

# MIPS R2000

## Práctica 1

Ariel Leonardo Fideleff

10 de agosto de 2022

# Capítulo 1

## Cuestiones

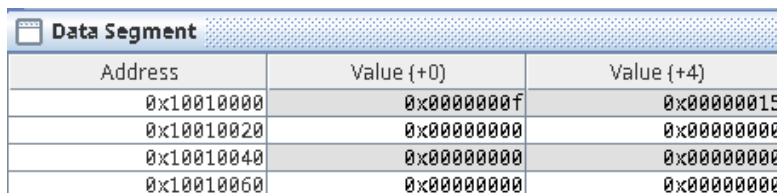
### 1. Apartado 1

#### – 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
<b>c1-1.asm</b>	
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.

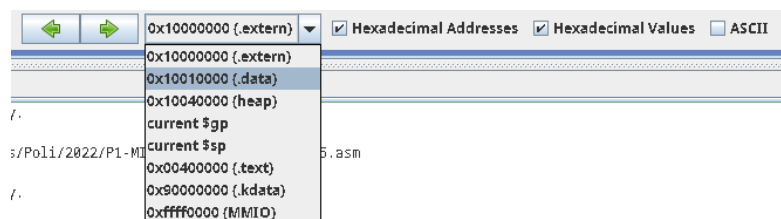


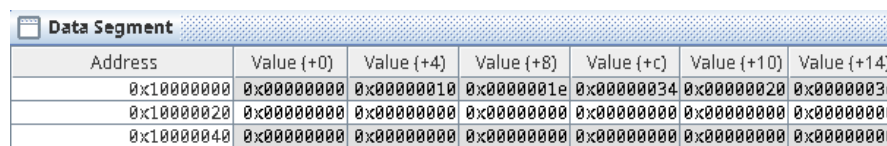
Figura 1.4: Menú desplegable para seleccionar la visualización del segmento de memoria correspondiente al utilizado por el programa planteado

## 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.

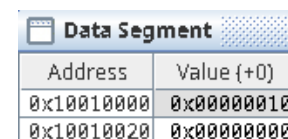
Con esto en cuenta, el ensamblador, sabiendo que indicamos el espacio para datos partiendo desde la posición de memoria `0x10000002`, buscó por la posición de memoria (mayor o igual) múltiplo de 4 más cercana, y a partir de allí ubicó los valores del vector de números reservado en el programa.

### – 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

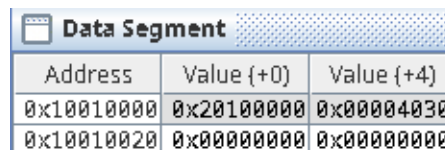
En este caso, se almacenan los valores **0x40302010** y **0x10203040** en las posiciones de memoria **0x10010000** y **0x10010004** respectivamente.

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.<sup>1</sup>

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

<sup>1</sup>Más 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.

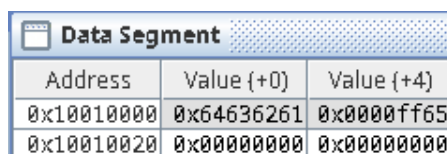
## – Declaración de cadenas de caracteres –

### 1.12

Para localizar la cadena ingresada en el programa dentro de la memoria, sabemos que debería de comenzar en la posición `0x10010000` ya que, como venimos repitiendo, es la dirección por defecto donde comienzan los datos si no se le provee un argumento a la directiva `.data`.

Observando la valor de la palabra en la dirección mencionada, nos encontramos con el valor `0x64636261`. El patrón que tiene este valor es familiar con el explorado en la sección interior, en el cual veíamos cómo múltiples bytes estaban almacenados dentro de una palabra. Con esto en cuenta, podemos pensar que ésta posiblemente contenga los bytes individuales `0x61`, `0x62`, `0x63` y `0x64`, leyendo de posiciones menores a mayores en memoria. Considerando que la forma en que estos valores crecen se asemeja mucho con cómo nuestra cadena contiene caracteres consecutivos y lexicográficamente crecientes, sumado a que el nombre de la directiva utilizada en el programa para almacenar la cadena recibe el nombre de `.ascii`, es intuitivo pensar que los bytes antes identificados se correspondan con las letras de la cadena en cuestión.

