

MIPS R2000

Práctica 1

Ariel Leonardo Fideleff

13 de agosto de 2022

Capítulo 1

Cuestiones

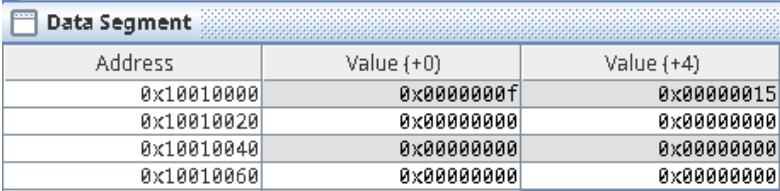
1. Apartado 1

– 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.

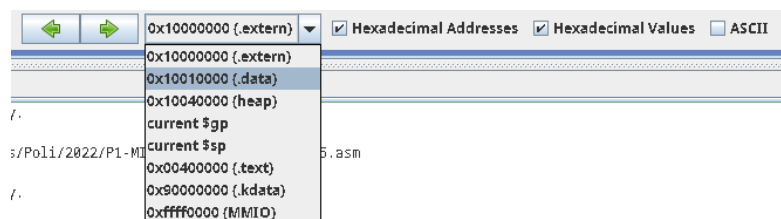


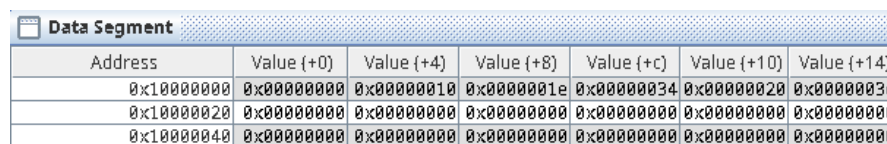
Figura 1.4: Menú desplegable para seleccionar la visualización del segmento de memoria correspondiente al utilizado por el programa planteado

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.

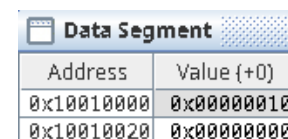
Con esto en cuenta, el ensamblador, sabiendo que indicamos el espacio para datos partiendo desde la posición de memoria `0x10000002`, buscó por la posición de memoria (mayor o igual) múltiplo de 4 más cercana, y a partir de allí ubicó los valores del vector de números reservado en el programa.

– 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

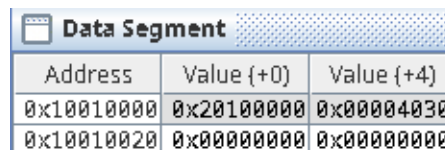
En este caso, se almacenan los valores **0x40302010** y **0x10203040** en las posiciones de memoria **0x10010000** y **0x10010004** respectivamente.

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.

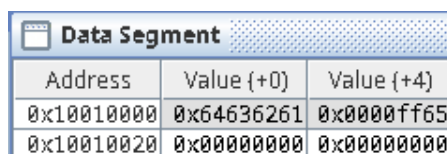
– Declaración de cadenas de caracteres –

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.

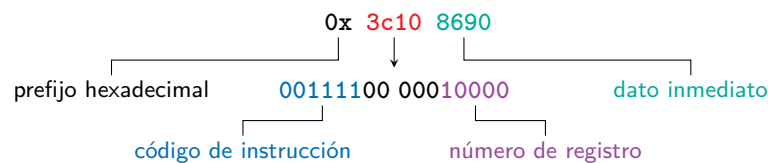


Figura 1.12: Formato del valor contenido en memoria para la instrucción ingresada

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 `$at` (abreviación de *Assembler Temporary*). Este registro está reservado por el ensamblador para las operaciones realizadas como *pseudo comandos*, es decir, operaciones intermedias llevadas a cabo por las instrucciones reales que fueron obtenidas de la interpretación de una instrucción en el código Assembly.² La interpretación de pseudoinstrucciones suele involucrar el uso del registro `$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.

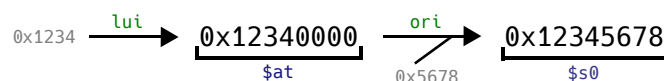


Figura 1.13: `li $s0, 0x12345678`, con las instrucciones reales que lo implementan

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)** –

²https://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.

- `lw $16, 0x00000000($1)` → `0x8c300000` = `10001100 00110000 00000000 00000000`

- La instrucción carga en el registro de destino **\$16** (también conocido como **\$s0**), el valor contenido en la dirección de memoria obtenida de la suma entre: el valor contenido en el registro **\$1**, y el valor inmediato proporcionado **0x0000**.

Como es de esperarse, al correr el programa, como `palabra` apunta a la dirección de memoria `0x10010000`, y el registro `$0` siempre contiene un valor 0; se carga el valor contenido en la dirección `0x10010000` (o sea, directamente a la que apunta `palabra`), `0x10203040`, en el registro de destino `$s0`.

Para poder cargar la dirección de un dato en un registro, podemos usar la pseudoinstrucción `la`. De esta forma podemos modificar el programa del ejercicio de manera que haga uso de dicha pseudoinstrucción, pero cumpliendo la misma tarea que el código original:

Verificando el panel del segmento de texto, podemos ver que la pseudoinstrucción utilizada, de forma prácticamente idéntica a la instrucción **li** vista en las Cuestiones 2.1-2.4, fue sustituida por las instrucciones **lui** y **ori**. Éstos son utilizados por el programa ensamblado para poder cargar los 2 bytes más significativos, y los 2 menos significativos, en el registro **\$t0**, respectivamente.

2.10

Partiendo del programa ya modificado de la Cuestión anterior, modificarlo nuevamente para cargar en el registro **\$s0** la palabra que se encuentra desde **palabra+1** es tan sencillo como cambiar el valor inmediato de la instrucción **lw**:

```
1      .data
2  palabra: .word 0x10203040
3      .text           # zona de instrucciones
```

```

4          la $t0, palabra
5 main:    lw $s0, 1($t0)

```

Ahora, en vez de usar el valor `0x0000`, usamos el número 1, para desfasar la dirección `+1` posiciones. Así vemos que también podemos usar valores decimales como valor inmediato de la instrucción `lw`.

Dicho esto, al intentar correr el programa, nos encontramos con un error de ejecución en la consola del simulador (Fig. 1.16).

```

Error in [redacted]/c2-10.asm line 5: Runtime exception at
0x00400008: fetch address not aligned on word boundary
0x10010001

Go: execution terminated with errors.

