

MIPS R2000

Práctica 1

Ariel Leonardo Fideleff

5 de agosto de 2022

Capítulo 1

Cuestiones

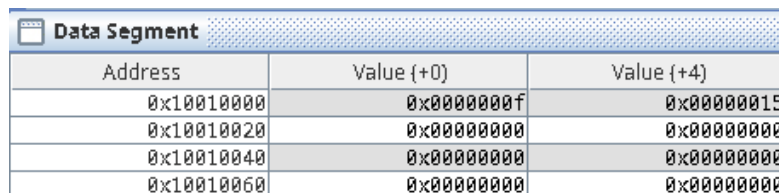
1. Apartado 1

– Declaración de palabras en memoria –

1.1 y 1.2

Como podemos ver en la Figura 1.1, los dos números enteros reservados en el programa se ubicaron en las posiciones `0x10010000` y `0x10010004`, distanciados justamente por 4 bytes ya que se corresponde con el tamaño de palabra. Es decir, si cada entero fue reservado como un `.word`, cada uno ocupa 4 bytes. Con esto, sumado a que el ensamblador los coloca uno seguido del otro en la memoria, dado que el primer valor por defecto comienza en la posición `0x10010000`, el segundo necesariamente deberá ubicarse 4 posiciones de memoria después (cada posición se corresponde con un byte).

En cuanto a los valores como tal, podemos ver la diferencia en que hayamos indicado uno en decimal, mientras el otro en hexadecimal. De esta forma, el número 15 en decimal es representado como `0x0000000f` en hexa, forma con la cual se presenta en el panel de datos. Mientras, el segundo valor, al haber sido especificado en el programa en base 16, lo reconocemos fácilmente como `0x00000015` en el panel en cuestión.



Address	Value (+0)	Value (+4)
0x10010000	0x0000000f	0x00000015
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000

Figura 1.1: Datos del programa según se indican en el panel de datos

1.3

De acuerdo a lo dicho en la teoría, las etiquetas `palabra1` y `palabra2` deberían tomar el valor de las posiciones de memoria a las que hacen referencia. En este caso, las ya mencionadas `0x10010000` y `0x10010004`, respectivamente.

De hecho, como se muestra en la Fig. 1.2, esto lo podemos comprobar en la ventana *Labels*, que se puede activar desde la configuración del simulador.

Labels	
Label ▲	Address
c1-1.asm	
palabra1	0x10010000
palabra2	0x10010004

Figura 1.2: Valores de las etiquetas *palabra1* y *palabra2* en la ventana *Labels*

1.4

Al ensamblar el programa dado, no parece presentar ninguna diferencia con respecto al primer programa visto, ya sea tanto en la memoria, como también en otras variables visibles en el simulador (por ejemplo, los registros).

1.5

El siguiente código cumple con la consigna planteada:

```

1      .data 0x10000000                                # comienzo zona de datos
2  vector: .word 0x10, 30, 0x34, 0x20, 60              # vector de 5 nros

```

Y en la Figura 1.3 podemos comprobar que los valores fueron almacenados de forma correcta, considerando que los números 30 y 60 en decimal se corresponden con 0x0000001e y 0x0000003c en hexadecimal, respectivamente.

Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10000000	0x00000010	0x0000001e	0x00000034	0x00000020	0x0000003c
0x10000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figura 1.3: Valores del vector de números declarado en el programa propuesto, según se indican en el panel de datos

Notar que al haber cambiado la dirección de memoria inicial donde se quiere que se almacenen los datos (respecto a la utilizada por defecto), debimos de expandir el panel de datos del simulador para poder visualizar el segmento de la memoria donde se ubicaban los valores reservados de nuestro vector, seleccionando la opción correspondiente desde un menú desplegable, tal como se lo muestra en la Figura 1.4.

