



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**? 23:08:10

► ¿Qué valor del reloj representa la palabra de bits **0xF397C84A**? 23:08:10

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

0x00102038 // 0x11102038 // 0x22102038

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
#####

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

#####
# Segmento de código
#####

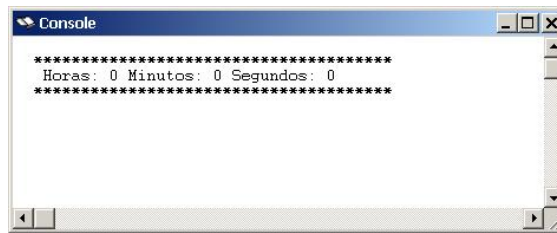
          .globl __start
          .text 0x00400000

__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 el valor de `a0` por defecto es `0x00000000`, y no se modifica

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.

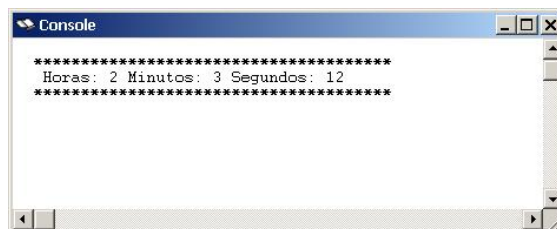
NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<b>inicializa_reloj</b>	<b>\$a0:</b> dirección del reloj <b>\$a1:</b> HH:MM:SS	reloj = HH:MM:SS

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
```

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
<b>inicializa_reloj_alt</b>	<b>\$a0:</b> dirección del reloj <b>\$a1:</b> HH <b>\$a2:</b> MM <b>\$a3:</b> 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
<b>inicializa_reloj_hh</b>	<b>\$a0:</b> dirección del reloj <b>\$a1:</b> HH	reloj.hh = HH
<b>inicializa_reloj_mm</b>	<b>\$a0:</b> dirección del reloj <b>\$a1:</b> MM	reloj.mm = MM
<b>inicializa_reloj_ss</b>	<b>\$a0:</b> dirección del reloj <b>\$a1:</b> 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?

Usando una máscara para quitar todos los bits que no queremos

► 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)
              li $t1, 0x00FFFF00
              and $t0, $t0, $t1
              or $t1, $t0, $a1
              sw $t1, 0($a0)
              jr $ra
```

## 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 ( $18 \times 4 = 72$ ) cabe en el registro `lo`, por lo que solamente se ha movido su contenido a `$s0`. En el caso de la división, el cociente ( $18 \div 4 = 4$ ) 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  $18 \times 3600 + 32 \times 60 + 45 = 66765$ .

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

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<code>devuelve_reloj_en_s</code>	<code>\$a0</code> : dirección del reloj	<code>\$v0</code> : 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*).

`lbu` para evitar excepciones y desbordamientos por el signo

► Implemente la subrutina `devuelve_reloj_en_s`.

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

addu, por la misma razón que lbu

► ¿Cuántas instrucciones de multiplicación se ejecutan en la subrutina devuelve\_reloj\_en\_s?

se usan 2 operaciones de mult (multu, multiplicación sin signo)

► ¿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?

Siempre se usará una instrucción de movimiento cada vez que se multiplique, así que se usan dos operaciones

► 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.

```
mfhi $t2
bnez $t2, 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?

$10 + 20 * 2 = 50$  ciclos  
instr      mult

## 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:

- $66765 \text{ segundos} \div 60 = 1112 \text{ minutos}$  (el resto son 45 segundos)
- $1112 \text{ minutos} \div 60 = 18 \text{ horas}$  (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:

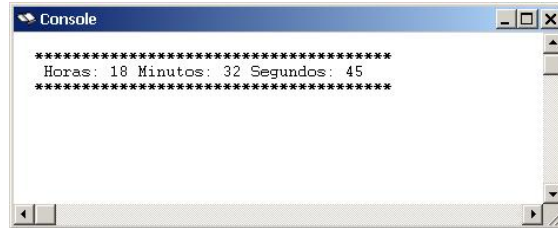
NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<code>inicializa_reloj_en_s</code>	<b>\$a0</b> : dirección del reloj <b>\$a1</b> : segundos	reloj = HH:MM:SS

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`?

2 operaciones de 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?

2 operaciones de movimiento

► 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`?

$9 + 70 * 2 = 149$  ciclos

► 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.

`beqz $t0, salir`