Prácticas de Introducción a la Arquitectura de Computadores

con el MIPS R2000

Sergio Barrachina **Maribel Castillo** **José M. Claver**
Departamento de Ingeniería y Ciencia de los Computadores
Universitat Jaume I

Índice general

Prefacio

Este libro de prácticas está dirigido principalmente 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 profundizar en sus conocimientos de Arquitectura de Computadores por medio de la realización de prácticas en lenguaje ensamblador.

Pese a que el hilo conductor de este libro de prácticas es la programación en lenguaje ensamblador, el lector no debe confundirlo con un curso de programación en ensamblador. Nuestro objetivo al escribir este libro no ha sido, ni de lejos, el de enseñar las técnicas de programación en ensamblador, sino el de ayudar a que el lector asimile y comprenda, por medio de una programación básica en ensamblador, algunos conceptos relacionados con la Arquitectura de Computadores. En particular, son objeto de este libro los siguientes conceptos: la relación entre la programación en lenguajes de alto nivel y el funcionamiento del procesador, la programación de computadores a bajo nivel, el juego de instrucciones, los registros, la organización de la memoria, la representación de la información, los tipos de datos y el tratamiento de interrupciones y excepciones.

Hemos intentando que el presente libro sea autocontenido. Así, conforme se requieran nuevos conceptos para la realización de ejercicios, éstos serán introducidos. De todas formas, 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.

Para la escritura de este libro, y para la orientación de las prácticas con ensamblador, ha sido necesario decantarse por un determinado procesador. Cabría pues, preguntarse cuál puede ser el procesador ideal para este cometido. Si observamos qué procesadores han sido escogidos, a lo largo de los años, por las asignaturas de introducción a la Arquitectura de Computadores de distintas Universidades, veremos que han sido varios los elegidos; y que la elección de uno u otro ha estado fuertemente condicionado por las modas. De todas formas, 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 com-

putadores. Cabe destacar entre estos procesadores a los siguientes: el PDP 11 de DEC, el 8088/8086 de Intel, el MC68000 de Motorola y el R2000/30000 de MIPS. De éstos, 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, sin duda el motivo determinante para su adopción ha sido la amplia difusión que han tenido los computadores personales de IBM (PCs) y compatibles.

Sin embargo, en la actualidad, el procesador más extendido en el ámbito de la enseñanza de la Arquitectura de Computadores, y el que se utiliza en este libro, es el MIPS R2000. Los principales motivos por los que ha sido escogido son que mantiene la simplicidad de los primeros procesadores RISC y que su arquitectura es la semilla de muchos de los diseños de procesadores superescalares actuales. Debido justamente a la simplicidad de su juego de instrucciones, al lector le será relativamente fácil desarrollar pequeños programas en ensamblador y observar el efecto de su ejecución. El hecho de que su arquitectura es la semilla de muchos de los diseños de procesadores superescalares actuales facilitará al lector la extensión de los conocimientos adquiridos en su estudio a arquitecturas más avanzadas.

Así pues, las prácticas propuestas con el MIPS R2000 debieran permitir al lector profundizar de forma gradual en el conocimiento de los siguientes temas: el juego de instrucciones del MIPS R2000; los modos de direccionamiento que éste soporta; los mecanismos disponibles para transferir información entre la memoria y los registros; cómo se implementan en ensamblador las llamadas a funciones y el paso de parámetros, tan habituales en los lenguajes de alto nivel; cómo se accede y transfiere información a y desde los dispositivos de entrada/salida; y, finalmente, cómo se gestionan las interrupciones y excepciones.

La metodología que hemos seguido para la realización de los capítulos es la siguiente. Cada capítulo introduce una serie de conceptos nuevos, que generalmente son ilustrados mediante un pequeño fragmento de código. A continuación, se proponen una serie de ejercicios, generalmente relacionados con el código presentado, que el lector debe resolver. Estos ejercicios suelen ser sencillos y buscan que el estudiante observe con detalle qué es lo que está ocurriendo en el computador o que repase los conocimientos teóricos relacionados. Por último, al finalizar cada capítulo se proponen una serie de problemas más elaborados. Es importante que el lector resuelva estos problemas finales ya que es justamente en ellos donde deberá esforzarse más y donde pondrá en práctica los distintos conceptos tratados en el capítulo.