```

Figura 1.16: Error en la consola del simulador al intentar correr el código modificado

Resulta que, como venimos insistiendo a lo largo del trabajo, las palabras en la memoria deben estar necesariamente alineadas con direcciones múltiplos de 4. Esto no excluye el acceso a la memoria, por lo que, sabiendo que `palabra` sí es una dirección de memoria válida, la dirección de memoria `palabra+1` resulta inválida al no ser múltiplo de 4 (considerando obvio que la dirección de memoria debe ser la del **primer** byte de la palabra). Debemos de entender que al sumarle un 1 a la dirección, a partir de cambiar el valor inmediato de la instrucción `lw`, ésta la avanza en una posición, que se corresponde con un byte; no cuatro.

De hecho, el error nos indica cuál es la dirección de memoria que produjo el error: `0x10010001`. Si el valor al que apunta `palabra` es el primero (y único) declarado en el segmento de datos, por defecto su dirección será `0x10010000`. Fácilmente se puede ver que $0x10010000 + 1 = 0x10010001$.

2.11

Finalmente, para poder guardar los dos bytes de mayor peso (los más significativos) de `palabra` en `$s0`, podemos ajustar el offset del valor inmediato de la instrucción `lh` en 2 posiciones respecto a la posición de `palabra`.

Esto funciona ya que al especificar una dirección, como estamos indicando que se cargue una media palabra, la operación será aplicada sobre el byte en la posición utilizada como argumento y la siguiente a ella; son 2 bytes. Por lo tanto, considerando lo explorado en la Cuestión 1.10, sabiendo que MIPS (o al menos el simulador utilizado) almacena los valores bajo el formato *Little-Endian*, implicaría que si utilizamos `palabra+2` como parámetro a `lh`, los dos bits más significativos de la palabra serían cargados en el registro.

Recordar que *Little-Endian* es un formato/orden donde los bits menos significativos se ubican en las posiciones de memoria más bajas. En este sentido, si no desfásamos la posición de memoria de `palabra`, se tomarían los dos bits *menos* significativos. Pues si la palabra comienza en la posición apuntada por la etiqueta, como ésta es la posición de memoria más baja entre las 4 que abarca, bajo el concepto de Little-Endian, resultaría que las primeras dos posiciones se corresponden con los bits de menor peso.

El código resultante es el siguiente:

```
1      .data
2 palabra: .word 0x10203040
3      .text                                # zona de instrucciones
4      la $t0, palabra
5 main:   lh $s0, 2($t0)
```

Llama la atención que al tratarse de medias palabras, no hay fallos del simulador en cuanto a que la dirección especificada no sea múltiplo de 4. Al parecer, con *halfwords*, las direcciones deben estar alineadas con múltiplos de 2, ya que si intentamos reemplazar la instrucción `[lh $s0, 2($t0)]` con `lh $s0, 3($t0)`, el simulador falla al ejecutar el programa de una forma muy similar a la vista en la Figura 1.16. La única diferencia es que el mensaje advierte sobre el *halfword boundary* en vez del *word boundary*.

– Carga de bytes (bytes de memoria a registro) –

2.12

Al ensamblar el programa, vemos que la instrucción `lb` es traducida en memoria de instrucciones de forma muy similar a cómo lo hemos visto con la instrucción `lw` en la Cuestión 2.6. Como ya vimos, el Assembler lleva a cabo este procedimiento debido a la incapacidad de poder especificar directamente direcciones de memoria completas en una sola instrucción. Por ello, debe recurrir a un procedimiento análogo al mencionado en la anterior Cuestión:

1. Cargar los 2 bytes más significativos de la dirección de memoria a la que apunta, en este caso *octeto*, al registro temporal `$at`. [Instrucción `lui`]
2. Añadir el valor del registro especificado en conjunto con el valor inmediato (`$0` para este caso), al valor en `$at`. [Instrucción `addu`]
3. Reformular la instrucción `lb` sumando el registro `$at` a un valor inmediato que contiene los dos bits menos significativos de *octeto*.

Es decir, en resumen, si bien `lb` carga el byte de una dirección de memoria en un registro, la forma de especificar la dirección, como es para nuestro ejemplo con la etiqueta *octeto*, no cambia el hecho que su longitud sea de 32 bits, y que por ello deba de cargarse como ya hemos visto, por las limitaciones de la longitud de los registros de instrucciones.

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	6: main: lb \$s0, octeto(\$0)
<input type="checkbox"/>	0x00400004	0x00200821	addu \$1,\$1,\$0	
<input type="checkbox"/>	0x00400008	0x80300000	lb \$16,0x00000000(\$1)	

Figura 1.17: Instrucciones traducidas por el simulador tras ensamblar el programa dado

2.13 y 2.14

Al ejecutar el programa, nos encontramos con un compartamiento un poco diferente al esperado. Lo que probablemente creíamos que sucedería es que, tal y como nos fue dicho en la teoría, que

el byte en la dirección de memoria `octeto` se cargue en el registro `$s0`. Sin embargo, el valor que termina cargándose en dicho registro, es `0xffffffff3`.

Ante la disyuntiva propuesta luego en la Cuestión 2.14, probamos ejecutar el programa tras cambiar la instrucción `lb` por `lbu`. Efectivamente, al correr el programa con este cambio, sí produce el efecto esperado. Ahora se carga el valor `0x000000f3` en el registro.

La pregunta entonces es, ¿cuál es la diferencia entre `lb` y `lbu`? ¿qué significa esa `u` al final?

```
octeto y $s0 (lbu)  →      0xf3   =  00000000 00000000 00000000 11110011
$s0 (lb)           →  0xffffffff3 =  11111111 11111111 11111111 11110011
```

Figura 1.18: Representaciones binarias de los números con los que trabajamos

Curiosamente, ya hemos visto la `u` mencionada al final de una instrucción: en `addu`. Para entender mejor esto, veamos las representaciones binarias de los números tratados (Fig. 1.18).

Como podemos ver, la única diferencia entre ambos números es el estado de los bits en los primeros tres bytes de mayor peso: con `lb` son todos establecidos en 1, mientras que se mantienen en 0 con `lbu`.