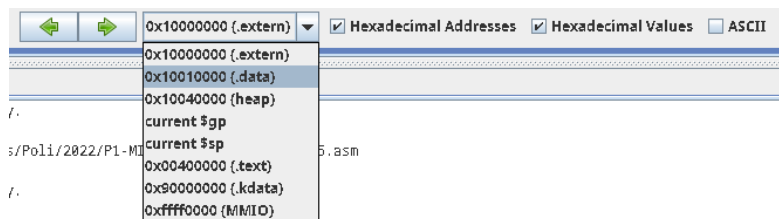


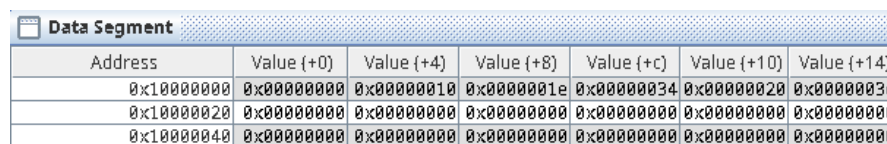
Figura 1.4: Menú desplegable para seleccionar la visualización del segmento de memoria correspondiente al utilizado por el programa planteado

1.6

Podemos probar cambiar el argumento de la directiva `.data` para intentar almacenar los datos partiendo desde la dirección `0x10000002`:

```
1          .data 0x10000002                                # comienzo zona de datos
2 vector:   .word 0x10, 30, 0x34, 0x20, 60                # vector de 5 nros
```

Hecho este cambio, el panel de datos nos muestra que los valores ahora son almacenados partiendo desde la dirección de memoria `0x10000004`, saltando de 4 en 4 (por el tamaño de palabra, Fig. 1.5).



Address	Value {+0}	Value {+4}	Value {+8}	Value {+c}	Value {+10}	Value {+14}
0x10000000	0x00000000	0x00000010	0x0000001e	0x00000034	0x00000020	0x0000003c
0x10000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figura 1.5: Valores y respectivas posiciones de memoria del programa con argumento de `.data` modificado

Esto difiere en principio de lo que uno podría esperar, ya que se le está indicando al ensamblador que ubique la información partiendo desde la dirección de memoria `0x10000002`. El motivo por el cual ejecuta el cambio descrito, es porque se requiere que la ubicación de todos los valores se encuentren en posiciones múltiplos de 4, de forma que la memoria “esté alineada”. Éste es un requisito de la arquitectura MIPS, o bueno, al menos estamos seguros basándonos en lo visto en la teoría, que lo es para el lenguaje de máquina de los microprocesadores MIPS R2000.

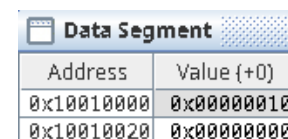
Con esto en cuenta, el ensamblador, sabiendo que indicamos el espacio para datos partiendo desde la posición de memoria `0x10000002`, buscó por la posición de memoria (mayor o igual) múltiplo de 4 más cercana, y a partir de allí ubicó los valores del vector de números reservado en el programa.

– Declaración de bytes en memoria –

1.7 y 1.8

Como ya sabemos, al no especificar ningún argumento para la directiva `.data`, los datos se ubicarán partiendo desde la dirección de memoria `0x10010000` por defecto.

Además, al tratarse de un único byte, y considerando que el valor indicado `0x10` no supera el tamaño permitido por la directiva (el máximo valor posible sería `0xFF`), éste se almacenaría “al comienzo de la palabra”, efectivamente implicando que el valor de la palabra que contiene el byte sea el mismo al especificado. Al fin y al cabo, el tamaño de una palabra es mayor al de un byte (justamente, 4 bytes), por lo que resulta en que el valor de la palabra sea el mismo al valor del tamaño de un byte, antecedido por 0s que no cambian el entero final almacenado.



Address	Value {+0}
0x10010000	0x00000010
0x10010020	0x00000000

Figura 1.6:
Representación del contenido especificado en la memoria, visto desde el panel de datos

1.9 y 1.10

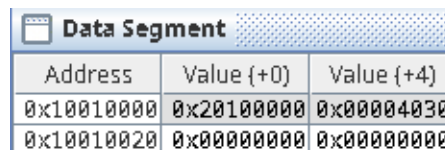
En este caso, se almacenan los valores **0x40302010** y **0x10203040** en las posiciones de memoria **0x10010000** y **0x10010004** respectivamente.

