

LAB MANUAL

Computer Architecture & Organization

Course Code: CEN-221



Department of Computer & Software Engineering
Bahria University Karachi Campus
13 National Stadium Road



Table of Contents

LAB 01: INTRODUCTION TO VVM.....	
LAB 02: INTRODUCTION TO VVM PROGRAMMING	
LAB 03: VVM PROGRAMMING	
LAB 04: INTRODUCTION TO MIPS ASSEMBLY LANGUAGE	
LAB 05: MIPS ASSEMBLY LANGUAGE ARITHMETIC OPERATIONS	
LAB 06: MIPS ASSEMBLY LANGUAGE USING ADDU INSTRUCTION'	
LAB 07: ARITHMETIC INSTRUCTIONS IN MIPS	
LAB 08: MULTIPLICATION AND DIVISION INSTRUCTIONS IN MIPS	
LAB 09: BIT MANIPULATION INSTRUCTIONS IN MIPS	
LAB 10: IF THEN ELSE; CONTROL STRUCTURE IN MIPS	
LAB 11: FOR LOOP; CONTROL STRUCTURE IN MIPS.....	
LAB 12: SWITCH; CONTROL STRUCTURE IN MIPS.	
LAB 13 : ARRAY; CONTROL STRUCTURE IN MIPS	
LAB 14 : DESIGNING GAME; TOWER OF HANOI.....	



LAB # 1

INTRODUCTION TO VVM

OBJECTIVE

Explore Visible Virtual Machine (VVM).

THEORY

Visible Virtual Machine (VVM)

The Visible Virtual Machine (VVM) is based on a model of a simple computer device called the Little Man Computer which was originally developed by Stuart Madnick in 1965, and revised in 1979. The revised Little Man Computer model is presented in detail in “The Architecture of Computer Hardware and System Software” (2nd), by Irv Englander (Wiley, 2000).

The VVM is a virtual machine because it only appears to be a functioning hardware device. In reality, the VVM “hardware” is created through a software simulation. One important simplifying feature of this machine is that it works in decimal rather than in the traditional binary number system. Also, the VVM works with only one form of data - decimal integers.

VVM is a 32-bit application for use on a Windows platform. The application adheres to the Windows style GUI guidelines and thus provides a short learning curve for experienced Windows users. Online context-sensitive help is available throughout the application.

VVM includes a fully functional Windows-style VVM Program Editor for creating and manipulating **VVM** programs. The editor provides a program syntax validating facility which identifies errors and allows them to be corrected. Once the program has been validated, it can be loaded into the VVM Virtual Hardware.

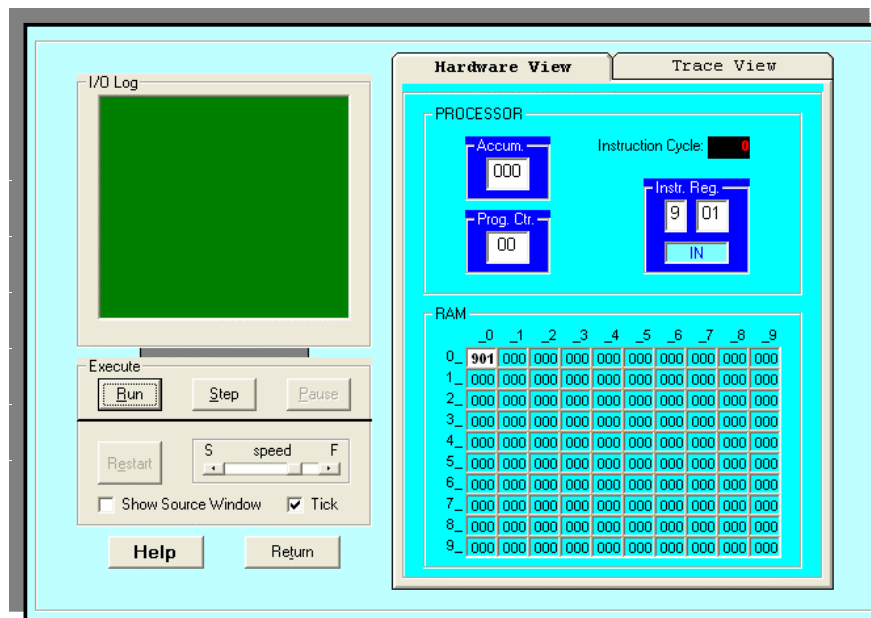
For simplicity, **VVM** works directly with decimal data and addresses rather than with binary values. Furthermore, the virtual machine works with only one form of data: decimal integers in the range ± 999 . This design alleviates the need to interpret long binary strings or complex hexadecimal codes.

When using **VVM**, the user is given total control over the execution of his or her program. Execution speed of the program can be increased or decreased via a mouse-driven speed control. The program can be paused and subsequently resumed at any point, at the discretion of the user. Alternatively, the user can choose to step through the program one statement at a time. As each program instruction is executed, all relevant hardware components (e.g., internal registers, RAM locations, output devices, etc.) are updated in full view of the user.

Hardware Components

The VVM machine comprises the following hardware components:

- **I/O Log.** This represents the system console which shows the details of relevant events in the execution of the program. Examples of events are the program begins, the program aborts, or input or output is generated.
- **Accumulator Register (Accum).** This register holds the values used in arithmetic and logical computations. It also serves as a buffer between input/output and memory. Legitimate values are any integer between -999 and +999. Values outside of this range will cause a fatal VVM Machine error. Non-integer values are converted to integers before being loaded into the register.
- **Instruction Cycle Display.** This shows the number of instructions that have been executed since the current program execution began.
- **Instruction Register (Instr. Reg.).** This register holds the next instruction to be executed. The register is divided into two parts: a one-digit *operation code*, and a two-digit *operand*. The Assembly Language mnemonic code for the operation code is displayed below the register.
- **Program Counter Register (Prog. Ctr.).** The two-digit integer value in this register “points” to the next instruction to be fetched from RAM. Most instructions increment this register during the *execute* phase of the instruction cycle. Legitimate values range from 00 to 99. A value beyond this range causes a fatal VVM Machine error.
- **RAM.** The 100 *data-word* Random Access Storage is shown as a matrix of ten rows and ten columns. The two-digit memory addresses increase sequentially across the rows and run from 00 to 99. Each storage location can hold a three-digit integer value between -999 and +999.



