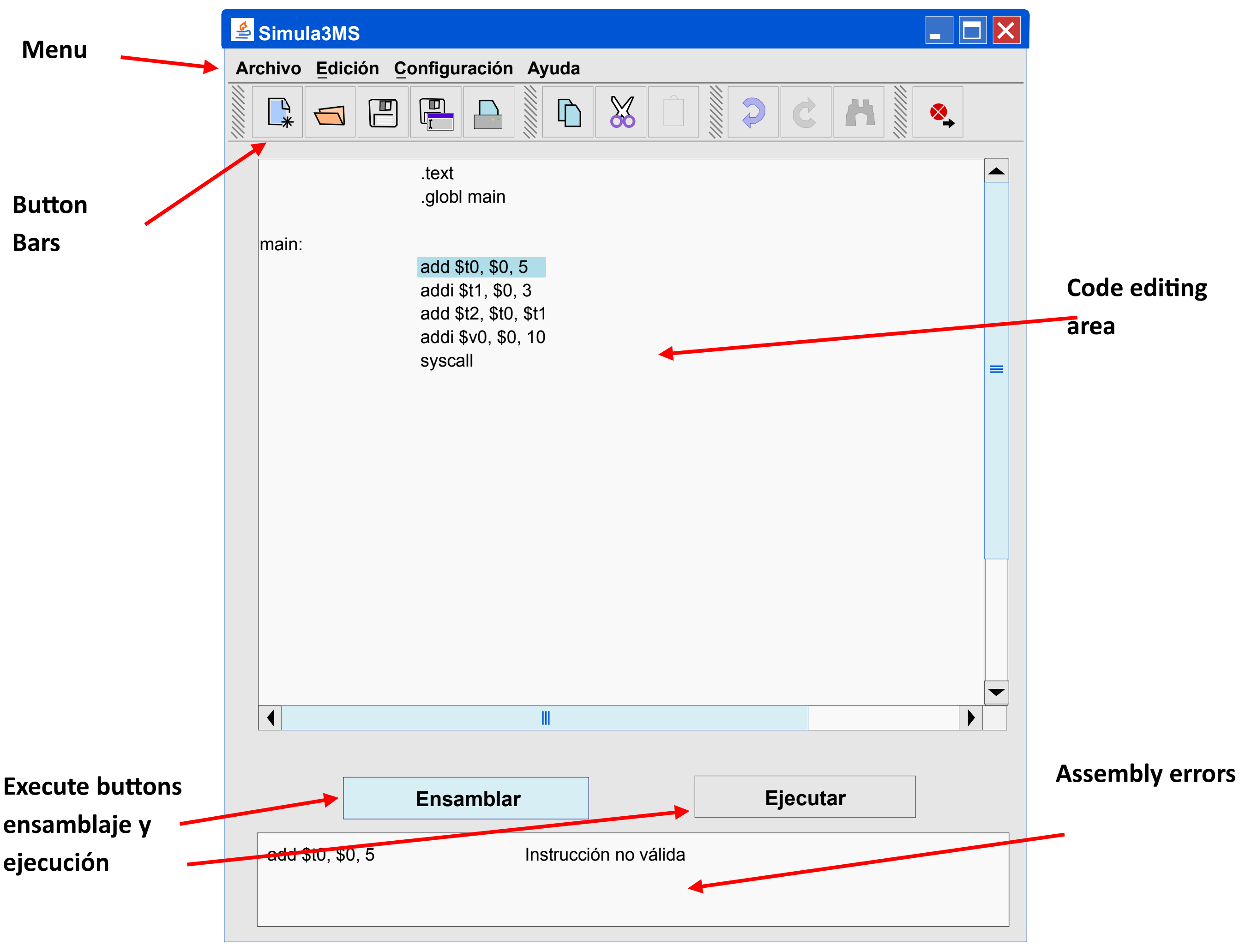
Simula3MS user's manual

In this manual, we explain how to use Simula3MS for programming in assembly language for the MIPS microprocessor. The student is advised to follow the exercises step by step, making sure to understand each of them and the related concepts, and trying to answer the proposed questions.