Si bien el segundo valor es indicado como tal de forma explícita en el código presentado, el primer valor, en cambio, es el resultado de almacenar un vector de 4 valores especificados del tamaño de 1 byte. Esto nos muestra que el simulador, por un lado, almacena los valores indicados con la directiva `.byte` de forma contigua, independientemente de que los valores sean accedidos de a una palabra a la vez. Por otro lado, nos indica también el orden que utiliza para guardar los datos dentro de una palabra, el cual puede ser identificado afín con el concepto de *Little-Endian*, el cual consiste en la organización y alineamiento de los datos de forma que, aplicado a este caso, “lo que va primero” se almacene en las direcciones de memoria más bajas, y así en adelante.¹

Esta conclusión la podemos obtener partiendo, primero, en que sabemos que el valor presentado en el panel de datos de una palabra **0xFFFFFFFF**, se corresponde con la representación en hexadecimal de los bytes en posiciones de memoria menores a mayores, interpretadas de derecha a izquierda. Para probar esto, podemos modificar el programa descripto para la consigna, y especificar que los datos se ubiquen partiendo desde la dirección de memoria **0x10010002**:

```
1      .data    0x10010002
2  palabra1:    .byte    0x10,0x20,0x30,0x40      # hexadecimal
3  palabra2:    .word    0x10203040                # hexadecimal
```

Con este cambio, la palabra en la dirección **0x10010000** almacena ahora el valor **0x20100000**, a la vez que la palabra en la dirección **0x10010004** almacena los dos valores restantes del vector de `.byte` (valor **0x00004030**, ver Fig. 1.7), demostrando cómo los bytes se corrieron dos posiciones desde la derecha (posiciones como bytes, que en hexadecimal se corresponde con dos dígitos, así representando un espacio de 4 dígitos para haber cambiado la dirección de memoria de `.data` dos posiciones adelante de la usada por defecto).



Address	Value (+0)	Value (+4)
0x10010000	0x20100000	0x00004030
0x10010020	0x00000000	0x00000000

Figura 1.7: Valores de las palabras en memoria correspondientes al vector de `.byte`, tras indicar que los datos se almacenen desde la posición **0x10010002**

Teniendo esto en cuenta, vemos que los valores del vector de `.byte` fueron almacenados en orden creciente respecto a posiciones de memoria que también crecen. Así, como en el programa indicamos los números **0x10**, **0x20**, **0x30** y **0x40** (en este orden), el primero de todos, el **0x10**, se encuentra en la posición de memoria **0x10010000**, luego el **0x20** en la posición **0x10010001**, y así sucesivamente.

Por ende, leemos el vector completo como el valor de una palabra **0x40302010** (siendo 4 elementos, exactamente la misma cantidad como hay de bytes en una palabra, siendo una sola suficiente

¹Más formalmente, el concepto de *Little Endian* hace referencia a que los bytes menos significativos dentro de una palabra sean almacenados en posiciones de memoria más chicas, relacionándose principalmente con la forma en la que se *internamente* se manejan y ordenan los bytes de una palabra.

para representar la totalidad de dicho vector). Mientras, guardar el valor 0x10203040 nos da una pauta de cómo se hubieran almacenado los valores del vector en cuestión si el ensamblador hubiera tenido un comportamiento que se acercara a *Big-Endian* (en el cual los “últimos números” se guardarían en las posiciones de memoria más pequeñas/bajas).

1.11

A pesar del comportamiento de juntar los valores del tamaño de 1 byte en palabras, esto no afecta sobre los valores de las etiquetas `palabra1` y `palabra2`, que indican la primera posición de memoria desde la cual parte cada valor (en este caso, digamos, el vector y el segundo valor). Además, ayuda que puntualmente el vector definido en la consigna pueda estar contenido en una sola palabra, como ya mencionamos.

Por lo tanto, las etiquetas `palabra1` y `palabra2` toman los valores 0x10010000 y 0x10010004 respectivamente.

– *Declaración de cadenas de caracteres* –