s0, $s0, 0x1234»?

Puesto que cargar una constante de 32 bits en un registro es una operación bastante frecuente, el ensamblador del MIPS proporciona una pseudo-instrucción para ello: la pseudo-instrucción «**li**» (del inglés ***l**oad **i**nmediate*). Se va a ver un ejemplo que utiliza dicha pseudoinstrucción:

✎ carga-li.s

```
1           .text                   # Zona de instrucciones
2 main:     li $s0, 0x12345678
```

EJERCICIOS

▶ **34** Carga y ejecuta el programa anterior. ¿Qué valor tiene el registro `$s0` después de la ejecución?

▶ 35 Puesto que «**li**» es una pseudo-instrucción, el ensamblador ha tenido que sustituirla por instrucciones máquina equivalentes. Examina la zona de visualización de código y localiza las instrucciones máquina generadas por el ensamblador.

Hasta ahora se ha visto la instrucción «**lui**» y la pseudo-instrucción «**li**» que permiten cargar un dato inmediato en un registro. Ambas sirven para especificar en el programa el valor constante que se desea cargar en el registro.

La última pseudo-instrucción que queda por ver en esta sección es la pseudo-instrucción «**la**» (del inglés ***l**oad **a**ddress*). Esta pseudo-instrucción permite cargar la dirección de un dato en un registro. Pese a que se podría utilizar una constante para especificar la dirección de memoria del dato, es más cómodo utilizar la etiqueta que previamente se haya asociado a dicha dirección de memoria y dejar que el ensamblador haga el trabajo sucio.

El siguiente programa contiene varias pseudo-instrucciones «**la**»:

✎ carga-la.s

```
1           .data                   # Zona de datos
2  palabra1: .word 0x10
3  palabra2: .word 0x20
4
5           .text                   # Zona de instrucciones
6  main:    la $s0, palabra1
7           la $s1, palabra2
8           la $s2, 0x10010004
```

EJERCICIOS

▶ 36 Carga el anterior programa en el simulador y contesta a las siguientes preguntas:

a) ¿Qué instrucción o instrucciones máquina genera el ensamblador para resolver la instrucción «**la** $s0, palabra1»?

b) ¿Y para la instrucción «**la** $s1, palabra2»?

c) ¿Y para la instrucción «**la** $s2, 0x10010004»?

▶ 37 Ejecuta el programa anterior y responde a las siguientes preguntas:

a) ¿Qué valor hay en el registro `$s0`?

b) ¿Y en el registro `$s1`?

c) ¿Y en el registro `$s2`?

▶ 38 Visto que tanto la instrucción «**la** $s1, palabra2» como la instrucción «**la** $s2, 0x10010004» realizan la misma acción, salvo por el hecho de que almacenan el resultado en un registro distinto, ¿qué ventaja proporciona utilizar la instrucción «**la** $s1, palabra2» en lugar de «**la** $s2, 0x10010004»?

3.2. Carga de palabras (de memoria a registro)

Para cargar una palabra de memoria a registro se utiliza la siguiente instrucción «**lw** rt,Inm(rs)» (del inglés ***load word***). Dicha instrucción lee una palabra de la posición de memoria indicada por la suma de un dato inmediato («Inm») y el contenido de un registro («rs») y la carga en el registro indicado («rt»).

Veamos un ejemplo:

carga-lw.s

```
1           .data                   # Zona de datos
2  palabra: .word 0x10203040
3
4           .text                   # Zona de instrucciones
5  main:    lw $s0, palabra($0)     # $s0<-M[palabra]
```

En el programa anterior, la instrucción «**lw** $s0, palabra($0)», se utiliza para cargar en el registro `$s0` la palabra contenida en la dirección de memoria indicada por la suma de la etiqueta «palabra» y el contenido del registro `$0`. Puesto que la etiqueta «palabra» se refiere a la posición de memoria `0x1001 0000` y el contenido del registro `$0` es 0, la dirección de memoria de la que se leerá la palabra será la `0x1001 0000`.

EJERCICIOS

Crea un fichero con el código anterior, cárgalo en el simulador y contesta a las siguientes preguntas.

▶ 39 Localiza la instrucción en la zona de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

▶ 40 Explica cómo se obtiene a partir de esas instrucciones la dirección de palabra. ¿Por qué traduce el simulador de esta forma la instrucción original?

▶ 41 Indica el formato de cada una de las instrucciones generadas y los campos que las forman.

▶ 42 ¿Qué hay en el registro `$s0` antes de ejecutar el programa? Ejecuta el programa. ¿Qué contenido tiene ahora el registro `$s0`?

▶ 43 Antes se ha visto una pseudo-instrucción que permite cargar la dirección de un dato en un registro. Modifica el programa original para que utilizando esta pseudo-instrucción haga la misma tarea. Comprueba qué conjunto de instrucciones sustituyen a la pseudo-instrucción utilizada una vez el programa ha sido cargado en la memoria del simulador.

▶ 44 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». Al intentar ejecutarlo se verá que no es posible. ¿Por qué no?

3.3. Carga de bytes (de memoria a registro)

La instrucción «**lb** rt ,Inm(rs)» (del inglés ***l**oad **b**yte*) carga un byte de memoria y lo almacena en el registro indicado. Al igual que en la instrucción «**lw**», la dirección de memoria se obtiene sumando un dato inmediato (Inm) y el contenido de un registro (rs). Veamos un programa de ejemplo:

✎ carga-lb.s

```
1           .data                   # Zona de datos
2  octeto:  .byte 0xf3
3  otro:    .byte 0x20
4
5           .text                   # Zona de instrucciones
6  main:    lb $s0, octeto($0)      # $s0<-M[octeto]
7           lb $s1, otro($0)        # $s1<-M[otro]
```

EJERCICIOS

▶ 45 Carga el programa anterior en el simulador y localiza las dos instrucciones «**lb**» ¿Qué instrucciones máquina ha puesto el ensamblador en lugar de cada una de ellas?

▶ 46 Observa la zona de visualización de datos, ¿qué valor contiene la palabra `0x1001 0000`?

▶ 47 Ejecuta el programa y responde a las siguientes preguntas:

a) ¿Qué valor contiene el registro `$s0` en hexadecimal?

b) ¿Qué valor contiene el registro `$s1` en hexadecimal?

c) ¿Qué entero representa `0xf3` en complemento a 2 con 8 bits?

d) ¿Qué entero representa `0xffff fff3` en complemento a 2 con 32 bits?

e) ¿Por qué al cargar un byte de memoria en un registro se modifica el registro entero? *(Pista: Piensa en lo que se querría hacer después con el registro.)*

La instrucción «**lb**» carga un byte de memoria en un registro manteniendo su signo. Esto es útil si el dato almacenado en el byte de memoria es efectivamente un número entero pero no en otro caso. Por ejemplo, si se trata de un número natural (de 0 a 255) o de la representación en código ASCII (extendido) de un carácter.

Cuando se quiera cargar un byte sin tener en cuenta su posible signo se utilizará la instrucción «**lbu** rt , Inm(rs)».

EJERCICIOS

▶ **48** Reemplaza en el programa anterior las instrucciones «**lb**» por instrucciones «**lbu**» y ejecuta el nuevo programa.

a) ¿Qué valor hay ahora en `$s0`? ¿Ha cambiado este valor con respecto al obtenido en el programa original? ¿Por qué?

b) ¿Qué valor hay ahora en `$s1`? ¿Ha cambiado este valor con respecto al obtenido en el programa original? ¿Por qué?

Dado el siguiente programa:

carga-lb2.s

```
1           .data                   # Zona de datos
2  octeto:  .word 0x10203040
3  otro:    .byte 0x20
4
5           .text                   # Zona de instrucciones
6  main:    lb $s0, octeto($0)      # $s0<--M[octeto]
7           lb $s1, otro($0)        # $s1<--M[otro]
```

▶ **49** ¿Cuál es el valor del registro `$s0` una vez ejecutado? ¿Por qué?

3.4. Almacenamiento de palabras (de registro a memoria)

Para almacenar una palabra desde un registro a memoria se utiliza la siguiente instrucción: «**sw** rt ,Inm(rs)» (del inglés ***store word***). Dicha instrucción lee la palabra almacenada en el registro indicado («rt») y la almacena en la posición de memoria indicada por la suma de un dato inmediato («Inm») y el contenido de un registro («rs»).

Veamos un ejemplo:

✎ carga-sw.s

```
1           .data                   # Zona de datos
2  palabra1: .word 0x10203040
3  palabra2: .space 4
4  palabra3: .word 0xffffffff
5
6           .text                    # Zona de instrucciones
7  main:    lw $s0, palabra1($0)
8           sw $s0, palabra2($0)    # M[palabra2]<-$s0
9           sw $s0, palabra3($0)    # M[palabra3]<-$s0
```

EJERCICIOS

▶ **50** ¿Qué hará dicho programa? ¿Qué direcciones de memoria se verán afectadas? ¿Qué valores habrán antes y después de ejecutar el programa?

▶ **51** Carga el programa en el simulador y comprueba que los valores que hay en memoria antes de la ejecución son los que se habían predicho. Ejecuta el programa y comprueba que realmente se han modificado las direcciones de memoria que se han indicado previamente y con los valores predichos.

▶ **52** ¿Qué instrucciones máquina se utilizan en lugar de la siguiente instrucción en ensamblador: «**sw** $s0, palabra3($0)»?

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

Para almacenar un byte desde un registro a memoria se utiliza la instrucción «**sb** rt ,Inm(rs)» (del inglés ***s**tore **b**yte*). Dicha instrucción lee el **byte de menor peso** almacenado en el registro indicado («rt») y lo almacena en la posición de memoria indicada por la suma de un dato inmediato («Inm») y el contenido de un registro («rs»).

Veamos un ejemplo:

✎ carga-sb.s

```
1           .data                   # Zona de datos
2  palabra:  .word 0x10203040
3  octeto:   .space 2
4
5           .text                   # Zona de instrucciones
6  main:    lw $s0, palabra($0)
7           sb $s0, octeto($0)     # M[octeto]<-byte menor peso de $s0
```

EJERCICIOS

▶ **53** ¿Qué hará dicho programa? ¿Qué direcciones de memoria se verán afectadas? ¿Qué valores habrán antes y después de ejecutar el programa?

▶ 54 Carga el programa en el simulador y comprueba que los valores que hay en memoria antes de la ejecución son los que se habían predicho. Ejecuta el programa y comprueba que realmente se han modificado las direcciones de memoria que se han indicado previamente y con los valores predichos.

▶ 55 Modifica el programa para que el byte sea almacenado en la dirección «octeto+1» (basta con cambiar la instrucción «**sb** $s0, octeto ($0)» por «**sb** $s0, octeto+1($0)»). Comprueba y describe el resultado de este cambio.

▶ 56 Vuelve a modificar el programa para que el byte se almacene en la dirección de memoria «palabra+3» en lugar de en «octeto+1». ¿Qué valor tiene la palabra almacenada en la dirección `0x1001 0000` después de la ejecución?

3.6. Problemas del capítulo

EJERCICIOS

▶ 57 Desarrolla un programa ensamblador que inicialice un vector de enteros, V, definido como $V = (10, 20, 25, 500, 3)$. El vector debe comenzar en la dirección de memoria `0x1000 0000`. El programa debe cargar los elementos de dicho vector en los registros `$s0` al `$s4`.

▶ 58 Amplía el anterior programa para que además copie a memoria el vector V comenzando en la dirección `0x1001 0000`. *(Pista: En un programa ensamblador se pueden utilizar tantas directivas del tipo «**.data**» como sean necesarias.)*

▶ 59 Desarrolla un programa ensamblador que dada la siguiente palabra, `0x1020 3040`, almacenada en una determinada posición de memoria, la reorganice en otra posición invirtiendo el orden de sus bytes.

▶ 60 Desarrolla un programa ensamblador que dada la siguiente palabra, `0x1020 3040`, almacenada en una determinada posición de memoria, la reorganice en la misma posición intercambiando el orden de sus medias palabras. (Nota: las instrucciones «**lh**» (del inglés ***l**oad **h**alf*) y «**sh**» (del inglés ***s**ave **h**alf*) cargan y almacenan medias palabras, respectivamente).

▶ 61 Desarrolla un programa ensamblador que inicialice cuatro bytes a partir de la posición `0x1001 0002` con los valores `0x10`, `0x20`, `0x30`, `0x40` y reserve espacio para una palabra a partir de la dirección `0x1001 0010`. El programa debe transferir los cuatro bytes contenidos a partir de la posición `0x1001 0002` a la dirección `0x1001 0010`.

CAPÍTULO 4

OPERACIONES ARITMÉTICAS, LÓGICAS Y DE DESPLAZAMIENTO

Índice

4.1. Operaciones aritméticas **36**
4.2. Operaciones lógicas **39**
4.3. Operaciones de desplazamiento **41**
4.4. Problemas del capítulo **43**

Hasta ahora se ha visto cómo se puede indicar en lenguaje ensamblador que se quieren inicializar determinadas posiciones de memoria con determinados valores, cómo se pueden cargar estos valores de la memoria a los registros del procesador y cómo almacenar en memoria la información contenida en los registros.

Es decir, ya se sabe cómo definir e inicializar las variables de un programa, cómo transferir el contenido de dichas variables en memoria a los registros, para así poder realizar las operaciones oportunas y, finalmente, cómo transferir el contenido de los registros a memoria para almacenar el resultado de las operaciones realizadas.

Sin embargo, no se han visto cuáles son las operaciones que puede realizar el procesador. En éste y en el siguiente capítulo se verán algunas de las operaciones básicas más usuales que puede realizar el procesador. Y las instrucciones que las implementan.

Este capítulo, como su nombre indica, presenta algunas de las instrucciones proporcionadas por el MIPS32 para la realización de operaciones aritméticas (suma, resta, multiplicación y división), lógicas (AND y OR) y de desplazamiento de bits (a izquierdas y a derechas, aritmético o lógico).

4.1. Operaciones aritméticas

Para introducir las operaciones aritméticas que el MIPS32 puede realizar se comienza viendo el siguiente programa que suma un operando almacenado originalmente en memoria y un valor constante proporcionado en la propia instrucción de suma. Observa que para realizar la suma, será necesario en primer lugar, cargar el valor que se quiere sumar de la posición de memoria en la que se encuentra, a un registro.

✎ oper-addiu.s

```
1           .data                   # Zona de datos
2 numero:   .word 2147483647        # Máx. positivo representable en Ca2(32)
3                                   # (en hexadecimal 0x7fff ffff)
4
5           .text                   # Zona de instrucciones
6 main:     lw    $t0, numero($0)
7           addiu $t1, $t0, 1       # $t1<-$t0+1
```

La instrucción «**addiu** $t1, $t0, 1», que es del tipo «**addiu** rt, rs, Inm», realiza una suma de dos operandos sin signo. Uno de los operandos está almacenado en un registro, en `rs`, y el otro en la propia instrucción, en el campo `Inm`; el resultado se almacenará en el registro `rt`. Puesto que el campo destinado al almacenamiento del número inmediato es de 16 bits y el operando ha de tener 32 bits, es necesario extender la información proporcionada en «Inm» de 16 a 32 bits. Esta extensión se realiza en las operaciones aritméticas extendiendo el signo a los 16 bits más significativos. Por ejemplo, el dato inmediato de 16 bits `0xfffe` cuando se utiliza como operando de una operación aritmética, se extiende al valor de 32 bits `0xffff fffe` (que en binario es $1111111\,1111111\,1111111\,11111110_2$).

EJERCICIOS

Carga el fichero anterior en el simulador SPIM y ejecútalo.

▶ **62** Localiza el resultado de la suma. ¿Cuál ha sido? ¿El resultado obtenido es igual a $2.147.483.647 + 1$?

▶ **63** Sustituye en el programa anterior la instrucción «**addiu**» por la instrucción «**addi**». Borra los valores de la memoria, carga de nuevo el fichero fuente y vuelve a ejecutarlo (no en modo paso a paso). ¿Qué ha ocurrido al efectuar este cambio? ¿Por qué?

Operaciones con signo y «sin signo»

El método de representación utilizado por el MIPS32 para expresar los números positivos (o sin signo) es, como no podía ser de otra forma, el binario natural. En cuanto a la representación de números enteros (con signo), el método de representación escogido ha sido el complemento a 2 (Ca2). De hecho, la mayoría de procesadores suelen utilizar dicha representación debido a las ventajas que proporciona frente a las otras posibles representaciones de números enteros. Una de estas ventajas es la facilidad de implementación de las operaciones de suma, resta y cambio de signo: la suma es simplemente la suma binaria de los operandos —da igual el signo de los operandos—; la resta consiste en cambiar el signo del segundo operando y sumar; y el cambio de signo en la inversión de unos por ceros y ceros por unos y sumar 1.

Otra de las ventajas de utilizar la representación en Ca2 es que de este modo, la implementación de las operaciones de suma y resta es independiente de si los operandos son enteros (representados en Ca2) o no (representados en binario natural). Es decir, no es necesario disponer de hardware diferenciado para sumar o restar números enteros o positivos.

De hecho, tan sólo se ha de tener en cuenta si los operandos son enteros o positivos para la detección del desbordamiento (*overflow*), es decir, para la detección de si el resultado de la operación requiere más bits de los disponibles para su correcta representación en el método de representación correspondiente. Cuando esto ocurre, el procesador debe generar una excepción (llamada *excepción por desbordamiento*) que informe de que ha ocurrido este error y que el resultado obtenido no es, por tanto, correcto.

Puesto que la detección de desbordamiento es distinta en función de si los operandos son con o sin signo, el juego de instrucciones del MIPS32 proporciona instrucciones diferenciadas para sumas y restas de operandos con y sin signo. Así, por ejemplo, la instrucción «**add**» suma el contenido de dos registros y la instrucción «**addu**» («u» de *unsigned*) realiza la misma operación pero considerando que los operandos son números positivos.

En la práctica, las aplicaciones en las que se utilizan números sin signo no suelen generar desbordamientos, ya que su rango de aplicación suele estar delimitado: tratamiento de cadenas, cálculo de posiciones de memoria... Debido a esto, las instrucciones del MIPS32 para realizar operaciones aritméticas sin signo se diferencian de las que operan con signo simplemente en que no generan una excepción en el caso de que se produzca un desbordamiento.

Las operaciones aritméticas en las que uno de los operandos es una constante aparecen con relativa frecuencia, p.e. para incrementar la variable índice en bucles del tipo «**for**(**int** i; i<n; i++)». Sin embargo, es más habitual encontrar instrucciones que requieran que ambos operandos sean variables (es decir, que el valor de ambos operandos no sea conocido en el momento en el que el programador escribe la instrucción).

El siguiente programa presenta la instrucción «**subu** rd, rs, rt» que realiza una resta de números sin signo, donde los operandos fuente están en los registros `rs` y `rt`. En concreto, la instrucción «**subu** rd, rs, rt» realiza la operación `rd` ← `rs` − `rt`.

✎ oper-subu.s

```
1          .data                  # Zona de datos
2  numero1: .word -2147483648     # Máx. negativo representable en Ca2(32)
3                                 # (en hexadecimal 0x80000000)
4  numero2: .word 1
5  numero3: .word 2
6
7          .text                  # Zona de instrucciones
8  main:   lw   $t0, numero1($0)
9          lw   $t1, numero2($0)
10         subu $t0, $t0, $t1     # $t0 <- $t0-$t1
11         lw   $t1, numero3($0)
12         subu $t0, $t0, $t1     # $t0 <- $t0-$t1
13         sw   $t0, numero3($0)
```

EJERCICIOS

Carga el programa anterior en el simulador, ejecútalo paso a paso y contesta a las siguientes preguntas:

▶ **64** ¿Qué es lo que hace? ¿Qué resultado se almacena en la dirección de memoria etiquetada como «numero3»? ¿Es correcto?

▶ **65** ¿Se produce algún cambio si sustituyes las instrucciones «**subu**» por instrucciones «**sub**»? En caso de producirse, ¿a qué es debido este cambio?

Hasta ahora se han visto instrucciones para realizar sumas y restas. A continuación se verá cómo realizar las operaciones de multiplicación y división. Dichas operaciones utilizan de forma implícita los registros `HI` y `LO` para almacenar el resultado de la operación correspondiente. En los ejercicios que aparecerán a continuación se descubrirá la utilidad de dichos registros y la información que se almacena en cada uno de ellos en función de la operación realizada.

Como primer ejemplo se va a ver el siguiente programa que multiplica dos números y almacena el resultado de la operación (2 palabras) en las posiciones de memoria contiguas al segundo operando.

oper-mult.s

```
1           .data                   # Zona de datos
2  numero1: .word 0x7fffffff        # Máx. positivo representable en Ca2(32)
3  numero2: .word 16
4           .space 8
5
6           .text                   # Zona de instrucciones
7  main:    lw   $t0, numero1($0)
8           lw   $t1, numero2($0)
9           mult $t0, $t1             # HI,LO <- $t0 x $t1
10          mfhi $t0                  # $t0 <- HI
11          mflo $t1                  # $t1 <- LO
12          sw   $t0, numero2+4($0) # M[numero2+4] <- $t0
13          sw   $t1, numero2+8($0) # M[numero2+8] <- $t1
```

EJERCICIOS

Carga el programa anterior en el simulador, ejecútalo paso a paso y contesta a las siguientes preguntas:

▶ **66** La instrucción «**mult** $t0, $t1» almacena el resultado en los registros `HI` y `LO`. ¿Qué instrucciones se utilizan para recuperar dicha información?

▶ **67** ¿Qué resultado se obtiene después de ejecutar «**mult** $t0, $t1»? ¿En qué registros se almacena el resultado? ¿Cuál es este resultado? ¿Por qué se utilizan dos palabras?

▶ **68** Observa que dos de las instrucciones del programa anterior utilizan la suma de una etiqueta y una constante para determinar el valor del dato inmediato. ¿Qué instrucciones son?

▶ **69** Utilizando como guía el programa anterior, crea uno nuevo que divida `10` entre `3` y almacene el cociente y el resto en las dos palabras siguientes a la dirección de memoria referencia por la etiqueta «numero2». Para ello, ten en cuenta que la instrucción de división entera, «**div** rs, rt», divide `rs` entre `rt` y almacena el resto en `HI` y el cociente en `LO`.

4.2. Operaciones lógicas

El procesador MIPS32 proporciona un conjunto de instrucciones que permite realizar operaciones lógicas del tipo AND, OR, XOR, entre otras. Se debe tener en cuenta que las operaciones lógicas, aunque toman como operandos dos palabras, actúan bit a bit. Así, por ejemplo, la AND de dos palabras genera como resultado una palabra en la que el bit 0 es la AND de los bits 0 de los operandos fuente, el bit 1 es la AND de los bits 1 de los operandos fuente, y así sucesivamente.

Se va a ver como ejemplo de las instrucciones que implementan las operaciones lógicas, la instrucción que implementa la operación AND.

La instrucción «**andi** rs , rt , Inm» realiza la operación lógica AND bit a bit del número almacenado en `rt` y el número almacenado en el campo `Inm` de la propia instrucción. Puesto que el campo destinado al almacenamiento del número inmediato es de 16 bits y el operando ha de tener 32 bits, es necesario extender la información proporcionada en «Inm» de 16 a 32 bits. Esta extensión se realiza en las operaciones lógicas colocando a 0 los 16 bits más significativos. Por ejemplo, el dato inmediato de 16 bits `0xfffe` cuando se utiliza como operando de una operación lógica, se extiende al valor de 32 bits `0x0000 fffe`, que en binario es:

$$00000000\,00000000\,11111111\,11111110_2.$$

oper-andi.s

```
1           .data                   # Zona de datos
2 numero:   .word 0x01234567
3           .space 4
4
5           .text                   # Zona de instrucciones
6 main:     lw   $t0, numero($0)
7           andi $t1, $t0, 0xfffe # 0xfffe es en binario:
8                                 #   1111 1111 1111 1110
9           sw   $t1,numero+4($0)
```

Por lo tanto, la instrucción «**andi** $t1 , $t0 , 0xfffe » realiza la AND lógica entre el contenido del registro `$t0` y el valor `0x0000 fffe`.

Como ya se sabe, la tabla de verdad de la operación AND es la siguiente:

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Es decir, sólo cuando los dos bits, a y b, valen 1 el resultado es 1.

De todas formas, para el motivo que nos ocupa, resulta más interesante describir el funcionamiento de la operación AND de otra manera: si uno de los bits es 0, el resultado es 0; si es 1, el resultado será igual al valor del otro bit. Que puesto en forma de tabla de verdad queda:

a	a AND b
0	0
1	b

Así, al realizar la AND de `0x0000 fffe` con el contenido de `$t0`, se sabe que independientemente de cual sea el valor almacenado en `$t0`, el resultado tendrá los bits 31 al 16 y el 0 con el valor 0 y los restantes bits con el valor

indicado por el número almacenado en `$t0`. Cuando se utiliza una secuencia de bits con este fin, ésta suele recibir el nombre de máscara de bits; ya que sirve para «ocultar» determinados bits del otro operando, a la vez que permite «ver» los bits restantes. Veamos un ejemplo: dada la máscara de bits anterior, `0x0000 fffe`, y suponiendo que el contenido de `$t0` fuera `0x0123 4567`, entonces, la instrucción «**andi** $t1, $t0, 0xfffe» equivaldría a:

$$\begin{array}{rr} & 00000000\,00000000\,11111111\,11111110_2 \\ \text{AND} & 00000001\,00100011\,01000101\,01100111_2 \\ \hline & 00000000\,00000000\,01000101\,01100110_2 \end{array}$$

Por lo tanto, el anterior programa pone a cero los bits del 31 al 16 (los 16 más significativos) y el 0 del número almacenado en «numero» y almacena el resultado en «numero+4».

EJERCICIOS

▶ **70** Carga el anterior programa en el simulador y ejecútalo. ¿Qué valor, expresado en hexadecimal, se almacena en la posición de memoria «numero+4»? ¿Coincide con el resultado calculado en la explicación anterior?

▶ **71** Modifica el código para que almacene en «numero+4» el valor obtenido al conservar tal cual los 16 bits más significativos del dato almacenado en «numero», y poner a cero los 16 bits menos significativos, salvo el bit cero, que también debe conservar su valor original. *(Pista: La instrucción «**and** rd, rs, rt» realiza la AND lógica de los registros `rs` y `rt` y almacena el resultado en `rd`.)*

▶ **72** Desarrolla un programa, basado en los anteriores, que almacene en la posición de memoria «numero+4» el valor obtenido al conservar tal cual los 16 bits más significativos del dato almacenado en «numero» y poner a uno los 16 bits menos significativos, salvo el bit cero, que también debe conservar su valor original. *(Pista: La operación lógica AND no sirve para este propósito, ¿que operación lógica se debe realizar?, ¿basta con una operación en la que uno de los operandos sea un dato inmediato?)*

4.3. Operaciones de desplazamiento

El procesador MIPS32 también proporciona instrucciones que permiten desplazar los bits del número almacenado en un registro un determinado número de posiciones a la derecha o a la izquierda.

El siguiente programa presenta la instrucción de desplazamiento aritmético a derechas (*shift right arithmetic*). La instrucción, «**sra** $t1, $t0, 4», desplaza el valor almacenado en el registro `$t0`, 4 bits a la derecha conservando su signo y almacena el resultado en `$t1`.

✎ oper-sra.s

```
1          .data                  # Zona de datos
2  numero: .word 0xffffff41
3
4          .text                  # Zona de instrucciones
5  main:   lw   $t0, numero($0)
6          sra  $t1, $t0, 4       # $t1 <- $t0>>4
```

EJERCICIOS

▶ **73** Carga el programa anterior en el simulador y ejecútalo, ¿qué valor se almacena en el registro `$t1`? ¿Qué se ha hecho para conservar el signo del número original en el desplazado? Modifica el programa para comprobar su funcionamiento cuando el número que se debe desplazar es positivo.

▶ **74** Modifica el programa propuesto originalmente para que realice un desplazamiento de 3 bits. Como se puede observar, la palabra original era `0xffff ff41` y al desplazarla se ha obtenido la palabra `0xffff ffe8`. Representa ambas palabras en binario y comprueba si la palabra `0xffff ffe8` corresponde realmente al resultado de desplazar `0xffff ff41` 3 bits a la derecha conservando su signo.

▶ **75** La instrucción «**srl**», desplazamiento lógico a derechas (*shift right logic*), también desplaza a la derecha un determinado número de bits el valor indicado. Sin embargo, no tiene en cuenta el signo y rellena siempre con ceros. Modifica el programa original para que utilice la instrucción «**srl**» en lugar de la «**sra**» ¿Qué valor se obtiene ahora en `$t1`?

▶ **76** Modifica el código para desplazar el contenido de «numero» 2 bits a la izquierda. *(Pista: La instrucción de desplazamiento a izquierdas responde al nombre en inglés de* shift left logic*)*

▶ **77** Desplazar n bits a la izquierda equivale a una determinada operación aritmética (siempre que no se produzca un desbordamiento). ¿A qué operación aritmética equivale? ¿Según esto, desplazar 1 bit a la izquierda a qué equivale? ¿Y desplazar 2 bits?

▶ **78** Por otro lado, desplazar n bits a la derecha conservando el signo también equivale a una determinada operación aritmética. ¿A qué operación aritmética equivale? (Nota: si el número es positivo el desplazamiento corresponde siempre a la operación indicada; sin embargo, cuando el número es negativo, el desplazamiento no produce siempre el resultado exacto.)

4.4. Problemas del capítulo

EJERCICIOS

▶ **79** Desarrolla un programa en ensamblador que defina el vector de enteros de dos elementos $V = (10, 20)$ en la memoria de datos comenzando en la dirección `0x1000 0000` y almacene la suma de sus elementos en la primera dirección de memoria no ocupada después del vector.

▶ **80** Desarrolla un programa en ensamblador que divida entre 5 los enteros 18 y -1215, estos números deben estar almacenados en las direcciones `0x1000 0000` y siguiente, respectivamente; y que almacene el cociente de ambas divisiones comenzando en la dirección de memoria `0x1001 0000`. *(Pista: Recuerda que en un programa ensamblador se puede utilizar la directiva «**.data**» tantas veces como se requiera.)*

▶ **81** Desarrolla un programa que modifique el entero `0xabcd12bd` almacenado en la dirección de memoria `0x1000 0000` de la siguiente forma: los bits 9, 7 y 3 deberán ponerse a cero mientras que los bits restantes deben conservar el valor original.

▶ **82** Desarrolla un programa que multiplique por 32 el número `0x1237`, almacenado en la dirección de memoria `0x1000 0000`. No puedes utilizar instrucciones aritméticas.

CAPÍTULO 5

OPERACIONES DE SALTO CONDICIONAL Y DE COMPARACIÓN

Índice

5.1. Operaciones de salto condicional **46**
5.2. Operaciones de comparación **48**
5.3. Evaluación de condiciones **48**
5.4. Problemas del capítulo **53**

Según lo visto hasta ahora, una vez que el procesador ha completado la ejecución de una instrucción, continúa con la ejecución de la siguiente instrucción: aquella que se encuentra en la posición de memoria siguiente a la de la instrucción que acaba de ejecutar. La ejecución de una instrucción tras otra, siguiendo el órden en el que están en memoria, es lo que se denomina *ejecución secuencial*. Ejemplos de ejecución secuencial son los programas vistos en los capítulos anteriores. Éstos estaban formados por una serie de instrucciones que se debían ejecutar una tras otra, siguiendo el orden en el que se encontraban en memoria, hasta que el programa finalizaba.