Data and Addresses

All data and address values are maintained as decimal integers. The 100 data-word memory is addresses with two-digit addressed in the range 00-99. Each memory location holds one data-word which is a decimal integer in the range -999 - +999. Data values beyond this range cause a data overflow condition and trigger a VVM system error.

Trace View

The Trace View window provides a history of the execution of your program. Prior to the execution of each statement, the window shows:

1. The instruction cycle count (begins at 1)
2. The address from which the instruction was fetched
3. The instruction itself (in VVM Assembly Language format)
4. The current value of the Accumulator Register

VVM System Errors

Various conditions or events can cause VVM System Errors. The possible errors and probable causes are as follows:

- **Data value out of range.** This condition occurs when a data value exceeds the legitimate range -999 - +999. The condition will be detected while the data resides in the *Accumulator Register*. Probable causes are an improper addition or subtraction operation, or invalid user input.
- **Undefined instruction.** This occurs when the machine attempts to execute a three-digit value in the *Instruction Register* which can not be interpreted as a valid instruction code. See the help topic “VVM Language” for valid instruction codes and their meaning. Probable causes of this error are attempting to use a data value as an instruction, an improper *Branch* instruction, or failure to provide a *Halt* instruction in your program.
- **Program counter out of range.** This occurs when the Program Counter Register is incremented beyond the limit of 99. The likely cause is failure to include a *Halt* instruction in your program, or a branch to a high memory address.
- **User cancel.** The user pressed the “Cancel” button during an *Input* or *Output* operation.

VVM Program Example 1

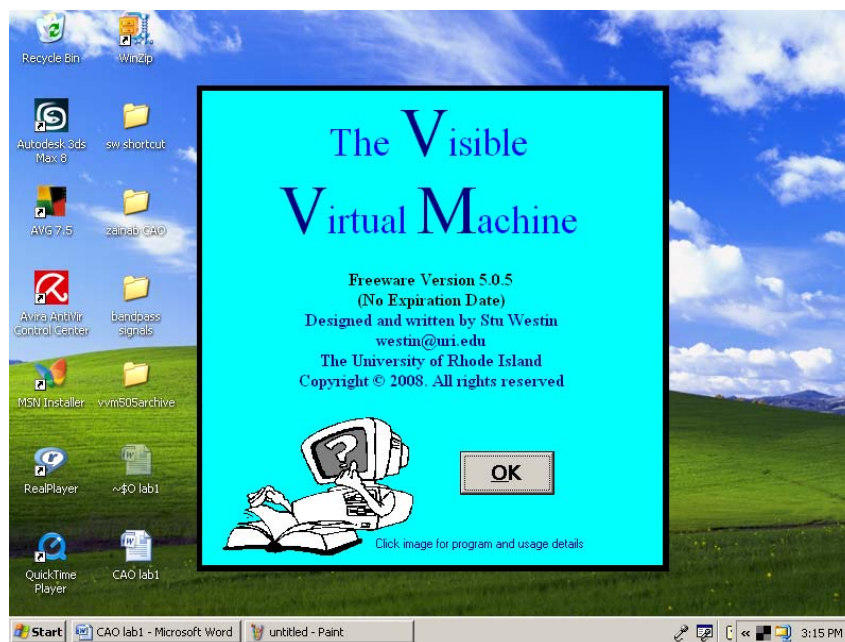
A simple VVM Assembly Language program which adds an input value to the constant value -1 is shown below (note that lines starting with “//” and characters to the right of program statements are considered comments, and are ignored by the VVM machine).

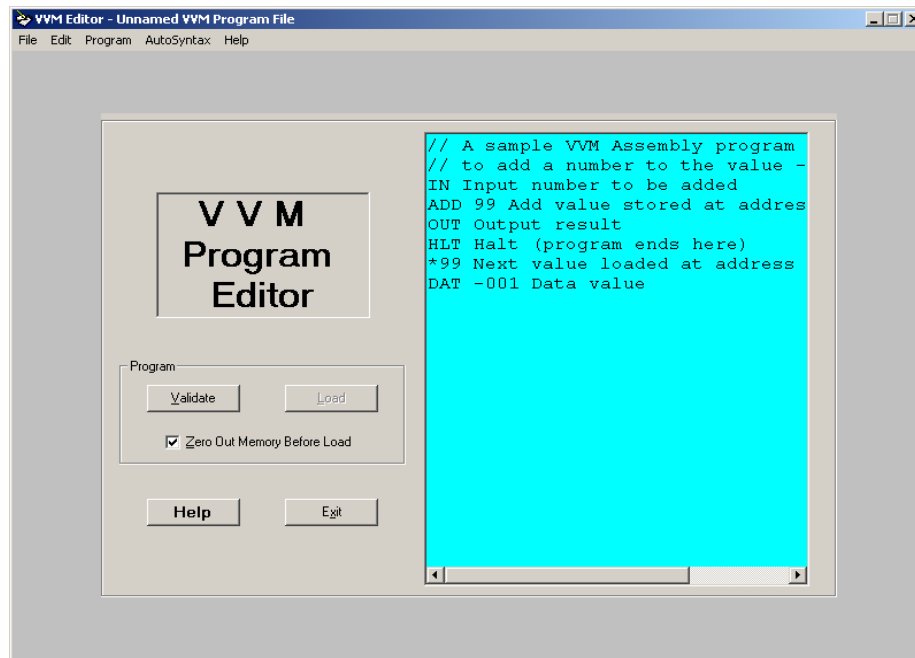
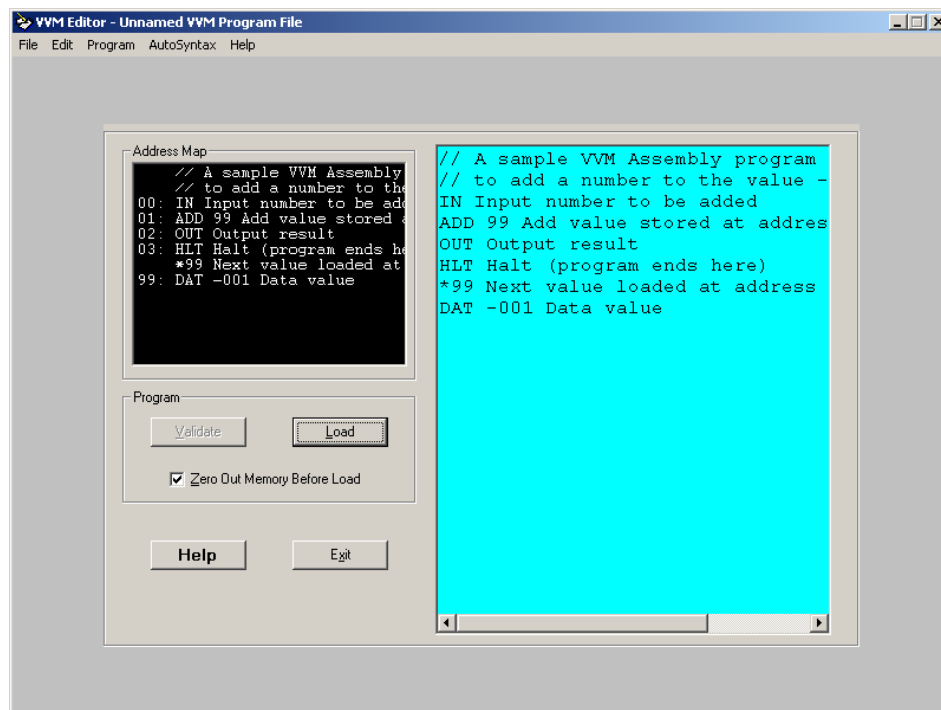
```
// A sample VVM Assembly program
// to add a number to the value -1.
IN Input number to be added
ADD 99 Add value stored at address 99 to input
OUT Output result
HLT Halt (program ends here)
*99 Next value loaded at address 99
DAT -001 Data value
```

