# LAB # 4

INTRODUCTION TO MIPS ASSEMBLY LANGUAGE

**OBJECTIVES**

Introduction to MIPS Assembly language. Simulating the given MIPS program using MARS.

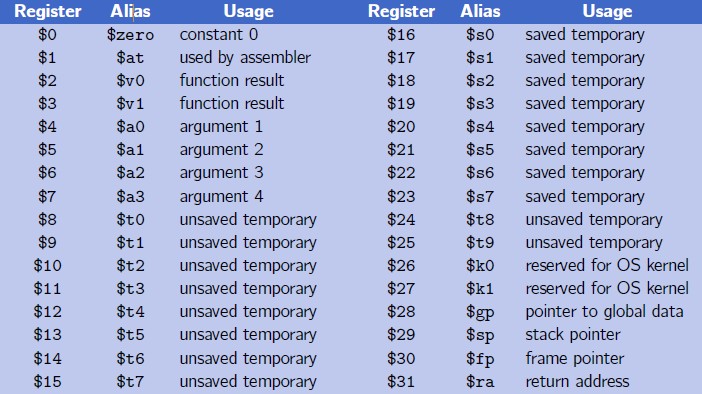
**THEORY**

##### The MIPS Architecture

**MIPS** (originally an acronym for **Microprocessor without Interlocked Pipeline Stages**) is a **Reduced Instruction Set Computer** (RISC). MIPS is a register based architecture, meaning the CPU uses registers to perform operations on. Registers are memory just like RAM, except registers are much smaller than RAM, and are much faster. In MIPS the CPU can only do operations on registers, and special immediate values. MIPS processors have 32 registers, but some of these are reserved. A fair number of registers however are available for your use.

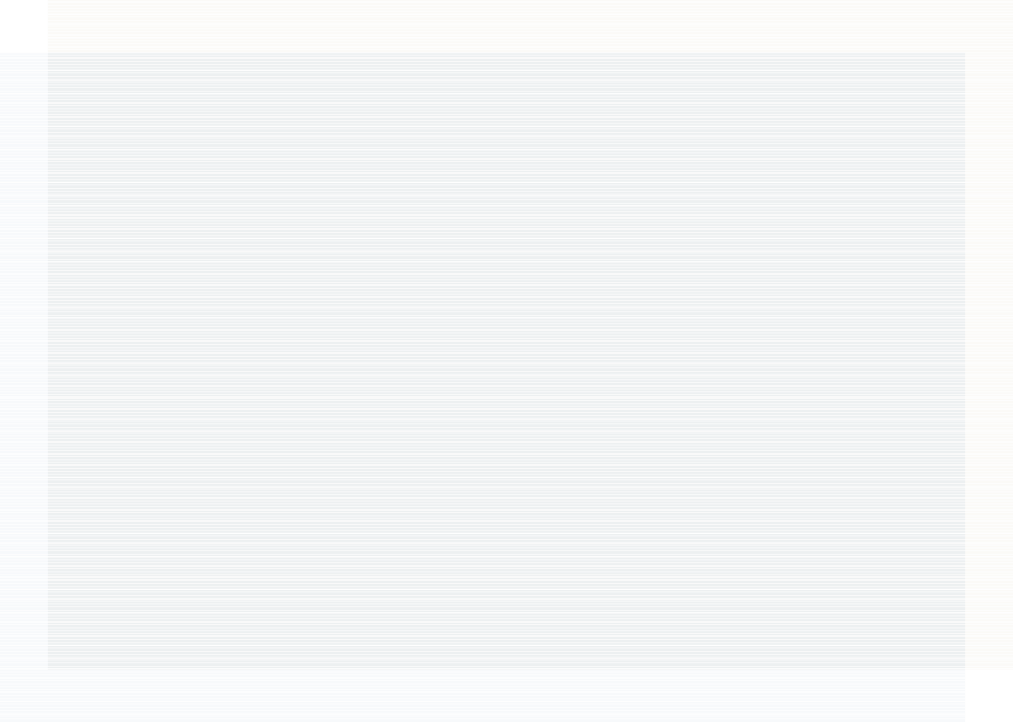
##### MIPS: Registers

The MIPS registers are arranged into a structure called a **Register File.** MIPS comes with 32 general purpose registers named $0. . . $31. Registers also have symbolic names reflecting their conventional use:



**Introduction to MIPS Assembly Language**

##### Assembly Language Program Template



**################# Code segment #####################**

**.text**

**.globl main main:**

**. . .**

**. . .**

**li $v0, 10 syscall**

**# main program entry**

**# Exit program**

**.data**

**. . .**

**. . .**

***# Input:***

**# Output:**

**################# Data segment #####################**

***Filename: Date:***

***# Title:***

***# Author:***

***# Description:***

**Assembly language instruction format**

Assembly language source code lines follow this format:

[*label*:] [*instruction*/*directive*] [*operands*] [#*comment*]

where [*label*] is an optional symbolic name; [*instruction*/*directive*] is either the mnemonic for an instruction or pseudo-instruction or a directive; [*operands*] contains a combination of one, two, or three constants, memory references, and register references, as required by the particular instruction or directive; [#*comment*] is an optional comment.

