Sintaxis del lenguaje ensamblador del MIPS R2000

El *lenguaje ensamblador* es la representación simbólica de la codificación binaria de un computador, *el lenguaje máquina*. El lenguaje máquina está constituido por *instrucciones máquina* que indican al computador lo que tiene que hacer, es decir, son las órdenes que el computador es capaz de comprender. Cada instrucción máquina está constituida por un conjunto ordenado de unos y ceros, organizados en diferentes campos. Cada uno de estos campos contiene información que se complementa para indicar al procesador la acción a realizar.

El lenguaje ensamblador ofrece una representación más próxima al programador y simplifica la lectura y escritura de los programas. Las principales herramientas que proporciona la programación en ensamblador son: la utilización de nombres simbólicos para operaciones y posiciones de memoria, y unos determinados recursos de programación que permiten aumentar la claridad (legibilidad) de los programas. Entre estos recursos cabe destacar: la utilización de directivas, pseudoinstrucciones y macros que permiten al programador ampliar el lenguaje ensamblador básico definiendo nuevas operaciones.

Una herramienta denominada *programa ensamblador* traduce un programa fuente escrito de forma simbólica, es decir, en lenguaje ensamblador, a instrucciones máquina. El programa ensamblador lee un fichero fuente escrito en lenguaje ensamblador y genera un fichero objeto. Este fichero contiene instrucciones en lenguaje máquina e información que ayudará a combinar varios ficheros objeto para crear un fichero ejecutable (puede ser interpretado por el procesador).

Otra herramienta denominada *linker* , combina el conjunto de ficheros objeto para crear un único fichero que puede ser ejecutado por el computador.

En el simulador utilizado, para cargar un programa en memoria se utiliza la opción *load*, que lleva a cabo todos los pasos necesarios, desde la traducción del programa ensamblador a lenguaje máquina hasta la carga del mismo en memoria. En caso de encontrar errores sintácticos de ensamblado aparece un mensaje de error en la ventana de mensajes y, a partir de esa instrucción, no se traduce ni se carga en memoria ninguna otra.

Es necesario introducir algunos conceptos básicos. Los primeros que se verán están referidos a los recursos de programación que permite utilizar la programación en ensamblador y que facilitan la programación:

Comentarios. Estos son muy importantes en los lenguajes de bajo nivel ya que ayudan a seguir el desarrollo del programa y, por tanto, se usan con profusión. Comienzan con un carácter de almohadilla "#" y desde este carácter hasta el final de la línea es ignorado por el ensamblador.

Identificadores. Son secuencias de caracteres alfanuméricos, guiones bajos (_) y puntos (.), que no comienzan con un número. Los códigos de operación son palabras reservadas que no pueden ser utilizadas como identificadores.

Etiquetas. Son identificadores que se sitúan al principio de una línea y seguidos de dos puntos. Sirven para hacer referencia a la posición o dirección de memoria del elemento definido en ésta. A lo largo del programa se puede hacer referencia a ellas en los modos de direccionamiento de las instrucciones.

Pseudoinstrucciones. Son instrucciones que no tienen traducción directa al lenguaje máquina que entiende el procesador, pero el ensamblador las interpreta y las convierte en una o más instrucciones máquina reales. Permiten una programación más clara y comprensible. A lo largo del desarrollo de las prácticas se irán introduciendo diferentes pseudoinstrucciones que permite utilizar este ensamblador.

Directivas. Tampoco son instrucciones que tienen traducción directa al lenguaje máquina que entiende el procesador, pero el ensamblador las interpreta y le informan a éste de cómo tiene que traducir el programa. Son identificadores reservados, que el ensamblador reconoce y que van precedidos por un punto. A lo largo del desarrollo de las prácticas se irán introduciendo las distintas directivas que permite utilizar este ensamblador.

Por otro lado, los números se escriben, por defecto, en base 10. Si van precedidos de `0x`, se interpretan en hexadecimal. Las cadenas de caracteres se encierran entre comillas dobles (“). Los caracteres especiales en las cadenas siguen la convención del lenguaje de programación C:

Salto de línea: \n

Tabulador: \t

Comilla: \”

El primer paso para el desarrollo de un programa en ensamblador es definir los datos en memoria. La programación en ensamblador permite utilizar directivas que facilitan reservar espacio de memoria para datos e inicializarlos a un valor. En este capítulo se trata el uso de estas directivas.