Sin embargo, este forma de ejecución tiene bastantes limitaciones. Un programa de ejecución secuencial no puede, por ejemplo, tomar diferentes acciones en función de los datos de entrada, o de los resultados obtenidos, o de la interacción con el usuario. Tampoco puede realizar un número n de veces ciertas operaciones repetitivas (a no ser que el programador haya escrito n veces

las mismas operaciones; y en este caso, n debería de ser un número predeterminado y no podría variar en sucesivas ejecuciones).

Debido a la gran ventaja que supone el que un programa pueda tomar diferentes acciones y que pueda repetir un conjunto de instrucciones un determinado número de veces, los lenguajes de programación proporcionan estructuras de control que permiten llevar a cabo dichas acciones: las estructuras de control condicionales y repetitivas. Estas estructuras de control permiten modificar el flujo secuencial de instrucciones. En particular, las estructuras de control condicionales permiten la ejecución de ciertas partes del código en función de una serie de condiciones. Y las estructuras de control repetitivas permiten la repetición de cierta parte del código hasta que se cumpla una determinada condición de parada.

Para poder implementar las estructuras de control condicionales y repetitivas, el juego de instrucciones del MIPS32 incorpora un conjunto de instrucciones que permiten realizar saltos condicionales y comparaciones.

En este capítulo se presentan las instrucciones de salto condicional y de comparación del MIPS32. Y en el siguiente, cómo utilizar dichas instrucciones para implementar algunas de las estructuras de control condicional y repetitivas que suelen proporcionar los lenguajes de alto nivel.

Este capítulo está estructurado como sigue. En la primera sección se presentan las instrucciones de salto condicional. En la segunda, las instrucciones de comparación. La tercera sección proporciona ejemplos y ejercicios de cómo evaluar condiciones simples y compuestas. Finalmente, se proporcionan varios ejercicios sobre los temas tratados en el capítulo.

5.1. Operaciones de salto condicional

El lenguaje máquina del MIPS32 proporciona dos instrucciones básicas de salto condicional: «**beq**» (*branch if equal*, salta si igual) y «**bne**» (*branch if not equal*, salta si distinto). Su sintaxis es la siguiente:

- «**beq** rs , rt , etiqueta »
- «**bne** rs , rt , etiqueta »

Ambas instrucciones realizan una comparación de los valores almacenados en los registros `rs` y `rt` y dependiendo del resultado de dicha comparación, *verdadera* o *falsa*, saltan a la dirección de la instrucción referenciada por la « etiqueta » o no. En el caso de la instrucción «**beq**», el resultado de la comparación será *verdadero* cuando los valores almacenados en ambos registros

sean idénticos. En el caso de la instrucción «**bne**», el resultado será *verdadero* cuando los valores almacenados en ambos registros sean distintos.

Además de las instrucciones de salto condicional anteriores, que comparan el contenido de dos registros para decidir si se realiza o no el salto indicado, el procesador MIPS proporciona instrucciones para comparar el contenido de un registro con el número `0`. Estas instrucciones son: «**bgez**» (*branch if greater of equal than zero*, salta si mayor o igual que cero), «**bgtz**» (*branch if greater than zero*, salta si mayor que cero), «**blez**» (*branch if less or equal than zero*, salta si menor o igual que cero) y «**bltz**» (*branch if less than zero*, salta si menor que cero). La sintaxis de estas instrucciones de comparación de un registro con el número `0` es:

- «**bgez** rs , etiqueta »
- «**bgtz** rs , etiqueta »
- «**blez** rs , etiqueta »
- «**bltz** rs , etiqueta »

Las instrucciones anteriores comparan el contenido del registro `rs` con el número `0` y saltan a la dirección de la instrucción referenciada por « etiqueta » cuando $\texttt{rs} \geq 0$ (caso de «**bgez**»), $\texttt{rs} > 0$ («**bgtz**»), $\texttt{rs} \leq 0$ («**blez**») y $rs < 0$ («**bltz**»).

Las instrucciones «**beq**», «**bne**», «**bgez**», «**bgtz**», «**blez**» y «**bltz**», son todas las instrucciones de salto condicional que proporciona el procesador MIPS32. Sin embargo, y para facilitar la programación en ensamblador, el ensamblador del MIPS32 proporciona además las siguientes pseudo-instrucciones:

- «**bge** rs , rt , etiqueta » (salta si $\texttt{rs} \geq \texttt{rt}$)
- «**bgt** rs , rt , etiqueta » (salta si $\texttt{rs} > \texttt{rt}$)
- «**ble** rs , rt , etiqueta » (salta si $\texttt{rs} \leq \texttt{rt}$)
- «**blt** rs , rt , etiqueta » (salta si $\texttt{rs} < \texttt{rt}$)

EJERCICIOS

▶ **83** ¿Qué significa en inglés el nemónico «**bge**»? ¿Y «**blt**»?

5.2. Operaciones de comparación

Conviene introducir aquí un poco de notación. Para representar que el resultado de una comparación se almacena en un registro, se utiliza una expresión similar a: «rd $<-$ (rs $<$ rt)» (esta expresión en particular, indica que se almacena en `rd` el resultado de comparar si `rs` es menor que `rt`).

El resultado de una comparación puede ser *verdadero* o *falso*. Para representar en un computador[1] el valor *verdadero* se utiliza el número `1` y para representar el valor *falso* se utiliza el número `0`. Así, «rd $<-$ (rs $<$ rt)» indica que en el registro `rd` se almacenará un `1` o un `0` dependiendo de si el resultado de la comparación «rs $<$ rt» es *verdadero* o *falso*, respectivamente.

El lenguaje máquina del MIPS32 dispone de una instrucción que justamente realiza una comparación del tipo *menor que* entre dos registros y carga un `1` o un `0` en un tercer registro dependiendo del resultado de la comparación (*verdadero* o *falso*, respectivamente). Ésta es la instrucción « **slt** » (*set if less than*) y presenta la siguiente sintaxis: « **slt** rd, rs, rt ».

De hecho, la instrucción « **slt** » es la única instrucción de comparación que proporciona el procesador MIPS32. Sin embargo, y al igual que ocurría con las instrucciones de salto condicional, el ensamblador proporciona una serie de pseudo-instrucciones que facilitan la programación en ensamblador. Éstas son: «**sge**» (poner a `1` si mayor o igual), «**sgt**» (poner a `1` si mayor), «**sle**» (poner a `1` si menor o igual), «**sne**» (poner a 1 si distinto), «**seq**» (poner a `1` si igual).

EJERCICIOS

Describe utilizando la nomenclatura introducida en esta sección el significado de las siguientes pseudo-instrucciones (utiliza «==» como operador de igualdad y «$\neq$» como operador de desigualdad):

▶ **84** «**sge** rd, rs, rt»

▶ **85** «**sne** rd, rs, rt»

▶ **86** «**seq** rd, rs, rt»

5.3. Evaluación de condiciones

Las siguientes secciones muestran cómo se pueden evaluar condiciones utilizando las instrucciones proporcionadas por el MIPS32. En la primera sección se muestran ejemplos de cómo implementar la evaluación de condiciones

[1] Un computador interpretará un `0` como el valor booleano *falso* y cualquier valor distinto de `0` como el valor booleano *verdadero*.

simples: aquellas en las que se realiza una única comparación o función lógica de dos valores. En la segunda sección se muestra cómo implementar la evaluación de condiciones compuestas: aquellas que están formadas por varias comparaciones y funciones lógicas.

5.3.1. Evaluación de condiciones simples

Sea el siguiente código que compara dos variables, «dato1» y «dato2» (con los valores `30` y `40` en el ejemplo) y almacena en la variable «res» el resultado de dicha comparación.

✎ comp-slt.s

```
1           .data                   # Zona de datos
2  dato1:   .word   30
3  dato2:   .word   40
4  res:     .space  1
5
6           .text                   # Zona de instrucciones
7  main:    lw   $t0, dato1($0)     # Carga M[dato1] en $t0
8           lw   $t1, dato2($0)     # Carga M[dato2] en $t1
9           slt  $t2, $t0, $t1      # $t2 <- (t0<t1) ? 1 : 0
10          sb   $t2, res($0)       # Almacena $t2 en M[res]
```

EJERCICIOS

▶ **87** Carga y ejecuta el programa anterior. ¿Qué valor se almacena en la posición de memoria «res»?

▶ **88** Inicializa las posiciones de memoria «dato1» y «dato2» con los valores `50` y `20`, respectivamente. Para ello, sigue los siguientes pasos:

1. Pulsa el botón `set value` del simulador.
2. Introduce «0x10010000» (sin las comillas) en el campo de texto *register/location* y «50» (también sin comillas) en el campo de texto *value* y pulsa el botón `set`. (Observa como en la posición de memoria `0x1001 0000` ahora aparece el valor `0x32`, que es `50` en hexadecimal.)
3. Vuelve a pulsar el botón `set value` del simulador.
4. Introduce «0x10010004» en el campo de texto *register/location* y «20» en el campo de texto *value* y pulsa el botón `set`. (Observa que en la posición de memoria `0x1001 0004` ahora aparece el valor `0x14`, que es `20` en hexadecimal.)

▶ **89** Ejecuta de nuevo el programa utilizando el botón `run` (recuerda reiniciar la dirección de inicio —*starting address*— al valor `0x0040 0000` antes de pulsar `ok`). ¿Qué valor se carga ahora en la posición de memoria «res»?

▶ **90** ¿Qué condición se ha evaluado?

▶ **91** Modifica el código anterior para que evalúe la siguiente condición: $res \leftarrow (dato1 \geq dato2)$. Utiliza la pseudo-instrucción que implementa dicha comparación.

▶ **92** Resuelve el ejercicio anterior pero esta vez utilizando la instrucción «**sle**».

5.3.2. Evaluación de condiciones compuestas

Una condición compuesta está formada por varias comparaciones y operadores lógicos. Por ejemplo, para averiguar si una persona está entre los 20 y los 40 años se podría evaluar la siguiente condición compuesta[2]. Dicha condición compuesta está formada por dos comparaciones unidas por el operador lógico «y».

En esta sección se muestran varios programas en ensamblador que evalúan condiciones compuestas. Se verá en primer lugar el siguiente programa en ensamblador que evalúa una condición compuesta en la que intervienen los datos almacenados en «dato1» y «dato2»:

✎ comp-compuesta.s

```
1           .data                   # Zona de datos
2  dato1:   .word   40
3  dato2:   .word  -50
4  res:     .space  1
5
6           .text                   # Zona de instrucciones
7  main:    lw   $t8, dato1($0)     # Carga M[dato1] en $t8
8           lw   $t9, dato2($0)     # Carga M[dato2] en $t9
9           and  $t0, $t0, $0
10          and  $t1, $t1, $0
11          beq  $t8, $0, igual     # Salta a igual si $t8==0
12          ori  $t0, $0, 1
13 igual:   beq  $t9, $0, fineval   # Salta a fineval si $t9==0
14          ori  $t1, $0, 1
15 fineval: and  $t2, $t0, $t1
16          sb   $t2, res($0)
```

EJERCICIOS

▶ **93** Carga el programa anterior en el simulador y ejecútalo. ¿Qué valor se almacena en la posición de memoria referenciada por la etiqueta «res»?

[2] En lenguaje C la condición se podría expresar como: «anyos>=20 && anyos<=40»: *edad mayor o igual que 20 y edad menor o igual que 40*

▶ 94 Para cada par de valores de los mostrados en la siguiente tabla realiza las siguientes operaciones. *(i)* Modifica los valores almacenados en las posiciones de memoria referenciadas por las etiquetas «dato1» y «dato2» con los valores indicados (sigue un procedimiento similar al descrito en el ejercicio 88). *(ii)* Ejecuta el programa de nuevo (modificando la dirección de comienzo en caso necesario). *(iii)* Anota el valor almacenado en la posición de memoria referenciada por la etiqueta «res».

*dato*1	*dato*2	*res*
10	20	
20	0	
0	0	

Analiza ahora cómo funciona el código contestando a las siguientes preguntas.

▶ 95 ¿Para qué sirve la instrucción «**and** $t0, $t0, $0»? ¿Y la instrucción «**and** $t1, $t1, $0»?

▶ 96 ¿Qué es lo que hace la instrucción «**ori** $t0, $0, 1»? ¿Qué condición debe cumplirse para que se ejecute?

▶ 97 ¿Qué es lo que hace la instrucción «**ori** $t1, $0, 1»? ¿Qué condición debe cumplirse para que se ejecute?

▶ 98 ¿Qué es lo que hace la instrucción «**and** $t2, $t0, $t1»?

▶ 99 A la vista de las respuestas a las 4 preguntas anteriores, ¿qué condición compuesta está evalúando el programa anterior? Comprueba que la expresión deducida es correcta viendo si se cumple la tabla rellenada en el ejercicio 94.

▶ 100 Por último, modifica el código anterior para que la condición evaluada sea:

$$res \leftarrow (dato1 \neq 0) \text{ and } (dato1 \neq dato2).$$

Sea el siguiente programa que implementa una condicional compuesta distinta a la del ejemplo anterior:

✎ comp-compuesta2.s

```
1           .data                   # Zona de datos
2   dato1:  .word   30
3   dato2:  .word   -50
4   res:    .space  1
5
6           .text                   # Zona de instrucciones
7   main:   lw  $t8, dato1($0)      # Cargan dato1 y dato2 en $t8 y $t9
8           lw  $t9, dato2($0)
9           and $t0, $t0, $0        # Inicializa $t0 a cero
10          and $t1, $t1, $0        # Inicializa $t1 a cero
11          beq $t8, $0, igual
```

```
12          ori $t0, $0, 1
13  igual:  slt $t1, $t9, $t8
14  fineval: and $t0, $t0, $t1
15          sb  $t0, res($0)
```

EJERCICIOS

▶ **101** Comenta adecuadamente el código para entender qué es lo que hace. ¿Cuál es la condición compuesta evaluada?

▶ **102** Inicializa la memoria y registros del simulador, carga el código y ejecútalo. ¿Qué valor se almacena en la posición de memoria «res»?

▶ **103** Sobrescribe las posiciones de memoria «dato1» y «dato2» con los valores `10` y `20` (sigue un procedimiento similar al descrito en el ejercicio 88) y ejecuta de nuevo el código (teniendo en cuenta que la dirección de comienzo ha de ser la `0x0040 0000`).

¿Qué valor se almacena ahora en la posición de memoria «res»?

▶ **104** Sobrescribe las posiciones de memoria «dato1» y «dato2» con los valores `0` y `-20` y vuelve a ejecutarlo ¿Qué valor se ha almacenado ahora en la posición de memoria «res»?

▶ **105** ¿Coinciden los resultados obtenidos en las tres preguntas anteriores con el resultado de evaluar la condición compuesta que encontraste en el ejercicio 101?

▶ **106** Por último, modifica el código anterior para que la condición evaluada sea:

$$res \leftarrow ((dato1 \neq dato2) \text{ and } (dato1 \leq dato2)).$$

Sea, por último, el siguiente programa que implementa una condicional compuesta distinta a la de los ejemplos anteriores:

✎ comp-compuesta3.s

```
1           .data                   # Zona de datos
2   dato1:  .word   30
3   dato2:  .word   -20
4   res:    .space  1
5
6           .text                   # Zona de instrucciones
7   main:   lw  $t8, dato1($0)      # Cargan dato1 y dato2 en $t8 y $t9
8           lw  $t9, dato2($0)
9           and $t0, $t0, $0        # Inicializa $t0 a cero
10          and $t1, $t1, $0        # Inicializa $t1 a cero
11          slt $t0, $t8, $t9
12          bne $t9, $0, fineval
13          ori $t1, $0, 1
14  fineval: or $t0, $t0, $t1
15          sb $t0,res($0)
```

EJERCICIOS

▶ **107** Comenta adecuadamente el código para entender qué es lo que hace. ¿Cuál es la condición compuesta evaluada?

▶ **108** Inicializa la memoria y registros del simulador, carga el código y ejecútalo. ¿Qué valor se almacena en la posición de memoria «res»?

▶ **109** Sobrescribe las posiciones de memoria «dato1» y «dato2» con los valores `-20` y `10` (sigue un procedimiento similar al descrito en el ejercicio 88) y ejecuta de nuevo el código (teniendo en cuenta que la dirección de comienzo ha de ser la `0x0040 0000`).

¿Qué valor se almacena ahora en la posición de memoria «res»?

▶ **110** Sobrescribe las posiciones de memoria «dato1» y «dato2» con los valores `10` y `0` y vuelve a ejecutarlo ¿Qué valor se ha almacenado ahora en la posición de memoria «res»?

▶ **111** Sobrescribe las posiciones de memoria «dato1» y «dato2» con los valores `20` y `10` y vuelve a ejecutarlo ¿Qué valor se ha almacenado ahora en la posición de memoria «res»?

▶ **112** ¿Coinciden los resultados obtenidos en las cuatro preguntas anteriores con el resultado de evaluar la condición compuesta que encontraste en el ejercicio 107?

▶ **113** Por último, modifica el código anterior para que la condición evaluada sea:

$$res \leftarrow ((dato1 \leq dato2) \text{ or } (dato1 \leq 0)).$$

5.4. Problemas del capítulo

EJERCICIOS

▶ **114** Desarrolla un programa en ensamblador que defina un vector de booleanos, V, implementado mediante un vector de bytes. Los valores que pueden tomar los elementos del vector serán `0` para representar el valor booleano *falso* y `1` para representar el valor booleano *verdadero*. Los valores iniciales deberán ser tales que $V = [0, 1, 1, 1, 0]$.

Reserva espacio también para un vector de boolenos, res, con 3 elementos. El objetivo del programa es el de escribir en el vector res el resultado de las siguientes operaciones:

- $res[0] = V[0] \text{ and } V[4]$.

- $res[1] = V[1]$ or $V[3]$.
- $res[2] = (V[0]$ or $V[1])$ and $V[2]$.

▶ **115** Desarrolla un programa en ensamblador que defina e inicialice un vector de enteros, $V = [2, -4, -6]$. El objetivo del programa es el de almacenar en un vector de booleanos res, de tres elementos, el resultado de las siguientes comparaciones $V[i] >= 0,\ \forall i \in [0..2]$.

▶ **116** Diseña un programa en ensamblador que defina e inicialice un vector de enteros, $V = [1, -4, -5, 2]$ y obtenga como resultado una variable booleana, $resultado$, que tomará el valor `1` si todos los elementos del vector V son menores que cero (`0` en caso contrario).

CAPÍTULO 6

ESTRUCTURAS DE CONTROL CONDICIONALES Y DE REPETICIÓN

En el capítulo anterior se presentaron las instrucciones de salto condicional y las instrucciones de comparación. También se comentó en dicho capítulo que estas instrucciones sirven para la implementación de las estructuras de control condicionales y de repetición. La importancia de dichas estructuras de control radica en que permiten el desarrollo de programas más elaborados que los que podrían realizarse si tan sólo se contara con la ejecución secuencial.

Este capítulo muestra cómo implementar, utilizando las instrucciones presentadas en el capítulo anterior, las estructuras de control condicionales y de repetición encontradas habitualmente en los lenguajes de alto nivel. En concreto, las estructuras que serán objeto de estudio son las siguientes (del lenguaje C): «**if**», «**if** − **else**», «**while**» y «**for**».

6.1. Estructuras de control condicionales

En esta sección se tratan las estructuras condicionales «**if**» e «**if** − **else**». La primera de ellas evalúa una condición y si se cumple, ejecuta un determinado código; en caso contrario, no hace nada. La segunda estructura de control condicional, «**if** − **else**», «**if** − **else**», permite la ejecución de una porción u otra de código en función de si la condición evaluada es *verdadera* o *falsa*.

6.1.1. Estructura de control condicional `if`

El primer ejemplo que se va a ver corresponde a un pequeño programa que utiliza una estructura de control «**if**». El programa intenta evitar que se realice una división por cero (lo que generaría una excepción). Para ello, comprueba si el divisor es distinto de cero antes de realizar la división, y en caso contrario, simplemente devuelve un `0` como resultado. En concreto, el programa propuesto almacena en la variable «res» el cociente de la división entera de dos números, «dato1» y «dato2», o un `0` si «dato2» es igual a `0`. Una versión en lenguaje C de dicho programa sería la siguiente:

```c
int main(int argc, char** argv) {
  int dato1=40;
  int dato2=30;
  int res;

  res=0;
  if (dato2!=0) {
    res=dato1/dato2;
  }
}
```

Una versión en ensamblador del MIPS32 del anterior programa en C podría ser la siguiente:

✎ control-if.s

```
            .data
dato1:      .word   40
dato2:      .word   30
res:        .space  4

            .text
main:       lw    $t1, dato1($0)    # Carga dato1 en $t1
            lw    $t2, dato2($0)    # Carga dato2 en $t2
            and   $t0, $0, $0       # $t0 <- 0
si:         beq   $t2, $0, finsi    # Si $t2 == 0 salta a finsi
entonces:   div   $t1, $t2          # $t1/$t2
            mflo  $t0               # Almacena LO en $t0
finsi:      sw    $t0, res($0)      # Almacena $t0 en res
```


EJERCICIOS

▶ **117** Identifica en el programa en ensamblador la instrucción que evalúa la condición y controla el flujo del programa. Compárala con la condición del programa en lenguaje C. ¿Qué condición evalúa dicha instrucción en el programa en ensamblador? ¿Qué condición evalúa la instrucción «**if**» del programa en C? ¿Qué relación hay entre ambas condicionales?

▶ **118** Identifica en el programa ensamblador las instrucciones que corresponden a la estructura «**if**» completa (la instrucción que evalúa la condición, más las instrucciones que se ejecutan en caso de cumplirse la condición del «**if**» original).

▶ **119** Carga y ejecuta el programa. ¿Qué valor se almacena en la variable «res» después de su ejecución?

▶ **120** Reemplaza el contenido de «dato2» por un `0`. ¿Qué valor se almacena en la variable «res» después de ejecutar de nuevo el programa?

▶ **121** El siguiente programa escrito en C es una versión del programa anterior en el que sólo realiza la división si el divisor es un número positivo. Implementa dicha versión en ensamblador.

```
1  int main(int argc, char** argv) {
2    int dato1=40;
3    int dato2=30;
4    int res;
5
6    res=0;
7    if (dato2>0) {
8      res=dato1/dato2;
9    }
10 }
```

. . . . . .

Los programas anteriores evaluan condiciones simples: el primero evalúa si «dato2!=0» y el segundo, si «dato2>0». ¿Cómo se debería implementar la estructura condicional «**if**» si la condición fuera una condición compuesta?

Veamos un ejemplo. Sea el siguiente programa en C que indica si un determinado número está dentro de un rango definido por otros dos poniendo a `1` la variable «res» en caso afirmativo y a `0` en caso contrario. Una posible aplicación sería la de comprobar si la coordenada x de un punto gráfico, (x, y), se encuentra dentro del rango permitido por una pantalla de p.e, 800×600; es decir, en este caso se querría saber si $0 \leq x < 800$.

```
1  int main(int argc, char** argv) {
2    int min=0;
3    int max=800;
4    int x=30;
5    int res;
6
7    res=0;
8    if ((x>=min) && (x<max)) {
9      res=1;
10   }
11 }
```

Una versión en ensamblador del MIPS32 del anterior programa en C podría ser la siguiente:

✎ control-if2.s

```
1          .data
2  min:    .word  0
3  max:    .word  800
4  x:      .word  30
5  res:    .space 4
6
7          .text
8  main:   lw   $t1, min($0)      # Carga min en $t1
```

```
9           lw   $t2, max($0)      # Carga max en $t2
10          lw   $t3, x($0)        # Carga x   en $t3
11          and  $t0, $0, $0       # $t0 <- 0
12  si:     blt  $t3, $t1, finsi   # Si $t3<$t1 salta a finsi
13          bge  $t3, $t2, finsi   # Si $t3>=$t2 salta a finsi
14  entonces: ori  $t0, $0, 1      # $t0 <- 1
15  finsi:  sw   $t0, res($0)      # Almacena $t0 en res
```

EJERCICIOS

▶ **122** Identifica en el programa en ensamblador la instrucción o instrucciones que evalúan la condición y controlan el flujo del programa. Compárala con la condición del programa en lenguaje C. ¿Qué condición evalúan la o las instrucciones en el programa en ensamblador? ¿Qué condición evalúa la instrucción «**if**» del programa en C? ¿Qué relación hay entre éstas?

▶ **123** Identifica en el programa ensamblador las instrucciones que corresponden a la estructura «**if**» completa (la instrucción o instrucciones que evalúan la condición, más las instrucciones que se ejecutan en caso de cumplirse la condición del «**if**» original).

▶ **124** Carga y ejecuta el programa. ¿Qué valor se almacena en la variable «res» después de su ejecución?

▶ **125** Reemplaza el contenido de «x» por un `-10`. ¿Qué valor se almacena en la variable «res» después de ejecutar de nuevo el programa?

▶ **126** El siguiente programa escrito en C también comprueba si x pertenece a un rango, pero en este caso, en lugar de evaluar si $0 \leq x < 800$, evalúa si $0 \leq x \leq 799$. Implementa dicho programa en ensamblador.

```
1  int main(int argc, char** argv) {
2    int min=0;
3    int max=799;
4    int x=30;
5    int res;
6
7    res=0;
8    if ((x>=min) && (x<=max)) {
9      res=1;
10   }
11 }
```

6.1.2. Estructura de control condicional `if - else`

Ahora se va a ver cómo se implementa en ensamblador la estructura de control condicional «**if** − **else**». Crea un fichero con el siguiente fragmento de código que contiene una estructura de control de dicho tipo.

control-ifelse.s

```
1           .data
2  dato1:   .word  30
3  dato2:   .word  40
4  res:     .space 4
5
6           .text
7  main:    lw   $t1, dato1($0)    # Carga dato1 en $t1
8           lw   $t2, dato2($0)    # Carga dato2 en $t2
9  si:      bge  $t1, $t2, sino    # Si $t1>=$t2 ir a sino
10 entonces: sw  $t1, res($0)      # Almacena $t1 en res
11          j    finsi             # Ir a finsi
12 sino:    sw   $t2, res($0)      # Almacena $t2 en res
13 finsi:
```

EJERCICIOS

▶ **127** ¿Qué hace el programa? Dicho de otra forma, ¿qué valor se almacena en «res» en función del contenido de las variables «dato1» y «dato2»?

▶ **128** Carga el fichero en el simulador y ejecuta el programa. ¿Qué valor se almacena en «res» después de ejecutarlo?

▶ **129** Modifica el contenido de la dirección de memoria «dato1» para que almacene el valor `67` en lugar del actual `30`. Ejecuta de nuevo el programa. ¿Qué valor se almacena ahora en «res»?

▶ **130** Identifica en el lenguaje máquina generado por el simulador el conjunto de instrucciones que implementan la pseudoinstrucción «**bge**». ¿Cuáles son?

▶ **131** Implementa en ensamblador el siguiente programa escrito en C que calcula el valor absoluto de la resta de dos números.

```
1  int main(int argc, char** argv) {
2    int dato1=30;
3    int dato2=40;
4    int res;
5
6    if (dato1>=dato2) {
7      res=dato1-dato2;
8    } else {
9      res=dato2-dato1;
10   }
11 }
```

El siguiente programa también utiliza una estructura de control «**if** − **else**» pero esta vez la condición evaluada es una condición compuesta.

control-ifelse2.s

```
1           .data
2  minima:  .word  16
3  maxima:  .word  65
4  edad:    .word  15
5  res:     .space 4
6
```

```
7           .text
8  main:    lw   $t1, minima($0)     # Carga minima en $t1
9           lw   $t2, maxima($0)     # Carga maxima en $t2
10          lw   $t3, edad($0)       # Carga edad en $t3
11 si:      blt  $t3, $t1, entonces  # Si $t3<$t1 ir a entonces
12          ble  $t3, $t2, sino      # Si $t3<=$t2 ir a sino
13 entonces: addi $t4, $0, 1         # $t4<-1
14          j    finsi               # Ir a finsi
15 sino:    and  $t4,$0,$0           # $t4<-0
16 finsi:   sw   $t4,res($0)         # Almacena $t4 en res
```

EJERCICIOS

▶ **132** ¿Qué crees que hace el programa? ¿Qué condicional compuesta es evaluada?

▶ **133** Carga y ejecuta el programa. ¿Qué valor se almacena en «res» después de ejecutar el programa?

▶ **134** Modifica la variable «edad» para que contenga el valor `28`, en lugar del actual `15`. Ejecuta de nuevo el programa, ¿qué valor se almacena ahora en «res»?

6.2. Estructuras de control repetitivas

En esta sección se presentan las estructuras repetitivas «**while**» y «**for**». La primera de ellas, «**while**», permite la ejecución reiterada de una serie de instrucciones mientras se cumpla una determinada condición. La segunda de ellas, «**for**», permite la ejecución reiterada de una serie de instrucciones un número predeterminado de veces.

6.2.1. Estructura de control repetitiva `while`

El siguiente programa en C utiliza una estructura de control «**while**» para descubrir el número de caracteres de una cadena dada (que termina con el carácter nulo).

```
1 int main(int argc, char** argv) {
2   char cadena[]="Hola, ¿cómo estás?";
3   int  n=0;
4
5   while (cadena[n]!=0) {
6     n=n+1;
7   }
8 }
```

Una versión en ensamblador del MIPS32 del anterior programa en C podría ser la siguiente:

control-while.s

```
1                .data
2  cadena:       .asciiz "hola, ¿cómo estás?"
3                .align  2
4  n:            .space  4
5
6                .text
7  main:         andi $t0, $t0, 0          # $t0<-0
8  mientras:     lb   $t1, cadena($t0)     # $t1<-M[cadena+$t0] (un byte)
9                beq  $t1, $0, finmientras # Si $t1==0 salta a finmientras
10               addi $t0, $t0, 1          # $t0<-$t0+1
11               j    mientras             # Saltar a mientras
12 finmientras:  sw   $t0, n($0)           # Almacena $t0 en n
```

EJERCICIOS

▶ **135** Ejecuta paso a paso el programa anterior y comprueba detenidamente la función de cada una de las instrucciones que constituyen el programa ensamblador.

▶ **136** ¿Qué valor se almacena en «n» después de ejecutar el programa? ¿Cuántos caracteres tiene la cadena? ¿Se ha contabilizado el carácter nulo?

6.2.2. Estructura de control repetitiva `for`

La estructura de control repetitiva «**for**» se diferencia de la «**while**» en que el número de iteraciones de la primera es conocido de antemano. Así pues, se hace necesario el uso de una variable que almacene el número de iteraciones que se van realizando para así poder compararlo con el número de iteraciones que deben realizarse. La variable que almacena dicha cuenta recibe el nombre de *contador*. Generalmente, un contador se suele inicializar con el valor `0` e incrementar de `1` en `1` hasta que alcanza el número total de iteraciones menos `1`.

Hay que tener en cuenta, no obstante, que también es posible realizar la cuenta en orden descendente. En este caso, el contador se iniciaría con el número total de iteraciones menos `1` y se decrementaría de `1` en `1` hasta llegar a `0`. Por último, en determinados casos, puede ser interesante que el decremento o incremento del contador se realice en cantidades distintas a la unidad.

De todas formas, el siguiente ejemplo se ceñirá al caso más común: aquel en el contador se inicializa a `0` y se incrementa de `1` en `1` hasta alcanzar el número total de iteraciones menos uno.

