

Práctica 11. El simulador MARS.

1. Introducción

Las prácticas de ensamblador de la asignatura de Estructura de Computadores consisten en el estudio detallado del procesador MIPS para comprobar los conocimientos impartidos en las clases teóricas.

2. Introducción al simulador MARS

El procesador MIPS se ha utilizado principalmente con fines educativos debido a su simplicidad. Sin embargo, éste no es el único ámbito en el que se emplea, ya que su arquitectura es la base de muchos de los diseños de procesadores utilizados actualmente en diversos dispositivos como, por ejemplo, estaciones de trabajo de Silicon Graphics, routers de Cisco, consolas de Sony, etc.

En el laboratorio, en lugar de trabajar con una máquina MIPS real, se utilizará un simulador denominado MARS escrito en Java. Su funcionamiento es sencillo, e incluye un depurador que, entre otras muchas funciones, permite ejecutar los programas paso a paso y muestra el estado de los registros.

Para ejecutar el programa MARS una vez descargado deberemos abrir un terminal (*shell* del sistema) y ejecutaremos la orden `$java -jar Mars4_5.jar`.

El simulador MARS está disponible en <http://courses.missouristate.edu/KenVollmar/MARS/>

3. Funcionamiento del simulador

Al ejecutar el programa se abrirá una ventana (ver Figura 1) en la que se distinguen cuatro partes:

- *Panel de Menús/Botones:* Muestra las operaciones que se pueden realizar como crear un programa nuevo, ejecutar, ejecutar paso a paso, guardar, etc.
- *Panel de Registros:* Muestra los registros del procesador, tanto los de propósito general (\$zero, \$at, ... \$t0, \$t1, ...), como el **pc**, el registro **hi** y el registro **lo**.
- *Panel de Edición y Ejecución:*
 - La pestaña *Edit* muestra el programa escrito en lenguaje ensamblador del MIPS.
 - La pestaña *Execute* muestra el estado de la memoria. Se distinguen dos partes en la memoria:
 - el segmento de texto, que contiene las instrucciones del programa en lenguaje máquina (0s y 1s)
 - el segmento de datos que contiene los datos del programa (0s y 1s).
- *Panel de Mensajes y Consola:* Muestra los mensajes de información del simulador y la consola o interfaz del usuario con el MIPS (se muestran los mensajes de los programas y se leen datos).



Fig. 1: Ventana principal de MARS

4. Instrucciones aritméticas

Operaciones con registros:

- **Suma (con desbordamiento):** `add $rd, $rs, $rt`
- **Suma (sin desbordamiento):** `addu $rd, $rs, $rt`
- **Resta (con desbordamiento):** `sub $rd, $rs, $rt`
- **Resta (sin desbordamiento):** `subu $rd, $rs, $rt`

Guardan la suma o resta de los registros `$rs` y `$rt` en el registro `$rd`.

Las *operaciones con desbordamiento* provocan una excepción de desbordamiento si el resultado no es representable en 32 bits. El rango de representación de los números enteros con signo de 32 bits en complemento a 2 va desde $-2^{31} = -2.147.483.648$ hasta $2^{31} - 1 = 2.147.483.647$ (de `0x80000000` a `0x7FFFFFFF`). Por tanto, el siguiente código produce un desbordamiento aritmético (nos aparece el mensaje en el panel *Mars Messages*):

```
li $t0, 0x7FFFFFFF
addi $t1, $t0, 1
```

Las *operaciones sin desbordamiento* (las acabadas en `u`) no tienen en cuenta el desbordamiento. Por ejemplo, el siguiente código no produce desbordamiento y `$t0` pasa a valer `0x80000000` (sería el número siguiente a `0x7FFFFFFF` si consideramos números sin signo).

```
li $t0, 0x7FFFFFFF
addiu $t1, $t0, 1
```

- **Multiplicación:** `mult $rs, $rt`
 - **Resultado:** `mflo $rd`

- **Multiplicación (pseudoinstrucción¹):** `mul $destino, $fuente1, $fuente2`

Guarda la multiplicación de los registros `$fuente1` y `$fuente2` en el registro `$destino`.

