

ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Sesión de laboratorio número 6

ARITMÉTICA ENTERA: SUMAS, RESTAS Y DESPLAZAMIENTOS

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 suma, resta y desplazamiento 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.
- Sustituir instrucciones de multiplicación de enteros por un conjunto de sumas, restas y desplazamientos.
- Cuantificar la mejora del tiempo de ejecución debido a la sustitución de las instrucciones de multiplicación de enteros por un conjunto equivalente de instrucciones de suma, resta y desplazamiento.
- Comprender la operación de suma para variables temporales y tratar el acarreo.

Material

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

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

El formato horario y su inicialización

En esta práctica vamos a trabajar de nuevo con la 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) en una única palabra de memoria de 32 bits atendiendo a la distribución de campos que muestra la figura siguiente.



En la sesión de laboratorio anterior se ha trabajado con este formato horario y se han implementado algunas subrutinas que trabajan con él. En esta sesión vamos a necesitar alguna de estas subrutinas; en particular, la que inicializa una variable reloj y también la que convierte en segundos un valor

horario expresado por la tripleta HH:MM:SS. La tabla siguiente muestra el funcionamiento de estas dos subrutinas.

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<code>inicializa_reloj</code>	<code>\$a0</code> : dirección del reloj <code>\$a1</code> : HH:MM:SS	reloj = HH:MM:SS
<code>devuelve_reloj_en_s</code>	<code>\$a0</code> : dirección del reloj	<code>\$v0</code> : segundos

Considere como punto de partida el programa en ensamblador contenido en el fichero `reloj.s`, al que el alumno le debe haber añadido, al menos, las dos subrutinas anteriores implementadas en la sesión anterior del laboratorio. 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`.**

Multiplicación mediante sumas y desplazamientos

Dado que **la ejecución de una instrucción de multiplicación suele ser muy costosa** en número de ciclos de reloj, en las ocasiones en que alguno de los operandos lo permite, **los compiladores pueden optar por reemplazarlas por un conjunto de sumas, restas y desplazamientos** que, en términos globales, suponen un tiempo de ejecución inferior. Recordemos que una instrucción de multiplicación puede tardar, según la implementación del procesador, entre 5 y 32 ciclos de reloj. Las operaciones de suma, resta y desplazamiento, sin embargo, tardan un solo ciclo en ejecutarse.

La multiplicación por un número que es potencia entera de dos puede hacerse utilizando desplazamientos hacia la izquierda. Por ejemplo, para multiplicar un número entero por 4 ($4 = 2^2$) basta con desplazarlo hacia la izquierda dos posiciones. **Por ejemplo, consideremos el producto $7 \times 2^2 = 28$.** Si codificamos el número 7 en un byte y lo expresamos en binario obtenemos 00000111; si **desplazamos los bits dos posiciones hacia la izquierda** y rellenamos los huecos generados con ceros obtenemos 00011100, byte que corresponde a la codificación binaria del número 28.

Pero también podemos basarnos en la propiedad anterior y aprovecharnos de ella para multiplicar por constantes que no son potencias enteras de dos. Por ejemplo, imaginemos que queremos multiplicar el contenido del registro \$a0 por la constante 15. En principio, podríamos utilizar directamente la instrucción de multiplicación:

```
li $t0, 15
mult $a0, $t0    # lo = $a0*15
mflo $v0         # $v0 = lo
```

Si la instrucción de multiplicación tarda 20 ciclos de reloj y el resto de instrucciones tarda un ciclo, entonces el código anterior tarda en ejecutarse un total de 1+20+1=22 ciclos de reloj.

Por otro lado, el número 15 se puede expresar como suma de potencias enteras de dos: $15 = 2^3 + 2^2 + 2^1 + 2^0$. Por tanto, el producto $\$a0 \times 15$ se transforma de forma equivalente en $\$a0 \times (2^3 + 2^2 + 2^1 + 1)$. De acuerdo con la expresión anterior, hay que hacer una serie de desplazamientos y sumas atendiendo a los unos que tenga la constante ($15 = 00001111_2$). En este caso concreto podemos transformar la operación de multiplicación por tres sumas y tres desplazamientos (el término 2^0 no genera ningún desplazamiento). Así pues, el código anterior se puede sustituir completamente por el siguiente código alternativo y de idéntico resultado:

En \$v0 se guarda el resultado de la suma

En \$t0 se guarda la potencia de 2 con la que se opera

En \$a0 se guarda el número original

```
sll $v0, $a0, 3    # $v0 = $a0*2^3
sll $t0, $a0, 2    # $t0 = $a0*2^2
addu $v0, $v0, $t0 # $v0 = $a0*(2^3 + 2^2)
sll $t0, $a0, 1    # $t0 = $a0*2^1
addu $v0, $v0, $t0 # $v0 = $a0*(2^3 + 2^2 + 2^1)
addu $v0, $v0, $a0 # $v0 = $a0*(2^3 + 2^2 + 2^1 + 2^0)
```

sll: desplazamiento lógico hacia la izquierda

Se deslaza tantas unidades como valor tiene el exponente

addu: suma sin signo

Este código alternativo de 6 instrucciones de desplazamientos y suma tarda en ejecutarse 6 ciclos de reloj, lo que supone una mejora del tiempo de ejecución respecto del primero de $20/6=3.33$ veces, esto es, el código anterior se ejecuta 3.33 veces más rápidamente que el primero.

Para utilizar esta nueva forma de llevar a cabo la multiplicación en el programa que nos ocupa de conversión del reloj en segundos tenemos que considerar los productos por las constantes 3600 y 60. Estas dos constantes se pueden expresar como potencias enteras de dos de la siguiente manera:

$$3600 = 0000\ 1110\ 0001\ 0000_2 = 2^{11} + 2^{10} + 2^9 + 2^4$$

$$60 = 0011\ 1100_2 = 2^5 + 2^4 + 2^3 + 2^2$$

Así pues, para multiplicar un valor por cualquiera de estas constantes hacen falta cuatro desplazamientos hacia la izquierda (instrucciones sll) y tres sumas (instrucciones addu). Nótese que en este caso, dado que no aparece el término 2^0 , todas las potencias de dos generan una operación de desplazamiento.

► Escriba el código necesario para multiplicar el contenido del registro \$a0 por la constante 36 y devolver el resultado en el registro \$v0 utilizando sumas y desplazamientos.

$$36 = 32 + 4 = 2^5 + 2^2$$

Haciendo uso de la idea que se acaba de exponer, vamos a diseñar una versión alternativa de la subrutina devuelve_reloj_en_s denominada devuelve_reloj_en_s_sd en la cual las

instrucciones de multiplicación se substituyen por un conjunto equivalente de instrucciones de suma y desplazamiento. El funcionamiento de la subrutina es el siguiente:

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<code>devuelve_reloj_en_s_sd</code>	<code>\$a0</code> : dirección del reloj	<code>\$v0</code> : segundos

► Implemente el código de la subrutina `devuelve_reloj_en_s_sd`.

► 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 `devuelve_reloj_en_s_sd`? ¿Cuántas veces es más rápida esta última subrutina que `devuelve_reloj_en_s`?

La subrutina `devuelve_reloj_en_s_sd` tarda en ejecutarse 20 ciclos de reloj, 30 menos que la subrutina `devuelve_reloj_en_s`. Es por tanto (50/20) 2.5 veces más rápida.

Multiplicación mediante sumas, restas y desplazamientos

La técnica de multiplicación por descomposición de una constante en suma de potencias que hemos usado en el apartado anterior se puede adaptar a la escritura del multiplicador según la codificación de Booth. En esta codificación el peso de cada dígito es idéntico al de la codificación normal, pero ahora los dígitos son tres, 0, +1 y -1.

Por ejemplo, la constante 15 se puede expresar como el byte $0000\ 1111_2$ en binario natural, y como $000+1000-1_{\text{Booth}}$ según la codificación de Booth. Dicho de otro modo, la constante 15 se puede descomponer, con la primera opción, en $2^3+2^2+2^1+2^0 = 8+4+2+1$ y con la segunda opción, en $2^4-2^0 = 16-1$. (Porque $1000-1$ se codifica a decimal como: $1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 - 1 \cdot 2^0 = 2^3 - 1 = 16 - 1$)

Se puede entender que, para este caso particular, la constante 15 tiene una descomposición en potencias de dos más corta, que ahora incluye la operación de resta, y que por tanto, puede ayudar a

simplificar la generación de código. Así, la multiplicación del registro \$a0 por 15 que analizábamos en el apartado anterior se puede implementar con el siguiente código:

```
sll $v0, $a0, 4      # $v0 = $a0*24
subu $v0, $v0, $a0   # $v0 = $a0*(24 - 20)
```

Es evidente que esta técnica solamente resulta de utilidad práctica cuando la codificación de Booth de las constantes enteras proporciona un número reducido de dígitos +1 y -1 comparado con el número de unos de la codificación en binario natural.