Así pues, el siguiente ejemplo muestra un programa en C que acumula los elementos de un vector «V» en una variable llamada «res». Para ello, utiliza una estructura condicional «**for**» y un contador llamado «i».

```
1 int main(int argc, char** argv) {
```

```
2   int V[]={6, 7, 8, 9, 10, 1};
3   int n=6;
4   int i;
5   int res;
6
7   res=0;
8   for (i=0; i<n; i++) {
9     res=res+V[i];
10  }
11 }
```

Una versión en ensamblador del MIPS32 del anterior programa en C podría ser la siguiente:

✎ control-for.s

```
1           .data
2 vector:   .word  6, 7, 8, 9, 10, 1
3 n:        .word  6
4 res:      .space 4
5
6           .text
7 main:     la   $t2, vector       # $t2 <- dirección de vector
8           and  $t3, $0, $0       # $t3 <- 0
9           and  $t0, $0, $0       # $t0 <- 0
10          lw   $t1, n($0)        # $t1 <- Mem[n]
11 para:    bge  $t0, $t1, finpara # Si $t0>=$t1 salta a finpara
12          lw   $t4, 0($t2)       # Carga elemento de vector en $t4
13          add  $t3, $t3, $t4     # Acumula los elementos del vector
14          addi $t2, $t2, 4       # Actualiza el puntero $t2 (<-$t2+4)
15          addi $t0, $t0, 1       # $t0 <- $t0+1
16          j    para              # Salta a para
17 finpara: sw   $t3, res($0)      # Almacena $t3 en res
```

EJERCICIOS

▶ **137** Ejecuta paso a paso el programa y comprueba detenidamente la función que realizan cada una de las instrucciones que componen el programa en ensamblador.

▶ **138** ¿En qué posición de memoria está la variable «res»?

▶ **139** ¿Qué valor se almacena en «res» después de ejecutar el código anterior?

▶ **140** ¿Qué registro se ha utilizado para actualizar el contador? ¿Qué valor posee al finalizar la ejecución dicho registro? ¿Cuántas veces se ha ejecutado el bucle que comienza en la instrucción etiquetada con «para»?

6.3. Problemas del capítulo

EJERCICIOS

▶ **141** Desarrolla un programa en ensamblador que en primer lugar almacene en memoria las siguientes variables enteras: «min1=2», «max1=10», «min2=50», «max2=70» y «num=34». Después, debe reservar 1 palabra para almacenar el resultado: variable «res».

Una vez definidas las variables anteriores, el programa deberá almacenar en la variable «res» un `1` si el valor de «num» está en alguno de los intervalos formados por «min1» y «max1» o «min2» y «max2». En caso contrario, deberá almacenar un `0` en «res».

▶ **142** Desarrolla un programa en ensamblador que dado un vector de enteros, obtenga como resultado cuántos elementos son iguales a cero. Este resultado se debe almacenar en una variable llamada « total ». El programa deberá inicializar los elementos del vector, «V», en memoria, así como una variable, «n», que almacenará el número de elementos que tiene el vector «V».

▶ **143** Desarrolla 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: «min» y «max». El programa deberá inicializar los elementos del vector, «V», una variable, «n», que almacenará el número de elementos que tiene el vector y dos variables, «min» y «max», donde se almacenarán los valores mínimo y máximo del rango. También deberá reservar espacio para la variable «res», en la que se almacenará el resultado.

▶ **144** Desarrolla 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 una cadena en memoria que finalice con el carácter nulo (un `0`). También deberá reservar espacio para la variable « resultado ».

CAPÍTULO 7

INTRODUCCIÓN A LA GESTIÓN DE SUBRUTINAS

Índice

7.1. Llamada y retorno de una subrutina **66**
7.2. Paso de parámetros **70**
7.3. Problemas del capítulo **77**

Una subrutina está formada por un conjunto de instrucciones separadas del programa principal, realiza una determinada tarea y puede ser invocada desde cualquier punto de un programa.

El uso de subrutinas permite dividir un problema largo y complejo en subproblemas más sencillos. La ventaja de esta división radica en la mayor facilidad con la que se puede escribir, depurar y probar cada una de los subproblemas por separado. Esto es, se puede desarrollar y probar una subrutina independientemente del resto del programa y posteriormente, una vez que se ha verificado que su comportamiento es el esperado, se puede integrar en el programa que la va a utilizar.

Otra ventaja de programar utilizando subrutinas es que si una misma tarea se realiza en varios puntos del programa, no es necesario escribir el mismo código una y otra vez a lo largo del programa. Si no fuera posible utilizar subrutinas se debería repetir la misma fracción de código en todas y cada una de las partes del programa en las que éste fuera necesario. No es sólo la ventaja de no tener que escribir varias veces el mismo código: ¿qué ocurriría si ese tro-

zo de código tuviera un error que fuera preciso corregir?, que sería necesario revisar todas las partes del programa en las que éste aparece para rectificar el mismo error. De igual forma, cualquier mejora de dicha parte del código implicaría revisar todas las partes del programa en las que aparece. Por el contrario, si se utiliza una subrutina y se detecta un error en ella o se quiere mejorar su implementación, basta con modificar su código; una sola vez. En resumen, otra de las ventajas que aporta utilizar subrutinas es la reducción del tiempo de implementación y depuración del código.

Por último, el uso de subrutinas tiene una ventaja aún mayor: subproblemas que aparecen con frecuencia en el desarrollo de ciertos programas pueden ser implementados como subrutinas y agruparse en bibliotecas (*libraries* en inglés[1]). Cuando un programador requiere resolver un determinado problema ya implementado, le basta con recurrir a una determinada biblioteca e invocar a la subrutina adecuada. Es decir, gracias a la agrupación de subrutinas en bibliotecas, el mismo código puede ser reutilizado en varios programas.

Desde el punto de vista que nos ocupa, el de los mecanismos que proporciona un procesador para realizar ciertas tareas de programación, el uso de subrutinas implica que se deben facilitar las siguientes acciones: la llamada a una subrutina; el paso de los parámetros con los que debe operar la subrutina; la devolución de los resultados; y la continuación de la ejecución del programa a partir de la siguiente instrucción a la que invocó a la subrutina.

En este capítulo se presentan los aspectos más básicos relacionados con el manejo de subrutinas en el ensamblador del MIPS32. Es decir, cómo llamar y retornar de una subrutina y cómo intercambiar información entre el programa invocado y la subrutina utilizando registros.

7.1. Llamada y retorno de una subrutina

El juego de instrucciones del MIPS32 dispone de dos instrucciones para gestionar la llamada y el retorno de una subrutina. Éstas son:

- «**jal** etiqueta» Se utiliza en el programa invocador para **llamar** a la subrutina que comienza en la dirección de memoria indicada por la etiqueta. La ejecución de esta instrucción conlleva las siguientes acciones:
  - Se almacena la dirección de memoria de la siguiente instrucción a la que contiene la instrucción «**jal**» en el registro `$ra`. Es decir, `$ra`←`PC`+`4`.

[1] La palabra inglesa *library* se suele traducir erróneamente en castellano como «librería»; una traducción más adecuada es la de «biblioteca».

- Se lleva el control del flujo del programa a la dirección indicada en el campo « etiqueta ». Es decir, se realiza un salto incondicional a la dirección especificada en « etiqueta » (`PC`←« etiqueta »).

- «**jr** $ra» Se utiliza en la subrutina para **retornar** al programa invocador. Esta instrucción realiza un salto incondicional a la dirección contenida en el registro `$ra` (*return address*). Es decir, `PC`←`$ra`.

La correcta utilización de estas dos instrucciones permite realizar de forma sencilla la llamada y el retorno desde una subrutina. En primer lugar, el programa invocador debe llamar a la subrutina utilizando la instrucción «**jal** etiqueta ». Esta instrucción almacena en `$ra` la dirección de vuelta y salta a la dirección indicada por la etiqueta. Por último, cuando finalice la subrutina, ésta debe ejecutar la instrucción «**jr** $ra» para retornar al programa que la invocó.

Esta forma de proceder será la correcta siempre que el contenido del registro `$ra` no sea modificado durante la ejecución de la subrutina. Este funcionamiento queda esquematizado en la Figura 7.1.

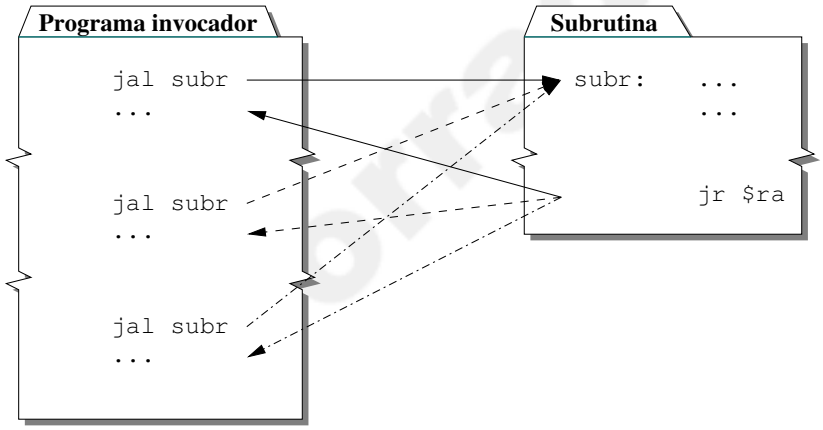


Figura 7.1: Llamada y retorno de una subrutina

Como primer ejemplo de subrutinas, el siguiente código muestra un programa que utiliza una subrutina llamada «suma». Esta subrutina suma los valores almacenados en los registros `$a0` y `$a1`, y devuelve la suma de ambos en el registro `$v0`. Como se puede observar, la subrutina se llama desde dos puntos del programa principal (la primera línea del programa principal está etiquetado como «main»).

✎ suma-dato1-dato2-v1.s

```
1           .data       # Zona de datos
2 dato1:    .word   1
```

```
3  dato2:    .word   3
4  dato3:    .word   5
5  dato4:    .word   4
6  res1:     .space  4
7  res2:     .space  4
8
9             .text          # Zona de código
10
11 # Subrutina
12
13 suma:      add $v0, $a0, $a1
14            jr  $ra
15
16 # Programa invocador
17
18 main:      lw  $a0, dato1($0)
19            lw  $a1, dato2($0)
20 primera:   jal suma
21            sw  $v0, res1($0)
22            lw  $a0, dato3($0)
23            lw  $a1, dato4($0)
24 segunda:   jal suma
25            sw  $v0, res2($0)
```

Carga el programa anterior en el simulador y contesta a las siguientes preguntas mientras realizas una ejecución paso a paso.

EJERCICIOS

▶ **145** ¿Cuál es el contenido del `PC` y del registro `$ra` antes y después de ejecutar la instrucción «**jal** suma» etiquetada como «primera»?

▶ **146** ¿Cuál es el contenido de los registros `PC` y `$ra` antes y después de ejecutar la instrucción «**jr** $ra» la primera vez que se ejecuta la subrutina «suma»?

▶ **147** ¿Cuál es el contenido del `PC` y del registro `$ra` antes y después de ejecutar la instrucción «**jal** suma» etiquetada como «segunda»?

▶ **148** ¿Cuál es el contenido de los registros `PC` y `$ra` antes y después de ejecutar la instrucción «**jr** $ra» la segunda vez que se ejecuta la subrutina «suma»?

▶ **149** El programa que aparece a continuación es idéntico al anterior salvo que se han modificado las posiciones de memoria que ocupan el programa invocador y la subrutina. En primer lugar se ha almacenado el programa invocador y a continuación la subrutina. Cárgalo en el simulador y ejecútalo paso a paso.

✎ `suma-dato1-dato2-v2.s`

```
1             .data          # Zona de datos
2  dato1:     .word   1
3  dato2:     .word   3
4  dato3:     .word   5
5  dato4:     .word   4
6  res1:      .space  4
```

```
7  res2:     .space  4
8
9                 .text   # Zona de código
10
11 # Programa invocador
12
13 main:     lw  $a0, dato1($0)
14          lw  $a1, dato2($0)
15 primera: jal suma
16          sw  $v0, res1($0)
17          lw  $a0, dato3($0)
18          lw  $a1, dato4($0)
19 segunda: jal suma
20          sw  $v0, res2($0)
21
22 # Subrutina
23
24 suma:     add $v0, $a0, $a1
25          jr  $ra
```

¿Qué ocurre cuando se ejecuta? Hazlo paso a paso y contesta a las siguientes preguntas:

- ¿Ha terminado el programa invocador después de ejecutar la instrucción «**sw** $v0, res2($0)»?
- ¿O por el contrario, la ejecución ha continuado con las instrucciones de la subrutina «suma»?
- ¿Ha entrado en un bucle sin fin?

. . . . . .

Para resolver el problema planteado en el último de los ejercicios anteriores es necesario incluir en el código del programa un par de instrucciones que le indican al monitor (un sistema operativo muy simple) del simulador SPIM que el programa ha finalizado y que la ejecución debe terminar.

El siguiente listado muestra un ejemplo en el que se han introducido dicho par de instrucciones al finalizar el programa principal. A partir de ahora, es conveniente que se utilicen dichas instrucciones en los programas que se desarrollen.

✎ suma-dato1-dato2-v3.s

```
1          .data          # Zona de datos
2  dato1:  .word   1
3  dato2:  .word   3
4  dato3:  .word   5
5  dato4:  .word   4
6  res1:   .space  4
7  res2:   .space  4
8
9          .text          # Zona de código
10
11 # Programa invocador
12
13 main:     lw  $a0, dato1($0)
```

```
14              lw   $a1, dato2($0)
15  primera:    jal  suma
16              sw   $v0, res1($0)
17              lw   $a0, dato3($0)
18              lw   $a1, dato4($0)
19  segunda:    jal  suma
20              sw   $v0, res2($0)
21
22              li $v0,10       # Finalizar la ejecución del programa
23              syscall
24
25  # Subrutina
26
27  suma:       add $v0, $a0, $a1
28              jr   $ra
```

EJERCICIOS

▶ **150** Carga el programa en el simulador y ejecútalo paso a paso. ¿Cuándo acaba ahora su ejecución? Ten cuidado al ir paso a paso ya que si después de ejecutar la última instrucción del programa, vuelves a pulsar el botón `step` el programa entero volverá a comenzar a ejecutarse.

7.2. Paso de parámetros

El mecanismo mediante el cual el programa invocador y la subrutina intercambian información, recibe el nombre de *paso de parámetros*.

Los parámetros intercambiados entre el programa invocador y la subrutina pueden ser de tres tipos según la dirección en la que se transmita la información: de *entrada*, de *salida* o de *entrada/salida*. Se denominan parámetros de entrada a los que proporcionan información del programa invocador a la subrutina. Se denominan parámetros de salida a los que devuelven información de la subrutina al programa invocador. Por último, los parámetros de *entrada/salida* proporcionan información del programa invocador a la subrutina y devuelven información de la subrutina al programa invocador.

Por otro lado, para realizar el paso de parámetros es necesario disponer de algún lugar físico donde se pueda almacenar y leer la información que se quiere transferir. Las dos opciones más comunes son: utilizar registros o utilizar la pila. Que se utilicen registros o la pila depende de la arquitectura en cuestión y del convenio que se siga para el paso de parámetros en dicha arquitectura.

Este capítulo se va a ocupar únicamente del paso de parámetros por medio de registros. De hecho, la arquitectura MIPS32 establece un convenio para el paso de parámetros mediante registros: para los parámetros de entrada se deben utilizar los registros `$a0`, `$a1`, `$a2` y `$a3`; y para los parámetros de salida se deben utilizar los registros `$v0` y `$v1`.

Hasta aquí se ha visto que hay parámetros de entrada, de salida y de entrada/salida. Además, también se ha visto que los parámetros pueden pasarse por medio de registros o de la pila. El último aspecto a considerar del paso de parámetros es cómo se transfiere cada uno de los parámetros. De hecho, hay dos formas de hacerlo. Un parámetro puede pasarse por valor o por referencia. Se dice que un parámetro se pasa por valor cuando lo que se transfiere es el dato en sí. Un parámetro se pasa por referencia cuando lo que se transfiere es la dirección de memoria en la que se encuentra dicho dato.

Según todo lo anterior, el desarrollo de una subrutina implica determinar en primer lugar:

- El número de parámetros necesarios.
- Cuáles son de entrada, cuáles de salida y cuáles de entrada/salida.
- Si se van a utilizar registros o la pila para su transferencia.
- Qué parámetros deben pasarse por valor y qué parámetros por referencia.

Naturalmente, el programa invocador deberá ajustarse a los requerimientos que haya fijado el desarrollador de la subrutina en cuanto a cómo se debe realizar el paso de parámetros.

En las siguientes secciones se utilizarán parámetros de entrada, salida o entrada/salida según sea necesario. Además, el paso de parámetros se hará utilizando los registros fijados por el convenio del MIPS32. La primera de las siguientes secciones muestra cómo se realiza el paso de parámetros por valor y la segunda, cómo se realiza el paso de parámetros por referencia.

7.2.1. Paso de parámetros por valor

El paso de parámetros por valor implica la siguiente secuencia de acciones (véase la Figura 7.2):

1. Antes de realizar la llamada a la subrutina, el programa invocador carga el valor de los parámetros de entrada en los registros elegidos para ello.
2. La subrutina, finalizadas las operaciones que deba realizar y antes de devolver el control al programa invocador, carga el valor de los parámetros de salida en los registros elegidos para ello.
3. El programa invocador recoge los parámetros de salida de los registros elegidos para ello.

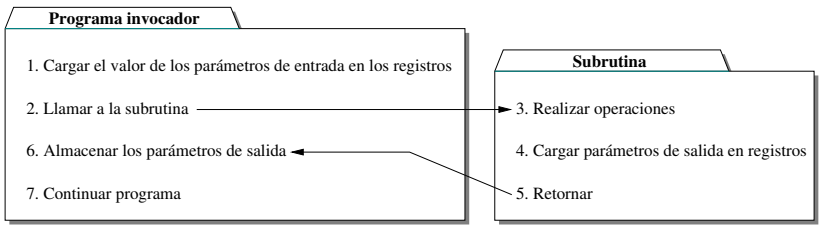


Figura 7.2: Paso de parámetros por valor

Como ejemplo de paso de parámetros por valor se presenta el siguiente código que ya se había visto con anterioridad. Éste muestra un ejemplo de paso de parámetros de entrada y de salida por valor.

✎ suma-dato1-dato2-v3.s

```
1           .data           # Zona de datos
2   dato1:  .word   1
3   dato2:  .word   3
4   dato3:  .word   5
5   dato4:  .word   4
6   res1:   .space  4
7   res2:   .space  4
8
9           .text           # Zona de código
10
11  # Programa invocador
12
13  main:   lw   $a0, dato1($0)
14          lw   $a1, dato2($0)
15  primera: jal suma
16          sw   $v0, res1($0)
17          lw   $a0, dato3($0)
18          lw   $a1, dato4($0)
19  segunda: jal suma
20          sw   $v0, res2($0)
21
22          li $v0,10       # Finalizar la ejecución del programa
23          syscall
24
25  # Subrutina
26
27  suma:   add $v0, $a0, $a1
28          jr   $ra
```

EJERCICIOS

▶ **151** Enumera los registros que se han utilizado para pasar los parámetros a la subrutina.

▶ **152** ¿Los anteriores registros se han utilizado para pasar parámetros de entrada o de salida?

▶ **153** Anota qué registro se ha utilizado para devolver el resultado al programa invocador.

▶ 154 ¿El anterior registro se ha utilizado para pasar un parámetro de entrada o de salida?

▶ 155 Señala qué instrucciones se corresponden a cada uno de las acciones enumeradas en la Figura 7.2.

. . . . . .

7.2.2. Paso de parámetros por referencia

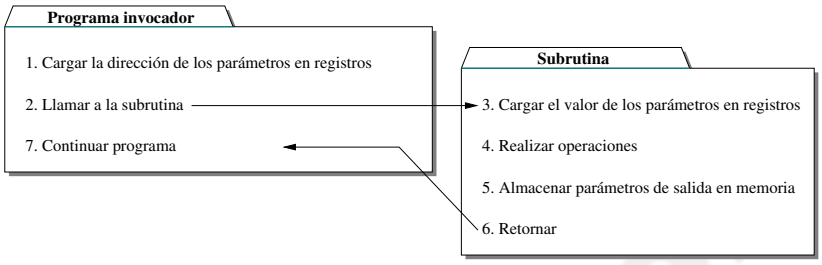


Figura 7.3: Paso de parámetros por referencia

En cuanto al paso de parámetros por referencia, éste implica la siguiente secuencia de acciones (véase la Figura 7.3):

- Antes de realizar la llamada a la subrutina, el programa invocador carga en los registros elegidos para realizar el paso de parámetros, las direcciones de memoria en las que está almacenada la información que se quiere pasar.
- La subrutina carga en registros el contenido de las direcciones de memoria indicadas por los parámetros de entrada (recuerda que el MIPS32 no puede operar directamente con datos en memoria). Y opera con ellos.
- La subrutina, una vez ha finalizado y antes de devolver el control al programa principal, almacena los resultados en las direcciones de memoria que le había proporcionado el programa invocador.

El siguiente programa muestra un ejemplo en el que se llama a una subrutina utilizando el paso de parámetros por referencia tanto para los parámetros de entrada como los de salida.

✎ suma-dato1-dato2-v4.s

```
1           .data                   # Zona de datos
2 dato1:    .word   1
3 dato2:    .word   3
4 dato3:    .word   5
5 dato4:    .word   4
6 res1:     .space  4
```

```
7  res2:     .space  4
8
9             .text                   # Zona de código
10
11 # Programa invocador
12
13 main:      la   $a0, dato1
14            la   $a1, dato2
15            la   $a2, res1
16 primera:   jal  suma
17            la   $a0, dato3
18            la   $a1, dato4
19            la   $a2, res2
20 segunda:   jal  suma
21
22            li   $v0, 10            # Finalizar la ejecución del programa
23            syscall
24
25
26 # Subrutina
27
28 suma:      lw   $t0, 0($a0)
29            lw   $t1, 0($a1)
30            add  $v0, $t0, $t1
31            sw   $v0, 0($a2)
32            jr   $ra
```

EJERCICIOS

▶ **156** Enumera los registros que se han utilizado para pasar la dirección de los parámetros de entrada a la subrutina.

▶ **157** Anota qué registro se ha utilizado para pasar la dirección del parámetro de salida a la subrutina.

▶ **158** Señala qué instrucciones se corresponden con cada una de las acciones enumeradas en la Figura 7.3.

7.2.3. Paso de parámetros, ¿por valor o por referencia?

Una vez descritas las dos formas de paso de parámetros a una subrutina, queda la tarea de decidir cuál de las formas, por referencia o por valor, es más conveniente en cada caso. Los ejemplos anteriores eran un poco artificiales ya que todos los parámetros se pasaban o bien por valor o por referencia. En la práctica, se debe analizar para cada parámetro cuál es la forma idónea de realizar el paso. Es decir, generalmente, no todos los parámetros de una subrutina utilizarán la misma forma de paso.

De hecho, la decisión última de cómo se pasan los parámetros a una subrutina depende en gran medida del tipo de información que se quiera pasar. Si se trata de un dato de tipo estructurado (vectores, matrices, cadenas, estructuras...), el paso siempre se hará por referencia. Tanto si el parámetro en cuestión es de entrada, de salida o de entrada/salida.

En cambio, si el dato que se quiere pasar es de tipo escalar (un número entero, un número real, un carácter...), éste se puede pasar por valor o por referencia. Esta última decisión depende de la utilización que se le vaya a dar al parámetro: es decir, si se trata de un parámetro de entrada, de salida o de entrada/salida. La siguiente lista muestra las opciones disponibles según el tipo de parámetro:

- Parámetro de entrada. Un parámetro de este tipo es utilizado por la subrutina pero no debería ser modificado: es preferible pasar este tipo de parámetros por valor.

- Parámetro de salida. Un parámetro de este tipo permite a la subrutina devolver el resultado de las operaciones realizadas. Para este tipo de parámetros se puede optar por cualquiera de las opciones: que el parámetro sea devuelto por valor o por referencia.

    - Por valor: la subrutina devuelve el dato utilizando el registro reservado para ello.
    - Por referencia: la subrutina almacena en memoria el dato. La dirección de memoria la obtiene la subrutina del registro que había sido utilizado para ello.

- Parámetro de entrada/salida. Un parámetro de este tipo proporciona un valor que la subrutina necesita conocer pero en el que posteriormente devolverá el resultado. En este caso, es preferible pasarlo por referencia.

7.2.4. Un ejemplo más elaborado

A continuación se plantea el desarrollo de un programa en lenguaje ensamblador donde se aplican todos los conceptos presentados en este capítulo.

Dicho programa tiene por objeto calcular cuántos elementos de un vector dado tienen el valor `12`. El programa consta de una subrutina que devuelve el número de elementos de un vector que son menores a uno dado. Llamando dos veces a dicha subrutina con los valores `12` y `13` y realizando una simple resta, el programa será capaz de determinar el número de elementos que son iguales a `12`.

Se debe desarrollar dicho programa paso a paso. En primer lugar, se debe desarrollar una subrutina que contabilice cuántos elementos de un vector son menores a un valor dado. Para ello, hay que determinar qué parámetros debe recibir dicha subrutina, así como qué registros se van a utilizar para ello, y cuáles se pasarán por referencia y cuáles por valor.

Una vez desarrollada la subrutina y teniendo en cuenta los parámetros que requiere, se deberá desarrollar el programa que llame a dicha subrutina y obtenga el número de elementos del vector dado que son iguales a `12`.

Una descripción detallada del funcionamiento que debiera tener dicho programa se muestra en el siguiente listado en lenguaje C:

```
1  int nummenorque(int* vector_s, int dim_s, int dato_s)
2  {
3    int i; /* Índice    */
4    int n; /* Contador */
5
6    n = 0;    /* Contador a 0 */
7    for (i=0; i<dim_s; i++){
8      if (vector_s[i] < dato_s) {
9        n = n + 1;
10     }
11   }
12   return n;
13 }
14
15 int main(int argc, char** argv)
16 {
17   int vector[9]={5, 3, 5, 5, 8, 12, 12, 15, 12};
18   int dim=9;
19   int num=12;
20   int res1, res2, res;
21
22   res1 = nummenorque(vector, dim, num);
23   num = num+1;
24   res2 = nummenorque(vector, dim, num);
25   res = res2-res1;
26 }
```

EJERCICIOS

▶ **159** Conforme desarrollado el código de la subrutina, contesta las siguientes preguntas:

a) ¿Qué parámetros debe pasar el programa invocador a la subrutina? ¿Y la subrutina al programa invocador?

b) ¿Cuáles son de entrada y cuáles de salida?

c) ¿Cuáles se pasan por referencia y cuáles por valor?

d) ¿Qué registros se van a utilizar para cada uno de ellos?

▶ **160** Completa el desarrollo del fragmento de código correspondiente a la subrutina.

▶ **161** Desarrolla el fragmento de código correspondiente al programa invocador.

▶ **162** Comprueba que el programa escrito funciona correctamente. ¿Qué valor se almacena en «res» cuando se ejecuta? Modifica el contenido del vector «vector», ejecuta de nuevo el programa y comprueba que el resultado obtenido es correcto.

7.3. Problemas del capítulo

Para la realización de los siguientes problemas se deberá consultar el Apéndice B en el que se describen las llamadas a los servicios que proporciona el simulador SPIM. El primero de los problemas propuestos permitirá practicar el uso de dichos servicios.

▶ **163** Desarrolla un programa que lea números enteros por teclado hasta que el usuario introduzca el número `0`. Una vez el usuario ha introducido el número `0`, el programa deberá mostrar por pantalla cuántos números se han introducido y su suma. Un ejemplo de ejecución de dicho programa sería el siguiente:

```
# Introduce un número (0 para terminar):
4
# Introduce un número (0 para terminar):
3
# Introduce un número (0 para terminar):
0
Has introducido 2 números que suman 7.
```

▶ **164** Realiza el siguiente ejercicio:

a) Desarrolla una subrutina que calcule cuántos elementos de un vector de enteros son pares (múltiplos de 2). La subrutina debe recibir como parámetros el vector y su dimensión y devolver el número de elementos pares.

b) Implementa un programa en el que se inicialicen dos vectores de 10 elementos cada uno, «vector1» y «vector2», y que almacene en sendas variables, «numpares1» y «numpares2», el número de elementos pares de cada uno de ellos. Además, el programa deberá mostrar por pantalla el número de elementos pares de ambos vectores. Naturalmente, el programa deberá utilizar la subrutina desarrollada en el apartado a) de este ejercicio.

▶ **165** Realiza el siguiente ejercicio:

a) Implementa una subrutina que calcule la longitud de una cadena de caracteres que finalice con el carácter nulo.

b) Implementa un programa que solicite dos cadenas de caracteres por el teclado y calcule cuál de las dos cadenas es más larga. Utiliza la subrutina desarrollada en el apartado a) de este ejercicio.

▶ **166** Realiza el siguiente ejercicio:

a) Desarrolla una subrutina que sume los elementos de un vector de enteros de cualquier dimensión.

b) Desarrolla un programa que sume todos los elementos de una matriz de dimensión $m \times n$. Utiliza la subrutina desarrollada en el apartado a) de este ejercicio para sumar los elementos de cada fila de la matriz.

En la versión que se implemente de este programa utiliza una matriz con $m = 5$ y $n = 3$, es decir, de dimensión 5×3 con valores aleatorios (los que se te vayan ocurriendo sobre la marcha). Se debe tener en cuenta que la matriz debería poder tener cualquier dimensión, así que se deberá utilizar un bucle para recorrer sus filas.

CAPÍTULO 8

GESTIÓN DE SUBRUTINAS

Índice

8.1. La pila **80**
8.2. Bloque de activación de la subrutina **84**
8.3. Problemas del capítulo **98**

Cuando se realiza una llamada a una subrutina (procedimiento o función) en un lenguaje de alto nivel, los detalles de cómo se cede el control a dicha rutina y la gestión de información que dicha cesión supone, quedan convenientemente ocultos.

Sin embargo, un compilador, o un programador en ensamblador, sí que debe explicitar todos los aspectos que conlleva la gestión de la llamada y ejecución de una subrutina. Algunos de dichos aspectos se trataron en el capítulo anterior. Como la transferencia de información entre el programa invocador y la subrutina y la devolución del control al programa invocador cuando finaliza la ejecución de la subrutina.

Otro aspecto relacionado con la llamada y ejecución de subrutinas, y que no se ha tratado todavía, es el de la gestión de la información requerida por la subrutina. Esta gestión abarca las siguientes tareas:

- El almacenamiento y posterior recuperación de la información almacenada en determinados registros.

- El almacenamiento y la recuperación de la dirección de retorno para permitir que una subrutina llame a otras subrutinas o a si misma (*recursividad*).
- La creación y utilización de variables temporales y locales de la subrutina.

Para poder llevar a cabo dichas tareas es necesario crear y gestionar un espacio de memoria donde la subrutina pueda almacenar la información que necesita durante su ejecución. A este espacio de memoria se le denomina *bloque de activación de la subrutina*. La gestión del bloque de activación de la subrutina constituye un tema central en la gestión de subrutinas.

Este capítulo muestra cómo crear y gestionar el bloque de activación de una subrutina. Puesto que el bloque de activación se implementa utilizando una estructura de datos conocida como *pila*, la primera parte del capítulo se dedica a mostrar cómo se utiliza una pila en ensamblador. Posteriormente, en la segunda parte, se muestra en qué consiste la construcción y gestión del bloque de activación.

