Práctica 6
Variables

Objetivos

- Entender la distribución de la memoria en el MIPS.
- Conocer y utilizar las instrucciones que nos permiten leer y escribir en la memoria.
- Conocer cómo definir datos en la memoria.
- Utilizar funciones del sistema que permiten la entrada y salida de cadenas de caracteres.

Materiales

Simulador MARS y un código Fuente de partida

Teoría

1. Introducción

Ha llegado la hora de estudiar un elemento que hemos obviado en las prácticas precedentes, la memoria. Hemos supuesto hasta ahora que el programa a ejecutar está almacenado en la memoria y en cada ciclo de instrucción se lee una instrucción y se ejecuta en el procesador. Ahora nos preguntamos si los datos utilizados por las instrucciones necesariamente tienen que provenir de los registros del procesador o del teclado. ¿Cabe la posibilidad de que la memoria aloje datos? La respuesta es clara, sí hay datos en la memoria. Además, el ensamblador nos permite tanto leer como escribir valores en la memoria. Eso sí, hay que tener presente que el MIPS sólo permite operar con el ALU con valores inmediatos o con datos alojados en registros (cómo hemos visto hasta ahora). Esto significa que antes de operar con un dato de la memoria tenemos que traerlo a un registro del procesador. El MIPS nos proporciona las instrucciones que nos posibilitan hacerlo. Pero vamos por partes, primero describiremos la memoria del MIPS.

2. La memoria

El modelo de la memoria que implementa el MIPS es el de una memoria plana de 32 bits direccionable por bytes con direcciones de 32 bits. Esto quiere decir que para el programador la memoria del procesador MIPS es una memoria continua de bytes agrupados en palabras de 32 bits que empieza en la dirección 0x00000000 y acaba en la dirección 0xFFFFFFFF. El programador podría utilizar 4GBytes de datos. Ahora bien, esto no significa que toda esa memoria está a disposición del programador, parte de la memoria está reservada para utilizarla el sistema operativo (lo que se denomina el

kernel data), otra parte la utiliza el subsistema de entrada/salida, etc. En la figura 1 se muestra un diagrama de cómo están configurados los 4GB de la memoria del MIPS.

