UNIVERSITY OF BEJAIA

PRACTICAL WORK SUPPORT

Computer Architecture
Programming in MIPS R3000 assembly language

Author:
Louiza BELKHIRI

Educational document intended for computer science students

Speciality: Computer Science
Level: Second year of undergraduate studies (Licence 2)
in the

Department of Computer Science
Faculty of Exact Sciences
University of Bejaia

2024/2025

Contents

List of Figures

List of Tables

List of Abbreviations

A

ASCII	American Standard Code for Information Interchange
ASCIIZ	American Standard Code for Information Interchange with Zero
ALU	Arithmetic and Logic Unit

C

CPU	Central Processing Unit
CR	Cause Register

H

Hi	High Register

I

I	Immediate
IDE	Integrated Development Environment
I/O	Input/output

L

Lo	LOw Register
LSb	Least Significant bit
LSB	Least Significant Byte

M

Mac OS	Macintosh Operating System
MARS	MIPS Assembler and Runtime Simulator
MIPS	Microprocessor without Interlocked Pipeline Stages
MSb	Most Significant bit
MSB	Most Significant Byte

P

PC	Program Counter
PW	Particle Work

Q

QTSPIM	Qt framework with the SPIM simulator

R

RD	Register Destination
RS	Register Source

S

sh	shift amount
SIMIPS	Simulator for Microprocessor without Interlocked Pipeline Stages
SPIM	Simulator for Processing Interactive MIPS
SR	Status Register

Introduction

The study of computer architecture and assembly language programming is an essential part of a computer science curriculum, as it provides students with a deep understanding of how software interacts with hardware. This knowledge is critical for writing efficient programs, understanding system-level operations, and preparing for advanced areas such as operating systems, compilers, and embedded development.

This document serves as a practical guide for the computer architecture module aimed at second-year computer science students. Its primary objective is to introduce students to programming in the MIPS R3000 assembly language. By providing a structured approach to learning, this resource enables students to understand the fundamental concepts of assembly programming, including instruction sets, memory management, and processor operations.

Through assembly programming, students will explore a new dimension of coding, gaining insights into how instructions and data are represented and loaded into memory, as well as into the various registers of the processor. They will learn how these elements are processed by the corresponding units, deepening their understanding of abstract data structures and fundamental high-level programming principles. This hands-on experience will enhance their ability to think critically about the relationship between software and hardware.

The discovery-based learning method is an educational approach where students explore, experiment, and find solutions on their own, rather than passively receiving information. Popularized by American psychologist Jerome Bruner, this method is based on the idea that learners actively construct knowledge by interacting with their environment and solving problems, leading to a deeper and more lasting understanding of concepts. In this context, we have developed a series of pedagogical exercises grounded in discovery learning. This approach empowers students to explore and uncover the roles and functions of each instruction studied, enabling them to understand not only how to use these instructions effectively but also where to apply them in practical scenarios.

The rest of this document is organized as follows:

Part 1 outlines the assembly language of the MIPS R3000 processor, along with various conventions for writing assembly language programs. We begin by describing the external architecture of the MIPS R3000 microprocessor. This architecture includes the organization of the memory, the various registers used by the processor to execute instructions, and the instruction set, which includes all the operations performed by the processor. Secondly, we will discuss the general syntax and rules for writing a program in MIPS R3000 assembly language. This section includes the directives used to declare sections and variables, comments, supported macro instructions, and finally the various system calls that allow interaction with the user.

Part 2 describes the working environment and the tool used to create and execute programs written in MIPS R3000 assembler language. It begins with a presentation of the steps and tools required to develop an assembler program, from editing to assembly and execution. It then presents the SIMIPS R3000 emulator chosen for the implementation of assembler programs. This IDE was chosen for its simplicity, its perfect conformity with the rules of programming in this language and its simulation of the architecture of the MIPS R3000 machine. We provide a detailed user guide for this environment, with graphical interfaces that show each step in the process of developing and testing MIPS R3000 assembler programs.

Part 3 presents a collection of structured, hands-on exercises in MIPS R3000 assembly language programming. Each series focuses on a specific type of instruction through carefully designed exercises that support student learning. Generally, each series includes two types of activities: discovery exercises and application exercises.

In the discovery exercises, students are introduced to the fundamental assembly instructions, running provided programs and analyzing their outputs. These exercises encourage students to explore the function of each instruction by responding to guiding questions. Once the basics are understood, application exercises and comprehension tests enable students to apply their knowledge, develop their own assembly programs, and refine their understanding through practice. This approach allows students to bridge theory with hands-on programming skills, essential for mastering MIPS R3000 assembly language.

Finally, we provide a conclusion that summarizes the key points discussed in this document.

Part 1

MIPS R3000 assembly language overview

1.1 Introduction

This part provides an overview of MIPS R3000 assembly language. Firstly, the MIPS R3000 external computer architecture is covered, focusing on the basics of the instruction set defined by the various operations and instructions supported by the MIPS R3000 processor, memory organization and addressing. Important instruction operands such as immediate values, labels and general purpose registers are then described. The assembly language for the MIPS R3000 is described in the last section. It provides the program syntax which includes directives, comments, structure and macro instructions. Finally, input/output operations are shown to demonstrate interaction with external devices. This part provides the fundamental information necessary for writing and understanding assembly language programs for the MIPS R3000 processor.

1.2 MIPS computer Architecture

Understanding the external architecture of a microprocessor is essential when programming in assembly language. The architecture provides critical information about the processor's instruction set, CPU (Central Processing Unit) registers, memory organization and addressing, input/output interfaces and control signals. Without this knowledge, it is impossible to efficiently manage the flow of data, optimize performance or properly utilize the processor's resources. Structure directly affects how instructions are executed and how memory and peripherals are accessed, making it a fundamental aspect of assembler programming.

MIPS R3000 is a machine based on the Von Neumann architecture. It includes the following units (see Figure 1.1): a control unit, an Arithmetic and Logic Unit (ALU), central memory, and input/output units. The CPU is equipped with a variety of registers that support instruction execution and system control. Among these are **general purpose registers** that the processing unit uses to temporarily store data, addresses, and intermediate results during instruction execution, as well as other registers such as the **Program Counter (PC)** that contains the address of either the instruction that is currently being executed or the instruction that will be executed next, and the **Status Register (SR)** that indicates the current state of the processor and records key information about the outcome of operations (flags for zero, carry, overflow, negative results, etc.). For more details, readers can refer to [6, 5, 1, 10].

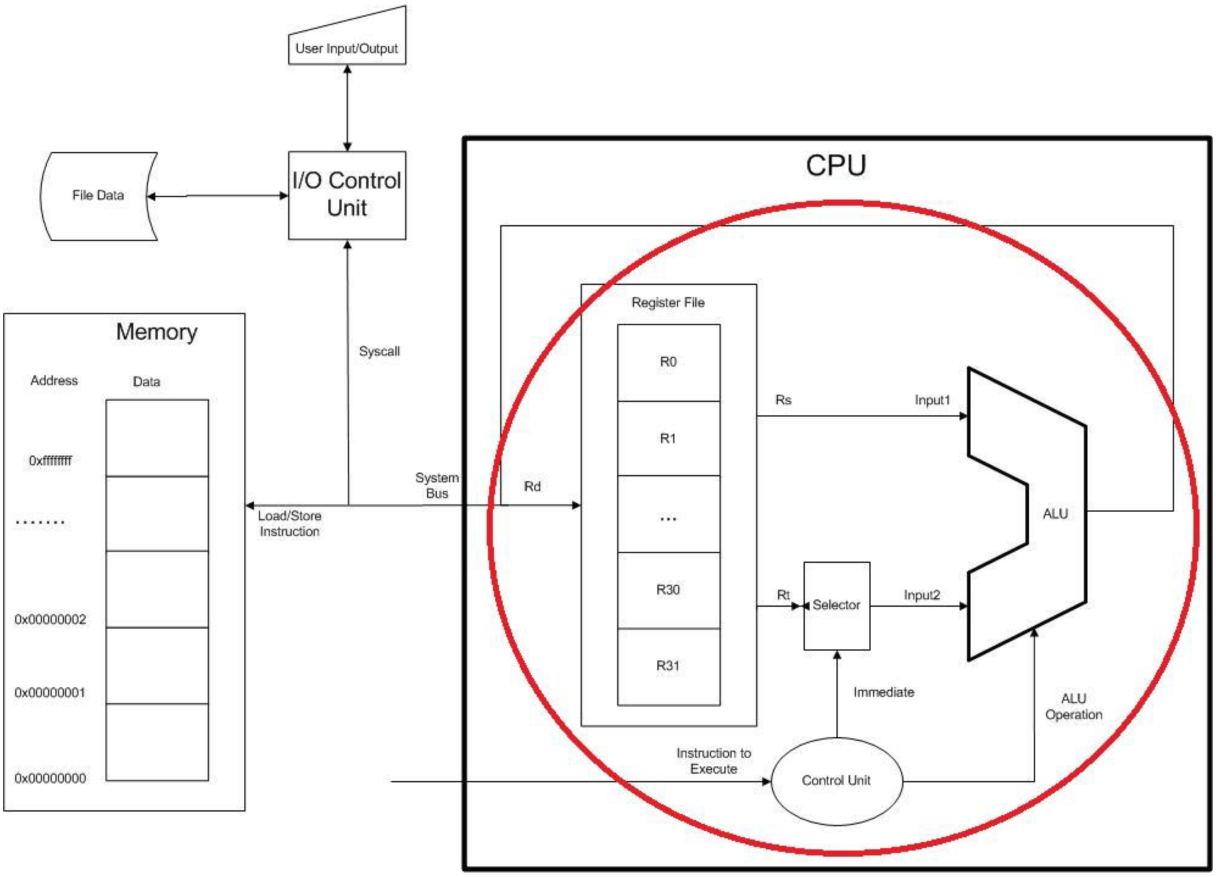

FIGURE 1.1: MIPS computer architecture [6]

1.2.1 Memory organization

- In the MIPS R3000 memory, the usable addressable space is divided into two segments: the user segment and the kernel segment, which are identified by the most significant bit of the address. In particular:
  - adr[31] = 0 indicates the user segment;
  - adr[31] = 1 refers to the kernel segment.
- Each segment is devised in 3 sections: Text, Data and Stack. The description of the MIPS R3000 memory sections is summarized in Table 1.1. An illustrative scheme of these partitions is shown in Figure 1.2.

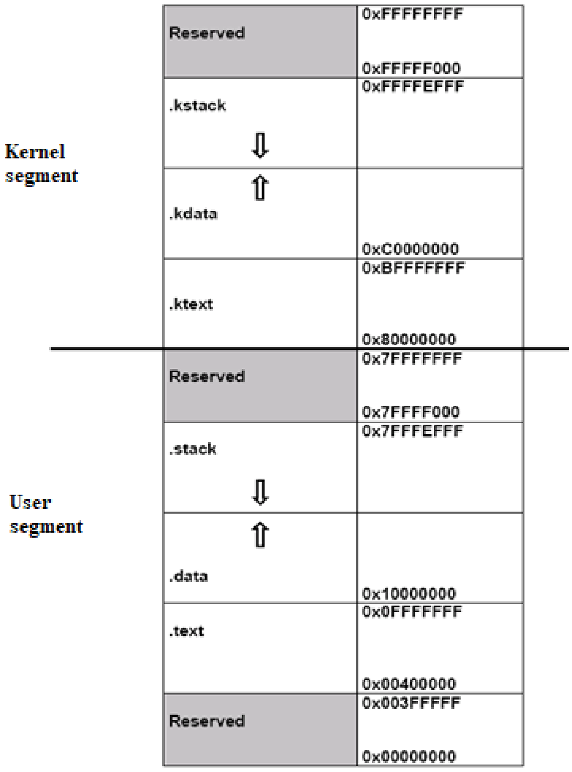

FIGURE 1.2: Partitions of MIPS R3000 memory [4]

Section Name	**Start Address**	**Description**
Text	0x00400000	Contains the instructions of the user program Each instruction is stored as a word (32 bits or 4 bytes)
Data	0x10000000	Contains global data manipulated by the user program The size of the elements is assigned at program creation
Stack	0x7FFFF000	Dynamic area allocated for subprograms Contains all subprograms local variables
KText	0x80000000	Contains machine instructions that are exclusive to the kernel of the operating system
Kdata	0xC0000000	Contains global data managed by the operating system in kernel mode
Kstack	0xFFFFF000	Contains the execution stack for the kernel
Reserved	Remaining memory	Memory reserved for the MIPS platform Addresses in this area are not usable by a program

TABLE 1.1: MIPS R3000 memory sections description [3]

- A memory address is defined on 32 bits (from address 0x00000000 to 0xFFFFFFFF).
- A memory location is defined as being 8 bits (one byte) wide only. Thus to save or load a 32 bits-instruction, 4 consecutive locations are needed.
- Data exchanges with memory occur byte-wise, half-word (2 consecutive bytes), or word-wise (4 consecutive bytes).
- A half-word's address must be a multiple of 2, whereas the address of a data word or instruction must be a multiple of 4. If an instruction calculates an address that deviates from these constraints, the processor will raise an exception.