En primer lugar recordar que, aunque la unidad base de direccionamiento es el *byte*, las memorias de estos computadores tienen un ancho de 4 bytes o 32 bits, que se llamará palabra o *word*, el mismo ancho que el del bus de datos. Así pues, cualquier acceso a una palabra de memoria supondrá leer cuatro bytes (el byte con la dirección especificada y los tres almacenados en las siguientes posiciones). Las direcciones de palabra deben estar alineadas en posiciones múltiplos de cuatro. Otra posible unidad de acceso a memoria es transferir media palabra (*half-word*).

Declaración de palabras en memoria

En este apartado se verá el uso de las directivas `.data` y `.word`. Para ello, en el directorio de trabajo se crea un fichero con la extensión '.s' y con el siguiente contenido:

```
          .data          # comienza zona de datos
palabra1: .word 15       # decimal
palabra2: .word 0x15     # hexadecimal
```

Descripción:

Este programa en ensamblador incluye diversos elementos que se describen a continuación: la directiva `.data [dir]` indica que los elementos que se definen a continuación se almacenarán en la zona de datos y, al no aparecer ninguna dirección como argumento de dicha directiva, la dirección de almacenamiento será la que hay por defecto (`0x10010000`). Las dos sentencias que aparecen a continuación reservan dos números enteros, de tamaño *word*, en dos direcciones de memoria, una a continuación de la otra, con los contenidos especificados.

Carga en el simulador el código provisto

.

Cuestión 1.1: Encuentra los datos almacenados en memoria en el programa anterior. Localiza dichos datos en el panel de datos e indica su valor en hexadecimal.

Cuestión 1.2: ¿En qué direcciones se han almacenado dichos datos? ¿Por qué?

Cuestión 1.3: ¿Qué valores toman las etiquetas `palabra1` y `palabra2`?

Crea otro fichero o modifica el anterior con el siguiente código:

```
          .data 0x10010000   # comienza zona de datos
palabras: .word     15,0x15  # decimal/hexadecimal
```

Borra los valores de la memoria mediante el botón *clear* y carga el fichero.

Cuestión 1.4: Comprueba si hay diferencias respecto al programa anterior.

Cuestión 1.5: Crea un fichero con un código que defina un vector de cinco palabras (word), que esté asociado a la etiqueta `vector`, que comience en la dirección 0x10000000 y con los valores 0x10, 30, 0x34, 0x20 y 60. Comprueba que se almacena de forma correcta en memoria.

Cuestión 1.6: ¿Qué ocurre si se quiere que el vector comience en la dirección 0x10000002? ¿En qué dirección comienza realmente? ¿Por qué?

Declaración de bytes en memoria

La directiva `.byte valor` inicializa una posición de memoria, de tamaño byte, con el contenido `valor`.

Crea un fichero con el siguiente código:

```
          .data          # comienza zona de datos
octeto:   .byte     0x10 # hexadecimal
```

Borra los valores de la memoria mediante el botón *clear* y carga el fichero.

Cuestión 1.7: ¿Qué dirección de memoria se ha inicializado con el contenido especificado?

Cuestión 1.8: ¿Qué valor se almacena en la palabra que contiene el byte?

Crea otro fichero o modifica el anterior con el siguiente código:

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

Borra los valores de la memoria y carga el fichero.

Cuestión 1.9: ¿Cuáles son los valores almacenados en memoria?

Cuestión 1.10: ¿Qué tipo de alineamiento y organización de los datos (Big-endian o Little-endian) utiliza el simulador? ¿Por qué?

Cuestión 1.11: ¿Qué valores toman las etiquetas `palabra1` y `palabra2`?

Declaración de cadenas de caracteres

La directiva `.ascii "tira"` permite cargar en posiciones de memoria consecutivas, cada una de tamaño byte, el código ASCII de cada uno de los caracteres que componen `"tira"`.