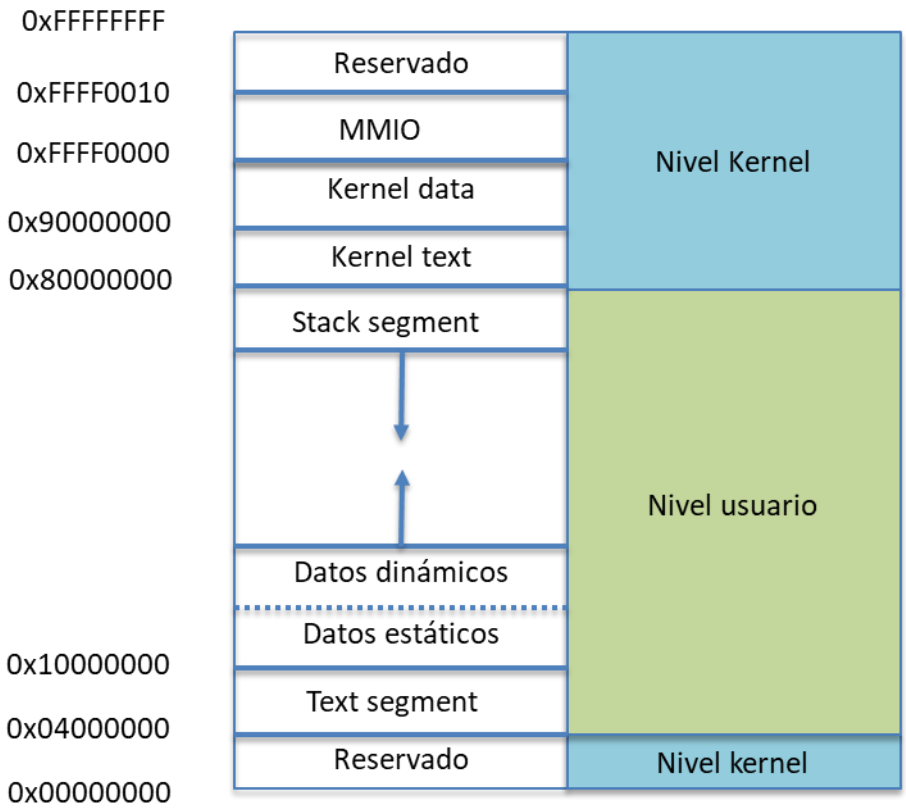

Figura 1. Modelo de la memoria del MIPS

La parte de la memoria accesible para el usuario comprende el **Text segment** (direcciones 0x0040 0000 hasta 0x1000 00000) que es donde se almacena el código del programa. Cómo hemos comentado ya, cada instrucción se almacena en una palabra de 32 bits o 4 bytes. Todo programa escrito hasta ahora empieza con .texto. Se trata de una directiva que define el comienzo del segmento de código. Si no indicamos una dirección, el ensamblador asume por defecto el valor 0x00400000.

Los datos estáticos (direcciones 0x1001 0000 hasta 0x1004 0000) es donde se guardan los datos del programa. El tamaño de los elementos de esta sección se asigna cuando el programa se ensambla y se carga en la memoria. La definición de los elementos no se puede modificar durante la ejecución del programa. Cuando se escribe un programa podemos definir los datos estáticos mediante el uso de directivas que comentaremos en el próximo apartado. De forma similar al **Text segment**, la definición de datos empezará con la directiva `.data`. Si no indicamos una dirección, el ensamblador asume por defecto el valor 0x10010000.

Datos dinámicos (direcciones 0x1004 0000 hasta donde empiece la pila) siempre creciente hacia las direcciones altas. Es donde se almacenan los datos dinámicos que van creándose por necesidad durante la ejecución de los programas (por ejemplo, con un nuevo operador en Java).

El Stack segment (o segmento de pila) (direcciones 0x7ffffe00 hasta que se encuentre con los datos dinámicos) siempre crece hacia las direcciones bajas. Los datos van almacenándose en este segmento conforme se crean mediante las operaciones de push y pop para las subrutinas. Lo estudiaremos en prácticas posteriores.

Desarrollo de la práctica

1. El Text segment

Con cualquier de los programas escritos hasta ahora podemos observar como alberga el *text segment* el programa en ejecución, recordad que todos los programas empiezan con `.text`. Para observarlo simplemente tenéis que seleccionar, una vez ensamblado el programa, la opción de ver el text segment en pantalla (ver Figura 2). Se mostrará en cada fila 8 posiciones de memoria, podéis comprobar que su contenido es el mismo que indica la columna *code* de la ventana superior.

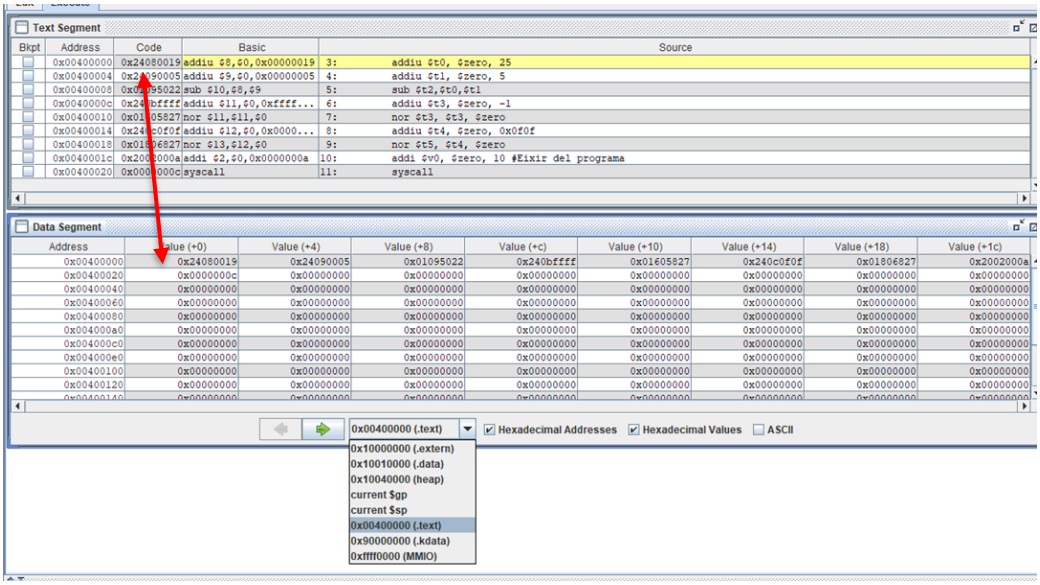

Figura 2. Visualización del text segment de la memoria del MIPS en el MARS.