- **División:** `div $rs, $rt`
 - **Cociente:** `mflo $rd`
 - **Resto:** `mfhi $rd`

Guarda el cociente y el resto de los registros `$rs` y `$rt` en el registro `$rd`.

- **División (pseudoinstrucción):** `div $destino, $fuente1, $fuente2`

Guarda el cociente de los registros `$fuente1` y `$fuente2` en el registro `$destino`.

Operaciones con inmediatos:

- **Suma con inmediato (con desbordamiento):** `addi $rt, $rs, inm`
- **Suma con inmediato (sin desbordamiento):** `addiu $rt, $rs, inm`

Guardan la suma del registro `$rs` y el valor inmediato `inm` en el registro `$rt`.

- **Multiplicación:** `mul $rt, $rs, inm`
- **División:** `div $rt, $rs, inm`

Guardan en `$rt` la multiplicación o cociente de `$rs` con el valor inmediato `inm`.

5. Instrucciones lógicas

- **Desplazamiento lógico a la izquierda:** `sll $rd, $rt, shift`
- **Desplazamiento lógico a la derecha:** `srl $rd, $rt, shift`

Guardan en `$rd` el valor de `$rt` desplazado `inm` bits a la izquierda o a la derecha, respectivamente.

- **Desplazamiento lógico a la izquierda variable:** `sllv $rd, $rt, $rs`
- **Desplazamiento lógico a la derecha variable:** `srlv $rd, $rt, $rs`

Guardan en `$rd` el valor de `$rt` desplazado tantos bits a la izquierda o a la derecha, respectivamente, como indique el registro `$rs`.

- **And lógico:** `and $rd, $rs, $rt`
- **Or lógico:** `or $rd, $rs, $rt`
- **Xor lógico:** `xor $rd, $rs, $rt`
- **Nor lógico:** `nor $rd, $rs, $rt`

Guardan en `$rd` el resultado de operar bit a bit los registros `$rs` y `$rt` con la operación correspondiente.

- **And lógico con inmediato:** `andi $rt, $rs, inm`
- **Or lógico con inmediato:** `ori $rt, $rs, inm`

¹ Una pseudoinstrucción es una instrucción ofrecida por el ensamblador pero que no realiza la circuitería.

- **Xor lógico con inmediato:** `xori $rt, $rs, inm`

Guardan en `$rt` el resultado de operar bit a bit el registro `$rs` y el valor inmediato `inm` con la operación correspondiente.

6. Los registros del procesador MIPS

La CPU del procesador MIPS contiene 32 registros de propósito general, numerados del 0 al 31. Se localizan en el panel de registros. Se puede hacer referencia a cualquiera de los registros mediante su nombre precedido del símbolo del dólar (`$t0`, `$t1`, `$s0`, ...), aunque también es posible hacerlo mediante su número precedido del símbolo del dólar. Así, el registro `$t0` es equivalente al registro `$8`, `$a0` es equivalente a `$4`, etc. (ver Tabla 1).

El registro `$zero` (o también `$0`) siempre contendrá el valor constante 0, puesto que está "cableado" en la CPU, resultando así imposible su modificación.

Con respecto al uso que se debe dar a cada registro, la especificación de la arquitectura MIPS establece un conjunto de normas acerca de la manera más conveniente (y estándar) de utilizar cada uno. Un programa que no respete estas convenciones podría no funcionar correctamente si tiene que interactuar con otros programas. Por ello se recomienda encarecidamente que se utilice cada registro para la finalidad descrita en la siguiente tabla:

Tabla 1: Registros del MIPS.

