COMPUTER ORGANIZATION

Lab Session 5

INTEGER ARITHMETIC:
MULTIPLICATION AND DIVISION

Introduction

This is the first of two lab sessions about integer arithmetic on the MIPS R2000. This lab session focuses on integer multiplication and division. You will use these operations to implement several routines (user functions) that operate on variables representing time in a digital clock.

Goals

- To understand how the MIPS R2000 performs integer multiplication and division operations.
- To measure the execution time of programs that use integer multiplication and division operations.

Material

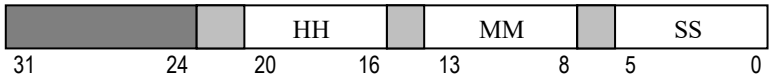
The material for this session can be found in the corresponding PoliformaT folder

- MIPS R2000 Simulator: PCSpim (in folder LAB/TOOLS)
- Source file: reloj.s (in folder LAB/P05)

Encoding and initialisation of time values

Consider a data type to represent time values. This type represents a time as a triplet HH:MM:SS (hours, minutes, and seconds). The valid range of time values goes from 00:00:00 to 23:59:59. Values such as 43:90:21 or 128:40:298 are invalid.

To implement this data type, we need to make room for the three components of the triplet. Given the valid ranges of HH, MM, and SS, it is possible to pack all three values in a single 32-bit word. We need 5 bits to encode HH (to cover the range 0…23) and 6 bits for each MM and SS (to cover the range 0…59), for a total of 17 bits. The following figure shows the data structure we will be using to represent time values in this session. To facilitate access to the three components of a time, each one is encoded in one of the three lower bytes of a word.



There are unused bits in the word (in grey) whose values are irrelevant and should be ignored. In particular, the highest byte (bits 24 ... 31) is not used; the HH field takes only 5 bits of its byte to cover values from 0 to 23; and the MM and SS fields have 6 bits each to cover the range 0…59.

Given this description of the time type,

► What is the time (HH:MM:SS) represented by the time value 0x0017080A?

► What is the time (HH:MM:SS) represented by the time value 0xF397C84A?

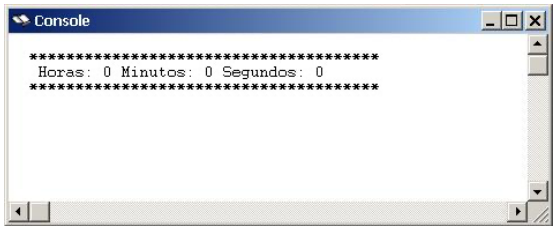
► Write three different encodings of a time variable representing the time 16:32:28.

Consider the assembly program given in file **`reloj.s`**. Part of this program is listed below:

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

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

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

                .globl __start
                .text 0x00400000

__start         la $a0, reloj
                jal imprime_reloj

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

The variable *reloj* is declared in the data segment to store a time value with the format described above. The program calls the user function **`imprime_reloj`** (included in reloj.s). This function prints the value of a time variable in a human-readable form. The function takes the address of a time value as its only argument in register $a0. In this program, the argument passed to *imprime_reloj* is the address of the variable *reloj*. After printing the value of *reloj*, the program terminates with a call to the *exit* system function.

► Load and run **`reloj.s`** in the PCSpim simulator. The result displayed on the console should be the following:



► Why is this time 00:00:00 displayed?

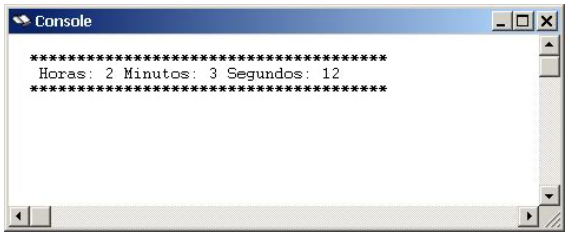
Because the value of reloj is 0

Our next goal is to implement several user functions to set the value of a time variable. The first one, **`inicializa_reloj`**, takes (in $a1) a time value encoded in the time format and assigns it to a variable whose address is given in $a0. The specification of this function is as follows:

FUNCTION NAME	INPUT PARAMETERS	RESULT
inicializa_reloj	**$a0**: memory address of a time variable **$a1**: a time value HH:MM:SS	*variable* = HH:MM:SS

The following example initializes *reloj* to 02:03:12 and then prints its value on the console:

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

la $a0, reloj
jal imprime_reloj
```



►Implement the function **`inicializa_reloj`**.

Now you need to provide an alternative function (**`inicializa_reloj_alt`**) to initialise a time variable, but this function takes the three fields as three separate arguments. The specification is:

FUNCTION NAME	INPUT PARAMETERS	RESULT
inicializa_reloj_alt	**$a0**: memory address of a time variable **$a1**: number of hours HH **$a2**: number of minutes MM **$a3**: number of seconds SS	*variable* = HH:MM:SS

► Implement the function **`inicializa_reloj_alt`**.

As homework, not during this lab session, implement a set of routines to separately initialise each of the fields of a time variable, as specified in the following table:

FUNCTION NAME	INPUT PARAMETERS	RESULT
inicializa_reloj_hh	**$a0**: memory address of a time variable **$a1**: number of hours HH	*variable.hh* = HH
inicializa_reloj_mm	**$a0**: memory address of a time variable **$a1**: number of minutes MM	*variable.mm* = MM
inicializa_reloj_ss	**$a0**: memory address of a time variable **$a1**: number of seconds SS	*variable.ss* = SS

For example, to set just the MM field of the time variable *reloj* to the value 59, you would write:

```
la $a0, reloj
li $a1, 59
jal inicializa_reloj_mm
```

► Up to now, a time HH: MM: SS could be encoded differently depending on the values of the unused bits of the time format. Now, we want to force all time values to be represented in only one form, with **all unused bits set to zero**. For example, the time 02:03:12 can only be encoded as 0x0002030C, while other combinations, such as 0x6502030C, 0x89E203CC, etc., would be *invalid* encodings of time values. Modify **inicializa_reloj** and **inicializa_reloj_alt** to implement this redefinition of the time type. (Hint: use the logic operation *and* with 0 to clear all unused bits before you store them in the time variable).

► The following function is known to operate on a time variable whose address is passed as an argument in $a0. It also uses a value *X,* passed in $a1. Explain the result of calling this function.

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

The result is the or operation between the bits of the time variable that represent hours and minutes and the value X passed in $a1.

Multiplication, division, and their execution times

Integer multiplication and division are implemented in the MIPS R2000 architecture using two machine instructions for signed integers, **`mult`** and **`div`**, and their corresponding unsigned counterparts, **`multu`** and **`divu`**.

The results of these instructions are stored in a pair of special registers named ***hi*** and ***lo***. After executing a multiplication instruction, ***hi*** and ***lo*** contain the product's most and least significant parts, respectively. After unsigned multiplication, if the product requires more than 32 bits to be represented, then ***hi*** will be different to zero. If the multiplication is signed, we know that the result requires more than 32 bits if ***hi*** is different from the sign extension of ***lo***. Detection of these 32-bit overflow conditions is a programmer's responsibility: multiplication instructions never cause exceptions.

Division instructions also use registers ***hi*** and ***lo*** to store two results. After a division, ***lo*** contains the quotient, and ***hi*** contains the remainder of the operation. In MIPS R2000, division by zero produces an indefinite result, and there is no hardware detection for this condition. Therefore, the programmer must verify that the divisor is non-zero before executing a division.

After multiplication or division, results can be moved from the special-purpose registers ***hi*** and ***lo*** to general-purpose registers. The instructions **`mfhi`** (*move from **hi***) and **`mflo`** (*move from **lo***) serve this purpose. Registers ***hi*** and ***lo*** cannot be used with any other instruction. Consider, for example, the following piece of code:

```
li $t0, 18          # $t0 = 18
li $t1, 4           # $t1 = 4
mult $t0, $t1       # lo = 18*4 = 0x00000048;       hi = 0x00000000
mflo $s0            # $s0 = lo