Cómo veis, para facilitar la visualización de la memoria y mostrar más palabras a la vez en la pantalla, el MARS muestra 8 palabras por fila en lugar de mostrar cada palabra en una fila, tal como se indica en la figura 3:

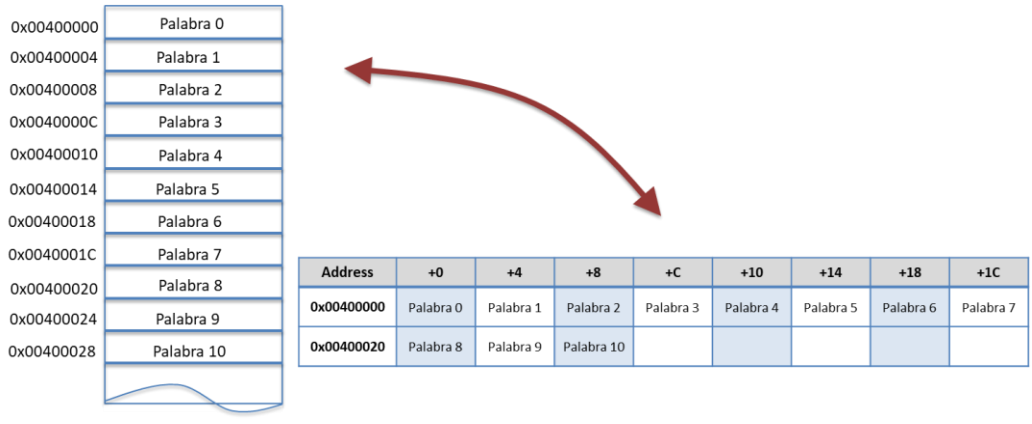

Figura 3. El MARS muestra 8 palabras de la memoria en la misma línea.

El procesador accede al *text segment* de la memoria para leer la instrucción en el registro $pc que indica la dirección de la palabra donde se encuentra la instrucción a ejecutar.

2. Direccionamiento de la memoria

Como se ha comentado, el MIPS utiliza un modelo plano para la memoria, así, el programador ve la memoria como una sucesión de bytes a los que puede acceder mediante una dirección. Al poder acceder a cada byte individual, se dice que la memoria es direccionable por byte. Empezando desde la dirección 0x00000000, estos bytes se organizan en grupos de diferentes tamaños: como bytes individuales, como parejas de bytes (se denominan medias palabras, *half word*), como grupos de 4 bytes (llamadas palabras, *word*) y como grupos de 8 bytes (llamadas dobles palabras, *double word*). A cada uno de estos grupos se puede acceder mediante la dirección del byte menos significativo (denominado *LSB*) del grupo, es decir, el byte con la dirección más baja. Estos grupos de bytes tienen que estar alineados. Esto quiere decir que para seleccionar un byte es suficiente que la dirección sea múltiplo de 1, pero para acceder a una media palabra tiene que ser múltiple de 2, a una palabra tiene que ser múltiple de 4 y a una doble palabra tiene que ser múltiple de 8, etc. Por ejemplo, con la dirección 0x10001000 podemos seleccionar un byte, una media palabra o una palabra entera, pero con la dirección 0x10001001 sólo podemos seleccionar un byte. La forma de distinguir a que grupo se está referenciando depende del operador utilizado para acceder. En la figura se muestra esta idea:

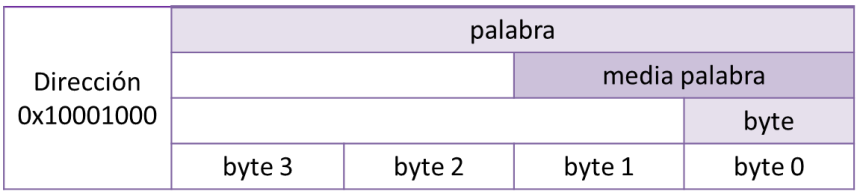

Figura 4. Organización de los bytes en la memoria del MIPS.

