1 Introduction to MARS

1.1 Objectives

After completing this lab, you will:

- Get familiar with the MARS simulator
- Learn how to assemble, run, and debug a MIPS program

1.2 The MARS Simulator

MARS, the MIPS Assembly and Runtime Simulator, is an integrated development environment (IDE) for programming in MIPS assembly language. It allows editing, assembling, debugging and simulating the execution of MIPS assembly language programs. MARS is written in Java.

There are two main windows in MARS, as shown in Figure 1.1.

- The *Edit* window: used to create and modify a MIPS program.
- The *Execute* window: used to run and debug a MIPS program.

To switch between the *Edit* and the *Execute* windows, use the tabs at the top.

The *Execute* window contains three main panes:

1. *Text Segment*: shows the machine code and related addresses.
2. *Data Segment*: shows memory locations that hold variables in the data segment.
3. *Labels*: shows addresses of labeled items, i.e. variables and jump endpoints.

There are two tabbed message areas at the bottom of Figure 1.1:

1. The *Mars Messages* tab: Used for messages such as assembly or runtime errors and informational messages. You can click on assembly error messages to select the corresponding line of code in the editor.
2. The *Run I/O* tab: Used at runtime for displaying console output and entering console input as program execution progresses.

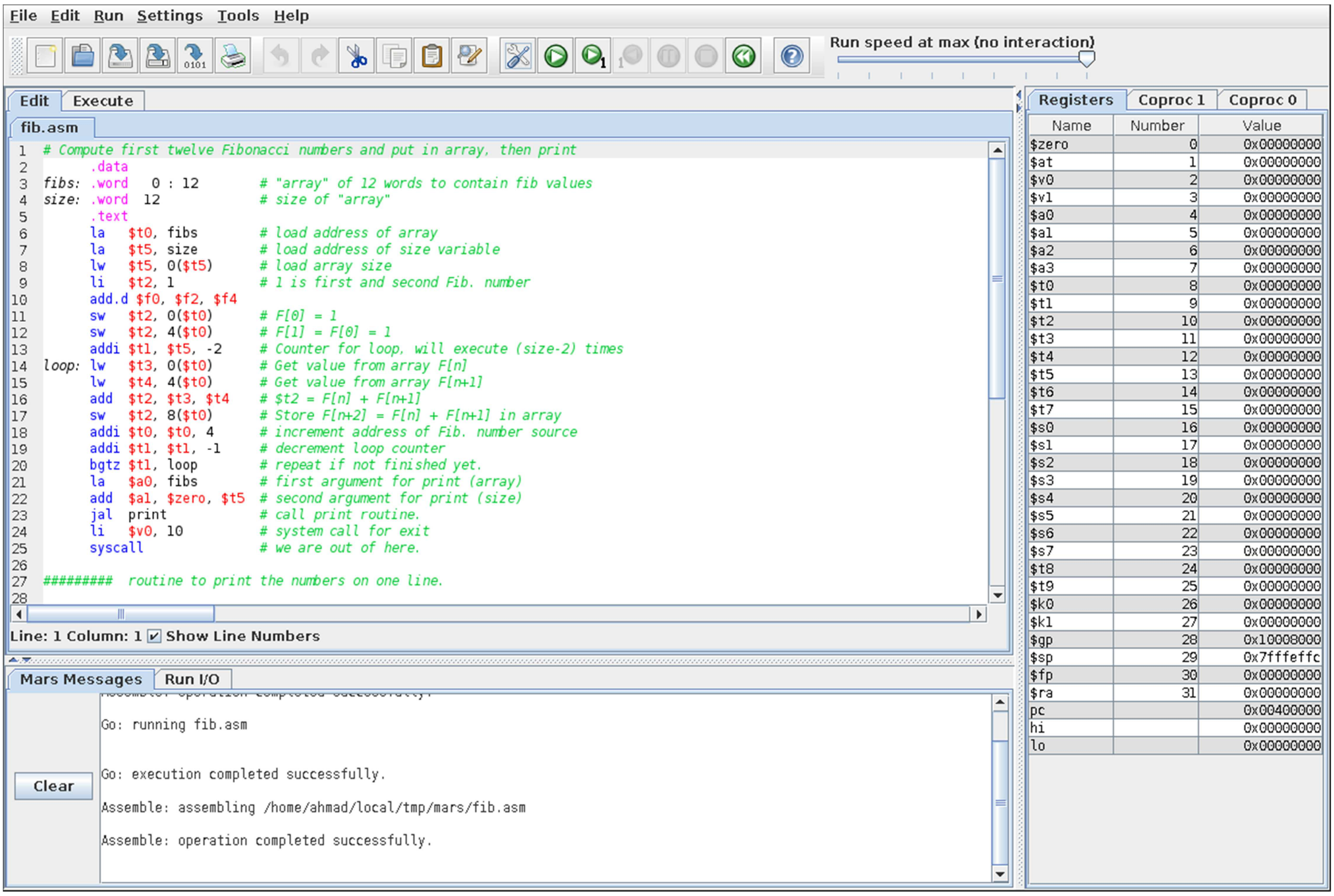


Figure 1.1: The MARS Integrated Development Environment (IDE)

Figure 1.2 shows the MARS Execute window's panes, and emphasizes the following features:

1. The Execute window's tab.
2. Assembly code displayed with addresses and machine code.
3. Values stored in the data segment. These are directly editable.
4. Controls for navigating the data memory area. Allows switching to view the stack segment.
5. Switching between decimal and hexadecimal addresses and values in memory and registers.
6. Labels and their corresponding memory addresses.
7. Values stored in registers. These are directly editable.
8. Checkboxes used to setup breakpoints for each MIPS instruction. Useful in debugging.
9. Execution speed selection. Useful in debugging.

