ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Sesión de laboratorio número 5

ARITMÉTICA ENTERA: MULTIPLICACIÓN Y DIVISIÓN

Introducción

En esta práctica se trabaja con la aritmética entera del MIPS R2000. En particular, esta sesión de laboratorio se centra en el trabajo con las operaciones de multiplicación y división de números enteros. Para ello se plantea el diseño de un conjunto de subrutinas que manejan variables que representan la hora de un reloj. La herramienta de trabajo es el simulador del procesador MIPS R2000 denominado PCSpim.

Objetivos

- Entender el manejo de las operaciones de multiplicación y división de enteros.
- Medir el coste de ejecución de un programa que trabaja con operaciones de multiplicación y división de enteros.

Material

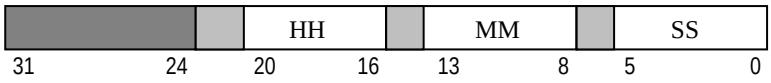
El material se puede obtener de la carpeta de recursos de PoliformaT.

- Simulador PCSpim del MIPS R2000.
- Archivo fuente: `reloj.s`

Codificación de un formato horario y su inicialización

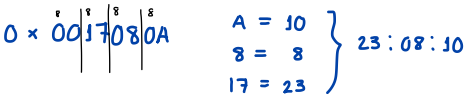
En esta práctica vamos a trabajar con una variable que representa el estado de un reloj. Supondremos que el valor del reloj se expresa como una tripleta HH:MM:SS (horas, minutos y segundos). El rango de representación horaria abarca desde 00:00:00 hasta 23:59:59. A modo de ejemplo, el reloj puede contener valores horarios tales como 13:25:43, 00:10:53, 07:59:32, etcétera. En particular, el campo de horas puede contener los valores 0…23, el de minutos 0…59 y el de segundos 0…59. En consecuencia, valores como 43:90:21 o 128:40:298 no son válidos.

Para manejar este tipo de variables en el computador se utiliza una única palabra de memoria de 32 bits atendiendo a la distribución de campos que muestra la figura siguiente. Como se aprecia en la figura, cada elemento de la tripleta se codifica en un único byte y está situado en los bits de menor peso de este byte.



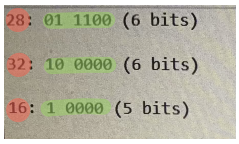
Ahora bien, considerando los valores horarios válidos, no todos los bits de los 32 que tiene la palabra se utilizan en la codificación del estado del reloj. Los bits no utilizados han sido señalados gráficamente con un tono de gris, y su valor, en principio, no está definido. En particular, el byte de mayor peso (bits 24…31) no se utiliza. El campo HH necesita 5 bits (2^5=32) ya que puede contener 24 valores distintos. Los campos MM y SS son de 6 bits (2^6=64) porque pueden contener 60 valores distintos.

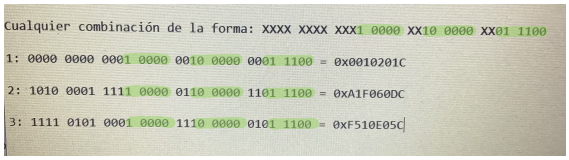
► ¿Qué valor del reloj representa la palabra de bits **`0x0017080A`**?



► ¿Qué valor del reloj representa la palabra de bits **`0xF397C84A`**?

► Indique tres codificaciones distintas de la variable reloj para el valor horario 16:32:28.