Por otro lado, y puesto que el libro está orientado a estudiantes de primeros cursos,

hemos tenido especial cuidado en reducir al máximo la complejidad algorítmica de los ejercicios propuestos, así como el nivel de conocimientos necesarios de otros lenguajes de programación de alto nivel. Sólo en contadas ocasiones hemos recurrido a programas escritos en lenguaje C y en estos casos hemos limitado al máximo su complejidad.

Para la realización de los ejercicios propuestos sugerimos la utilización de un simulador del MIPS R2000: el simulador SPIM de James Larus. Este simulador puede obtenerse de forma gratuita y está disponible en versiones para Windows y GNU/Linux. Por lo tanto, será fácil para el lector realizar, si lo desea, los ejercicios propuestos por su cuenta en su propio computador.

La organización del resto del libro es la siguiente. [POR DESARROLLAR :-( ]

Convenios tipográficos

En esta sección presentamos los convenios tipográficos que hemos elegido para la redacción de este libro.

Empezaremos con la notación empleada para diferenciar los valores numéricos del texto que los rodean, éstos utilizan un formato propio. Un valor numérico puede estar expresado en alguna de las siguientes bases: 10, 16 ó 2. Cuando expresemos un valor numérico en base 10, éste aparecerá con un formato similar al siguiente: `1024`. Por otro lado, un valor numérico expresado en hexadecimal utilizará el mismo formato anterior pero precedido de «`0x`», p.e. `0x4A`. Además, cuando el valor numérico corresponda a una palabra de 32 bits, aparecerá así: `0x0040 0024` (con un pequeño espacio en medio para facilitar su lectura). De forma similar, un valor numérico expresado en binario, aparecerá con el formato 0010_2 (con el subíndice 2). Y cuando se trate de una palabra de 32 bits, aparecerá como:

$$00000001\,00100011\,01000101\,01100111_2$$

(con un pequeño espacio cada 8 bits para facilitar su lectura).

Mostraremos los códigos o fragmentos de código, recuadrados tal y como aparece en el siguiente ejemplo (no te preocupes si no entiendes nada de lo que aparece a continuación, conforme avances en el libro podrás descubrir su significado):