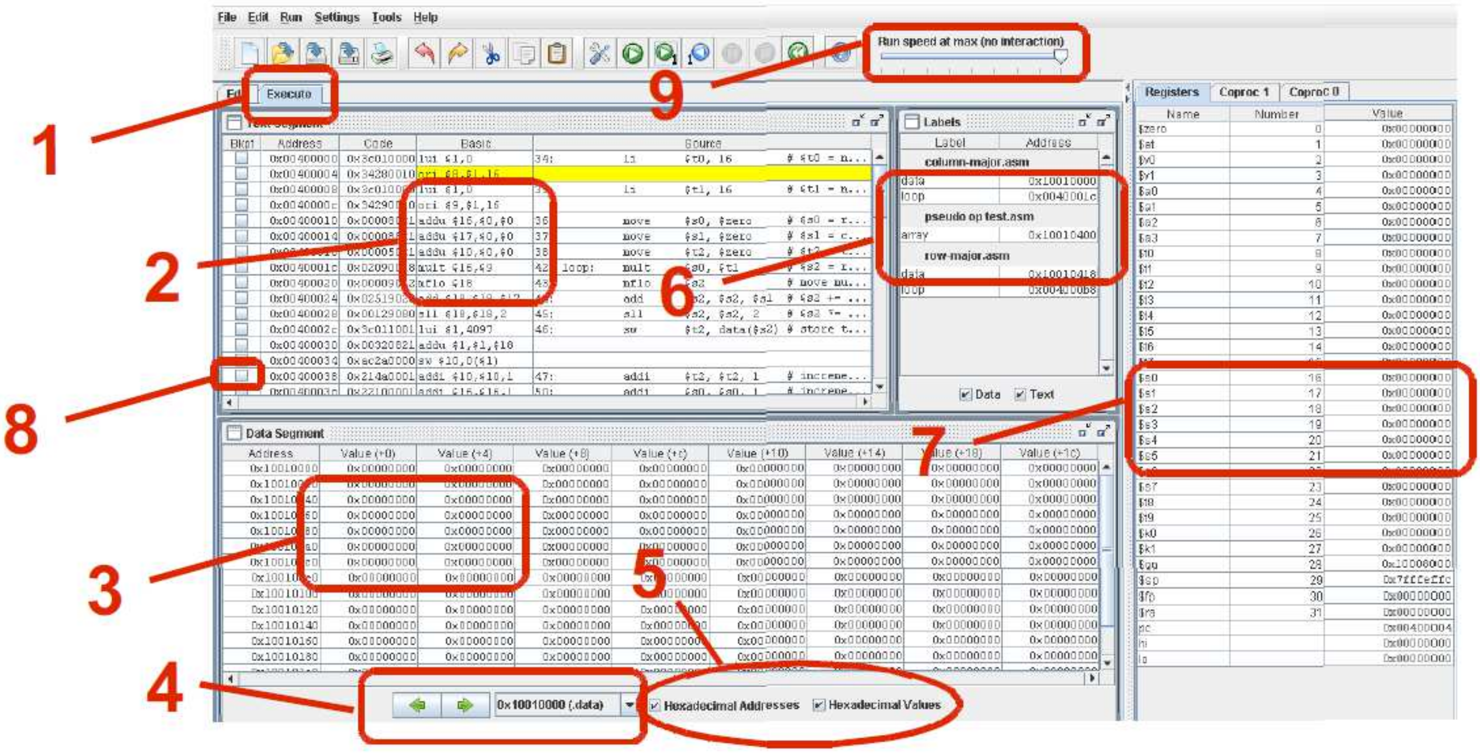


Figure 1.2: The MARS Execute Window

At all times, the MIPS register window appears on the right-hand side of the screen, even when you are editing and not running a program. While writing a program, this serves as a useful reference for register names and their use. Move the mouse over the register name to see the tool tips.

There are three register tabs:

The Register File: integer registers **`$0`** through **`$31`**, **`HI`**, **`LO`**, and the Program Counter **`PC`**.

Coprocessor 0: exceptions, interrupts, and status codes.

Coprocessor 1: floating point registers.