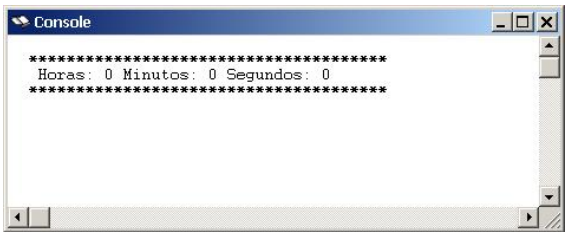


Considere como punto de partida el programa en ensamblador contenido en el fichero `reloj.s`. A continuación referimos los elementos más significativos de la declaración de variables y del programa principal.

```
                ############################################################
                # Segmento de datos
                ############################################################

                .data 0x10000000
reloj:          .word 0                 # HH:MM:SS (3 bytes de menor peso)

                ############################################################
                # Segmento de código
                ############################################################

                .globl __start
                .text 0x00400000

__start         la $a0, reloj
                jal imprime_reloj

salir:          li $v0, 10              # Código de exit (10)
                syscall                 # Última instrucción ejecutada
```

El programa dispone en memoria la variable **`reloj`** para almacenar una palabra de acuerdo con el formato horario que hemos descrito. La subrutina **`imprime_reloj`** imprime en pantalla el valor contenido en la variable horaria que se le pasa por referencia a través del registro **`$a0`**. El programa acaba ejecutando la llamada al sistema **`exit`**.

► Cargue el fichero `reloj.s` y ejecútelo en el simulador. Tal y como está, el resultado mostrado en la consola debe ser el siguiente:



► ¿Por qué se ha impreso la hora 00:00:00?

Porque la variable que almacena el valor de la hora ($a0) está inicializada a 0

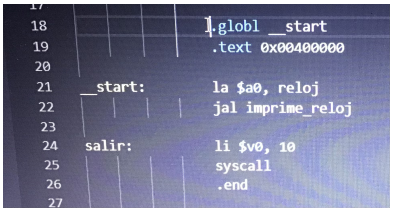
Se quieren diseñar dos subrutinas que inicialicen la variable **`reloj`**. En primer lugar diseñaremos una subrutina para inicializar todos los campos del reloj con un valor concreto de horas, minutos y segundos. La siguiente tabla especifica su funcionamiento.

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
`inicializa_reloj`	**`$a0`**: dirección del reloj **`$a1`**: HH:MM:SS	reloj = HH:MM:SS

Por ejemplo, para inicializar el reloj con la hora 02:03:12 y ver el resultado en pantalla habrá que ejecutar el código:

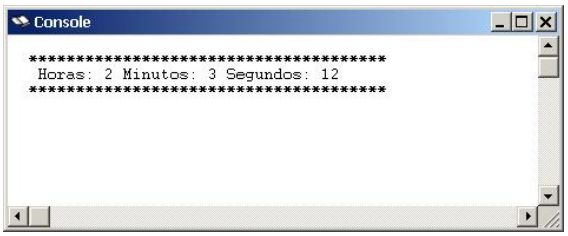
```
la $a0, reloj
li $a1, 0x0002030C
jal inicializa_reloj

la $a0, reloj
jal imprime_reloj
```

Lo he implementado en el código inicial



Código sin modificar

El resultado de esta ejecución debe ser:



► Implemente la subrutina **`inicializa_reloj`**.

Ahora se quiere diseñar una subrutina alternativa de inicialización del reloj con la siguiente especificación:

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
`inicializa_reloj_alt`	**`$a0`**: dirección del reloj **`$a1`**: HH **`$a2`**: MM **`$a3`**: SS	reloj = HH:MM:SS

Aunque el resultado de la subrutina es idéntico al de la subrutina **`inicializa_reloj`** que ya hemos diseñado, su funcionamiento es diferente. Ahora los tres campos del reloj se especifican como argumentos en tres registros distintos. Además, se quiere que en el diseño de esta nueva subrutina intervenga solamente una única instrucción de acceso a memoria. Esto significa que el código ha de construir la palabra de 32 bits que represente el valor HH:MM:SS y escribirla en memoria haciendo uso de una única instrucción. Esta construcción se puede hacer utilizando instrucciones lógicas y de desplazamiento.

► Implemente la subrutina `inicializa_reloj_alt`.