Núm.	Nombre	Uso
0	<code>zero</code>	Constante con el valor cero
1	<code>at</code>	Reservado para el ensamblador
2	<code>v0</code>	Evaluación de una expresión y resultado de una función
3	<code>v1</code>	Evaluación de una expresión y resultado de una función
4	<code>a0</code>	Argumento 1 de una subrutina
5	<code>a1</code>	Argumento 2 de una subrutina
6	<code>a2</code>	Argumento 3 de una subrutina
7	<code>a3</code>	Argumento 4 de una subrutina
8	<code>t0</code>	Temporal (no mantenido durante la llamada a subrutina)
9	<code>t1</code>	Temporal (no mantenido durante la llamada a subrutina)
10	<code>t2</code>	Temporal (no mantenido durante la llamada a subrutina)
11	<code>t3</code>	Temporal (no mantenido durante la llamada a subrutina)
12	<code>t4</code>	Temporal (no mantenido durante la llamada a subrutina)
13	<code>t5</code>	Temporal (no mantenido durante la llamada a subrutina)
14	<code>t6</code>	Temporal (no mantenido durante la llamada a subrutina)
15	<code>t7</code>	Temporal (no mantenido durante la llamada a subrutina)
16	<code>s0</code>	Temporal salvado (mantenido durante la llamada a subrutina)
17	<code>s1</code>	Temporal salvado (mantenido durante la llamada a subrutina)
18	<code>s2</code>	Temporal salvado (mantenido durante la llamada a subrutina)
19	<code>s3</code>	Temporal salvado (mantenido durante la llamada a subrutina)
20	<code>s4</code>	Temporal salvado (mantenido durante la llamada a subrutina)
21	<code>s5</code>	Temporal salvado (mantenido durante la llamada a subrutina)
22	<code>s6</code>	Temporal salvado (mantenido durante la llamada a subrutina)
23	<code>s7</code>	Temporal salvado (mantenido durante la llamada a subrutina)
24	<code>t8</code>	Temporal (no mantenido durante la llamada a subrutina)
25	<code>t9</code>	Temporal (no mantenido durante la llamada a subrutina)
26	<code>k0</code>	Reservado para el <i>kernel</i> del Sistema Operativo
27	<code>k1</code>	Reservado para el <i>kernel</i> del Sistema Operativo
28	<code>gp</code>	Puntero al área global (<i>global pointer</i>)
29	<code>sp</code>	Puntero de pila (<i>stack pointer</i>)
30	<code>fp</code>	Puntero de bloque de activación (<i>frame pointer</i>)
31	<code>\$ra</code>	Dirección de retorno para llamadas a funciones (<i>return address</i>)

Los registros `$at` (`$1`), `$k0` (`$26`) y `$k1` (`$27`) están reservados para su uso por parte del ensamblador y el sistema operativo y no deben ser empleados por programas de usuario ni compiladores.

Los registros **\$a0** al **\$a3** (\$4 a \$7) se usan para pasar los primeros cuatro parámetros a las subrutinas (el resto se pasan a través de la pila).

Los registros **\$v0** y **\$v1** (\$2 y \$3) se usan para devolver valores calculados en subrutinas. Además, **\$v0** también se utiliza para almacenar un código de llamada al sistema (ver apartado “Llamadas al Sistema Operativo”).

Los registros **\$t0** al **\$t9** (\$8 al \$15, \$24, \$25) se utilizan para mantener datos temporales que no necesitan ser preservados durante las llamadas a subrutinas (se utilizan de forma análoga a las variables locales). Sin embargo, los registros **\$s0** al **\$s7** (\$16 a \$23) contendrán datos de vida más larga que se deben preservar durante las llamadas a subrutinas (algo así como, salvando las distancias, las variables globales).

El registro **\$gp** (\$28) es un puntero global que apunta al centro de un bloque de 64K de memoria en el segmento de datos estáticos. El registro **\$sp** (\$29) apunta a la última posición en uso de la pila. El registro **\$fp** (\$30) es el puntero del bloque de activación (*frame pointer*). La instrucción **jal**, que se utiliza para realizar las llamadas a subrutinas, escribe en el registro **\$ra** (\$31) la dirección de retorno de una llamada.