1.3 Assemble, Run, and Debug a MIPS Program

To assemble the file currently in the *Edit* tab, select *Assemble* from the *Run* menu, or use the *Assemble* toolbar icon.

If there are syntax errors in the program, they will appear in the *Mars Messages* tab at the bottom of the MARS screen. Each error message contains the line and column where the error occurred.

Once a MIPS program assembles successfully, the registers are initialized, and the Text Segment and the Data Segment are filled, as shown in Figure 1.3.

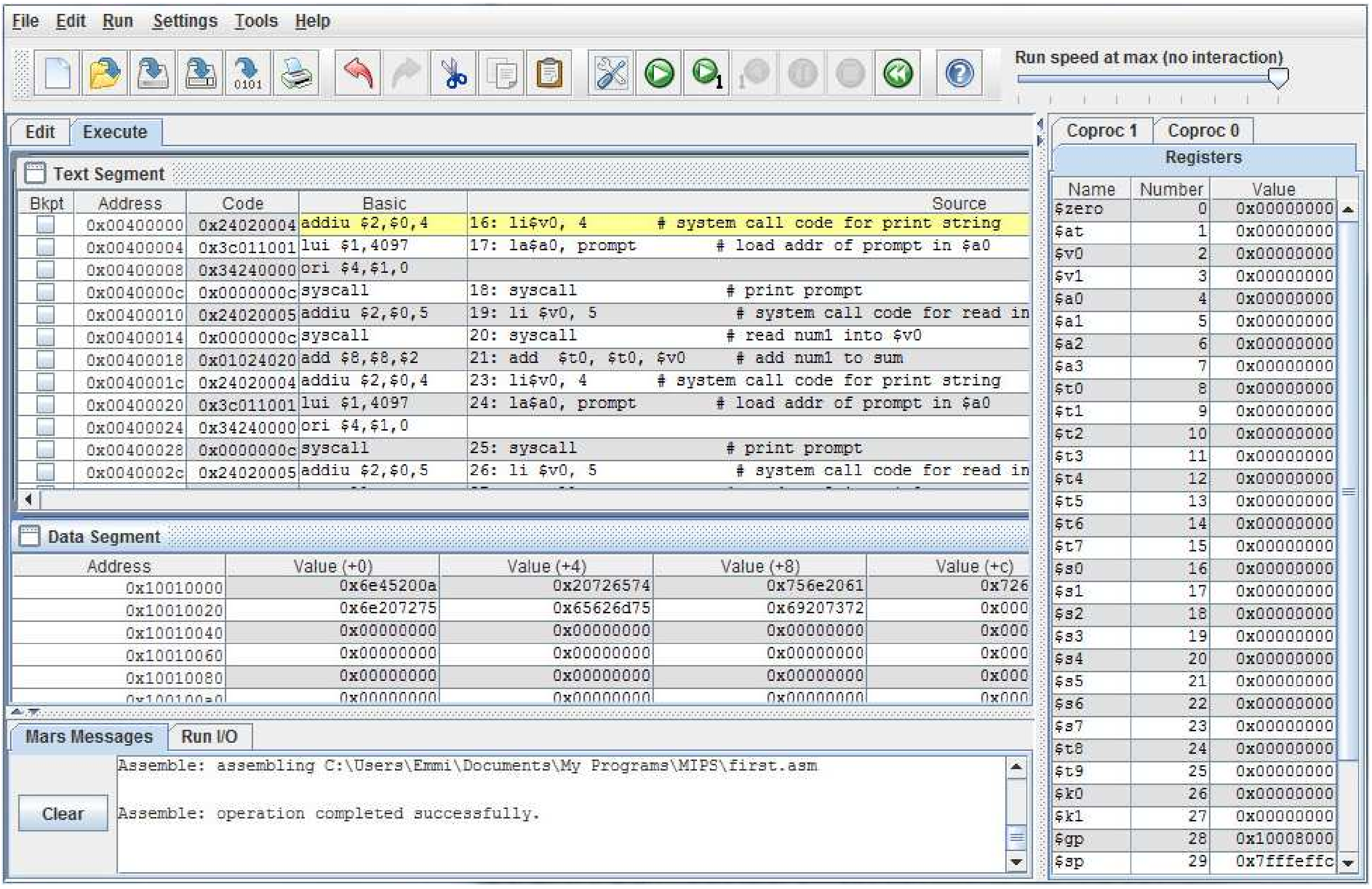
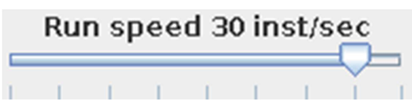