3. Data segment

Los datos en MIPS se almacenan en el segmento de datos (data segment). Los datos se definen cuando el programa se ensambla y se colocan en el segmento cuando el programa se carga en la memoria y empieza a ejecutarse. Por eso, estos datos son estáticos, es decir, su tamaño no puede modificarse durante la ejecución del programa.
Para definir los datos se utiliza la directiva .data <dirección>. La dirección es opcional, si no lo indicamos supone que empieza en 0x10001000. A continuación, podemos definir datos del tamaño que nos interesa.
Por ejemplo, el código:

```
.data
A: .word 0   #A estará en la dirección 0x10001000
B:  half 0   #B estará en la dirección 0x10001004
C:  byte 0   #D estará en la dirección 0x10001006
```

Define a partir de la dirección 0x10001000 los datos A de tamaño palabra, B de tamaño media palabra (*half*) y C de tamaño byte. Fijaos que hemos utilizado etiquetas para definir los datos (las etiquetas hacen referencia a la dirección donde está el dato) y las directivas `.word`, `.half` y `.byte` para delimitar el tamaño. El valor que hay después, en todos los casos 0, es el valor inicial que le damos.

4. Acceso a la memoria

El MIPS sólo permite acceder a la memoria mediante dos operaciones, la lectura y la escritura en la memoria. Los operadores que proporciona el MIPS para hacer estas operaciones nos permiten acceder a bytes (lb y sb), medias palabras (lh y sh) o palabras (lw, sw). Veamos ahora como hace el MIPS las operaciones de lectura y escritura en memoria:

Lectura y escritura en memoria

Las instrucciones de leer en memoria lw (*load word*), lh (*load half*) y lb (*load byte*) siguen la forma general, `op rt,k(rs)` donde op es la operación (lw, lh, lb) y k es un desplazamiento de 16 bits con signo que se sumará al contenido del registro rs para formar la dirección donde se encuentra el dato. Una vez leída se almacenará en el registro rt. En la figura se puede observar el proceso:

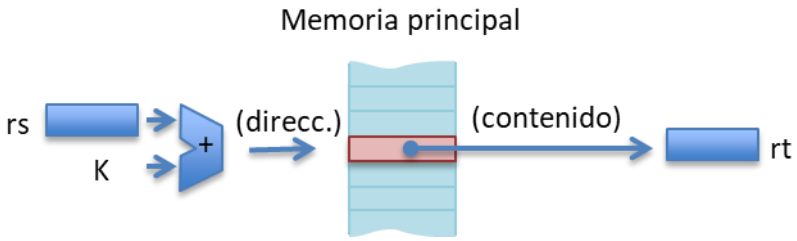

Figura 5. Operación de lectura en memoria (lw, lh, lb).

De manera similar, las instrucciones de escritura en memoria sw (*store word*), sh (*store half)* y sb (*store byte*) siguen la forma general, `op rt,k(rs)` donde op es la operación (sw, sh, sb) y k es un desplazamiento de 16 bits con signo que se sumará al registro rs para formar la dirección donde se almacenará el dato que hay en el registro rt. En la figura se puede observar el proceso:

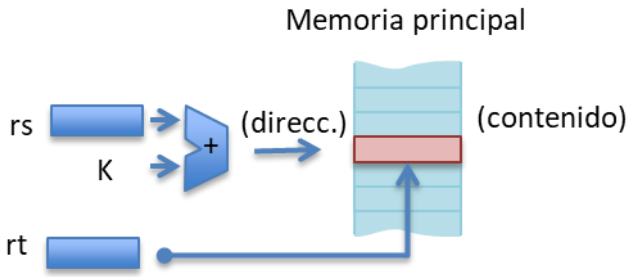

Figura 6. Operación de escritura en memoria (sw, sh, sb).

Tanto las instrucciones de lectura como de escritura en memoria siguen el formato de instrucción I, en el cual además del campo código de operación, hay dos campos para

especificar los registros Rs y Rt y un campo de 16 bits que indicará el desplazamiento expresado en complemento a 2.

Código ejemplo 1

```
#Acceso a la memoria. Lectura y escritura

.data                #Definición segmento de datos
A: .word 25
B: .word 10
C: .word 0

.text                #Comienza el programa
la $t0,A
la $t1,B
la $t2,C
lw $s0,0($t0)
lw $s1,0($t1)
add $s2,$s1,$s0
add $s2,$s2,$s2
sw $s2,0($t2)

li $v0, 10           #Acaba el programa
syscall
```

Como los tres datos del programa están representados por etiquetas podemos utilizar la pseudoinstrucción `la` para obtener su dirección en memoria y hacer uso en cualquier lugar del programa.

