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

Copyright © 2007–2011 Sergio Barrachina Mir, Maribel Castillo Catalán, José M. Claver Iborra y Juan Carlos Fernández Fernández. Reservados todos los derechos.

Se autoriza la copia de este libro para para su uso en centros públicos de enseñanza o para ser utilizado con fines autodidactas. En el primero de los casos, tan solo se cargarán al estudiante los costes de reproducción. La reproducción total o parcial con ánimo de lucro o con cualquier finalidad comercial queda estrictamente prohibida, salvo que cuente con el permiso escrito de los autores.

Prólogo

Este libro de prácticas está dirigido a estudiantes de primeros cursos del «Grado en Ingeniería Informática» 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 en lenguaje ensamblador.

Pese a que el hilo conductor del libro es la programación en lenguaje ensamblador, el lector no debe confundir este libro con un manual 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 la 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 los basados en la arquitectura 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 (PC) IBM y compatibles.

Sin embargo, la arquitectura más extendida en la actualidad en el ámbito de la enseñanza de arquitectura de computadores, y la que se propone en este libro, es la MIPS32[1]. Dicha arquitectura, por un lado, mantiene la simplicidad de las primeras arquitecturas RISC y, por otro, constituye la semilla de muchos de los diseños de procesadores superescalares actuales.

Debido a la simplicidad del juego de instrucciones de la arquitectura 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 a MIPS32 para utilizarla en la enseñanza de arquitectura de computadores es que se trata de una arquitectura real pero lo suficientemente sencilla como para estudiarla 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 de la arquitectura MIPS32, es la existencia de simuladores que permiten comprobar cómodamente su funcionamiento, sin la necesidad de tener acceso a un computador real basado en dicha arquitectura. En este libro se propone el simulador SPIM de James Larus [Lar]. Este simulador puede obtenerse de forma gratuita desde la página web del autor y está disponible para GNU/Linux, Mac OSX y Windows.

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: la interfaz hardware/software*», de David A. Patterson y John L. Hennesy [Pat11], en el que también se estudia la arquitectura MIPS32.

De esta forma, un curso introductorio a la arquitectura de computadores podría utilizar el anterior libro como referencia de la parte teórica y el presente libro para la realización de las prácticas de laboratorio de dicho curso.

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. $\texttt{0010}_2$. Si el número expresado en binario corresponde a una palabra de 32 bits, este se representa con un pequeño espacio cada 8 bits, para facilitar su lectura, p.e.: $\texttt{00000001 00100011 01000101 01100111}_2$.

Para diferenciar una sentencia en ensamblador del texto que la rodea, se utiliza el siguiente formato: «`la $a0, texto`». Cuando se hace referencia a un registro, este se marca de la siguiente forma: `$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, estos 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, y en la asignatura «Estructura de Computadores» del Grado en Ingeniería Informática 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 Fabregat Llueca, Germán León Navarro, Rafael Mayo Gual, Fernando Ochera Bagan y Ximo Torres Sospedra.

También nos gustaría agradecer desde aquí a los siguientes estudiantes de la Universidad Jaume I, que revisaron 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 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 . . . 37
4.2. Operaciones lógicas . . . 39
4.3. Operaciones de desplazamiento . . . 42
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 61
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 78

8 Gestión de subrutinas 81
8.1. La pila 82
8.2. Bloque de activación de la subrutina 85
8.3. Problemas del capítulo 101

9 Gestión de la entrada/salida mediante consulta de estado 103
9.1. Dispositivos periféricos en el simulador SPIM 106
9.2. Entrada de datos desde el teclado 108
9.3. Salida de datos por pantalla 110
9.4. Entrada/Salida de datos 111
9.5. Problemas del capítulo 112

10 Gestión de la entrada/salida mediante interrupciones 113
10.1. Registros para el tratamiento de las excepciones 115
10.2. Tratamiento de las excepciones 119
10.3. Opciones del programa SPIM para la simulación de interrupciones 125
10.4. Excepciones software 126
10.5. Interrupciones 129
10.6. Problemas del capítulo 146

Bibliografía 149

A Manual de uso del comando xspim 151

B Llamadas al sistema operativo 155
B.1. Introducción 155
B.2. Finalización del programa en curso: `exit` 156
B.3. Impresión de un entero: `print_int` 157
B.4. Impresión de una cadena: `print_string` 158
B.5. Lectura de un entero: `read_int` 158
B.6. Lectura de una cadena: `read_string` 159

C Registros del coprocesador 0 161

D Cuadro resumen del juego de instrucciones MIPS32 163

Capítulo 1

Introducción al simulador SPIM

Índice

1.1. Descripción del simulador SPIM **2**
1.2. Sintaxis del lenguaje ensamblador 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 MIPS32 en la que se comenta la sintaxis básica 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 la arquitectura 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 solo 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 los siguientes apartados 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.

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.