Figure 1.3: MARS screen after running the Assemble command

After running the Assemble command, you can now execute the program. The Run menu and the toolbar contain the following execution options:

Menu Item	Icon	Action
Run > Assemble		Assemble the program.
Run > Go		Run the program to completion, or until the next breakpoint.
Run > Reset		Reset the program and simulator to initial values. Allows restarting program execution.
Run > Step		Single-step execution: execute one instruction at a time. Allows debugging the program by inspecting register and memory after executing each single instruction.
Run > Backstep		Single-step backwards: "unexecute" the last executed instruction.
		The Run Speed Slider allows running the program at full speed or slowing it down so you can watch the execution. Affects normal execution only, not single-step execution.

You can set a breakpoint at any instruction by checking the checkbox in front of the instruction in the text segment pane.

During execution, the instruction being executed is highlighted in yellow, and the register that was last modified is highlighted in green. Also, the variable that was last updated in the data segment is highlighted in blue. It's usually only possible to see the highlighting when you are stepping or running at less than full speed.

For more details about the MARS simulator, refer to the MARS documentation at the following link: http://courses.missouristate.edu/KenVollmar/MARS/

1.4 In-Lab Tasks

1. Test a simple MIPS program. Consider the following program shown below:


```
# Program Name: HelloWorld.asm   (Optional)
        .data                           # data segment
hello:  .asciiz "Hello, world!\n"       # a null terminated string
        .text                           # code segment
        .globl  main                    #
main:
        la      $a0, hello              # load string address
        li      $v0, 4                  # specify system write service
        syscall                         # call the kernel (write string)
        li      $v0,10                  # exit to OS
        syscall
```

   a) Type the program shown in the Figure above.

   b) Find out how to show and hide line numbers.

   c) Assemble and run the program.

   d) What output does the program produce? and where does it appear?

2. Explore the MARS simulator:

   a) Download and assemble the **`Fibonacci.asm`** program from the MARS website.

   b) Identify the locations and values of the initialized data.

   c) Toggle the display format between decimal and hexadecimal.

   d) Run the program at a speed of 3 instructions per second or less.

   e) Single-step through the program and watch how register and memory values change.

f) Observe the output of the program in the *Run I/O* display window.

g) Set a breakpoint at the first instruction that prints results. What is the address of this instruction?

h) Run the program at full speed and watch how it stops at the breakpoint.

i) Change the line:

```
space: .asciiz " "    # space to insert between numbers
```

to:

```
space: .asciiz "\n"   # space to insert between numbers
```

Run the program again. What do you notice?