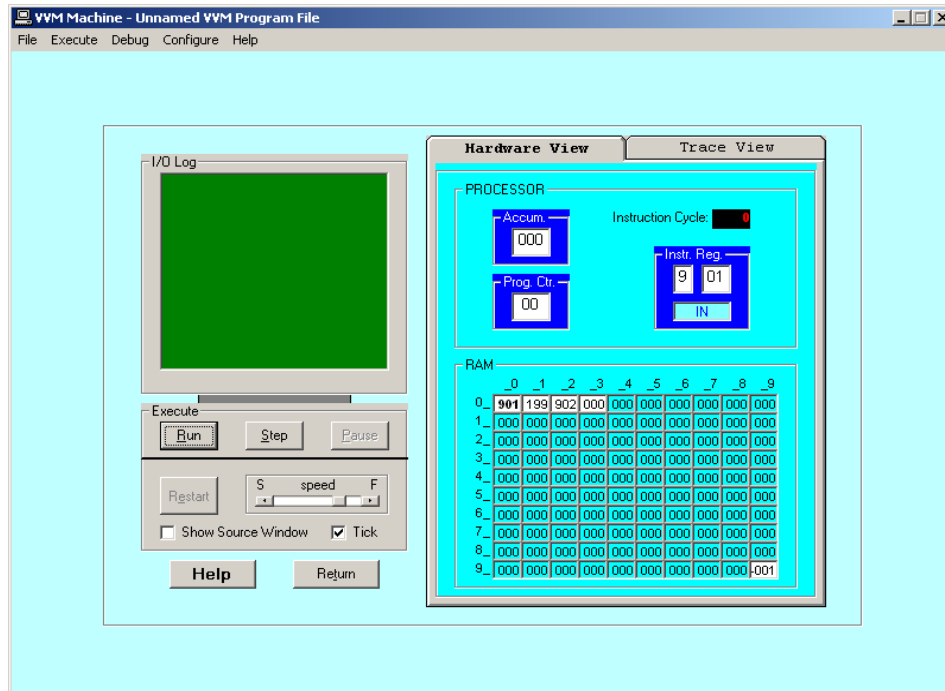
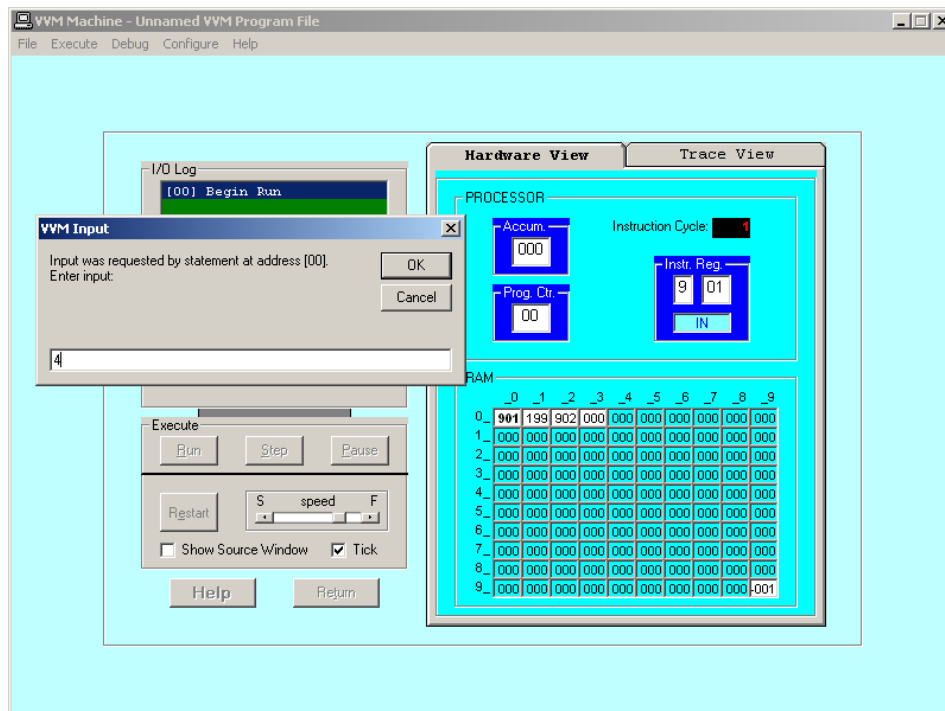
This same program could be written in VVM Machine Language format as follows:

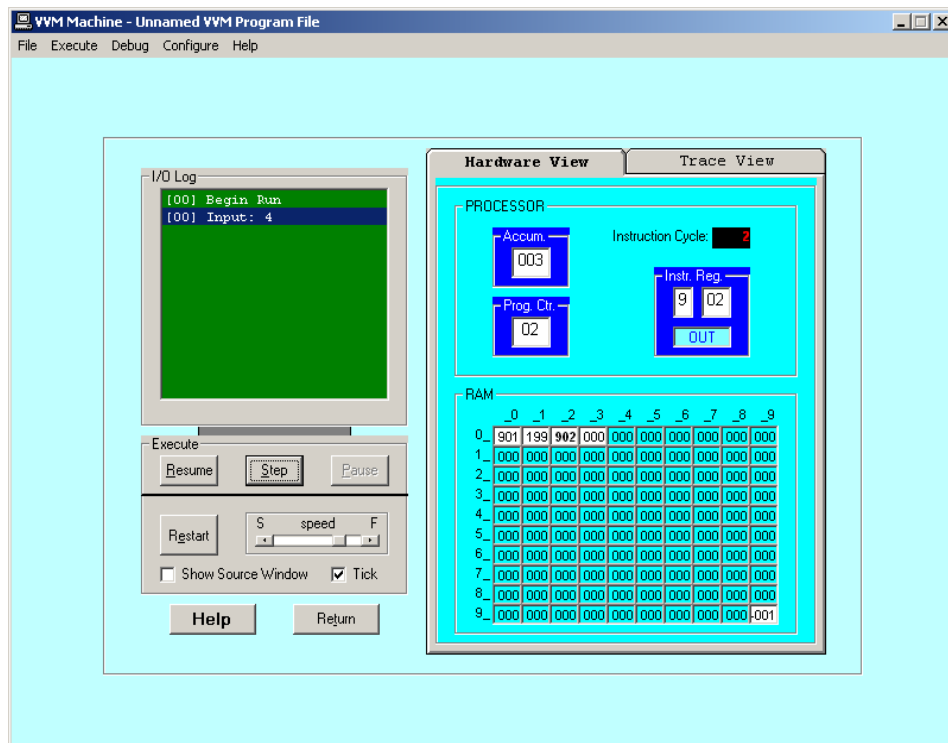
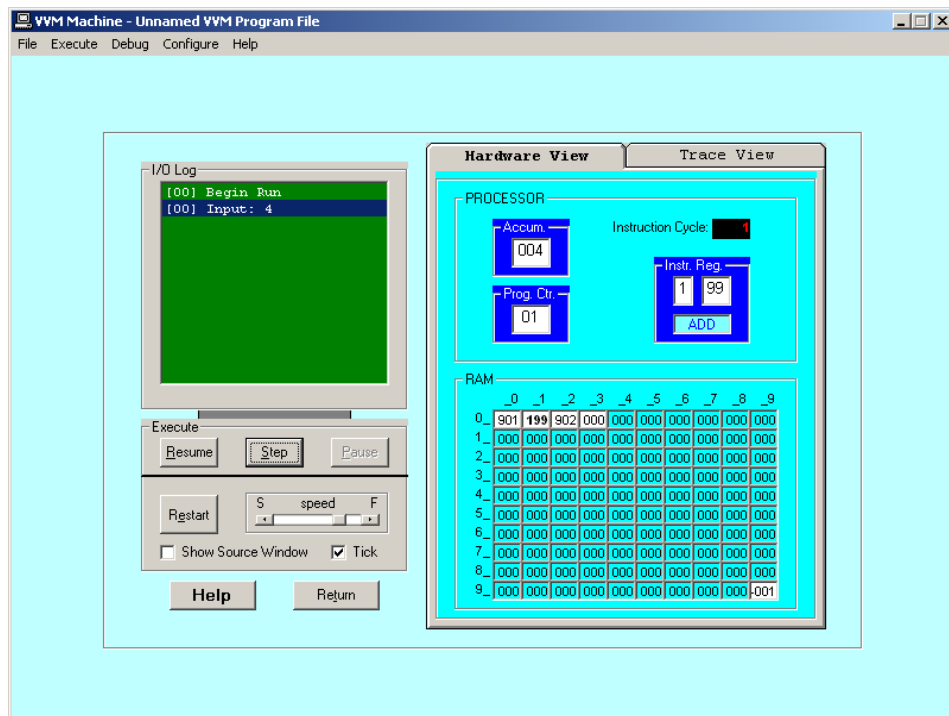
```
// The Machine Language version
901 Input number to be added
199 Add value stored at address 99 to input
902 Output result
000 Halt (program ends here)
*99 Next value loaded at address 99
-001 Data value
```

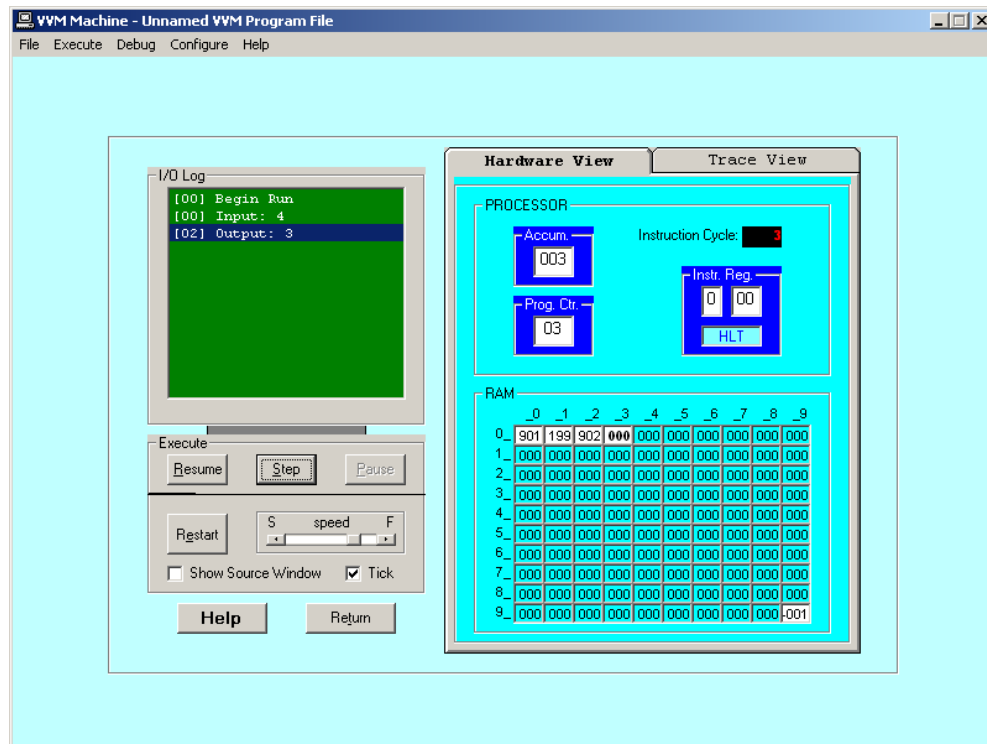
STEP1: Load VVM



STEP2: Copy the code and paste in the blue editor window**STEP3: Press the Validate button**