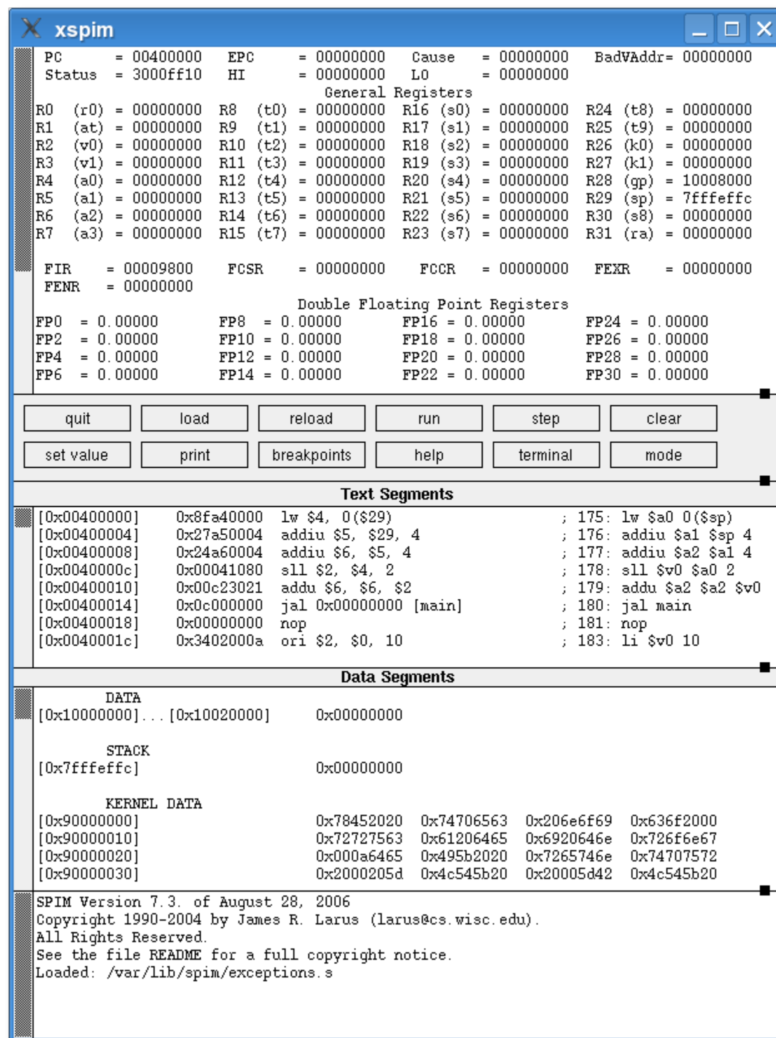


Figura 1.1: Ventana principal del simulador XSPIM

La información presentada en cada uno de estos paneles se describe en los siguientes apartados.

Panel de visualización de registros

En este panel (véase la Figura 1.2) se muestran los diversos registros 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 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`.

Registro Alias Contenido

```
PC     = 00400000    EPC    = 00000000    Cause   = 00000000    BadVAddr= 00000000
Status = 00000000    HI     = 00000000    LO      = 00000000
                                General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
R6  (a2) = 00000000  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
                            Double Floating Point Registers
FP0  = 0.00000       FP8  = 0.00000       FP16 = 0.00000       FP24 = 0.00000
FP2  = 0.00000       FP10 = 0.00000       FP18 = 0.00000       FP26 = 0.00000
FP4  = 0.00000       FP12 = 0.00000       FP20 = 0.00000       FP28 = 0.00000
FP6  = 0.00000       FP14 = 0.00000       FP22 = 0.00000       FP30 = 0.00000
                            Single Floating Point Registers
```

Figura 1.2: Panel de visualización de registros de XSPIM

Como se puede observar, el contenido de todos los registros, salvo los de coma flotante, se muestra en hexadecimal. Puesto que los registros MIPS32 son de 4 bytes (32 bits), 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 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 solo 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 solo 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 basada en la arquitectura MIPS32 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).

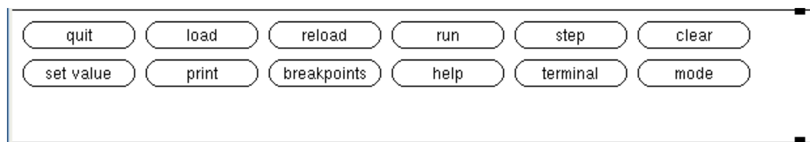


Figura 1.3: Panel de botones de XSPIM

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`).

```
                              Text Segments
[0x00400000]    0x8fa40000  lw $4, 0($29)              ; 140: lw $a0, 0($sp)
[0x00400004]    0x27a50004  addiu $5, $29, 4           ; 141: addiu $a1, $sp, 4
[0x00400008]    0x24a60004  addiu $6, $5, 4            ; 142: addiu $a2, $a1, 4
[0x0040000c]    0x00041080  sll $2, $4, 2              ; 143: sll $v0, $a0, 2
[0x00400010]    0x00c23021  addu $6, $6, $2            ; 144: addu $a2, $a2, $v0
[0x00400014]    0x0c000000  jal 0x00000000 [main]      ; 145: jal main
[0x00400018]    0x00000000  nop                        ; 146: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10             ; 148: li $v0 10
[0x00400020]    0x0000000c  syscall                    ; 149: syscall
```