1.2.2 Memory addressing

- Memory access instructions are included in the type I format and have the syntax **xy Rt, I (Rs)** where x={l or s} and y={w, h or b}.

  - The source register (Rs) is added to the immediate value to create an effective address, which is then used to reference memory.

  - The second register (Rt) serves either as the destination in a memory load or as the source in a memory store.
- The MIPS R3000 presents only one addressing mode for reading or writing data in memory: indexed register addressing with offset: I (Rs).
- For all load and store operations, the address is obtained by adding the offset I (positive or negative) to the content of the Rs register. For example, for the instruction `lw Rt, I (Rs)` the address is:

  @ = (Rs) + I (with sign extension of the immediate).
- **Examples**
  Assume that register \$10 contains the value 0x10000004 and that (\$22) = 0x710FFFFC.

  1. `lw $12, 20($10)` # \$12 <– Word[(\$10) + 20] *Load operation*
     The load address is: @ = (\$10) + $(20)_{10}$ = 0x10000004 + 0x14 = 0x10000018;
  2. `sw $20, -24($22)` # Word[(\$22) + (-24)] <– \$20 *Store operation*
     The store address is: @ = (\$22) - $(24)_{10}$ = 0x710FFFFC + 0xFFFFFFE8 = 0x710FFFE4.

1.2.3 Instruction set [8, 4, 3, 6]

A processor's instruction set is a set of operations that the CPU can use to perform various tasks. These instructions cover basic activities such as data transfer (move, load, store), logic (AND, OR, NOT), arithmetic (add, subtract, multiply) and control flow

(jump, branch, subprograms call). The processor decodes and executes each instruction according to a predetermined format, often using registers and memory addresses. The processor's capabilities are defined by its instruction set, which affects how well it can run programs and perform complex calculations.
The different instructions of MIPS R3000 processor are summarized in Tables 1.2, 1.3, 1.4, 1.5 and 1.6. The following notations are used:

$\ll$	shift left	$N_{x..y}$	Bits from position x to y of N
$\gg$	shift right	$\|\|$	concatenation
LSB	Least Significant Byte	MSB	Most Significant Byte
LSb	Least Significant bit	MSb	Most Significant bit

Assembly syntax	Operation	Effect	Format
Conditional branches			
Beq Rs, Rt, Label	Branch if Equal	$PC \leftarrow PC + 4 + 4 \times \text{I} \quad \text{if } Rs = Rt$	I
Bne Rs, Rt, Label	Branch if Not Equal	$PC \leftarrow PC + 4 + 4 \times \text{I} \quad \text{if } Rs \neq Rt$	I
Bgez Rs, Label	Branch if Greater or Equal Zero	$PC \leftarrow PC + 4 + 4 \times \text{I} \quad \text{if } Rs \geq 0$	I
Bgtz Rs, Label	Branch if Greater Than Zero	$PC \leftarrow PC + 4 + 4 \times \text{I} \quad \text{if } Rs > 0$	I
Blez Rs, Label	Branch if Less or Equal Zero	$PC \leftarrow PC + 4 + 4 \times \text{I} \quad \text{if } Rs \leq 0$	I
Bltz Rs, Label	Branch if Less Than Zero	$PC \leftarrow PC + 4 + 4 \times \text{I} \quad \text{if } Rs < 0$	I
Bgezal Rs, Label	Branch if Greater or Equal Zero and link	$\begin{cases} R31 \leftarrow PC + 4 \\ PC \leftarrow PC + 4 + 4 \times \text{I} \end{cases} \text{if } Rs \geq 0$	I
Bltzal Rs, Label	Branch if Less Than Zero and link	$\begin{cases} R31 \leftarrow PC + 4 \\ PC \leftarrow PC + 4 + 4 \times \text{I} \end{cases} \text{if } Rs < 0$	I
Unconditional branches			
J Label	Jump	$PC \leftarrow PC_{31..28} \,\|\|\, 4 \times \text{I}$	J
Jal Label	Jump and Link	$\begin{cases} R31 \leftarrow PC + 4 \\ PC \leftarrow PC_{31..28} \,\|\|\, 4 \times \text{I} \end{cases}$	J
Jr Rs	Jump Register	$PC \leftarrow Rs$	R
Jalr Rs	Jump and Link Register	$\begin{cases} R31 \leftarrow PC + 4 \\ PC \leftarrow Rs \end{cases}$	R

TABLE 1.2: Table of Branch instructions of MIPS R3000

Assembly syntax	Operation	Effect	Format
ALU transfer operations (move from/to register)			
Mfhi Rd	Move From HI	Rd ← HI	R
Mflo Rd	Move From LO	Rd ← LO	R
Mthi Rs	Move To HI	HI ← Rs	R
Mtlo Rs	Move To LO	LO ← Rs	R
Lui Rt, I	Load Upper Immediate Immediate is loaded into the 2 MSB of Rt and 0's are added to its LSB	Rt ← I « 16	I

TABLE 1.3: Table of Arithmetic and Logical Operations of MIPS R3000 (ALU Transfer instructions)

Assembly syntax	Operation	Effect	Format
	Addition		
add Rd, Rs, Rt	addition (Overflow detection)	Rd ← Rs + Rt	R
addu Rd, Rs, Rt	unsigned addition	Rd ← Rs + Rt	R
addi Rt, Rs, I	addition immediate (Overflow detection) Signe extended immediate	Rt ← Rs + I	I
addiu Rt, Rs, I	unsigned addition immediate	Rt ← Rs + I	I
	Subtraction		
sub Rd, Rs, Rt	subtraction (Overflow detection)	Rd ← Rs - Rt	R
subu Rd, Rs, Rt	unsigned subtraction	Rd ← Rs - Rt	R
	Multiplication and division		
mult Rs, Rt	multiplication	$Hi \leftarrow (R_s \times R_t)_{63..32}$ $Lo \leftarrow (R_s \times R_t)_{31..0}$	R
multu Rs, Rt	unsigned multiplication	$Hi \leftarrow (R_s \times R_t)_{63..32}$ $Lo \leftarrow (R_s \times R_t)_{31..0}$	R
div Rs, Rt	division	Hi ← Rs mod Rt Lo ← Rs / Rt	R
divu Rs, Rt	unsigned division	Hi ← Rs mod Rt Lo ← Rs / Rt	R
	Logical operations		
Or Rd, Rs, Rt	Logical OR	Rd ← bit-wise Or of Rs and Rt	R
And Rd, Rs, Rt	Logical AND	Rd ← bit-wise And of Rs and Rt	R
Xor Rd, Rs, Rt	Exclusive OR	Rd ← bit-wise Xor of Rs and Rt	R
Nor Rd, Rs, Rt	NOR	Rd ← bit-wise Nor of Rs and Rt	R
Ori Rt, Rs, I	Immediate OR (Immediate extended with zero)	Rt ← bit-wise Or of Rs and I	I
Andi Rt, Rs, I	Immediate AND (Immediate extended with zero)	Rt ← bit-wise And of Rs and I	I
Xori Rt, Rs, I	Immediate Exclusive OR (Immediate extended with zero)	Rt ← bit-wise Xor of Rs and I	I
	Shift operations		
Sllv Rd, Rt, Rs	**Shift Left Logical Variable** Rt is left-shifted according to the 5 LSb of Rs with zeros added in the LSb	Rd ← Rt « Rs	R
Srlv Rd, Rt, Rs	**Shift Right Logical Variable** Rt is right-shifted according to the 5 LSb of Rs with zeros added in the MSb	Rd ← Rt » Rs	R
Srav Rd, Rt, Rs	**Shift Right Arithmetic Variable** Rt is right-shifted according to the 5 LSb of Rs The sign bit of Rt is added in the MSb	Rd ← Rt » Rs	R
Sll Rd, Rt, sh	**Shift Left Logical** Rt is left-shifted according to the value of sh with zeros added in the LSB	Rd ← Rt « sh (sh= shift amount = shamt)	R
Sra Rd, Rt, sh	**Shift Right Arithmetic** Rt is right-shifted according to the value of sh The sign bit of Rt is added in the MSb	Rd ← Rt » Rs	R
	Conditional test operations (set if less than)		
Slt Rd, Rs, Rt	Set if Less Than	Rd <- 1 if Rs<Rt else 0	R
Sltu Rd, Rs, Rt	Set if Less Than Unsigned	Rd <- 1 if Rs < Rt else 0	R
Slti Rt, Rs, I	Set if Less Than Immediate (sign extended Immediate)	Rt <- 1 if Rs < I else 0	I
Sltiu Rt, Rs, I	Set if Less Than immediate (unsigned immediate)	Rt <- 1 if Rs < I else 0	I

TABLE 1.4: Table of Arithmetic and Logical Operations of MIPS R3000

Syntax	Operation	Effect	Format
	Load instructions		
Lw Rt, I(Rs)	**Load Word**	Rt ← M(Rs + I)	I
	Four memory bytes are loaded from address and placed in register Rt		
Lh Rt, I(Rs)	**Load Half Word**	Rt ← M(Rs + I)	I
	- The 2 memory bytes are loaded into the LSB of Rt		
	- The sign bit of the loaded bytes is extended to the remaining bits		
Lhu Rt, I(Rs)	**Load Half Word Unsigned**	Rt ← M(Rs + I)	I
	- The 2 memory bytes are loaded into the LSB of Rt		
	- The other bits are set to zero		
Lb Rt, I(Rs)	**Load Byte**	Rt ← M(Rs + I)	I
	- The memory byte is loaded into the LSB of Rt		
	- The sign bit of loaded byte is extended to the remaining bits		
Lbu Rt, I(Rs)	**Load Byte Unsigned**	Rt ← M(Rs + I)	I
	- The memory byte is loaded into the LSB of Rt		
	- The other bits are set to zero		
	Store instructions		
Sw Rt, I(Rs)	**Store Word**	M(Rs + I) ← Rt	I
	The value in register RT is stored in memory starting at the address Rs + I		
Sh Rt, I(Rs)	**Store Half Word**	M(Rs + I) ← Rt	I
	The 2 less significant bytes (16 bits) of Rt are stored in memory		
Sb Rt, I(Rs)	**Store Byte**	M(Rs + I) ← Rt	I
	The less significant byte (8 bits)of Rt is stored in memory		

TABLE 1.5: Table of memory instructions of MIPS R3000

Syntax	Operation	Effect	Format
Rfe	**Restore From Exception**	$SR \leftarrow SR_{31..4} \parallel SR_{5..2}$	R
	Privileged instruction		
	Restore the previous IT mask and mode		
Break n	**Breakpoint Trap**	$SR \leftarrow SR_{31..6} \parallel SR_{3..0} \parallel "00"$	R
	Branch to exception handler	$PC \leftarrow "80000080"$	
	n defines the breakpoint number	$CR \leftarrow cause$	
Syscall	**System Call Trap**	$SR \leftarrow SR_{31..6} \parallel SR_{3..0} \parallel "00"$	R
	Branch to exception handler	$PC \leftarrow "80000080"$	
		$CR \leftarrow cause$	
Mfc0 Rt, Rd	**Move From Control Coprocessor**	$Rt \leftarrow Rd$	R
	Privileged instruction		
	The register Rd of the control Coprocessor is moved into the integer register Rt		
Mtco Rt, Rd	**Move To Control Coprocessor**	$Rd \leftarrow Rt$	R
	Privileged instruction		
	The integer register Rt is moved into the register Rd of the Control Coprocessor		

TABLE 1.6: Table of system instructions of MIPS R3000

1.3 MIPS R3000 instruction operands

MIPS R3000 instructions use three types of operands that are registers, immediates and labels:

- Registers: used by instructions in formats R and I;
- Immediates: used by instructions in format I ;
- Labels: used by instructions in formats I and J ;

1.3.1 General purpose registers

Registers are a limited amount of memory which exists on the CPU. No data can be operated on the CPU that is not stored in a register. Data from memory, the user, or disk drives must first be loaded into a register before the CPU can use it [6].