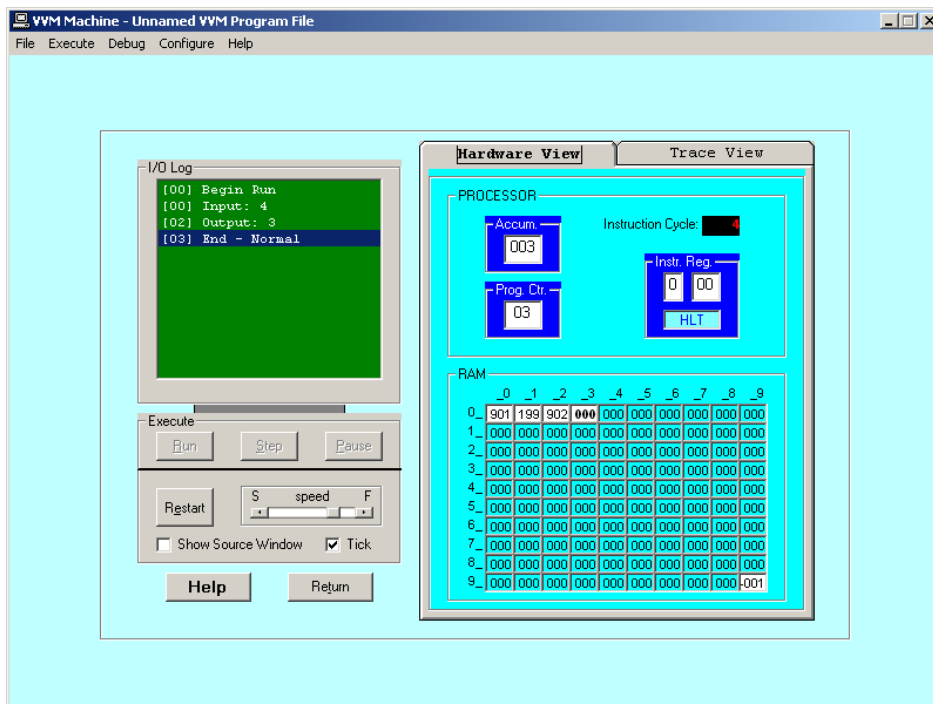
STEP4: Press the Load button**STEP5: Press the Step button**



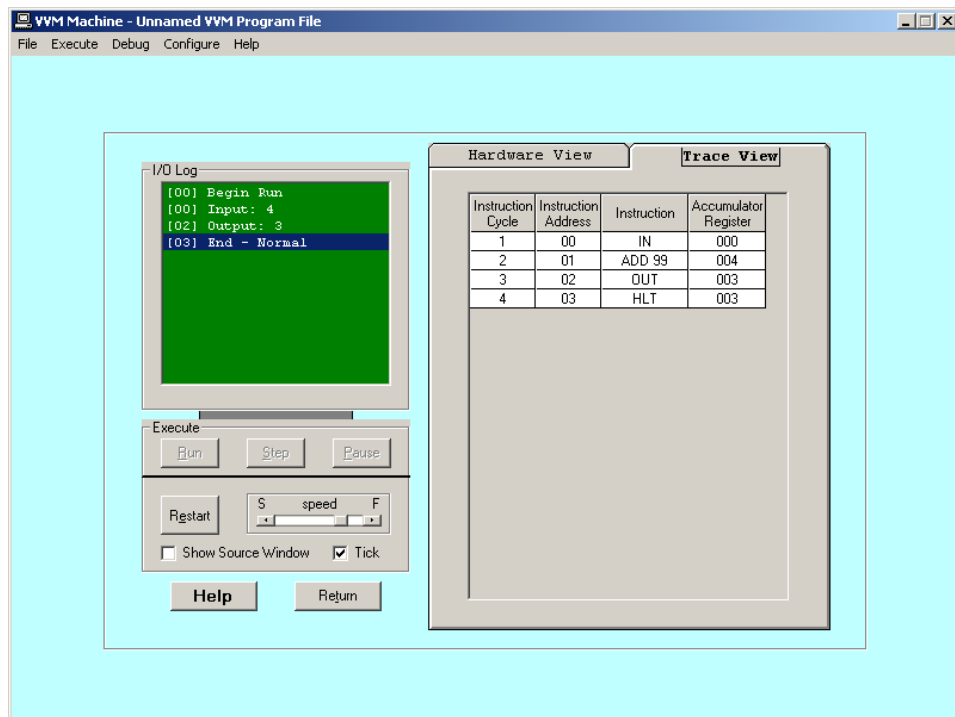


OUTPUT

Hardware View



Trace View



LAB TASK

1. To take input and Subtract.
2. To take two input as hardcore and Add them.

LAB # 2

INTRODUCTION TO VVM PROGRAMMING

OBJECTIVE

Learn VVM Programming and simulate VVM program.

THEORY

VVM Programming

VVM has its own simple Programming Language which supports such operations as conditional and unconditional branching, addition and subtraction, and input and output, among others. The language allows the student to create reasonably complex programs, and yet the language is quite easy to learn and to understand -- only eleven unique operations are provided. When VVM programs go awry, as in the case of endless loops or data overflows, VVM (virtual) system errors are triggered before the user's eyes.

VVM programs can be written in Machine Language, in Assembly Language, or in a combination of both. The Machine Language format is represented in decimal values, so there is no need for the student user to interpret long binary machine codes. In the Machine Language format, each instruction is a three-digit integer where the first digit specifies the operation code (op code), and the remaining two digits represent the operand. In the Assembly Language format, the operation code is replaced by a three-character mnemonic code. The two-digit operand usually represents a memory address. The sample program below is shown in both formats.

Following the automatic syntax validation process, VVM programs are converted to machine language format and loaded into the 100 data-word virtual RAM which is fully visible to the user during program execution.

The Language Instructions

The eleven operations of the VVM Language are described below. The Machine Language codes are shown in parentheses, while the Assembly Language version is in square brackets.

- **Load Accumulator (5nn) [LDA nn]** The content of RAM address *nn* is copied to the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.