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
```

Como puedes observar en el ejemplo precedente, en la parte superior derecha aparece dibujada una pestaña en la que se muestra el nombre del fichero que se ha escogido para el código. Además, las líneas del código aparecen numeradas en el margen izquierdo. Esta numeración tiene por objeto facilitar la referencia a puntos concretos del programa y, naturalmente, no debe reproducirse en caso de copiar el código. También puedes observar en el ejemplo anterior que los espacios en blanco en las cadenas de texto, están representados por el carácter «␣». Como es lógico, si copias el programa, deberás utilizar espacios en su lugar.

Los bloques de ejercicios también tienen su propio formato. Éstos se representan delimitados entre dos líneas horizontales punteadas. Cada uno de los ejercicios posee un número único: de esta forma, se puede utilizar dicho número para la identificación[1] del ejercicio en cuestión. A modo de ejemplo:

EJERCICIOS

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

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

Además, cuando entre el texto aparezca una parte de un programa o una sentencia en ensamblador, ésta aparecerá con el siguiente formato: «`la $a0, texto`». Por último, cuando en medio de un texto se haga referencia a un registro, éstos se diferenciarán utilizando el siguiente formato: `$s0`.

Agradecimientos

Este texto es fruto de la experiencia docente del profesorado de las asignaturas de «Introducción a los 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.

En particular, se ha enriquecido con las aportaciones, comentarios y correcciones de los siguientes profesores del departamento de Ingeniería y Ciencia de los Computadores de la Universitat Jaume I de Castellón: [lista de colaboradores ;-)]. Para todos ellos, nuestro agradecimiento.

Nos gustaría, además, agradecer de antemano la colaboración de cuantos nos hagan llegar sugerencias o las erratas que detecten, ya que esto nos permitirá mejorar este libro en sus futuras ediciones.

[1] Siendo puristas, los ejercicios 1 y 2 aparecen duplicados debido al ejemplo de ejercicios presentado en esta página. Naturalmente, los ejercicios 1 y 2 reales no son éstos.

Capítulo 1

Introducción al simulador SPIM

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

Existen versiones del simulador SPIM tanto para GNU/Linux como para Windows. Aunque en este capítulo vamos a describir únicamente la versión para GNU/Linux, gran parte de la descripción que hagamos del funcionamiento de esta 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.

