

Lab Session 5

INTEGER ARITHMETIC: MULTIPLICATION AND DIVISION

Introduction

In this session we work with the integer arithmetic of MIPS R2000. In particular, this lab focuses on the multiplication and division of integer numbers. To do this, we propose the implementation of a set of routines that handle variables representing the time in a digital clock. We will use for this purpose the PCSpim simulator.

Goals

- To understand the integer multiplication and division operations performed in MIPS R2000.
- To measure the runtime of a program that works with integer multiplication and division operations.

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 with a variable (named *reloj*) that represents the time in a digital clock. We assume 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. As example, the clock may show time values such as 13:25:43, 00:10:53, 07:59:32, and so on. In particular, the hour field can show values from 0 to 23, the minutes and seconds fields values from 0 to 59. Consequently, values such as 43:90:21 or 128: 40: 298 are not valid.

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.



Considering the valid values to show the time, not all the 32 bits of the word are used when coding the time. Unused bits are shown in the figure with a shade of gray, and their value, by default, is not defined. In particular, the highest byte (bits 24 ... 31) is not used. The HH field needs 5 bits to

codify its value ($2^5 = 32$) since it can contain 24 different values. The MM and SS fields have 6 bits ($2^6 = 64$) because they can represent 60 different values.

► Which time represents the word 0x0017080A?

► Which time represents the word 0xF397C84A?

► Write three different encodings of the variable to show the time 16:32:28.

Consider the assembler program contained in the file **reloj.s**. The most important parts of this program are listed below:

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

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

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

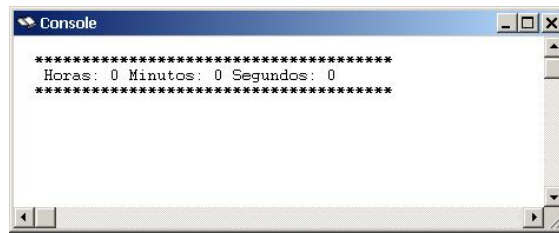
.globl __start
.text 0x00400000

__start  la $a0, reloj
        jal imprime_reloj

salir:   li $v0, 10    # exit code (10)
        syscall      # end
```

A *reloj* variable has been reserved in memory to store a word according to the time format described above. The **print_time** routine prints the value contained in the *reloj* variable. The memory address of this value is passed to the routine in register \$a0. The program ends when executes the system call *exit*.

► Load and execute in the simulator the **reloj.s** program. In its current status the result displayed on the console should be the following:



► Why the time 00:00:00 is displayed?

Now, let's go to implement two routines to initialize the *reloj* variable. The first one will set all the fields of the variable at the same time (using a single instruction) with a specific value for hours, minutes and seconds. The following table shows the specification for this routine.

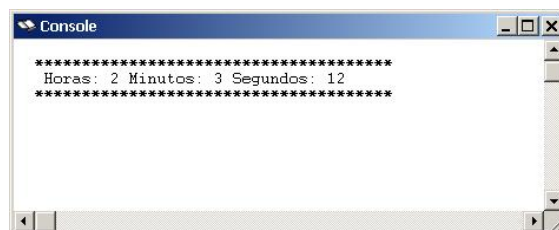
NAME	INPUT PARAMETERS	EXIT
inicializa_reloj	\$a0 : memory address for variable reloj \$a1 : HH:MM:SS	reloj = HH:MM:SS

For example, to initialize *reloj* with the time 02:03:12 and to display the result, the following code has to be executed:

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