In the MIPS CPU, there are only 32 registers, each of which can be used to store a single 32 bit values. Because the number of these registers is so limited, it is vital that the programmer use them effectively.

The conventions for using these registers are outlined below. Note that in some special situations, the registers will take on special meaning, such as with exceptions. These special meanings will be covered when they are needed in the text.

Mnemonic or symbolic name	Register Number	Name or designation
$zero	$0	Zero register
$at	$1	Assembler Temporary register
$v0-$v1	$2-$3	Value registers
$a0-$a3	$4-$7	Argument registers
$t0-$t7	$8-$15	Temporary registers
$t8-$t9	$24-$25	Additional Temporary registers
$s0-$s8	$16-$23	Saved registers
$k0-$k1	$26-$27	Kernel registers
$sp	$29	Stack Pointer
$gp	$28	Global Pointer
$ra	$31	Return Address register

TABLE 1.7: Register Conventions in MIPS R3000

- $zero ($0) **Zero register** : a special purpose register that always contains a constant value of 0. It is a read-only register that cannot be modified.

- $at ($1) **Assembler Temporary** : is often used as a temporary register (e.g. for pseudo instructions) by the assembler. Therefore, this register is not available for use by the programmer.
- $v0-$v1 ($2-$3) **Value Registers** : they are normally used for return values for subprograms. When a function returns a result, it is usually placed in one of these registers. So, $2 (v0) is commonly used for this purpose.
  $v0 is also used to input the requested service to syscall.
- $a0-$a3 ($4-$7) **Argument Registers**: they are used to pass arguments (or parameters) into subprograms. When a function is called, the arguments are placed in these registers. If more arguments are needed, they are passed on the stack.
  Example: $4 contains the integer to be read in a keyboard read operation.
- $t0-$t9 ($8-$15, $24-$25) **Temporary Registers** : they are used to store temporary variables. These registers are not preserved across function calls. They are used to hold values during computations or for temporary storage.
- $s0-$s8 ($16-$23) **Saved Registers** : are used to store saved values. These registers are typically saved by the called function at the beginning of a function and restored before returning, ensuring that their values remain intact across function calls.
- $k0-$k1 ($26-$27) **Kernel Registers** : that registers are reserved by the operating system and are not available to the programmer.
- $gp ($28) **Global Pointer**: Points to the middle of the data segment in memory. It is used to access global and static data variables efficiently.
- $sp ($29) **Stack Pointer**: Used to keep track of the beginning of the data for this method in the stack and to manage stack operations such as pushing and popping values.
- $fp ($30) **Frame Pointer**: used with the $sp for maintaining information about the stack. It points to the base of the current stack frame.
- $ra ($31) **Return Address**: A pointer to the address to use when returning from a subprogram. Used to store the address to return to after a function call. The jal instruction sets this register, and the jr $ra instruction uses it to return.

NB: In addition to the 32 general-purpose registers, there are the **HI** and **LO** registers, which are used for multiplication and division operations. These two 32-bit registers store the result of a multiplication or division, which produces a 64-bit result.

1.3.2 Immediate values

An immediate is a constant value that is encoded directly in a format "I" MIPS instruction. Instructions using immediate allow to perform operations with a fixed value without having to load it from memory. Immediate values are generally integers, and are directly included in the machine code.
Characteristics of MIPS immediates

- Immediate values are 16-bit integers (2 bytes).
- Its value can be positive, negative or null.
- MIPS instructions that use immediates often end with the letter "i" to indicate a version with an immediate constant (e.g. addi, andi, ori, etc.).
- An immediate expressed in decimal is written as is.
- An immediate expressed in hexadecimal must be prefixed with "0x".
- For arithmetic and logic instructions with immediates (such as addi), an extension of the immediate must be applied in order to perform the operation with the second operand, which is often a 32-bit register. So 16 bits are added to the upper part of the immediate, depending on the type of operation applied and the sign of the immediate (see Section 1.2.3).

1.3.3 Labels

A label is an identifier followed by a colon (:). It marks either a data section or an instruction in the program. When the assembler runs the code, it replaces the label with the actual memory address or instruction address.
Labels are typically used for two general purposes:
- **Instruction labels**: used to indicate the position of an instruction so that it can be used as an operand of a branching or jumping instruction.
- **Data labels**: used to identify memory regions in the data segment, providing for quick access to variables and constants. In this case, the label value will be accessed using the macro "la (load address)".

Specific writing rules : A correct identifier must conform to the following rules :

- Starts with a letter
- No reserved words
- No special symbols (except underscore _)

- Must end with a :
- Is case sensitive
- Must have reasonable length.

Example:

```
.data
        X: .word 20    # X is a label declared in data section
```

1.4 SYNTAX of a MIPS R3000 assembly program

1.4.1 Program structure

A MIPS R3000 program consists of two main parts (see Figure 1.3):

- The declarative part: Declared by the **data** directive.
- The instruction part: Declared by the **text** directive.

NB: A program may not contain a data section.

```
.data
        # Variable declaration
        # ...
        # ...
        # ------------------------------------
.text
        # Area reserved for assembly instructions
_start:
        # Main program
        # ...
        # ...
        # ------------------------------------
        ori  $2, $0, 10 # Exit
        syscall
```

FIGURE 1.3: Overall structure of a MIPS R3000 program

The code resides in the **.text** section, and the main program typically starts with the **main** label or after **_start**. The label main indicates the place to begin execution. It does not need to be included as the program begins at the first line in the assembled program.

Example 1:

```
.text
main:
        addi $10, $0, 0x1234   # Load immediate value 0x1234 into $10
        addi $2, $0, 10        # Load immediate value 10 into $2
        syscall                # Exit
```

Example 2:

```
.text
_start:
        addi $10, $0, 0x1234   # Load immediate value 0x1234 into $10
        addi $2, $0, 10        # Load immediate value 10 into $2
        syscall                # Exit
```

NB: in MIPS, instructions are simply separated by new lines, and no special punctuation is needed between them.

1.4.2 Comments

Comments in MIPS assembly start with the # symbol or ; and continue to the end of the line.
Examples:

```
.text
_start:
        add $10, $11, $12   # This is an example of an addition with registers
        add $10, $11, $12   ; This is an example of an addition with registers
        addi $2, $0, 10     # Load immediate value 10 into $2
        syscall             # Exit
```

1.4.3 Directives

A directive in MIPS R3000 is an instruction that is not executed by the processor but is used by the assembler to configure various aspects of the program or the assembly process. These directives are also called pseudo-instructions or assembly directives, and they do not generate machine code. Instead, they provide instructions to the assembler about how to organize, store, or process the code [5].

In MIPS and most assembly languages in general, a "." before a text string indicates that the token (string) following it is an assembler directive.
The common directives used in MIPS R3000 are the following:

- **.text** directive: indicates that the instructions that follow are part of a program text (i.e. the program) and will be stored in the text section of memory. This is where the assembler places machine instructions.
- **.data** directive: implies that the following is program data (such as global variables, tables, etc.), which will be placed in memory's static data section.
- **.word** directive: Reserves one or more words (one word = 4 bytes in MIPS) in memory for storing data, typically used for defining variables or initializing values.
  NB: Be careful as it is incorrect to think of a the .word directive as a declaration for an integer, as this directive simply allocates and initializes 4 bytes of memory, it is not a data type. What is stored in this memory can by any type of data [6].
  Examples:

```
.data
    x : .word 20                   # Reserves 4 bytes of memory at the label x and
                                   initializes them with the value 20
    Tab: .word 10, 5, -3, 0x45A    # Reserves 16 bytes at the label Tab and initializes
                                   them with values 10, 5, -3, 0x45A in order
```

- **.byte** directive: reserves 1 byte (8 bits) of memory and allows to initialize it with a value.
  Examples:

```
.data
    x : .byte 20           # Reserves 1 byte at label x and initializes it with the value 20
    y : .byte 5, 10, 15    # Reserves 3 bytes in memory from address y and stores
                           the values 5, 10 and 15 respectively.
```

- **.half** directive reserves 2 bytes (16 bits) of memory and allows to initialize it with a value.
  Example:

```
.data
    y: .half 100    # Reserves 2 bytes at label y and initializes them with the value 100
```

- **.space n** directive: allocates **n** bytes of memory in the data region of the program without initializing them. It is typically used in the **.data** section to reserve memory for variables, arrays, or buffers where the initial content does not matter or will be set later.
  Example :

```
.data
        Tab: .space 40    # Reserves 40 bytes from address Tab
```

- **.ASCII** and **.ASCIIZ** directive: In MIPS assembly, a string is a sequence of ASCII characters which are terminated with a null value (a null value is a byte containing 0x00). Thus when handling strings, an extra byte must always be added to include the null terminator [6]. This is also the reason for the assembler directives .ascii and .asciiz:
  - The **.ascii** directive only allocates the ASCII characters but does not add a null terminator. If the user wants to end the string with a null byte, he would have to manually add `\0` to the declaration.
  - The **.asciiz** directive allocates the characters terminated by a null. So the .asciiz allocates a string.

  Example :

```
.data
    msg1: .ascii "Hello world!"     # The declared string does not end with a zero
    msg2: .asciiz "Hello world!"    # The declared string ends with a zero
```

- **.float** directive: Used to define single-precision (32-bit) floating-point constants in memory. It is used to reserve space for one or more floating-point numbers and assign them an initial value.

```
.data
    x : .float 20.5        # Reserves 4 bytes at label x and initializes them with
                           # the value 20.5
    y : .float 5.6, -2.4   # The floating-point values 5.6, -2.4 are stored in memory
                           from the label y , each occupying 32 bits.
```

- **.double** directive: Used to define double-precision (64-bit) floating-point constants in memory.

```
.data
    x : .double 20.5        # Reserves 8 bytes at label x and initializes them with
                            # the value 20.5
    y : .double 5.6, -2.4   # The floating-point values 5.6, -2.4 are stored in memory
                            from the y label, each occupying 64 bits.
```

Summary example: x, y and z are integer variables. They can be declared as follows:

```
.data
        x : .word 0x12345678
        y : .half 0xA345
        z : .byte 5
        tab : .byte -1,5,20,11
```

NB: The last declaration allows values (-1, 5, 20, 11) to be arranged in consecutive bytes in memory, starting at address tab, and can be used to declare an array of integers.
-The memory schema (hexadecimal representation) obtained after these declarations is shown in Figure 1.4:

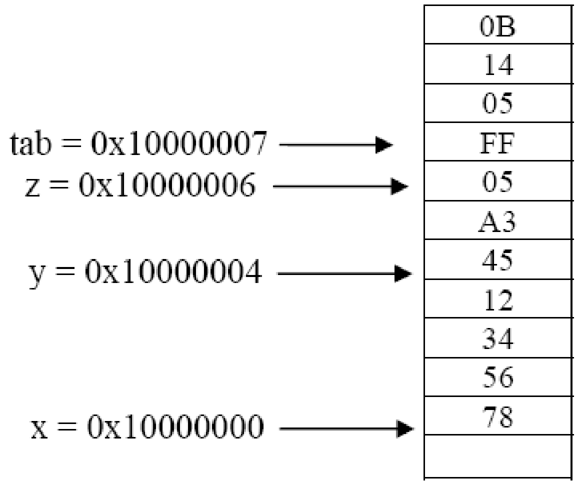

FIGURE 1.4: An example of data representation in memory

- **.align** directive: Used to ensure that data is aligned to a specific boundary in memory. This can improve access efficiency, especially when dealing with larger data types or when the hardware requires data to be aligned to certain boundaries.