Crea un fichero con el siguiente código:

```
          .data
cadena:   .ascii    "abcde"   # defino string
octeto:   .byte     0xff
```

Borra los valores de la memoria y carga el fichero.

Cuestión 1.12: Localiza la cadena anterior en memoria.

Cuestión 1.13: ¿Qué ocurre si en vez de `.ascii` se emplea la directiva `.asciiz`? Describe lo que hace esta última directiva.

Cuestión 1.14: Crea otro fichero cargando la misma tira de caracteres a la que apunta la etiqueta `cadena` en memoria, pero ahora utilizando la directiva `.byte`.

Reserva de espacio en memoria

La directiva `.space n` sirve para reservar espacio para una variable en memoria, inicializándola a valor 0.

Crea un fichero con el siguiente código:

```
          .data
palabra1: .word     0x20
espacio:  .space    8    # reservo espacio
palabra2: .word     0x30
```

Borra los valores de la memoria y carga el fichero.

Cuestión 1.15: ¿Qué rango de posiciones se han reservado en memoria para la variable `espacio`?

Cuestión 1.16: ¿Cuántos bytes se han reservado en total? ¿Y cuántas palabras?

Alineación de datos en memoria

La directiva `.align n` alinea el siguiente dato a una dirección múltiplo de 2^n.

Crea un fichero con el siguiente código:

```
          .data
byte1:    .byte 0x10
espacio:  .space 4
byte2:    .byte 0x20
palabra:  .word 10
```

Cuestión 1.17: ¿Qué rango de posiciones se han reservado en memoria para la variable `espacio`?

Cuestión 1.18: ¿Estos cuatro bytes podrían constituir los bytes de una palabra? ¿Por qué?

Cuestión 1.19: ¿A partir de que dirección se ha inicializado `byte1`? ¿y `byte2`?

Cuestión 1.20: ¿A partir de que dirección se ha inicializado `palabra`? ¿Por qué?

Crea un fichero con el siguiente código:

```
          .data
 byte1:   .byte 0x10
          .align 2
 espacio: .space 4
 byte2:   .byte 0x20
 palabra: .word 10
```

Cuestión 1.21: ¿Qué rango de posiciones se ha reservado en memoria para la variable `espacio`?

Cuestión 1.22:¿Estos cuatro bytes podrían constituir los bytes de una palabra? ¿Por qué? ¿Qué ha hecho la directiva `.align`?

Problemas propuestos para entregar

1. Dado el siguiente ejemplo de programa ensamblador:

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

   Indica las etiquetas, directivas y comentarios que aparecen en el mismo.

2. Diseña un programa ensamblador que reserva espacio para dos vectores A y B de 20 palabras cada uno a partir de la dirección `0x10000000`.

3. Diseña un programa ensamblador que realiza la siguiente reserva de espacio en memoria a partir de la dirección 0x10001000: una palabra, un byte y otra palabra alineada en una dirección múltiplo de 4.
4. Diseña un programa ensamblador que realice la siguiente reserva de espacio e inicialización en memoria a partir de la dirección por defecto: 3 (palabra), 0x10 (byte), reserve 4 bytes a partir de una dirección múltiplo de 4, y 20 (byte).
5. Diseña un programa ensamblador que defina, en el espacio de datos, la siguiente cadena de caracteres: “Esto es un problema” utilizando
   a) .ascii
   b) .byte
   c) .word
6. Sabiendo que un entero se almacena en un word, diseña un programa ensamblador que defina en la memoria de datos la matriz A de enteros definida como

$$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

   a partir de la dirección `0x10010000` suponiendo que:
   a) La matriz A se almacena por filas.
   b) La matriz A se almacena por columnas.

Ahora abordaremos la carga y almacenamiento de datos entre memoria y los registros del procesador. Dado que la arquitectura del R2000 es RISC, utiliza un subconjunto concreto de instrucciones que permiten las acciones de carga y almacenamiento de los datos entre los registros del procesador y la memoria. Generalmente, las instrucciones de carga de un dato de memoria a registro comienzan con la letra "l" (de *load* en inglés) y las de almacenamiento de registro en memoria con "s" (de *store* en inglés), seguidos por la letra inicial correspondiente al tamaño del dato que se va a mover, *b* para byte, *h* para media palabra (*half word*) y *w* para palabra (*word*).