Por último, hay 32 registros de coma flotante de precisión simple; son los registros **\$f0** al **\$f31**. De éstos, se pueden utilizar los de numeración par (**\$f0**, **\$f2**, **\$f4**, ...) para operaciones de coma flotante de precisión doble.

7. Llamadas al Sistema Operativo

El simulador *Mars* proporciona ciertos servicios a través de la emulación de llamadas al Sistema Operativo con la instrucción **syscall**. Para realizar una llamada, es necesario primero cargar un código de llamada (que indicará si se quiere imprimir un carácter, salir del programa, etc.) en el registro **\$v0**.

Si la llamada necesita argumentos, éstos deberán estar en algunos de los registros **\$a0** al **\$a3**. Las llamadas al sistema que necesiten algún argumento en coma flotante, lo recibirán en el registro **\$f12**.

Algunas llamadas al sistema devuelven un resultado, y lo hacen en el registro **\$v0** (el mismo que se usa para indicar el código de llamada) o bien en **\$f0** si el resultado es de coma flotante. Por ejemplo, la llamada que lee un número introducido por el usuario mediante el teclado almacena este número en el registro **\$v0**. Tras finalizar la llamada al sistema es responsabilidad del programador llevarse este número a otro registro o a la memoria para usos posteriores.

Tabla 2: Llamadas al sistema.

Servicio	Código en \$v0	Argumentos	Resultado
print_int	1	\$a0 = entero	
print_float	2	\$f12 = real de tipo float	
print_double	3	\$f12 = real de tipo double	
print_string	4	\$a0 = dirección de un string	
read_int	5		\$v0 = entero leído
read_float	6		\$f0 = real de tipo float leído
read_double	7		\$f0 = real de tipo double leído
read_string	8	\$a0 = dirección, \$a1 = longitud	
sbrk	9	\$a0 = cantidad	\$v0 = longitud
exit	10		

Estos servicios tienen el siguiente significado:

- **print_int**: se pasa como argumento un entero (en el registro **\$a0**), el cual se imprime en la consola del *Mars* (panel RUN I/O).

- **print_float:** se imprime un número real en la consola del simulador.
- **print_double:** se imprime un número real con precisión doble en la consola del simulador.
- **print_string:** se imprime una cadena de caracteres residente en memoria y terminada en '\0'; para ello habrá que pasar la dirección donde comienza la cadena. Se debe tener en cuenta que el tratamiento de símbolos especiales (intro, tabulación, ...) sigue la notación de C ('\n', '\t', ...). Aún no se puede utilizar en esta práctica porque no se ha indicado cómo acceder al segmento de datos para leer/escribir.
- **read_int, read_float, read_double:** lee de la consola una línea completa e ignora todos los caracteres posteriores al número.
- **read_string:** lee una cadena de $n-1$ caracteres (siendo n el valor de `$a1`), los completa con '\0' y los guarda en la dirección de memoria indicada en `$a0`. Como en el caso de `print_string`, hasta prácticas posteriores no se podrá utilizar
- **sbrk:** devuelve un puntero a un bloque de memoria que contiene n bytes.
- **exit:** finaliza el programa.

El Listado 1 pide por la consola un número y a continuación lo muestra elevado al cuadrado:

Listado 1: Programa que interactúa con el usuario.

```
.data          # comienzo del segmento de datos
cad1: .asciiz "Introduzca un número" #Forma de declarar una cadena
cad2: .asciiz "El resultado es: " #Forma de declarar una cadena

.globl main    # la etiqueta main es global

.text          # comienzo del segmento de texto (instrucciones)

main:          # Comienza la función main

    li $v0, 4   # código de la llamada print_string
    la $a0, cad1 #imprime la cadena que está en la dirección $a0
    syscall

    li $v0, 5   # código de la llamada read_int
    syscall     # ahora se introduce un número en la consola

    move $t0,$v0 # se guarda el contenido de $v0 (el número) en $t1

    mul $t1,$t0,$t0 # se eleva al cuadrado

    li $v0, 4   # código de la llamada print_string
    la $a0, cad2 #imprime la cadena que está en la dirección $a0
    syscall

    li $v0,1    # código de llamada print_int
    move $a0, $t1 # $a0 = número que se quiere imprimir
    syscall     # se realiza la llamada

    li $v0, 10  # código de llamada exit
    syscall     # llamada al sistema
```