Consultando en documentación de MIPS, resulta que dicha `u` proviene de *unsigned*. Nos estábamos saltando por alto la forma en la que la computadora interpretaba internamente los números.

Según hemos visto con LC-3, necesariamente los números deben estar representados en código binario para poder ser almacenados en la memoria. Por lo tanto, si bien en el simulador vemos su representación hexadecimal (o decimal, cambiando una opción), no podemos simplemente asumir que los números se van a sumar mágicamente como estamos acostumbrados.

Con esto en cuenta, debemos comprender que en el caso de MIPS, los números son representados como complemento a 2 (*2's complement*), y soporta operaciones tanto con números **signed** como **unsigned**. Nos referimos a que con los números **signed**, podemos representar números negativos, con el costo de poder representar poco menos de la mitad de números positivos (concretamente, $2^{31} - 1$ positivos, pero también 2^{31} negativos, y el 0). Así, cuando una instrucción que lleve a cabo operaciones aritméticas o que maneje números en memoria, no tenga una letra `u` al final de su nombre, se entiende que tratará los valores proporcionados como enteros signados.

Volviendo entonces a la diferencia entre el comportamiento de `lb` y `lbu`, podemos entender cómo `lb`, al ver que el bit más significativo en el byte al que apunta `octeto` es un 1 (indicando que es negativo), al ser cargado en un registro con una longitud del cuádruple del valor dicho, extiende aquel 1 hacia todos los bits nuevos en el registro `$s0`, con el fin de **mantener el signo** del número original.

En cambio, como `lbu` trata los números como enteros sin signo, ignora este detalle, y carga el valor en el registro, dejando el resto de bits en 0.

2.15

Si se cambia la definición de `octeto` para ahora apuntar a un byte cuyo valor sea `0x30`, no existe diferencia entre el uso de `lb` y `lbu`. Esto se debe a que si observamos la representación binaria de `0x30` (`00110000`), el estado del bit más significativo es 0, indicando que como entero signado, es positivo. Por consecuencia, si usamos `lb`, para mantener el signo establecerá los bits restantes en `$s0` en 0. Mientras, como `lbu` ya por defecto deja el resto de bits en 0, el efecto de ambas instrucciones es el mismo.

2.16

Definiendo ahora a `octeto` para apuntar al valor de una palabra cuyo valor sea `0x10203040`, el valor cargado en `$s0` es `0x00000040`.

El motivo de esto es similar al visto en la Cuestión 2.11 usando la instrucción `lb` para cargar bytes, en vez de `lh` para medias palabras. Cuando `lb` recibe la dirección de memoria del comienzo de la palabra a la que apunta `octeto`, la instrucción interpretará cargar en el registro el byte que se encuentra en esa dirección. Sabiendo que el simulador almacena los valores en memoria bajo el formato *Little-Endian*, el valor de este byte será entonces el de los 4 bits menos significativos de la palabra en `octeto`, el cual en este caso es `0x40`.

Además, el hecho considerar el número como signado mediante el uso de `lb` no afecta a los bits restantes en el registro, pues el estado del bit más significativo en `0x40` es 0.

2.17

Cambiando la instrucción en `main` para cargar en el registro el valor en `octeto+1`, en vez de `octeto`, resulta en que el valor en `$s0` sea `0x00000020` tras ejecutar el programa.

En este caso no se produce error alguno, a diferencia de cómo si habíamos visto que sucedía con la instrucción `lw` para cargar palabras, por ejemplo. La razón de esto es que en este caso estamos cargando *bytes*, por lo que cualquier dirección de memoria es válida para un byte. El concepto de que “la memoria esté alineada” con múltiplos de 4 no aplica para bytes, ya que con las palabras, éstas tienen una longitud de 4 bytes, mientras las variables declaradas como `.byte`, bueno, ocupan un byte. En todo caso, la regla consistiría en que las direcciones de memoria “estén alineadas con múltiplos de 1”, lo cual abarca a todas las direcciones de memoria posibles.

La idea de que la memoria esté alineada está directamente relacionada con la longitud del campo con el que tratamos. Con *palabras* es con múltiplos de 4 por sus 4 bytes de longitud, con *medias palabras* es con múltiplos de 2 por sus 2 bytes de longitud, y con los *bytes* es con múltiplos de 1 por su único 1 byte de longitud.

En lo que concierne al motivo del valor cargado en `$s0`, estamos viendo el efecto de haber cargado el valor al que apunta la etiqueta `siguiente`. Como `octeto` apunta a un `.byte`, según lo ya visto en el apartado de *declaración de bytes en memoria*, como la siguiente variable declarada (`siguiente`) también es del tipo `.byte`, éste se ubica en la posición de memoria inmediatamente siguiente a la de `octeto`. Por lo tanto, la dirección a la que apunta `octeto+1`, es efectivamente la misma a la que apunta `siguiente`, la cual contiene el byte `0x20` que fue cargado en `$s0`.

Comentar también que como el estado del bit más significativo en `0x20` es 0, no se presentan diferencias que partan del uso de `lb` para cargar este valor en el registro `$s0`.

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

2.18

La primera instrucción del programa es `lw`, la cual ya hemos visto y mencionado reiteradas veces en este trabajo. En consecuencia, su traducción por parte del simulador es idéntica a la descripta y analizada en la Cuestión 2.7. Ver la Figura 1.19 por el detalle de este caso concreto.

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	7: main: lw \$s0, palabra1(\$0)
<input type="checkbox"/>	0x00400004	0x00200821	addu \$1,\$1,\$0	
<input type="checkbox"/>	0x00400008	0x8c300000	lw \$16,0x00000000(\$1)	

Figura 1.19: Traducción del simulador de la primera instrucción del programa provisto

2.19

Para poder comprobar mejor el efecto de la ejecución del programa, comparemos lado a lado el estado de la memoria antes y después de ejecutarlo (Fig. 1.20).

Data Segment				
Address	Value {+0}	Value {+4}	Value {+8}	
0x10010000	0x10203040	0x00000000	0xffffffff	
0x10010020	0x00000000	0x00000000	0x00000000	

(a) Estado antes de ejecutar el programa

Data Segment				
Address	Value {+0}	Value {+4}	Value {+8}	
0x10010000	0x10203040	0x10203040	0x10203040	
0x10010020	0x00000000	0x00000000	0x00000000	

(b) Estado después de ejecutar el programa

Figura 1.20: Comparación del estado de los valores de la memoria, antes y después de ejecutar el programa

Antes de ejecutar el programa, están cargados 3 valores en memoria, cada una ocupando el espacio de una palabra. En orden: 0x10203040, 0x00000000 y 0xffffffff.