Carga de datos inmediatos (constantes)

Crea un fichero con el siguiente código:

```
          .text     #zona de instrucciones
main:     lui $s0, 0x8690
```

Descripción:

La directiva `.text` sirve para indicar el comienzo de la zona de memoria dedicada a las instrucciones. Por defecto esta zona comienza en la dirección `0x00400000` y en ella se pueden ver las instrucciones que ha introducido el simulador para ejecutar, de forma adecuada, nuestro programa. La primera instrucción se referencia con la etiqueta `main` y le indica al simulador dónde está el principio del programa que debe ejecutar. Por defecto hace referencia a la dirección `0x00400020`. A partir de esta dirección, el simulador cargará el código de nuestro programa en el segmento de memoria de instrucciones.

La instrucción `lui` es la única instrucción de carga inmediata real, y almacena la media palabra que indica el dato inmediato de 16 bits en la parte alta del registro especificado, en este caso `$s0`. La parte baja del registro, correspondiente a los bits de menor peso de éste, se pone a 0.

Borra los valores de la memoria del simulador y carga el fichero anterior.

Cuestión 2.1: Localiza la instrucción en memoria de instrucciones e indica:

La dirección donde se encuentra.

El tamaño que ocupa.

La instrucción máquina, analizando cada campo de ésta e indicando qué tipo de formato tiene.

Ejecuta el programa mediante el botón *run* del simulador.

Cuestión 2.2: Comprueba el efecto de la ejecución del programa en el registro.

El ensamblador del MIPS ofrece la posibilidad de cargar una constante de 32 bits en un registro utilizando una pseudoinstrucción. Ésta es la pseudoinstrucción `li`.

Crea un fichero con el siguiente código:

```
          .text     #zona de instrucciones
main:     li $s0,0x12345678
```

Borra los valores de la memoria y carga este fichero.

Ejecuta el programa mediante el botón *run* del simulador.

Cuestión 2.3: Comprueba el efecto de la ejecución del programa en el registro.

Cuestión 2.4: Comprueba qué conjunto de instrucciones reales implementan esta pseudoinstrucción.

Carga de palabras (palabras de memoria a registro)

Crea un fichero con el siguiente código:

```
          .data
palabra:  .word 0x10203040
          .text               #zona de instrucciones
main:     lw $s0,palabra($0)
```

Descripción:

La instrucción `lw` carga la palabra contenida en una posición de memoria, cuya dirección se especifica en la instrucción, en un registro. Dicha posición de memoria se obtiene sumando el contenido del registro (en este caso `$0`, que siempre vale cero) y el identificador `palabra`.

Borra los valores de la memoria del simulador y carga el fichero anterior.

Cuestión 2.5: Localiza la instrucción en memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Cuestión 2.6: Explica cómo se obtiene a partir de esas instrucciones la dirección de `palabra`. ¿Por qué crees que el simulador traduce de esta forma la instrucción original?

Cuestión 2.7: Analiza cada uno de los campos que componen estas instrucciones e indica el tipo de formato que tienen.

Ejecuta el programa mediante el botón *run*.

Cuestión 2.8: Comprueba el efecto de la ejecución del programa.

Cuestión 2.9: ¿Qué pseudoinstrucción permite cargar la dirección de un dato en un registro? Modifica el programa original para que utilice esta pseudoinstrucción, de forma que el programa haga la misma tarea. Comprueba qué conjunto de instrucciones sustituyen a la pseudoinstrucción utilizada una vez el programa se carga en la memoria del simulador.

Cuestión 2.10: Modifica el código para que en lugar de transferir la palabra contenida en la dirección de memoria referenciada por la etiqueta palabra, se transfiera la palabra que está contenida en la dirección referenciada por `palabra+1`. Explica qué ocurre y por qué.

Cuestión 2.11: Modifica el programa anterior para que guarde en el registro `$s0` los dos bytes de mayor peso de `palabra`. Nota: Utiliza la instrucción `lh` que permite cargar medias palabras (16 bits) desde memoria a un registro (en los 16 bits de menor peso del mismo).