En este ejemplo ha aparecido una instrucción nueva, `move`, que será de gran utilidad para transferir el contenido de un registro a otro, operación muy frecuente cuando se realizan llamadas al sistema:

- **Mover contenido de registros:** `move $destino, $fuente`

Copia el contenido de `$fuente` en `$destino` (manteniendo el valor original en `$fuente`).

8. El segmento de datos

El *segmento de datos* se define como la zona de la memoria de la máquina donde se almacenan los *datos* que necesita un programa, en contraposición al *segmento de texto* en donde están las instrucciones.

Las *directivas de ensamblador* (`.data`, `.globl main`, `.text`) no son instrucciones del microprocesador, sino que sirven para indicar ciertos parámetros de la ejecución del programa, definir el comienzo de los segmentos de datos y texto, o declarar datos en el segmento de datos, principalmente. Se distinguen tres tipos de directivas, y de cada tipo las más importantes son las siguientes:

- **Directivas de declaración de segmentos**

`.data <addr>`

Indica el comienzo del segmento de datos, y por tanto todas las directivas que aparezcan a continuación, y hasta que se indique el comienzo de otro segmento, serán directivas de declaración de datos (`.ascii`, `.byte`, `.half`, etc.). El parámetro `<addr>` es opcional e indica la posición de comienzo del segmento de datos. Se utiliza en caso de que no se quiera que el segmento comience en la dirección por defecto `0x10010000`.

`.text <addr>`

Indica el comienzo del segmento de texto. En el segmento de texto se encontrarán únicamente instrucciones del microprocesador. Si se indica una dirección con el parámetro `<addr>`, los datos se almacenan a partir de ésta en lugar de la dirección por defecto `0x00400000`.

- **Directiva de etiqueta global**

`.globl etiqueta`

Sirve para declarar una *etiqueta* como global para que pueda ser referenciada por programas escritos en otros ficheros. Es habitual (no obligatorio) declarar la etiqueta `main` como global, aunque no se llegará a ver el mecanismo de referencia a etiquetas globales desde ficheros externos, puesto que todos los programas que se realizarán van a residir en un único fichero.

- **Directiva de alineación de datos**

`.align n`

Alinea el siguiente dato a una dirección múltiplo de 2^n . Si se mezclan unos tipos de datos con otros hay que prestar atención a aquellos tipos que necesiten alineación. Los datos de tipo `.byte` no necesitan estar alineados. Los datos de tipo `.half` necesitan estar en direcciones múltiplo de 2 (`.align 1`), y los datos de tipo `.word` necesitan estar en direcciones múltiplo de 4 (`.align 2`).

Considerando el siguiente ejemplo:

```
.data 0x10000000
.byte 0x10 # se ocupa el byte (8 bits) 10010000
.word 0xEA # word que comienza en la dirección 0x10010001 (no múltiplo de
4)
# Si se intenta cargar esta palabra en un registro se produce
# un error, pues su dirección no es válida
```

Se podrían mantener estos dos datos en memoria si se alinea el dato de tipo `.word`:

```
.data
.byte 0x10 # se ocupa el byte (8 bits) 10010000
.align 2    # se alinea el siguiente dato a una dirección múltiplo de 4
.word 0xEA # esta palabra ya tiene una dirección válida, ahora está
           # y comienza en la dirección 0x10010004.
```

Estado de la memoria (suponiendo *big endian*):

0x10010000	0x10			
0x10010004	0	0	0	0xEA
0x10010008				
0x1001000c				

- Directivas de declaración de datos (en el segmento de datos)

`.ascii "str"`

Almacena en memoria la cadena "str" (se escribe siempre entre comillas dobles) sin introducir el carácter '\0' al final. En MARS, las cadenas siguen la misma notación que en C, es decir, '\n' significa salto de línea, '\t' es el tabulador, '\0' es el carácter de final de cadena, etc.