- Ensamblad el programa y comprobad que podéis ver tanto el segmento de texto como el segmento de datos. Comprobad que podéis ver los datos en hexadecimal y en decimal.
- ¿Cuántos bytes de la memoria principal están ocupados por datos del programa?
- ¿Cuántas instrucciones de acceso a la memoria contiene el programa?
- ¿Qué valor tiene el registro $t1 cuando se ejecuta la instrucción `lw $s1,0($t1)`?
- ¿En qué dirección se almacena el resultado?
- Sustituid la instrucción `sw $s2,0($t2)` por `sw $s2,2($t2)` ¿Qué ocurre cuando se intenta ejecutar el programa? Razonad la respuesta
- ¿Cuál es la codificación en lenguaje máquina de la instrucción `lw $s1,0($t1)`? Desglosa la instrucción en los distintos campos del formato.

Instrucciones de acceso a la memoria

En la siguiente tabla se resumen las instrucciones de lectura y escritura en memoria y algunas pseudoinstrucciones proporcionadas por el MIPS:

Instrucción	Ejemplo	Significado	Comentarios
Load word	lw Rt, k(Rs)	Rt ← Memoria[Rs+k]	Carga la palabra de la dirección *Rs*+k en el registro *Rt*.
Load halfword	lh $1, k(Rs)	Rt ← Memoria [Rs+k]	Carga la media palabra de la dirección *Rs*+k en el registro *Rt*.
Load halword unsigned	lhu Rt, k(Rs)	Rt ← Memoria [Rs+k]	Carga la media palabra sin signo de la dirección *Rs*+k en el registro *Rt*.

Load byte	lb Rt, k(Rs)	Rt ← Memoria [Rs+k]	Carga el byte de la dirección *Rs*+k en el registro *Rt*.
Load byte unsigned	lbu Rt, k(Rs)	Rt ← Memoria [Rs+k]	Carga el byte sin signo de la dirección *Rs*+k en el registro *Rt*.
Load label	lw Rt, Etiqueta	Rt ← M[Etiqueta]	Pseudoinstrucción. Carga la palabra señalada por la Etiqueta en el registro *Rt*.
Store word	sw Rt, k(Rs)	Memoria [Rs+k] ← Rt	Almacena en la palabra de la dirección *Rs*+k el contenido de *Rt*.
Store halfword	sh Rt, k(Rs)	Memoria [Rs+k] ← Rt	Almacena la media palabra de la dirección *Rs*+k el contenido de *Rt*.
Store byte	sb Rt, k(Rs)	Memoria [Rs+k] ← Rt	Almacena en el byte de la dirección *Rs*+k el contenido de *Rt*.
Store label	sw Rt, Etiqueta	Memoria [Etiqueta] ← Rt	Pseudoinstrucción. Almacena en la palabra señalada por la Etiqueta el contenido del registro *Rt*.

Tabla 1. Instrucciones y pseudodinstrucciones de acceso a la memoria

Las pseudoinstrucciones de acceso a la memoria se apoyan con el registro $at para obtener la dirección. Al igual que comentaremos al estudiar las instrucciones de salto, tened cuidado al programar porque el ensamblador podría destruir el valor contenido en ese registro

4. Uso de la memoria

Utilizamos un código de partida ejemplo para clarificar el uso de la memoria del *data segment* por el MIPS.

```
# Ejemplo 1: Uso de la memoria

.data                       # comienza zona de datos
palabra1:.word 15           # decimal
palabra2: .word 0x15        # hexadecimal
mediapalabra1: .half 2
mediapalabra2: .half 6
Dosbytes: .byte 3,4
.align 2                    #Alinea a palabra
byte1: .byte 8
byte2: .byte 5
espacio: .space 4           # Reserva 4 bytes a 0
cadena1: .asciiz "Estructura de los computadores"
```