2 Introduction to MIPS Assembly Programming

2.1 Objectives

After completing this lab, you will:

- Learn about the MIPS assembly language
- Write simple MIPS programs
- Use system calls for simple input and output

2.2 MIPS Assembly Language Program Template

A MIPS assembly language program template is shown in Figure 2.1.

```
# Title:
# Author:
# Date:
# Description:
# Input:
# Output:
################### Data segment #####################
.data
 . . .
################### Code segment #####################
.text
.globl main
main:                       # main function entry
 . . .
li $v0, 10
syscall                     # system call to exit program
```

Figure 2.1: MIPS Assembly Language Program Template

There are three types of statements that can be used in assembly language, where each statement appears on a separate line:

1. *Assembler directives*: These provide information to the assembler while translating a program. Directives are used to define segments and allocate space for variables in memory. An assembler directive always starts with a dot. A typical MIPS assembly language program uses the following directives:

   `.data` Defines the data segment of the program, containing the program's variables.

   `.text` Defines the code segment of the program, containing the instructions.

   `.globl` Defines a symbol as global that can be referenced from other files.

2. *Executable Instructions*: These generate machine code for the processor to execute at runtime. Instructions tell the processor what to do.
3. *Pseudo-Instructions and Macros*: Translated by the assembler into real instructions. These simplify the programmer task.

In addition there are comments. Comments are very important for programmers, but ignored by the assembler. A comment begins with the **#** symbol and terminates at the end of the line. Comments can appear at the beginning of a line, or after an instruction. They explain the program purpose, when it was written, revised, and by whom. They explain the data and registers used in the program, input, output, the instruction sequence, and algorithms used.

2.3 The Edit-Assemble-Link-Run Cycle

Before you can run a MIPS program, you must convert the assembly language code into an executable form. This involves two steps:

1. *Assemble*: translate the MIPS assembly language code into a binary *object file*. This is done by the *assembler*. If there is more than one assembly language file, then each should be assembled separately.
2. *Link*: combine all the object files together (if there is more than one) as well as with libraries. This is done by the *linker*. The linker checks if there are any calls to functions in libraries. The result is an *executable file*.

Figure 2.2 summarizes the *Edit-Assemble-Link-Run* cycle of the program development process. If a program is written in assembly language, the *assembler* detects any *syntax errors* and will report them to the programmer. Therefore, you should edit your program and assemble it again if there any syntax errors.

It is typical that the first executable version of your program to have some *runtime errors*. These errors are not detected by the assembler but occur when you are running your program. For example, your program might compute erroneous results. Therefore, you should *debug* your program to identify the errors at runtime. You can run your program with various inputs and under different conditions to verify that it is working correctly. You can use the slow execution mode in

MARS, the single-step feature, or breakpoints to identify the sources of the errors. Single-step execution is a standard and essential feature in a debugger. It allows inspecting the effect of each instruction on CPU registers and main memory.