- **.globl** or **.global** directive: is generally used to define the entry point of a program, or to share variables or functions between several files in a project.

1.4.4 Macro instructions

A macro-instruction is a single command that expands into a set of instructions, simplifying repetitive tasks and reducing code complexity. During assembly or compilation, the macro is expanded to generate the code corresponding to the instructions it encapsulates. MIPS R3000 implements the following macro instructions:

- **Li** : The li (**load immediate**) macro is used to load a constant value into a register. Since the MIPS architecture does not support loading large immediate values in a single instruction, li is a macro that can expand into multiple instructions, to handle larger immediate (32 bits).

- **La** : The la (**load address**) macro loads the memory address of a label into a register. Since MIPS lacks a direct load address instruction, this is a pseudo-instruction that combines two basic instructions (lui and ori) to load a 32-bit address into a register.

A detailed description of these macros is provided in Table 1.8.

Assembly syntax	Operation	Effect	Corresponding code
la Rd, $Label_{31..0}$	Load Address	Rd $\leftarrow$ $Label_{31..0}$	$lui Rd, Label_{31..16}$ $ori Rd, Rd, Label_{15..0}$
li Rd, $Imm_{31..0}$	Load Immediate	Rd $\leftarrow$ $Imm_{31..0}$	$lui Rd, Imm_{31..16}$ $ori Rd, Rd, Imm_{15..0}$

TABLE 1.8: Table of macro instructions of MIPS R3000

1.4.5 Input/Output operations

There is a second way to read/write data to/from a register. If the data to be accessed is on an external device, such as a user terminal or disk drive, the syscall instruction is used. The syscall operator allows the CPU to talk to an I/O controller to retrieve/write information to the user, disk drive, etc [6].

The **'syscall'** instruction is used by a user program to make a "system call" in order to perform certain actions that require operating system control, such as input/output tasks like reading or writing a string or number to the console. It is common to pass any arguments through registers **$4** and **$5**, and to store the system call number in register **$2**. There are five primary system calls available that are detailed in Table 1.9.

Service	Code	Arguments	Return Value
Display integer	1	$4: stores the desired integer	
Display string	4	$4: address of the desired string	
Read integer	5		$2: the read integer
Read character	8	$4: address of the buffer $5: max number of characters to read	
Exit	10		

TABLE 1.9: Descriptive table of MIPS R3000 system calls

Methodology

To execute a system call, the following steps must be followed:

1. Write the instruction to load the desired service into register **$2**.
2. Write the instructions to load arguments into registers **$4** and/or **$5** (case of displaying operation).
3. Write the instruction **syscall**.

4. Write the instructions to retrieve a return value from the **syscall** (case of reading operation).

1.5 Conclusion

This part provided details of the external architecture of the MIPS R3000 processor, covering visible registers, memory addressing rules and various available instructions. Here we have sequentially presented the memory organization and the addressing mechanism, the main syntax rules of the language, assembler supported instructions, macro instructions, and the available system calls.

Part 2

Presentation of the working environment: SIMIPS emulator

2.1 Introduction

This second part provides an overview of the tools and procedures essential for writing, assembling, and running programs in MIPS assembly language within the 'SIMIPS emulator' environment. It begins with a presentation of the tools required to develop assembly language programs, detailing each component: a text editor, assembler, linker, loader, and debugger. This section concludes by citing some of the most commonly used IDEs for MIPS R3000 assembly programming, which are predominantly used for educational purposes.

The SIMIPS software is an emulator designed to simulate the R3000 MIPS processor, a 32-bit architecture often used for educational purposes. Developed at the University of Pierre and Marie Curie, SIMIPS allows users to explore the internal workings of the MIPS architecture, focusing on instruction execution and system calls. This emulator is particularly valuable for students and researchers exploring computer architecture, as it facilitates hands-on learning of processor operations at a low level.

The last section of this part introduces the 'SIMIPS emulator', guiding readers through its main interfaces and key functionalities. Practical instructions on initializing the emulator, entering code in the editor, assembling and loading the program, and executing the final code are provided through a step-by-step approach. This structured introduction equips users with the foundational knowledge to effectively utilize the SIMIPS environment for MIPS assembly programming.

2.2 Tools needed to develop and run a program written in assembly language

Program development, from initial problem analysis to final debugging and implementation, involves a wide range of software tools that facilitate different stages of the process. These tools form what is known as a programming environment, which supports tasks such as coding, testing, and debugging.

2.2.1 Text Editor

A text editor is an interactive software that allows users to input text from a keyboard and store it in a file. The information stored in the file is plain text in ASCII code. The main functions of a text editor include displaying part of the text on the screen, moving and positioning the cursor, editing the text by inserting, deleting, or replacing characters, and searching for specific strings of text.
The source code for an assembly program is entered using a text editor and typically has the file extension **.s** or **.asm**.

2.2.2 The Assembler

An assembler is a translation program that converts assembly source code into machine language. The resulting object program is saved in a file with the .obj extension (object file). During the assembly process, each instruction in the source code is translated into its corresponding machine instruction (binary code) [9]. The assembler performs the following key functions:

- **Translation:** Converts assembly language instructions into corresponding machine code.
- **Address assignment:** Generates relative addresses for instructions and data in the object file.
- **Symbol table creation:** Builds a symbol table that maps labels to their respective addresses.
- **Object file generation:** Produces an object file (e.g., `.obj`) containing the machine code and unresolved references.
- **Error detection:** Identifies syntax errors and issues in the assembly code during the translation process.

2.2.3 The linker [2]

The .obj file contains the binary output of the assembly process, but it is not usable in its current form; the system cannot load or execute it. The linker is a crucial component in the software development process that combines one or more object files generated by an assembler or compiler into a single executable file. It serves to ensure that all the program's code and data can be correctly referenced and executed. It performs the following key functions:

- **Combining object files:** Merges multiple object files into one executable.
- **Resolving references:** Links external symbols and functions from different files.
- **Address assignment:** Assigns absolute memory addresses to instructions and data.
- **Creating executable:** Generates the final executable file (e.g., `.exe`).
- **Optimization:** May eliminate unused code and optimize memory layout.

2.2.4 The loader

A program can only be executed if it is loaded into main memory. The component responsible for this task is called the loader. A special utility of the operating system is responsible for reading the executable file, loading it into main memory, and then launching the program. The main functions of the loader are the following:

- **Reading executable files:** Reads the executable file (`.exe` or `.bin` formats).
- **Memory allocation:** Allocates memory space in the main memory for the program's code, data, and stack.
- **Loading into memory:** Copies the program's machine code and data from the executable file into the allocated memory space.
- **Relocation:** Adjusts memory addresses in the program as necessary, especially if the program is not loaded at its preferred memory address.
- **Launching execution:** Transfers control to the program, initiating its execution from the entry point defined in the executable file.

2.2.5 The debugger

The debugger is a software tool that facilitates the debugging of programs. It allows users to examine the contents of registers and perform memory dumps. This enables step-by-step execution of a program, that is, instruction by instruction, which helps in understanding what happens during execution.The main functions of the debugger are the following:

- **Step-by-Step Execution:** Allows the user to execute a program one instruction at a time to observe its behavior and state at each step.
- **Breakpoints:** Enables users to set breakpoints, which pause the execution of the program at specified lines of code, allowing for detailed inspection.
- **Variable Inspection:** Allows examination and modification of variable and register values during execution, aiding in the detection of logical errors.
- **Memory Dumping:** Allows for the dumping and inspection of memory contents, which helps to analyze how data is stored and manipulated.
- **Call Stack Navigation:** Displays the call stack, enabling users to see the function call hierarchy and trace the flow of execution through the program.
- **Error Detection:** Assists in identifying and diagnosing runtime errors, such as segmentation faults, infinite loops, and unhandled exceptions.

2.2.6 Popular IDEs and simulators for MIPS R3000 Programming

There are a number of MIPS simulators available, some for educational use, and some for commercial use. Those simulators allows students and researchers to run MIPS assembly code on platforms where a physical MIPS processor is unavailable, making it an essential learning tool for those studying computer architecture and assembly programming. Below, we list some of the most commonly used simulators for MIPS R3000 assembly programming:

- **SIMIPS** fo SImple MIPS is an emulator designed specifically for the MIPS R3000 processor. Known for its accurate representation of the MIPS R3000 instruction set and architecture, SIMIPS allows students to explore the mechanics of low-level assembly programming. Its straightforward setup and realistic emulation make it suitable for learning assembly language fundamentals in a way that closely resembles real-world applications.

- **MARS** for MIPS Assembler and Runtime Simulator [11] is a lightweight interactive development environment for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's Computer Organization and Design.
- **SPIM** [7] for Simulator for Processing Interactive MIPS is a widely-used open-source simulator developed by Dr. James R. Larus in the late 1980s to emulate the MIPS R2000/R3000 processors for educational purposes. is a self-contained simulator that runs MIPS-32 assembly language programs. SPIM also provides a simple debugger and minimal set of operating system services. It provides a clear view of how MIPS instructions function on a simplified processor model.
- **QtSPIM** [7] for Qt framework with the SPIM simulator is the newest version of Spim, and unlike all of the other version, it runs on Microsoft Windows, Mac OS X, and Linux. It maintains compatibility with the original SPIM but adds a graphical display for memory and registers, as well as a streamlined user experience. It's a popular choice for students, as it provides the same accurate simulation while being easier to navigate.

This document uses the SIMIPS emulator. Based on our experience, we chose the SIMIPS tool because it most closely matches the MIPS R3000 version studied in our courses. Unlike other environments that implement instruction sets from more advanced versions of MIPS, SIMIPS accurately represents the instruction set, memory layout, processor registers, and immediate value formats of the MIPS R3000 processor.

2.3 Getting started with the SIMIPS emulator and presentation of the main interfaces

The various steps that are required in order to write and run an assembly language program are the following :

1. Install a simulator (SIMIPS or MARS).
2. Create a new file and write the code above.
3. Assemble the code.
4. Load the program in main memory.
5. Run the program.

2.3.1 Launching the software

On the desktop, click on the icon corresponding to the MIPS R3000 microprocessor. The following window will appear:

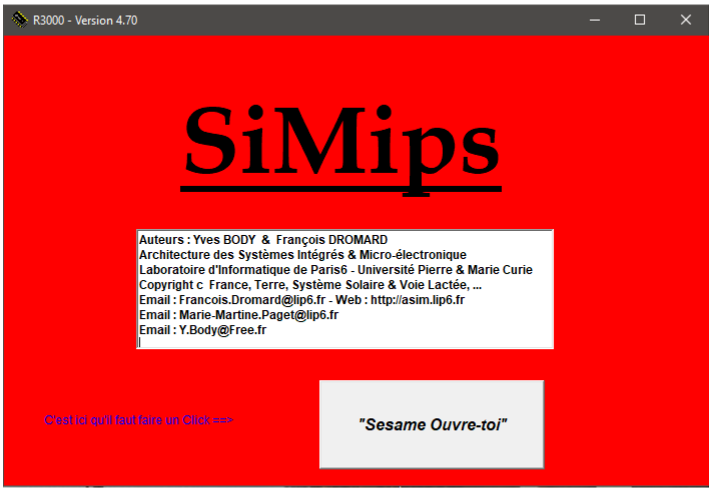

FIGURE 2.1: SIMIPS launching interface

Click on 'Sesame ouvre-toi,' which will allow you to open the emulator and access all its separated windows (see Figure 2.2).