Comprobando nuestras sospechas, `0x61` equivale al número 97 en decimal, número correspondiente a la letra 'a' en el código ASCII. Sucesivamente el resto de bytes equivalen en decimal entonces a los números 98, 99 y 100, correspondientes a las letras 'b', 'c' y 'd' en ASCII.



Data Segment		
Address	Value {+0}	Value {+4}
0x10010000	0x64636261	0x0000ff65
0x10010020	0x00000000	0x00000000

Figura 1.8: Valores en memoria tras el ensamblado del código indicado

De todas formas, debemos recordar que nuestra cadena era "abcde", por lo que nos estaría faltando ubicar la letra 'e'. Como podemos ver en la Figura 1.8, la próxima palabra en la posición `0x10010004` contiene el valor `0x0000ff65`. El 65 en las posiciones menos significativas del valor surge como continuación de la cadena de caracteres, el cual se encuentra en la posición de memoria siguiente al 64 de la palabra anterior, pero que se nos muestra de esta forma al tener

que ubicarse por fuera del espacio de la primera palabra (en simples palabras, “no entraba” en la primera palabra), y por lo ya discutido en la sección anterior.

Luego, el `ff` que le sigue al `65` se corresponde con el valor almacenado con la directiva `.byte` dentro del programa.

### 1.13

Si empleamos la directiva `.asciiz` en vez de la directiva `.ascii` en el programa de la Cuestión anterior, el cambio que se observa es el agregado de lo que sería el equivalente a un byte *vacío* entre el final de la cadena, y el próximo valor almacenado con la directiva `.byte`. Es decir, mientras la primera palabra no cambia, la segunda en memoria toma ahora el valor `0x00ff0065`.

Este compartimiento es similar a la forma en la que, en lenguajes de programación como C, se agrega un caracter con valor ASCII = 0 al final de una cadena, conocido como el *terminador* (generalmente representado como `'\0'`). Haciendo uso de este caracter, se puede determinar en qué punto finaliza una cadena de caracteres, incluso conociendo sólo el comienzo de la misma y que sus elementos se encuentran en posiciones de memoria consecutivas. Por lo tanto, al operar con ella, es sólo cuestión de recorrer todas estas posiciones partiendo desde la primera, hasta encontrarse con un byte `0x00` que indica el final.

### 1.14

Podemos reemplazar la directiva `.ascii` con la `.byte` cargando posiciones en memoria con un vector de los caracteres de la cadena original, representados con su valor ASCII:

```
1      .data
2  cadena:      .byte  97, 98, 99, 100, 101          # defino string
3  octeto:      .byte  0xff
```

De hecho, podemos incluso utilizar los caracteres como tales, en vez de sus equivalentes en ASCII, y el ensamblador se encargará de transformarlos en sus valores numéricos correspondientes:

```
1      .data
2  cadena:      .byte  'a', 'b', 'c', 'd', 'e'        # defino string
3  octeto:      .byte  0xff
```

## – Reserva de espacio en memoria –

### 1.15 y 1.16

Observando el panel de datos con los valores en la memoria (Fig. 1.9), podemos ver que se ha reservado el tamaño equivalente a exactamente dos palabras, para la variable `espacio`. Esto equivale a 8 bytes, entendiéndose así que el parámetro de la directiva `.space` está expresado en cantidad de bytes que se quiere reservar en memoria.

Dicho esto, entonces el rango de posiciones que se han reservado en la memoria para la variable es `[0x10010004, 0x1001000b]`. Las palabras en posiciones `0x10010000` y `0x1001000c` contienen los valores `0x20` y `0x30` respectivamente, declarados con la directiva `.word` en el programa.

