COMPUTER ORGANISATION

Lab Session 6

INTEGER ARITHMETIC:

ADDITION, SUBTRACTION, SHIFT OPERATIONS

Introduction

In this session we deal again with the integer arithmetic of the MIPS R2000 processor. In particular, this lab session focuses on the addition, subtraction and shift operations for integers. To this end, the design of a set of subroutines that handle variables representing the time of a clock is proposed. The tool used for this purpose is the simulator of the MIPS R2000 processor named PCSpim.

Goals

- To understand the multiplication and division operations for integer numbers.
- To replace integer multiplication instructions with a set of addition, subtraction and shift instructions
- To quantify the improvement in runtime achieved with this replacement.

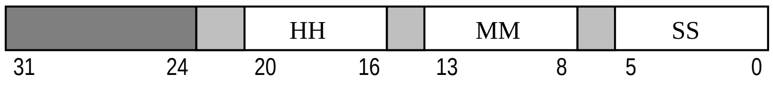
Material

The material for the session can be found in the corresponding PoliformaT folder

- MIPS R2000 Simulator: PCSpim
- Source file: reloj.s

Coding and Initialization of a time format

In this session we work again with the variable that represents the time of a clock (named *reloj*). Remember that time is expressed as a HH: MM: SS triplet (hours, minutes, and seconds). The range of time representation ranges from 00:00:00 to 23:59:59. To handle this type of variable in the computer, a single 32-bit memory word is used, with the field distribution shown in the next figure. As shown, each element of the triplet is encoded in a single byte and it is located in the lower bits of this byte.



In this session we need some of the routines we implemented in Session 5. In particular, the one that initializes a *reloj* variable and also the one that converts in seconds a time value expressed by the triplet HH: MM: SS. The following table shows the operation of these two subroutines.

NAME	INPUT PARAMETERS	OUTPUT
inicializa_reloj	**$a0**: memory address for variable reloj **$a1**: HH:MM:SS	reloj = HH:MM:SS
devuelve_reloj_en_s	**$a0: memory address for variable reloj**	**$v0**: seconds

Start from the assembler program contained in the file *reloj.s* plus the two previously mentioned routines that you already implemented in Session 5. Next you can see the most important elements in the variable declaration and main program.

```
        ################################################################
        # Data segment
        ################################################################

        .data 0x10000000
reloj:    .word 0         # HH:MM:SS (3 less significant bytes)

        ################################################################
        # Code segment
        ################################################################

        .globl __start
        .text 0x00400000

__start:   la $a0, reloj
        jal imprime_reloj

salir:     li $v0, 10        # Code for exit (10)
        syscall           # last executed instruction
```

The program uses the variable *reloj* stored in a memory word according to the time format that we have previously described. The routine *imprime_reloj* prints on the display the value contained in the *reloj* variable that is passed to it by reference through the register $a0. The program ends up executing a call to the system function *exit*.

Multiplication using addition and shift instructions

Since the execution of a multiplication instruction has usually a high runtime, sometimes when one of the operands allows it, the compilers may choose to replace this instruction with a set of additions, subtractions and shift instructions that, in global terms, they imply a lower runtime. Recall that a multiplication instruction can take between 5 and 32 clock cycles, depending on the processor architecture. The operations of addition, subtraction and shift, however, take a single cycle to be executed.

The multiplication by a number that is an integer power of two can be done by using shifts to the left. For example, to multiply an integer number by 4 ($4 = 2^2$) just shift it two positions to the left. Another example, let´s consider the product $7 \times 2^2 = 28$. If we code in binary the number 7 in a single byte we have 00000111; then shifting this number two positions to the left and inserting 0 on the right part we obtain 00011100, that is the binary code for 28.

But, we can also use this property and take advantage of it in order to multiply by constants that are not completely powers of two. For example, consider that we want to multiply the contents of the register $a0 by the constant 15. Obviously, we could directly use the multiplication instruction as follows:

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

In this case, if the multiplication instruction takes 20 clock cycles to be executed and the rest of instructions take a single cycle, then this code takes a total of 1+20+1=22 clock cycles to be executed.