Como ejercicio para casa se propone diseñar un conjunto de subrutinas para inicializar por separado cada uno de los campos de una variable reloj, según se especifica en la siguiente tabla:

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
`inicializa_reloj_hh`	**`$a0`**: dirección del reloj **`$a1`**: HH	reloj.hh = HH
`inicializa_reloj_mm`	**`$a0`**: dirección del reloj **`$a1`**: MM	reloj.mm = MM
`inicializa_reloj_ss`	**`$a0`**: dirección del reloj **`$a1`**: SS	reloj.ss = SS

Por ejemplo, para inicializar el campo MM del reloj con el valor 59 se hará:

```
la $a0, reloj
li $a1, 0x3B
jal inicializa_reloj_mm
```

► Implemente el código de las subrutinas **`inicializa_reloj_hh`**, **`inicializa_reloj_mm`** e **`inicializa_reloj_ss`**.

► En principio, un único valor de reloj HH:MM:SS puede codificarse de diferentes maneras según los valores que asignemos a los bits que no entran a formar parte de la codificación de los campos HH, MM y SS. Ahora queremos obligar a que todas las horas se representen de una única manera haciendo que los bits del reloj que no están definidos sean siempre cero. Por ejemplo, la hora 02:03:12 solamente se codifica como `0x0002030C`, mientras que otras combinaciones como `0x6502030C`, `0x89E203CC` o `0xFFC2038C` no están permitidas. ¿Cómo será ahora la subrutina **`inicializa_reloj`** para cumplir con esta condición?

► La siguiente subrutina opera sobre una variable reloj cuya dirección se pasa como argumento en el registro `$a0` y con un valor X que se pasa en el byte menos significativo de `$a1`. Explique razonadamente qué efecto produce la ejecución de la subrutina.

```
subrutina:      lw $t0, 0($a0)      Guarda en $t0 el valor de los segundos (el valor de tiempo dado)
                li $t1, 0x00FFFF00  Carga en ese registro ese valor
                and $t0, $t0, $t1   Guarda en $t0 0x00000000
                or $t1, $t0, $a1    Guarda en $t1el valor de $a1
                sw $t1, 0($a0)      Escribe en el byte menos significativo de $a0 el valor de $t1
                jr $ra
```

Un mismo valor de tiempo puede codificarse de diferentes formas, pero nuestro programa solo lee aquella en la que todos los bits que no forman parte de la expresión son 0. Esta subrutina se encarga de, sea cual sea la expresión de tiempo, poner dichos bits a 0.

La multiplicación y la división de enteros y su coste temporal

Las operaciones de multiplicación y división de enteros se implementan en la arquitectura del MIPS R2000 mediante dos instrucciones máquina para enteros con signo, `mult` y `div`, respectivamente, y `multu` y `divu`, para números sin signo. Estas instrucciones dejan el resultado en una pareja especial de registros denominados `hi` y `lo`.

La interpretación del contenido de estos registros particulares `hi` y `lo` depende de la instrucción que se ejecute. Después de ejecutar una instrucción de multiplicación, `hi` y `lo` contienen la parte alta y baja, respectivamente, del resultado. En este caso se considera que ha habido desbordamiento en la multiplicación cuando el resultado necesita más de 32 bits para ser representado, es decir, cuando `hi` es distinto de cero para `multu` o es distinto del signo replicado 32 veces para `mult`; la detección de este posible desbordamiento es responsabilidad del programador. Por otro lado, en una operación de división, `hi` contiene el resto y `lo` el cociente. En la arquitectura del MIPS R2000 la división por cero es una operación indefinida. Por lo tanto, es responsabilidad del programador comprobar que el divisor es diferente de cero antes de ejecutar una instrucción de división.

Para poder operar con los valores contenidos en estos dos registros especiales hace falta trasladarlos previamente al banco de registros de la unidad aritmético-lógica mediante las instrucciones de movimiento `mfhi` (*move from hi*) y `mflo` (*move from lo*).