Tras ejecutar el programa, las tres palabras contienen el mismo valor: 0x10203040.

Si vemos el paso a paso de las instrucciones del programa, podemos entender el motivo de esto:

1. La instrucción **lw** carga la palabra en la dirección de memoria marcada con la etiqueta **palabra1**, 0x10203040, en el registro **\$s0**.
2. Luego, la instrucción **sw** escribe el valor en el registro **\$s0**, 0x10203040, en la dirección de memoria a la que apunta **palabra2** (recordar que no agregamos offset porque el registro de operando es **\$0**, que siempre contiene el valor 0), **reemplazando** el valor anteriormente allí almacenado, 0x00000000.
3. Finalmente, de forma similar a la anterior instrucción, **sw** escribe el valor de **\$s0** (0x10203040) en la dirección de memoria a la que apunta la etiqueta **palabra3**, **reemplazando** el valor anteriormente allí almacenado 0xffffffff.

En conclusión, el programa sobrescribe el valor en las palabras a las que apuntan las etiquetas **palabra2** y **palabra3**, con el valor en la palabra a la que apunta **palabra1**.

– *Almacenado de bytes (bytes de registro a memoria)* –

2.20

Al igual que en la Cuestión 2.18, la primera instrucción es `lw`, por lo que ya sabemos cómo es traducida por el simulador.

En cuanto a la instrucción `sw`, su traducción es similar a la de `sw`, donde la forma en la que se carga la dirección de memoria a la que apunta `octeto` se corresponde con la ya descrita en la Cuestión 2.6, con la diferencia de que la tercera instrucción es `sb` en vez de `lw`, pues es la instrucción que tradujo el simulador.

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	6: main: lw \$s0, palabra{\$0}
<input type="checkbox"/>	0x00400004	0x00200821	addu \$1,\$1,\$0	
<input type="checkbox"/>	0x00400008	0x8c300000	lw \$16,0x00000000{\$1}	
<input type="checkbox"/>	0x0040000c	0x3c011001	lui \$1,0x00001001	7: sb \$s0, octeto{\$0}
<input type="checkbox"/>	0x00400010	0x00200821	addu \$1,\$1,\$0	
<input type="checkbox"/>	0x00400014	0xa0300004	sb \$16,0x00000004{\$1}	

Figura 1.21: Traducción de las instrucciones del programa por el simulador

2.21

Antes de ejecutar el programa, `octeto` apunta a un espacio en memoria del tamaño de media palabra.

Luego de correrlo, la instrucción `lw` carga la palabra a la que apunta `palabra` (0x10203040), en el registro `$s0`. Posteriormente, aquel valor es utilizado por la instrucción `sb` para escribir el byte **menos significativo** de la palabra, en la dirección de memoria a la que apunta `octeto`. Como dijimos que el valor es 0x10203040, su byte menos significativo se encuentra en la primera posición de memoria de la palabra (por la naturaleza *Little-Endian* del simulador), así éste byte siendo **0x40**.

En consecuencia, `octeto` guarda el valor **0x0040**, el cual expresamos como media palabra pues ésta es la longitud del espacio a la que apunta la etiqueta.

2.22

Para lograr el cambio pedido, podemos cambiar la instrucción `[sb $s0, octeto($0)]`, por `[sb $s0, octeto+1($0)]`. Producto de este cambio, ahora `octeto` contiene el valor **0x4000** tras ejecutar el programa.

Antes, al usar `octeto` directamente, se apuntaba al byte menos significativo del espacio al que la etiqueta apuntaba, por el formato *Little-Endian*. Al desfazar la dirección de memoria por un byte (reemplazando `octeto` por `octeto+1`), ahora el byte menos significativo de la palabra almacenada en `$s0` (allí cargada por la instrucción previa `lw`), se escribe mediante el uso de `sb` en el segundo byte menos significativo de `octeto` (que siendo el caso de una media palabra, es directamente la posición de mayor peso).

Es por este motivo que ahora `octeto` toma el valor **0x4000**, y en el código original **0x0040**: vemos el desfase de un byte de la posición a la que `sb` escribe.

2.23

Notar que como el programa original utiliza la instrucción `lw` para cargar la palabra **entera** en el registro `$s0`, y luego `sb` toma el byte menos significativo de allí para escribirlo en octeto, **no podemos** simplemente cambiar `palabra` por `palabra+3` en la instrucción `lw`. La instrucción `lw` requiere la posición de una palabra, entonces tiene que ser múltiplo de 4, a lo que `palabra+3` no cumple esta condición.

Por lo tanto, para lograr la consigna pedida, podemos cambiar la instrucción `lw` por `lbu`, la cual ya vimos en la Cuestión 2.14. De esta forma, ahora sí podemos reemplazar `palabra` por `palabra+3`, cargando desde un principio el byte ubicado en `palabra+3`, al registro `$s0`. Luego la instrucción `sb` tomará el byte en el registro, y lo escribirá en la posición de menor peso del espacio al cual apunta `octeto`.

El código resultante es el siguiente:

```
1      .data
2  palabra:      .word 0x10203040
3  octeto:       .space 2
4
5      .text                                # zona de instrucciones
6  main:        lbu $s0, palabra+3($0)
7              sb $s0, octeto($0)
```

Aclarar que hacemos uso de la instrucción `lbu` en vez de `lb`, para evitar cualquier diferencia que pueda surgir de interpretar el byte como un número signado. Si bien, en teoría, en este caso no debería hacer la diferencia dado que el bit menos significativo de `0x10` no está prendido, usamos `lbu` para más generalidad.

3. Apartado 3

– Operaciones aritméticas con datos inmediatos (constantes) –

3.1

El resultado de la suma efectuada se almacena en el registro `$t1`. Se suma 1 al valor de la palabra a la que apunta la etiqueta `numero`, el cual corresponde al máximo número positivo representable `0x7fffffff`. Por consiguiente, el valor contenido en `$t1` tras ejecutar el programa es `0x80000000`.