Editing and simulating a simple program

After starting *Simula3MS,* a valid assembly code must be introduced either by editing or loading it from a file. As a first example, let's type the following piece of code in the area labeled as "code editing area" in Figure 1.

```
.text                          # These first 3 lines should be located
.globl main                    # at the beginning of the program
main:                          # It's really compulsory!!

addi $t0, $0, 5
addi $t1, $0, 3
add $t2, $t0, $t1
addi $v0, $0, 10
syscall
```



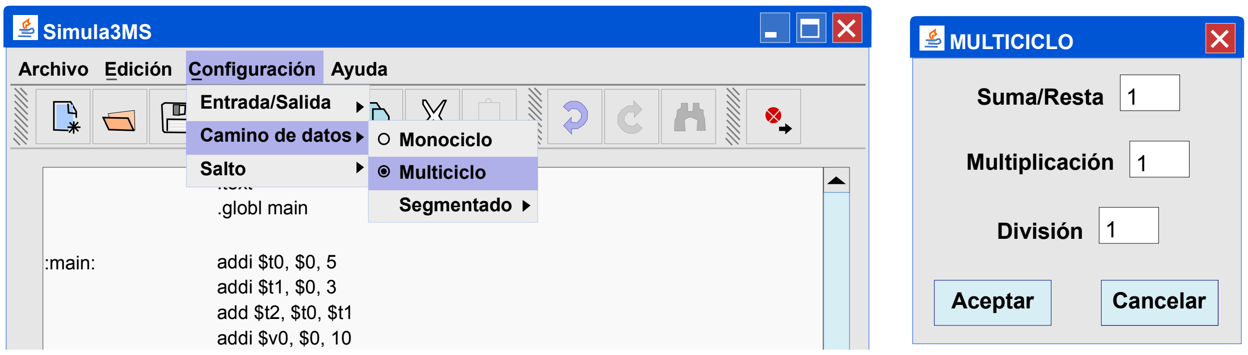
```
            .text
            .globl main

main:
            add $t0, $0, 5
            addi $t1, $0, 3
            add $t2, $t0, $t1
            addi $v0, $0, 10
            syscall
```

```
add $t0, $0, 5               Instrucción no válida
```

Figure 1. Editor's window

It is possible to save the program into a disk file and load it afterwards when desired. This can be done by accessing the menu options below the *Archivo* category, or using the buttons in the top bar.

One of the options below the *Configuración* category allows us to select the *Multiciclo* (multicycle) option (Figure 2), which is required for observing how instructions are executed in a step by step fashion. If the *Monociclo* option is preferred, every instruction will be executed in a single step. The *Multiciclo* option is recommended as it conforms to the behavior explained during the lectures.

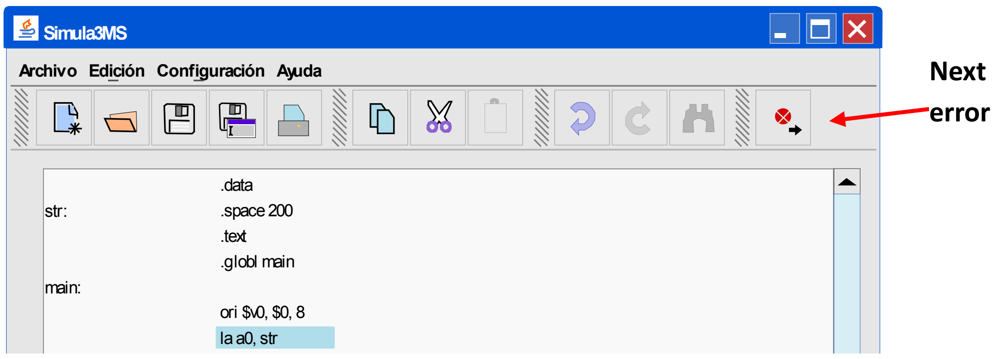


```
                    .globl main

:main:              addi $t0, $0, 5
                    addi $t1, $0, 3
                    add $t2, $t0, $t1
                    addi $v0, $0, 10
```

Suma/Resta	1
Multiplicación	1
División	1

Figure 2. Selecting the *Multiciclo* choice

Then, click on *Ensamblar*. If the code is syntactically correct, the *Ejecutar* button will be enabled. If not, the errors should be corrected. The errors in the code are highlighted, and the *Next error* button (see Figure 3) can be used to iterate through them. By clicking on *Ejecutar*, the simulation is started.