- **Store Accumulator (3nn) [STO nn] (or [STA nn])** The content of the Accumulator Register is copied to RAM address *nn*, replacing the current content of the address. The content of the Accumulator Register remains unchanged. The Program Counter Register is incremented by one.
- **Add (1nn) [ADD nn]** The content of RAM address *nn* is added to the content of the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.
- **Subtract (2nn) [SUB nn]** The content of RAM address *nn* is subtracted from the content of the Accumulator Register, replacing the current content of the register. The content of RAM address *nn* remains unchanged. The Program Counter Register is incremented by one.
- **Input (901) [IN] (or [INP])** A value input by the user is stored in the Accumulator Register, replacing the current content of the register. The Program Counter Register is incremented by one.
- **Output (902) [OUT] (or [PRN])** The content of the Accumulator Register is output to the user. The current content of the register remains unchanged. The Program Counter Register is incremented by one.
- **Halt (0nn) [HLT] (or [COB])** Program execution is terminated. The operand value *nn* is ignored in this instruction and can be omitted in the Assembly Language format.
- **Branch if Zero (7nn) [BRZ nn]** This is a conditional branch instruction. If the value in the Accumulator Register is zero, then the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. Otherwise (Accumulator $\neq 0$), the Program Counter Register is incremented by one, and the next sequential instruction is executed.
- **Branch if Positive or Zero (8nn) [BRP nn]** This is a conditional branch instruction. If the value in the Accumulator Register is positive or zero, then the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. Otherwise (Accumulator < 0), the Program Counter Register is incremented by one, and the next sequential instruction is executed.
- **Branch (6nn) [BR nn] (or [BRU nn] or [JMP nn])** This is an unconditional branch instruction. The current value of the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be

executed will be taken from address *nn* rather than from the next sequential address. The value of the Program Counter Register is not incremented with this instruction.

- **No Operation (4nn) [NOP] (or [NUL])** This instruction does nothing other than increment the Program Counter Register by one. The operand value *nn* is ignored in this instruction and can be omitted in the Assembly Language format. (This instruction is unique to the VVM and is not part of the original Little Man Model.)

Embedding Data in Programs

Data values used by a program can be loaded into memory along with the program. In Machine or Assembly Language form simply use the format “*snnn*” where *s* is an optional sign, and *nnn* is the three-digit data value. In Assembly Language, you can specify “DAT *snnn*” for clarity.

The VVM Load Directive

By default, VVM programs are loaded into sequential memory addresses starting with address 00. VVM programs can include an additional load directive which overrides this default, indicating the location in which certain instructions and data should be loaded in memory. The syntax of the Load Directive is “**nn*” where *nn* represents an address in memory. When this directive is encountered in a program, subsequent program elements are loaded in sequential addresses beginning with address *nn*.

Program#1

Simple conditional structure using “brp” & “br” instruction.

```
in      Input A
sto 98  Store A
in      Input B
sto 99  Store B
lda 98  Load value of A
sub 99  Subtract B from A
brp 11  If A >= B, branch to 11 A is < B Find difference
lda 98  Load value of A
sub 99  Subtract value of B
sto 97  Store C
br 14   Jump to 14
lda 98  [11] Load A (A is >= B)
add 99  Add B
sta 97  Store C
out [14] Print result
hlt     Done
```

Equivalent BASIC program:

```
INPUT A
INPUT B
IF A >= B THEN
C = A + B
ELSE
C = A - B
ENDIF
PRINT C
END
```

LAB TASKS

1. Take any integer as input, if the number is greater than 5 print it
If $a > 5$, print a
Else if $a = 0$, then Halt
Else if $a < 5$, then halt
2. Take two numbers as input and print the larger number.

LAB # 3

VVM PROGRAMMING

OBJECTIVE

VVM Programs using “BRP”, “BRZ” & “BR” instructions.

THEORY

BRP *nn*

Branch if Positive or Zero (8nn) [BRP *nn*] This is a conditional branch instruction. If the value in the Accumulator Register is positive or zero, then the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. Otherwise (Accumulator < 0), the Program Counter Register is incremented by one, and the next sequential instruction is executed.

BRZ *nn*

Branch if Zero (7nn) [BRZ *nn*] This is a conditional branch instruction. If the value in the Accumulator Register is zero, then the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. Otherwise (Accumulator \neq 0), the Program Counter Register is incremented by one, and the next sequential instruction is executed.

BR *nn*

Branch (6nn) [BR *nn*] (or [BRU *nn*] or [JMP *nn*]) This is an unconditional branch instruction. The current value of the Program Counter Register is replaced by the operand value *nn*. The result is that the next instruction to be executed will be taken from address *nn* rather than from the next sequential address. The value of the Program Counter Register is not incremented with this instruction.

VVM Program 1

Simple looping example.

```

In      Input A
sto 99  Store A
brp 04  [02] If A >= 0 then skip next
br 10   Jump out of loop (Value < 0)
brz 10  [04] If A = 0 jump out of loop
lda 99  Load value of A (don't need to)
out     Print A
in      Input new A
sto 99  Store new value of A
br 02   Jump to top of loop
hlt     [10] Done

```

Equivalent to the following BASIC program:

```

INPUT A
DO WHILE A > 0
PRINT A
INPUT A
LOOP
END

```

VVM Program 2

Sample program to print the square of any integer in the range 1-31.

```

in      Input value to be squared
sto 99  Store input at 99
lda 98  Load current sum (top of loop)
add 99  Add value to sum
sto 98  Store the sum
lda 97  Load current index
add 96  Add 1 to index
sto 97  Store new index value
sub 99  Subtract value from index
brz 11  Jump out if index = value
br 02   Do it again (bottom of loop)
lda 98  Done looping - load the sum
out     Display the result
hlt     Halt (end of program)
// Data used by program follows
*96     Resume loading at address 96
dat 001 Constant for counting
dat 000 Initial index value
dat 000 Initial sum

```

LAB TASK

Write a VVM program which take an integer input and display table of that integer.