3.2 y 3.3

Al intentar correr el programa después de cambiar la instrucción `addiu` por `addi`, vemos que la ejecución falla informándonos el motivo el origen del error en la consola (Fig. 1.22).

```
Error in [redacted]/c3-2.asm line 6: Runtime exception at
0x0040000c: arithmetic overflow
```

```
Step: execution terminated with errors.
```

Figura 1.22: Error en la consola del simulador al intentar correr el código tras cambiar la instrucción `addiu` por `addi`

A partir de haber corrido el programa paso a paso, y como también se menciona en el error devuelto por consola, el problema surge en la línea 6 del programa. La causa es advertida como *arithmetic overflow*. Es decir, al sumarle un número positivo (valor inmediato igual a 1) al ya máximo número entero positivo representable (contenido en la palabra a la que apunta la etiqueta `numero`), el resultado se excede del límite representable, resultando en una excepción de desbordamiento (*overflow*).

Precisamente ésta es la diferencia entre las instrucciones `addi` y `addiu`: ambas permiten sumar un valor inmediato a un registro, pero con la diferencia en que la variante **no** terminada en *u*, provoca una excepción en el caso de producirse una instancia de *overflow* al llevar a cabo la operación aritmética.

– Operaciones aritméticas con datos en memoria –

3.4 y 3.5

El programa en cuestión obtiene el resultado de la resta de los valores a los que apuntan las etiquetas: `numero1` – `numero2` – `numero3`. Esta diferencia se almacena en la palabra a la que apunta `numero1` al final del programa (**NO** en `numero3`, como sugiere el enunciado), mediante la instrucción `sw`.

En este caso, el resultado almacenado en dicha dirección es `0x7ffffffe`. Sin embargo, considerando que los números con los que operamos están representados como enteros signados, el valor obtenido es **incorrecto**. Considerando que `0x80000000` es el mínimo número negativo representable (en decimal, -2147483648), restarle 2 ($-1 - 1$, concretamente) si bien no falla al usar la instrucción `subu`, resulta en un valor que “salta” al otro lado de la recta numérica, resultando en un valor correspondiente a un número positivo (en decimal, 2147483646).

Es por esta situación de desbordamiento que hay diferencia al usar la instrucción `sub` en vez de `subu`. Sustituyendo las instrucciones de esta manera, el programa falla en la línea 9 al ejecutarlo a paso, con un error idéntico al visto en la Figura 1.22. La única diferencia con lo visto en la Cuestión 3.3, es que en este caso el error se produce por sobrepasar el límite del *mínimo número entero signado representable*, en vez del límite del *máximo entero signado representable*, como fue el caso de la Cuestión mencionada.

– Multiplicación y división con datos en memoria –

3.6

Después de realizar la operación, se obtiene el resultado de la multiplicación entre los valores contenidos en las palabras a las que apuntan las etiquetas `numero1` y `numero2`, es decir, `0x7ffffff` · 16.

Debido a que el producto entre los dos números en cuestión sobrepasa el límite representable de números positivos signados, la instrucción `mult` permite prescindir del argumento que especifique el registro al que se almacene el resultado (en comparación con `mul`), de forma que se almacene el resultado dividido en dos registros especiales: `$hi` y `$lo`.

Como pueden llegar a sugerir los nombres de estos registros, `$hi` almacena los 4 bytes más significativos del resultado, mientras que `$lo` almacena los 4 menos significativos. Para poder acceder a estos registros especiales, se requiere del uso de las instrucciones `mfhi` y `mflo` respectivamente, las cuales cargarán su valor en un registro especificado como argumento.

Así, para el código dado, el resultado es almacenado en las posiciones `0x10010008` y `0x1001000c`, que al concatenarlas nos da el valor `0x7fffffff0` (en decimal, 4611686014132420609), que como es de esperarse, representa correctamente el producto entre los factores aclarados en el primer párrafo.

Finalizando con una aclaración a lo explicado del resultado del producto, notar que nunca necesitaremos más de dos palabras para almacenarlo, siempre que los factores sean dos números almacenados cada uno en una palabra. Pues el producto más grande posible, $0x7fffffff \cdot 0x7fffffff$ (o en decimal, $2147483647 \cdot 2147483647 = 2147483647^2$), da igual a `0x3fffffff00000001`, que es menor al mayor número representable con dos palabras (64 bits), `0x7fffffffffffffff`.

3.7

Podemos modificar de forma sencilla el código anterior para que divida el valor en `numero1` por el de `numero2`. Luego de cambiar los datos a los que apuntan las etiquetas dichas según lo indica el ejercicio, podemos cambiar la instrucción `mult` por `div`, la cual guarda el resto de la división en el registro especial `$hi`, y el cociente en el registro especial `$lo`.

Como ya leíamos los contenidos de estos registros y los escribíamos a continuación de los números en memoria (por usar `mult` previamente), el único cambio restante es invertir las instrucciones `mflo` y `mfhi` de lugar, con el único propósito de almacenar primero el cociente, y luego el resto en memoria, según se da a entender en el enunciado.

El programa resultante es el siguiente:

```
1      .data
2  numero1: .word 10
3  numero2: .word 3
4          .space 8
5          .text
6  main:   lw $t0, numero1($0)
7          lw $t1, numero2($0)
8          div $t0, $t1      # divide los dos números
9          mflo $t0          # resto
10         mfhi $t1          # cociente
11         sw $t0, numero2+4($0) # 32 bits más peso
12         sw $t1, numero2+8($0) # 32 bits menos peso
```

3.8

Modificamos el código del enunciado para cumplir con la consigna, a partir de sustituir el valor inmediato usado en `andi` por `0xffff0001`, ya que su representación en binario tiene los 16 bits más significativos y el bit 0 prendidos: 11111111 11111111 00000000 00000001.

```
1      .data
2  numero:  .word 0x3ff41
3            .space 4
4            .text
5  main:    lw $t0, numero($0)
6           andi $t1, $t0, 0xffff0001
7           sw $t1, numero+4($0)
```

El valor almacenado en `numero+4` con estas modificaciones resulta `0x00030001`. Comparándolo con el valor original, cumple con la tarea solicitada.