On the other hand, number 15 can be expressed as the addition of several power of two: 15 = $2^3+2^2+2^1+2^0$. Then, the product $a0 × 15 can be transformed in the equivalent operation: $a0 × ($2^3+2^2+2^1+1$). Considering the previous expression, it is necessary to make a series of shifts and additions according to the number and position of digits 1 that the constant has (15 = 00001111_2). In this particular case, the multiplication can be replaced by 3 additions and 3 shift operations (2^0 do not need any shift). So, the previous code can be replaced with the next alternative code producing the same result:

```
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)
```

This alternative code has 6 instructions (addition and shift) and consequently only needs 6 clock cycles to be executed which means an improvement of the runtime with respect to the first code of 20/6 = 3.33 times, that is, the alternative code is executed 3.33 times faster than the first one.

To use this method in our problem of time conversion we have to express the constant values 3600 and 60 as their equivalent power of two:

$$\mathbf{3600 = 0000\ 1110\ 0001\ 0000_2 = 2^{11} + 2^{10} + 2^9 + 2^4}$$
$$\mathbf{60 = 0011\ 1100_2 = 2^5 + 2^4 + 2^3 + 2^2}$$

So, for multiplying any value by one of these two constants 4 shift left instructions (sll) and 3 addition instructions (addu) are needed. Note that as the value 2^0 does not appear, every power of two requires a shift operation.

► Implement the chunk of code appropriate to multiply the contents of register $a0 by the constant value 36 using only addition and shift instructions. The result must be returned in $v0.

Making use of the idea just explained, let´s design an alternative version of the routine **`devuelve_reloj_en_s`** called **`devuelve_reloj_en_s_sd`** in which the multiplication instructions are replaced by an equivalent set of addition and shift instructions. The routine specification is:

NAME	INPUT PARAMETERS	OUT PARAMETERS
devuelve_reloj_en_s_sd	**$a0**: memory address of *reloj* variable	**$v0**: seconds

► Implement the routine **`devuelve_reloj_en_s_sd`** according to this specification

► Let´s suppose that every instruction takes 1 clock cycle to be executed except the multiplication one that takes 20 cycle. Which is the runtime for the new routine? Which is the improvement with respect the routine *devuelve_reloj_en_s*?

Multiplication using addition, subtraction and shift operations.

The previous idea used for multiplications can be adapted to re-code the multiplicator according to Booth encoding. With Booth encoding the weight of each digit is equal to the usual encoding but now, we use 3 different digits: 0, +1 y –1.

For example, the constant value 15 can be written as the byte $0000\ 1111_2$ using the natural binary encoding, and also can be written as $000{+}1000{-}1_{Booth}$ according to Booth encoding. That means that the value 15 can be expressed according to the first option as $2^3+2^2+2^1+2^0$ = 8+4+2+1 and, as 2^4-2^0 = 16–1 according to Booth encoding.

Note that for this particular case the value 15 has a simpler coding that includes a subtraction operation and this permits to simplify the program code. So, the multiplication of the content of register $a0 by the constant value 15 can be implemented with the following piece of code:

```
sll $v0, $a0, 4      # $v0 = $a0*2^4
subu $v0, $v0, $a0   # $v0 = $a0*(2^4 – 2^0)
```

Obviously, this technique is only of practical utility when the Booth encoding of the integer constants provides a reduced number of digits +1 and -1 in comparison with the number of 1 of the encoding in natural binary.

Now, we can encode the constant values 3600 and 60 using Booth technique to see if it permits to save on addition and shift instructions. The Booth encodings of these values are:

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

Consequently, we see that only the constant 60 allows a reduction in the number of operations, originally we needed 4 shifts operations and 3 additions and now we only need to perform 2 shifts and 1 subtraction. The complexity associated with the constant 3600 remains the same because there has only been a substitution of 2 additions by 2 subtractions.

In any case, remember that we are always trying to implement the same operation (multiplication) but in different ways (addition, subtraction and shift to left) by using different encodings of one of the operands of multiplication:

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

► Implement the chunk of code appropriate to multiply the contents of register $a0 by the constant value 31 using only addition, subtraction and shift instructions. The result must be returned in register $v0.