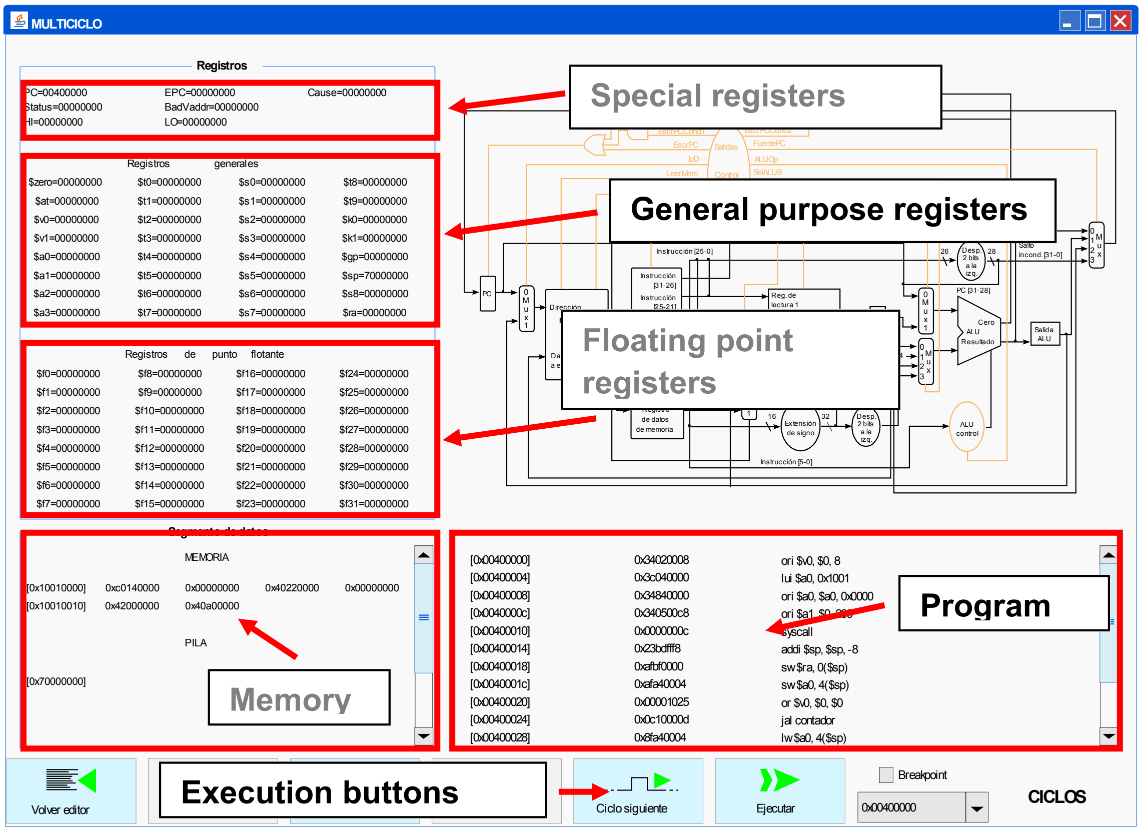
```
                    .data
str:                .space 200
                    .text
                    .globl main
main:
                    ori $v0, $0, 8
                    la a0, str
```

Figure 3. Next error button

After correcting the errors, if any, clicking on *Ejecutar* will launch the simulation. The execution/simulation screen in shown in Figure 4.