Veamos un ejemplo sencillo que ilustra todo lo que se acaba de exponer:

```
li $t0, 18        # $t0 = 18
li $t1, 4         # $t1 = 4
mult $t0, $t1     # lo = 18*4 =0x00000048 y hi = 0x00000000
mflo $s0          # $s0 = lo
div $t0, $t1      # lo = 18÷4 = 4 y hi = 18%4 = 2
mfhi $s1          # $s1 = hi
mflo $s2          # $s2 = lo
```

El resultado de la multiplicación (18×4=72) cabe en el registro `lo`, por lo que solamente se ha movido su contenido a `$s0`. En el caso de la división, el cociente (18÷4=4) se ha almacenado en `lo` y el resto (18%4=2) en `hi`; han hecho falta sendas instrucciones de movimiento para llevar estos dos valores a los registros `$s1` y `$s2`, respectivamente.

Una cuestión muy importante a tener en cuenta es la complejidad temporal de las operaciones de multiplicación y división de enteros. Así como las operaciones lógicas, de desplazamiento o aritméticas básicas como la suma o la resta se pueden ejecutar en un solo ciclo de reloj del

procesador, no ocurre así con la multiplicación y la división. De estas dos últimas operaciones, la segunda es generalmente la que más tarda. El coste temporal exacto, sin embargo, depende de la implementación del procesador y, en algunas ocasiones concretas, del tamaño de los operandos. Para hacernos una idea, una operación de división puede tardar entre 35 y 80 ciclos de reloj, mientras que una operación de multiplicación tarda entre 5 y 32 ciclos de reloj. Si suponemos que la multiplicación tarda 20 ciclos y la división 70, podemos estimar el tiempo de ejecución del código anterior en 1+1+20+1+70+1+1=95 ciclos de reloj. Nótese que en este cálculo se ha tenido en cuenta que, dado que los valores de las constantes pueden codificarse con 16 bits, las pseudoinstrucciones `li` del código se pueden traducir por una única instrucción máquina de tipo inmediato (por ejemplo, `li $t0, 18` se puede traducir en `ori $t0, $zero, 18`).

La operación de multiplicación: conversión de HH:MM:SS a segundos

Se quiere diseñar una subrutina que convierte en segundos el valor de una variable reloj codificada mediante la tripleta con la forma HH:MM:SS. Por ejemplo, la hora 18:32:45 equivale a 66765 segundos; para el cálculo basta con hacer 18×3600+32×60+45=66765.

La subrutina se llama `devuelve_reloj_en_s`. Su especificación es la siguiente:

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
`devuelve_reloj_en_s`	**`$a0`**: dirección del reloj	**`$v0`**: segundos

El programa proporcionado inicialmente en el fichero `reloj.s` incluye la subrutina `imprime_s` para imprimir segundos (se pasan como argumento en `$a0`). Por ejemplo, el código siguiente calcula en segundos la hora 18:32:45 e imprime el resultado:

```
la $a0, reloj
li $a1, 0x0012202D
jal inicializa_reloj
la $a0, reloj
jal devuelve_reloj_en_s
move $a0, $v0
jal imprime_s
```

El resultado mostrado por pantalla es:

► Para leer de memoria por separado cada uno de los campos del reloj (HH, MM y SS) se puede usar una instrucción de lectura de byte. Razone si hay que utilizar `lb` (*load byte*) o `lbu` (*load byte unsigned*).

► Implemente la subrutina `devuelve_reloj_en_s`.

► ¿Qué tipo de instrucciones de suma han de utilizarse en la subrutina, `add` o `addu`?

Lo más correcto sería utilizar addu para realizar sumas sin signo, pero puede utilizarse cualquiera de las dos

► ¿Cuántas instrucciones de multiplicación se ejecutan en la subrutina `devuelve_reloj_en_s`?

Dos, una para pasar de horas a segundos y otra para pasar de minutos a segundos

► ¿Cuántas instrucciones de movimiento de información entre los registros del banco de enteros y los registros `hi` y `lo` se ejecutan en la subrutina diseñada?