8.1. La pila

Una *pila* o *cola LIFO* (*Last In First Out*) es una estructura de datos en la que el último dato almacenado es el primero en ser recuperado. Una analogía que se suele emplear para describir una pila es el de un montón de libros puestos uno sobre otro encima de una mesa. Para que dicha analogía sea correcta se debe limitar la forma en la que se pueden añadir o quitar libros de dicho montón. Cuando se quiera añadir un libro, éste deberá colocarse encima de los que ya haya (ésto es, no es posible insertar un libro entre los que ya estén en el montón). Cuando se quiera quitar un libro, sólo se podrá quitar el libro que está más arriba en el montón (ésto es, no se puede quitar un libro en particular si previamente no se han quitado todos los que están encima suyo). Así, el último libro añadido a la pila de libros será el primero en ser sacado de ella.

La acción de añadir elementos a una pila se denomina apilar (*push*, en inglés) y la de extraer elementos de una pila, desapilar (*pop*, en inglés).

Cuando se implementa una pila en la memoria de un computador es necesario, en primer lugar, decidir la dirección en la que va a crecer la pila y, en segundo lugar, disponer de un *puntero* que indique la dirección de memoria en la que se encuentra el último dato introducido en la pila. Esta dirección de memoria es lo que se conoce como el *tope de la pila*.

En cuanto a la dirección de crecimiento de la pila, se suele seguir el convenio de que la pila crezca de direcciones de memoria altas a direcciones de

memoria bajas. Siguiendo dicho convenio, los últimos elementos que se incorporen a la pila estarán en direcciones de memoria más bajas que los que ya se encontraban en ella. Por tanto, el tope de pila disminuirá al añadir elementos y aumentará al quitar elementos.

Por otro lado, para almacenar la dirección del tope de pila se suele utilizar un registro que recibe el nombre de *puntero de pila*. Así, cuando se quiera añadir elementos a la pila, se deberá decrementar el contenido del puntero de pila. De igual forma, cuando se quieran extraer elementos de la pila, se deberá incrementar el contenido del puntero de pila. El registro `$29` es el utilizado por MIPS como puntero de pila. Dicho registro también recibe el nombre de `$sp` (*stack pointer*).

A continuación se describen con detalle los pasos que se deben realizar para apilar y desapilar en el MIPS32.

La operación de apilar se realiza en dos pasos. En el primero de ellos se decrementa el puntero de pila en tantas posiciones como el tamaño de los datos que se desean apilar. En el segundo, se almacena el dato que se quiere apilar en la dirección indicada por el puntero de pila. Así por ejemplo, si se quisiera apilar la palabra que contiene el registro `$t0`, los pasos que se deberán realizar son: (i) decrementar el puntero de pila en 4 posiciones, «**addi** $sp, $sp, −4», y (ii) almacenar el contenido del registro `$t0` en la dirección indicada por `$sp`, «**sw** $t0, 0($sp)».

La Figura 8.1 muestra el contenido de la pila y el valor del registro `$sp` antes y después de apilar una palabra.

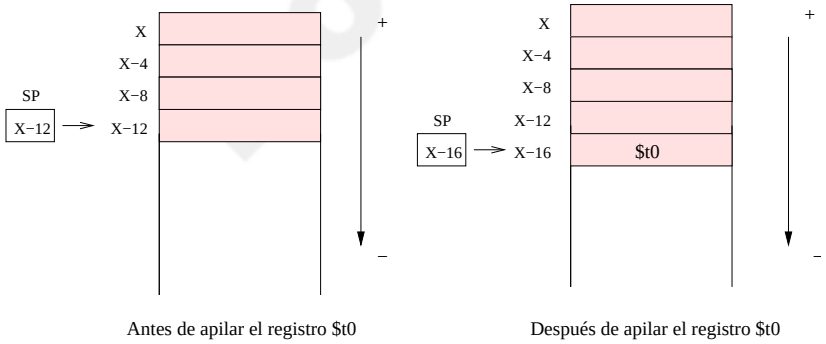


Figura 8.1: Apilar una palabra

La operación de desapilar también consta de dos pasos. En el primero de ellos se recupera el dato que está almacenado en la pila. En el segundo, se incrementa el puntero de pila en tantas posiciones como el tamaño del dato que se desea desapilar. Así por ejemplo, si se quisiera desapilar una palabra para cargarla en el registro `$t0`, los pasos que se deberán realizar son: (i) cargar el

dato que se encuentra en la posición indicada por el registro `$sp` en el registro `$t0`, «**lw** $t0, 0($sp)», y (ii) incrementar en 4 posiciones el puntero de pila, «**addi** $sp, $sp, 4».

La Figura 8.2 muestra el contenido de la pila y el valor del registro `$sp` antes y después de desapilar una palabra.

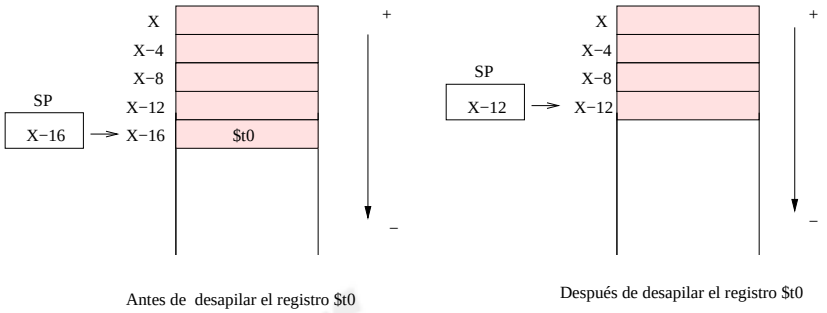


Figura 8.2: Desapilar una palabra

Realiza los siguientes ejercicios sobre la operación apilar.

EJERCICIOS

▶ **167** Suponiendo que el puntero de pila contiene el valor `0x7fffeffc` y que se desea apilar una palabra (4 bytes). ¿Qué valor deberá pasar a tener el puntero de pila antes de almacenar la nueva palabra en la pila? ¿Qué instrucción se utilizará para hacerlo en el ensamblador del MIPS32?

▶ **168** ¿Qué instrucción se utilizará para almacenar en la posición apuntada por el puntero de pila el contenido del registro `$t1`?

▶ **169** A partir de los dos ejercicios anteriores indica qué dos instrucciones permiten apilar en la pila el registro `$t1`.

El siguiente fragmento de código apila, uno detrás de otro, el contenido de los registros `$t0` y `$t1`:

✎ sub-apila-t0t1.s

```
1          .text                # Zona de instrucciones
2  main:   li   $t0, 10         # $t0<-10
3          li   $t1, 13         # $t1<-13
4          addi $sp, $sp, -4    # Actualiza $sp ($sp<-$sp-4)
5          sw   $t0, 0($sp)     # Apila $t0
6          addi $sp, $sp, -4    # Actualiza $sp ($sp<-$sp-4)
7          sw   $t1, 0($sp)     # Apila $t1
```

Crea el programa anterior, inicia el simulador, carga el programa y realiza los siguientes ejercicios:

EJERCICIOS

▶ **170** Ejecuta el programa paso a paso y comprueba en qué posiciones de memoria, pertenecientes al segmento de pila, se almacenan los contenidos de los registros `$t0` y `$t1`.

▶ **171** Modifica el programa anterior para que en lugar de actualizar el puntero de pila cada vez que se pretende apilar un registro, se realice una única actualización al principio del puntero de pila y, a continuación, se almacenen los registros `$t0` y `$t1`. Los registros deben quedar apilados en el mismo orden que en el programa original.

▶ **172** Realiza los cambios necesarios en el programa original para que apile únicamente el byte más bajo del registro `$t0` y los 4 bytes del registro `$t1`. Justifica la propuesta.

Realiza los siguientes ejercicios sobre la operación desapilar.

EJERCICIOS

▶ **173** ¿Qué instrucción se utilizará para desapilar el dato contenido en el tope de la pila y cargarlo en el registro `$t1`?

▶ **174** ¿Qué instrucción en ensamblador del MIPS32 se utilizará para actualizar el puntero de pila?

▶ **175** A partir de los dos ejercicios anteriores indica qué dos instrucciones permiten desapilar de la pila el registro `$t1`.

▶ **176** Suponiendo que el puntero de pila contiene el valor `0x7fffeff8` ¿Qué valor deberá pasar a tener el puntero de pila después de desapilar una palabra (4 bytes) de la pila?

EJERCICIOS DE AMPLIACIÓN

▶ **177** Desarrolla un programa que realice las siguientes acciones:

- Reserve espacio en memoria para almacenar una tira de caracteres de 10 elementos como máximo.
- Lea una cadena de caracteres desde el teclado.
- Invierta el contenido de esta cadena, almacenando la cadena resultante en las mismas posiciones de memoria que la original.

(Pista: Si se apilan elementos en la pila y luego se desapilan, se obtienen los mismos elementos pero en el orden inverso.)

8.2. Bloque de activación de la subrutina

El bloque de activación (BA) de una subrutina está formado por el segmento de pila que contiene la información manejada por una subrutina durante su ejecución. Dicho bloque tiene varios propósitos:

- En el caso de llamadas anidadas, almacenar la dirección de retorno.
- Proporcionar espacio para las variables locales de la subrutina.
- Almacenar los registros que la subrutina necesita modificar pero que el programa que ha hecho la llamada no espera que sean modificados.
- Mantener los valores que se han pasado como argumentos a la subrutina.

8.2.1. Anidamiento de subrutinas

El anidamiento de subrutinas tiene lugar cuando un programa invocador llama a una subrutina y ésta, a su vez, llama a otra, o a si misma. El que una subrutina pueda llamarse a si misma es lo que se conoce como recursividad.

Cuando se anidan subrutinas, el programa invocado se convierte en un momento dado en invocador, lo que obliga a ser cuidadoso con la gestión de las direcciones de retorno. Como se ha visto, la instrucción utilizada para llamar a una subrutina es la instrucción «**jal**». Dicha instrucción, antes de realizar el salto propiamente dicho, almacena la dirección de retorno del programa invocador en el registro `$ra`. Si durante la ejecución del programa invocado, éste llama a otro (o a si mismo) utilizando otra instrucción «**jal**», cuando se ejecute dicha instrucción, se almacenará en el registro `$ra` la nueva dirección de retorno. Por tanto, el contenido original del registro `$ra`, es decir, la dirección de retorno almacenada tras ejecutar el primer «**jal**», se perderá. Y si no se hiciera nada para evitar perder la anterior dirección de retorno, cuando se ejecuten las correspondientes instrucciones de vuelta, «**jr** $ra», se retornará siempre a la misma posición de memoria, la almacenada en el registro `$ra` por la última instrucción «**jal**».

El siguiente fragmento de código ilustra este problema realizando dos llamadas anidadas.

✎ subr-llamada.s

```
1           .data
2 A:        .word 1, 2, 3, 4, 5, 6
3 dim:      .word 6
4
5           .text
6 main:     lw   $a1, dim($0)   # Paso de parámetros para sum_total
7           beqz $a1, fin       # Ejecuta el programa si dim>0
```

```
8             la   $a0, A
9   salto1:   jal  sum_total      # Llama a la subrutina sum_total
10
11  fin:      li   $v0, 10        # Finaliza la ejecución
12            syscall
13
14  sum_total:# ...
15            add $a0, $t1, $a0   # Paso de parámetros para suma_elem
16            add $a1, $t2, $0
17  salto2:   jal suma_elem       # Llama a la subrutina suma_elem
18            # ...
19  salto4:   jr $ra
20
21
22  sum_elem: # ...
23  salto3:   jr $ra
```

Edita el programa anterior, cárgalo en el simulador y ejecútalo paso a paso.

EJERCICIOS

▶ **178** ¿Dónde se pasa el control del programa tras la ejecución de la instrucción etiquetada por « salto1 »? ¿Qué valor se carga en el registro `$ra` después de ejecutar la instrucción etiquetada por « salto1 »?

▶ **179** ¿Dónde se pasa el control del programa tras la ejecución de la instrucción etiquetada por « salto2 »? ¿Qué valor se carga en el registro `$ra` después de ejecutar la instrucción etiquetada por « salto2 »?

▶ **180** ¿Dónde se pasa el control del programa tras la ejecución de la instrucción etiquetada por « salto3 »? ¿Qué valor se carga en el registro `$ra` después de ejecutar la instrucción etiquetada por « salto3 »?

▶ **181** ¿Dónde se pasa el control del programa tras la ejecución de la instrucción etiquetada por « salto4 »? ¿Qué valor se carga en el registro `$ra` después de ejecutar la instrucción etiquetada por « salto4 »?

▶ **182** Explica qué ocurre en el programa.

El ejercicio anterior muestra que es necesario utilizar alguna estructura de datos que permita almacenar las distintas direcciones de retorno cuando se realizan llamadas anidadas.

Dicha estructura de datos debe satisfacer dos requisitos. En primer lugar, debe ser capaz de permitir recuperar las direcciones de retorno en orden inverso a su almacenamiento (ya que es el orden en el que se producen los retornos). En segundo lugar, el espacio reservado para este cometido debe poder crecer de forma dinámica (la mayoría de las veces no se conoce cuántas llamadas se van a producir, pueden depender de los datos del problema).

La estructura de datos que mejor se adapta a los anteriores requisitos es la pila. Para almacenar y recuperar las direcciones de retorno utilizando una

pila basta con proceder de la siguiente forma. Antes de realizar una llamada a otra subrutina (o a si misma), se deberá apilar la dirección de retorno actual. Y antes de retornar, se deberá desapilar la última dirección de retorno apilada.

Por tanto, el programa invocador deberá apilar el registro `$ra` antes de invocar al nuevo programa y desapilarlo antes de retornar. Es decir, en el caso de que se realicen llamadas anidadas, el contenido del registro `$ra` formará parte de la información que se tiene que apilar en el bloque de activación de la subrutina.

La Figura 8.3 muestra de forma esquemática que es lo que puede ocurrir si no se almacena la dirección de retorno. En dicha figura se muestra el contenido del registro `$ra` justo después de ejecutar una instrucción «**jal**» o «**jr**». Como se puede ver, a partir de la llamada a la subrutina S3, el registro `$ra` sólo mantiene almacenada la dirección de retorno del último «**jal**», « dir_ret2 ». Por tanto, todos los retornos que se produzcan a partir de la última llamada, rutina S3, retornarán a la posición « dir_ret2 ». El retorno de la rutina S3 será correcto, pero cuando se ejecute el retorno de la rutina S2 se volverá de nuevo a la posición « dir_ret2 », provocando la ejecución de un bucle infinito.

La Figura 8.4 muestra de forma esquemática que es lo que ocurre cuando sí que se almacena la dirección de retorno. En dicha figura se muestra cuando se debe apilar y desapilar el registro `$ra` en la pila para que los retornos se produzcan en el orden adecuado. Se puede observar el estado de la pila y del registro `$ra` justo después de ejecutar una instrucción «**jal**» o «**jr**».

EJERCICIOS

▶ **183** Modifica el fragmento de código anterior para que la dirección de retorno se apile y desapile de forma adecuada.

8.2.2. Variables locales de la subrutina

Una subrutina puede requerir durante su ejecución utilizar variables locales que sólo existirán mientras se está ejecutando. Dependiendo del tipo de variables, bastará con utilizar los registros no ocupados o será necesario utilizar la memoria principal. En el caso de que sea necesario utilizar la memoria principal, la variable deberá almacenarse en el bloque de activación de la subrutina.

En el caso de datos de tipo escalar, siempre que sea posible, se utilizarán los registros del procesador que no estén ya siendo utilizados.

Pero en el caso de los datos de tipo estructurado, se hace necesario el uso de memoria principal. Es decir, las variables de tipo estructurado formarán parte del bloque de activación de la subrutina.

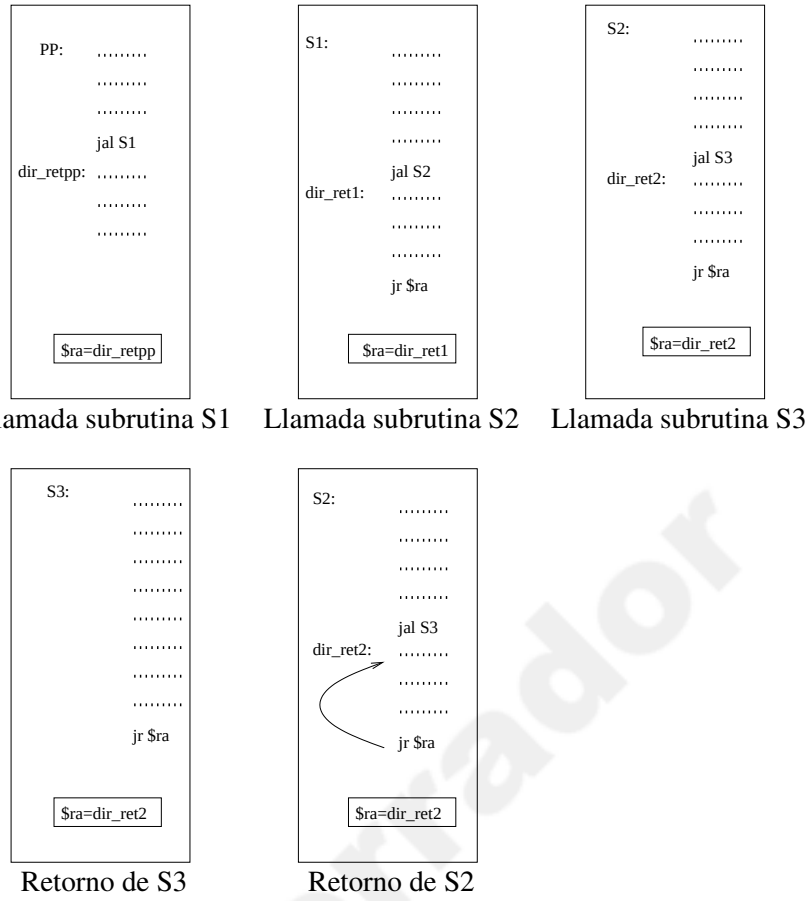


Figura 8.3: Llamadas anidadas a subrutinas (sin apilar las direcciones de retorno)

La forma de utilizar el bloque de activación para almacenar las variables locales de una subrutina es la siguiente. Al inicio de la subrutina se deberá reservar espacio en el bloque de activación para almacenar dichas variables. Y antes del retorno se deberá libera dicho espacio.

8.2.3. Almacenamiento de los registros utilizados por la subrutina

Como se ha visto en el apartado anterior, la subrutina puede utilizar registros como variables locales, y por tanto, el contenido original de dichos registros puede perderse en un momento dado. Si la información que contenían dichos registros es relevante para que el programa invocador pueda continuar su

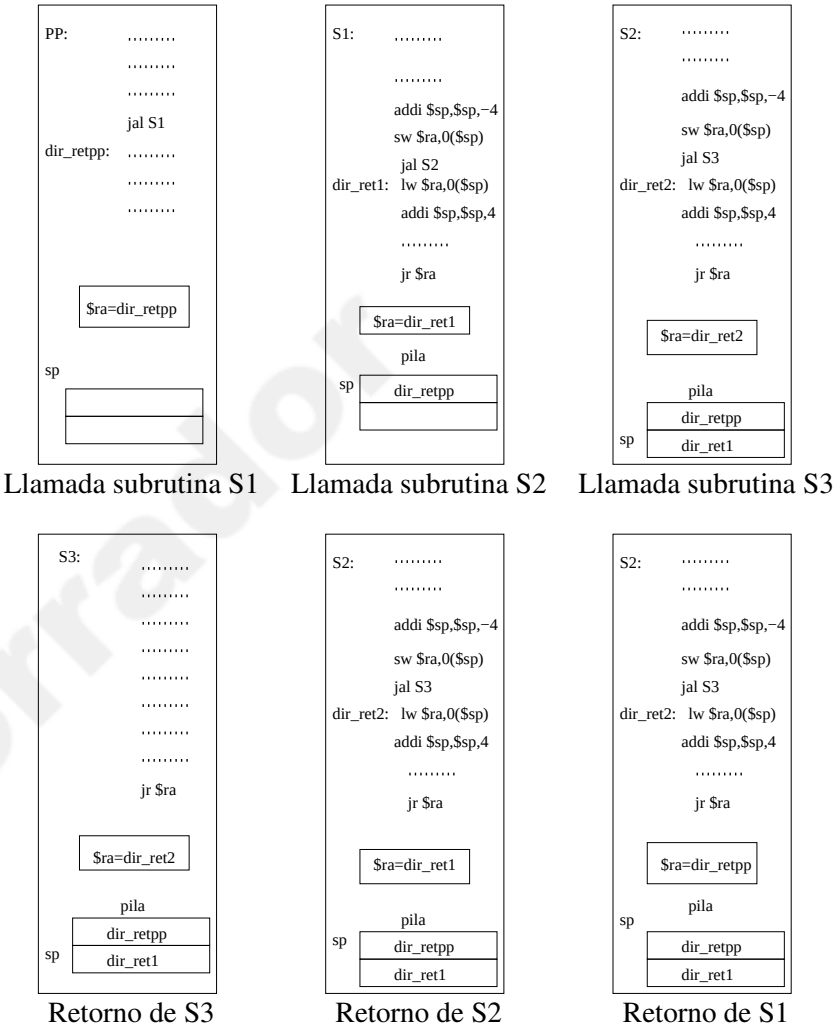


Figura 8.4: Llamadas anidadas a subrutinas (apilando las direcciones de retorno)

ejecución tras el retorno, será necesario almacenar temporalmente dicha información en algún lugar. Este lugar será el bloque de activación de la subrutina.

La forma en la que se almacena y restaura el contenido de aquellos registros que vaya a sobreescribir la subrutina en el bloque de activación es la siguiente. La subrutina, antes de modificar el contenido de los registros, los apila en el bloque de activación. Una vez finalizada la ejecución de la subrutina, y justo antes del retorno, los recupera.

Este planteamiento implica almacenar en primer lugar todos aquellos registros que vaya a modificar la subrutina, para posteriormente recuperar sus valores originales antes de retornar al programa principal. Para disminuir el número de registros que deben ser guardados y restaurados cada vez que se llame a una subrutina, MIPS32 establece el siguiente convenio:

- Los registros `$t0` a `$t9` (`$8` a `$15`, `$24` y `$25`) pueden ser utilizados por el programa invocado (subrutina) para almacenar datos temporales sin que sea necesario guardar sus valores originales. Se supone que la información que contienen no es relevante para el programa que realiza la llamada. En caso de que lo fuera, es el programa invocador quien deberá apilarlos antes de realizar la llamada y recuperarlos tras el retorno.

- Los registros `$s0` a `$s7` (`$16` a `$23`) también pueden ser utilizados por el programa invocado para almacenar datos temporales, sin embargo, estos sí que deben ser almacenados por el programa invocado para posteriormente recuperar su valor original.

En cuanto a qué registros puede modificar o no la subrutina, hay que tener en cuenta que los registros `$a0` a `$a3`, utilizados para el paso de parámetros, generalmente deberán ser preservados, por lo que en el caso de que vayan a ser modificados, se deberán almacenar en el bloque de activación para posteriormente ser restaurados.

Como ejemplo de cómo se utiliza el bloque de activación se propone el siguiente programa en C del que a continuación se mostrará una versión equivalente en ensamblador.

```
1  /* program.c */
2
3  #define N 6
4
5  void main ()
6  {
7    int A[N]={1,2,3,4,5,6};
8
9    if (N>0) suma (A,N);
10 }
11
```

```c
void suma(int *A_s, int dim_s)
{
  int B_s[N];
  int i;

  for (i=0; i<dim_s; i++)
     B_s[i] = suma_elem(&A_s[i], dim_s-i);

  for (i=0;i<dim_s;i++)
    A_s[i] = B_s[i];
}

int suma_elem(int *A_s, int dim_s)
{
  int i;
  int suma;

  suma= 0;
  for (i=0;i<dim_s;i++)
    suma = suma + A_s[i];

  return(suma);
}
```

A continuación se muestra el equivalente en ensamblador del MIPS32 del anterior programa en C.

✎ subr-varlocal.s

```
                .data
A:              .word 1, 2, 3, 4, 5, 6
dim:            .word 6

                .text
main:           lw   $a1, dim($0)
                beqz $a1, fin
                la   $a0, A
                jal  sum_total

fin:            li   $v0,10
                syscall

sum_total:      # ---- 1 ----
                addi $sp, $sp, -12
                # ...
                sw   $ra, 8($sp)
                sw   $a0, 4($sp)
                sw   $a1, 0($sp)

                addi $sp, $sp, -24

                # Parte principal de la subrutina

uno:            add  $t0, $sp, $0
                add  $t1, $0, $0
                add  $t2, $a1, $0

for1:           beqz $t2, finfor1

                # ---- 2 ----
                add  $a0, $t1, $a0
                add  $a1, $t2, $0
```

```
34
35                  # ── 3 ──
36                  addi $sp, $sp, -12
37                  sw   $t0, 8($sp)
38                  sw   $t1, 4($sp)
39                  sw   $t2, 0($sp)
40
41                  jal suma_elem
42
43                  # ── 4 ──
44                  lw   $t0, 8($sp)
45                  lw   $t1, 4($sp)
46                  lw   $t2, 0($sp)
47                  addi $sp, $sp, 12
48
49                  sw   $v0, 0($t0)
50
51                  addi $t0, $t0, 4
52                  addi $t2, $t2, -1
53                  add  $t1, $t1, 4
54                  lw   $a0, 44($sp)
55                  j    for1
56
57  finfor1:        add  $t0, $sp, $0
58
59                  # ── 5 ──
60                  lw   $a0, 44($sp)
61                  lw   $a1, 40($sp)
62
63  for2:           beqz $a1, finfor2
64                  lw   $t1, 0($t0)
65                  sw   $t1, 0($a0)
66                  addi $t0, $t0, 4
67                  addi $a0, $a0, 4
68                  addi $a1, $a1, -1
69                  j for2
70
71                  # ── 6 ──
72  finfor2:        lw   $ra, 48($sp)
73                  addi $sp, $sp, 52
74                  # ...
75                  jr   $ra
76
77
78
79  sum_elem:       add  $v0, $0, $0
80  for3:           beqz $a1, finfor3
81                  lw   $t0, 0($a0)
82                  addi $a0, $a0, 4
83                  add  $v0, $v0, $t0
84                  addi $a1, $a1, -1
85                  j    for3
86  finfor3:        jr   $ra
```

EJERCICIOS

▶ **184** Dibuja el bloque de activación de la subrutina «sum_total». Justifica cada una de las informaciones que forman parte del bloque de activación.

▶ **185** ¿Por qué se apilan el contenido de los registros `$t0`, `$t1` y `$t2`?

▶ **186** De momento no se ha creado el bloque de activación de la subrutina «sum_elem», ¿a qué crees que es debido?

8.2.4. Estructura y gestión del bloque de activación

Como se ha visto, el bloque de activación de una subrutina está localizado en memoria y se implementa por medio de una estructura de tipo pila. El bloque de activación visto hasta este momento se muestra en la Figura 8.5.

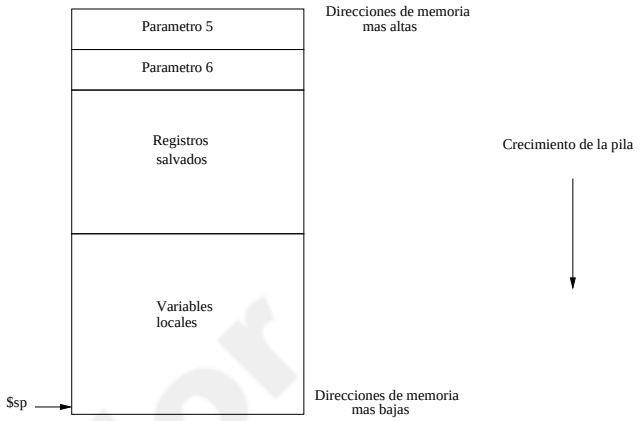


Figura 8.5: Esquema del bloque de activación (sin utilizar el registro `$fp`)

Un aspecto que influye en la eficiencia de los programas que manejan subrutinas y sus respectivos bloques de activación es el modo en el que se accede a la información contenida en los respectivos bloques de activación.

El modo más sencillo para acceder a un dato en el bloque de activación es utilizando el modo indexado. En el modo indexado la dirección del dato se obtiene sumando una dirección base y un desplazamiento. Como dirección base se puede utilizar el contenido del puntero de pila, que apunta a la posición de memoria más baja del bloque de activación (véase la Figura 8.5). El desplazamiento sería entonces la posición relativa del dato con respecto al puntero de pila. De esta forma, sumando el contenido del registro `$sp` y un determinado desplazamiento se obtendría la dirección de memoria de cualquier dato que se encontrara en el bloque de activación. Por ejemplo, si se ha apilado una palabra en la posición 8 por encima del `$sp`, se podría leer su valor utilizando la instrucción «**lw** $t0, 8($sp)».

Sin embargo, utilizar el registro `$sp` como dirección base para calcular la dirección de un dato presenta un inconveniente: el contenido del registro `$sp` puede variar durante la ejecución de la subrutina y, por tanto, habría que variar de manera acorde el desplazamiento necesario para acceder a cada uno de los datos.

Para conseguir que el desplazamiento de cada uno de los datos sea constante, basta con disponer de una referencia fija al bloque de activación. Dicha

referencia se puede almacenar en un registro determinado y dicho registro se puede utilizar entonces como dirección base (en lugar del `$sp`).

El ensamblador del MIPS32 reserva un registro para dicho fin, el registro `$fp` (*frame pointer*, en inglés), que corresponde al registro `$30` del banco de registros.

Por convenio se utiliza como referencia fija al bloque de activación la dirección de memoria que indica el registro `$sp` cuando se entra en la subrutina menos 4. De esta forma, cuando se requiera acceder a algún dato apilado por la subrutina, se utilizará un desplazamiento negativo respecto a dicha referencia. Por otro lado, cuando se quiera acceder a algún dato apilado por el programa invocador justo antes de llamar a la subrutina se utilizará un desplazamiento positivo respecto a dicha referencia.

Hay que tener en cuenta que, puesto que el registro `$fp` mantiene una referencia fija al bloque de activación de la subrutina en ejecución, el valor de dicho registro debe ser preservado entre llamadas a subrutinas. Así que el valor previo del registro `$fp` deberá ser apilado por la subrutina para poder restaurarlo antes de devolver el control al programa invocador.

La Figura 8.6 muestra el bloque de activación de una subrutina tal y como se ha descrito.

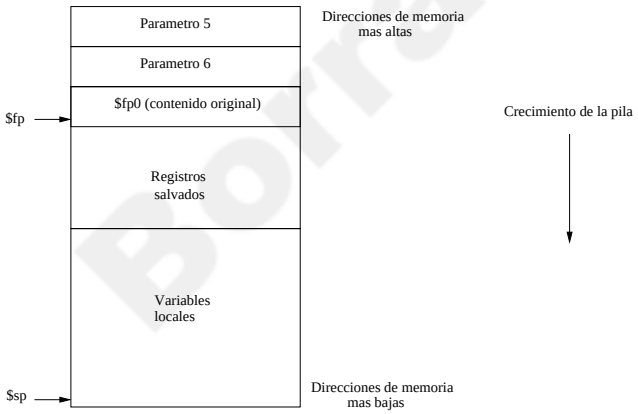


Figura 8.6: Esquema del bloque de activación de una subrutina

8.2.5. Convenio para la llamada a subrutinas

Tanto el programa invocador como el invocado intervienen en la creación y gestión del bloque de activación de una subrutina. La gestión del bloque de activación se produce principalmente en los siguientes momentos:

- Justo antes de que el programa invocador pase el control a la subrutina.

- En el momento en que la subrutina toma el control.
- Justo antes de que la subrutina devuelva el control al programa invocador.
- En el momento en el que el programa invocador recupera el control.