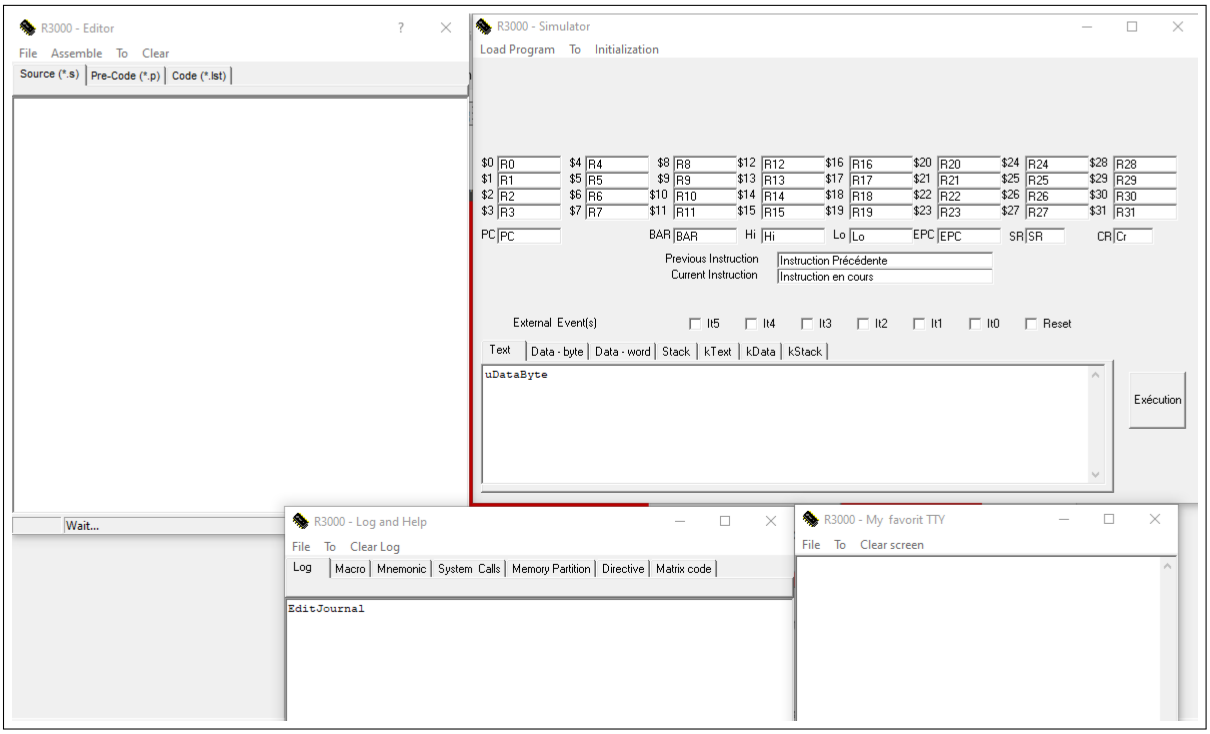

FIGURE 2.2: SIMIPS windows

2.3.2 Step 1: Entering the program in the editor (R3000-Editor)

1- Click on the window named (R3000-Editor) shown in Figure 2.3.

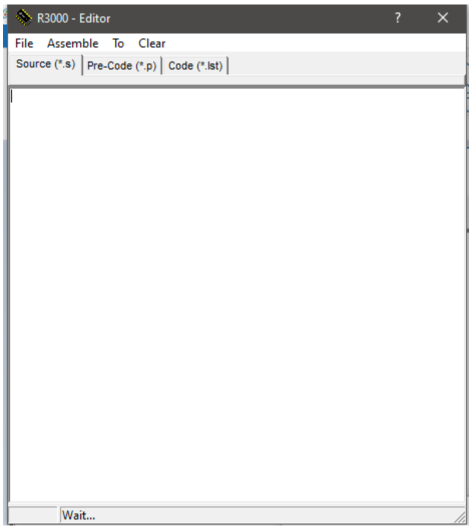

FIGURE 2.3: SIMIPS editor interface

2- Enter the following program in the source of the editor (R3000-Editor).

```
.text
_start:
    addi $17, $0, 5       # First assembly instruction
    addi $18, $0, 10      # Second assembly instruction
    add $18, $18, $17     # Third assembly instruction
    addi $2, $0, 10       # Fourth assembly instruction
    syscall               # Last assembly instruction
```

2.3.3 Step 2 : Assembly and code generation

Still in the window corresponding to the editor:

- First click on the **'Assemble'** menu,
- Then on **'Generate Code + Assembler'** as shown in Figure 2.4

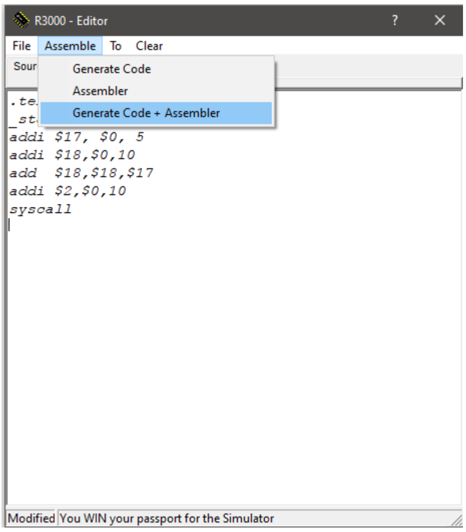

FIGURE 2.4: Assembly and code generation interface

Note: Two scenarios may arise

- Case 1: Your program contains errors

Example: Remove the $ symbol preceding the value 17 on the third line of the previous code. The assembler will then display the following message:

```
You're unlucky!  Revise the Source Text
```

Below the erroneous line, an error message is shown with arrows indicating the mistake. For example:

```
One operand is missing:  17.  A register is necessary
```

This error message means that you need to change the operand 17 to a register by adding the $ symbol before it, resulting in: `add $18, $18, $17`.

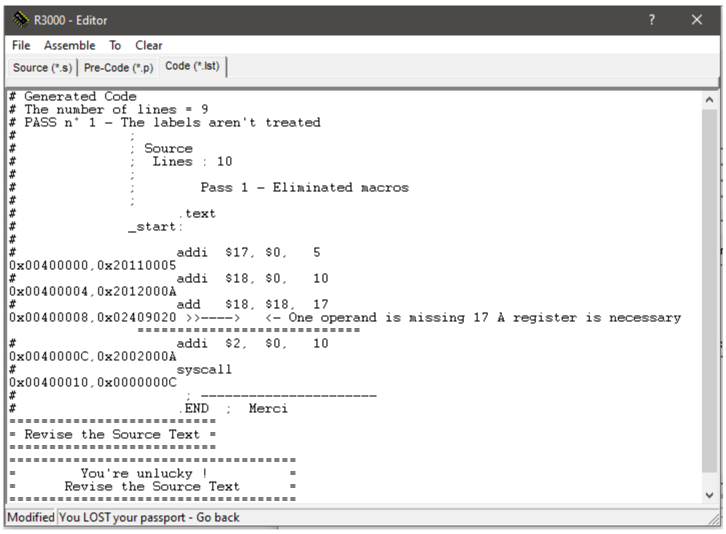

FIGURE 2.5: Error message interface

- To correct the errors, proceed as follows:

1. Click on **Source** in the editor window ;
2. Make the necessary changes to correct errors ;
3. Click **Assemble**, then **Generate code + Assembler** ;
4. Repeat 1, 2 and 3 until the success message is displayed.

Case 2: Your program contains no errors
The message **"Second Pass — Success — Bravo!"** will be displayed, as shown in Figure 2.6.

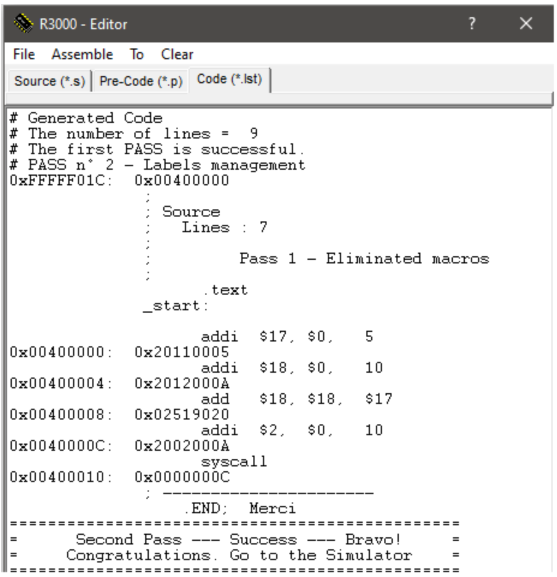

FIGURE 2.6: Success message interface

During this phase, the assembler associates two pieces of binary information with each instruction, expressed in hexadecimal (prefixed with 0x) and separated by a colon (:). This information corresponds to the instruction's load address in main memory and the machine code of the assembler instruction, respectively.
Example:
For the instruction **addi $17, $0, 5**, the assembler generates 0x00400000 : 0x20110005.
0x00400000 : Loading address of the instruction addi $17, $0, 5 in main memory;
0x20110005: Machine code (hexadecimal format) of the instruction addi $17, $0, 5.

2.3.4 Step 3: Loading the program into the simulator

- In the editor:

- First click on **'To'**
- Then **'Simulator'** as shown in Figure 2.7.

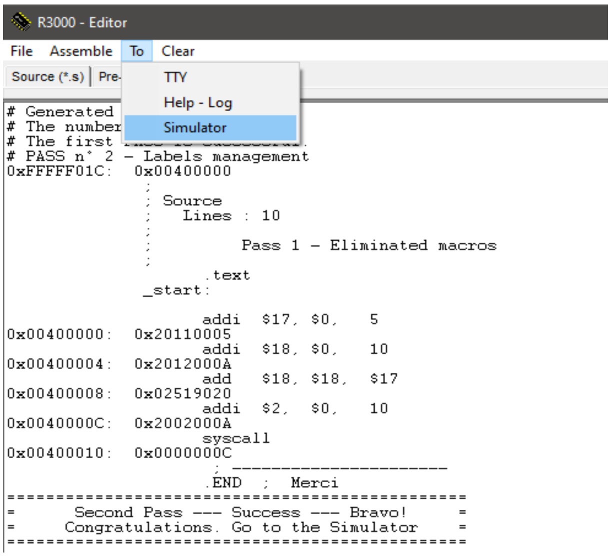

FIGURE 2.7: Access interface to the simulator from the editor

The simulator allows to visualize the components of the physical R3000 machine. It displays the main memory, the registers of the processing unit, and those of the control unit, as shown in Figure 2.8.