`.asciiz "str"`

Funciona igual que la directiva anterior, pero introduce automáticamente un carácter '\0' de fin de cadena al final.

`.space n`

Reserva *n* bytes de memoria consecutivos pero no les asigna ningún valor. Esta directiva es útil cuando se quiere reservar espacio en la memoria donde almacenar, por ejemplo, resultados de operaciones, cadenas de texto introducidas por el usuario, etc.

`.byte b1, b2, ... , bn`

Almacena *n* bytes (palabras de 8 bits) de valores *b1*, *b2*, ... , *bn* en posiciones consecutivas de memoria. Estos bytes pueden estar escritos en notación decimal (10, 3, 532), hexadecimal (0xA, 0x3, 0x214) o como caracteres ('a', 'X', '\t', '6'), Evidentemente, en este caso lo que se almacena en memoria es el valor ASCII del carácter

`.half h1, h2, ..., hn`

Almacena *n* palabras de 16 bits en posiciones consecutivas de memoria. Como en el caso anterior, la notación puede ser decimal, hexadecimal o ASCII. Cuando se introduce un valor representable en menos de 16 bits se rellena con el signo por la izquierda.

`.word w1, w2, ... , wn`

Almacena *n* palabras de 32 bits en posiciones consecutivas de memoria. Como en el caso anterior, la notación puede ser decimal, hexadecimal o ASCII. Cuando se introduce un valor representable en menos de 32 bits se rellena con el signo por la izquierda.

`.float f1, f2, ... , fn`

Almacena *n* números en coma flotante con precisión simple en posiciones consecutivas de memoria.

`.double d1, d2, ... , dn`

Almacena *n* números en coma flotante con precisión doble en posiciones consecutivas de memoria.

Con respecto a las directivas de declaración de datos, caben múltiples posibilidades para llevar a cabo la misma tarea. Naturalmente, es decisión del programador hacer las cosas de uno u otro modo en función de su propio criterio. Así, si se desea almacenar una cadena en el segmento de datos se puede optar por emplear la directiva `.asciiiz`:

```
.asciiiz "La suma es\n"
```

Pero también se puede almacenar cada carácter individualmente como un byte:

```
.byte 'L','a',' ','s','u','m','a',' ','e','s','\n','\0'
```

Incluso se puede almacenar cada carácter como un byte pero utilizando su valor ASCII:

```
.byte 76, 97, 32, 115, 117, 109, 97, 32, 101, 115, 10, 0
```

9. Etiquetas

Las *etiquetas* son identificadores que se sitúan al principio de una línea y van seguidos de dos puntos (por ejemplo, `main:`). Sirven para hacer referencia a la posición o dirección de memoria del elemento definido en ésta, bien sea una instrucción del segmento de texto o un dato del segmento de datos. A lo largo del programa se puede hacer referencia a ellas en las instrucciones de acceso a memoria (para etiquetas definidas en el segmento de datos) o en las instrucciones de salto (para etiquetas definidas en el segmento de texto).

Para esta práctica, una etiqueta va a ser la forma más simple de hacer referencia a un dato del segmento de datos. Así, si se declara una cadena precedida de una etiqueta:

```
.data  
cad1: .asciiiz "Hola, soy una cadena\n"
```

Será muy fácil, por ejemplo, imprimir esta cadena por la consola mediante el uso de la llamada al sistema `print_string`:

```
.text  
main:  
    la $a0, cad1 # se indica la dirección de la cadena  
                  # mediante el nombre de la etiqueta  
    li $v0, 4     # código de llamada print_string  
    syscall
```

La instrucción `la rdest, direccion (load address)` carga en el registro `rdest` la dirección especificada a continuación. **Carga tan sólo la dirección, no su contenido.** Una dirección de memoria responde al concepto ya conocido de puntero. Esta dirección puede venir especificada en múltiples formatos, siendo el más sencillo una etiqueta.