A continuación se describe con más detalle qué es lo que debe realizarse en cada uno de dichos momentos.

Justo antes de que el programa invocador pase el control a la subrutina:

1. Almacenar en la pila los registros que deben ser salvados por el invocador. Según el convenio, si al invocador le interesa preservar el contenido de algunos de los registros `$t0` a `$t9`, éstos deben ser apilados antes de hacer la llamada.
2. Paso de parámetros. Cargar los parámetros en los lugares establecidos. Los cuatro primeros se cargan en registros, `$a0` a `$a3`, y los restantes se apilan en el bloque de activación.

En el momento en que la subrutina toma el control:

1. Reservar memoria en la pila para el resto del bloque de activación.
   El tamaño se calcula sumando el espacio en bytes que ocupa el contenido de los registros que vaya a apilar la subrutina (`$fp`, `$s0` a `$s7`, `$ra`, `$a0` a `$a3`) más el espacio que ocupan las variables locales que se vayan a almacenar en el bloque de activación.
2. Almacenar el contenido del registro `$fp` en la dirección más alta del espacio reservado en el bloque de activación.
3. Actualizar el puntero de bloque de activación (registro `$fp`) para que apunte a la dirección más alta del espacio reservado (la misma en la que se ha guardado su valor previo).
4. Almacenar en el bloque de activación aquellos registros que vaya a modificar la subrutina.

Justo antes de que la subrutina devuelva el control al programa invocador:

1. Cargar el valor (o valores) que deba devolver la subrutina en los registros `$v0` y `$v1`.
2. Desapilar los registros apilados por la subrutina, excepto el registro `$fp`.

3. Copiar el contenido del registro `$fp` en el registro `$sp` (con lo que se libera el espacio en pila ocupado por la subrutina para las variables locales).

4. Desapilar el registro `$fp`.

En el momento en el que el programa invocador recupera el control:

1. Eliminar del bloque de activación los parámetros que hubiera apilado.

2. Desapilar los registros que había apilado.

3. Recoger los parámetros devueltos.

En la Figura 8.7 se muestra el estado de la pila después de que un programa haya invocado a otro, siguiendo los pasos que se han descrito. En dicha figura aparece enmarcado el bloque de activación creado por el programa invocador y el invocado.

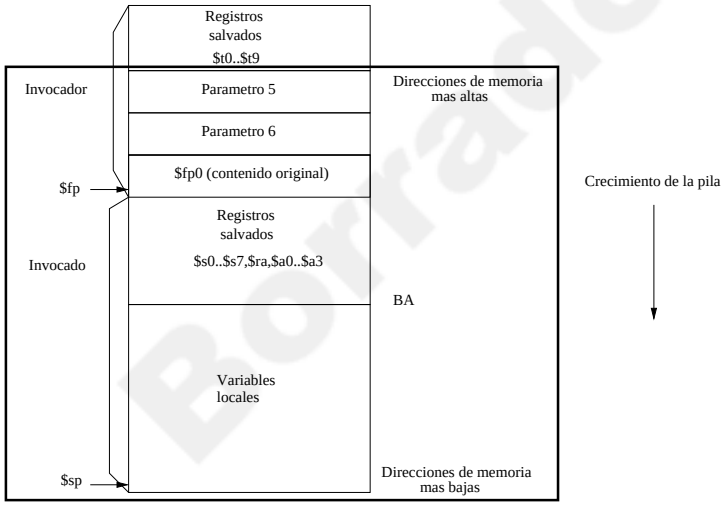


Figura 8.7: Estado de la pila después de una llamada a subrutina

Analiza el código del programa `subr-sc.s` y contesta a las siguientes cuestiones.

✎ subr-sc.s

```
1               .data
2 A:            .word 1, 2, 3, 4, 5, 6
3 dim:          .word 6
4
5               .text
6 main:         lw    $a1, dim($0)
7               beqz  $a1, fin
8               la    $a0, A
```

```
9             jal   subr
10
11  fin:      li    $v0, 10
12            syscall
13
14  sum_total:
15            # ---- 1 ----
16            addi  $sp, $sp, -16
17            sw    $fp, 12($sp)
18            addi  $fp, $sp, 12
19            sw    $ra, -4($fp)
20            sw    $a0, -8($fp)
21            sw    $a1, -12($fp)
22            addi  $sp, $sp, -24
23
24            add   $t0, $sp, $0
25            add   $t1, $0, $0
26            add   $t2, $a1, $0
27
28  for1:     beqz  $t2, finfor1
29
30            # ---- 2 ----
31            add   $a0, $t1, $a0
32            add   $a1, $t2, $0
33
34            # ---- 2 ----
35            addi  $sp, $sp, -12
36            sw    $t0, 8($sp)
37            sw    $t1, 4($sp)
38            sw    $t2, 0($sp)
39
40            jal   suma_elem
41
42            # ---- 3 ----
43            lw    $t0, 8($sp)
44            lw    $t1, 4($sp)
45            lw    $t2, 0($sp)
46            addi  $sp, $sp, 12
47
48            # ---- 4 ----
49            sw    $v0, 0($t0)
50
51            addi  $t0, $t0, 4
52            addi  $t2, $t2, -1
53            add   $t1, $t1, 4
54            lw    $a0, -8($fp)
55            j     for1
56
57  finfor1:  add   $t0, $sp, $0
58
59            # ---- 5 ----
60            lw    $a0, -8($fp)
61            lw    $a1, -12($fp)
62
63  for2:     beqz  $a1, finfor2
64            lw    $t1, 0($t0)
65            sw    $t1, 0($a0)
66            addi  $t0, $t0, 4
67            addi  $a0, $a0, 4
68            addi  $a1, $a1, -1
69            j     for2
70
71            # ---- 6 ----
72  finfor2:  lw    $ra, -4($fp)
73            add   $sp, $fp, $0
74            lw    $fp, 0($sp)
75            addi  $sp, $sp, 4
```

```
76              jr    $ra
77
78
79  suma_elem:
80              # ---- 7 ----
81              addi  $sp, $sp ,-4
82              sw    $fp, 0(fp)
83              addi  $fp, $sp, 0
84
85              add $v0, $0, $0
86  for3:       beqz  $a1, finfor3
87              lw    $t0, 0($a0)
88              addi  $a0, $a0, 4
89              add   $v0, $v0, $t0
90              addi  $a1, $a1, -1
91              j     for3
92
93  finfor3:
94              # ---- 8 ----
95              addi  $sp, $fp, 0
96              lw    $fp, 0($sp)
97              addi  $sp, $sp, 4
98
99              # ---- 9 ----
100             jr  $ra
```


EJERCICIOS

▶ **187** Localiza el fragmento de código del programa ensamblador donde se pasan los parámetros a la subrutina « sum_total ». Indica cuántos parámetros se pasan, el lugar por donde se pasan y el tipo de parámetros.

▶ **188** Indica el contenido del registro `$ra` una vez ejecutada la instrucción « **jal** sum_total ».

▶ **189** Dibuja y detalla (con los desplazamientos correspondientes) el bloque de activación creado por la subrutina « sum_total ». Justifica el almacenamiento de cada uno de los datos que contiene el bloque de activación.

▶ **190** ¿Por qué la subrutina « suma_total » apila en el bloque de activación el contenido de los registros `$t0`, `$t1` y `$t2`? Dibuja el estado de la pila después de este apilamiento.

▶ **191** Indica el fragmento de código del programa ensamblador donde se pasan los parámetros a la subrutina «suma_elem». Indica cuántos parámetros se pasan, el lugar por donde se pasan y el tipo de parámetros.

▶ **192** Indica el contenido del registro `$ra` una vez ejecutada la instrucción « **jal** suma_elem».

▶ **193** Dibuja el bloque de activación de la subrutina «suma_elem».

▶ **194** Indica el contenido del registro `$ra` una vez ejecutada la instrucción «**jr** $ra» de la subrutina «suma_elem» ¿Dónde se recupera el valor que permite retornar al programa principal?

▶ 195 Localiza el fragmento de código donde se desapila el bloque de activación de la subrutina «sum_total».

▶ 196 Indica el contenido del registro `$ra` antes de ejecutar la instrucción «**jr** $ra» en la subrutina «sum_total».

8.3. Problemas del capítulo

EJERCICIOS

▶ 197 Desarrolla dos subrutinas en ensamblador: «subr1» y «subr2». La subrutina «subr1» tomará como entrada una matriz de enteros de dimensión $n \times m$ y devolverá dicha matriz pero con los elementos de cada una de sus filas invertidos. Para realizar la inversión de cada una de las filas se deberá utilizar la subrutina «subr2». Es decir, la subrutina «subr2» deberá tomar como entrada un vector de enteros y devolver dicho vector con sus elementos invertidos.

▶ 198 Desarrolla tres subrutinas en ensamblador, «subr1», «subr2» y «subr3». La subrutina «subr1» devolverá un 1 si las dos cadenas de caracteres que se le pasan como parámetro contienen el mismo número de los distintos caracteres que las componen. Es decir, devolverá un 1 si una cadena es un anagrama de la otra. Por ejemplo, la cadena «ramo» es un anagrama de «mora».

La subrutina «subr1» se apoyará en las subrutinas «subr2» y «subr3». La subrutina «subr2» tiene como finalidad calcular cuántos caracteres de cada tipo tiene la cadena que se le pasa como parámetro. La subrutina «subr3» devuelve un 1 si el contenido de los dos vectores que se le pasa como parámetros son iguales.

Suponer que las cadenas están compuestas por el conjunto de letras que componen el abecedario en minúsculas.

▶ 199 Desarrolla en ensamblador la siguiente subrutina recursiva descrita en lenguaje C:

```
int ncsr(int n, int k)
{
 if (k>n) return 0;
 else if (n == k || k == 0) return 1;
      else return ncsr(n-1,k)+ncsr(n-1,k-1);
}
```

▶ 200 Desarrolla un programa en ensamblador que calcule el máximo de un vector cuyos elementos se obtienen como la suma de los elementos fila de una matriz de dimensión $n \times n$. El programa debe tener la siguiente estructura:

- Deberá estar compuesto por 3 subrutinas: «subr1», «subr2» y «subr3».
- «subr1» calculará el máximo buscado. Se le pasará como parámetros la matriz, su dimensión y devolverá el máximo buscado.
- «subr2» calculará la suma de los elementos de un vector. Se le pasará como parámetros un vector y su dimensión y la subrutina devolverá la suma de sus elementos.
- «subr3» calculará el máximo de los elementos de un vector. Se le pasará como parámetros un vector y su dimensión y devolverá el máximo de dicho vector.
- El programa principal se encargará de realizar la entrada de la dimensión de la matriz y de sus elementos y llamará a la subrutina «subr1», quién devolverá el máximo buscado. El programa principal deberá almacenar este dato en la posición etiquetada con «max».

CAPÍTULO 9

GESTIÓN DE LA ENTRADA/SALIDA MEDIANTE CONSULTA DE ESTADO

Índice

9.1. Dispositivos periféricos en el simulador SPIM **104**
9.2. Entrada de datos desde el teclado **106**
9.3. Salida de datos por pantalla **107**
9.4. Entrada/Salida de datos **109**
9.5. Problemas del capítulo **110**

Los computadores no funcionan de forma aislada. Deben relacionarse e interactuar con el entorno que les rodea. Ya sea con sus usuarios, con otros computadores o con otros aparatos electrónicos. Los dispositivos que permiten la comunicación de un computador con el exterior, reciben el nombre de periféricos.

Algunos periféricos sirven para que el computador presente información o lleve a cabo determinadas acciones. Estos periféricos reciben el nombre de periféricos de salida. Periféricos de salida son, por ejemplo, los monitores y las impresoras.

También existen periféricos que sirven para introducir información en el computador. Éstos reciben el nombre de periféricos de entrada. Periféricos de entrada son, por ejemplo, los teclados y los escáneres.

Por último, también hay periféricos que permiten que la información vaya en los dos sentidos. Son los llamados periféricos de entrada/salida. Periféricos

de entrada/salida son, por ejemplo, los discos duros y las pantallas táctiles.

De hecho, existe una gran variedad de periféricos. Cada uno de ellos realiza funciones muy específicas y presenta determinados requerimientos de comunicación. Por ejemplo, un teclado permite introducir caracteres en un computador y, por tanto, debe enviar la información de qué teclas se están pulsando al computador. Sin embargo, la velocidad con la que debe transmitir dicha información no tiene que ser demasiado alta. Al fin y al cabo, el teclado estará siendo utilizado por una persona, que en el peor de los casos utilizará más de dos dedos y tecleará sin mirar el teclado. Por otro lado, un disco duro permite almacenar y recuperar información digital: documentos, programas, etc. Pero en el caso de un disco duro, la velocidad con la que se transmite la información es crucial. De hecho, será tanto mejor cuanto más rápido permita la lectura o escritura de la información.

Además de por las funciones que realizan y de los requerimientos de comunicación, los periféricos también se diferencian por la tecnología que emplean, es decir, por la forma en la que llevan a cabo su cometido. Por ejemplo, en el caso de las impresoras, éstas se pueden clasificar en: térmicas, de impacto, de inyección de tinta, láseres, por sublimación, etc. Cada una de ellas utiliza una tecnología distinta para llevar a cabo el mismo fin: presentar sobre papel, o otro tipo de material, una determinada información.

Toda esta variedad de funciones, necesidades de comunicación y tecnologías hace necesaria la existencia de hardware encargado de facilitar la comunicación del procesador con los distintos periféricos. El hardware encargado de facilitar la comunicación del procesador con uno o varios periféricos recibe el nombre de *controlador de E/S*.

Los controladores de E/S ocultan al procesador las dificultades propias de la gestión de los diferentes dispositivos y proporcionan una interfaz común al procesador. Por ejemplo, en el caso de los discos duros, cada fabricante utiliza una determinada tecnología para la fabricación de sus productos. Cada disco duro incorpora un controlador de E/S que conoce las peculiaridades de funcionamiento dicho disco duro en concreto. Pero además, dicho controlador de E/S cumple con una serie de normas que son comunes a todos los controladores de E/S para discos duros y que permiten la comunicación del disco duro con el procesador.

Los controladores proporcionan una serie de registros que son direccionables por el procesador y que son los utilizados para realizar las operaciones de E/S.

Los registros que habitualmente se encuentran en los controladores de E/S son:

- Un *registro de control*. Lo utiliza el procesador para indicar al controlador qué operación de E/S se debe realizar.
- Un *registro de estado*. Almacena el estado del periférico y de la operación de E/S que se acaba de realizar.
- Un *registro de datos*. Almacena el dato que se va a intercambiar con el periférico durante una operación de entrada o salida. En una operación de entrada, el controlador cargará en este registro el dato obtenido por el periférico. De esta forma, el procesador podrá leer este registro y obtener el dato proporcionado por el periférico. En una operación de salida, por contra, será el procesador quien cargue en este registro el dato que se debe enviar al periférico. De esta forma, el controlador enviará al periférico el dato indicado por el procesador.

  Este registro también recibe el nombre de *puerto de entrada o salida*, según se utilice para realizar operaciones de entrada o salida.

Para realizar una operación de E/S con un determinado periférico, el procesador debe leer o escribir en los registros de su controlador de E/S. Por tanto, dichos registros deben ser accesibles por el procesador. El acceso a dichos registros puede conseguirse de una de las siguientes formas:

- Haciendo que los registros formen parte del espacio de direcciones de memoria del computador. En este caso se dice del sistema de E/S que está *mapeado en memoria*. El procesador lee o escribe en los registros de los controladores de E/S de la misma forma que lo haría en cualquier otra parte de la memoria. Ésta es la modalidad utilizada por MIPS.
- Haciendo que el computador disponga de un mapa de direcciones para la E/S independiente del mapa de direcciones de memoria. En este caso se dice del sistema de E/S que está *aislado*. El procesador lee o escribe en los registros de los controladores de E/S utilizando instrucciones especiales. Ésta es la modalidad utilizada por los procesadores basados en la arquitectura x86.

Uno de los aspectos más importantes en la comunicación entre el procesador y un periférico es la correcta sincronización entre ambos. Cuando se pide un dato a un periférico, es necesario esperar a que dicho dato esté disponible. De igual forma, cuando se envía un dato a un periférico es necesario esperar a que el envío se haya concluido antes de enviar un nuevo dato. Dicha sincronización se puede llevar a cabo de dos formas distintas. En este capítulo se explorará la forma más sencilla de sincronización, que recibe el nombre de *E/S por consulta de estado*.

Cuando la gestión de la E/S se realiza por medio de la consulta de estado, el programa que se comunica con un periférico es quien se encarga de la sincronización entre el procesador y dicho periférico. En esencia, la sincronización se realiza por medio de un bucle de espera (*espera activa*) en el que se comprueba el estado del periférico. Dicho bucle termina cuando al comprobar el estado del periférico, éste indica que está listo para intercambiar la información requerida.

9.1. Dispositivos periféricos en el simulador SPIM

SPIM simula el funcionamiento de dos dispositivos de E/S. Un periférico de entrada, un teclado, y un periférico de salida, una pantalla (véase la Figura 9.1). El teclado simulado permite la introducción de caracteres desde el teclado del computador. La pantalla simulada permite escribir caracteres en una de las ventanas del simulador. Ambas unidades son independientes. Es decir, si se pulsa una tecla, el carácter introducido no se imprime automáticamente en la pantalla. Para que esto sea así, además de realizar una operación de entrada sobre el teclado, una vez obtenido el carácter, hay que realizar una operación de salida sobre la pantalla.

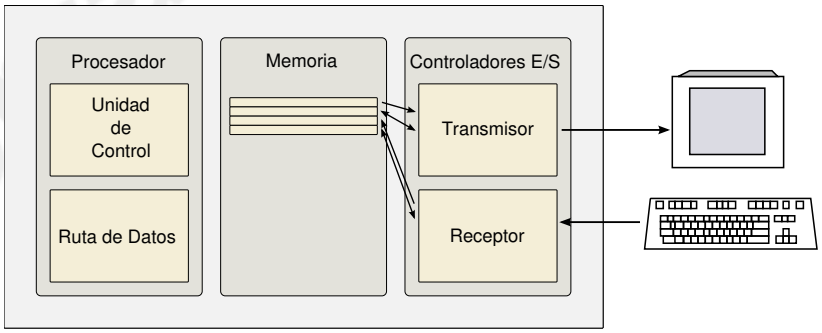


Figura 9.1: Dispositivos periféricos en el simulador SPIM

Cada controlador de E/S posee dos registros: uno de `control/estado` y otro de `datos` (véase la Figura 9.2). Cada uno de dichos registros tiene asignada una dirección dentro del espacio de direcciones de memoria.

A continuación se describen los registros de `control/estado` y de `datos` de los controladores del teclado y la pantalla. Los registros del controlador del teclado son:

Registro de `control/estado` Este registro tiene asignada la dirección de memoria `0xFFFF 0000`. Cuando se escribe en dicha posición de me-

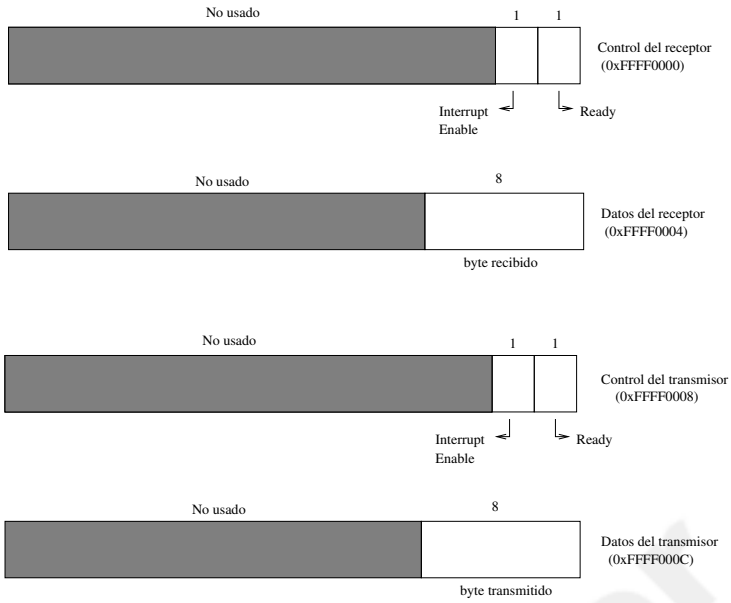


Figura 9.2: Registros de los dispositivos de E/S de SPIM

moria, se accede al registro de `control`. Cuando se realiza una operación de lectura sobre dicha posición de memoria, se accede al registro de `estado`. En realidad, tan sólo se usan dos de sus bits:

- El bit 0 de este registro, llamado *ready*, es un bit de sólo lectura. Es decir, se ignoran las escrituras que se hagan sobre él. Este bit cambia automáticamente de 0 a 1 cuando se teclea un carácter en el teclado. Se pone automáticamente a 0 en el momento que el procesador recoge el carácter introducido.
- El bit 1 de este registro es un bit de escritura que se deberá fijar a 0. En el próximo capítulo se explicará su funcionalidad.

Registro de `datos` Este registro tiene asignada la dirección `0xFFFF0004`. Es de sólo lectura y su contenido cambia cuando se pulsa un nuevo carácter en el teclado. Los 8 bits de menor peso de este registro almacenan el código ASCII del último carácter pulsado en el teclado. Los demás bits contienen ceros.

En cuanto a los registros del controlador de la pantalla, éstos son:

Registro de `control/estado` Está localizado en la dirección de memoria `0xFFFF0008`. Su funcionalidad es similar a la del mismo registro en el controlador del teclado. Sólo se usan dos de sus bits:

- El bit 0, llamado *ready*, es de sólo lectura. Indica si se ha terminado de escribir el carácter enviado en pantalla. Si ésta todavía está ocupada escribiendo en la pantalla el último carácter enviado, el bit *ready* estará a 0. Si ya se ha escrito el último carácter en la pantalla, el bit *ready* pasará a 1.
- El bit 1 de este registro es un bit de escritura que se deberá fijar a 0. En el próximo capítulo se explicará su funcionalidad.

Registro de `datos` Este registro tiene asignado la dirección `0xFFFF000C`. Es un registro de sólo escritura. Cuando se escribe un valor en dicha posición de memoria, los 8 bits de menor peso de dicho valor se envían a la pantalla. Sólo se debe escribir en el registro de `datos` del controlador cuando el bit *ready* valga 1. Si la pantalla no está preparada, las escrituras en el registro de datos del controlador se ignorarán. Es decir, los caracteres enviados no se imprimirán en pantalla.

9.2. Entrada de datos desde el teclado

En este apartado se muestra un ejemplo de cómo se lleva a cabo una operación de entrada mediante la técnica de E/S programada por consulta de estado. La operación se va a realizar sobre el teclado del simulador SPIM.

El siguiente programa lee los caracteres introducidos desde el teclado sin utilizar ninguna técnica que sincronice las operaciones de entrada.

esc-lee.s

```
1           # Zona de datos
2           .data 0x10010000
3   long:   .word   7           # Tamaño del buffer
4   buffer: .space  7           # Buffer donde se almacenan los
5                               #  carácteres que se pulsan
6
7           # Zona de instrucciones
8           .text
9   main:   lui   $a1,0xffff   # $a1 <- 0xffff0000
10                              # (dir base de los registros de E/S)
11          la    $a0, buffer   # Carga la dirección del buffer
12          lw    $v0, long($0) # Carga la longitud del buffer
13          addi  $v0, $v0, -1
14  ctr:    # [En blanco, por ahora]
15          # [En blanco, por ahora]
16          # [En blanco, por ahora]
17          lb    $s0, 4($a1)   # Lee del puerto del teclado
18          sb    $s0, 0($a0)   # Almacena el dato en el buffer
19          addi  $a0, $a0, 1   # Incrementa el puntero del buffer
20          addi  $v0, $v0, -1  # Decr. el tamaño del buffer
21          bne   $v0, $0, ctr  # ¿Se ha llenado el buffer?
22  fin:    li    $s0, 0x00
23          sb    $s0, 0($a0)   # Almacena fin de cadena en buffer
24
25          li    $v0,10        # Finaliza la ejecución
26          syscall
```

EJERCICIOS

▶ **201** Ejecuta el simulador XSPIM con la opción «`-mapped_io`», carga el programa anterior y ejecútalo. Analiza qué ocurre.

Como habrás podido comprobar, el programa ha finalizado sin que se haya podido introducir nada por el teclado. Para que el programa funcione correctamente se deberá leer el puerto de entrada únicamente cuando se haya pulsado una tecla. Para ello, se debe modificar el programa anterior incluyendo el siguiente fragmento de código a partir de la etiqueta « ctr »:

```
14 ctr:     lb   $t0,0($a1)
15          andi $t0,$t0,1
16          beqz $t0,ctr
```

EJERCICIOS

▶ **202** Describe con detalle qué hace el fragmento de código anterior.

▶ **203** Comprueba que el nuevo programa funciona de la siguiente forma. Si sitúas el cursor del ratón sobre la ventana de la pantalla y tecleas 6 letras, el programa acaba. Al finalizar su ejecución, XSPIM refresca el contenido de los paneles y entonces se puede observar que a partir de la dirección de memoria « buffer », aparecen los códigos ASCII de las letras tecleadas. (NOTA: *el programa sólo lee del teclado, no muestra por pantalla lo que se escribe, por lo tanto, mientras dure la ejecución del programa no se verá nada de lo que se esté tecleando.*)

▶ **204** Indica qué instrucción ejecuta el procesador para recoger el dato almacenado en el registro de `datos` (introducido a través del teclado). ¿Qué tipo de instrucción es? ¿Por qué se utiliza ese tipo de instrucción?

▶ **205** Modifica de nuevo el programa para añadir la siguiente funcionalidad: cuando el usuario pulse la tecla de retorno de carro, «`\n`» (`0x0d`, en hexadecimal), se tiene que almacenar dicho carácter seguido del carácter de fin de cadena, `0x00`. Esto servirá para indicar que la entrada de datos ha terminado antes de que se haya llenado el « buffer ».

9.3. Salida de datos por pantalla

En este apartado se describe cómo se realiza una operación de salida utilizando la técnica de E/S programada por consulta de estado. La operación se va a realizar sobre la pantalla del simulador SPIM.

El siguiente programa lee los caracteres almacenados en memoria a partir de la dirección «cadena» y los escribe en el puerto de salida, para mostrarlos en la pantalla del simulador. Sin embargo, al igual que ocurría en el apartado anterior, no se ha implementado, por el momento, la sincronización necesaria para que la operación de salida tenga lugar de una forma ordenada.

✎ esc-escribe.s

```
1          # Zona de datos
2          .data 0x10010000
3  cadena: .asciiz "Esto es un texto de prueba."
4
5          # Zona de instrucciones
6          .text
7  main:   lui  $a1,0xffff      # $a0 <- 0xFFFF0000
8                               # (dir base de los registros de E/S)
9          la   $a0, cadena     # Carga la dirección de la cadena
10         lb   $s0, 0($a0)     # Lee un byte de cadena
11
12 ctr1:                        # [Bucle de retardo]
13         sb   $s0, 0xC($a1)   # Escribe en la pantalla
14         addi $a0, $a0, 1     # Incrementa el puntero de cadena
15         lb   $s0, 0($a0)     # Lee el siguiente byte de cadena
16         bne  $s0, $0, ctr1   # ¿Se ha llegado al final de cadena?
17 ctr2:                        # [Bucle de retardo]
18         sb   $s0, 0xC($a1)   # Escribe en la pantalla el caracter \0
19
20         li   $v0,10          # Finaliza la ejecución
21         syscall
```

Ejecuta el programa en el simulador (recuerda ejecutar XSPIM con la opción «`-mapped_io`»). Como puedes comprobar, no se muestra en la pantalla toda la frase almacenada en memoria.

Inserta ahora los siguientes fragmentos de código dónde antes sólo estaban las líneas etiquetadas con «ctr1» y «ctr2»:

```
12 ctr1:                        # [Bucle de retardo]
13         li   $t0, 10         # Retardo en función del valor de $t0
14 cont1:  addi $t0, $t0, -1
15         bnez $t0, cont1      # [Fin bucle de retardo]
```

```
21 ctr2:                        # [Bucle de retardo]
22         li   $t0, 10         # Retardo en función del valor de $t0
23 cont2:  addi $t0, $t0, -1
24         bnez $t0, cont2      # [Fin bucle de retardo]
```

EJERCICIOS

Fíjate en que se ha puesto `10` como valor inicial del registro `$t0` en los dos bucles de retardo, carga el nuevo programa en el simulador y ejecútalo.

▶ 206 Es posible que ahora aparezcan más caracteres de la cadena en la pantalla ¿Los caracteres que aparecen tienen algún sentido? ¿Forman parte de la cadena? ¿Siguen algún patrón?

▶ **207** Sustituye el número `10` que aparece en las líneas 13 y 22 por números cada vez mayores y comprueba si con cada modificación del programa aparecen más caracteres de la cadena en la pantalla. ¿A partir de qué valor aparece la cadena completa?

▶ **208** ¿Qué se está haciendo realmente al incrementar el valor asignado a `$t0`?

▶ **209** ¿Crees que el número que permite ver toda la cadena sería válido si se ejecutara el programa en un computador más o menos rápido que el que has utilizado? ¿Por qué motivo?

▶ **210** Tomando como ejemplo la sección anterior, modifica el código para que realice el control de la salida de datos por la pantalla mediante consultas de estado (en lugar de utilizar los bucles de retardo).

▶ **211** ¿Serviría este programa para ser ejecutado en un computador distinto al que has utilizado? ¿Por qué motivo?

▶ **212** Indica qué instrucción ejecuta el procesador para que el dato que se va a imprimir en la pantalla se almacene en el registro de `datos` del transmisor. ¿Qué tipo de instrucción es? ¿Porqué se utiliza ese tipo de instrucción?

9.4. Entrada/Salida de datos

El siguiente código lee los caracteres pulsados desde el teclado (hasta un total de 10) y los muestra por la pantalla. Sin embargo, se ha omitido a propósito el código de las rutinas «wi» y «wo» que deben realizar el control de la consulta de estado de los periféricos de entrada y salida, respectivamente.

✎ esc-lee-escribe.s

```
1          # Zona de datos
2          .data 0x10010000
3  long:   .word 10              # Tamaño del buffer
4
5
6          # Zona de instrucciones
7          .text
8  main:   lui  $t1, 0xFFFF      # Dir base de los registros de E/S
9          lw   $s0, long($0)
10         addi $s0, $s0, -1
11         li   $t6, 0x0D        # Caracter de retorno de carro '\n'
12
13 ctri:   jal  wi
14         lb   $t7, 4($t1)      # Lee del teclado
15         jal  wo
16         sb   $t7, 0xC($t1)    # Escribe en la pantalla
17         addi $s0, $s0, -1
18         beq  $t7, $t6, fin    # ¿Se ha pulsado retorno de carro?
```

```
19              bne   $s0, $0, ctri     # ¿Se ha llenado el buffer?
20              j     cero
21  fin:        li    $t7, 0x0A
22              jal   wo
23              sb    $t7, 0xC($t1)     # Escribe en la pantalla
24  cero:       andi  $t7, $t7, 0
25              jal   wo
26              sb    $t7, 0xC($t1)     # Escribe en la pantalla
27
28              li $v0,10               # Finaliza la ejecución
29              syscall
30
31  # -<[ Consulta del estado de la entrada  ]>-
32  wi:
33
34  # -<[ Consulta del estado de la salida   ]>-
35  wo:
```

EJERCICIOS

▶ **213** Basándote en los ejemplos anteriores, completa las rutinas de control de los periféricos, «wi» y «wo». Comprueba el funcionamiento del programa en el simulador.