Comenzaremos viendo una breve descripción de la interfaz gráfica del simulador y de la información que proporciona. Una vez realizada esta descripción, prestaremos atención a las operaciones que, de forma más habitual, realizarás durante el seguimiento de este libro: cargar el código fuente de un programa en el simulador y depurar errores mediante la ejecución fraccionada del programa. Seguiremos con una introducción al ensamblador del MIPS R2000 en la que comentaremos la sintaxis básica de este lenguaje (que se irá ampliando, conforme se necesite, en los siguientes capítulos). Finalmente, presentaremos un sencillo ejemplo que te permitirá practicar y familiarizarte con el funcionamiento del simulador descrito en este capítulo.

Es importante que estudies 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 [**?**] (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 MIPS R2000 y R3000. Puede descargarse

gratuitamente desde la siguiente página web:
http://www.cs.wisc.edu/~larus/spim.html

En dicha página web también puedes 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 utilizas una distribución GNU/Linux RedHat, SuSE o similar, puede que te sea más sencillo buscar en Internet un paquete `RPM` actualizado del simulador e instalarlo (p.e. con «`rpm -i spim.rpm`»); si, en cambio, utilizas Debian o Gentoo, estás de suerte: puedes instalar el simulador simplemente ejecutando «`apt-get install spim`» o «`emerge spim`», respectivamente.

En las siguientes sección describimos 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 (ver 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: la utiliza el simulador para informar de qué esta haciendo y avisar de los errores que ocurran durante el ensamblado o ejecución de un programa.

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

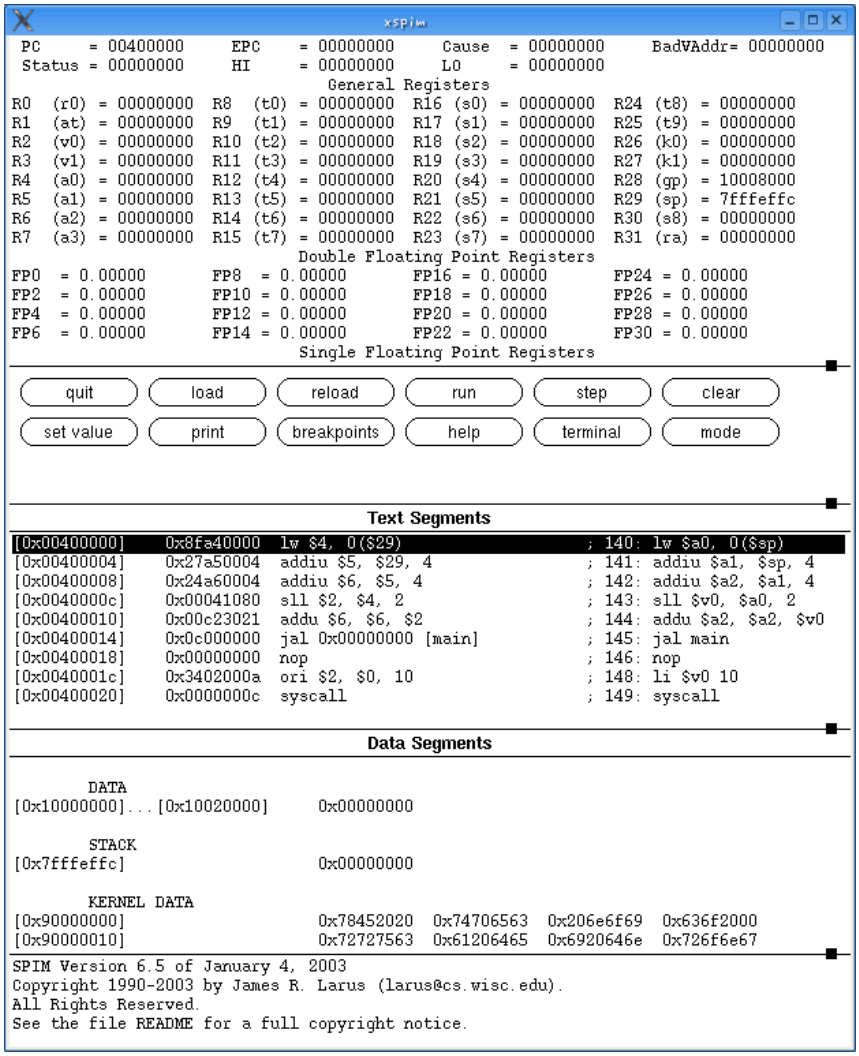

Figura 1.1: Ventana principal del simulador XSPIM.

Panel de visualización de registros

En este panel (ver Figura 1.2) se muestran los diversos registros del R2000. 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 podemos referirnos 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 podemos representar un número real en el R2000 utilizando los formatos IEEE 754 de simple y doble precisión, el simulador nos 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 podemos observar, el contenido de todos los registros, salvo los de coma flotante, se muestra en hexadecimal. Fíjate que como los registros son de 4 bytes (32 bits) se utilizan 8 dígitos hexadecimales para representar su contenido: recuerda que cada dígito hexadecimal corresponde a 4 bits, por tanto, dos dígitos hexadecimales, constituyen un byte.