Volviendo de nuevo a nuestro programa de cálculo de segundos, podemos considerar la codificación de Booth de las constantes 3600 y 60 para ver si nos permite un ahorro en el número de sumas y desplazamientos. La codificación de Booth de estas dos constantes es:

$$\begin{aligned} 3600 &= 000+1 \ 00-10 \ 00+1-1 \ 0000_{\text{Booth}} = 2^{12} - 2^9 + 2^5 - 2^4 \\ 60 &= \quad \quad \quad 0+100 \ 0-100_{\text{Booth}} = 2^6 - 2^2 \end{aligned}$$

En consecuencia, vemos que solamente la constante 60 permite una reducción del número de operaciones, ya que de cuatro desplazamientos y tres sumas pasamos a dos desplazamientos y una resta. La complejidad asociada a la constante 3600 permanece igual porque solamente ha habido una sustitución de dos sumas por dos restas.

En cualquier caso recordemos que en todo momento estamos tratando de implementar la misma operación (multiplicación) pero de distinto modo (sumas, restas y desplazamientos) ayudándonos de las diferentes codificaciones de uno de los operandos de la multiplicación:

$$\begin{aligned} \$a0 * 3600 &= \$a0 * (2^{11} + 2^{10} + 2^9 + 2^4) = \$a0 * (2^{12} - 2^9 + 2^5 - 2^4) \\ \$a0 * 60 &= \$a0 * (2^5 + 2^4 + 2^3 + 2^2) = \$a0 * (2^6 - 2^2) \end{aligned}$$

► Escriba el código necesario para multiplicar el contenido del registro \$a0 por la constante 31 y devolver el resultado en el registro \$v0 utilizando sumas, restas y desplazamientos.

31 -> Binario natural -> 0001 1111 -> Codificación de Booth -> 0010 000-1 = $1*2^5 - 1*2^0 = 31$

```
sll $v0, $a0, 5
subu $v0, $v0, $a0
```

Haciendo uso de la idea que se acaba de exponer, vamos a diseñar otra versión alternativa de la subrutina devuelve_reloj_en_s denominada devuelve_reloj_en_s_srd en la cual se sustituya solamente la instrucción de multiplicación por la constante 60 por un conjunto equivalente de instrucciones de suma, resta y desplazamiento. El funcionamiento de la subrutina es el siguiente:

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
devuelve_reloj_en_s_srd	\$a0: dirección del reloj	\$v0: segundos

► Implemente el código de la subrutina devuelve_reloj_en_s_srd. Compruebe que el resultado de la ejecución es el mismo que en el caso anterior.

► 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 `devuelve_reloj_en_s_srd`? ¿Cuántas veces es más rápida esta última subrutina que `devuelve_reloj_en_s`?

Tarda en ejecutarse 16 ciclos de reloj. Es por tanto (50/16) 3,13 veces más rápida.

Incremento del reloj

En este último apartado vamos a considerar el problema de incrementar el valor del reloj. En primer lugar consideremos la subrutina `pasa_hora` que viene incluida en el fichero `reloj.s`.