10. Instrucciones de carga y almacenamiento

Para hacer referencia al contenido del segmento de datos, se necesita un conjunto específico de instrucciones de acceso a memoria. El procesador MIPS tiene una arquitectura de tipo *load/store (carga/almacenamiento)*, que significa que sólo ciertas instrucciones específicas, las de *carga* (cogen un dato de la memoria y lo transfieren a un registro) y *almacenamiento* (guardan en memoria un dato contenido en un registro) pueden acceder a la memoria. El resto de las instrucciones del repertorio se limitan a operar con valores guardados en registros o bien con valores inmediatos.

- Instrucciones de carga

- **Load byte:** `lb $rt, Inm($rs)`
Carga en el registro `$rt` el byte (con signo) almacenado en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.
- **Load byte unsigned:** `lbu $rt, Inm($rs)`
Carga en el registro `$rt` el byte (sin signo) almacenado en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.
- **Load half:** `lh $rt, Inm($rs)`
Carga en el registro `$rt` los dos bytes (con signo) almacenados en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.
- **Load half unsigned:** `lhu $rt, Inm($rs)`
Carga en el registro `$rt` los dos bytes (sin signo) almacenados en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.
- **Load word:** `lw $rt, Inm($rs)`
Carga en el registro `$rt` los cuatro bytes (con signo) almacenados en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.

(no hay instrucción de carga de `.word` sin signo)

- Instrucciones de carga (pseudoinstrucciones)

- **Load byte:** `lb rdest, etiqueta`
Carga el byte (con signo) apuntado por `etiqueta` en el registro `rdest`.
- **Load byte unsigned:** `lbu rdest, etiqueta`
Carga el byte (sin signo) apuntado por `etiqueta` en el registro `rdest`.
- **Load half:** `lh rdest, etiqueta`
Carga la palabra de 16 bits (con signo) apuntada por `etiqueta` en el registro `rdest`.
- **Load half unsigned:** `lhu rdest, etiqueta`
Carga la palabra de 16 bits (sin signo) apuntada por `etiqueta` en el registro `rdest`.
- **Load word:** `lw rdest, etiqueta`
Carga la palabra de 32 bits apuntada por `etiqueta` en el registro `rdest`.

(no hay instrucción de carga de `.word` sin signo)

- Instrucciones de almacenamiento

- **Store byte:** `sb $rt, Inm($rs)`
Almacena los 8 bits menos significativos del registro `$rt` en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.
- **Store half:** `sh $rt, Inm($rs)`
Almacena los 16 bits menos significativos del registro `$rt` en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.
- **Store word:** `sw $rt, Inm($rs)`

Almacena el contenido íntegro del registro `$rt` en la dirección de memoria especificada por la suma de `Inm` más el contenido del registro `$rs`.

• Instrucciones de almacenamiento (pseudoinstrucciones)

- **Store byte:** `sb rdest, etiqueta`
Almacena los 8 bits menos significativos del registro `rdest` en la dirección de memoria indicada por `etiqueta`.
- **Store half:** `sh rdest, etiqueta`
Almacena los 16 bits menos significativos del registro `rdest` en la dirección de memoria indicada por `etiqueta`.
- **Store word:** `sw rdest, etiqueta`
Almacena el contenido íntegro del registro `rdest` en la dirección de memoria indicada por `etiqueta`.

11. Modos de direccionamiento

Para hacer referencia a las posiciones de memoria del segmento de datos se pueden utilizar etiquetas, pero este método es incómodo y poco flexible. Por eso, el simulador *MARS* permite utilizar diversas técnicas de referencia a memoria para acceder a los datos del segmento de datos. Estas técnicas se conocen como **modos de direccionamiento**.