Now, using Booth encoding implement a new alternative for the routine **`devuelve_reloj_en_s`** named **`devuelve_reloj_en_s_srd`** replacing only the multiplication by constant 60 by the equivalent addition subtraction and shift instructions. The specification for the routine is the following:

NAME	INPUT PARAMETERS	OUTPUT PARAMETER
devuelve_reloj_en_s_srd	**$a0**: memory address of *reloj* variable	**$v0**: seconds

► Write the code of the routine and check that the result obtained is the same than in the previous routines.

► Let´s suppose that every instructions takes 1 clock cycle to be executed except the multiplication one that takes 20 cycle. Which is the runtime for the new routine *devuelve_reloj_en_s_srd*? Which is the improvement with respect the routine *devuelve_reloj_en_s*?

Stepping up the time

Here we are going to deal with the problem of stepping up the value of the variable *reloj*. First of all, you have to select the routine **`pasa_hora`** included in the file *reloj.s*.

NAME	INPUT PARAMETER	OUTPUT PARAMETER
pasa_hora	**$a0**: *memory address of reloj variable*	reloj = HH:MM:SS + 1 h

Below you can see the routine´s code:

```
pasa_hora:       lbu $t0, 2($a0)       # $t0 = HH
                 addiu $t0, $t0, 1     # $t0 = HH++
                 li $t1, 24
                 beq $t0, $t1, H24     # If HH==24 then reset HH (HH=0)
                 sb $t0, 2($a0)        # Store HH++
                 j fin_pasa_hora
H24:             sb $zero, 2($a0)      # Store HH = 0
fin_pasa_hora:   jr $ra
```

As it can be seen, the routine checks if the increase of field HH exceeds its maximum value (24). If that happens the field HH is reset (HH=0). For instance, if time is 07:48:21 and we step up 1 hour the new time will be 08:48:21. This can be seen in the next code:

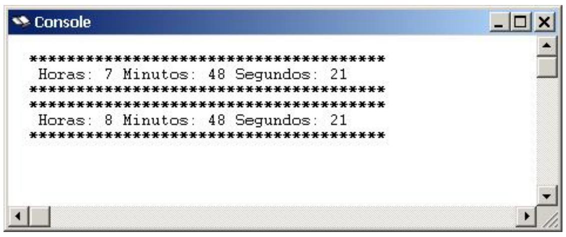
```
la $a0, reloj
li $a1, 0x00073015     # Hora 07:48:21
jal inicializa_reloj

la $a0, reloj
jal imprime_reloj

la $a0, reloj
jal pasa_hora         # increase 1 hour

la $a0, reloj
jal imprime_reloj
```

The displayed result is:



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

Considering a cyclic clock, if time is 23:48:21 and we increase 1 hour, the new time value will be: 00:48:21 and not 24:48:21.

Now let´s implement a routine to increase time in 1 second. The routine is named **pasa_segundo** and its specification is the following:

NAME	INPUT PARAMETER	OUTPUT
pasa_segundo	**$a0**: *memory address of reloj variable*	reloj = HH:MM:SS + 1 s

The routine has to read the value of *reloj* variable from main memory (time) and increase this value in 1 second. For example, if time is 19:22:40 after the execution of the routine the new time will be 19:22:41. But, you have to consider two special cases: 1) when increasing seconds affects the value of minutes; for example, time 19:22:59 will be 19:23:00. And 2) when an increase in minutes affects the value of the hour; for example, time 23:59:59 will be 00:00:00.

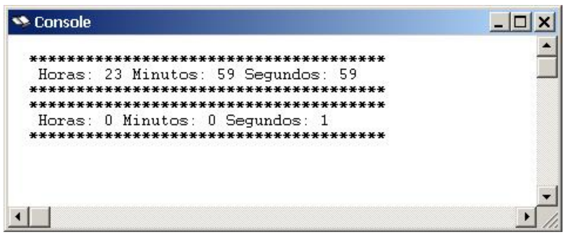
As an example, the following code starts from this time an increase it in 2 seconds:

```
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       # Increase 1 second
jal pasa_segundo       # Increase 1 second
la $a0, reloj
jal imprime_reloj
```

The displayed result will be:



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

► Implement the code for the routine **`pasa_segundo.`**

► Write the appropriate code to initialize a *reloj* variable with the value 21:15:45 then increase this time in 3 hours and 40 seconds. Make use of the routines *inicializa_reloj*, *pasa_hora* y *pasa_segundo*. Which is the time displayed?