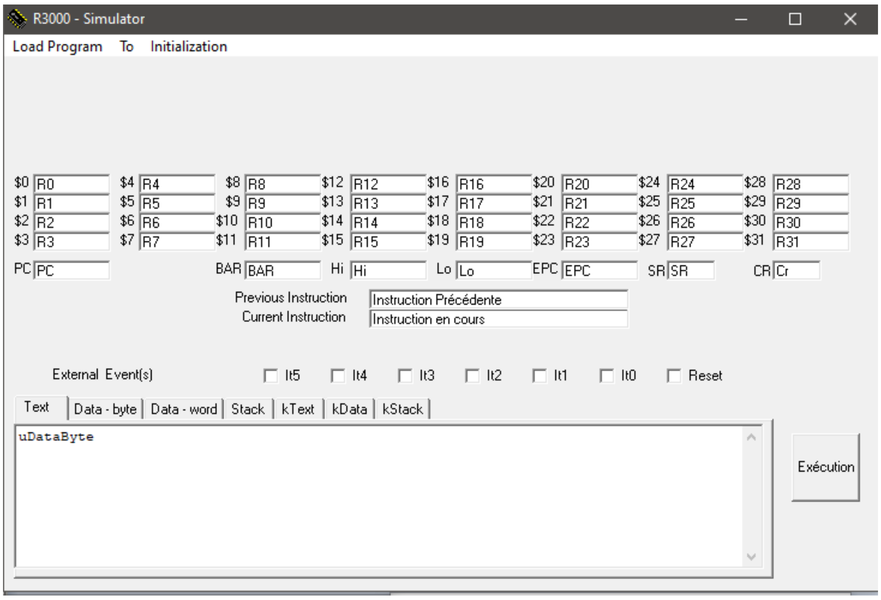

FIGURE 2.8: Simulator interface

- In the simulator window (R3000 Simulator):

- Click on **Load Program** ;
- Select **Generated Code in the Editor window** (see Figure 2.9).

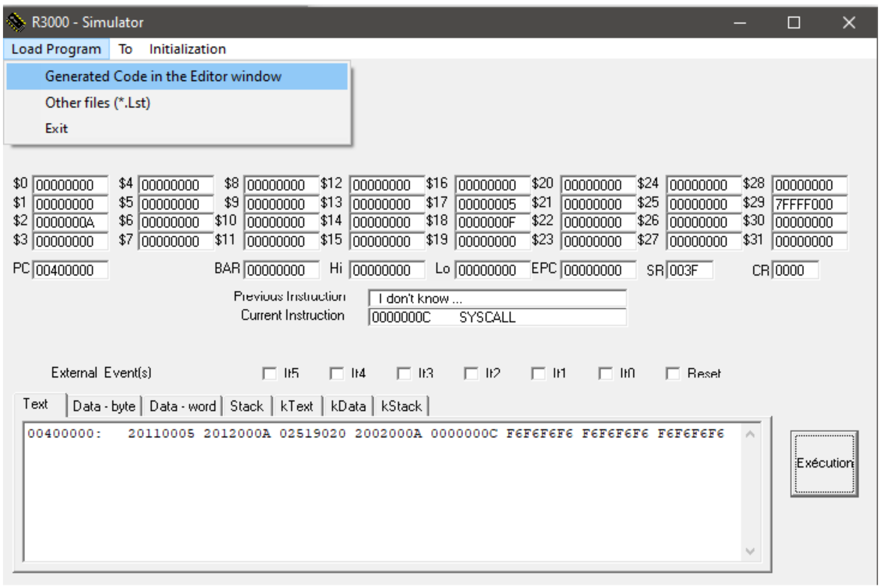

FIGURE 2.9: Loading program interface

2.3.5 Step 4: Executing the Program

The execution of the assembly program occurs step by step (i.e., instruction by instruction) by clicking the **Exécution** button.

Note: When you click on **Exécution**, the instruction being executed is shown in ***Previous Instruction*** (here, `addi $17, $0, 0x0005`), and the instruction displayed in the ***Current Instruction*** register will be the next to execute (here, `addi $18, $0, 0x000A`) (see Figure 2.10).
In this example, after executing `addi $17, $0, 0x0005`, the value of $17 is set to 5.

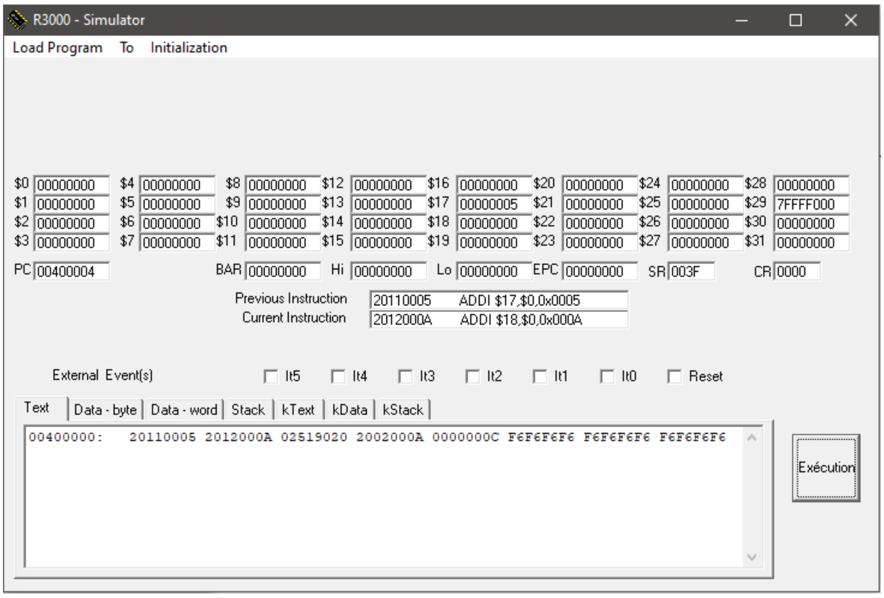

FIGURE 2.10: Executing program interface

Very Important Note: For each new execution of a program, you must first reload it and then reset the user registers. To do this:

- Click on **Initialization** ;
- Select **User Registers**, as shown in Figure 2.11.

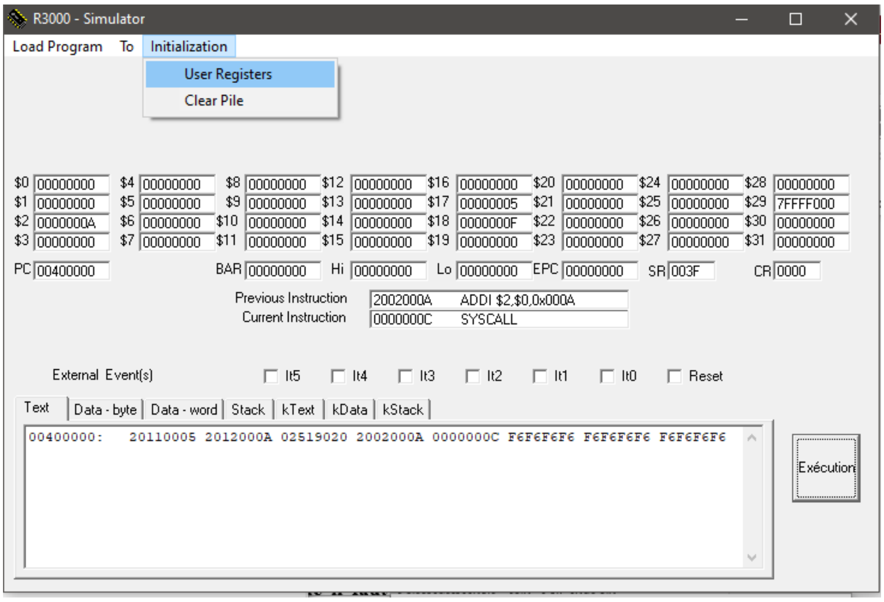

FIGURE 2.11: Registers initialization interface

Thus, all general registers will be reset to zero, and the other registers will be reinitialized. The program counter will point to the first address of the program to be executed. - If the program uses the memory stack, it must also be reinitialized by clicking on **Initialization**, then **Clear Pile**.

2.4 Conclusion

In this part, we presented a comprehensive user guide to help students get started with the SIMIPS emulator, serving as a practical manual for writing, assembling, and running their first R3000 assembly program. Covering the essential tools and components such as the editor, assembler, linker, loader, and debugger alongside a detailed step-by-step guide to using SIMIPS, this manual equips students with the knowledge needed to develop and test MIPS R3000 programs.

Part 3

Series of practical exercises

3.1 Introduction

Rather than just studying theory, students and researchers can gain real experience by practicing and observing how the processor functions through direct application. This part offers practical exercises to reinforce MIPS R3000 assembly language concepts. It presents a series of structured, practical exercises to enhance comprehension of fundamental MIPS assembly language instructions and techniques.

With particle work (PW) N°1, students will begin by entering, assembling, and executing their first MIPS program using the R3000-Editor and simulator, establishing the essential workflow of writing and running assembly code.

In PW N°2, the focus shifts to arithmetic and logical instructions, where students will practice key operations, including an example of an arithmetic operation, an example of a logical operation, the "lui" (load upper immediate) instruction, with some comprehension exercises to apply these instructions, such as loading an immediate into a register.

PW N°3 introduces input/output instructions, enabling students to work with system calls for basic I/O tasks like writing integers and displaying strings, while PW N°4 guides students through memory operations, such as load and store, building familiarity with memory access and management.

Finally, PW N°5 covers conditional and unconditional branching, allowing students to explore decision-making constructs, such as "if..then" statements and "while" loops.

This progression of exercises builds foundational skills in MIPS assembly language, providing essential tools and techniques for low-level programming and algorithm implementation.

3.2 PW N°1 : Write and execute your first MIPS R3000 assembly program

3.2.1 Step 1: Enter the following program in the editor (R3000-Editor)

```
.text
_start:
    addi $17, $0, 5       # First assembly instruction
    addi $18, $0, 10      # Second assembly instruction
    add $18, $18, $17     # Third assembly instruction
    addi $2, $0, 10       # Fourth assembly instruction
    syscall               # Fifth assembly instruction
```

3.2.2 Step 2: Assembly and code generation

- Still in the window corresponding to the editor, first click on the **'Assemble'** menu, then on **'Generate Code + Assembler'**.
- Make sure the message **"Second Pass — Success — Bravo!"** is displayed, indicating that the assembly was successfully completed and the machine code for your program has been generated.
- Otherwise, the message **"You're unlucky! Revise the Source Text"** will be displayed to indicate that your program contains errors, which will be highlighted with **arrows**. Review and correct your code, then reassemble and generate the code again until the success message is displayed.
- During this phase, the assembler associates two pieces of binary information with each instruction, expressed in hexadecimal (prefixed with 0x) and separated by a colon (:). This information corresponds to the instruction's load address in main memory and the machine code of the assembler instruction, respectively.
Example:
For the instruction **addi $17, $0, 5**, the assembler generates **0x00400000 : 0x20110005**.
0x00400000 : Loading address of the instruction addi $17, $0, 5 in main memory ;
0x20110005: Machine code (hexadecimal format) of the instruction addi $17, $0, 5.
Questions: Carefully observe the results of the assembly and code generation, then answer the following questions:

1. What does the information `0x00400004` correspond to?
2. At what address is the instruction `addi $2, $0, 10` loaded?

3. What is the machine code (in hexadecimal form) for the instruction `syscall`?
4. How many bits is a memory address defined on?
5. How many bits is a machine instruction defined on?
6. By which value does the address increment from one instruction to the next?

3.2.3 Step 3: Loading the program into the simulator

In the editor window: Click on **'To'** and then **'Simulator'**.
The simulator provides a view of the components of the physical R3000 machine, including the main memory and the registers of the CPU.

- In the simulator window (R3000 Simulator): Click on **'Load Program'** and then select **'Generated Code in the Editor window'**.

7. What are the contents of registers **$2, $17, $18**, and PC before executing this program?

3.2.4 Step 4: Executing the Program

The execution of the assembly program occurs step by step (i.e., instruction by instruction) by clicking the **Exécution** button.
Questions:

8. Execute the instructions of the program step by step and provide the contents of the registers **$2, $17, $18** and PC at the end of the execution.
9. What does this program do?
10. Remove the last two instructions and re-execute the program by going through steps 2, 3, and 4. What do you notice?
11. Infer the role of the instructions `addi $2, $0, 10` and `syscall` combined.