Data Segment				
Address	Value {+0}	Value {+4}	Value {+8}	Value {+c}
0x10010000	0x00000020	0x00000000	0x00000000	0x00000030
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000

Figura 1.9: Valores en memoria tras ensamblar el programa dado

## – Alineación de datos en memoria –

### 1.17

Se reservó el rango de posiciones `[0x10010001, 0x10010004]` en memoria para la variable `espacio`.

### 1.18

Si bien los cuatro bytes reservados podrían contener la información de una palabra, éstos no podrían funcionar como tal ya que, como dijimos en la Cuestión 1.6, la memoria de una palabra tiene que *estar alineada*, en el sentido que cada una debe comenzar en una posición de memoria múltiplo de 4.

Desde lo visto en la teoría, la arquitectura de MIPS nos permite leer información de la memoria de a una palabra a la vez, pues su tamaño equivale al ancho del bus de datos, y esto necesariamente debe ser en posiciones iniciales múltiplos de 4. Por tal motivo, si quisiéramos usar los 4 bytes reservados para `espacio` como un único espacio de memoria del tamaño de una palabra, realmente deberíamos acceder para cada operación necesaria a las palabras en las direcciones `0x10010000` y `0x10010004`, que lo hace poco práctico e ineficiente para este propósito. Esto sin mencionar las operaciones adicionales que se requerirían para juntar ambas palabras, o también descartar información en otras posiciones que se encuentren en las palabras, pero que no corresponda al espacio asignado para la tarea.

### 1.19

El `byte1`, al haberse declarado al comienzo del programa, fue inicializado en la posición de memoria `0x10010000`. Mientras, como el `byte2` fue declarado al final del programa, éste fue posicionado en la dirección de memoria `0x10010005`, después del espacio reservado para la variable `espacio`.

### 1.20

Finalmente, la variable `palabra` fue inicializada partiendo de la posición `0x10010008` en la memoria. Esto se debe a que como la variable fue declarada con la directiva `.word`, ésta debe de ocupar una palabra entera, a pesar de que la memoria esté ocupada hasta la posición `0x10010005`. En consecuencia, la variable se ubica en la posición ya mencionada, teniendo en cuenta que la anterior palabra en la posición `0x10010004` tiene un byte ocupado por la variable `byte2`.

Esta situación es similar a la vista en la Cuestión 1.6, en el cual la declaración de una variable con `.word` fue dada a partir del comienzo de la *palabra* más próxima libre, en vez de comenzar desde la primera *posición* libre.



Data Segment			
Address	Value {+0}	Value {+4}	Value {+8}
0x10010000	0x00000010	0x00002000	0x0000000a
0x10010020	0x00000000	0x00000000	0x00000000

Figura 1.10: Vista de la memoria posterior al ensamblado del código de las Cuestiones 1.17-1.20

## 1.21 y 1.22

De haber entendido correctamente el rol de la directiva `.align`, ésta hará que la próxima directiva de declaración de algún dato se coloque partiendo desde la próxima potencia de  $2^n$ , siendo  $n$  el argumento que recibe la directiva. En este caso, como la directiva recibe al número `2` como argumento, la variable `espacio` se reservará partiendo desde la primera posición múltiplo de  $2^2 = 4$  libre más cercana. Considerando que la primera posición de memoria libre a este punto es la `0x10010001` (ya que `byte1` ocupa la anterior), la próxima posición múltiplo de 4 disponible es la `0x10010004`. Podemos confirmar nuestra sospecha al ver el valor de la posición de memoria a la que hace referencia `espacio` en la ventana Labels (Fig. 1.11).

Labels	
Label ▲	Address
c1-21.asm	
byte1	0x10010000
byte2	0x10010008
espacio	0x10010004
palabra	0x1001000c

Figura 1.11: Valores en memoria de las etiquetas para el código de las Cuestiones 1.21-1.22

Luego, como la directiva `.space` recibe un `4` como argumento, se reservan 4 posiciones desde la inicial, efectivamente inicializándose en el rango `[0x10010004, 0x10010007]`.