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 de memoria `0x9000 0000`.

```
                              Data Segments

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

        STACK
[0x7fffeffc]                     0x00000000

        KERNEL DATA
[0x90000000]                     0x78452020  0x74706563  0x206e6f69  0x636f2000
[0x90000010]                     0x72727563  0x61206465  0x6920646e  0x726f6e67
```

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.

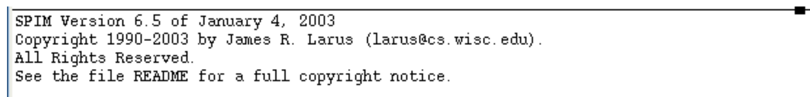


Figura 1.6: Panel de visualización de mensajes de XSPIM

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.

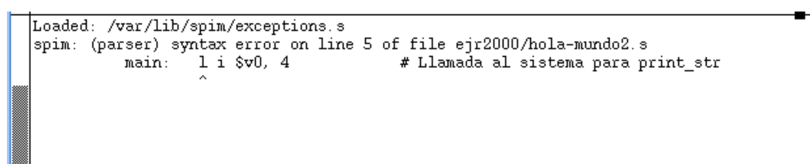


Figura 1.7: Mensaje de error al ensamblar un programa

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 «ˆ» 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`.

```
[0x00400014]    0x0c100009  jal 0x00400024 [main]           ; 179: jal main
[0x00400024]    0x3c011001  lui $1, 4097                    ; 6: lw $10,otro($0)
[0x00400028]    0x8c2a0001  lw $10, 1($1)
Exception occurred at PC=0x00400028
  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 este apartado. 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 Apartado 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 Apartado 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 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 el Apartado 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 el Apartado 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 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 de 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 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

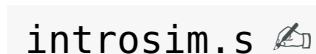
Cuando se escribe el código fuente correspondiente a un programa en ensamblador se debe prestar especial atención a los siguientes puntos:

- 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.



```
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 la arquitectura MIPS32 pueden acceder a la memoria a nivel de byte. Esto implica que cada dirección de memoria debe indicar 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 de la arquitectura 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 computador, la arquitectura MIPS32 obliga a que las palabras en memoria estén alineadas en múltiplos de 4. Así pues, la dirección de memoria de una palabra en memoria siempre será múltiplo de 4. Gracias a esta medida, la transferencia de palabras entre memoria y procesador será más rápida que si las palabras pudieran estar en cualquier posición de memoria.

Por último, además de acceder a bytes y a palabras, también es posible acceder a medias palabras (*half-words*). Una media palabra está formada 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 ....

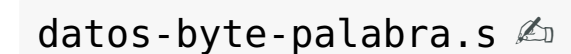
Limpia el contenido de la memoria y carga el siguiente código en el simulador:

```
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:



```
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`»?

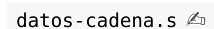
....

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:



```
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?

▶ **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 a partir de cuál «`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?

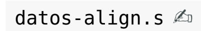
. . . .

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 solo 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 `0x10010000`, 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 juego de instrucciones MIPS32 es del tipo carga/almacenamiento (*load/store*). Esto quiere decir que 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 el primer apartado se verán las instrucciones de carga de datos inmediatos. Los restantes apartados están dedicados a las instrucciones de carga y almacenamiento en memoria.

3.1. Carga de datos inmediatos (constantes)

En este apartado se verá la instrucción «**lui**» que carga un dato inmediato en los 16 bits de mayor peso de un registro y las pseudoinstrucciones «**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 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, solo 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 `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) ¿Qué valor contiene `$s0` después de ejecutar «**lui** `$s0, 0x8690`»?

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

....

Puesto que cargar una constante de 32 bits en un registro es una operación bastante frecuente, el ensamblador MIPS32 proporciona una pseudo-instrucción para ello: la pseudo-instrucción «**li**» (del inglés ***l**oad **i**nmediate*). En el siguiente ejemplo se utiliza dicha pseudo-instrucción:

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 este apartado 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`» (basta con cambiar la instrucción «`lw $s0, palabra($0)`» por «`lw $s0, palabra+1($0)`»). 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 `0x10010000`?

▶ **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 `0xfffffff3` 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 ***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 hará dicho programa? ¿Qué direcciones de memoria se verán afectadas? ¿Qué valores habrá 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 habrá 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`.

. . . .