<code>numero</code>	→	<code>0x0003ff41</code>	=	00000000	00000011	11111111	01000001
<code>numero+4</code>	→	<code>0x00030001</code>	=	00000000	00000011	00000000	00000001

Figura 1.23: Representaciones binarias de la palabra original y el resultado tras ejecutar el programa modificado

3.9

Como el ejercicio requiere que ciertos bits queden en 1 en vez de 0, no podemos cumplir con la consigna tan solo modificando la instrucción `andi`. Como se describe en la teoría, este operador puede mantener bits si es que están establecidos en ambos operandos, o apagarlos si es que se encuentra en este estado en alguno de los dos números. Por lo tanto, como no podemos modificar el valor en la palabra, aplicando la operación de AND binario con un valor inmediato solo podemos apagar o mantener bits ya prendidos en `numero`.

Para asegurarnos que los bits indicados siempre estén en 1 en el resultado, haremos uso de una instrucción adicional `ori`, la cual aplica el OR binario entre un número en un registro y un valor inmediato. Esta operación mantiene un bit apagado si así se encuentra en **ambos** operandos, o permite encenderlo si el bit se encuentra en 1 en **alguno** de los dos números.

Como tenemos control sobre el valor inmediato, tras haber hecho la operación AND necesaria para mantener los bits pedidos por el ejercicio, luego usamos la instrucción `ori` con la constante `0xfffe`. Este valor que usamos como valor inmediato, tiene todos los 16 bits menos significativos prendidos, a excepción del bit 0 (que debía de permanecer igual que en `numero` para el resultado), asegurándonos que dichos bits siempre estén en 1: 11111111 11111110.

```
1      .data
2  numero:  .word 0x3ff41
3            .space 4
4            .text
5  main:    lw $t0, numero($0)
6           andi $t1, $t0, 0xffff0001
```



```

7      ori $t1, $t1, 0xffff
8      sw $t1, numero+4($0)

```

El valor resultante de la ejecución del programa es **0x0003ffff**. Comparándolo con el valor original, cumple con la tarea solicitada.

```

numero    →  0x0003ff41  =  00000000 00000011 11111111 01000001
numero+4  →  0x0003ffff  =  00000000 00000011 11111111 11111111

```

Figura 1.24: Representaciones binarias de la palabra original y el resultado tras ejecutar el programa modificado

– Operaciones de desplazamiento –

3.10, 3.11 y 3.14

Vemos que a pesar de haber realizado una operación de desplazamiento de bits a la derecha, si bien los bits menos significativos correspondientes fueron descartados, los 4 bits más significativos fueron rellenados con el valor 1. En concreto, estos últimos bits fueron rellenados acorde al bit de signo (el más significativo), el cual está prendido para el caso del valor al que apunta **numero** en nuestro programa. El valor almacenado en **\$t1** tras ejecutar el programa es **0xffffffff4**.

Para entender mejor el motivo de esto, intentaremos ahora reemplazar la instrucción **sra** por **srl**. Ahora, al correr el programa, el valor guardado en el registro **\$t1** es **0x0ffffff4**. Es decir, con este cambio, los 4 bits más significativos **no** fueron rellenados con el bit de signo, y quedaron en 0.

Lo que está sucediendo se encuentra en la diferencia entre las dos instrucciones utilizadas. Por un lado, **sra** lleva a cabo del desplazamiento *aritmético* de un número hacia la derecha, una cantidad de posiciones indicadas como argumento. Mientras, **srl** ejecuta el desplazamiento *lógico* de un número, una cantidad de posiciones dada.

Con esto queremos decir que desde el punto de vista lógico, si desplazamos un número n bits a la derecha, sabemos que los n bits menos significativos serán descartados, y que el resto de bits se desplazarán de forma acorde, quedando las posiciones de los anteriores n bits más significativos en 0, pues podemos decir que todo número tiene “infinitos ceros” a su izquierda.

En cambio, si queremos desplazar un número una cantidad de posiciones desde el punto de vista aritmético, lo más probable es que sea de interés mantener el signo del número original. Como shifteamos bits en un sistema de representación binario, correrlos n posiciones a la derecha es equivalente a dividir el número por 2^n y redondear el resultado para abajo (operación floor).

Considerando que al desplazar **numero** 4 posiciones a la derecha lo quisiéramos dividir por $2^4 = 16$, podemos terminar de cerrar la idea comparando las representaciones decimales de los números obtenidos con las instrucciones **sra** y **srl** (Fig. 1.25).

```

numero    =  0xffffffff41  =  -191
$t1 (sra) =  0xffffffff4   =  -12 =  $\lfloor \frac{-191}{16} \rfloor$ 
$t1 (srl) =  0x0ffffff4    =  268435444

```

Figura 1.25: Diferencia en los resultados obtenidos entre desplazar **numero** con **sra** y **srl**

3.12 y 3.13

Para desplazar el contenido de `numero` 2 bits a la izquierda, reemplazamos la instrucción `sra` por `sll` y cambiamos el argumento de la cantidad de posiciones que se shiftea:

```
1      .data
2  numero:  .word 0xffffffff41
3      .text
4  main:    lw $t0, numero($0)
5           sll $t1, $t0, 2
```

Si comparamos la representación decimal del número original con el del resultado obtenido (Fig. 1.26), podemos ver que acabamos de multiplicar el número por 2^n , siendo $n = 2$ la cantidad de posiciones que desplazamos del número, o sea, $2^2 = 4$.

$$\begin{array}{rclcl} \text{numero} & = & \text{0xffffffff41} & = & -191 \\ \$t1 \text{ (sll)} & = & \text{0xfffffd04} & = & -764 = -191 \cdot 2^2 \end{array}$$

Figura 1.26: Diferencia entre el número original y el resultado obtenido usando `sll`

Notar que usamos la instrucción `sll`, finalizada en *l*, indicando que es una operación del tipo *lógica*, y no *aritmética* como distinguimos en la Cuestión 3.11. Esto se debe a que **no existe** una instrucción tal como “`sla`”, pues simplemente **no es necesaria**.

Recordemos que el inconveniente que surgía de usar el desplazamiento de bits lógico hacia la derecha, era que no se preservaba el bit de signo. En cambio, al realizar shifts hacia la izquierda, el signo se preserva. Si bien los bits más significativos en las n posiciones de mayor peso son descartados (desplazando n posiciones), los bits en las posiciones subsiguientes ocuparán su lugar manteniendo el signo según sea acorde.