Análisis del código

Este código sólo contiene directivas, es decir, instrucciones al ensamblador que le indican como almacenar datos en el segmento de datos de la memoria. Al ensamblar este código podéis comprobar que el segmento de texto es vacío porque no hay instrucciones, pero en el segmento de datos encontramos los valor que hemos definido. Con las directivas se pueden definir variables de distintos tipos e iniciarlas con valores; se pueden definir palabras (.word) medias palabras (.half), bytes (.byte) y cadenas ASCII (.asciiz). La directiva .space permite reservar memoria del segmento de datos. Resulta útil para definir variables sin inicializar. Con la directiva .align 2 indicamos al

ensamblador que la siguiente variable que definimos estará alineada a 2^2 bytes (recuerda lo que se ha comentado antes sobre el direccionamiento de la memoria)
El segmento de datos tendría un aspecto parecido al de la figura 7:

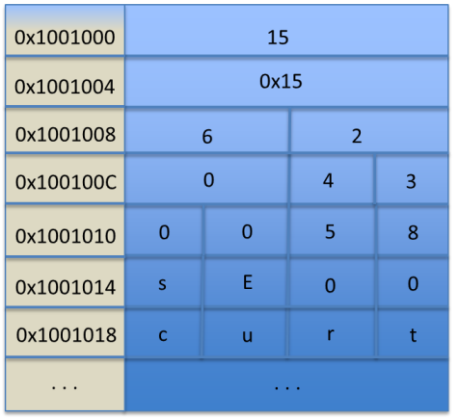

Figura 7. Aspecto de la memoria al hacer uso de directivas.

- Escribe y ensambla el código y comprueba como queda el segmento de datos.
- Cambia el tipo de visualización de los datos (hexadecimal, decimal, ASCII) marcando las casillas adecuadas.
- ¿Cuál es la dirección del byte donde está el carácter '1'?

En la siguiente tabla se resumen algunas de las principales directivas del MIPS que utilizaremos en las próximas prácticas.

Directiva	Descripción
.align n	Alinea el siguiente dato sobre un límite de 2^n bytes
.ascii "cadena"	Almacena la cadena de carácteres en la memoria, no acaba con carácter nulo.
.asciiz "cadena"	Almacena la cadena de carácteres en la memoria, acaba con carácter nulo.
.byte b1,..bn	Almacena n cantidad de 8 bits en posiciones consecutivas de la memoria.
.data <dirección>	Los elementos siguientes son almacenados en el segmento de datos. Si està presente el argumento <dirección>, los datos se almacenarán a partir de dicha dirección.
.double d1,..dn	Almacena n números de doble precisión y coma flotante (64 bits) en posiciones consecutivas de la memoria
.half h1,..hn	Almacena n cantidad de 16 bits en posiciones consecutivas de memoria.
.space n	Reserva n bytes de espacio en el segmento de datos.
.text < dirección >	Define el comienzo del segmento de código. Si se especifica la dirección, comienza a partir de esa posición.
.word w1,..wn	Almacena n cantidad de 32 bits en posiciones consecutivas de la memoria.

Tabla 2. Directivas del MIPS.

5. Los punteros

Se denomina puntero a cualquier variable (bien un registro bien una posición de memoria) que contiene una dirección de la memoria. Un ejemplo es el registro Contador de Programa (PC) que contiene la dirección de la siguiente instrucción a ejecutar, hay procesadores en los cuales este registro recibe el nombre de Instruction Pointer (IP).
La imagen mental que se hacen los programadores con los punteros es el de una flecha que apunta a una posición de memoria, de aquí el nombre. Así decimos que el PC *apunta* a la siguiente instrucción que ejecutará el procesador.
Ya conocemos la pseudointrucción `la` (*load address*) que asigna una dirección a un registro, ahora podemos decir que esta pseudoinstrucción nos sirve para que un registro *apunte* a una posición de memoria representada por una etiqueta.
Con los punteros podemos hacer uso de operaciones de suma y resta. Por ejemplo, el PC se incrementa en 4 en cada ciclo de instrucción para *apuntar* a la siguiente instrucción. Con una instrucción de salto se suma una cantidad al PC para desplazarse hacia arriba o hacia abajo en el código.
Como las direcciones son siempre positivas las operaciones con punteros se hacen siempre en binario natural y se utilizan exclusivamente las instrucciones addu y addiu, no tiene sentido hablar de signo.
Observamos ahora como se representan los punteros definidos en alto nivel en lenguaje ensambladores del MIPS. La siguiente definición de código en C:

```
int X = 10;
int * p;
```

Tiene la siguiente equivalencia en ensamblador del MIPS:

```
.data
X: .word 10
p: .space 4
```