Panel de botones

En este panel (ver Figura 1.3) se encuentran los botones que permiten controlar el funcionamiento del simulador. Pese a que están representados con forma de botones, su

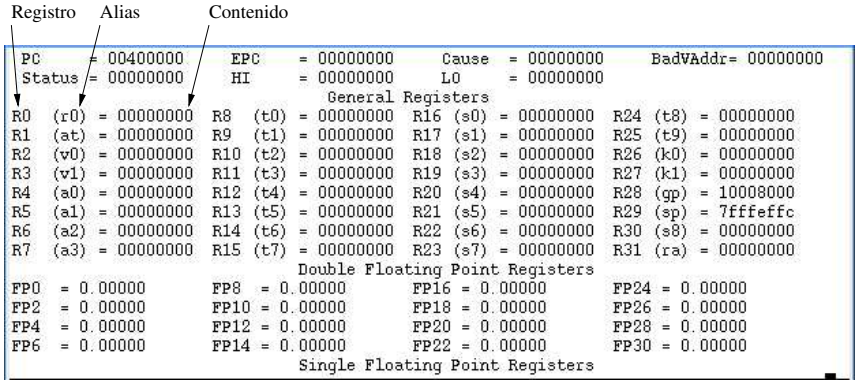

Figura 1.2: Panel de visualización de registros de XSPIM.

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 queremos 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, podemos utilizar este botón para ensamblar de nuevo dicho fichero. De esta forma, si el programa que estás probando no ensambla o no funciona, puedes editarlo y recargarlo 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 pensamos 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 quede de programa de forma continua. Es conveniente, sobre todo al principio, que ejecutes los programas instrucción a instrucción para comprender el

funcionamiento de cada una de ellas y su contribución al programa. Si te limitas a pulsar el botón `Run` tan sólo verás si el programa ha hecho lo que se esperaba de él o no, pero no comprenderás 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 nos permite indicar si queremos 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 carácteres 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).

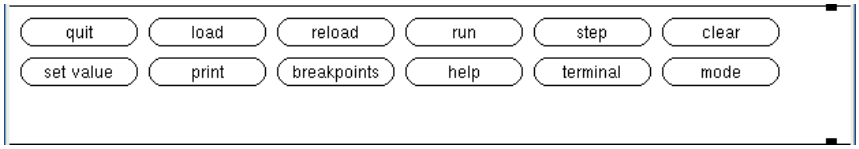

Figura 1.3: Panel de botones de XSPIM.

Panel de visualización de código

Este panel (ver Figura 1.4) muestra el código máquina presente en el simulador. Podremos 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`).

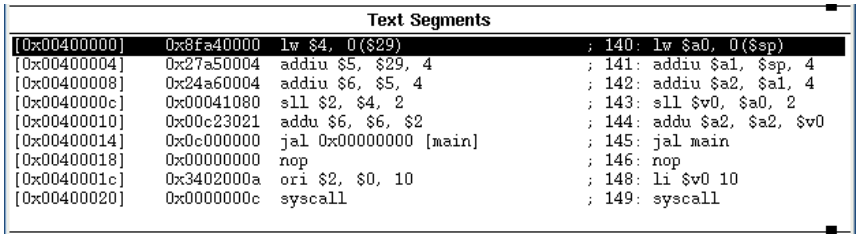

Figura 1.4: Panel de visualización de código de XSPIM.

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 cargues un programa en el simulador, verás 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 (ver Figura 1.5) podemos ver el contenido de las siguientes zonas de memoria:

- Datos de usuario (*DATA*): que 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 `0x7fff effc`.
- Núcleo del simulador (*KERNEL*): a partir de la dirección `0x9000 0000`.

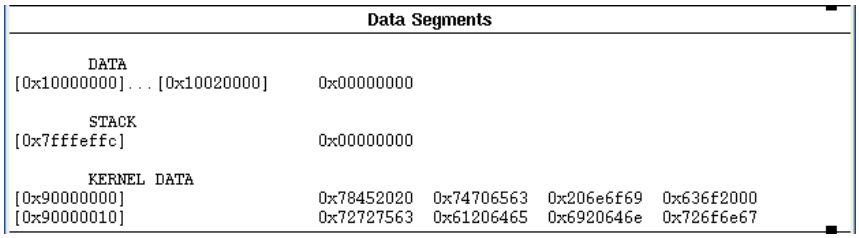

Figura 1.5: Panel de visualización de datos de XSPIM.

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 posiciones `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:

```
[0x10000010]...[0x10020000] 0x00000000 ,
```

que indica que todas las palabras de memoria desde la dirección `0x10000010` hasta la `0x10020000` contienen el valor `0x00000000`.

Panel de visualización de los mensajes del simulador

Este panel (ver 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.

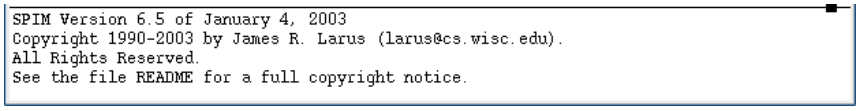

Figura 1.6: Panel de visualización de mensajes de XSPIM.

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.

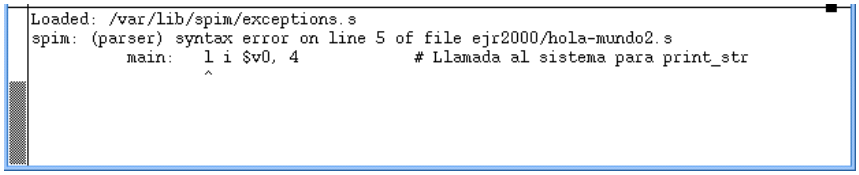

Figura 1.7: Mensaje de error al ensamblar un programa.

En el mensaje anterior se puede ver que el simulador nos 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 `0x00400028` y el error es `Unaligned address in inst/data fetch: 0x10010001`.

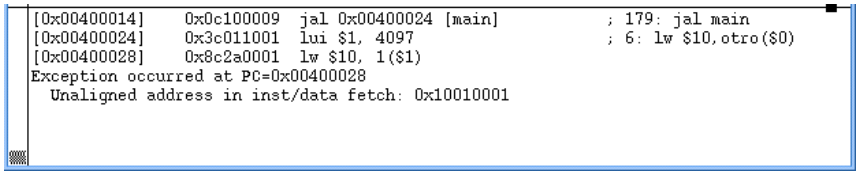

Figura 1.8: Mensaje de error durante la ejecución de un programa.

De momento no te preocupes si no entiendes 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, recuerdes que debes consultar la información mostrada en este panel para averiguar qué es lo que debes corregir.

1.1.2. Opciones de la línea de comandos de XSPIM

El simulador XSPIM acepta una serie de opciones de línea de comandos. De éstas, nos interesarán las siguientes:

-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 `0x80000080`, 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 la utilidad de E/S ubicada en memoria. Deberá utilizarse cuando la gestión de la E/S la realice el usuario a través de los puertos asociados a cada dispositivo.

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, hemos considerado oportuno no detallarlas. De todas formas, puedes 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 debemos pulsar el botón `run`. En el cuadro de diálogo que aparece después de pulsar este botón podemos cambiar la dirección de comienzo de la ejecución (en hexadecimal). Normalmente no deberemos cambiar la que aparece por defecto (`0x0040 0000`).

1.1.4. Depuración de programas

Cuando se desarrolla un programa podemos cometer distintos tipos de errores. El más común de éstos es el cometido al teclear de forma incorrecta 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, tenemos 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 nos 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 puedes 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 nos 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, podremos ejecutar el programa por partes y comprobar si cada una de las partes hace lo que esperamos 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 pulsamos el botón `step` (dentro de este cuadro de diálogo) se ejecutarán tantas instrucciones como hayamos indicado.

La otra herramienta disponible es la de indicar en que posiciones de memoria queremos que se detenga la ejecución de programa. Estas paradas reciben el nombre de puntos de ruptura. Por ejemplo, si creamos un punto de ruptura en la posición de memoria `0x0040 0024` y pulsamos 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 nos permite observar el contenido de la memoria y los registros y podríamos comprobar si realmente es el esperado. Cuando queramos reanudar la ejecución bastará con pulsar de nuevo el botón `run` (o podríamos continuar la ejecución paso a paso utilizando el botón `step`).

Por último, es conveniente que sepas que cada vez que rectifiquemos un fichero fuente tras haber detectado algún error será necesario volver a cargarlo en el simulador. No basta con corregir el fichero fuente, debemos decirle al simulador que vuelva 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 hemos detectado errores realizaremos las siguientes acciones: `clear` → «*memory & registers*» seguido de `reload` → «*assembly file*».

1.2. Sintaxis del lenguaje ensamblador del MIPS R2000

Aunque irás conociendo más detalles sobre la sintaxis del lenguaje ensamblador conforme vayas siguiendo este libro, es conveniente que te familiarices con algunos conceptos básicos.

El código máquina es el lenguaje que entiende el procesador. Un programa en código máquina, como ya sabrás, 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 R2000 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 surgiera rápidamente la necesidad de desarrollar programas capaces de leer instrucciones escritas en un lenguaje más natural, que pudieran ser fácilmente convertidas en instrucciones máquina. Los programas que realizan esta función reciben el nombre de *ensambladores*, y el lenguaje utilizado el de *lenguaje ensamblador*.

El lenguaje ensamblador ofrece por tanto una representación más próxima al programador, aunque no demasiado, y simplifica la lectura y escritura de programas. No proporciona únicamente nemónicos (palabras fáciles de recordar asociadas a cada una de las instrucciones máquina) sino que también ofrece una serie de recursos que tienen por objeto aumentar la legibilidad de los programas. A continuación se muestran algunos de los recursos proporcionados por el ensamblador del R2000 y 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 ensamblador proporciona instrucciones adicionales que no pertenecen al juego de instrucciones del procesador. El ensamblador se encarga de sustituirlas por una o más instrucciones máquina que realicen su función. Permiten una programación más clara. Su sintaxis es similar a la de las instrucciones.

Identificadores Son secuencias de carácteres alfanuméricos, guiones bajos («_») y puntos («.»), que no comienzan con un número. Las instrucciones y pseudo-instrucciones