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<code>pasa_hora</code>	<code>\$a0</code> : dirección del reloj	reloj = HH:MM:SS + 1 h

El código de esta subrutina se muestra a continuación:

```
pasa_hora:    lbu $t0, 2($a0)        # $t0 = HH
              addiu $t0, $t0, 1     # $t0 = HH++
              li $t1, 24
              beq $t0, $t1, H24      # Si HH==24 se pone HH a cero
              sb $t0, 2($a0)        # Escribe HH++
              j fin_pasa_hora
H24:         sb $zero, 2($a0)       # Escribe HH a 0
fin_pasa_hora: jr $ra
```

Como se puede apreciar, la subrutina comprueba si el incremento del campo HH hace que supere su valor máximo; si es así, el campo HH pasa a valer 0. Por ejemplo, si el reloj con valor 07:48:21 se incrementa en una hora pasa a valer 08:48:21. Esto se muestra en el código siguiente:

```
la $a0, reloj
li $a1, 0x00073015      # Hora 07:48:21
jal inicializa_reloj

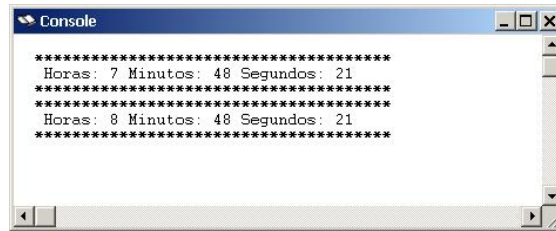
la $a0, reloj
jal imprime_reloj

la $a0, reloj
jal pasa_hora           # Incrementa el reloj en una hora

la $a0, reloj
```

```
jal imprime_reloj
```

y su resultado en pantalla:



```
*****
Horas: 7 Minutos: 48 Segundos: 21
*****
Horas: 8 Minutos: 48 Segundos: 21
*****
```

Sin embargo, considerando que el reloj funciona de forma cíclica, cuando el reloj tenga el valor 23:48:21 y se incremente en una hora entonces su nuevo valor pasa a ser 00:48:21 y no 24:48:21.

Consideremos ahora el diseño de una subrutina que incrementa la hora del reloj en un segundo. Su nombre es `pasa_segundo` y su funcionamiento se especifica en la siguiente tabla.

NOMBRE	ARGUMENTOS DE ENTRADA	SALIDA
<code>pasa_segundo</code>	<code>\$a0</code> : dirección del reloj	reloj = HH:MM:SS + 1 s

La subrutina ha de acceder al valor del reloj almacenado en memoria y, como indica su nombre, incrementarlo en un segundo. Por ejemplo, el reloj 19:22:40 pasa a valer 19:22:41. Sin embargo, hay que tener en cuenta un par de casos especiales. En primer lugar, cuando el incremento de un segundo involucra a su vez el incremento de los minutos. Por ejemplo, la hora 19:22:59 pasaría a valer 19:23:00. En segundo lugar, el incremento del campo de los minutos también pueden dar lugar al incremento del campo de las horas; éste es el caso de la hora 23:59:59, que tras el incremento de un segundo pasa a valer 00:00:00. A modo de ejemplo, la ejecución del código siguiente parte del reloj con este valor horario y lo incrementa en dos segundos:

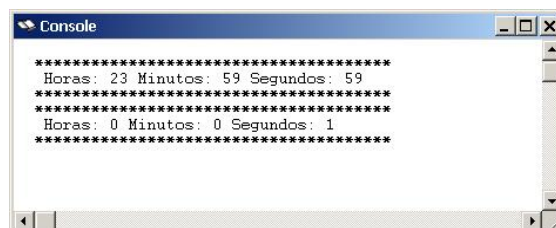
```
la $a0, reloj
li $a1, 0x00173b3b      # Hora 23:59:59
jal inicializa_reloj

la $a0, reloj
jal imprime_reloj

la $a0, reloj
jal pasa_segundo        # Incrementa el reloj en un segundo
jal pasa_segundo        # Incrementa el reloj en un segundo

la $a0, reloj
jal imprime_reloj
```

El resultado que ofrece este código debe ser el siguiente:



```
*****
Horas: 23 Minutos: 59 Segundos: 59
*****
Horas: 0 Minutos: 0 Segundos: 1
*****
```

► Implemente el código de la subrutina `pasa_segundo`.

► Escriba el código necesario para inicializar una variable reloj con el valor 21:15:45 e incrementarlo en tres horas y 40 segundos. Use las subrutinas `inicializa_reloj`, `pasa_hora` y `pasa_segundo`. ¿Cuál es el valor final del reloj?

21 = 0x15
15 = 0x0F
45 = 0x2D

} 0x00150F2D

EL VALOR FINAL DEL RELOJ ES 0:16:25