It is important to spend some time examining the execution screen and identifying the different sub-windows, which are:

- The general purpose registers, where all the 32 general purpose registers in the MIPS processor can be inspected. After every writing onto a register, the value on this window will be updated, so that the operation of the program can be followed
- The program segment, where the current instruction is highlighted. For every instruction in this sub-window, the machine code and the memory address are also shown
- The data segment (Memory) were all the writing operations on memory are reflected
- The execution buttons, that will be described in more detail in the next paragraph



```
Registros
PC=00400000              EPC=00000000            Cause=00000000
Status=00000000          BadVaddr=00000000
HI=00000000              LO=00000000

                 Registros        generales
$zero=00000000   $t0=00000000   $s0=00000000   $t8=00000000
$at=00000000     $t1=00000000   $s1=00000000   $t9=00000000
$v0=00000000     $t2=00000000   $s2=00000000   $k0=00000000
$v1=00000000     $t3=00000000   $s3=00000000   $k1=00000000
$a0=00000000     $t4=00000000   $s4=00000000   $gp=00000000
$a1=00000000     $t5=00000000   $s5=00000000   $sp=70000000
$a2=00000000     $t6=00000000   $s6=00000000   $s8=00000000
$a3=00000000     $t7=00000000   $s7=00000000   $ra=00000000

                 Registros   de   punto   flotante
$f0=00000000     $f8=00000000   $f16=00000000   $f24=00000000
$f1=00000000     $f9=00000000   $f17=00000000   $f25=00000000
$f2=00000000     $f10=00000000  $f18=00000000   $f26=00000000
$f3=00000000     $f11=00000000  $f19=00000000   $f27=00000000
$f4=00000000     $f12=00000000  $f20=00000000   $f28=00000000
$f5=00000000     $f13=00000000  $f21=00000000   $f29=00000000
$f6=00000000     $f14=00000000  $f22=00000000   $f30=00000000
$f7=00000000     $f15=00000000  $f23=00000000   $f31=00000000

Segmento de datos
                          MEMORIA
[0x10010000]   0xc0140000   0x00000000   0x40220000   0x00000000
[0x10010010]   0x42000000   0x40a00000

                          PILA
[0x70000000]

[0x00400000]   0x34020008   ori $v0, $0, 8
[0x00400004]   0x3c040000   lui $a0, 0x1001
[0x00400008]   0x34840000   ori $a0, $a0, 0x0000
[0x0040000c]   0x340500c8   ori $a1, $0, 200
[0x00400010]   0x0000000c   syscall
[0x00400014]   0x23bdfff8   addi $sp, $sp, -8
[0x00400018]   0xafbf0000   sw $ra, 0($sp)
[0x0040001c]   0xafa40004   sw $a0, 4($sp)
[0x00400020]   0x00001025   or $v0, $0, $0
[0x00400024]   0x0c10000d   jal contador
[0x00400028]   0x8fa40004   lw $a0, 4($sp)

0x00400000
```

Figure 4. Execution screen

Whereas it is possible to execute the whole program by clicking on *Ejecutar*, it is fairly more interesting to proceed instruction by instruction by clicking on ***Paso siguiente***. The button ***Paso anterior*** allows undoing the last instruction. When in *Monociclo* mode, the buttons to press would be ***Ciclo siguiente*** and ***anterior***. For starting over, just click ***Volver editor*** to return to the code editor and then click on ***Ejecutar***.

Exercise 1

Type the code from the first page, simulate it step by step following this sequence of operations, and mark the checkboxes to show your progress:

❒ Observe the *Segmento de texto*. The first instruction is `addi $t0, $0, 5`

  ❒ That instruction will assign $0 + 5 to register $t0, that is, $t0 = 5

❒ Search for register `$t0` on the general purpose registers area. Check its value.

❒ Click on *Paso siguiente*. Check that the value of `$t0` has actually changed.

☑

❒ The first instruction is now highlighted. So far, it is the only executed instruction

❒ The second instruction is `addi $t1, $0, 3`, to make `$t1 = 3`

❒ Observe the value of `$t1`. Click on *Paso siguiente*. Check the value `$t1` again

❒ The third instruction is `add $t2, $t0, $t1`, to make `$t2 = $t0 + $t1`

  ❒ Check that the addition is actually calculated

❒ The last 2 instructions will terminate the program execution

❒ Execute `addi $v0, $0, 10`, that will load the special code “`10`” in `$v0`

  ❒ Check that it works, and then execute syscall for terminating the program

❒ Execute the program again, this time in a single step, by clicking on *Ejecutar*. If preferred, use the step by step execution to better understand how it works

The registers

The value of every register is shown on screen as explained in Figure 4. The registers belong to one of these 3 categories:

- **Special registers:**
    - *PC* is the program counter. This register is initialized by the Operating System to point at the memory address where the first instruction in the program is located. After loading a new instruction from memory, the PC is incremented in such a way that it points now at the next instruction to be loaded.