La máquina pura sólo ofrece tres modos de direccionamiento: registro, inmediato y desplazamiento. Este último modo de direccionamiento a memoria se especifica por `inm($rs)` y utiliza un registro como puntero (dirección de memoria) al que se le suma un inmediato `inm` para acceder a una posición de memoria. En cambio, la máquina virtual ofrecida por el simulador *MARS* ofrece los siguientes modos de direccionamiento para las instrucciones `load` y `store`:

Tabla 1: Modos de direccionamiento del MARS.

Formato	Cálculo de la dirección
(registro)	Contenido del registro
<code>inm</code>	Inmediato
<code>inm(registro)</code>	Inmediato + Contenido del registro (único modo soportado por la máquina real)
<code>etiqueta</code>	Dirección de la etiqueta
<code>etiqueta ± inm</code>	Dirección de la etiqueta ± Inmediato
<code>etiqueta ± inm(registro)</code>	Dirección de la etiqueta ± (Inmediato + Contenido del registro)

Recuerda que estos formatos serán usados en las instrucciones de carga/almacenamiento conocidas allí donde sea necesario hacer referencia a una *dirección*.

12. Ejemplo


Escribiremos el primer programa en lenguaje ensamblador del MIPS. Para ello le indicaremos que cree un archivo nuevo:

- Mediante el Menú *File* -> *New*.
- Mediante el Botón *New File*.

A continuación, en la pestaña *Edit* escribimos el programa del Listado 1.

Se pide:

1. Abre el simulador **mars**.

2. Guarda el archivo denominado `listado1.asm`.
3. Ensambla el programa `listado1.asm`. Para ello, hay que utilizar la opción del menú **Run->Assemble** o el botón correspondiente .
4. En la pestaña **Execute** veremos cómo se han cargado en memoria las instrucciones de nuestro programa (la primera está en la dirección `0x00400000`). Como las instrucciones ocupan 4 bytes, ocupan las posiciones `0x00400000`, `0x00400004`, etc. Además, en la columna de la derecha (**source**) vemos la instrucción en lenguaje ensamblador y una columna más a la izquierda (**basic**) vemos la instrucción a la que corresponde en el caso de que se trate de una pseudoinstrucción. En la columna **code** nos aparece la instrucción en lenguaje máquina (en hexadecimal).
5. Ejecuta el programa mediante el comando **Run -> Go** o el botón correspondiente.
6. Comprueba que al finalizar la ejecución del programa el valor del registro `$t1` es el cuadrado del número y en `$t0` tienes el número que leíste.
7. Ensambla de nuevo el programa y ejecútalo mediante el comando **step**. ¿Qué hace este comando?

13. Ejercicio 2

Utilizando como guía la descripción de las instrucciones anteriores, se proponen los siguientes ejercicios. Hay que tener siempre presente que con el simulador *Mars* se puede utilizar el modo de ejecución paso a paso y emplear *breakpoints* para analizar detalladamente la ejecución de un programa, permitiendo detectar errores en la programación.

Se recuerda que la estructura general de un programa presenta un aspecto como el siguiente:

Listado 2: Estructura genérica de un programa escrito en MIPS.

```
.data # Zona de memoria de la máquina donde se almacenan los datos

#####
# Todos los datos necesarios en el programa
#####

.globl main # Directiva de etiqueta global
.text      # Inicio del segmento de texto (instrucciones)

main: #etiqueta main
#####
#Zona de memoria de la máquina donde se almacenan las
#instrucciones de nuestro programa.
# Aquí va el código de nuestros ejercicios.
#####

li $v0,10 # Llamada al sistema de tipo "exit"
syscall
```

1. Realice un programa llamado `ejercicio2.asm` que declare dos datos de tipo Word en el segmento de datos con valores 3 y 7. Además tiene que leer cuatro enteros por pantalla y que muestre lo siguiente:
 - a. $A+B$
 - b. $C-D$
 - c. $A+B+C+D$
 - d. Mostrar A al cuadrado
 - e. $A \ll 3$
 - f. Leerá los dos enteros del segmento de datos e imprimirá el resto de dividir 7 entre 3.