Se ejecutan un total de dos instrucciones mflo

► Se quiere completar la subrutina `devuelve_reloj_en_s` con la detección de desbordamiento de la multiplicación. Dado que estamos trabajando con números positivos, hay que incluir las instrucciones necesarias para detectar si, después de llevar a cabo la operación de multiplicación, el resultado ocupa más de 32 bits. En este último caso, hay que saltar a la etiqueta **`salir`** para terminar la ejecución del programa. Indique las instrucciones necesarias para esta detección de desbordamiento.

Si usamos 'multu' harían falta las instrucciones 'mfhi', y 'beqz $HI, salir' (almacenamos lo que hay en HI en un registro y si es = a 0, se salta a 'salir'

► Supongamos que todas las instrucciones tardan un ciclo de reloj en ejecutarse y las de multiplicación tardan 20 ciclos. ¿Cuánto tiempo tarda en ejecutarse el código de la subrutina de conversión a segundos?

1+1+20+1+1+1+20+1+1+1+1+1 = 50 ciclos

La operación de división: conversión de segundos a HH:MM:SS

En este apartado planteamos el problema inverso considerado en la sección anterior: ahora se quiere inicializar el valor del reloj partiendo previamente de una cantidad determinada de segundos. Así, queremos diseñar una subrutina que, dado un número de segundos, inicialice el reloj con el valor correspondiente pero expresado en formato HH:MM:SS. Por ejemplo, un tiempo de 66765 segundos equivale al valor 18:32:45, es decir, 18 horas, 32 minutos y 45 segundos. Para pasar de segundos a HH:MM:SS hace falta dividir dos veces por la constante 60, según se ilustra en este caso concreto:

- 66765 segundos ÷ 60 = 1112 minutos (el resto son 45 segundos)
- 1112 minutos ÷ 60 = 18 horas (el resto son 32 minutos)

Así, SS corresponde al resto de la primera división, MM al resto de la segunda división y HH al cociente de la segunda división.

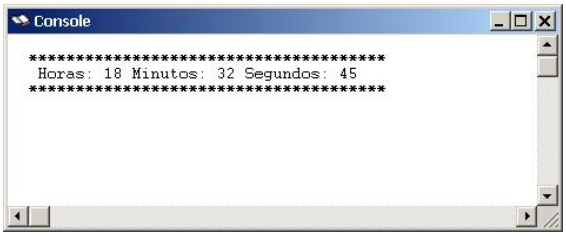
La subrutina que implementa esta inicialización se llamará `inicializa_reloj_en_s`. Su especificación es la siguiente:

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
`inicializa_reloj_en_s`	**`$a0`**: dirección del reloj **`$a1`**: segundos	reloj = HH:MM:SS

Por ejemplo, el código siguiente inicializa el reloj con la hora correspondiente a 66765 segundos e imprime el resultado:

```
la $a0, reloj
li $a1, 66765
jal inicializa_reloj_en_s

la $a0, reloj
jal imprime_reloj
```

El resultado mostrado por pantalla es:



► Implemente el código de la subrutina `inicializa_reloj_en_s`.

► ¿Cuántas instrucciones de división se ejecutan en la subrutina `inicializa_reloj_en_s`?

En total se ejecutan dos instrucciones div

► ¿Cuántas instrucciones de movimiento de información entre los registros del banco de enteros y los registros `hi` y `lo` se ejecutan en la subrutina diseñada?

En total 2 mfhi y 2 mflo

► Supongamos que todas las instrucciones tardan un ciclo de reloj en ejecutarse mientras que las de división tardan 70 ciclos. ¿Cuál será el tiempo que tarda la ejecución del código de la subrutina `inicializa_reloj_en_s`?

1+70+1+1+1+70+1+1+1+1+1= 149

► Se quieren evitar posibles errores en la operación de división haciendo que, antes de que se pueda producir una división por cero, la subrutina lo detecte y salte a la etiqueta **`salir`** para terminar la ejecución del programa. Indique las instrucciones necesarias para llevar a cabo esta detección de la división por cero.

Después del primer mfhi, (por ejemplo mfhi $t1) haría falta una instrucción del tipo 'beqz' (por ejemplo beqz, $t1, salir)