IMPORTANT: Execute the program again step by step and check how the value of PC changes. Regard how the value in PC is related to the memory address in which each instruction is stored in memory. That address can be seen at the first column in the *Segmento de texto*, sub-window.

    - *EPC, Cause, Status* and *BadVaddr,* are special registers that are required to manage exceptions e interruptions.
    - *Hi and Lo* are 2 registers that are used to operate with-64 bit numbers. These are required when calculating multiplications and quotients.

- **Floating point registers:** In the MIPS R2000 architecture, floating point calculations are carried out using a co-processor, which makes us of a set of 32, 32-bit registers, that can also be organized as 16, 64-bit registers for double precision operations. Floating point operations will not be tackled in this subject.
- **General registers:** The MIPS architecture is built around a set of 32 general purpose registers, 32-bit each. Most of them are meant for programmer’s use (see Table 1). Registers are numbered (`$0-$31`), but they can be more commonly referred to by their names.

Name	**Number**	**Purpose**
`$zero`	0	Constant 0 value. Read-only
`$at`	1	Reserved for the assembler
`$v0-$v1`	2-3	For passing results and evaluating expressions
`$a0-$a3`	4-7	Subroutine arguments
`$t0-$t7`	8-15	Temporal registers
`$s0-$s7`	16-23	Saved registers
`$t8-$t9`	24-25	Temporal registers
`$k0-$k1`	26-27	Reserved for the Operating System
`$gp`	28	Global pointer. Not implemented in *Simula3MS*
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer. Not implemented in *Simula3MS*
`$ra`	31	Return address for subroutines

Table 1. Numbering and naming convention for MIPS registers

Exercise 2. Moving data among registers

At the editor, type the following code (except the ellipses)

```
.text
.globl main
main:

addi $t0, $0, 5
addi $t1, $0, 3
addi $t2, $0, 14
…
…
addi $v0, $0, 10
syscall
```

Next, assembly and execute the program step by step. After checking how it works, extend the program writing new code where the ellipses are located. The following must be accomplished:

❒ Exchange the values in registers `$t0` and `$t1` using a temporal register

  ❒ **Do not** load the values as constants again

❒ Check the code step by step and inspect the values at the registers

❒ Make a rotation of the values in registers `$t0`, `$t1` and `$t2`, so that each value passes through the 3 registers

❒ Can you see value "*14*", does it represent a problem?

❒ Try to write as few instructions as possible: can you use just 4 instructions?

Exercise 3. Basic arithmetic

Edit the previous code to restore it to its original form. Then, modify it to perform the following operations:

❒ Add the 3 registers and put the result on a different one

❒ Try to use as few instructions and registers as possible

❒ Execute step by step and check that the result is correct

❒ Calculate `$t0 – $t1`, `$t1 – $t2` and `$t2 – $t0` putting the results into other registers

❒ Check the results. Do any of them look weird? Think about it

❒ Add the results of the 3 subtractions

Exercise 4. Executing a program with data input/ouput

Back to the code editor, type the following piece of code. Comments may be omitted (#).

```
.text
.globl main
main:

addi $v0, $0, 5

syscall                 # Read an integer number by keyboard

add $a0, $v0, $0        # Copy the integer number to register $a0
addi $v0, $0, 1
syscall                 # write the same value to screen

addi $v0, $0, 10
syscall                 # Terminate the program
```

❒ Assembly and execute the program step by step

❒ After executing the first `syscall` instruction, the program will ask you to type a number. The program will resume afterwards

❒ Then, the program will display that number on the screen

❒ Then, it terminates

The 3 tasks in the previous piece of code make use of `syscall`. The instruction `syscall` carries out different operations depending on the service code passed by in register `$v0`. The list of services that we will use in this subject is shown below:

Service	Code in $v0	Arguments	Result
print_int	1	`$a0` = integer number	
print_string	4	`$a0` = text string	
read_int	5		Integer in `$v0`
read_string	8	`$a0` = buffer, `$a1` = length	
exit	10		

Some of those services require passing some arguments in register `$a0` (and occasionally in `$a1`). Other services return a result in register `$v0`.