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:

Register	Alias	Usage	Register	Alias	Usage
\$0	\$zero	constant 0	\$16	\$s0	saved temporary
\$1	\$at	used by assembler	\$17	\$s1	saved temporary
\$2	\$v0	function result	\$18	\$s2	saved temporary
\$3	\$v1	function result	\$19	\$s3	saved temporary
\$4	\$a0	argument 1	\$20	\$s4	saved temporary
\$5	\$a1	argument 2	\$21	\$s5	saved temporary
\$6	\$a2	argument 3	\$22	\$s6	saved temporary
\$7	\$a3	argument 4	\$23	\$s7	saved temporary
\$8	\$t0	unsaved temporary	\$24	\$t8	unsaved temporary
\$9	\$t1	unsaved temporary	\$25	\$t9	unsaved temporary
\$10	\$t2	unsaved temporary	\$26	\$k0	reserved for OS kernel
\$11	\$t3	unsaved temporary	\$27	\$k1	reserved for OS kernel
\$12	\$t4	unsaved temporary	\$28	\$gp	pointer to global data
\$13	\$t5	unsaved temporary	\$29	\$sp	stack pointer
\$14	\$t6	unsaved temporary	\$30	\$fp	frame pointer
\$15	\$t7	unsaved temporary	\$31	\$ra	return address

Introduction to MIPS Assembly Language

Assembly Language Program Template

```

# Title:                               Filename:
# Author:                             Date:
# Description:
# Input:
# Output:
##### Data segment #####
.data
...
...
##### Code segment #####
.text
.globl main
main:
...                # main program entry
...
li $v0, 10          # Exit program
svscall

```

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)

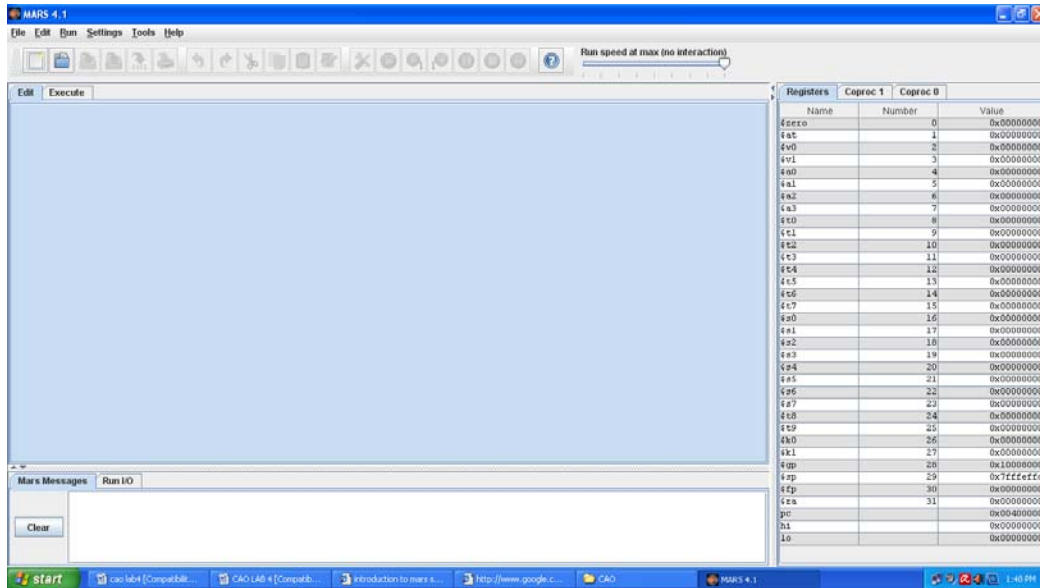
- ❖ Programs do input/output through system calls
- ❖ MIPS provides a special **syscall** instruction
 - To obtain services from the operating system
 - Many services are provided in the MARS simulators
 - There are 10 different services provided.
- ❖ Using the **syscall** system services
 - Load the 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	Supported by MARS
Read Char	12	\$v0 = character read	

Introduction to MARS

MARS, the **MIPS Assembly and Runtime Simulator**, 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.

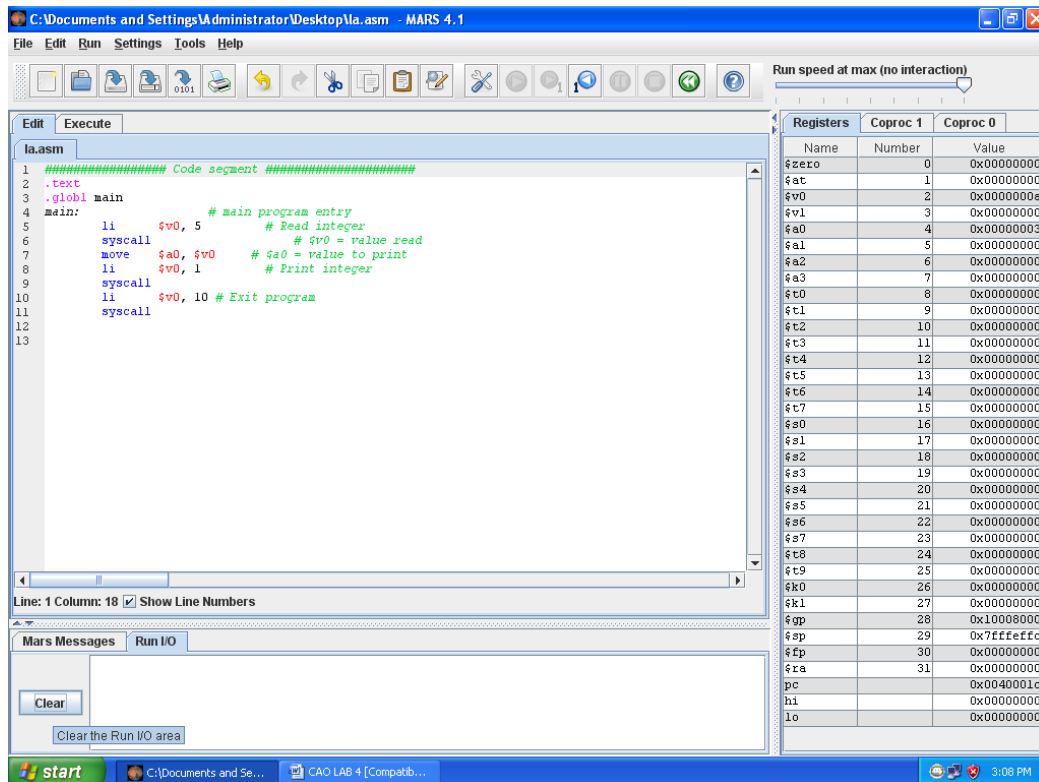
The assembler and simulator are invoked from the IDE when you select the *Assemble*, *Go*, or *Step* operations from the *Run* menu or their corresponding toolbar icons or keyboard shortcuts. MARS messages are displayed on the *MARS Messages* tab of the message area at the bottom of the screen. Runtime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