##### Labels

Labels are nothing more than names used for referring to numbers and character strings or memory locations within a program. Labels let you give names to memory variables, values, and the locations of particular instructions.

The label *main* is equivalent to the address of the first instruction in program1.

##### li $v0, 5

**Directives**

Directives are required in every assembler program in order to define and control memory space usage. Directives only provide the framework for an assembler program, though; you also need lines in your source code that actually DO something, lines like

beq $v0, $0, end

##### .DATA directive

* Defines the data segment of a program containing data
* The program’s variables should be defined under this directive
* Assembler will allocate and initialize the storage of variables
* You should place your memory variables in this segment. For example,

**.**DATA

First: **.**space 100

Second: **.**word 1, 2, 3

Third: **.**byte 99, 2, 3

##### .TEXT directive

* Defines the code segment of a program containing instructions

##### .GLOBL directive

* Declares a symbol as global
* Global symbols can be referenced from other files
* We use this directive to declare *main* procedure of a program

##### .ASCII Directive

* Allocates a sequence of bytes for an ASCII string

##### .ASCIIZ Directive

* Same as .ASCII directive, but adds a NULL char at end of string
* Strings are null-terminated, as in the C programming language

**.SPACE n** Directive

* Allocates space of *n* uninitialized bytes in the data segment

##### Pseudo-instructions

Pseudo-instructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. For example, one of the frequent steps needed in programming is to copy the value of one register into another register. This actually can be solved easily by the instruction:

add $t0, $zero, $t1

However, it is more natural to use the pseudo-instruction move $t0, $t1.

The assembler converts this pseudo-instruction into the machine language equivalent of the prior instruction.

##### MIPS INSTRUCTIONS

|  |  |
| --- | --- |
| Instructions | Description |
| la Rdest, var | **Load Address**. Loads the address of var into Rdest. |
| li Rdest, imm | **Load Immediate**. Loads the immediate value imm into  Rdest. |

**SYSTEM I/O (INPUT/OUTPUT)**

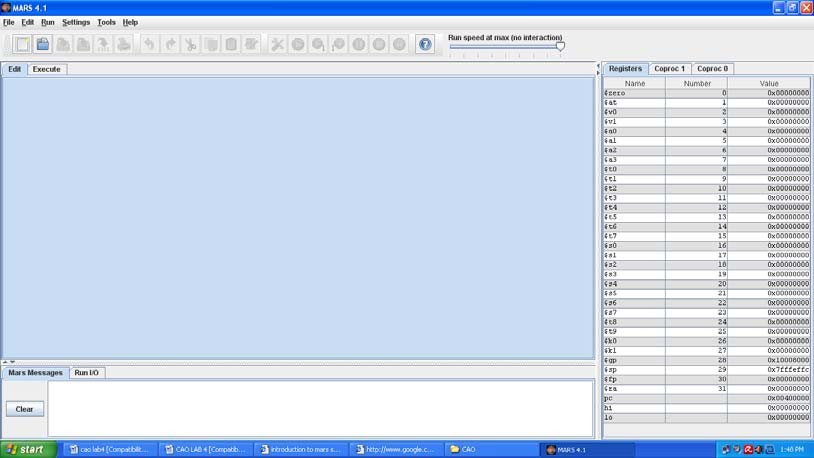
* Programs do input/output through system calls
* MIPS provides a special **syscall** instruction
  + To obtain services from the operating system
  + Many services are provided in the MARS simulators
  + There are 10 different services provided.
* Using the **syscall** system services
  + Load the service number in register $v0
  + Load argument values, if any, in registers $a0, $a1, etc.
  + Issue the **syscall** instruction
  + Retrieve return values, if any, from result registers

|  |  |  |  |
| --- | --- | --- | --- |
| Service | Code in $v0 | Argument(s) | Result(s) |
| Print integer | 1 | $a0 = number to be printed |  |
| Print String | 4 | $a0 = address of string in  memory |  |
| Read Integer | 5 |  | Number returned in $v0. |
| Read String | 8 | $a0 = address of input buffer in memory.  $a1 = length of buffer (n) |  |
| Exit | 10 |  |  |
| Print Char | 11 | $a0 =character to print |  |
| Read Char | 12 | $v0 = character read |  |

### Introduction to MARS

MARS, the **M**IPS **A**ssembly and **R**untime **S**imulator, will assemble and simulate the execution of MIPS assembly language programs. It can be used either from a command line or through its integrated development environment (IDE). MARS is written in Java and requires at least Release 1.5 of the J2SE Java Runtime Environment (JRE) to work.