▶ **214** Modifica el código del programa anterior para utilizar una única rutina de consulta de estado que tome como parámetro el registro de control del periférico y sirva para realizar tanto la sincronización de la entrada como la de la salida. Comprueba su funcionamiento en el simulador.

9.5. Problemas del capítulo

Los ejercicios propuestos a continuación deben realizarse sin utilizar la instrucción « **syscall** » de llamada al sistema —salvo para la finalización del programa.

EJERCICIOS

▶ **215** Desarrolla un programa en ensamblador que, dado un número entero almacenado en memoria, muestre su valor por la pantalla:

a) En decimal.

b) En hexadecimal.

▶ **216** Desarrolla un programa que pida un entero por el teclado y lo guarde en memoria.

▶ **217** Desarrolla un programa que, dada una dirección de memoria, muestre por la pantalla el contenido de los diez bytes de memoria almacenados a partir de dicha dirección. Expresa el valor de cada byte en hexadecimal y separa la representación de cada uno de los bytes con guiones («-»).

CAPÍTULO 10

GESTIÓN DE LA ENTRADA/SALIDA MEDIANTE INTERRUPCIONES

Índice

10.1. Registros para el tratamiento de las excepciones **113**
10.2. Tratamiento de las excepciones **117**
10.3. Opciones del programa SPIM para la simulación de interrupciones **124**
10.4. Excepciones software **124**
10.5. Interrupciones **127**
10.6. Problemas del capítulo **141**

En el capítulo anterior se ha visto cómo gestionar la entrada y salida de un computador utilizando la técnica de consulta de estado.

La técnica de consulta de estado obliga al procesador a encargarse de revisar periódicamente el estado de los distintos dispositivos de entrada y salida. Cuando podría estar realizando en su lugar otro tipo de tareas. Por tanto, al tener que consultar el estado de los dispositivos, parte de la potencia de cálculo de un computador se malgasta en una tarea que la mayor parte de las veces será improductiva.

Una gestión más eficiente de la entrada y salida conlleva que sean los propios dispositivos los que avisen al procesador en el momento en el que se haya completado una operación de entrada o salida. Cuando se completa una operación de entrada o salida, el dispositivo genera lo que se conoce como una

interrupción. De esta forma, el procesador no pierde el tiempo interrogando periódicamente a los dispositivos para saber si se ha completado o no una operación de entrada o salida. Por el contrario, son los dispositivos los que avisan al procesador cuando dicha operación de entrada o salida ha terminado.

El uso de interrupciones permite compaginar eficientemente las diferentes velocidades de funcionamiento de los distintos periféricos y del procesador. Permite que el procesador y los periféricos trabajen de forma paralela. Mientras un determinado periférico está realizando una operación de entrada y salida, el procesador puede estar ejecutando instrucciones que resuelven otra tarea distinta o atendiendo a otro periférico.

Para soportar la gestión de la entrada y salida por medio de interrupciones, el procesador debe disponer de hardware específico. Si no dispusiera de dicho hardware, no sería posible realizar el tratamiento de la entrada y salida por medio de interrupciones.

En primer lugar, es necesario que disponga de al menos una entrada de control, llamada en ocasiones `IRQ` (*Interruption ReQuest*). Dicha entrada será la que activarán los periféricos para indicar que están generando una interrupción y que solicitan la atención del procesador.

En segundo lugar, el procesador debe ser capaz de comprobar si algún periférico ha generado una petición y de llevar a cabo las acciones necesarias para su tratamiento. La forma en la que el procesador comprueba si se ha generado una interrupción es la siguiente. Una vez completa la ejecución de la instrucción en curso, y antes de comenzar la ejecución de la siguiente instrucción, comprueba el estado de la entrada de interrupción. Si dicha entrada está activa, sabe que hay una petición de interrupción pendiente. Para atender dicha petición, transfiere el control a una rutina especial llamada rutina de tratamiento de interrupciones (RTI).

En tercer y último lugar, puesto que el tratamiento de una interrupción implica el dejar de ejecutar el programa en curso y transferir el control a la rutina de tratamiento de interrupciones, el procesador debe ser capaz de almacenar el estado en el que éste se encontraba antes de llamar a dicha rutina y de recuperar dicho estado antes de devolver el control al programa original. Es decir, el programa que estaba siendo ejecutado no debe verse afectado por el hecho de que se haya producido una interrupción.

Además de los procesos de entrada y salida, durante el funcionamiento de un computador se pueden producir otra serie de eventos que requieren ser tratados de forma inmediata por el procesador. Dichos eventos reciben el nombre, en la terminología MIPS, de *excepciones*.

Las excepciones se pueden agrupar en:

- Excepciones software (internas al procesador). Por ejemplo, un desbordamiento aritmético, una división por cero, el intento de ejecutar una instrucción con código de operación incorrecto o de referenciar una dirección de memoria prohibida, la ejecución de una instrucción de llamada al sistema operativo (syscall), etc.
- Excepciones hardware (externas al procesador). Por ejemplo, un corte de corriente o un error de acceso a memoria producida por un periférico. En este grupo también se encuentran las generadas durante el funcionamiento normal de los periféricos de entrada y salida (que se denominan, como se ha visto, interrupciones).

Destacar que, al contrario de lo que ocurre con las interrupciones propiamente dichas, el resto de excepciones se atienden en el momento en que se producen. Es decir, el procesador no espera a que la instrucción en curso sea completada.

Este capítulo muestra cómo se realiza el procesamiento de las excepciones en el procesador MIPS32. En la primera sección se describen los registros para el tratamiento de excepciones proporcionados por el simulador SPIM. En la segunda sección se describe cómo se lleva a cabo el procesamiento de las excepciones. En la tercera sección cómo ejecutar el simulador para poder realizar las prácticas propuestas con excepciones. En las dos siguientes secciones se muestra cómo capturar las excepciones producidas por desbordamiento y por direccionamiento erróneo, respectivamente. En la sexta sección se tratan las interrupciones. Por último, se proponen una serie de ejercicios más avanzados.

10.1. Registros para el tratamiento de las excepciones

En un procesador MIPS, en el llamado coprocesador 0, se encuentran los registros necesarios para el tratamiento de las excepciones. La Tabla 10.1 muestra un resumen de dichos registros.

A continuación se explica el significado y función de cada uno de dichos registros.

Registro `Status` Este registro se utiliza para almacenar información referente al modo de trabajo del procesador y el estado de las interrupciones.

El procesador puede trabajar en uno de los dos siguientes modos. El modo núcleo, que proporciona a los programas en ejecución acceso a

Nombre	Número	Función
`Status`	12	Máscara de interrupciones y bits de autorización.
`Cause`	13	Tipo de excepción y bits de interrupciones pendientes.
`EPC`	14	Dirección de memoria de la última instrucción ejecutada por el procesador.
`BadVaddr`	8	Dirección de memoria que ha provocado la excepción.
`Count`	9	Temporizador (*timer*).
`Compare`	11	Valor que debe compararse con el del registro *Count*

Tabla 10.1: Registros para el tratamiento de excepciones e interrupciones

todos los recursos del sistema, y el modo usuario, en el que el acceso a los recursos está limitado.

En cuanto al estado de las interrupciones, éstas pueden estar habilitadas o enmascaradas. Si las interrupciones están habilitadas, el procesador atenderá las peticiones de interrupción en cuanto se produzcan. Si están enmascaradas, el procesador ignorará las peticiones de interrupción.

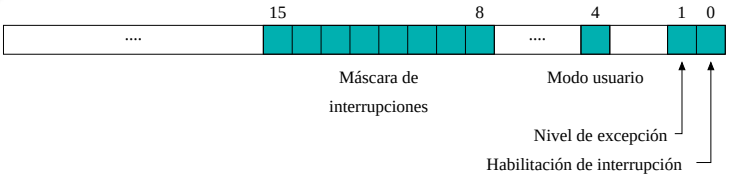


Figura 10.1: Registro `Status`

La Figura 10.1 muestra los campos del registro `Status`.

En dicho registro, el bit 4 se utiliza para indicar el modo de funcionamiento. Si está a 0, el modo de funcionamiento será el de núcleo. Si está a 1, el modo de funcionamiento será el de usuario. Puesto que el simulador SPIM no implementa el modo núcleo, el bit 4 siempre estará a 1.

El resto de bits mostrados en la Figura 10.1, es decir, los bits 0, 1 y 8 al 15, se utilizan para el tratamiento de las excepciones. El bit 0 (*habilitación de interrupciones*) se utiliza para habilitar o enmascarar, de forma

general, todas las interrupciones. Cuando se pone a 1 se habilitan las interrupciones. Cuando se pone a 0 se enmascaran las interrupciones.

El bit 1 (*nivel de excepción*) indica si actualmente se está tratando una excepción. En el momento en que una excepción comienza a ser atendida, dicho bit se pone automáticamente a 1. De esta forma, se evita que el tratamiento de una excepción sea interrumpido por una nueva excepción. La rutina de tratamiento de la excepción debe volver a colocar dicho bit a 0 antes de devolver el control al programa en curso.

MIPS32 dispone de seis niveles de prioridad para el tratamiento de excepciones hardware y dos más para excepciones software. Por tanto, es posible asignar a las distintas causas de excepción un nivel de prioridad determinado. La existencia de diferentes niveles de prioridad, permite al procesador, ante la llegada de dos o más excepciones simultáneas, atender primero a la que sea más prioritaria.

Los bits 8 al 15 (*máscara de interrupciones*) de este registro sirven para habilitar o enmascarar cada uno de dichos niveles de prioridad. Si uno de los bits de la máscara de interrupciones está a 1, se permite que las excepciones de dicho nivel interrumpan al procesador. Sin un bit de la máscara está a 0, se enmascaran las excepciones de dicho nivel.

La correspondencia entre los bits de la máscara de interrupciones y los niveles de excepción hardware y software es la siguiente. Los bits 10 al 15 de la máscara de interrupciones sirven para habilitar o enmascarar los niveles de interrupción hardware 0 al 5, respectivamente. Los bits 8 y 9 de la máscara de interrupciones sirven para habilitar o enmascarar los niveles de excepción software 0 y 1, respectivamente.

Para que una excepción de un determinado nivel de prioridad pueda ser atendida, es necesario que tanto el bit de *habilitación de interrupción* como el bit correspondiente a dicho nivel de la *máscara de interrupciones* estén a 1.

Registro `Cause` Este registro se utiliza para almacenar el código de la última excepción o interrupción generada. La Figura 10.2 muestra los campos de este registro.

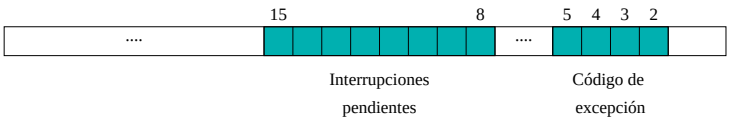


Figura 10.2: Registro `Cause`

Los ocho bits del campo *interrupciones pendientes* se corresponden con los ocho niveles de excepción: seis hardware (bits 10 al 15) y dos software (bits 8 y 9). Cada uno de estos bits se pone a 1 cuando se produce una excepción en el nivel de prioridad correspondiente. Dicho bit permanecerá a 1 hasta que la excepción correspondiente haya sido atendida. Aunque debe ser puesta a 1 explícitamente.

El campo *código de excepción* (bits 2 al 5) se utiliza para codificar la causa de la excepción. La siguiente tabla muestra los códigos utilizados para la codificación de las distintas causas de excepción soportadas.

Núm.	Nombre	Descripción
0	INT	Interrupción externa
4	ADDRL	Excepción por dirección errónea (carga o lectura de instrucción)
5	ADDRS	Excepción por dirección errónea (almacenamiento)
6	IBUS	Error de bus durante la lectura de instrucción
7	DBUS	Error de bus durante la carga o almacenamiento de datos
8	SYSCALL	Excepción por llamada al sistema (*syscall*)
9	BKPT	Excepción por punto de ruptura (*breakpoint*)
10	RI	Excepción por instrucción reservada
12	OVF	Excepción por desbordamiento aritmético

Registro `EPC` *(Exception Program Counter)* Este registro se utiliza para almacenar una dirección de memoria. Esta dirección de memoria será, en el caso de que se haya producido una excepción software, la dirección de la instrucción que ha provocado la excepción. Si en lugar de una excepción se ha producido una interrupción, la dirección de memoria almacenada en este registro será la dirección de memoria de la instrucción que debe ejecutarse después de procesar la interrupción (para continuar la ejecución del programa que haya sido interrumpido).

Registro `BadVaddr` (*Bad Virtual Address*) Cuando se produce una excepción por acceso a una dirección de memoria errónea, se almacena en este registro la dirección de memoria por la que se ha generado dicha excepción.

Registro `Count` Es un contador. Mientras el simulador esté en el modo de ejecución de instrucciones, su valor se actualiza con una frecuencia fija (cada 10 milisegundos).

Registro `Compare` Se utiliza para almacenar un número que será comparado de forma automática con el contenido del registro `Count`. Cuando el valor del registro `Count` llegue al número almacenado en este registro, se generará una interrupción de nivel 5 (máxima prioridad).

Para acceder a los registros del coprocesador 0 se utilizan las instrucciones: «**mfc0**» (***m**ove **f**rom **c**oprocessor 0*) y «**mtc0**» (***m**ove **t**o **c**oprocessor 0*). La primera de ellas, «**mfc0**», permite transferir el contenido de un registro del coprocesador 0 a uno de los registros del procesador. La segunda de ellas, «**mtc0**», permite realizar la operación inversa: transferir el contenido de un registro del procesador a uno de los registros del coprocesador 0. La sintaxis completa de dichas instrucciones es la siguiente:

- «**mfc0** rt , rd»: Transfiere el contenido del registro `rd` del coprocesador 0 al registro `rt` del procesador.
- «**mtc0** rd, rt »: Transfiere el contenido del registro `rd` del procesador al registro `rt` del coprocesador 0.

EJERCICIOS

▶ **218** Ejecuta el simulador SPIM y comprueba el contenido inicial del registro `Status`.

▶ **219** Considerando el valor original del registro `Status`, ¿qué valor se debería cargar en dicho registro si se pretenden habilitar las interrupciones de nivel 0 y 1 y enmascarar las restantes? Enumera las instrucciones que llevarían a cabo dicha operación y comenta brevemente su cometido.

▶ **220** Considerando el valor original del registro `Status`, ¿qué valor se debería cargar en dicho registro si se quieren enmascarar todas las interrupciones? Enumera las instrucciones que llevarían a cabo dicha operación.

▶ **221** ¿Qué valor se almacenará en el registro `EPC` si se produce una excepción mientras se ejecuta la instrucción almacenada en `0x0004 0020`?

▶ **222** ¿Qué valores se almacenarán en los registros `EPC` y `BadVAddr`, cuando se ejecute la instrucción «**lw** $t0 ,11($0)», almacenada en la posición de memoria `0x0004 0040`?

10.2. Tratamiento de las excepciones

Para poder tratar adecuadamente una excepción es necesario conocer en primer lugar quién la ha generado.

Para identificar quién ha generado una excepción, algunas arquitecturas, como las del 8086 de Intel o el MC68000 de Motorola, implementan interrupciones *vectorizadas*. Se llaman así porque se asigna a cada causa de excepción ó dispositivo un número que la identifica y que se corresponde con el índice de un vector. En dicho vector se encuentran almacenadas las direcciones de comienzo de las rutinas de tratamiento de excepciones asociadas a las diferentes causas de excepción.

En este caso, el problema de identificación de qué dispositivo periférico ha generado la interrupción se convierte en un problema de asignación de los números de interrupción disponibles entre los periféricos existentes. Originalmente, dicha asignación se realizaba de forma manual. Había que variar la posición de determinados *jumpers* en las placas de los dispositivos para indicar qué número de IRQ se asignaba al dispositivo. Actualmente, los dispositivos —y los buses— están preparados para poder variar dicho recurso de forma electrónica. Así, la BIOS del computador o el Sistema Operativo pueden modificar las IRQs asignadas a los diferentes dispositivos de forma automática para resolver los conflictos que puedan surgir en la asignación de IRQs.

Una estrategia diferente, y que es la utilizada por la arquitectura MIPS32, es la utilización de una gestión *no vectorizadas*. En este caso, no hay que asignar números a las distintas causas de excepciones. A cambio, y puesto que no existe dicha diferenciación, la rutina de tratamiento de excepciones deberá encargarse de, en una primera fase, averiguar quién ha generado la excepción.

Para describir los procesos que se llevan a cabo durante una excepción se van a considerar los siguientes momentos:

- El instante en el que se produce.
- El instante en el que el procesador decide tratar la excepción.
- El tratamiento de la excepción propiamente dicho.

El instante en el que se produce una excepción En el instante en el que se produce una excepción, se almacena en el campo *código de excepción* del registro `Cause` el código que identifica la causa de dicha excepción. Además, si es una interrupción, en el campo de *Interrupciones pendientes* del registro `Cause` se activa el bit correspondiente al nivel de prioridad de dicha excepción.

El instante en el que el procesador decide procesar la excepción. En el instante en el que el procesador decide procesar la excepción, almacena el

estado actual y genera un salto a la dirección de memoria donde está almacenada la rutina de tratamiento de excepciones.

Generar el salto a la rutina de tratamiento de excepciones conlleva las siguientes acciones:

1. Almacenamiento en el registro `EPC` de una de dos: si la excepción se ha producido al intentar ejecutar una determinada instrucción, se almacena la dirección de la instrucción causante de la excepción; si la excepción ha sido motivada por una interrupción, se almacena la dirección de memoria de la instrucción siguiente a la que acaba de ser ejecutada.
2. Si la excepción ha sido provocada por el acceso a una posición de memoria errónea, almacenamiento en el registro `BadVAddr` de la dirección de memoria a la que se ha intentado acceder.
3. Escritura de un 1 en el bit *nivel de excepción* del registro `Status`.
4. Almacenamiento en el registro `PC` de la dirección de comienzo de la rutina de tratamiento de excepciones. En el caso del simulador SPIM, la dirección de comienzo de la rutina de tratamiento de excepciones está en la posición de memoria `0x8000 0180`.

El tratamiento de la excepción propiamente dicho. A partir de este momento se lleva a cabo la ejecución de la rutina de tratamiento de excepciones. Dicha rutina forma parte del sistema operativo instalado en el computador. En el caso del simulador SPIM, el código fuente de dicha rutina se proporciona en el fichero `exceptions.s`.

La primera de las tareas que debe realizar la rutina de tratamiento de excepciones es la de identificar quién ha generado la excepción que se está atendiendo. Para ello, debe utilizar el contenido del registro `Cause` para determinar la causa de la excepción y transferir el flujo del programa a la parte de la rutina que se encarga del tratamiento de dicho tipo de excepciones.

Una vez encontrado el causante de la excepción y habiendo realizado las acciones que dicha excepción requiera, el siguiente paso consiste en decidir si:

- debe abortar la ejecución del programa interrumpido (sólo si éste ha sido el causante de la excepción), o
- debe continuar la ejecución del programa interrumpido.

Dependiendo de un caso u otro se deberá escribir una dirección de vuelta distinta en el registro `EPC`. En el primer caso, deberá darse el control al Sistema Operativo, por lo que se realizará una llamada al mismo para finalizar la ejecución del programa en curso.

En el segundo de los casos, cuando se deba continuar la ejecución del programa interrumpido, se deberá poner en el registro `EPC` la dirección siguiente a la instrucción que se estaba ejecutando (caso de una excepción software) o se había ejecutado (caso de una interrupción) cuando el programa fue interrumpido.

Por tanto, si se debe continuar la ejecución del programa interrumpido y se estaba tratando una excepción, se deberá incrementar en 4 el contenido del registro `EPC`. Ya que en este caso, el registro `EPC` contiene la dirección de la instrucción causante de la excepción y se quiere retornar a la siguiente instrucción a ésta.

Por otro lado, si se debe continuar la ejecución del programa interrumpido y se estaba tratando una interrupción, no hay que hacer nada con el `EPC`. Ya que el contenido del registro `EPC` es justamente la dirección de la siguiente instrucción que se debía ejecutar cuando el programa fue interrumpido.

Una vez se ha puesto en el registro `EPC` la dirección correcta de retorno, se debe poner a cero el registro `Cause` y a continuación llevar a cabo el retorno de la rutina de tratamientos de excepciones.

Para llevar a cabo el retorno de la rutina de tratamiento de excepciones se debe utilizar la instrucción «**eret**». La ejecución de dicha instrucción provoca las siguientes acciones:

1. Pone a 0 el bit *nivel de excepción* del registro `Status`.
2. Carga en el registro `PC` el valor almacenado en el registro `EPC`.

Y con esta instrucción finaliza la rutina de tratamiento de excepciones.

A continuación se muestra una rutina de tratamiento de excepciones basada en la proporcionada con el simulador SPIM.

✎ exceptions.s

```
1 #
2 # Rutina de tratamiento de excepciones y programa cargador
3 #   (Derivados de la rutina y cargador proporcionados con SPIM)
4 #
5
6 #================================================================
7 # Rutina de tratamiento de las excepciones
```

```
8  #=====================================================================
9  #
10 # Esta rutina de tratamiento de excepciones se ejecuta cuando ocurre
11 # una excepción. Distingue entre excepciones que no son interrupciones e
12 # interrupciones. Simplemente imprime un mensaje indicado qué tipo de
13 # excepción se ha generado.
14 #
15
16         .kdata
17 _exc_:  .asciiz "␣*␣Se␣ha␣generado␣una␣excepción␣software␣*\n"
18 _int_:  .asciiz "␣*␣Se␣ha␣generado␣una␣interrupción␣*\n"
19 s1:     .word 0
20 s2:     .word 0
21
22
23 # Debido a que la rutina se ejecuta en el kernel, se pueden utilizar
24 # los registros $k0 y $k1 sin almacenar sus valores antiguos. El resto
25 # de registros utilizados deberán almacenarse y restaurarse.
26
27         .ktext 0x80000180       # Comienzo del vector de excepciones
28
29         #---------------------------------------------------------------
30         # Los registros que vayan a utilizarse deberán guardarse para
31         # posteriormente restaurarse. No se puede utilizar la pila.
32         # Se guardan:
33         #  * El registro $at en el registro $k1
34         #  * El registro $v0 en la posición de memoria s1
35         #  * El registro $a0 en la posicion de memoria s2
36         .set noat
37         move $k1, $at
38         .set at
39         sw $v0, s1
40         sw $a0, s2
41         #---------------------------------------------------------------
42
43         #---------------------------------------------------------------
44         # Determinación de la causa de la excepción
45         mfc0 $k0, $13           # Registro Cause
46         srl  $a0, $k0, 2        # Extracción del Código de Excepción
47         andi $a0, $a0, 0x1f
48         #---------------------------------------------------------------
49
50         #---------------------------------------------------------------
51         # Si el Código de Excepción es 0, es una interrupción
52         bne  $a0, 0, exc_sw
53         nop
54         #---------------------------------------------------------------
55
56
57 int:    #=============================================================
58         # Tratamiento de interrupciones
59         li $v0, 4               # syscall 4 (print_str)
60         la $a0, _int_
61         syscall
62         j  ret
63         #=============================================================
64
65
66 exc_sw: #=============================================================
67         # Tratamiento de excepciones (no interrupciones)
68         li $v0, 4               # syscall 4 (print_str)
69         la $a0, _exc_
70         syscall
71         mfc0  $k0, $14          # +
72         addiu $k0, $k0, 4       # | Excepción: EPC<-EPC+4
73         mtc0  $k0, $14          # +
74         #=============================================================
```

```
75
76
77  ret:    #-----------------------------------------------------------------
78          # Restauración de los registros utilizados por la rutina
79          lw $v0, s1
80          lw $a0, s2
81          .set noat
82          move $at $k1
83          .set at
84          #-----------------------------------------------------------------
85
86          #-----------------------------------------------------------------
87          # Restauración del estado
88          mtc0 $0, $13            # Puesta a 0 del registro Cause
89          #-----------------------------------------------------------------
90
91          #-----------------------------------------------------------------
92          # Retorno de la rutina de tratamiento de excepciones
93          eret
94          #-----------------------------------------------------------------
95
96
97
98  #=====================================================================
99  # Programa cargador. Invoca la rutina "main" con los argumentos:
100 #       main(argc, argv, envp)
101 #=====================================================================
102         .text
103         .globl __start
104 __start:
105         lw $a0 0($sp)           # argc
106         addiu $a1 $sp 4         # argv
107         addiu $a2 $a1 4         # envp
108         sll $v0 $a0 2
109         addu $a2 $a2 $v0
110         jal main
111         nop
112
113         li $v0 10
114         syscall                 # syscall 10 (exit)
115
116         .globl __eoth
117 __eoth:
```

EJERCICIOS

▶ **223** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
39          sw $v0, s1
40          sw $a0, s2
```

▶ **224** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
43          #-----------------------------------------------------------------
44          # Determinación de la causa de la excepción
45          mfc0 $k0, $13           # Registro Cause
46          srl  $a0, $k0, 2        # Extracción del Código de Excepción
47          andi $a0, $a0, 0x1f
48          #-----------------------------------------------------------------
49
50          #-----------------------------------------------------------------
51          # Si el Código de Excepción es 0, es una interrupción
52          bne  $a0, 0, exc_sw
```

```
53          nop
54          #---------------------------------------------------------------
```

▶ **225** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
57  int:    #===============================================================
58          # Tratamiento de interrupciones
59          li $v0, 4               # syscall 4 (print_str)
60          la $a0, _int_
61          syscall
62          j   ret
63          #===============================================================
```

▶ **226** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
66  exc_sw: #===============================================================
67          # Tratamiento de excepciones (no interrupciones)
68          li $v0, 4               # syscall 4 (print_str)
69          la $a0, _exc_
70          syscall
71          mfc0   $k0, $14         # +
72          addiu  $k0, $k0, 4      # | Excepción: EPC<-EPC+4
73          mtc0   $k0, $14         # +
74          #===============================================================
```

▶ **227** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
78          # Restauración de los registros utilizados por la rutina
79          lw $v0, s1
80          lw $a0, s2
```

▶ **228** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
86          #---------------------------------------------------------------
87          # Restauración del estado
88          mtc0 $0, $13            # Puesta a 0 del registro Cause
89          #---------------------------------------------------------------
```

▶ **229** ¿Qué función tiene el siguiente fragmento de `exceptions.s`?

```
91          #---------------------------------------------------------------
92          # Retorno de la rutina de tratamiento de excepciones
93          eret
94          #---------------------------------------------------------------
```

▶ **230** Una vez analizada paso a paso la rutina de tratamiento de excepciones del simulador SPIM, explica de forma resumida qué acciones se llevan a cabo para tratar las excepciones e interrupciones.

. . . . . .

10.3. Opciones del programa SPIM para la simulación de interrupciones

Para poder realizar correctamente los ejercicios propuestos en las siguientes secciones, es necesario ejecutar el simulador SPIM evitando que cargue la rutina por defecto de tratamiento de excepciones y habilitando el mapeo a memoria de la entrada y salida.

En la versión de SPIM para GNU/Linux, ésto se consigue ejecutando XSPIM con la siguiente línea de comandos:

```
«xspim -noexception -mapped_io»
```

En la versión de SPIM para Windows, lo anterior se consigue desactivando la opción *Load trap file* y activando la opción *Mapped I/O* en el cuadro de diálogo *Settings* (véase la Figura 10.3).

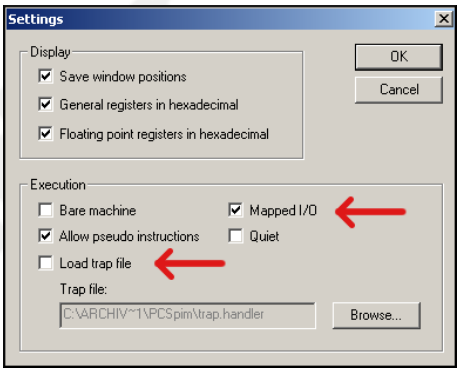


Figura 10.3: Cuadro de diálogo de preferencias de la versión para Windows de SPIM

Una vez hecho lo anterior, para cargar la nueva rutina de tratamiento de excepciones y el código que se desee probar, es necesario cargar en primer lugar la nueva versión de la rutina y después el programa que se quiere probar.

10.4. Excepciones software

Las excepciones software se producen por errores que ocurren durante la ejecución de una instrucción o son generados por instrucciones específicas. Las causas de estos errores pueden ser muy diversas, entre ellas: desbordamiento aritmético, error de direccionamiento, ejecución de una instrucción ilegal, división por cero, etc.

En los dos siguientes apartados se estudian dos de las excepciones más comunes que se pueden producir durante la ejecución de una instrucción: la de

error por desbordamiento y la de error por direccionamiento erróneo.

10.4.1. Tratamiento de la excepción por desbordamiento

Se dice que se ha producido un desbordamiento en una operación aritmética cuando el resultado no puede ser correctamente representado. Puesto que el resultado obtenido no es el correcto, el procesador debe avisar de este error. La forma en la que lo hace es generando una excepción por desbordamiento.

El objetivo de los siguientes ejercicios es justamente el de explorar la forma en la que se tratan las excepciones por desbordamiento en el simulador SPIM y proponer una modificación a la rutina de tratamiento de excepciones que finalice un programa si éste provoca una excepción por desbordamiento.

EJERCICIOS

▶ **231** Examina el código del programa `exceptions.s` mostrado en la sección 10.2 y explica qué ocurrirá cuando se produzca una excepción por desbordamiento.

▶ **232** Desarrolla un pequeño programa en ensamblador que sume los 10 elementos del siguiente vector y almacene el resultado en una variable «suma».

$$V = (\ \texttt{0x0000 0001}, \texttt{0x0000 0002}, \texttt{0x7FFF FFFF}, \texttt{0x0000 0003}, \texttt{0x0000 0004}, \texttt{0x0000 0005}, \texttt{0x0000 0006}, \texttt{0x0000 0007}, \texttt{0x0000 0008}, \texttt{0x0000 0009}\ )$$

▶ **233** Ejecuta el programa, ¿qué se almacena en la variable «suma»? ¿Qué ha ocurrido durante la ejecución de dicho programa?

▶ **234** ¿Qué valor se carga en el registro `EPC` cuando se produce la excepción?

▶ **235** ¿Qué valor se carga en el registro `Cause` cuando se produce la excepción? Comprueba que el campo *código de excepción* contiene el valor 12.

EJERCICIOS DE AMPLIACIÓN

▶ **236** Modifica la rutina `exceptions.s` para que el simulador SPIM para que cuando se produzca una excepción por desbordamiento, aparezca un mensaje que lo indique y finalice la ejecución del programa en curso. Para probar su funcionamiento emplea el código del ejercicio 232.

10.4.2. Tratamiento de la excepción por direccionamiento erróneo en almacenamiento

La arquitectura MIPS32 obliga a que las direcciones de memoria estén alineadas en múltiplos de 2 para las medias palabras (2 bytes) y en múltiplos de 4 para las palabras (4 bytes). Es decir, una media palabra tan sólo puede escribirse o leerse de una dirección de memoria múltiplo de 2. De igual forma, una palabra tan sólo puede escribirse o leerse de una dirección de memoria múltiplo de 4. Por tanto, una instrucción como «**lw** $t0, 1($0)» no sería correcta ya que intentaría leer una palabra de una dirección de memoria (la 1) que no es múltiplo de 4. Por contra, «**lw** $t0, 0($0)» y «**lw** $t0, 4($0)», sí que serían correctas ya que leen una palabra de la posición de memoria 0 y 4, respectivamente.