Carga de bytes (bytes de memoria a registro)

Crea un fichero con el siguiente código:

```
            .data
octeto:     .byte 0xf3
siguiente:  .byte 0x20

            .text    #zona de instrucciones
main:       lb $s0, octeto($0)
```

Descripción:

La instrucción `lb` carga el byte de una dirección de memoria en un registro. Al igual que antes, la dirección del dato se obtiene sumando el contenido del registro `$0` (siempre vale cero) y el identificador `octeto`.

Borra los valores de la memoria y carga el fichero.

Cuestión 2.12: Localiza la instrucción en memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Ejecuta el programa.

Cuestión 2.13: Comprueba el efecto de la ejecución del programa.

Cuestión 2.14: Cambia en el programa la instrucción `lb` por `lbu`. ¿Qué sucede al ejecutar el programa? ¿Qué significa esto?

Cuestión 2.15: Si `octeto` se define como:

"`octeto: .byte 0x30`", ¿existe diferencia entre el uso de la instrucción `lb` y `lbu`?¿ Por qué?

Cuestión 2.16: ¿Cuál es el valor del registro `$s0` si `octeto` se define como:

"`octeto: .word 0x10203040`"? ¿Por qué?

Cuestión 2.17: ¿Cuál es el valor del registro `$s0` si se cambia en `main` la instrucción existente por la siguiente:

```
main:    lb $s0,octeto+1($0)?
```

¿Por qué? ¿Por qué en este caso no se produce un error de ejecución (excepción de error de direccionamiento)?

Almacenado de palabras (palabras de registro a memoria)

Crea un fichero con el siguiente código:

```
            .data
palabra1: .word 0x10203040
palabra2: .space 4
palabra3: .word 0xffffffff

            .text    #zona de instrucciones
main:    lw $s0, palabra1($0)
         sw $s0, palabra2($0)
         sw $s0, palabra3($0)
```

Descripción:

La instrucción `sw` almacena la palabra contenida en un registro en una dirección de memoria. Esta dirección se obtiene sumando el contenido de un registro más un desplazamiento especificado en la instrucción (identificador).

Borra los valores de la memoria del simulador y carga el fichero.

Cuestión 2.18: Localiza la primera instrucción de este tipo en la memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Ejecuta el programa.

Cuestión 2.19: Comprueba el efecto de la ejecución del programa.

Almacenado de bytes (bytes de registro a memoria)

Crea un fichero con el siguiente código:

```
                .data
palabra:  .word 0x10203040
octeto:         .space 2

                .text     #zona de instrucciones
main:           lw $s0, palabra($0)
                sb $s0, octeto($0)
```

Descripción:

La instrucción "`sb`" almacena el byte de menor peso de un registro en una dirección de memoria. La dirección se obtiene sumando el desplazamiento indicado por el identificador y el contenido de un registro.

Borra los valores de la memoria y carga el fichero.

Cuestión 2.20: Localiza la instrucción en memoria de instrucciones e indica cómo ha transformado dicha instrucción el simulador.

Ejecuta el programa.

Cuestión 2.21: Comprueba el efecto de la ejecución del programa.

Cuestión 2.22: Modifica el programa para que el byte se almacene en la dirección "`octeto+1`". Comprueba y describe el resultado de este cambio.

Cuestión 2.23: Modifica el programa anterior para transferir a la dirección de `octeto` el contenido de la posición `palabra+3`.

Problemas propuestos para entregar

7. Diseña un programa ensamblador que defina el vector de enteros V=(10, 20, 25, 500, 3) en la memoria de datos a partir de la dirección 0x10000000 y cargue todos sus componentes en los registros $s0 - $s4.

8. Diseña un programa ensamblador que copie el vector definido en el problema anterior a partir de la dirección 0x10010000.

9. Diseña un programa ensamblador que, dada la palabra 0x10203040 almacenada en una posición de memoria, la reorganice en otra posición, invirtiendo el orden de sus bytes.