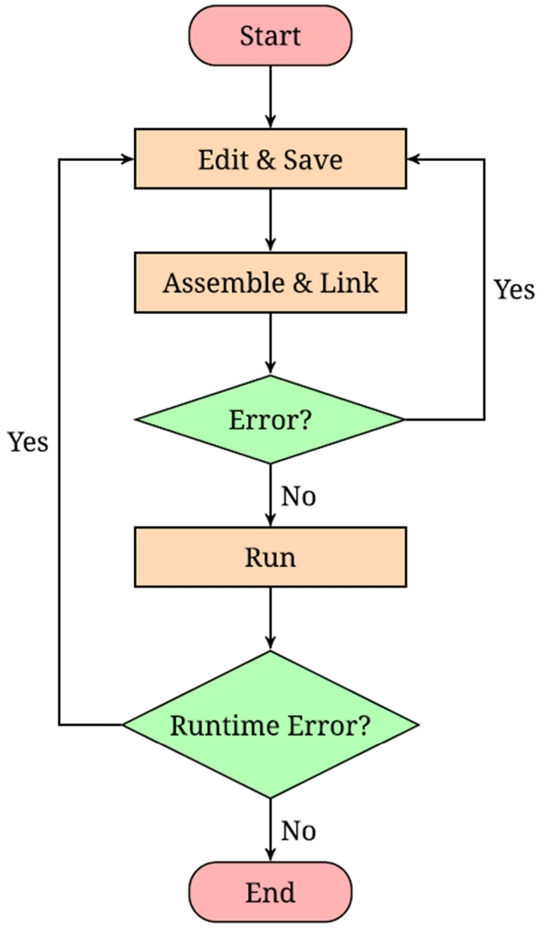


Figure 2.2: *The Edit-Assemble-Link-Run* Cycle

2.4 MIPS Instructions, Registers, Format and Syntax

All MIPS instructions are 32-bit wide and occupy 4 bytes in memory. The address of a MIPS instruction in memory is always a multiple of 4 bytes. There are three basic MIPS instruction formats: Register (R-Type) format, Immediate (I-Type) format, and Jump (or J-Type) format as shown in Figure 2.3.

All instructions have a 6-bit opcode that defines the format and sometimes the operation of an instruction. The R-type format has two source register fields: **`Rs`** and **`Rt`**, and one destination register field **`Rd`**. All register fields are 5-bit long and address 32 general-purpose registers. The **`sa`** field is used as the ***shift amount*** for shift instructions and the **`funct`** field defines the ALU function for R-type instructions.

The I-type format has two register fields only: **`Rs`** and **`Rt`**, where **`Rs`** is always a source register, while **`Rt`** can be a destination register or a second source depending on the opcode. The 16-bit immediate field is used as a constant in arithmetic instructions, or as an offset in load, store, and branch instructions.

The J-type format has no register field. The 26-bit Immediate field is used as an address in jump and function call instructions.

R-Type Format

Op^6	Rs^5	Rt^5	Rd^5	sa^5	$funct^6$

I-Type Format

Op^6	Rs^5	Rt^5	$Immediate^{16}$

J-Type Format

Op^6	$Immediate^{26}$

Figure 2.3: MIPS Instruction Formats

The MIPS architecture defines 32 general-purpose registers, numbered from **`$0`** to **`$31`**. The **`$`** sign is used to refer to a register. To simplify software development, the assembler can also refer to registers by name as shown in Table 2.1. The assembler converts a register name to its corresponding number.

Register Name	Number	Register Usage by Software
`$zero`	`$0`	Always zero, forced by hardware
`$at`	`$1`	Assembler Temporary register, reserved for assembler use
`$v0 - $v1`	`$2 - $3`	Results of a function
`$a0 - $a3`	`$4 - $7`	Arguments of a function
`$t0 - $t7`	`$8 - $15`	Registers for storing temporary values
`$s0 - $s7`	`$16 - $23`	Registers that should be saved across function calls
`$t8 - $t9`	`$24 - $25`	Registers for storing more temporary values
`$k0 - $k1`	`$26 - $27`	Registers reserved for the OS kernel use
`$gp`	`$28`	Global Pointer register that points to global data
`$sp`	`$29`	Stack Pointer register that points to top of stack
`$fp`	`$30`	Frame Pointer register that points to stack frame
`$ra`	`$31`	Return Address register used to return from a function call

Table 2.1: General-Purpose Registers and their Usage

The general assembly language syntax of a MIPS instruction is:

```
[label:] mnemonic [operands] [# comment]
```

The **`label`** is optional. It marks the memory address of the instruction. It must have a colon. In addition, a **`label`** can be used for referring to the address of a variable in memory.

The **`mnemonic`** specifies the operation: **`add`**, **`sub`**, etc.

The **`operands`** specify the data required by the instruction. Different instructions have different number of operands. Operands can be registers, memory variables, or constants. Most arithmetic and logical instructions have three operands.

An example of a MIPS instruction is shown below. This example uses the **`addiu`** to increment the **`$t0`** register:

```
L1:   addiu $t0, $t0, 1     # increment $t0
```

To be able to write programs, a basic set of instructions is needed. Only few instructions are described in the following tables. Table 2.2 lists the basic arithmetic instructions and Table 2.3 lists basic control instructions.