Puesto que la implementación de una arquitectura MIPS32 no permite el acceso a direcciones de memoria no alineadas, si una determinada instrucción intenta acceder a una posición de memoria no alineada, el procesador deberá avisar de este hecho. La forma en la que lo hace es generando una excepción por direccionamiento erróneo.

El objetivo de los siguientes ejercicios es justamente el de explorar la forma en la que se tratan las excepciones por direccionamiento erróneo en el simulador SPIM y proponer una modificación a la rutina de tratamiento de excepciones que finalice un programa si éste provoca una excepción por direccionamiento erróneo.

EJERCICIOS

▶ **237** Examina el código del programa `exceptions.s` mostrado en la sección 10.2 y explica qué ocurrirá cuando se produzca una excepción por direccionamiento erróneo en almacenamiento.

▶ **238** Desarrolla un sencillo programa en ensamblador en el que se produzca un error de direccionamiento durante su ejecución.

▶ **239** ¿Termina correctamente la ejecución del programa?

▶ **240** ¿Qué valor se almacena en el registro `EPC` cuando se produce la excepción?

▶ **241** ¿Qué valor se carga en el registro `Cause` cuando se produce la excepción? Comprueba que el campo *código de excepción* contiene el valor 5.

▶ **242** ¿Qué valor se almacena en el registro `BadVaddr` cuando se produce la excepción?

EJERCICIOS DE AMPLIACIÓN

▶ 243 Modifica la rutina `exceptions.s` para que cuando se produzca una excepción por direccionamiento erróneo, aparezca un mensaje que lo indique y finalice la ejecución del programa en curso. Para probar su funcionamiento emplea el código del ejercicio 238.

10.5. Interrupciones

En las dos secciones anteriores se ha visto como el procesador genera dos tipos de excepciones ante los errores de desbordamiento y de direccionamiento erróneo. Y como la rutina de tratamiento de excepciones puede reaccionar ante dichas excepciones, por ejemplo, finalizando la ejecución de los programas que las han provocado.

En esta sección se verá como se puede gestionar la entrada y salida de un computador utilizando interrupciones. En este caso no será la ejecución de una instrucción la que provoque la excepción. Si no que serán los dispositivos los que avisen al procesador de que requieren de su atención generando una petición de interrupción.

SPIM simula tres dispositivos capaces de generar interrupciones: un reloj programable, un teclado y una pantalla. En las siguientes secciones se proporcionan ejercicios que muestran como gestionar dichos dispositivos utilizando interrupciones.

10.5.1. Reloj programable

El simulador SPIM proporciona un reloj programable por medio de los registros `Count` y `Compare` del coprocesador 0. El registro `Count` es un contador ascendente no cíclico que incrementa su valor en una unidad cada 10 milisegundos. Cuando el contenido de este registro alcanza el valor almacenado en el registro `Compare`, el coprocesador 0 genera una interrupción, llamada de reloj, con nivel de prioridad 5.

Para habilitar la interrupción de reloj, es decir, para que el procesador atienda la interrupción que se genera cuando el valor de `Count` alcanza el de `Compare`, se deberán poner a 1 los bits 15 y 0 del registro `Status`. Para enmascarar la interrupción de reloj, se deberá poner a 0 el bit 15 del registro `Status` (conviene recordar que poner a 0 el bit 0 implicaría enmascarar todas las interrupciones, no sólo la de reloj).

El funcionamiento de este reloj programable se describe a continuación. El registro `Count` no está siempre contando. Es necesario inicializarlo. Para ha-

cerlo, se debe habilitar la interrupción de reloj y escribir en el registro `Count` el valor inicial del contador (que habitualmente será 0). Una vez inicializado el contador, el registro `Count` incrementa su valor cada 10 ms hasta que alcanza el valor almacenado en el registro `Compare`. Cuando lo hace, se escribe un 0 en el campo *código de excepción* del registro `Cause` y un 1 en el bit 15 del mismo registro y se genera una interrupción de nivel de prioridad 5. El procesador atenderá dicha petición de interrupción en cuanto complete la ejecución de la instrucción en curso.

Es fácil ver, según lo anterior, que el tiempo que transcurre desde que se inicializa el contador hasta que el procesador atiende la interrupción se puede calcular como:

$$Tiempo = (Compare - Initial\_Count) \times 10ms$$

Para poder comprobar como se utiliza el reloj programable de SPIM se proporcionan los siguientes listados. El primero de ellos, `codigo-reloj.s`, inicializa el reloj programable para después entrar en un bucle sin fin en el que imprime por pantalla la frase «Esto es una prueba». El segundo de los listados presentados, `exceptions-2.s`, es una modificación del fichero `exceptions.s` en la que se han añadido las instrucciones necesarias para poder atender la interrupción de reloj.

✎ codigo-reloj.s

```
1  #
2  # Programa que utiliza el reloj programable
3  #
4          .data
5  cadena: .asciiz "Esto es una prueba\n"
6
7          .text
8  main:
9          # Habilitación de la interrupción de reloj
10         mfc0 $t0, $12
11         ori  $t0, $t0, 0x8001
12         mtc0 $t0, $12
13
14         # Inicialización del contador
15         addi $t2, $0, 100
16         mtc0 $t2, $11
17         mtc0 $0,  $9
18
19         # Bucle infinito que imprime la cadena "Esto es un prueba"
20 bucle: la   $a0, cadena
21         li   $v0, 4
22         syscall
23         addi $t7, $0, 30000
24         sll  $t7, $t7, 3      # $t7 <- 30000*8
25 retardo: addi $t7, $t7, -1
26         bnez $t7, retardo
27         beqz $0,  bucle
```

exceptions-2.s

```
#
# Rutina de tratamiento de excepciones y programa cargador
#   (Derivados de la rutina y cargador proporcionados con SPIM)
#

#============================================================================
# Rutina de tratamiento de las excepciones
#============================================================================
#
# Esta rutina de tratamiento de excepciones se ejecuta cuando ocurre
# una excepción. Distingue entre excepciones que no son interrupciones e
# interrupciones. Simplemente imprime un mensaje indicado qué tipo de
# excepción se ha generado.
#

        .kdata
_exc_:  .asciiz " * Se ha generado una excepción software *\n"
_int_:  .asciiz " * Se ha generado una interrupción *\n"
_clk_:  .asciiz " * Se ha generado una interrupción de reloj *\n"
s1:     .word 0
s2:     .word 0


# Debido a que la rutina se ejecuta en el kernel, se pueden utilizar
# los registros $k0 y $k1 sin almacenar sus valores antiguos. El resto
# de registros utilizados deberán almacenarse y restaurarse.

        .ktext 0x80000180       # Comienzo del vector de excepciones

        #-----------------------------------------------------------------
        # Los registros que vayan a utilizarse deberán guardarse para
        # posteriormente restaurarse. No se puede utilizar la pila.
        # Se guardan:
        #  * El registro $at en el registro $k1
        #  * El registro $v0 en la posición de memoria s1
        #  * El registro $a0 en la posicion de memoria s2
        .set noat
        move $k1, $at
        .set at
        sw $v0, s1
        sw $a0, s2
        #-----------------------------------------------------------------

        #-----------------------------------------------------------------
        # Determinación de la causa de la excepción
        mfc0 $k0, $13           # Registro Cause
        srl  $a0, $k0, 2        # Extracción del Código de Excepción
        andi $a0, $a0, 0x1f
        #-----------------------------------------------------------------

        #-----------------------------------------------------------------
        # Si el Código de Excepción es 0, es una interrupción
        bne  $a0, 0, exc_sw
        nop
        #-----------------------------------------------------------------


int:    #-----------------------------------------------------------------
        # Tratamiento de interrupciones

        andi $a0, $k0, 0x8000   # ¿Es de reloj?
        bne  $a0, $0, intclk    #

        li $v0, 4               # Interrupción no reconocida
        la $a0, _int_
        syscall
```

```
67          j   ret
68
69  intclk: li $v0, 4               # +
70          la $a0, _clk_           # | Mensaje de interrupción de reloj
71          syscall                 # +
72          addi $k0, $0, 100       # +
73          mtc0 $k0, $11           # | Vuelta a programar el reloj
74          mtc0 $0, $9             # +
75          j    ret
76          #================================================================
77
78
79  exc_sw: #================================================================
80          # Tratamiento de excepciones (no interrupciones)
81          li $v0, 4               # syscall 4 (print_str)
82          la $a0, _exc_
83          syscall
84          mfc0  $k0, $14          # +
85          addiu $k0, $k0, 4       # | Excepción: EPC<-EPC+4
86          mtc0  $k0, $14          # +
87          #================================================================
88
89
90  ret:    #----------------------------------------------------------------
91          # Restauración de los registros utilizados por la rutina
92          lw $v0, s1
93          lw $a0, s2
94          .set noat
95          move $at $k1
96          .set at
97          #----------------------------------------------------------------
98
99          #----------------------------------------------------------------
100         # Restauración del estado
101         mtc0 $0, $13            # Puesta a 0 del registro Cause
102         #----------------------------------------------------------------
103
104         #----------------------------------------------------------------
105         # Retorno de la rutina de tratamiento de excepciones
106         eret
107         #----------------------------------------------------------------
108
109
110
111 #=====================================================================
112 # Programa cargador. Invoca la rutina "main" con los argumentos:
113 #       main(argc, argv, envp)
114 #=====================================================================
115         .text
116         .globl __start
117 __start:
118         lw $a0 0($sp)           # argc
119         addiu $a1 $sp 4         # argv
120         addiu $a2 $a1 4         # envp
121         sll $v0 $a0 2
122         addu $a2 $a2 $v0
123         jal main
124         nop
125
126         li $v0 10
127         syscall                 # syscall 10 (exit)
128
129         .globl __eoth
130 __eoth:
```

Recuerda lo comentado en la sección 10.3 sobre la ejecución de SPIM

para la simulación de interrupciones y sobre la carga del nuevo fichero de interrupciones antes del código que se quiere ejecutar.

Para detener el bucle sin fin en el que entrará el simulador, se puede pulsar la combinación de teclas `Ctrl+C`.

EJERCICIOS

▶ **244** ¿Qué instrucciones del programa `codigo-reloj.s` habilitan la interrupción de reloj? ¿Qué bits del registro `Status` se ponen a 1?

▶ **245** ¿Qué instrucciones del programa `codigo-reloj.s` inicializan el reloj? ¿Cuánto tiempo pasará hasta que se genere la primera interrupción de reloj?

▶ **246** ¿Qué hace el simulador entre interrupciones de reloj?

▶ **247** Estudia con detalle el código de la rutina `exceptions-2.s`. Explica cómo se captura la interrupción de reloj y qué cambios se han añadido con respecto a la rutina `exceptions.s`. Justifica dichos cambios.

▶ **248** Detalla los procesos que tienen lugar cuando se produce una interrupción de reloj.

▶ **249** Explica cómo se ejecuta la rutina `exceptions-2.s` si el procesador está ejecutando el programa `codigo-reloj.s`.

▶ **250** Modifica los códigos presentados para que se generen interrupciones de reloj cada 20 segundos.

▶ **251** La interrupción de reloj también puede utilizarse para evitar que los programas se queden en un bucle sin fin o para forzar que terminen en caso de sobrepasar un determinado tiempo de ejecución. En estos casos, se programa el temporizador para que genere una interrupción cuando se haya sobrepasado el tiempo máximo de ejecución. Si el programa no ha finalizado en el tiempo establecido, se generará una interrupción de reloj y la rutina de tratamiento de excepciones finalizará la ejecución del programa. Si por el contrario, el programa ha finalizado antes de que se genere la interrupción de reloj, deberá deshabilitar la interrupción de reloj.

Modifica el programa `codigo-reloj.s` para que programe una interrupción de reloj transcurrido un minuto. Además, modifica el programa `exceptions-2.s` para que cuando atienda dicha interrupción de reloj, finalice la ejecución del programa en curso.

10.5.2. Teclado y pantalla

SPIM simula dos dispositivos de entrada/salida que pueden gestionarse por medio de interrupciones: un teclado y una pantalla. El teclado permite comprobar cómo se realizaría la gestión de un dispositivo de entrada por medio de interrupciones. De igual forma, la pantalla permite comprobar cómo se realizaría la gestión de un dispositivo de salida por medio de interrupciones.

Ambos dispositivos permiten ser instruidos sobre si deben generar o no interrupciones. Cada uno de dichos dispositivos posee un registro de `Control`. El bit 1 de dicho registro determina si el dispositivo en cuestión debe generar o no interrupciones. Si dicho bit está a 1 el dispositivo generará interrupciones. En caso contrario, no lo hará. Así, si se quiere gestionar el teclado por medio de interrupciones se deberá poner a 1 el bit 1 de su registro de `Control`.

Si se activa el manejo por interrupciones del teclado, el controlador del teclado generará una interrupción cada vez que se pulse una tecla. De esta forma, no es necesario que el procesador interrogue periódicamente al teclado para saber si se ha pulsado una tecla o no.

Por otro lado, si se activa el manejo por interrupciones de la pantalla, el controlador de la pantalla generará una interrupción cuando el dato escrito en su registro de `Datos` haya sido impreso. De esta forma, no es necesario que el procesador interrogue periódicamente a la pantalla para saber si ya ha terminado de escribir el último dato enviado o todavía no.

En cuanto al nivel de prioridad de las interrupciones generadas, el teclado y la pantalla tienen asignados dos niveles de prioridad distintos. El teclado tiene asignado el nivel de prioridad 1 y la pantalla el nivel de prioridad 0. Por tanto, cuando el controlador del teclado genere una interrupción, se pondrá a 1 el bit correspondiente al nivel de prioridad 1 del campo de *interrupciones pendientes* del registro `Cause`. Esto es, se pondrá a 1 el bit 11 del registro `Cause`. De forma similar, cuando el controlador de la pantalla genere una interrupción, se pondrá a 1 el bit correspondiente al nivel de prioridad 0 del campo de *interrupciones pendientes* del registro `Cause`. Esto es, se pondrá a 1 el bit 10 del registro `Cause`.

Además, cuando el teclado o la pantalla generen una interrupción, también se pondrá a 0 el campo *código de excepción* del registro `Cause`. Dicho código de excepción indica, tal y como se ha visto anteriormente, que se ha producido una interrupción.

EJERCICIOS

▶ **252** Se ha explicado cómo se puede activar o desactivar la generación de interrupciones en los dispositivos simulados por SPIM: modificando el bit 1 del registro de `Control` de dichos dispositivos.

Sin embargo, para gestionar los dispositivos de entrada/salida por medio de interrupciones, no basta con que éstos generen las interrupciones. Es necesario, además, habilitar la atención de dichas interrupciones en el procesador.

Explica qué se tendría que hacer para habilitar en el procesador las interrupciones de teclado y pantalla.

▶ **253** Escribe el código necesario para habilitar en el procesador la interrupción de teclado y enmascarar la de la pantalla.

▶ **254** Escribe el código necesario para enmascarar las interrupciones tanto de pantalla como de teclado, manteniendo el resto de interrupciones tal y como estén.

Entrada de datos gestionada mediante interrupciones

El siguiente programa, `exceptions-3.s`, es una nueva modificación del código de la rutina de tratamiento de excepciones. Tiene por objeto capturar la interrupción del teclado y gestionar la entrada de datos (del teclado) por interrupciones. Analiza la nueva versión de la rutina de tratamiento de excepciones y contesta a las cuestiones que aparecerán a continuación.

✎ exceptions-3.s

```
#
# Rutina de tratamiento de excepciones y programa cargador
#   (Derivados de la rutina y cargador proporcionados con SPIM)
#

#============================================================================
# Rutina de tratamiento de las excepciones
#============================================================================
#
# Esta rutina de tratamiento de excepciones se ejecuta cuando ocurre
# una excepción. Distingue entre excepciones que no son interrupciones e
# interrupciones. Simplemente imprime un mensaje indicado qué tipo de
# excepción se ha generado.
#

        .kdata
_exc_:  .asciiz "\n*\nSe\nha\ngenerado\nuna\nexcepción\nsoftware\n*\n"
_int_:  .asciiz "\n*\nSe\nha\ngenerado\nuna\ninterrupción\n*\n"
_kbd_:  .asciiz "\n*\nSe\nha\ngenerado\nuna\ninterrupción\nde\nteclado\n*\n"
s1:     .word 0
s2:     .word 0


# Debido a que la rutina se ejecuta en el kernel, se pueden utilizar
# los registros $k0 y $k1 sin almacenar sus valores antiguos. El resto
# de registros utilizados deberán almacenarse y restaurarse.

        .ktext 0x80000180       # Comienzo del vector de excepciones

        #-----------------------------------------------------------------
        # Los registros que vayan a utilizarse deberán guardarse para
        # posteriormente restaurarse. No se puede utilizar la pila.
```

```
33          # Se guardan:
34          #  * El registro $at en el registro $k1
35          #  * El registro $v0 en la posición de memoria s1
36          #  * El registro $a0 en la posicion de memoria s2
37          .set noat
38          move $k1, $at
39          .set at
40          sw $v0, s1
41          sw $a0, s2
42          #-------------------------------------------------------------
43
44          #-------------------------------------------------------------
45          # Determinación de la causa de la excepción
46          mfc0 $k0, $13           # Registro Cause
47          srl  $a0, $k0, 2        # Extracción del Código de Excepción
48          andi $a0, $a0, 0x1f
49          #-------------------------------------------------------------
50
51          #-------------------------------------------------------------
52          # Si el Código de Excepción es 0, es una interrupción
53          bne  $a0, 0, exc_sw
54          nop
55          #-------------------------------------------------------------
56
57
58  int:    #=============================================================
59          # Tratamiento de interrupciones
60
61          andi $a0, $k0, 0x0800   # ¿Es de teclado?
62          bne  $a0, $0, intkbd    #
63
64          li $v0, 4               # Interrupción no reconocida
65          la $a0, _int_
66          syscall
67          j  ret
68
69  intkbd: li $v0, 4               # +
70          la $a0, _kbd_           # | Mensaje de interrupción de teclado
71          syscall                 # +
72          j    ret
73          #=============================================================
74
75
76  exc_sw: #=============================================================
77          # Tratamiento de excepciones (no interrupciones)
78          li $v0, 4               # syscall 4 (print_str)
79          la $a0, _exc_
80          syscall
81          mfc0  $k0, $14          # +
82          addiu $k0, $k0, 4       # | Excepción: EPC<-EPC+4
83          mtc0  $k0, $14          # +
84          #=============================================================
85
86
87  ret:    #-------------------------------------------------------------
88          # Restauración de los registros utilizados por la rutina
89          lw $v0, s1
90          lw $a0, s2
91          .set noat
92          move $at $k1
93          .set at
94          #-------------------------------------------------------------
95
96          #-------------------------------------------------------------
97          # Restauración del estado
98          mtc0 $0, $13            # Puesta a 0 del registro Cause
99          #-------------------------------------------------------------
```

```
100
101         #-----------------------------------------------------------
102         # Retorno de la rutina de tratamiento de excepciones
103         eret
104         #-----------------------------------------------------------
105
106
107
108 #=========================================================================
109 # Programa cargador. Invoca la rutina "main" con los argumentos:
110 #       main(argc, argv, envp)
111 #=========================================================================
112         .text
113         .globl __start
114 __start:
115         lw $a0 0($sp)           # argc
116         addiu $a1 $sp 4         # argv
117         addiu $a2 $a1 4         # envp
118         sll $v0 $a0 2
119         addu $a2 $a2 $v0
120         jal main
121         nop
122
123         li $v0 10
124         syscall                 # syscall 10 (exit)
125
126         .globl __eoth
127 __eoth:
```

EJERCICIOS

▶ **255** Analiza la rutina `exceptions-3.s`. Explica las diferencias con la rutina `exceptions-2.s`.

▶ **256** Explica de qué tipo de son las excepciones que trata el programa `exceptions-3.s` y en qué consiste su tratamiento.

Para gestionar mediante interrupciones la entrada de datos desde un teclado es necesario habilitar las interrupciones de teclado tanto en el procesador como en el controlador del teclado. El código necesario para llevar a cabo dichas operaciones se muestra en el siguiente programa.

✎ `codigo-teclado.s`

```
1  #
2  # Programa que gestiona el teclado por medio de interrupciones
3  #
4           .data
5  cadena:  .asciiz  "Esto es una prueba\n"
6
7           .text
8
9  main:    # $t1 <- dirección base de los controladores (0xffff 0000)
10          lui   $t1, 0xffff
11
12          # Activación de las interrupciones en el teclado
13          lw    $t0, 0($t1)
14          ori   $t0, $t0, 2
15          sw    $t0, 0($t1)
16
```

```
17          # Habilitación de las interrupciones de nivel de prioridad uno
18          mfc0 $t0, $12
19          ori  $t0, $t0, 0x801
20          mtc0 $t0, $12
21
22          # Bucle infinito que imprime la cadena "Esto es un prueba"
23  bucle:  la   $a0, cadena
24          li   $v0, 4
25          syscall
26          addi $t7, $0, 30000
27          sll  $t7, $t7, 3        # $t7 <- 30000*8
28  retardo: addi $t7, $t7, -1
29          bnez $t7, retardo
30          beqz $0,  bucle
```

Para comprobar el funcionamiento de la nueva rutina de excepciones y del programa `codigo-teclado.s` se deberá cargar el simulador con la opción de entrada/salida mapeada en memoria y sin la rutina por defecto de tratamiento de excepciones. Tal y como se ha explicado en la sección 10.3.

Una vez el simulador esté en ejecución se deberá cargar, en primer lugar, la rutina `exceptions-3.s` y, a continuación, el fichero correspondiente al programa `codigo-teclado.s`. Una vez cargados ambos códigos, se puede proceder a su ejecución (comenzando en la dirección de memoria `0x0040 0000`).

EJERCICIOS

▶ **257** Explica la función que tiene cada instrucción (o grupo de instrucciones) del programa `codigo-teclado.s`.

▶ **258** Describe cómo interactúan los programas `codigo-teclado.s` y `exceptions-3.s`.

▶ **259** Modifica la rutina de tratamiento de la interrupción de teclado para que cuando se produzca una interrupción de teclado, el código ASCII de la tecla pulsada se escriba en la pantalla.

EJERCICIOS DE AMPLIACIÓN

▶ **260** Modifica la rutina de tratamiento de la interrupción del teclado, implementada en el ejercicio 259, para que cuando se pulse una determinada tecla, por ejemplo la letra «t», se termine la ejecución del programa.

▶ **261** Modifica la rutina de tratamiento de la interrupción del teclado, implementada en el ejercicio 259 para que cuando se pulse la tecla «e» se enmascaren las interrupciones de teclado. Por tanto, en cuanto se pulse la tecla «e» se deberán ignorar las peticiones de interrupción generadas por el teclado. Indica cuáles son las dos posibles formas de enmascarar la interrupción de teclado.

10.5.3. Tratamiento de interrupciones simultáneas

En los apartados anteriores se ha visto como tratar las interrupciones generadas por el reloj programable, el teclado y la pantalla. Pero en cada uno de los casos sólo se ha visto cómo tratar cada una de dichas interrupciones por separado. No se ha visto cómo gestionar el que varios dispositivos generen de forma simultánea una interrupción.

De hecho, en un computador es bastante habitual que varios dispositivos soliciten de forma simultánea la atención del procesador. Por tanto, la rutina de tratamiento de excepciones debe diseñarse para poder gestionar el que varias interrupciones se produzcan de forma simultánea.

El objetivo de este apartado es el de mostrar cómo se pueden gestionar las interrupciones producidas por dos dispositivos de forma simultánea. En concreto, se va a estudiar la gestión de interrupciones generadas simultáneamente por el reloj y el teclado y, posteriormente, la gestión de interrupciones generadas simultáneamente por el teclado y la pantalla.

Reloj programable y teclado

Cuando se produce una excepción, sea cual sea, ésta es tratada por la rutina de tratamiento de excepciones. Por tanto, es en dicha rutina donde debe decidirse qué es lo que se debe hacer cuando se han generado dos o más interrupciones simultáneas.

Se va a estudiar dicha problemática suponiendo que se pretende diseñar un sistema que disponga de un reloj programable y un teclado que se quieren gestionar mediante interrupciones.

El software de dicho sistema deberá tener en cuenta los siguientes aspectos:

1. La habilitación de las interrupciones pertinentes para que dichos dispositivos se puedan gestionar por interrupciones.

2. La identificación de la causa de la interrupción: reloj programable o teclado.

3. El procesamiento adecuado de cada una de dichas interrupciones, la de reloj y la de teclado.

A continuación se describe con más detalle cada uno de los anteriores aspectos.

Habilitación de las interrupciones. Consiste en la habilitación de las interrupciones, tanto en los dispositivos como en el procesador.

- Habilitar las interrupciones en el teclado.
- Programar el reloj para que genere interrupciones periódicas.
- Habilitar las interrupciones en el procesador: poner a 1 los bits correspondientes a los niveles de prioridad 1 y 5 en la *máscara de interrupciones* y poner a 1 el bit de *habilitación de interrupción*.

Identificación de la causa de la interrupción (reloj o teclado). Consiste en incluir en la rutina de tratamiento de excepciones el código necesario para determinar la causa de la interrupción para entonces saltar a la parte de código correspondiente.

Cuando se produce una interrupción, ya sea de reloj o de teclado, en el campo *código de excepción* del registro `Cause` se carga un 0. En ese mismo registro se pone a 1 el bit correspondiente del campo *interrupción pendiente*: 15 ó 11, según sea la interrupción pendiente del reloj o del teclado.

Procesamiento de las interrupciones. Consiste en codificar las operaciones que se deben llevar a cabo para el tratamiento de cada una de dichas interrupciones.

- Desarrollo del código que deberá tratar la interrupción de reloj y retornar al proceso interrumpido.
- Desarrollo del código que deberá tratar la interrupción del teclado y retornar al proceso interrumpido.

EJERCICIOS

▶ **262** Desarrolla un programa en ensamblador que realice las siguientes tareas:

- Inicialice el sistema para que los dispositivos de reloj y teclado se gestionen mediante interrupciones.
- Programe el reloj para que genere interrupciones cada 20 segundos.
- Ejecute un bucle infinito que imprima algún mensaje.

▶ **263** Modifica el programa `exceptions.s` para que se traten las interrupciones generadas por el reloj y el teclado, de acuerdo con las siguientes indicaciones:

- La parte de la rutina que trate la interrupción de reloj deberá imprimir por pantalla el mensaje `«INT DE RELOJ»`.

- La parte de la rutina que trate la interrupción del teclado deberá imprimir por pantalla el mensaje «INT DE TECLADO» y almacenar el carácter introducido en un buffer. Cuando se pulse la tecla «t», deberá imprimir los caracteres que se habían almacenado en dicho buffer. La sincronización de la operación de salida debe realizarse por consulta de estado.

▶ 264 Carga los dos programas anteriores en el simulador y comprueba su funcionamiento.

▶ 265 ¿En qué orden se servirán las interrupciones en caso de producirse simultáneamente la de reloj y la de teclado?

Teclado y pantalla

Se pretende diseñar un sistema que disponga de un teclado y una pantalla que se quieren gestionar mediante interrupciones.

El software de dicho sistema deberá tener en cuenta los siguientes aspectos:

1. La habilitación de las interrupciones pertinentes para que dichos dispositivos se puedan gestionar por interrupciones.
2. La identificación de la causa de la interrupción: teclado o pantalla.
3. El procesamiento adecuado de cada una de dichas interrupciones, la de teclado y la de pantalla.

A continuación se describe con más detalle cada uno de los anteriores aspectos.

Habilitación de las interrupciones. Consiste en la habilitación de las interrupciones, tanto en los dispositivos como en el procesador.

- Habilitar las interrupciones en el teclado.
- Habilitar las interrupciones en la pantalla.
- Habilitar las interrupciones en el procesador: poner a 1 los bits correspondientes a los niveles de prioridad 0 y 1 en la *máscara de interrupciones* y poner a 1 el bit de *habilitación de interrupción*.

Identificación de la causa de la interrupción (teclado o pantalla). Consiste en incluir en la rutina de tratamiento de excepciones el código necesario para determinar la causa de la interrupción para entonces saltar a la parte de código correspondiente.

Cuando se produce una interrupción, ya sea de teclado o de pantalla, en el campo *código de excepción* del registro `Cause` se carga un 0. En ese mismo registro se pone a 1 el bit correspondiente del campo *interrupción pendiente*: 11 ó 10, según sea la interrupción pendiente del teclado o de la pantalla.

Procesamiento de las interrupciones. Consiste en codificar las operaciones que se deben llevar a cabo para el tratamiento de cada una de dichas interrupciones.

- Desarrollo del código que deberá tratar la interrupción del teclado y retornar al proceso interrumpido.
- Desarrollo del código que deberá tratar la interrupción de la pantalla y retornar al proceso interrumpido.

EJERCICIOS

▶ **266** Un sistema basado en el procesador MIPS32 tiene conectados un teclado y una pantalla. El funcionamiento de las operaciones de entrada y salida realizadas a través de estos dispositivos serán gestionadas mediante interrupciones de la siguiente forma:

- Gestión de la entrada. El dato introducido por el teclado se almacenará en un buffer y se imprimirá en la pantalla. Mientras se esté realizando la impresión por pantalla se enmascararan las interrupciones de teclado.
- Gestión de la salida. Cada vez que finalice una operación de salida se imprimirá en la pantalla el siguiente mensaje: `«INT CONSOLA\n»`, se habilitarán las interrupciones de teclado y se comprobará si se han almacenado 10 caracteres en el buffer. Si es así, se imprimirá el contenido de dicho buffer, utilizando la función 4 del monitor, y se terminará el programa en ejecución.

El programa principal, una vez habilitadas las interrupciones pertinentes, simplemente deberá entrar en un bucle sin fin en el que no hará nada (es decir, no imprimirá nada por pantalla).

Para realizar este ejercicio, deberás seguir los siguientes pasos:

1. Modifica el programa `exceptions.s` para que el procesador sea capaz de gestionar y procesar las interrupciones del teclado y la pantalla.
2. Diseña un programa que inicialice los dispositivos, teclado y pantalla, para que las operaciones de entrada y salida realizadas a través de ellos se realicen mediante interrupciones según el funcionamiento descrito.

10.6. Problemas del capítulo

EJERCICIOS

▶ **267** Desarrolla un programa que gestione la entrada y salida de datos mediante interrupciones. La gestión se deberá realizar de la siguiente forma. Los códigos de las teclas pulsadas se irán almacenando en un buffer hasta que se pulse la tecla «w». A partir de dicho momento, se enmascarará la interrupción de teclado y se imprimirán, uno a uno, los códigos de las teclas pulsadas. La sincronización del envío de datos a la pantalla se deberá realizar mediante interrupciones. Cuando se haya enviado el último carácter y éste haya sido visualizado, se volverá a habilitar la interrupción del teclado y se repetirá todo el proceso de nuevo.

▶ **268** Desarrolla un programa que permita la ejecución alternada de dos códigos. El cambio de ejecución de un código a otro se realizará por medio de la pulsación de una tecla. Cuando se retome la ejecución del código previamente interrumpido, éste deberá continuar a partir del punto donde se realizó el último cambio. El cambio de ejecución entre un código y el otro será realizado por la rutina de tratamiento de la interrupción de teclado. Para poder observar claramente que dicho programa funciona como se ha indicado, pruébalo con los dos siguientes códigos: un código que imprima la secuencia de números del 0 al 9 y otro que imprima la secuencia del 9 al 0.