La sentencia en lenguaje C:

```
p = &X; /* p apunta a X */
```

Podría escribirse en ensamblador MIPS como:

```
la $s0,X
la $s1,p
sw $s0,0($s1)
```

o utilizando pseudoinstrucciones:

```
.text
la $s0,X
sw $s0,p
```

La sentencia en lenguaje C:

```
*p = *p + 1; /* incrementa el entero
                al que apunta p */
```

Se traduce como:

```
la $s0,p
lw $s1,0($s0)
addi $s1,$s1,1
sw $s1,0($s0)
```

Código de partida

Consideremos el siguiente código de partida:

```
.data
A:    .word 6
B:    .word 8
C:    .space 4
X:    .byte 1

VAE1: .asciiz "Estructuras de los Computadores"
VAE2: .asciiz "Curso 2018-2019\n"
VAE3: .asciiz "\n El resultado de la suma es: "

.text
la $a0,VAE1
li $v0,4
syscall
li $a0,'\n'
li $v0,11
syscall
la $a0,VAE2
li $v0,4
syscall

la $a0,VAE3
li $v0,4
syscall

la $t0,A       # $t0 = &A
lw $t1,0($t0)  # $t1 = *$t0
lw $t2,4($t0)  # $t1 = *($t0+4)
add $t3,$t1,$t2
sw $t3,8($t0)
move $a0, $t3
addi $v0,$0,1
syscall              #Escribe un valor

li $v0, 10           #Acaba el programa
syscall
```

- Analiza el código y averigua que hace.
- Ensambla el código y ejecútalo. ¿Qué hace la función 4 para la instrucción syscall?
- ¿Cuál es la codificación máquina de la instrucción syscall?
- ¿En qué dirección se guarda el resultado de la suma?

Cómo hemos podido experimentar, la instrucción syscall nos permite interaccionar con el teclado y la consola para leer y escribir distintos tipos de datos, incluidas las cadenas

de caracteres como habéis visto en el ejemplo anterior. En la siguiente tabla se recogen algunas de las funciones de llamada más utilizadas con esta instrucción.

Servicio	Código de llamada	Argumentos	Resultados
Print_int	1	$a0=entero	Imprime en la consola el entero en $a0
print_string	4	$a0=dirección del comienzo de la cadena	Imprime en la consola el string que comienza en $a0
Read_int	5		Lee de la consola un entero y lo guarda en $v0
Read_string	8	$a0=dirección del buffer de entrada $a1=longitud	Lee de la consola un string que no puede ser más largo de $a1-1 y lo guarda a partir de $a0
Exit	10		Finaliza la ejecución
Print_char	11	$a0=Carácter	Imprime en la consola el byte de menor peso de $a0
Read_char	12		Lee de la consola un carácter y lo guarda en $v0
Print_int_hex	34	$a0 = entero	Imprime un valor de 8 dígitos en hexadecimal. Rellena con ceros a la izquierda si fuera necesario
Print_int_bin	35	$a0 = entero	Imprime un valor de 32 bits rellenando con ceros a la izquierda si fuera necesario
Print_int_unsig	36	$a0 = entero	Imprime un valor decimal sin signo

Tabla 3. Algunas de las funciones Syscall disponibles en MARS.

Ejercicio a entregar

- Escribe el código que lee dos enteros del teclado mostrando sendos mensajes por consola: uno que pida al usuario que introduzca el primer valor y tras haberlo leído que muestre otro solicitando el segundo valor. Los datos se almacenarán en la memoria, para lo cual debes haber reservado previamente espacio en el segmento de datos. Una vez almacenados los datos tienes que llamar a una función, que denominaremos SWAP, que intercambie el contenido de las dos posiciones de memoria. Para finalizar se leerán los valores guardados en la memoria y se mostrarán ordenados de menor a mayor en la pantalla.

Resumen

- El código ensamblador se almacena en una zona de la memoria del MIPS denominada *text segmen*t y los datos en una zona llamada *data segment*.
- El acceso a la memoria se realiza mediante operaciones de lectura y escritura.
- La memoria de MIPS permite acceso por byte, media palabra, palabra y doble palabra.
- Los datos definidos en la memoria no pueden utilizarse en operaciones aritméticas o lógicas, se tienen que traer primero a un registro del banco de registros.