Por supuesto, habrá un punto en el que, por ejemplo, si tenemos un número negativo, al desplazar bits exista la posibilidad de que quede un 0 en la posición del bit de signo, según cómo sea la representación binaria del número tratado. Pero ya para este punto, o incluso para varios shifts antes, el valor obtenido habría superado el límite representable por un número entero signado de 32 bits, requiriendo otro tipo de medidas para poder almacenar el resultado.

Capítulo 2

Problemas

– Apartado 1 –

1.

```
1      .data
2  dato:  .byte 3           # inicializo una posición de memoria a 3
3      .text
4      .globl main         # debe ser global
5  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.

```
1      .data 0x10000000
2  A:    .space 80
3  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.

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

4.

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

5.

```

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

```

6.

```

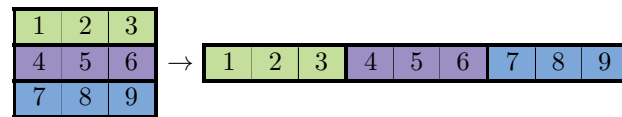
1      .data
2 por_filas:   .byte 1, 2, 3, 4, 5, 6, 7, 8, 9
3 por_columnas: .byte 1, 4, 7, 2, 5, 8, 3, 6, 9

```

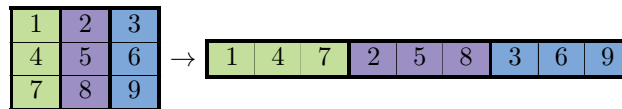
En ambos casos uso vectores de bytes, ya que los números de la matriz son lo suficientemente pequeños para ser almacenados de esta manera.

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

— Apartado 2 —

7.

```

1      .data 0x10000000
2 V:    .word 10, 20, 25, 500, 3
3
4      .text
5 main: lw $s0, V($0)
6      lw $s1, V+4($0)
7      lw $s2, V+8($0)

```

```

8         lw $s3, V+12($0)
9         lw $s4, V+16($0)

```

Cabe mencionar que como los números del vector son menores al máximo número representable con una media palabra, podría definirse un vector de valores del tipo `.half` en vez de `.word`, acomodando las operaciones y desfases utilizados en el programa:

```

1         .data 0x10000000
2 V:      .half 10, 20, 25, 500, 3
3
4         .text
5 main:   lh $s0, V($0)
6         lh $s1, V+2($0)
7         lh $s2, V+4($0)
8         lh $s3, V+6($0)
9         lh $s4, V+8($0)

```

8.

Podemos extender el programa del ejercicio anterior y utilizar la instrucción `sw` para cargar los valores que ya tenemos dentro de los registros `$s0-$s4`, en las palabras empezando desde la dirección `0x10010000`.

Además, podemos intentar optimizar la cantidad de instrucciones reales que usa nuestro programa, lo cual podría ayudar a la velocidad de nuestro programa. Por supuesto que con un programa tan pequeño, más aún corriendo en un simulador, no tiene mucho sentido optimizar la velocidad de ejecución del código. Sin embargo, resulta interesante explorar las distintas opciones y usos de las instrucciones ya vistas, para cumplir el objetivo del ejercicio.

```

1         .data 0x10000000
2 V:      .word 10, 20, 25, 500, 3
3
4         .text
5 main:   la $t0, V
6
7         lw $s0, 0($t0)
8         lw $s1, 4($t0)
9         lw $s2, 8($t0)
10        lw $s3, 12($t0)
11        lw $s4, 16($t0)
12
13        li $t1, 0x10010000    # cargar dirección inicial como referencia
14
15        sw $s0, 0($t1)
16        sw $s1, 4($t1)
17        sw $s2, 8($t1)
18        sw $s3, 12($t1)
19        sw $s4, 16($t1)

```

La idea fundamental de las optimizaciones realizadas, es evitar usar el uso de etiquetas o valores inmediatos muy grandes como parámetro a las instrucciones `lw` o `sw`, ahorrando las instrucciones

adicionales que requiere cargar valores que superen la longitud de media palabra.

Para lograr esto, indicamos los offsets en el campo del valor inmediato (ya que son números chicos), y usamos el campo del registro que es sumando para determinar la dirección de memoria desde la cual leer/escribir, con un registro temporal que contenga como valor la dirección de memoria inicial con la cual estamos copiando. Es decir, si queremos leer valores desde las direcciones V , $V+2$, ..., cargamos la dirección de V una sola vez a un registro temporal, y luego usamos ese registro como parámetro para la instrucción `lw`, mientras los valores inmediatos que se indican en las instrucciones son 1, 2, ...

9.

Una forma de resolver el problema planteado es cargando la palabra dada en un registro, y luego ir cargando en orden inverso, byte por byte en el espacio de la nueva palabra.

Para esto utilizaremos la instrucción `lw` para cargar la palabra en `$t1`, y luego la instrucción `sb` para cargar los bytes en otro espacio de memoria. Como ésta última carga el byte menos significativo del registro, también usaremos la instrucción `srl` que shiftea los bits hacia la derecha una cantidad de posiciones (concretamente 8 bits = 1 byte), descartando los bytes menos significativos una vez cargados en la nueva palabra, y corriendo de posición los más significativos a posiciones de menor peso para que podamos usar `sb`.

```
1      .data
2  palabra: .word 0x10203040
3  palabra2: .space 4
4
5      .text
6  main:   la $t0, palabra2
7
8          lw $t1, palabra($0)
9          sb $t1, 3($t0)
10         srl $t1, $t1, 8
11         sb $t1, 2($t0)
12         srl $t1, $t1, 8
13         sb $t1, 1($t0)
14         srl $t1, $t1, 8
15         sb $t1, 0($t0)
```

Como una alternativa, también podemos optar por una solución más rudimentaria, que aproveche la facilidad de cargar bytes desde una posición de memoria dada a un registro, y que luego los copie a otra posición de memoria en la palabra donde se almacenen los bytes en orden inverso. En este caso, podríamos prescindir del uso de la instrucción `srl`, usando solamente instrucciones vistas hasta este punto en el trabajo.

```
1      .data
2  palabra: .word 0x10203040
3  palabra2: .space 4
4
5      .text
6  main:   la $t0, palabra2
7          la $t1, palabra
```

```

8
9         lbu $t2, 0($t1)
10        sb $t2, 3($t0)
11        lbu $t2, 1($t1)
12        sb $t2, 2($t0)
13        lbu $t2, 2($t1)
14        sb $t2, 1($t0)
15        lbu $t2, 3($t1)
16        sb $t2, 0($t0)

```

10.

```

1         .data
2 palabra: .word 0x10203040
3
4         .text
5 main:    la $t0, palabra
6
7         lh $t1, 0($t0)
8         lh $t2, 2($t0)
9         sh $t1, 2($t0)
10        sh $t2, 0($t0)

```

11.