se consideran palabras reservadas y, por lo 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 carácteres. Los números en base 10 se escriben tal cual. Para expresar un valor en hexadecimal, éste deberá estar precedido por los carácteres «`0x`». Las cadenas de carácteres deben encerrarse entre comillas dobles (”). Es posible utilizar carácteres 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 escribas el código fuente correspondiente a un programa en ensamblador presta 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 pulses 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

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.

Cuando programas en un lenguaje de alto nivel utilizas 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 tu programa.

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

2.1. Declaración de palabras en memoria

En este apartado veremos las directivas **`.data`** y **`.word`**. Como punto de partida veamos el siguiente ejemplo:

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.

Bytes, palabras y medias palabras

Los computadores basados en el procesador R2000 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 carácteres ASCII, no requieren más que un byte por dato. Sin embargo, la capacidad de expresión de un byte es bastante reducida (p.e. si quisiéramos trabajar con números enteros tendríamos que contentarnos 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 R2000, una *palabra* equivale a 4 bytes. Todo el diseño del procesador tiene en cuenta este tamaño de palabra: los registros tienen un tamaño de 4 bytes, el bus de datos 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 R2000 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 como ya habrás supuesto 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.

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 quisiéramos que los datos comenzaran en la posición `0x1001 0020`, deberíamos 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. ¿Observas alguna diferencia en los valores almacenadas en memoria con 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 `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 0x10        # 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 carácteres

La directiva **`.ascii`** "cadena" le indica al ensamblador que debe almacenar el código ASCII de los carácteres que componen la cadena entrecomillada. Estos carácteres se almacenarán 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`**, ¿Puedes ver 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 rango de posiciones se han reservado en memoria para 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 que dirección se ha inicializado byte1? ¿Y byte2?

▶ **23** ¿A partir de que 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 rango de posiciones se ha reservado ahora para 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 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 en memoria a partir de la dirección por defecto: 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 programa ensamblador que inicialice, en el espacio de datos, la cadena de carácteres «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 como 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

El estilo del juego de instrucciones de MIPS 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 podemos 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 veremos 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 veremos 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.

Comenzaremos viendo un programa de ejemplo con la instrucción **`lui`**:

carga-lui.s

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

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

Directiva «`.text` [DIR]»

Como vimos 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`).

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 la instrucción **`lui`** $s0, 0x8690 en la zona de visualización de código 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 quisieramos cargar un dato inmediato de 32 bits en un registro, no podríamos utilizar la instrucción **`lui`**. De hecho, no podríamos utilizar ninguna de las instrucciones del juego de instrucciones del R2000 ya que todas ellas son de 32 bits y no podríamos ocupar toda la instrucción con el dato que deseamos cargar.

La solución consiste en utilizar dos instrucciones: la primera de ellas sería la instrucción **`lui`** en la que especificaríamos los 16 bits de mayor peso del dato de 32 bits; la segunda de las instrucciones sería una instrucción **`ori`** que nos serviría para cargar los 16 bits de menor peso respetando los 16 bits de mayor peso ya cargados. La instrucción **`ori`** la veremos con más detalle en el siguiente capítulo, por el momento, sólo nos interesa saber que podemos utilizarla en conjunción con **`lui`** para cargar un dato inmediato de 32 bits en un registro.

Supongamos que queremos cargar el dato inmediato `0x8690 1234` 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) ¿Que valor contiene `$s0` después de ejecutar **`lui`** $s0, 0x8690?

b) ¿Que 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*). Veamos 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.

Hemos visto hasta ahora 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 deseamos cargar en el registro.

La última pseudo-instrucción que nos 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 podríamos utilizar una constante para especificar la dirección de memoria del dato, es más cómodo utilizar la etiqueta que previamente hayamos 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 las instrucciones **la** $s1, palabra2 y **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 instrucción **la** $s2, 0x10010004?