El hecho de reservar 4 posiciones de memoria (4 bytes), como también haber alineado la variable con una posición múltiplo de 4, hacen que el espacio reservado pueda constituir una palabra. No sólo el tamaño es igual al de una palabra (como ya vimos en la pregunta de la Cuestión 1.18), sino que también está alineada apropiadamente con el comienzo de una palabra. Al fin y al cabo, justamente es la directiva `.align` la cual se encargó de hacerlo, ya que al indicarle el `2` como argumento, alineó el espacio con una posición múltiplo de  $2^2 = 4$ , como es apropiado para una palabra (según lo discutido en la Cuestión mencionada).

Con todo esto en cuenta, podemos decir resumidamente que:

*La directiva `.align` ubica en memoria el próximo dato declarado a partir de la primera posición libre que sea múltiplo de  $2^n$ , siendo  $n$  el argumento pasado a la directiva.*

## 2. Apartado 2

### – Carga de datos inmediatos (constantes) –

## 2.1

La dirección de memoria donde se encuentra la instrucción en cuestión es la **0x00400000**, y ocupa el tamaño de exactamente una palabra (4 bytes = 32 bits). El valor que la representa es **0x3c108690**. Como sabemos que es una traducción de código en Assembly que conocemos, podemos deducir las partes que lo componen. Sin embargo, si bien algunas de ellas pueden ser reconocidas desde su representación en hexadecimal, deberemos de remitirnos a su equivalente en binario para darle completo sentido a cómo es que realmente funciona.

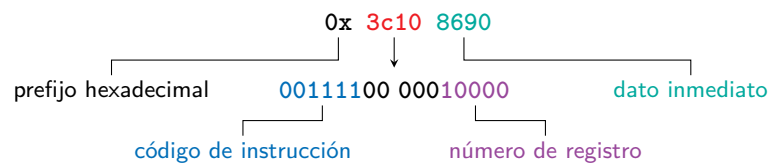


Figura 1.12: Formato del valor contenido en memoria para la instrucción ingresada

En la Figura 1.12 vemos las 4 partes del valor en memoria para la instrucción:

- El **prefijo hexadecimal**, el cual ya bien sabemos nos indica que un valor dado está representado en una notación numérica con base 16.
- El **código de instrucción** (conocido como *opcode*), es decir, un valor que nos indica qué tipo de instrucción se está indicando a ejecutarse. En el código tratado, se corresponde con informar que se quiere llevar a cabo la instrucción **lui**.

El motivo por el cual para poder separar este campo debemos convertir en binario los primeros dos bytes de la palabra, es que el código de instrucción ocupa los primeros 6 bits de la palabra. Por lo tanto, al pasarse a base 16, no es claramente distinguible. Por la misma razón así sucede con los siguientes dos grupos de 5 bits (el primero de ellos ignorado para la instrucción **lui**).

- El **número de registro**, que contiene el identificador del registro al cual el dato inmediato será cargado. Si bien en código Assembly éste es representado con un nombre **\$s0**, internamente los registros son enumerados del 0 al 31, haciendo que esta abstracción sea interpretada por el ensamblador al cargarse la instrucción en memoria. De hecho, en el programa dado se puede reemplazar **\$s0** por **\$16** sin repercusiones.
- El **dato inmediato**, el cual como ya vimos en la teoría, es un valor del tamaño de media palabra (2 bytes = 16 bits), el cual será cargado en los 16 bits más significativos del registro especificado. Debido a su tamaño en bits múltiplo de 4, sumado a que se lo representa en el código en hexadecimal dentro del código en cuestión, lo hace fácil de distinguir en la representación en hexa del valor en memoria.

## 2.2

Efectivamente, al correr el programa, los primeros 2 bytes del registro son cargados con el valor **0x8690**, así quedando la palabra almacenada en el registro como **0x86900000**.