div $t0, $t1        # lo = 18÷4 = 4;                hi = 18%4 = 2
mfhi $s1            # $s1 = hi (remainder)
mflo $s2            # $s2 = lo (quotient)
```

In this code, the multiplication result (18 * 4 = 72) can be represented in 32 bits, so the register ***lo*** holds the whole product, and ***hi*** is zero. Hence, moving ***lo*** to $s0 is sufficient in this case. In general, the programmer should check for potential overflow. In the division case, the quotient (18 ÷ 4 = 4) is stored in ***lo***, and the remainder (18 % 4 = 2) in ***hi***. After the execution of **`div`**, the example code moves these values to registers $s1 and $s2, respectively.

Unlike basic logical operations, shifts, or arithmetic addition and subtraction, which can be executed in a single clock cycle of the CPU, multiplication and division usually require more clock cycles. Indeed, their exact execution time depends on the speed of the respective operators. To get an idea, a division operation can take from 35 to 80 clock cycles, whereas multiplication needs between 5 and 32 cycles. For example, if multiplication takes 20 cycles and division takes 70, the execution time of the previous seven instructions would be 1 + 1 + 20 + 1 + 70 + 1 + 1 = 95 clock cycles[1].

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

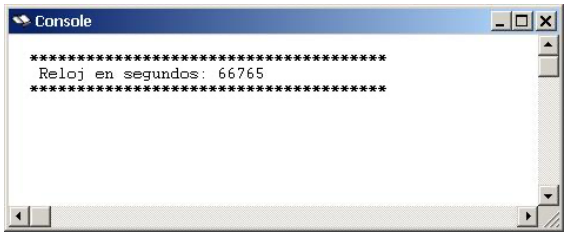
We will now use multiplication to implement a function that converts a time value (HH:MM:SS) into an equivalent number of seconds. For example, the time 18:32:45 corresponds to 66,765 seconds: (**18** h × 3600 s/h) + (**32** m × 60 s/m) + **45** s = 66,765 s.

► Implement the function *devuelve_reloj_en_s*, with the following specification:

FUNCTION NAME	INPUT PARAMETER	RESULT
devuelve_reloj_en_s	**$a0**: memory address of a time variable	**$v0** = seconds equivalent to the time given in variable pointed to by $a0

The program initially provided in file ***reloj.s*** includes the function **imprime_s** to print a given number of seconds on the console. The value of seconds to print is passed as an argument in $a0. You can use this function to display the result of a conversion. For example, the following code calculates the number of seconds of 18:32:45 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
```



►Given a time variable, one can use *load-byte* instructions to read HH, MM, and SS separately. Which one would be appropriate for this purpose, **`lb`** (load byte) or **`lbu`** (load byte unsigned)?

[1] Note that the pseudoinstructions *li* in this example can be translated into a single machine instruction each, because the constants can be encoded in 16 bits (for example, **`li $t0,18`** translates into **`ori $t0,$zero,18`**).

► Implement the function **devuelve_reloj_en_s**.

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

► How many multiplication instructions are executed inside the function ***devuelve_reloj_en_s***?

► How many instructions are needed in the function to move information between the general-purpose registers and registers ***hi*** and ***lo***?

► Modify the function ***devuelve_reloj_en_s*** to detect multiplication overflow. Since we are working with positive numbers, an overflow occurs if and only if the product takes more than 32 bits. In the case of overflow, use a branch instruction to quit the program.

► Assuming all instructions (except **`mult`**) take one clock cycle and the multiplication instruction takes 20 cycles, calculate the execution time (in cycles) of ***devuelve_reloj_en_s***.

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

Now, we tackle the opposite conversion, from the number of seconds in the day to a time value in hours, minutes, and seconds. For example, a time of 66,765 seconds corresponds to 18:32:45, that is, 18 hours, 32 minutes, and 45 seconds. To convert from seconds to HH:MM:SS, you need to divide the number of seconds by the constant 60, and then again, as shown next:

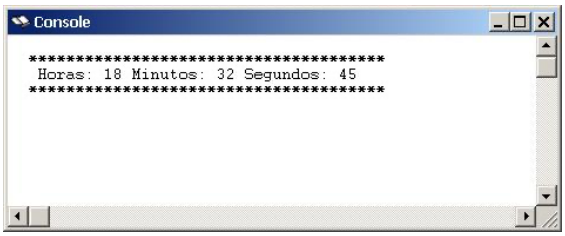
- 66765 seconds ÷ 60 = 1112 minutes (the remainder is **45** seconds)
- 1112 minutes ÷ 60 = **18** hours (the remainder is **32** minutes)

Therefore, SS is the remainder of the first division, MM is the remainder of the second division, and HH is in the quotient of that same second division. Your goal now is to implement the function ***inicializa_reloj_en_s*** specified as:

FUNCTION NAME	INPUT PARAMETERS	RESULT
inicializa_reloj_en_s	**$a0**: memory address of a time variable **$a1**: number of seconds	*variable* = HH:MM:SS corresponding to the number of seconds in $a1

For instance, the following code fragment initialises the *reloj* variable with the time corresponding to 66,765 seconds and prints the result in HH:MM:SS.

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

la $a0, reloj
jal imprime_reloj
```



► Implement the function ***inicializa_reloj_en_s***.

► How many division instructions are executed inside ***inicializa_reloj_en_s***?

► And how many instructions are needed to move data from ***hi*** or ***lo*** to general-purpose registers?

► If the function ***inicializa_reloj_en_s*** is correctly implemented, there is no chance of a division by zero occurring. However, the programmer should generally check for a potential division by zero. Modify the function to make sure it avoids division by zero. You must check that the divisor is not zero before executing the division or exit the program otherwise.

► Assuming the division instruction takes 70 cycles, and all other instructions take just one clock cycle, calculate the execution time of ***inicializa_reloj_en_s*** (in cycles).