

# MIPS R2000

## Práctica 1

Ariel Leonardo Fideleff

4 de agosto de 2022

# Capítulo 1

## Cuestiones

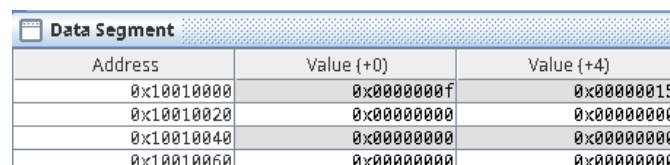
### 1. Apartado 1

#### – 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 0x00000015, respectivamente.

#### 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 1.2 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 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 1.2: 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 1.3.

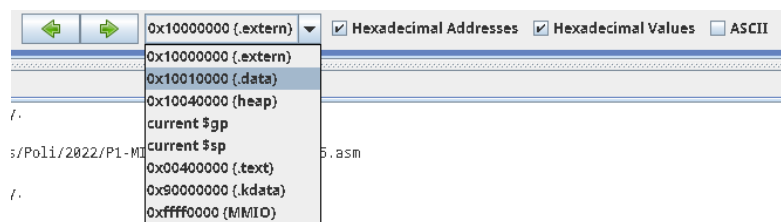


Figura 1.3: Menú desplegable para seleccionar la visualización del segmento de memoria correspondiente al utilizado por el programa planteado

## 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. 1.4).

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

Data Segment						
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 1.4: Valores y respectivas posiciones de memoria del programa con argumento de `.data` modificado

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 ubico los valores del vector de números reservado en el programa.