Considerando que los 4 bytes se encuentran a partir de la dirección de memoria 0x10010002, podemos resolver el problema con dos pares de operaciones de lectura y escritura de medias palabras usando `lh` y `sh` respectivamente:

- Copiar desde [0x10010002, 0x10010003] a [0x10010010, 0x10010011].
- Copiar desde [0x10010004, 0x10010005] a [0x10010012, 0x10010013].

Además, cargamos las direcciones de memoria del vector almacenado y el espacio de memoria reservados con la instrucción `la` en dos registros, para reducir la cantidad de operaciones según lo describimos en el Problema 8.

```

1         .data 0x10010002
2 bytes:   .byte 0x10, 0x20, 0x30, 0x40
3         .align 4
4 palabra: .space 4
5
6         .text
7 main:    la $t0, bytes
8         la $t1, palabra
9
10        lh $t2, 0($t0)
11        sh $t2, 0($t1)
12        lh $t2, 2($t0)
13        sh $t2, 2($t1)

```

12.

```
1          .data 0x10000000
2 ops:     .word 10, 20
3 sum:     .space 4
4
5          .text
6 main:
7          lw $t0, ops($0)
8          lw $t1, ops+4($0)
9          add $t0, $t0, $t1
10         sw $t0, sum($0)
```

13.

La opción más sencilla para resolver el ejercicio sería realizar la misma serie de pasos para cada número del vector:

- Cargar el valor del vector en un registro.
- Realizar la división con el número 5 en un registro donde fue precargado.
- Leer el resultado del registro especial `mflo`.
- Escribir el resultado en memoria.

```
1          .data 0x10000000
2 nums:    .word 18, -12, 15
3
4          .text
5 main:
6          la $t0, nums          # dirección vector
7          li $t1, 0x10010000    # dirección resultados
8          li $t3, 5             # divisor
9
10         lw $t2, ($t0)
11         div $t2, $t3
12         mflo $t2
13         sw $t2, ($t1)
14
15         lw $t2, 4($t0)
16         div $t2, $t3
17         mflo $t2
18         sw $t2, 4($t1)
19
20         lw $t2, 8($t0)
21         div $t2, $t3
22         mflo $t2
23         sw $t2, 8($t1)
```

También podemos explorar nuevos horizontes y usar branching consultando con la documentación:


```

1      .data 0x10000000
2  nums: .word 18, -12, 15
3
4      .text
5  main:
6      la $t0, nums          # dirección vector
7      li $t1, 0x10010000    # dirección resultados
8      li $t3, 5             # divisor
9
10     fun: lw $t2, ($t0)
11          div $t2, $t3
12          mflo $t2
13          sw $t2, ($t1)
14          addi $t0, $t0, 4
15          addi $t1, $t1, 4
16          ble $t0, 0x10000008, fun

```

Como siempre, probablemente existan otras formas más prácticas o más eficientes de realizar la tarea. Por ejemplo, seguramente se podría hacer uso de algún contador, en vez de comparar que la dirección de memoria de donde leemos el vector no pase de la posición del último número.

14.

Hacemos uso de la instrucción `andi` con la constante `0xfffffd77` que tiene todos los bits menos el 3, 7 y 9, establecidos en 1, manteniendo así el resto de bits del entero original, y apagando los exceptuados:

```

1      .data 0x10000000
2  int: .word 0xabcd12bd
3
4      .text
5  main:
6      # Asumiendo que el bit menos signif. es el 0°
7      lw $t0, int($0)
8      andi $t0, $t0, 0xfffffd77
9      sw $t0, int($0)

```

Tras ejecutar el programa, el entero en la memoria cambia al valor **0xabcd1035**. Comparando su representación binaria con la del número original, podemos ver cómo cumple con la consigna solicitada (Fig. 2.2).

0xabcd12bd	=	10101011	11001101	00010010	10111101
0xabcd1035	=	10101011	11001101	00010000	00110101
				9	7 3

Figura 2.2: Diferencia entre el número original y el modificado tras ejecutar el programa anterior usando la instrucción `andi`

15.

Interpretando que “cambiar” el valor de los bits del entero implica invertir su estado original, podemos usar la instrucción `xori`, que hace el XOR binario con un valor inmediato dado. Utilizaremos la constante `0x288` para esta operación, ya que tiene los bits 3, 7 y 9 encendidos, de forma que puedan cambiarse en el número original.

```
1      .data 0x10000000
2  int:  .word 0xff0f1235
3
4      .text
5  main:
6      # Asumiendo que el bit menos signif. es el 0°
7      lw $t0, int($0)
8      xori $t0, 0x288
9      sw $t0, int($0)
```

Después de correr el programa, el valor de la palabra es `0xff0f10bd`, el cual podemos comprobar que cumple con la consigna, comparando su representación binaria con la del número original (Fig. 2.3).

<code>0xff0f1235</code>	=	11111111	00001111	00010010	00110101
<code>0xff0f10bd</code>	=	11111111	00001111	00010000	10111101
				9	7 3

Figura 2.3: Diferencia entre el número original y el modificado tras ejecutar el programa anterior usando la instrucción `xori`

16.

Sabiendo que el número por el cual debemos multiplicar el dato almacenado en memoria, es una potencia de 2, podemos utilizar lo visto en la Cuestión 3.13 y usar la instrucción `sll` shifteándolo 5 posiciones a la izquierda (pues $32 = 2^5$).

```
1      .data 0x10000000
2  num:  .word 0x1237
3
4      .text
5  main:
6      lw $t0, num($0)
7      sll $t0, $t0, 5
8      sw $t0, num($0)
```

Comparando las representaciones decimales del número original y el resultado obtenido, podemos confirmar que el programa anterior resuelve el problema propuesto (Fig. 2.4).

<code>0x00001237</code>	=	4663
<code>0x000246e0</code>	=	$149216 = 4663 \cdot 2^5$

Figura 2.4: Diferencia entre el número original y el resultado obtenido usando `sll`