3.2. Carga de palabras (de memoria a registro)

Para cargar una palabra de memoria a registro utilizamos la instrucción **lw** rt,Inm(rs) (del inglés ***l**oad **w**ord*). 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 crees que el simulador traduce 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 hemos 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. Cuando intentes ejecutarlo verás 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 querríamos 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 queramos cargar un byte sin tener en cuenta su posible signo utilizaremos 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 utilizamos la instrucción **sw** rt,Inm(rs) (del inglés ***s**tore **w**ord*). 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 crees que 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 habías predicho. Ejecuta el programa y comprueba que realmente se han modificado las direcciones de memoria que has indicado previamente y con los valores predichos.

▶ 52 ¿Qué instrucciones máquina se utilizan en lugar de la 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 utilizamos 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 crees que 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 habías predicho. Ejecuta el programa y comprueba que realmente se han modificado las direcciones de memoria que has 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 `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 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 el vector de enteros $V = (10, 20, 25, 500, 3)$ comenzando en la dirección de memoria `0x1000 0000` y que cargue 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 puedes utilizar la directiva «`.data`» tantas veces como lo requieras.)*

▶ **59** Desarrolla un programa ensamblador que dada la 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 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

Hasta ahora hemos visto cómo podemos indicar en ensamblador que queremos inicializar determinadas posiciones de memoria con determinados valores, cómo podemos 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 sabemos 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 hemos visto cuáles son las operaciones que puede realizar el procesador. En éste y en el siguiente capítulo veremos 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 MIPS R2000 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 MIPS R2000 puede realizar comenzaremos 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

Operaciones con signo y «sin signo»

El método de representación utilizado por el R2000 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 R2000 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 R2000 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.

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 $\texttt{11111111 11111111 11111111 11111110}_2$).

EJERCICIOS

Carga el fichero anterior en el simulador SPIM y ejecútalo.

▶ **62** Localiza el resultado de la suma. ¿Cuál ha sido? ¿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?

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,rt,rs» 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,rt,rs» 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?

Hemos visto instrucciones para realizar sumas y restas. A continuación veremos como realizar las operaciones de multiplicación y división. Dichas operaciones utilizan de forma implicita los registros `HI` y `LO` para almacenar el resultado de la operación correspondiente. En los ejercicios que aparecerán a continuación podrás descubrir la utilidad de dichos registros y la información que se almacena en cada uno de ellos en función de la operación realizada.

Veamos como primer ejemplo 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 la instrucción «**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 MIPS R2000 proporciona un conjunto de instrucciones que permite realizar operaciones lógicas del tipo AND, OR, XOR... Debes tener en cuenta que las operaciones lógicas, aunque toman como operandos dos palabras, actúan a nivel de 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.

Veamos 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 que aparece en el programa anterior «**andi** $t1, $t0, 0xfffe» realiza la AND lógica entre el contenido del registro `$t0` y el valor `0x0000 fffe`.

Como ya sabrás, 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 `$t0`, sabemos que independientemente de cual sea el valor almacenado en `$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, `0x0000fffe`, y suponiendo que el contenido de `$t0` fuera `0x01234567`, 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 «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 debes realizar?, ¿basta con una operación en la que uno de los operandos sea un dato inmediato?)*

4.3. Operaciones de desplazamiento

El procesador MIPS R2000 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 utilizada en el programa, «**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 puedes 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 `0xffff ffe8` es realmente el 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 ele-

mentos $V = (10, 20)$ 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 -1.215, 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 puedes utilizar la directiva «* **`.data`** *» tantas veces como lo requieras.)*

▶ **81** Desarrolla un programa que modifique el entero `0xabcd 12bd` 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 ni pseudoinstrucciones de multiplicación.

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.