##### MARS Editor



**MARS Integrated Development Environment (IDE)**

The IDE is invoked from a graphical interface by double-clicking the mars.jar icon that represents this executable JAR file. The IDE provides basic editing, assembling and execution capabilities. Hopefully it is intuitive to use. Here are comments on some features.

* **Menus and Toolbar**: Most menu items have equivalent toolbar icons. If the function of a toolbar icon is not obvious, just hover the mouse over it and a tool tip will soon appear. Nearly all menu items also have keyboard shortcuts. Any menu item not appropriate in a given situation is disabled.
* **Editor**: MARS includes two integrated text editors. The default editor, new in Release 4.0, features syntax-aware color highlighting of most MIPS language elements and popup instruction guides. The original, generic, text editor without these features is still available and can be selected in the Editor Settings dialog. It supports a single font which can be modified in the Editor Settings dialog. The bottom border of either editor includes the cursor line and column position and there is a checkbox to display line numbers. They are displayed outside the editing area. If you use an external editor, MARS provides a convenience setting that will automatically assemble a file as soon as it is opened. See the Settings menu.
* **Message Areas**: There are two tabbed message areas at the bottom of the screen. The *Run I/O* tab is used at runtime for displaying console output and entering console input as program execution progresses. You have the option of entering console input into a pop-up dialog then echoes to the message area. The *MARS Messages* tab is used for other 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.

* **MIPS Registers**: MIPS registers are displayed at all times, even when you are editing and not running a program. While writing your program, this serves as a useful reference for register names and their conventional uses (hover mouse over the register name to see tool tips). There are three register tabs: the Register File (integer registers $0 through $31 plus LO, HI and the Program Counter), selected Coprocessor 0 registers (exceptions and interrupts), and Coprocessor 1 floating point registers.
* **Assembly**: Select *Assemble* from the *Run* menu or the corresponding toolbar icon to assemble the file currently in the Edit tab. Prior to Release 3.1, only one file could be assembled and run at a time. Releases 3.1 and later provide a primitive Project capability. To use it, go to the *Settings* menu and check *Assemble operation applies to all files in current directory.* Subsequently, the assembler will assemble the current file as the “main” program and also assemble all other assembly files (\*.asm; \*.s) in the same directory. The results are linked and if all these operations were successful the program can be executed. Labels that are declared global with the “.globl” directive may be referenced in any of the other files in the project. There is also a setting that permits automatic loading and assembly of a selected exception handler file. MARS uses the MIPS32 starting address for exception handlers: 0x80000180.
* **Execution**: Once a MIPS program successfully assembles, the registers are initialized and three windows in the Execute tab are filled: *Text Segment*, *Data Segment*, and *Program Labels*. The major execution-time features are described below.
* **Labels Window**: Display of the Labels window (symbol table) is controlled through the Settings menu. When displayed, you can click on any label or its associated address to center and highlight the contents of that address in the Text Segment window or Data Segment window as appropriate.

The assembler and simulator are invoked from the IDE when you select the *Assemble*, *Go*, or *Step* operations from the *Run* menu or their corresponding toolbar icons or keyboard shortcuts. MARS messages are displayed on the *MARS Messages* tab of the message area at the bottom of the screen. Runtime console input and output is handled in the *Run I/O* tab.