10. Diseña un programa ensamblador que, dada la palabra 0x10203040 definida en memoria la reorganice en la misma posición, intercambiando el orden se sus medias palabras. Nota: utiliza la instrucción lh y sh.

11. Diseña un programa en ensamblador que inicialice cuatro bytes a partir de la posición 0x10010002 a los siguientes valores 0x10, 0x20, 0x30, 0x40, y reserve espacio para una palabra a partir de la dirección 0x1001010. El programa transferirá los cuatro bytes contenidos a partir de la posición 0x10010002 a la dirección 0x1001010.

Se presentan las instrucciones que permiten realizar operaciones aritméticas, lógicas y de desplazamiento de datos. Las instrucciones aritméticas están constituidas por las operaciones de suma y resta (add, addu, addi, addiu, sub, subu) y las operaciones de multiplicación y división (mult, multu, div, divu). La diferencia entre las instrucciones acabadas o no acabadas en u (por ejemplo entre add y addu) está en que las primeras provocan una excepción de desbordamiento si el resultado de la operación no es representable en 32 bits y las segundas no tienen en cuenta el desbordamiento. Recordar que el rango de representación de los números enteros con signo de 32 bits en complemento a 2 va desde –2.147.483.648 a 2.147.483.647 (0x80000000 a 0x7fffffff).

Dentro del grupo de instrucciones que permiten realizar operaciones lógicas están: suma lógica (or y ori), producto lógico (`and` y `andi`) y la or exclusiva (xor y xori).

Finalmente, se presentan las instrucciones de desplazamiento aritmético y lógico (sra, sll, srl).

Operaciones aritméticas con datos inmediatos (constantes)

Crea un fichero con el siguiente código:

```
        .data           #zona de datos
     #Máx. Positivo representable 0x7FFFFFFF
numero: .word 2147483647
        .text       #zona de instrucciones
main:   lw  $t0,numero($0)
        addiu  $t1,$t0,1
```

Descripción:

La instrucción “addiu” es una instrucción de suma con un dato inmediato y sin detección de desbordamiento.

Borra los valores de la memoria, carga el fichero y ejecútalo paso a paso.

Cuestión 3.1: Localiza el resultado de la suma efectuada. Comprueba el resultado.

Cuestión 3.2: Cambia la instrucción “addiu” por la instrucción “addi”.

Borra los valores de la memoria, carga el fichero y ejecútalo paso a paso.

Cuestión 3.3: ¿Qué ha ocurrido al efectuar el cambio? ¿Por qué?

Operaciones aritméticas con datos en memoria

Crea un fichero con el siguiente código:

```
            .data
numero1:  .word 0x80000000 #max. Negativo represent.
numero2:  .word 1
numero3:  .word 1
          .text
main:
          lw   $t0,numero1($0)
          lw   $t1,numero2($0)
          subu $t0,$t0,$t1
          lw   t1,numero3($0)
          subu $t0,$t0,$t1
          sw   $t0,numero1($0)
```

Borra los valores de la memoria, carga el fichero y ejecútalo paso a paso.

Cuestión 3.4: ¿Qué hace el programa anterior? ¿Qué resultado se almacena en `numero3`? ¿Es correcto?

Cuestión 3.5: ¿Se producirá algún cambio si las instrucciones “subu” son sustituidas por instrucciones “sub”? ¿Por qué?

Multiplicación y división con datos en memoria

Crea un fichero con el siguiente código:

```
          .data
numero1:  .word 0x7FFFFFFF #Máx. Positivo represent.
numero2:  .word    16
          .space   8
```

```
          .text
main:     lw $t0,numero1($0)
       lw $t1,numero2($0)
       mult $t0,$t1# multiplica los dos números
       mfhi $t0
       mflo $t1
       sw $t0,numero2+4($0) #32 bits más peso
       sw $t1,numero2+8($0) #32 bits menos peso
```

Descripción:

El código realiza la multiplicación de dos números, almacenando el resultado de la multiplicación a continuación de los dos multiplicandos.

Borra los valores de la memoria, carga el fichero y ejecútalo.

Cuestion 3.6: ¿Qué resultado se obtiene después de realizar la operación? ¿Por qué se almacena en dos palabras de memoria?