3.2.5 Comparison between MIPS and Von Neumann registers

Compare the MIPS R3000 simulator with the Von Neumann machine and complete the following table:

Von Neumann Machine	**Program Counter**	**Instruction Register**	**Indicator Register**	**General Registers**	**Accumulator**
MIPS R3000 Simulator					

TABLE 3.1: Comparison between MIPS and a Von Neumann machine registers

3.3 PW N °2: Arithmetic and logical instructions

3.3.1 Exercise 1: Discovery of an example of an arithmetic instruction

Enter the following program, then assemble.

```
.text
_start:
    addi $8, $0, 5            # 5 is a positive immediate value expressed in decimal
    addi $9, $0, - 4          # - 4 is a negative immediate value expressed in decimal
    addi $10, $0, 0x 2259     # 0x2259 is a positive immediate value expressed in hexa
    addi $11, $0, 0x 9234     # 0x9234 is a negative immediate value expressed in hexa
    add $9, $9, $8
    addi $2, $0, 10
    syscall
```

1. Run the program step by step and complete the following table:

Instruction	**$8**	**$9**	**$10**	**$11**
`Before execution`				
`addi $8, $0, 5`				
`addi $9, $0, - 4`				
`addi $10, $0, 0x2259`				
`addi $11, $0, 0x9234`				
`add $9, $9, $8`				

   Analyze the results in this table and answer the following questions:

2. What does the instruction `Addi $8, $0, 5` do ?

3. How is the value 0xFFFFFFFC obtained from the value -4 ?

4. Given that the immediate value used in the instruction `addi $RD, $RS, imm` is defined over 16 bits, deduce the extension applied to this immediate value in the general case.

5. What does the instruction Add $9, $9, $8 do ?

3.3.2 Exercise 2: Discovery of an example of a logical instruction

Enter the following program, then assemble.

```
.text
_start:
    ori $8, $0, 5           # 5 is a positive immediate value expressed in decimal
    ori $9, $0, - 4         # - 4 is a negative immediate value expressed in decimal
    ori $10, $0, 0x 2259    # 0x2259 is a positive immediate value expressed in hexa
    ori $11, $0, 0x 9234    # 0x9234 is a negative immediate value expressed in hexa
    or $9, $9, $8
    addi $2, $0, 10
    syscall
```

1. Run the program step by step and complete the following table:

Instruction	**$8**	**$9**	**$10**	**$11**
`Before execution`				
`ori $8, $0, 5`				
`ori $9, $0, - 4`				
`ori $10, $0, 0x2259`				
`ori $11, $0, 0x9234`				
`or $9, $9, $8`				

   Analyze the results in this table and answer the following questions:

2. What does the instruction `ori $8, $0, 5` do ?

3. Given that the immediate value used in the instruction `ori $RD, $RS, imm` is defined over 16 bits, deduce the extension applied to this immediate value in the general case.

4. What is the difference between instructions or and ori ?

3.3.3 Exercise 3: Discovery of the instruction "lui" (load Upper Immediate)

Enter the following program, then assemble.

```
.text
_start:
        lui $8, 5            # 5 is a positive immediate value expressed in decimal
        lui $9, - 4          # - 4 is a negative immediate value expressed in decimal
        lui $10, 0x2259      # 0x2259 is a positive immediate value expressed in hexa
        lui $11, 0x9234      # 0x9234 is a negative immediate value expressed in hexa
        addi $2, $0, 10
        syscall
```

1. Run the program step by step and complete the following table:

Instruction	**$8**	**$9**	**$10**	**$11**
`Before execution`				
`lui $8, 5`				
`lui $9, -4`				
`lui $10, 0x2259`				
`lui $11, 0x9234`				

Analyze the results in this table and answer the following questions:

2. What does the instruction `lui $10, 0x2259` do ?
3. Given that the immediate value used in the instruction `lui $RD, imm` is defined over 16 bits, deduce how the content of register $RD is obtained.

3.3.4 Exercise 4 (Comprehension Test and some uses of the instructions seen previously)

1. Write the assembly code to load the value `0x1234` into register `$8`.
2. Write the assembly code to load the value `0x9456` into register `$9`.
3. Write the assembly code to load the value `-1` into register `$10`.
4. Write the assembly code to load the value `0x1234F678` into register `$11`.
5. The macro instruction `Li $RD, imm` allows a 32-bit immediate to be loaded into register `$RD`. Knowing that this macro is not part of the MIPS R3000 instruction set, write the corresponding MIPS instructions.

3.4 PW N°3: Input/output instructions

3.4.1 Introduction

To perform input/output operations involving reading or writing a number or a string to the console, the user program must use a system call with the syscall instruction.

By convention, the system call number is contained in register **$2**, and its arguments are in registers **$4** or **$5**. The various system calls are illustrated in the Table 3.2 :

System Call Number	**Corresponding Operation**
1	Displaying the integer in register **$4** on the console
4	Displaying the string whose address is in register **$4**
5	Reading an integer from the keyboard and placing it in register **$2**
8	Reading a string from the keyboard
10	Exiting the program properly

TABLE 3.2: System calls in MIPS R3000

3.4.2 Exercise 1 (Writing an integer: System call Number 1)

Follow the steps below:

1. Write the instruction that places the value of the integer to be displayed (e.g., 0x4567) in register $4.
2. Write the instruction that places the immediate 1 in register $2.
3. Write the system call instruction (`syscall`).
4. Write the instructions to properly terminate the program.
5. Assemble and execute your program.

NB: The result will be displayed in the TTY window. To display it click on 'To' in the simulator and then on 'TTY' (see Figure 3.1).

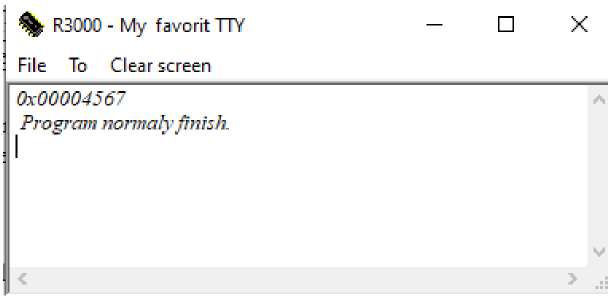

FIGURE 3.1: Window for displaying an integer on screen

3.4.3 Exercise 2 (Reading an integer from the keyboard: System call Number 5)

Follow the steps below:

1. Write the instruction to place the value 5 in register $2.
2. Write the system call instruction (`syscall`).
3. Write the instruction to transfer the read value (currently in register $2) to another register (e.g. $8).
4. Write the instructions to properly terminate the program.
5. Assemble and execute your program. After executing the syscall instruction, the following dialog box will appear, allowing you to enter the value from the keyboard (see Figure 3.1).
   NB: The value entered via the keyboard must be expressed in hexadecimal.

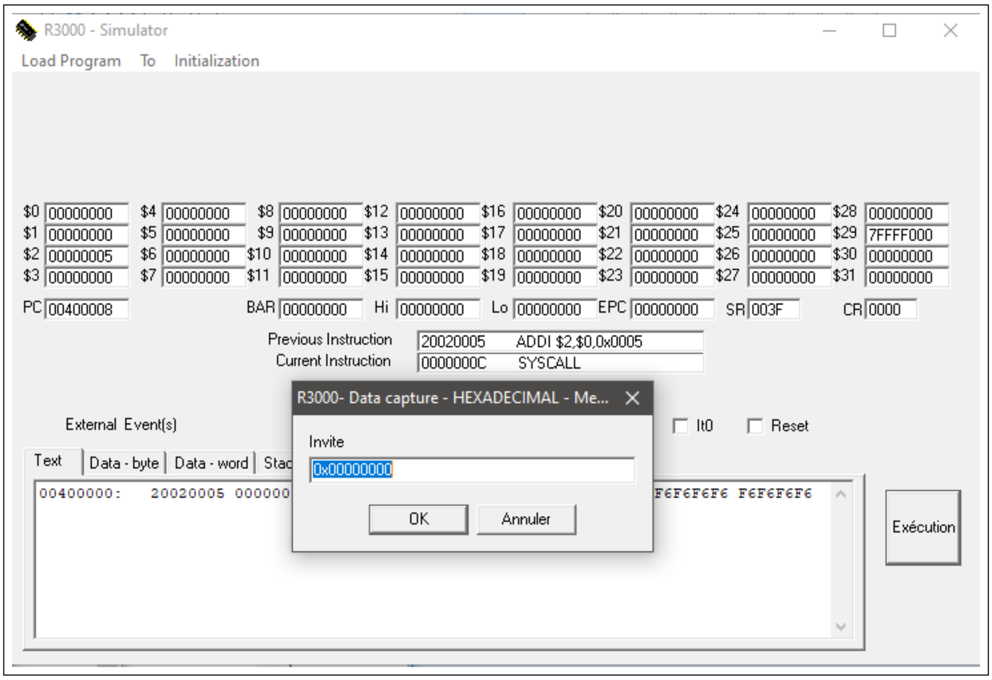

FIGURE 3.2: Dialog box for the reading of an integer from the keyboard

3.4.4 Exercise 3 (Displaying a string: System call number 4)

Follow the steps below:

1. In the data section (`.data`), declare the string to be displayed (e.g. `'Hello World!'`) using the following syntax:

```
.data
        string_id : .asciiz "String to display"
```

2. In the text section: Write the instructions to place the address of the string (your string_id) in register `$4`.
3. Write the instruction to place the immediate 4 in register `$2`.
4. Write the system call instruction (`syscall`).
5. Write the instructions to properly terminate the program.
6. Assemble and execute your program.

NB: The results will be displayed in the TTY window.

3.4.5 Exercise 4 (Understanding Test)

Write an assembly program that reads two integers from the keyboard, then calculates and displays their sum (ADD), difference (SUB), product (MUL), quotient (DIV), and remainder (MOD). The output should be displayed as follows:

```
The sum is:
The difference is:
The product is:
The quotient is:
The remainder is:
```

3.5 PW N°4: Memory (Load/store) instructions

Enter the following program, then assemble.

```
.data
        A: .word 0x12345678
.text
_start:
        la $10, A
        lw $11, 0($10)
        addi $2, $0, 10
        syscall
```

Part I: Exploring the main memory of MIPS R3000:

After loading the program and before starting its execution, observe the memory representation in the simulator (shown in Figure 3.3) and then answer the following questions:

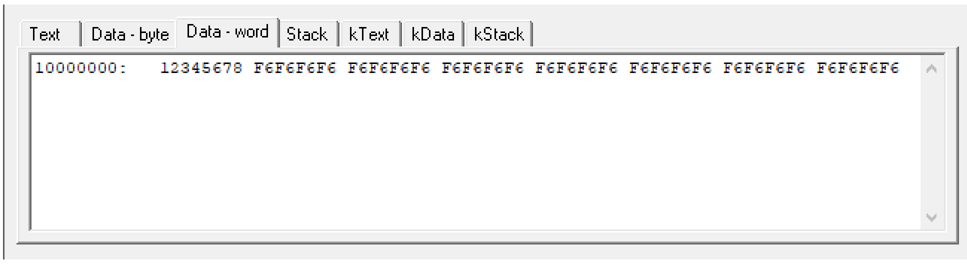

FIGURE 3.3: Representation of the main memory in SIMIPS environment

Questions:

1. What are the different sections of the MIPS R3000 memory?
   Example: Text, ...

2. What are the different segments of the MIPS R3000 memory?

3. What does the Text section of the MIPS R3000 memory contain?

4. What is the starting address of the Text section (Click on the **Text** tab)?

5. What does the Data section of the MIPS R3000 memory contain?

6. What is the starting address of the Data section (Click on the **Data-Word** tab)?

7. What does the Stack section of the MIPS R3000 memory contain?

8. What is the starting address of this section (See the contents of **$29**)?

9. How many bits are used to represent a memory address, a memory cell, and a word in MIPS R3000 ?

Part II: Exploring Load/Store Instructions

3.5.1 Exercise 1: Discovery of load word instruction

We are continuing with the program from Part I.