### Program#1:

**Reading and Printing an Integer**

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

**.text**

**.globl main**

**main: # main program entry li $v0, 5 # Read integer**

**syscall # $v0 = value read move $a0, $v0 # $a0 = value to print li $v0, 1 # Print integer syscall**

**li $v0, 10 # Exit program syscall**

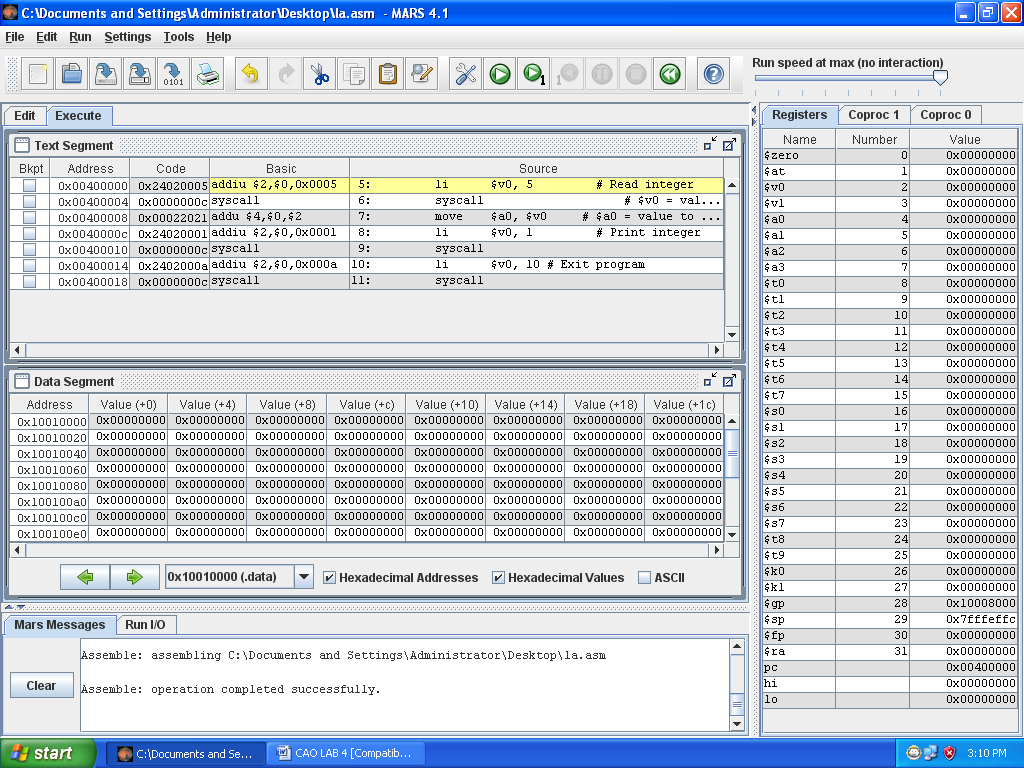
**STEP# 1**

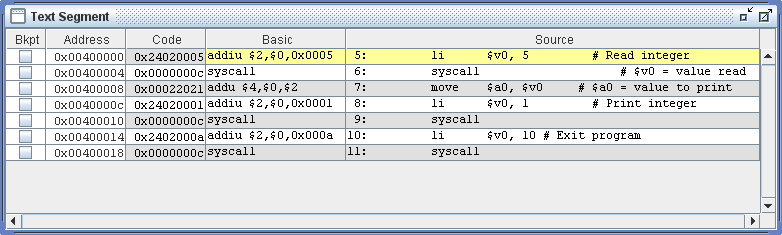
Load mars simulator, copy this code to the editor and save file with .asm extension.

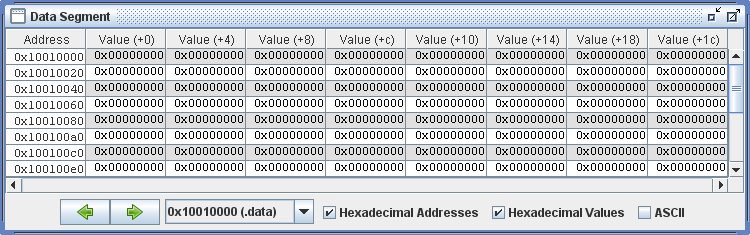


##### STEP# 2

Assemble program by pressing F3.

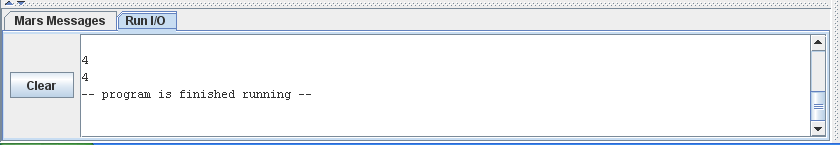






##### STEP# 3

Execute program by pressing F5. Type any integer number for input.



### LAB TASK

1. Where (to which window) is the output data displayed?
2. Write down the address of the first instruction of the program (see the text window)
3. Write down the value of the register **$sp** just before you start the program.
4. Write down the values of $a0 and $v0 after execution in Register window and why?
5. Write an assembly program that Read and Print character.