Instruction	Meaning
`add   Rd, Rs, Rt`	**`Rd = Rs + Rt`**. Overflow causes an exception.
`sub   Rd, Rs, Rt`	**`Rd = Rs – Rt`**. Overflow causes an exception.
`addi  Rt, Rs, Imm`	**`Rt = Rs + Imm`** (16-bit constant). Overflow causes an exception.
`li   Rt, Imm`	**`Rt = Imm`** (pseudo-instruction).
`la   Rt, var`	**`Rt = address of var`** (pseudo-instruction).
`move Rd, Rs`	**`Rd = Rs`** (pseudo-instruction).

Table 2.2: Basic Arithmetic Instructions.

Instruction	Meaning
`beq Rs, Rt, label`	if **`(Rs == Rt)`** branch to **`label`**.
`bne Rs, Rt, label`	if **`(Rs != Rt)`** branch to **`label`**.
`j   label`	Jump to **`label`**.

Table 2.3: Basic Control Instructions.

2.5 System Calls

Programs do input and output using system calls. On a real-system, the operating system provides system call services to application programs. The MIPS architecture provides a special **`syscall`** instruction that generates a system call exception, which is handled by the operating system.

System calls are operating-system specific. Each operating system provides its own set of system calls. Because MARS is a simulator, there is no operating system involved. The MARS simulator handles the **`syscall`** exception and provides system services to programs. Table 2.1 shows a small set of services provided by MARS for doing basic I/O.

Before using the **`syscall`** instruction, you should load the service number into register **`$v0`**, and load the arguments, if any, into registers **`$a0`**, **`$a1`**, etc. After issuing the **`syscall`** instruction, you should retrieve return values, if any, from register **`$v0`**.

Service	Code in $v0	Arguments	Result
Print Integer	**1**	**`$a0`** = integer to print	
Print String	**4**	**`$a0`** = address of null-terminated string	
Read Integer	**5**		**`$v0`** = integer read
Read String	**8**	**`$a0`** = address of input buffer **`$a1`** = maximum characters to read	
Exit program	**10**		Terminates program
Print char	**11**	**`$a0`** = character to print	
Read char	**12**		**`$v0`** = character read

Table 2.4: Basic System Call Services Provided by MARS.

Now, we are ready to write a MIPS assembly language program. A simple program that asks the user to enter an integer value and then displays the value of this integer is shown in Figure 2.4.

Five system calls are used. The first system call prints string *str1*. The second system call reads an input integer. The third system call prints *str2*. The fourth system call prints the integer value that was input by the user. The fifth system call exits the program.

```
.data
str1:       .asciiz     "Enter an integer value: "
str2:       .asciiz     "You entered "

.globl      main
.text
main:
    li      $v0, 4      # service code for print string
    la      $a0, str1   # load address of str1 into $a0
    syscall             # print str1 string
    li      $v0, 5      # service code for read integer
    syscall             # read integer input into $v0
    move    $s0, $v0    # save input value in $s0
    li      $v0, 4      # service code for print string
    la      $a0, str2   # load address of str2 into $a0
    syscall             # print str2 string
    li      $v0, 1      # service code to print integer
    move    $a0, $s0    # copy input value
    syscall             # print integer
    li      $v0, 10     # service code to exit program
    syscall             # exit program
```

Figure 2.4: MIPS Program that uses System Calls

2.6 In-Lab Tasks

1. Modify the program shown in Figure 2.4. Ask the user to enter an integer value, and then print the result of doubling that number. Use the **`add`** instruction.
2. Modify again the program shown in Figure 2.4. Ask the user whether he wants to repeat the program: **`"\nRepeat [y/n]? "`**. Use service code 12 to read a character and the branch instruction to repeat the main function if the user input is character **`'y'`**.
3. Write a MIPS program that asks the user to input his name and then prints **`"Hello "`**, followed by the name entered by the user.

4. Write a MIPS program that executes the statement: $s = (a + b) - (c + 101)$, where a, b, and c are user provided integer inputs, and s is computed and printed as an output. Answer the following:
    a. Suppose the user enters $a = 5$, $b = 10$, and $c = -30$, what is the expected value of s?
    b. Which instruction in your program computed the value of s and which register is used?
    c. What is the address of this instruction in memory?
    d. Put a breakpoint at this instruction and write the value of the register used for computing s in decimal and hexadecimal.
5. Write a MIPS program that inputs two integer values. The program should output **`equal`** if the two integers are equal. Otherwise, it should output **`not equal`**. Use the branch instruction to check for equality.