IMPORTANT: Execute the program step by step and analyze the purpose of instructions `add $a0, $v0, $0` and `addi $v0, $0, 1`

Using the memory

Computers' memory is large and there, is where most of the data used by a program is stored. Registers, on the contrary, may store just little pieces of information. The usual way in which a program works, consists of taking some data from the memory to the registers, make some calculations, and save the results back into memory.

In order to read and write data into memory, we need to know **where** those data are allocated. That information is given by the **address**. In the MIPS, there are 4,294,967,296 possible addresses. Those addresses are just numbers.

In order to easily work with them, **labels** are used. Labels are words or pieces of text that represent a memory address. Before using a label, it must be defined.

In a program, the start of the data segment is signaled by the directive **`.data`**. Simula3MS will assign a place in memory to all the data items declared in the data segment. When a label is written, Simula3MS will make the necessary calculations to convert the label to a valid memory address.

Exercise 5. Executing a program that accesses data in memory using labels

In the following program, an `.asciiz`–type text string (Hello world!) and 3 `.word`–type integer numbers have been declared. Also, 2 labels have been assigned: `myText` and `myNumbers`. Data and program code are always in different locations. Data is at the Data Segment, which begins at the **`.data`** directive. The Program Segment starts after the **`.text`** directive, and the label **`main`** signals the entry to the program.

```
.data
myText: .asciiz "Hello world!"
myNumbers: .word 1, 2, 3

.text
.globl main
main:

addi $v0, $0, 4
la $a0, myText                # what is this instruction for?
syscall                       # figure it out without executing the program

la $a2, myNumber

addi $v0, $0, 10
syscall
```

Type this piece of code, execute it, and check how it works. Next, execute it step by step, thinking about the following questions:

❒ Locate the Data Segment on the execution screen (see Figura 4). This is the first exercise for which the Data Segment contains interesting data.

❒ Try to locate the ASCII characters in "`Hello world!`" and the integers 1, 2 and 3.

❒ Observe what instruction `la $a0, myText` turns into, and execute it

❒ Check the value of `$a0`. Search for the same value into the Data Segment.

❒ Continue and execute `la $a2, myNumbers`

❒ Check the value of `$a2`. Inspect the Data Segment using `$a2` as an address. Which data can you find there?

IMPORTANT: The difference between the data stored in memory and their addresses should be clear for the student. In the previous piece of code, data are "Hello world!" and 1, 2, 3. Which are their addresses?

The addresses are stored in registers `$a0` and `$a2` but, does `$a0` contain the letters in "Hello world!"? Does `$a2` contain the numbers 1, 2 or 3?

Exercise 6. Reading data from memory

Using the same data as in the previous exercise, we will copy some numbers from memory to the registers, and we will also copy some bytes (ascii characters in this particular case).

```
.data
myText: .asciiz “Hello world!”
myNumbers: .word 1, 2, 3

.text
.globl main
main:

la $a0, myNumbers
lw $t0, 0($a0)                 # reads the first number to $t0
lw $t1, 4($a0)                 # reads the second number to $t1

la $a1, myText
lb $t4, 0($a1)                 # reads character ‘H’ to $t4
lb $t5, 1($a1)                 # reads character ‘e’ to $t5

addi $v0, $0, 10
syscall
```

- ❒ Check that `$t0` and `$t1` actually contain the 2 numbers we tried to read
- ❒ The 2 numbers are separated by a stride of 4 bytes, why?
- ❒ Check that `$t4` and `$t5` actually contain the **ASCII** characters for ‘H’ and ‘e’
- ❒ The 2 characters are separated by a stride of 1 byte, why?

Exercise 7. Writing data to memory

The instructions for writing to memory are **`sw`** and **`sb`**.

- ❒ Modify the previous program so that new values are written over the existing numbers (use **`sw`**) and some of the characters (use **`sb`**)
- ❒ Check that it actually works by inspecting the Data Segment
- ❒ You can also check it by printing on screen the piece of text or the numbers