## 2.3 y 2.4

De forma similar a la Cuestión anterior, el dato inmediato especificado como argumento a la instrucción `li` es cargado en el registro indicado como argumento. La diferencia es que este valor es del tamaño de una palabra completa, en vez de media palabra como sucedía con la instrucción `lui`.

Ante esto, tras revisar la interpretación del código ensamblado en el panel de texto (*Text Segment*), vemos que, siendo que `li` es realmente una *pseudoinstrucción*, fue descompuesta en dos instrucciones del procesador: la ya vista `lui`, y otra llamada `ori`.

Conociendo ya el funcionamiento de `lui` en base a lo visto en la Cuestión anterior, podemos entender que carga los primeros dos bytes (es decir, los más significativos) del valor inmediato utilizado en el código, al registro `$at` (abreviación de *Assembler Temporary*). Este registro está reservado por el ensamblador para las operaciones realizadas como *pseudo comandos*, es decir, operaciones intermedias llevadas a cabo por las instrucciones reales que fueron obtenidas de la interpretación de una instrucción en el código Assembly.<sup>2</sup> La interpretación de pseudoinstrucciones suele involucrar el uso del registro `$at`, como es el caso en cuestión.

Posteriormente, se hace uso de la instrucción `ori`, también llamada *OR Immediate*, el cual dado un registro de destino, un registro origen y un valor inmediato, almacena el resultado de la operación OR binaria entre los últimos dos, en el registro de destino. El valor inmediato dicho debe de tener un tamaño de, al igual que en `lui`, media palabra. Por lo tanto, en este caso, al correr esta instrucción se procede a almacenar en el registro de destino, originalmente especificado en la pseudoinstrucción `li`, el resultado de hacer el OR binario entre:

- Los dos bytes menos significativos del valor que se quería cargar originalmente al registro en el programa.
- Los otros dos bytes anteriormente almacenados en `$at` con la instrucción real `lui`.

Notar que, como la instrucción `lui` establece los otros dos bytes menos significativos de `$at` en 0, aplicar a continuación la operación OR no afectará al valor inmediato con el que se opera, resultando en lo que podría considerarse como “la carga de un valor” para estos dos bytes.

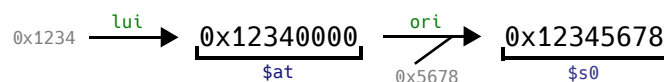


Figura 1.13: `li $s0, 0x12345678`, con las instrucciones reales que lo implementan

Pensando más a fondo, resulta claro el motivo de tener una pseudoinstrucción tal como `li`. Y es que, con un tamaño de palabra de 32 bits, donde los registros pueden almacenar una palabra, y las posiciones de memoria son accesibles por palabra, las instrucciones están limitadas a un máximo de 32 bits cada una. Como es estrictamente necesario que algunos de estos bits estén dedicados a indicar de qué instrucción se trata, es imposible disponer de los 32 bits que se quieran cargar sobre un registro, dentro de una sola instrucción. En consecuencia, la idea de una pseudoinstrucción que haga el trabajo de partir un valor inmediato proporcionado para que pueda ser cargado en un registro, ayuda a la legibilidad del código y facilita la tarea del programador.

– **Carga de palabras (palabras de memoria a registro)** –

<sup>2</sup>[https://en.wikibooks.org/wiki/MIPS\\_Assembly/Register\\_File](https://en.wikibooks.org/wiki/MIPS_Assembly/Register_File)

## 2.5 y 2.6

Como ya hemos dicho, la zona de memoria dedicada a instrucciones comienza a partir de la dirección `0x00400000`. Allí encontramos que el programa ensamblado ocupa 3 palabras, es decir, la única instrucción en el código después de la directiva `.text` fue descompuesta por el ensamblador en tres instrucciones reales. Podemos ver cuáles son y los valores con los que son representados en memoria desde el panel de segmento de texto (Figura 1.14).