Cuestión 3.7: Modifica los datos anteriores para que `numero1` y `numero2` sean 10 y 3 respectivamente. Escribe el código que divida `numero1` entre `numero2` (dividendo y divisor respectivamente) y coloque el cociente y el resto a continuación de dichos números.

Operaciones lógicas

Crea un fichero con el siguiente código:

```
          .data
numero:   .word    0x3ff41
          .space   4
          .text
main:     lw $t0,numero($0)
          andi $t1,$t0,0xfffe
          # 0xffe en binario es 0...0111111111111110
          sw $t1,numero+4($0)
```

Descripción:

El código anterior pone los 16 bits más significativos y el bit 0 de `numero` a 0, y almacena el resultado en la siguiente palabra (`numero+4`). Esto se consigue utilizando la instrucción “andi”, que realiza el producto lógico, bit a bit, entre el dato contenido en un registro y un dato inmediato. El resultado obtenido es que aquellos

bits que en el dato inmediato están a 1 no se modifican con respecto al contenido original del registro, es decir, los 16 bits de menor peso excepto el bit 0. Por otro lado, todos aquellos bits que en el dato inmediato están a 0, en el resultado también están a 0, es decir, los 16 bits de mayor peso y el bit 0. Los 16 bits de mayor peso del resultado se ponen a 0 puesto que, aunque el dato inmediato que se almacena en la instrucción es de 16 bits, a la hora de realizar la operación el procesador trabaja con un dato de 32 bits, poniendo los 16 de mayor peso a 0, fijando por tanto los bits del resultado también a 0.

Borra los valores de la memoria, carga el fichero y ejecútalo. Comprueba el resultado.

Cuestión 3.8: Modifica el código para obtener, en la siguiente palabra, que los 16 bits más significativos de `numero` permanezcan tal cual, y que los 16 bits menos significativos queden a cero, excepto el bit cero que también debe quedar como estaba.

Cuestión 3.9: Modifica el código para obtener, en la siguiente palabra, que los 16 bits más significativos de `numero` permanezcan tal cual, y que los 16 bits menos significativos queden a uno, excepto el bit cero que debe quedar como estaba.

Operaciones de desplazamiento

Crea un fichero con el siguiente código:

```
          .data
numero:   .word     0xffffff41
          .text
main:     lw $t0,numero($0)
          sra $t1,$t0,4
```

Descripción:

El código anterior desplaza el valor de `numero` cuatro bits a la derecha, rellenando el hueco generado a su izquierda con el bit del signo.

Borra los valores de la memoria, carga el fichero y ejecútalo.

Cuestión 3.10: ¿Para qué se ha rellenado `numero` con el bit del signo?

Cuestión 3.11: ¿Qué ocurre si se sustituye la instrucción “sra” por “srl”?

Cuestión 3.12: Modifica el código para desplazar el contenido de `numero` 2 bits a la izquierda.

Cuestión 3.13: ¿Qué operación aritmética acabamos de realizar?

Cuestión 3.14: Volviendo al código que iniciaba este apartado, indica qué operación aritmética tenia como consecuencia la ejecución de dicho código.

Problemas propuestos para entregar

12. Diseña un programa ensamblador que defina el vector de enteros de dos elementos V=(10,20) en la memoria de datos a partir de la dirección 0x10000000 y almacene su suma a partir de la dirección donde acaba el vector.

13. Diseña un programa ensamblador que divida los enteros 18,-1215 almacenados a partir de la dirección 0x10000000 entre el número 5 y que a partir de la dirección 0x10010000 almacene el cociente de dichas divisiones.

14. Pon a cero los bits 3,7,9 del entero 0xabcd12bd almacenado en memoria a partir de la dirección 0x10000000, sin modificar el resto.

15. Cambia el valor de los bits 3,7,9 del entero 0xff0f1235 almacenado en memoria a partir de la dirección 0x10000000, sin modificar el resto.

16. Multiplica el número 0x1237, almacenado en memoria a partir de la dirección 0x10000000, por 32 (2^5) sin utilizar las instrucciones de multiplicación ni las pseudoinstrucciones de multiplicación.