▶ **269** Desarrolla de nuevo un programa que permita la ejecución alternada de dos códigos. En esta nueva versión, el cambio de ejecución de un código a otro se realizará por medio de la interrupción de reloj. El cambio entre un código y el otro deberá realizarse cada minuto. Cuando se retome la ejecución del código previamente interrumpido, éste deberá continuar a partir del punto donde se realizó el último cambio. El cambio de ejecución entre un código y el otro será realizado por la rutina de tratamiento de la interrupción de reloj. Para poder observar claramente que dicho programa funciona como se ha indicado, pruébalo con los dos siguientes códigos: un código que imprima la secuencia de números del 0 al 9 y otro que imprima la secuencia del 9 al 0.

▶ **270** Desarrolla un programa que implemente el funcionamiento descrito a continuación. En todos los casos, la gestión de la entrada y salida deberá realizarse mediante interrupciones. El programa deberá comenzar enmascarando la interrupción de reloj. Cada vez que se pulse una tecla, se deberá almacenar el código correspondiente a dicha tecla en un buffer. Hasta que se pulse la tecla «v». A partir de dicho momento, se comenzará el vaciado del buffer mostrando en pantalla los caracteres que se habían almacenado en él. Coincidiendo con el comienzo del vaciado del buffer, se deberá enmascarar la interrupción del

teclado durante 20 segundos. Tiempo durante el cual no se deberán aceptar nuevos datos. Transcurrido dicho tiempo, se dará por vaciado del buffer, se enmascarará la interrupción de la pantalla, se enmascarará la interrupción de reloj y se habilitará la del teclado, comenzando de nuevo todo el proceso.

▶ **271** Desarrolla un programa que implemente el siguiente funcionamiento utilizando interrupciones para la gestión de todas las operaciones de entrada y salida. Durante 20 segundos, el sistema desarrollado debe permitir la entrada de datos a través del teclado. Los datos introducidos durante dicho intervalo se irán almacenando en un buffer. Transcurridos los 20 segundos, se imprimirán los datos del buffer por la pantalla. Por último, una vez vaciado el buffer se volverá a repetir todo el proceso.

▶ **272** Desarrolla un programa que simule el funcionamiento de un semáforo controlado por un pulsador. El funcionamiento de dicho semáforo deberá ser el siguiente.

El semáforo estará inicialmente en verde (se mostrará el texto «`Semáforo en verde, esperando pulsador`»).

Cuando se pulse la tecla «s», el semáforo deberá cambiar después de 20 segundos, a ámbar (al pulsar la tecla «s», se mostrará el texto «`Pulsador activado: en 20 segundos, el semáforo cambiará a ámbar`»).

Transcurridos los 20 segundos desde que se pulsó la tecla «s», el semáforo cambiará a ámbar (en pantalla se mostrará el texto «`Semáforo en ámbar, en 10 segundos, semáforo en rojo`»).

Transcurridos los 10 segundos desde que se puso en ámbar, el semáforo cambiará a rojo (en pantalla se mostrará el texto «`Semáforo en rojo, en 30 segundos, semáforo en verde`).

Por último, transcurridos 30 segundos con el semáforo en rojo, éste volverá a ponerse en verde, comenzando de nuevo todo el proceso.

BIBLIOGRAFÍA

[Lar] J. R. Larus. *SPIM S20: A MIPS32 Simulator*.

[Pat00] D. A. Patterson y J. L. Hennessy (2000). *Estructura y diseño de computadores: interficie circuitería/programación*. Editorial Reverté.

APÉNDICE A

MANUAL DE USO DEL COMANDO XSPIM

A continuación, se reproduce la salida del comando `man xspim` correspondiente a la versión 7.0 de dicho simulador.

```
spim(1)                                                          spim(1)


NAME
       xspim - A MIPS32 Simulator


SYNTAX
       xspim   [-asm/-bare       -exception/-noexception       -quiet/-noquiet
            -mapped_io/-nomapped_io
       -delayed_branches      -delayed_loads
       -stext  size       -sdata  size       -sstack  size       -sktext  size
            -skdata size      -ldata size      -lstack size      -lkdata size
            -hexgpr/-nohexgpr      -hexfpr/-nohexfpr]
       -file file -execute file


DESCRIPTION
       SPIM  S20 is a simulator that runs programs for the MIPS32 RISC comput-
       ers. SPIM can read and immediately execute  files  containing  assembly
       language or MIPS executable files.  SPIM is a self-contained system for
       running these programs and contains a debugger and interface to  a  few
       operating system services.

       SPIM comes in two versions.  The plain version is called spim.  It runs
       on any type of terminal.  It operates like most programs of this  type:
       you  type  a  line  of text, hit the return key, and spim executes your
       command.  The fancier version of SPIM is called xspim.  It uses the  X-
       window system, so you must have a bit-mapped display to run it.  xspim,
       however, is a much easier program to learn and use because its commands
       are  always  visible  on the screen and because it continually displays
```

the machine's registers.

OPTIONS

xspim has many options:

-asm Simulate the virtual MIPS machine provided by the assembler. This is the default.

-bare Simulate a bare MIPS machine without pseudo-instructions or the additional addressing modes provided by the assembler. Implies -quiet.

-exception
Load the standard exception handler and startup code. This is the default.

-noexception
Do not load the standard exception handler and startup code. This exception handler handles exceptions. When an exception occurs, SPIM jumps to location 0x80000080, which must contain code to service the exception. In addition, this file contains startup code that invokes the routine main. Without the startup routine, SPIM begins execution at the instruction labeled __start.

-quiet Print a message when an exception occurs. This is the default.

-noquiet Do not print a message at exceptions.

-mapped_io
Enable the memory-mapped IO facility. Programs that use SPIM syscalls to read from the terminal cannot also use memory-mapped IO.

-nomapped_io
Disable the memory-mapped IO facility.

-delayed_branches
Simulate MIPS's delayed control transfers by executing the instruction after a branch, jump, or call before transferring control. SPIM's default is to simulate non-delayed transfers, unless the -bare flag is set.

-delayed_loads
Simulate MIPS's original, non-interlocked load instructions. SPIM's default is to simulate non-delayed loads, unless the -bare flag is set.

-stext size -sdata size -sstack size -sktext size -skdata size
Sets the initial size of memory segment seg to be size bytes. The memory segments are named: text, data, stack, ktext, and kdata. The text segment contains instructions from a program. The data segment holds the program's data. The stack segment holds its runtime stack. In addition to running a

program, SPIM also executes system code that handles interrupts and exceptions. This code resides in a separate part of the address space called the kernel. The ktext segment holds this code's instructions and kdata holds its data. There is no kstack segment since the system code uses the same stack as the program. For example, the pair of arguments -sdata 2000000 starts the user data segment at 2,000,000 bytes.

-ldata size -lstack size -lkdata size
Sets the limit on how large memory segment seg can grow to be size bytes. The memory segments that can grow are data, stack, and kdata.

-hexgpr Disply the general purpose registers (GPRs) in hexadecimal.

-nohexgpr Disply the general purpose registers (GPRs) in decimal.

-hexfpr Disply the floating-point registers (FPRs) in hexadecimal.

-nohexfpr Disply the floating-point registers (FPRs) as floating-point values

-file file 10
Load and execute the assembly code in the file.

-execute file 10
Load and execute the MIPS executable (a.out) file. Only works on systems using a MIPS processors.

BUGS

Instruction opcodes cannot be used as labels.

SEE ALSO

spim(1)
James R. Larus, ''SPIM S20: A MIPS R2000 Simulator,'' included with SPIM distribution.

AUTHOR

James R. Larus, Computer Sciences Department, University of Wisconsin-Madison. Current address: James R Larus (larus@microsoft.com), Microsoft Research.

spim(1)

APÉNDICE B

LLAMADAS AL SISTEMA OPERATIVO

El simulador SPIM proporciona, por medio de la instrucción « **syscall** » (llamada al sistema), un pequeño conjunto de servicios similares a los que proporciona habitualmente un sistema operativo.

En este apéndice se tratan con detalle tan sólo algunas de las llamadas al sistema proporcionadas por SPIM: la que sirve para indicar la finalización del programa y algunas de las que proporcionan la entrada por teclado y salida por pantalla del simulador.

B.1. Introducción

La Tabla B.1 muestra todos los servicios proporcionados por el sistema operativo del simulador SPIM [Lar]. En dicha tabla se muestran los servicios disponibles, el código que se debe utilizar para indicar qué servicio se quiere utilizar, los argumentos de entrada, y en el caso de que el servicio en cuestión devuelva un resultado, cuál es el registro utilizado para ello.

Para solicitar un servicio, un programa debe cargar el *código de llamada al sistema* en el registro `$v0` y sus argumentos en los registros `$a0...$a3` (o `$f12` para los valores en coma flotante, según se muestra en la Tabla B.1). Aquellas llamadas al sistema que devuelven resultados proporcionan los resultados en el registro `$v0` (o `$f0` si el resultado es un número en coma flotante).

En este apéndice se muestran únicamente ejemplos de las llamadas a los siguientes servicios:

- `exit`: finaliza el programa en curso.

- `print_int`: imprime un entero en la pantalla.
- `print_string`: imprime una cadena de caracteres en la pantalla.
- `read_int`: lee un entero del teclado.
- `read_string`: lee una cadena de caracteres del teclado.

Servicio	Código de llamada al sistema	Argumentos	Resultado
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = cadena de caracteres	
read_int	5		integer (en `$v0`)
read_float	6		float (en `$f0`)
read_double	7		double (en `$f0`)
read_string	8	`$a0` = buffer, `$a1` = longitud	
sbrk	9	`$a0` = cantidad	dirección (en `$v0`)
exit	10		
print_character	11	`$a0` = carácter	
read_character	12		carácter (en `$v0`)
open	13	`$a0` = nombre de fichero, `$a1` = flags, `$a2` = modo	descriptor de fichero (en `$v0`)
read	14	`$a0` = descriptor de fichero, `$a1` = buffer, `$a2` = cuenta	bytes leídos (en `$v0`)
write	15	`$a0` = descriptor de fichero, `$a1` = buffer, `$a2` = cuenta	bytes escritos (en `$v0`)
close	16	`$a0` = descriptor de fichero	0 (en `$v0`)
exit2	17	`$a0` = valor	

Tabla B.1: Llamadas al sistema

B.2. Finalización del programa en curso: `exit`

Para finalizar un programa de forma correcta, es decir, informando al sistema operativo de que el programa ha concluido, se debe llamar al servicio `exit` (código `10`). Este servicio no requiere de ningún parámetro de entrada por lo que lo único que habrá que hacer antes de la instrucción « **syscall** » será cargar en el registro `$v0` el valor `10`. El siguiente programa de ejemplo suma dos números y luego llama al servicio `exit`.

syscall-exit.s

```
1           .text                   # Zona de programa
2   main:   li $t0, 10
3           li $t1, 20
4           add $t2, $t0, $t1
5
6           li $v0, 10              # Finalizar programa
7           syscall
```

Si no se utiliza al final de un programa la llamada al sistema `exit` y éste se ejecuta en el simulador, da la impresión de que el programa termina correctamente. Sin embargo, cuando no se utiliza la llamada al sistema, el procesador intenta decodificar la palabra que se encuentra a continuación de la última instrucción del programa; lo que generalmente provoca un error por intento de ejecución de algo que no es una instrucción. Por ejemplo, si en el programa anterior no se hubiera utilizado la llamada al sistema `exit`, al ejecutar el programa, éste hubiera terminado pero con el siguiente error (que se puede ver en el panel de mensajes):

```
Attempt to execute non-instruction at 0x00400030
```

B.3. Impresión de un entero: `print_int`

Para imprimir un entero en pantalla se debe utilizar el servicio `print_int` (código `1`). Este servicio requiere que se indique en el registro de entrada `$a0` el entero que se quiere imprimir en pantalla.

De esta forma, un programa que quiera imprimir en pantalla, por ejemplo, la secuencia de enteros `34`, `28` y `-1` debería llamar para cada uno de los números al servicio `print_int` (código `1`), pasando en el registro `$a0` el valor del entero que se quiere imprimir. Un programa que hace ésto es:

syscall-print-int.s

```
1           .data                   # Zona de datos
2   vector: .word 34 , 28 , -1
3
4           .text                   # Zona de instrucciones
5   main :  li $v0, 1               # Llamada al sistema para print_int
6           lw $a0, vector($0)
7           syscall                 # Muestra el primer número en pantalla
8           lw $a0, vector+4($0)
9           syscall                 # Muestra el segundo número en pantalla
10          lw $a0,vector+8($0)
11          syscall                 # Muestra el tercer numero en pantalla
12
13          li $v0, 10              # Finalizar programa
14          syscall
```

En el programa anterior, los números aparecen uno a continuación de otro. Si se quisiera introducir algún tipo de separador entre los números, habría que

utilizar el servicio `print_string` que se explica en la siguiente sección, entre las llamadas al servicio `print_int`.

B.4. Impresión de una cadena: `print_string`

Para imprimir una cadena en pantalla se debe utilizar el servicio llamado `print_string` (código `4`). Este servicio requiere que se indique en el registro de entrada `$a0` la dirección de comienzo de la cadena que se quiere imprimir en pantalla. La cadena debe terminar con el valor `0`.

De esta forma, un programa que quiera imprimir en pantalla «¡Hola mundo!» debería llamar al servicio `print_string` (código `4`), pasando en el registro `$a0` la dirección de comienzo de la cadena «¡Hola mundo!». Un programa que hace ésto es:

✎ syscall-hola-mundo.s

```
1           .data                   # Zona de datos
2  cadena:  .asciiz "¡Hola mundo!"
3
4           .text                   # Zona de instrucciones
5  main:    li $v0, 4               # Llamada al sistema para print_str
6           la $a0, cadena          # Dirección de comienzo de la cadena
7           syscall                 # Muestra la cadena en pantalla
8
9           li $v0, 10              # Finalizar programa
10          syscall
```

B.5. Lectura de un entero: `read_int`

Para leer un entero del teclado se debe utilizar el servicio `read_int` (código `5`). Este servicio no requiere de ningún parámetro de entrada y devuelve el resultado en el registro `$v0`.

De esta forma, un programa que quiera leer un entero del teclado, debería llamar al servicio `read_int` (código `5`), a la vuelta de dicha llamada el registro `$v0` tendrá el entero leído. El siguiente programa muestra un mensaje solicitando la introducción de un entero, lee un entero del teclado y luego imprime el número leído. Para ello, además de la llamada al servicio `read_int` utiliza llamadas a los servicios `print_int` y `print_string` explicadas en las secciones anteriores.

✎ syscall-read-int.s

```
1           .data                   # Zona de datos
2  mensaje1: .asciiz "Introduce un número entero: "
3  mensaje2: .asciiz "El número que has introducido es el: "
4  cr:       .asciiz "\n"  # retorno de carro
5
6           .text                   # Zona de instrucciones
```

```
7  main:    li   $v0, 4            # print_string mensaje1
8           la   $a0, mensaje1
9           syscall
10
11          li   $v0, 5            # Llamada al sistema para read_int
12          syscall                # Lee un número entero del teclado
13
14          ori $t0, $v0, 0        # $t0 <- $v0
15
16          li   $v0, 4            # print_string mensaje2
17          la   $a0, mensaje2
18          syscall
19          li   $v0, 1            # print_int
20          ori $a0, $t0, 0
21          syscall
22          li   $v0, 4            # print_string retorno de carro
23          la   $a0, cr
24          syscall
25
26          li   $v0, 10           # Finalizar programa
27          syscall
```

B.6. Lectura de una cadena: `read_string`

Para leer una cadena del teclado se debe utilizar el servicio `read_string` (código `8`). Este servicio requiere como parámetro de entrada la dirección inicial donde se debe almacenar la cadena leída y el tamaño disponible para dicha cadena.

De esta forma, un programa que quiera leer una cadena de caracteres del teclado, debería llamar al servicio `read_string` (código `8`), y proporcionar en el registro `$a0` la dirección de comienzo de la zona de memoria en la que se debe almacenar la cadena y en el registro `$a1` el tamaño en bytes de dicha zona de memoria. El siguiente programa muestra un mensaje solicitando la introducción de una cadena de caracteres, lee la cadena y luego la imprime de nuevo. Para ello, además de la llamada al servicio `read_string` utiliza la llamada al servicio `print_string` explicada en la sección B.4.

✎ syscall-read-string.s

```
1           .data                  # Zona de datos
2  mensaje1: .asciiz "¿Cómo te llamas?\n"
3  mensaje2: .asciiz "Hola "
4  buffer:   .space 100
5  size:     .word  100
6
7           .text                  # Zona de instrucciones
8  main:    li   $v0, 4            # print_string mensaje1
9           la   $a0, mensaje1
10          syscall
11
12          li   $v0, 8            # Llamada al sistema para read_string
13          la   $a0, buffer       # Dirección del buffer para la cadena
14          lw   $a1, size($0)     # Tamaño del buffer
15          syscall                # Lee una cadena de caracteres del teclado
16
```

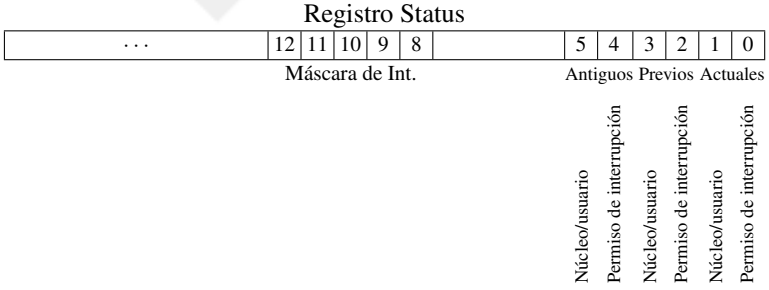
```
17          li   $v0, 4             # print_string mensaje2
18          la   $a0, mensaje2
19          syscall
20          li   $v0, 4             # print_string buffer
21          la   $a0, buffer
22          syscall
23
24          li   $v0, 10            # Finalizar programa
25          syscall
```

APÉNDICE C

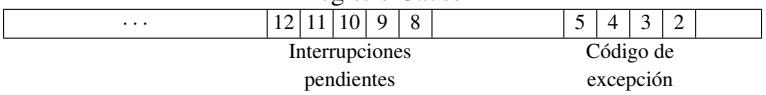
REGISTROS DEL COPROCESADOR 0

Nombre del registro	Número de registro	Función
BadVaddr	8	Contiene la dirección de memoria que ha provocado la excepción.
Status	12	Máscara de interrupciones y bits de autorización.
Cause	13	Tipo de excepción y bits de interrupciones pendientes.
EPC	14	Contiene la dirección de memoria de la última instrucción ejecutada por el procesador.

Registro Status



Registro Cause



Excepciones del MIPS

Núm.	Nombre	Descripción
0	INT	Interrupción externa
4	ADDRL	Excepción por dirección errónea (carga o lectura de instrucción)
5	ADDRS	Excepción por dirección errónea (almacenamiento)
6	IBUS	Error de bus en lectura de instrucción
7	DBUS	Error de bus en carga o almacenamiento de datos
8	SYSCALL	Excepción por llamada al sistema (*syscall*)
9	BKPT	Excepción por punto de ruptura (*breakpoint*)
10	RI	Excepción por instrucción reservada
12	OVF	Excepción por desbordamiento aritmético

APÉNDICE D

CUADRO RESUMEN DEL JUEGO DE INSTRUCCIONES DEL MIPS

CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000

CARGA

lw rt,dirección *Carga palabra* **I**

Carga los 32 bits almacenados en la palabra de memoria especificada por dirección en el registro rt.

```
lw $s0, 12($a0)   # $s0 ← Mem[ 12 + $a0 ]
```

lb rt, dirección *Carga byte y extiende signo* **I**

Carga los 8 bits almacenados en el byte de memoria especificado por dirección en el LSB del registro rt y extiende el signo

```
lb $s0, 12($a0)   # $s0(7..0) ← Mem[ 12 + $a0 ](1byte)
                  # $s0(31..8) ← $s0(7)
```

lbu rt, dirección *Carga byte y no extiende signo* **I**

Carga los 8 bits almacenados en el byte de memoria especificado por dirección en el LSB del registro rt sin extender el signo

```
lbu $s0, 12($a0)  # $s0 ← 0x000000(Mem[ 12 + $a0 ])(1byte)
```

lh rt, dirección *Carga media palabra y ext. signo* **I**

Carga media palabra (16 bits) almacenada en la media palabra de memoria especificada por la dirección en la parte baja del registro rt y extiende el signo

```
lh $s0, 12($a0)   # $s0 (15..0)← Mem[ 12 + $a0 ](2bytes)
                  # $s0 (31..16)← $s0(15)
```

lhu rt, dirección *Carga media palabra y no ext. signo* **I**

Carga media palabra (16 bits) almacenada en la media palabra de memoria especificada por la dirección en la parte baja del registro rt y no extiende el signo

```
lhu $s0, 12($a0)  # $s0← 0x0000Mem[ 12 + $a0 ](2bytes)
```

la reg, dirección *Carga dirección* **PS**

Carga la dirección calculada en reg

```
la $s0, VAR       # $s0← dir. asociada a etiqueta VAR
```

lui rt, dato *Carga inmediata superior* **I**

Carga el dato inmediato en los 16 MSB del registro rt

```
lui $s0, 12       # $s0(31..16) ← 12
                  # $s0(15..0) ← 0x0000
```

li reg, dato *Carga inmediato* **PS**

Carga el dato inmediato en el registro reg.

```
li $s0, 12        # $s0 ← 12
```

ARITMÉTICAS

add rd, rs, rt *Suma* **R**

Suma el contenido de los registros rs y rt, considerando el signo. El resultado se almacena en el registro rd

```
add $t0, $a0, $a1        # $t0 ←$a0 + $a1
```

addu rd, rs, rt *Suma sin signo* **R**

Suma el contenido de los registros rs y rt, sin considerar el signo. El resultado se almacena en el registro rd

```
addu $t0, $a0, $a1       # $t0 ←$a0 + $a1
```

sub rd, rs, rt *Resta* **R**

Resta el contenido de los registro rs y rt considerando el signo. El resultado se almacena en el registro rd

```
sub $t0, $a0, $a1        # $t0 ←$a0 - $a1
```

subu rd, rs, rt *Resta sin signo* **R**

Resta el contenido de los registros rs y rt, sin considerar el signo. El resultado se almacena en el registro r.

```
subu $t0, $a0, $a1       # $t0 ←$a0 - $a1
```

addi rt, rs, valor *Suma inmediata* **I**

Suma el contenido del registro rs con el valor inmediato, considerando el signo. El resultado se almacena en el registro rt.

```
addi $t0, $a0, -24       # $t0 ←$a0 + (-24)
```

addiu rt, rs, valor *Suma inmediata sin signo* **I**

Suma el contenido del registro rs con el valor inmediato, sin considerar el signo. El resultado se almacena en el registro rt.

```
addiu $t0, $a0, 24              # $t0 ←$a0 + 24
```

mult rs, rt *Multiplicación* **R**

Multiplica el contenido de los registros rs y rt. Los 32 MSB del resultado se almacenan en el registro HI y los 32 LSB en el registro LO

```
mult $s0, $s1    # $HI ←($s0 * $s1) (31…16)
                 # $LO←($s0 * $s1) (15…0)
```

div rs, rt *División* **R**

Divide el registro rs por el rt. El cociente se almacena en LO y el resto en HI.

```
div $s0, $s1     # $LO ← $s0 / $s1
                 # $HI ← $s0 % $s1
```

COMPARACIONES

slt rd, rs, rt *Activa si menor* **R**

Pone el registro rd a 1 si rs es menor que rt y a 0 en caso contrario

```
slt $t0, $a0, $a1        # if ( $a0 < $a1) $t0 ← 1
                         # else $t0 ← 0
```

slti rt, rs, inm *Activa si menor con inmediato* **I**

Pone el registro rt a 1 si rs es menor que el dato inmediato inm y a 0 en caso contrario

```
slti $t0, $a0, -15       # if ( $a0 < -15) $t0 ← 1
                         # else $t0 ← 0
```

seq rdest, rsrc1, rsrc2 *Activa si igual* **PS**

Pone el registro rdest a 1 si rsrc1 es igual que rsrc2 y a 0 en caso contrario

```
seq $t0, $a0, $a2        # if ( $a0 == $a2) $t0 ← 1
                         # else $t0 ← 0
```

sge rdest, rsrc1, rsrc2 *Activa si mayor o igual* **PS**

Pone el registro rdest a 1 si rsrc1 es mayor o igual que rsrc2 y a 0 en caso contrario

```
sge $t0, $a0, $a2        # if ( $a0 >= $a2) $t0 ← 1
                         # else $t0 ← 0
```

sgt rdest, rsrc1, rsrc2 *Activa si mayor* **PS**

Pone el registro rdest a 1 si rsrc1 es mayor que rsrc2 y a 0 en caso contrario

```
sgt $t0, $a0, $a2        # if ( $a0 > $a2) $t0 ← 1
                         # else $t0 ← 0
```

sle rdest, rsrc1, rsrc2 *Activa si menor o igual* **PS**

Pone el registro rdest a 1 si rsrc1 es menor o igual que rsrc2 y a 0 en caso contrario

```
sle $t0, $a0, $a2        # if ( $a0 <= $a2) $t0 ← 1
                         # else $t0 ← 0
```

sne rdest, rsrc1, rsrc2 *Activa si no igual* **PS**

Pone el registro rdest a 1 si rsrc1 es diferente de rsrc2 y a 0 en caso contrario

```
sne $t0, $a0, $a2        # if ( $a0 != $a2) $t0 ← 1
                         # else $t0 ← 0
```

CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000

ALMACENAMIENTO

sw rt, dirección *Almacena palabra* **I**

Almacena el contenido del registro rt en la palabra de memoria indicada por dirección

```
sw $s0, 12($a0)   # Mem[ 12 + $a0 ] ←$s0
```

sb rt, dirección *Almacena byte* **I**

Almacena el LSB del registro en el byte de memoria indicado por dirección

```
sb $s0, 12($a0)   # Mem[ 12 + $a0 ] ←$s0(7..0)
```

sh rt, dirección *Almacena media palabra* **I**

Almacena en los 16 bits de menos peso del registro en la media palabra de memoria indicada por dirección.

```
sh $s0, 12($a0)   # Mem[ 12 + $a0 ] ←$s0(15..0)
```

LÓGICAS

and rd, rs, rt *AND entre registros* **R**

Operación AND bit a bit entre los registros rs y rt. El resultado se almacena en rd

```
and $t0, $a0, $a1          # $t0 ←$a0 & $a1
```

andi rt, rs, inm *AND con inmediato* **I**

Operación AND bit a bit entre el dato inmediato, extendiento ceros, y el registro rs. El resultado se almacena en rt.

```
andi $t0, $a0, 0xA1FF      # $t0 ←$a0 & (0x0000A1FF)
```

or rd, rs, rt *OR entre registros* **R**

Operación OR bit a bit entre los registros rs y rt. El resultado se almacena en rd

```
or $t0, $a0, $a1           # $t0 ←$a0 | $a1
```

ori rt, rs, inm *OR con inmediato* **I**

Operación OR bit a bit entre el dato inmediato, extendiendo ceros, y el registro rs. El resultado se almacena en rt.

```
ori $t0, $a0, 0xA1FF       # $t0 ←$a0 | (0x0000A1FF)
```

MOVIMIENTO ENTRE REGISTROS

mfhi rd *mueve desde HI* **R**

Transfiere el contenido del registro HI al registro rd.

```
mfhi $t0          # $t0 ← HI
```

mflo rd *mueve desde LO* **R**

Transfiere el contenido del registro LO al registro rd.

```
mflo $t1          # $t1 ← LO
```

FORMATO DE LAS INSTRUCCIONES

tipo **R**

inst(6 bits)	rs(5bits)	rt (5bits)	rd(5bits)	shamt(5bits)	co (6bits)

tipo **I**

inst(6bits)	rs(5bits)	rt(5bits)	inm(16 bits)

tipo **J**

inst(6bits)	objetivo (26 bits)

DESPLAZAMIENTO

sll rd, rt, shamt *Desplamiento logico a la izquierda* **R**

Desplaza el registro rt a la izquierda tantos bits como indica shamt

```
sll $t0, $t1, 16     # $t0 ← $t1 << 16
```

srl rd, rt, shamt *Desplazamiento lógico a la derecha* **R**

Desplaza el registro rt a la derecha tantos bits como indica shamt.

```
srl $s0,$t1, 4       # $s0 ← $t1 >> 4
```

sra rd, rt, shamt *Desplaz. aritmético a la derecha* **R**

Desplaza el registro rt a la derecha tantos bits como indica shamt. Los bits MSB toman el mismo valor que el bit de signo de rt. El resultado se almacena en rd

```
sra $s0,$t1, 4       # $s0 ← $t1 >> 4
                     # $s0(31..28) ← $t1(31)
```

SALTOS INCONDICIONALES

j dirección *Salto incondicional* **J**

Salta a la instrucción apuntada por la etiqueta dirección

```
j finbucle           # $pc ← dirección etiqueta finbucle
```

jal dirección *Saltar y enlazar* **J**

Salta a la instrucción apuntada por la etiqueta dirección y almacena la dirección de la instrucción siguiente en $ra

```
jal rutina           # $pc ← dirección etiqueta rutina
                     # $ra ← dirección siguiente instrucción
```

jr rs *Saltar a registro* **R**

Salta a la instrucción apuntada por el contenido del registro rs.

```
jr $ra     # $pc ← $ra
```

SALTOS CONDICIONALES

beq rs, rt, etiqueta *Salto si igual* **I**

Salta a etiqueta si rs es igual a rt

```
beq $t0, $t1, DIR          # if ( $t0=$t1) $pc ← DIR
```

bgez rs, etiqueta *Salto si mayor o igual que cero* **I**

Salta a etiqueta si rs es mayor o igual que 0

```
bgez $t0, SLT              # if ( $t0>=0) $pc ← SLT
```

bgtz rs, etiqueta *Salto si mayor que cero* **I**

Salta a etiqueta si rs es mayor que 0

```
bgtz $t0, SLT              # if ( $t0>0) $pc ← SLT
```

blez rs, etiqueta *Salto si menor o igual que cero* **I**

Salta a etiqueta si rs es menor o igual que 0

```
blez $t1, ETQ              # if ( $t1<=0) $pc ← ETQ
```

bltz rs, etiqueta *Salto si menor que cero* **I**

Salta a etiqueta si rs es menor que 0

```
bltz $t1, ETQ              # if ( $t1<0) $pc ← ETQ
```

bne rs, rt, etiqueta *Salto si distinto* **I**

Salta a etiqueta si rs es diferente de rt

```
bne $t0, $t1, DIR          # if ( $t0<>$t1) $pc ← DIR
```

bge reg1, reg2, etiq *Salto mayor o igual* **PS**

Salta a etiq si reg1 es mayor o igual que reg2

```
bge $t0, $t1, DIR          # if ( $t0>=$t1) $pc ← DIR
```

bgt reg1, reg2, etiq *Salto mayor* **PS**

Salta a etiq si reg1 es mayor que reg2

```
bgt $t0, $t1, DIR          # if ( $t0>$t1) $pc ← DIR
```

ble reg1, reg2, etiq *Salto menor o igual* **PS**

Salta a etiq si reg1 es menor o igual que reg2

```
ble $t0, $t1, DIR          # if ( $t0<=$t1) $pc ← DIR
```

blt reg1, reg2, etiq *Salto menor* **PS**

Salta a etiq si reg1 es menor que reg2

```
blt $t0, $t1, DIR          # if ( $t0<$t1) $pc ← DIR
```