```
la $a0, reloj
jal imprime_reloj
```

After the execution, the displayed result will be:



► Implement the routine **inicializa_reloj**.

Now, we want to design a second routine that initializes separately each of the fields of the variable *reloj*. The new specification is:

NAME	INPUT PARAMETERS	OUTPUT
inicializa_reloj_alt	\$a0 : memory address for variable reloj \$a1 : HH \$a2 : MM \$a3 : SS	reloj = HH:MM:SS

- Implement the routine `inicializa_reloj_alt`.

As homework, we propose you to implement a set of routines to separately initialize each of the fields of a *reloj* variable, as specified in the following table:

NAME	INPUT PARAMETERS	OUTPUT
<code>inicializa_reloj_hh</code>	\$a0 : memory address for variable <i>reloj</i> \$a1 : HH	<code>reloj.hh = HH</code>
<code>inicializa_reloj_mm</code>	\$a0 : memory address for variable <i>reloj</i> \$a1 : MM	<code>reloj.mm = MM</code>
<code>inicializa_reloj_ss</code>	\$a0 : memory address for variable <i>reloj</i> \$a1 : SS	<code>reloj.ss = SS</code>

For example, to initialize the MM field with the value 59, the routine call is:

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

- Implement at home the routines `inicializa_reloj_hh`, `inicializa_reloj_mm` and `inicializa_reloj_ss`.

► For now a time HH: MM: SS can be coded in different ways depending on the values we assign to the bits that are not needed for coding the HH, MM and SS fields. Now we want to force all times to be represented in a unique way. To this end we can force to zero those bits of the field that are not used. For example, time 02:03:12 can be only coded as 0x0002030C, while other combinations such as 0x6502030C, 0x89E203CC or 0xFFC2038C are not allowed. How the routine `inicializa_reloj` can implement this condition?

► The following routine operates on a *reloj* variable whose memory address is passed as argument in register `$a0` and with an X value that is passed in the less significant byte of `$a1`. Explain what happens after this execution.

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

Integer multiplication and division and their runtime

Integer multiplication and division operations are implemented in the MIPS R2000 architecture using two machine instructions for signed integers **mult** and **div** respectively, and **multu** and **divu**, for unsigned numbers. The result of these instructions is temporally stored in a special pair of registers named **hi** and **lo**.

The content of these temporary registers will be interpreted according to the instruction executed. For example, after executing a multiplication instruction, **hi** contains the upper part of the result (product) and **lo** the lower part. In this case, an overflow appears in the multiplication when the result needs more than 32 bits to be represented, that is, when **hi** is different from zero after executing **multu** or it is different from the sign bit replicated 32 times for the instruction **mult**. The detection of this possible overflow is responsibility of the programmer.

On the other hand, after the execution of a division instruction **hi** contains the remainder and **lo** the quotient. In MIPS R2000 architecture division by zero is an indefinite operation. Therefore, it is the responsibility of the programmer to verify that the divisor is non-zero before executing a division instruction.

In order to get the values contained in these two special registers it is necessary to move them to any general purpose register by means of the instructions **mfhi** (move from **hi**) and **mflo** (move from **lo**).

Let's see an example of the use of these instructions:

```
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
```

The result of the multiplication ($18 * 4 = 72$) can be represented in 32 bits, so only one temporary register is needed (**lo**) to temporally store the product. So, after executing **mult** only **lo** content is moved to **\$s0**. In the case of division, the quotient ($18 \div 4 = 4$) is stored in **lo** and the remainder ($18 \text{ MOD } 4 = 2$) in **hi**. So, after the execution of instruction **div** we need to move these values to registers **\$s1** and **\$s2**, respectively.

A very important issue to consider is the complexity of integer multiplication and division operations and their runtime. Unlike basic logical operations, displacement, or arithmetic operations such as addition or subtraction that can be executed in a single clock cycle of the processor, multiplication and division are complex instructions that need more clock cycles to be executed. The division instruction is usually the one that takes the longest. However, the exact runtime depends on the implementation of the processors. To give you an idea, a division operation can take between 35 and 80 clock cycles, whereas a multiplication takes between 5 and 32 clock cycles. If we assume that multiplication takes 20 cycles and division 70, we can estimate the runtime of the previous code in $1 + 1 + 20 + 1 + 70 + 1 + 1 = 95$ clock cycles. Note that in this calculation it has been considered that the pseudoinstructions of the code can be translated by a

single machine instruction because the constant values can be encoded in 16 bits (for example, `li $t0,18` is translated into `ori $t0,$zero,18`).

The multiplication operation: conversion from HH: MM: SS to seconds

Now, we want to implement a routine to transform in seconds the value of a *reloj* variable that codes time in the format HH: MM: SS. For example, time 18:32:45 equals 66765 seconds. For the calculation simply do $18 \times 3600 + 32 \times 60 + 45 = 66765$.

The routine is named *devuelve_reloj_en_s*. Its specification is:

NAME	INPUT PARAMETER	OUTPUT
devuelve_reloj_en_s	\$a0 : memory address for variable reloj	\$v0 : seconds

The program initially provided in the *reloj.s* file also includes the routine **imprime_s** to print seconds (the value of seconds is passed as an argument in \$a0). For example, the following code calculates the time 18:32:45 in seconds and prints the result:

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

The result displayed after the execution is:



► To read separately each field of the reloj variable (HH, MM y SS) we can use a load byte instruction. Which instruction is more convenient for this **lb** (load byte) o **lbu** (load byte unsigned)?

► Implement the routine **devuelve_reloj_en_s**.

► Which addition instruction should be used in the routine, **add** or **addu**?

- How many multiplication instructions are executed in the subroutine *devuelve_reloj_en_s*?
- How many instructions to move information between the general purpose registers and registers **hi** and **lo** are executed in the routine?
- You want to modify the routine *devuelve_reloj_en_s* to detect multiplication overflow situations. Since we are working with positive numbers, we must include the necessary instructions to detect overflow if, after carrying out the multiplication operation, the result needs more than 32 bits to be encoded. In the case of an overflow appears, a branch instruction to quit the program is required. Provide the necessary instructions for this overflow detection.
- Let's suppose that all instructions (except **mult**) take a clock cycle to be executed and the multiplication instruction takes 20 cycles. Which is the runtime for the routine *devuelve_reloj_en_s*?

The Division operation: converting seconds in HH:MM:SS

In this section we propose you the opposite problem considered in the previous section: now we want to initialize the clock value starting from a predetermined amount of seconds. Thus, we want to design a routine that, given a number of seconds, initialize the clock with the corresponding time expressed in the HH: MM: SS format. For example, a time of 66765 seconds equals 18:32:45, that is, 18 hours, 32 minutes and 45 seconds. To convert from seconds to HH: MM: SS it is necessary to divide twice by the constant 60, as we can see below:

- $66765 \text{ seconds} \div 60 = 1112 \text{ minutes (remainder is 45 seconds)}$
- $1112 \text{ minutes} \div 60 = 18 \text{ hours (remainder is 32 minutes)}$

Therefore, SS are obtained in the remainder of the first division, MM in the remainder of the second division and HH in the quotient of the second division.

The routine implementing this new time initialization is name *inicializa_reloj_en_s*. Below you can see its specification.

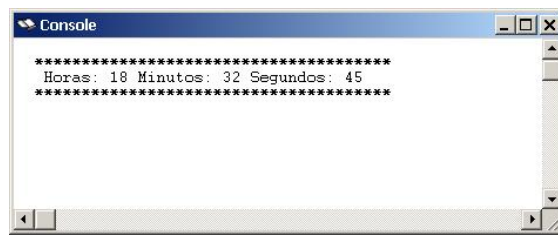
NAME	INPUT PARAMETERS	OUTPUT
inicializa_reloj_en_s	\$a0 : memory address for variable reloj \$a1 : seconds	reloj = HH:MM:SS

For instance, the next chunk of code initializes the *reloj* variable with the time corresponding to 66765 seconds and prints the result according to the format HH:MM:SS

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

la $a0, reloj
jal imprime_reloj
```

The displayed result is:



```
*****
Horas: 18 Minutos: 32 Segundos: 45
*****
```

- ▶ Implement the routine *inicializa_reloj_en_s*.
- ▶ How many division instructions are executed in the routine *inicializa_reloj_en_s*?
- ▶ How many instructions to move information between the file of general purpose registers and registers **hi** and **lo** are executed in the routine?
- ▶ Let's suppose that all instructions (except **div**) take a clock cycle to be executed and the multiplication instruction takes 70 cycles. Which is the runtime for the routine *inicializa_reloj_en_s*?
- ▶ You want to avoid possible errors when dividing by zero. So, you have to detect this situation before the division instruction is executed and quit the program if it happens. Write the instructions needed to perform this **division-by-zero detection**.