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	4: main: lw \$s0, palabra(\$0)
<input type="checkbox"/>	0x00400004	0x00200821	addu \$1,\$1,\$0	
<input type="checkbox"/>	0x00400008	0x8c300000	lw \$16,0x00000000(\$1)	

Figura 1.14: Direcciones de memoria, instrucciones y sus valores en memoria, posteriores al ensamblado del programa dado

Es curioso ver por qué el simulador tradujo la instrucción original en las 3 vistas en la figura anterior. Particularmente porque la instrucción `lw` **no** es una pseudoinstrucción, sin mencionar que la última de las mencionadas 3 *es una instrucción lw*.

Debemos de recordar que la etiqueta `palabra` apunta a una dirección de memoria, por lo que tiene una longitud de 32 bits. Ya vimos en la sección anterior que es imposible contener un valor de tamaño tal en una sola instrucción, y es justamente por esto que el simulador recurre a un par de instrucciones adicionales para traducir la instrucción en el programa, a código máquina.

De esta forma, vemos en la primera instrucción traducida, el comienzo de un mecanismo muy similar al visto en la traducción de la pseudoinstrucción `li`, donde los 2 bytes más significativos del valor inmediato son cargados en el registro temporal `$at` mediante la instrucción `lui`. Sin embargo, en contraste con el proceso visto en la Cuestión anterior, no se hace uso de la instrucción `ori`.

La segunda de las instrucciones traducidas añade el valor del registro ingresado `$0` (que ya sabemos siempre contendrá un cero) a los dos bytes más altos de `palabra` cargados en `$at`, pareciendo ignorar los dos bytes menos significativos de `palabra`. Así luego, la tercera instrucción traducida utiliza como destino al registro así indicado en la instrucción original (`$s0`), y el registro `$at` junto con un valor inmediato `0x00000000` como dirección de memoria desde la cual cargar, permitiendo usar directamente el valor en `$at` sin modificarlo.

A pesar de la omisión mencionada, como la dirección de `palabra` en este caso tiene sus dos bytes menos significativos en 0, el código ensamblado funciona correctamente. Por este motivo, podemos probar agregar en el código la declaración de una palabra adicional antes de `palabra`, que obligue a esta última a ubicarse en una posición de memoria donde sus dos bytes menos significativos sean distintos de cero:

```
1      .data
2      .word 0xffffffff      # palabra dummy
3  palabra: .word 0x10203040
4      .text                # zona de instrucciones
5  main:  lw $s0, palabra($0)
```

Tras ensamblar el código con esta modificación, si bien no vemos instrucciones adicionales que consideren los dos bytes que creíamos eran omitidos, resulta que éstos son contemplados a partir del valor inmediato de la tercera instrucción traducida `lw`. Como ésta admite una media palabra como valor inmediato, puede contener aquellos bytes menos significativos y sumárselos a los otros dos más significativos cargados en `$at`, efectivamente representando la dirección de memoria de `palabra`, cuyo valor quiere cargarse en el registro `$s0`.

Es por esto que vemos en el panel del segmento de texto, que el valor inmediato de `lw` es `0x00000004`, para contar por el desfase de bytes que tuvo la etiqueta `palabra` tras agregar la declaración de una variable con la directiva `.word` al comienzo del segmento de datos (`.data`) [Fig. 1.15].

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	5: main: lw \$s0, palabra{\$0}
<input type="checkbox"/>	0x00400004	0x00200821	addu \$1,\$1,\$0	
<input type="checkbox"/>	0x00400008	0x8c300004	lw \$16,0x00000004{\$1}	

Figura 1.15: Panel del segmento de texto tras introducir la declaración de una nueva variable del largo de una palabra, anterior a la declaración de `palabra`

## 2.7

- `lui $1, 0x00001001` → `0x3c011001` = `00111100 00000001 00010000 00000001`
  - `lui` es la instrucción, su opcode es `0xF`, abarca los primeros 6 bits.
  - `$1` es el registro de destino, abarca 5 bits.
  - `0x00001001` o acortado `0x1001` es un valor inmediato con una longitud de media palabra (últimos 16 bits).