1. Run the program and complete the following table (Click on *Data-word* in the memory section):

Instruction	Value of A $(Address)$	Word [A] $(Data - word)$	$10	$11
A : `.word 0x12345678`				
la $10, A				
lw $11, 0($10)				

2. What does the macro "la $10, A" do?

3. How is the load address calculated in the instruction `lw $11, 0($10)`?

4. What does the register $11 contain at the end of the program's execution?

5. Deduce the role of the program.

3.5.2 Exercise 2: Discovery of store word instruction

Enter the following program, then assemble.

```
.data
        A: .word 0x12345678
.text
_start:
        addi $11, $0, 0x3579
        la $10, A
        sw $11, 0($10)
        addi $2, $0, 10
        syscall
```

1. Run the program and complete the following table (Click on Data-word in the memory section):

Instruction	**Value of A** (*Address*)	**Word [A]** (*Dataword*)	**$10**	**$11**
A : `.word 0x12345678`				
`addi $11, $0, 0x3579`				
`la $10, A`				
`sw $11, 0($10)`				

2. What does the instruction `sw $11, 0($10)` modify?

3. Deduce the role of the program.

3.5.3 Exercise 3: Example of memory instruction application

Write an assembly program that performs the following tasks:

- Reserve two 32-bit words in memory at addresses **adr1** and **adr2** and initializing them to the values **0x65** and **0x34** respectively.

- Add their values and store the result in memory at address **adr3**.

3.5.4 Exercise 4 (Optional): Memory reading Instructions (Load)

1. Enter the following program, then assemble.

```
.data
        A: .word 0x1234F698
        B : .word 0xA3C8
.text
_start:
        la $10, A
        lw $11,0($10)
        lh $11,0($10)
        lhu $11,0($10)
        lh $11,4($10)
        lb $11,5($10)
        la $10, B
        lb $11,0($10)
        lbu $11,0($10)
        addi $2, $0,10
        syscall
```

2. Run the program step by step and complete the following table:

Instruction	**$10**	**$11**
`la $10, A`		
`lw $11, 0($10)`		
`lh $11, 0($10)`		
`lhu $11, 0($10)`		
`lh $11, 4($10)`		
`lb $11, 5($10)`		
`la $10, B`		
`lb $11, 0($10)`		
`lbu $11, 0($10)`		

3. Analyze the results obtained previously and complete the following table:

Mnemonic	**Meaning**	**Number and location of loaded bytes into RD**	**Extension applied to remaining bits of RD**
Lw RD,I(RS)	Load Word		None
Lh RD,I(RS)		2 bytes from memory will be loaded into the low part of register RD	
Lhu RD,I(RS)			
Lb RD,I(RS)			
Lbu RD,I(RS)			

3.5.5 Exercise 5 (Optional): Memory Write Instructions (Store)

1. Enter the following program, then assemble.

```
.data
        A: .word 6
.text
_start:
        la $10, A
        li $11, 0x12345678
        sw $11,0($10)
        li $11, 0x09ABCDEF
        sh $11,0($10)
        li $11, 0x567890AB
        sb $11, 0($10)
        sb $11, 2($10)
        addi $2, $0,10
        syscall
```

2. Run the program step by step and complete the following table:

Instruction	**\$11**	**Word[A]**
`A: .word 6`		
`li $11, 0x12345678`		
`sw $11,0($10)`		
`li $11, 0x09ABCDEF`		
`sh $11,0($10)`		
`li $11, 0x567890AB`		
`sb $11, 0($10)`		
`sb $11, 2($10)`		

3. Analyze the results obtained previously and complete the following table:

Mnemonic	**Meaning**	**Number and location of stored bytes**	**Extension applied**
sw RD,I(RS)	Save Word		None
sh RD,I(RS)		The two less significant bytes of register RD are stored in memory	
sb RD,I(RS)			

3.6 PW N°5 : Conditional and unconditional branch instructions

3.6.1 Exercise 1: The simple alternative instruction "if..then" and the double alternative "if..then..else"

The two following programs (written in Pascal) allow reading two integers, then calculating and displaying their maximum.

1. Write the corresponding MIPS R3000 assembly program for the program Max1.

Program	Max1;	# Example of a **double** alternative instruction
Var	x, y, max : integer ;	
Begin		
	Read(x,y) ;	# Read the two integers
	if $(x > y)$ **then**	# Comparing x and y values
	$max := x$	# Set the maximum value to x
	else	
	$max := y$;	# Set the maximum value to y
	Writeln(max);	# Display maximum value
End.		

2. Is it possible to write the corresponding MIPS R3000 assembly program for the program Max1 without using jump instructions ?

3. Write the corresponding MIPS R3000 assembly program for the following program Max2.

Program	Max2;	# Example of a **simple** alternative instruction
Var	x, y, max : integer ;	
Begin		
	Read(x,y) ;	# Read the two integers
	$max := x$;	# Initialize the maximum value to x
	if $(y > x)$ **then**	# Comparing x and y values
	$max := y$;	# Set the maximum value to y
	Writeln(max);	# Display maximum value
End.		

4. Rewrite the corresponding MIPS R3000 assembly program for the program Max2 without using a jump instruction.

NB: To test your programs, please consider the 3 possible cases:

1. $x < y$: for example, $x = 3$ and $y = 8$.
2. $x = y$: for example, $x = 3$ and $y = 3$.
3. $x > y$: for example, $x = 8$ and $y = 3$.

3.6.2 Exercise 2: Example of an iterative instruction "the While loop"

The following program (written in Pascal) allows for calculating and displaying the integer $P = x^y$ (x raised to the power of y):

```
Program  Power;
Var      x, y, p, i : integer ;
Begin
         Read(x,y) ;                  # Read x and y values
         p := 1 ;                     # Initialize p (power) to 1
         i := 1 ;                     # Initialize i (loop counter) to 1
         While (i <= y) do
                   begin
                         p := p * x;  # Multiply p by x
                         i := i + 1 ; # Increment i by 1
                   end;
         Write(p);                    # Display the value of p
End.
```

Answer the following questions:

1. Write the corresponding program in MIPS R3000 Assembly.
2. Rewrite the same program to store the following intermediate results x^0, x^1, $\ldots$, x^y in memory.

Indication: Store the values of variables x, y, p, and i in registers \$9, \$10, \$4, and \$11, respectively.

3.6.3 Exercise 3: Example of a program with branching instructions

Enter the following program, then assemble.

```
.data
            Tab: .word 0x12, 10 , -15, 7, 14
.text
_start:
            addi $4,$0,0
            la $9,Tab
            addi $10,$0,5
boucle:     beq $10, $0, Affichage
            lw $11,0($9)
            add $4, $4, $11
            addi $9, $9, 4
            addi $10, $10, -1
            j boucle
Affichage : addi $2, $0, 1
            syscall
            addi $2, $0,10
            syscall
```

Answer the following questions:

1. What is the value of the label "**Tab**"?
2. What is the value of the label "**boucle**"?
3. What is the value of the label "**Affichage**"?
4. What is contained in register 9 ?
5. What is contained in register 10 ?
6. What is contained in register 11 ?
7. What is contained in register 4 ?
8. What are the contents of registers $4, $9, $10, and $11 at the end of the program execution ?
9. Deduce the role of the program.

3.6.4 Exercise 4: Example of exercise on arrays

Write an assembly program that allows for the following:

- Read the number of elements of an array of integers *T* and store it in \$8;
- Read all the elements of the array and store them in memory starting from the address *T*;
- Calculate and display the number of strictly positive elements in the array *T*.

3.6.5 Exercise 5 (optional): Conversion of a decimal number to binary

Write an assembly program that converts a decimal number to base 2.

3.7 Conclusion

In this part, students were progressively introduced to essential instructions in MIPS R3000 assembly language programming.

PW N°1 introduced students to SIMIPS emulator, emphasizing the process of writing and running code. The rest of series began with an exploration of basic arithmetic and logical instructions, followed by an examination of input/output operations used to exchange data with the user. Then, memory management instructions were detailed in series 4. The last series aimed to introduce students about both conditional and unconditional branching, which are crucial for controlling the flow of execution in programs. Thus,the student will be able to translate any algorithm into a MIPS R3000 assembly language program using the basic instructions provided by this language.

Through these exercises, students have developed the foundational skills needed to translate algorithms into MIPS R3000 assembly language, preparing them to implement a wide range of basic to intermediate level programs using this language's core instruction set.

Conclusion

This document was designed to provide a comprehensive and structured learning experience for second year computer science students delving into the MIPS R3000 assembly language. It was designed to facilitate a deep understanding of both theoretical concepts and practical applications. Through a discovery-based learning approach and hands-on exercises, students will gain proficiency in working with instruction sets, memory management, and CPU operations.

Part 1 established the foundation by describing the MIPS R3000 processor's assembly language, including its external architecture, memory structure, CPU registers, and instruction set. This foundational understanding enables students to grasp the syntax and rules necessary for effectively creating assembly language programs.

Part 2 introduced the working environment and tools, including a thorough guide to using the SIMIPS R3000 emulator. This section explained the steps required for developing, assembling, and executing programs, ensuring students are well-prepared to navigate the programming environment.

Part 3 provided a series of practical exercises in MIPS R3000 assembly language programming, each focused on a particular type of instruction. Each exercise series offered two types of activities: discovery exercises, where students explored basic assembly instructions by running provided programs and observing results, and application exercises. Through discovery exercises, students gained insight into the role of each instruction by answering targeted questions. After building a foundational understanding, students were then able to apply their knowledge in application exercises and comprehension tests, allowing them to practice, develop, and test their own assembly programs. This structured progression helped to build foundational skills in MIPS assembly language, equipping students with essential tools for low-level programming and algorithm implementation. With practical exercises, students not only gained a deeper understanding of how to manipulate data at the hardware level but also developed the confidence to tackle more complex programming challenges.

This pedagogical approach ensures that learners actively construct their knowledge, fostering a sense of autonomy and confidence in their programming abilities.

Ultimately, the structured content of this guide aims to prepare students for more advanced studies in computer architecture and low-level programming, laying a solid foundation for their academic and professional journeys.

Bibliography

[1] Robert Britton. *MIPS Assembly Language Programming*. Prentice Hall, 2002.

[2] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2001.

[3] Jean-Lou Desbarbieux, François Dromard, Alain Greiner, Frédéric Pétrot, and Franck Wajsburt. *Processeur MIPS R3000. Architecture externe*. `https://fr.scribd.com/document/608840496/mips-externe-modifie`. Accessed: 05 , 2025. 2003.

[4] Jean-Lou Desbarbieux, François Dromard, Alain Greiner, Frédéric Pétrot, and Franck Wajsburt. *Processeur MIPS32 Langage d'assemblage*. `https://largo.lip6.fr/trac/sesi-ose/chrome/site/mips32assembleur_li312.pdf`. Accessed: 05 , 2025. 2009.

[5] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall PTR, Inc., 1988.

[6] Charles W. Kann. *Introduction to MIPS Assembly Language Programming*. 2. Open Textbooks, 2015. URL: `https://cupola.gettysburg.edu/oer/2`.

[7] James Larus. *SPIM: A MIPS32 Simulator*. `https://spimsimulator.sourceforge.net`. Accessed: 05 , 2025.

[8] Olivier Marchetti. *Architecture des ordinateurs – Mémento MIPS*. `https://www.academia.edu/35603222/CM2_Larchitecture_MIPS32`. Accessed: 05 , 2025.

[9] David A Patterson and John L Hennessy. *Computer organization and Design*. Morgan Kaufmann, 1994.

[10] Dominic Sweetman. *See MIPS run*. Elsevier, 2010.

[11] Kenneth Vollmar and Pete Sanderson. *MARS MIPS Assembler and Runtime Simulator*. `https://computerscience.missouristate.edu/mars-mips-simulator.htm`. Missouri State University, Accessed: 05 , 2025.