La instrucción carga en los 16 bits más significativos del registro identificado como `$1`, el valor inmediato `0x1001`.

- `addu $1, $1, $0` → `0x00200821` = `00000000 00100000 00001000 00100001`
  - `addu` es la instrucción, abarca los primeros 6 bits y comparte un opcode `0x0` con otras instrucciones.

Se identifica de otras instrucciones con dicho opcode a partir de los bits en el campo conocido como *funct*, que en este caso es `0x21`, y está arriba **resaltado en amarillo**.

- `$1` es el registro de destino, abarca 5 bits.
- `$1` y `$0` son ambos registros de origen y operandos de la instrucción, abarcan 5 bits cada uno.

La instrucción escribe en el registro de destino `$1`, el resultado de la suma *no signada* (es decir, sin importar el signo) de los números almacenados en los registros de origen `$1` y `$0`. Sabiendo que `$0` siempre contiene el número cero, y que el otro de los registros de origen es el mismo que el registro de destino, entonces el valor no cambia.

■

# Capítulo 2

## Problemas

### – Apartado 1 –

1.

```
.data
dato: .byte 3          # inicializo una posición de memoria a 3
      .text
      .globl main      # debe ser global
main:  lw $t0,dato($0)
```

En **rojo** se indican las etiquetas, en **azul** las directivas, en **gris** los comentarios, y en **celeste** las instrucciones.

2.

```
      .data 0x10000000
A:    .space 80
B:    .space 80
```

Utilizo dos directivas **.space** pasándoles el número *80* como argumento pues si una palabra son 4 bytes, y cada vector tiene que tener 20 palabras:  $20 \cdot 4 \text{ bytes} = 80 \text{ bytes}$ .

3.

```
      .data 0x10001000
palabra1: .space 4
byte1:   .space 1
         .align 2
palabra2: .space 4
```

4.

```
      .data
palabra1: .word 3
byte1:   .byte 0x10
         .align 2
espacio1: .space 4
byte2:   .byte 20
```

5.

```
.data
palabra1: .ascii "Esto es un problema"
palabra2: .byte 'E','s','t','o',' ','e','s',' ','u','n',' ','p','r','o',
↪ 'b','l','e','m','a'
palabra3: .word 0x6f747345, 0x20736520, 0x70206e75, 0x6c626f72,
↪ 0x00616d65
```

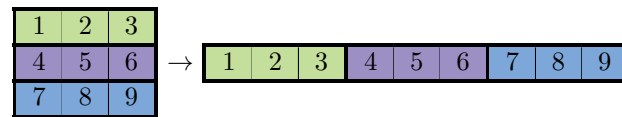
6.

```
.data
por_filas: .byte 1, 2, 3, 4, 5, 6, 7, 8, 9
por_columnas: .byte 1, 4, 7, 2, 5, 8, 3, 6, 9
```

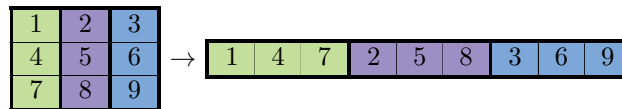
En ambos casos uso vectores de bytes, ya que los números de la matriz son lo suficientemente pequeños para ser almacenados de esta manera.

Al guardar la matriz por filas, termina resultando en un orden creciente, ya que entendemos que al “almacenar por filas”. vamos guardando las filas de arriba hacia abajo, y los valores de cada una se almacenarán de forma consecutiva, como se leen de izquierda a derecha.

Mientras, al guardar la matriz por columnas, entendemos que almacenamos las columnas de izquierda a derecha, y que los valores de cada una de ellas se almacenarán de forma consecutiva, como se leen de arriba a abajo.



(a) Almacenar por filas



(b) Almacenar por columnas

Figura 2.1: Visualización de las dos formas de almacenar la matriz  $A$  del ejercicio