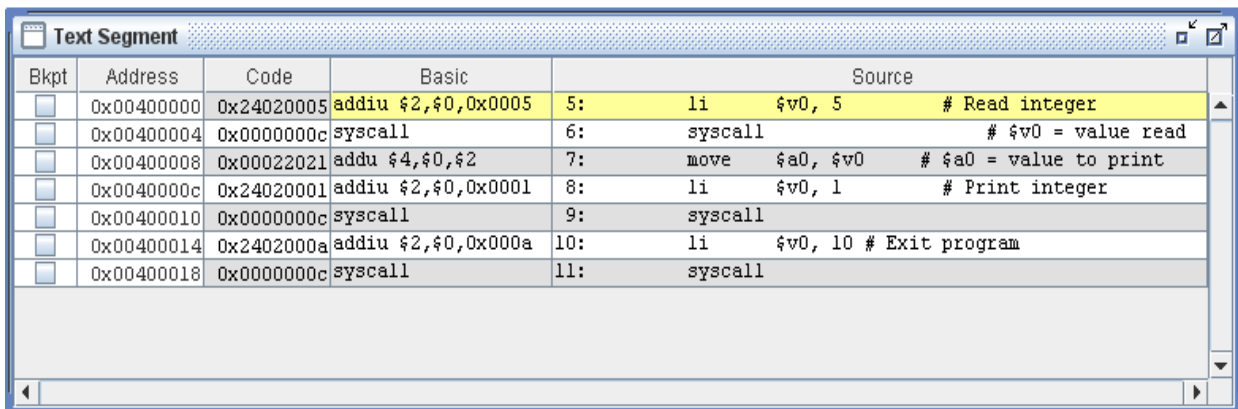
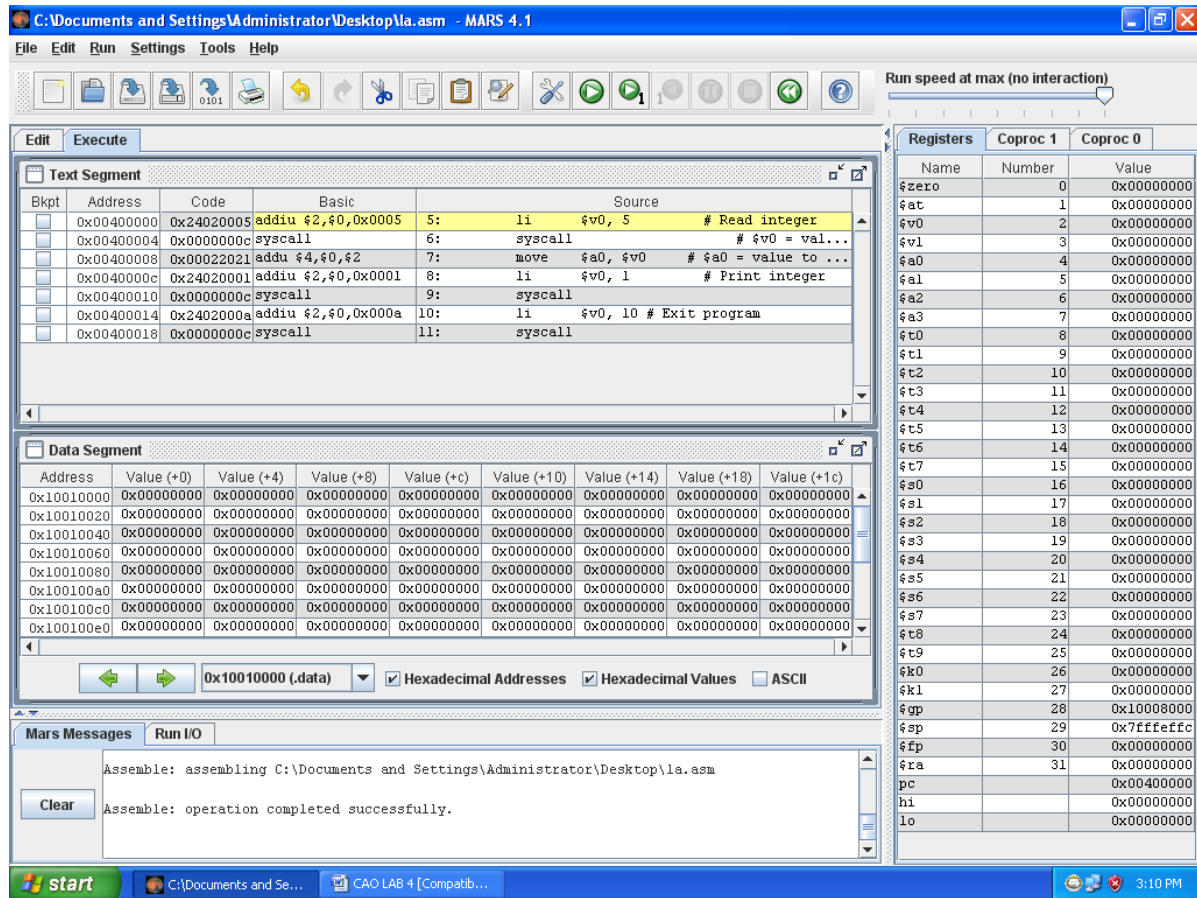
STEP# 1

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



STEP# 2

Assemble program by pressing F3.



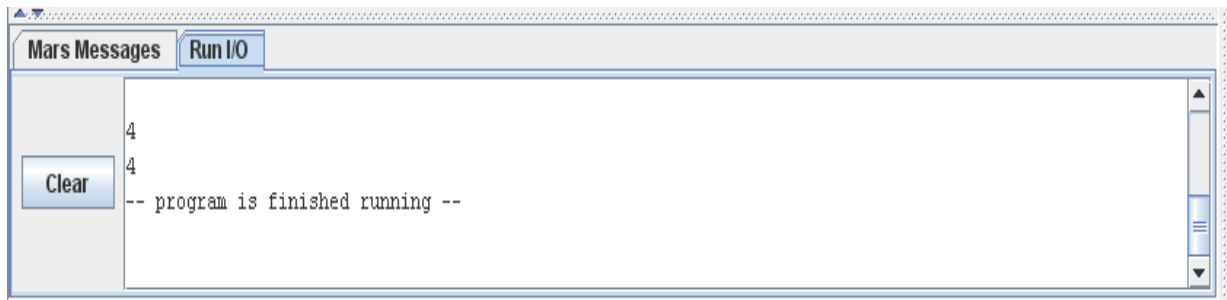
Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☒ Hexadecimal Addresses ☒ Hexadecimal Values ☐ ASCII

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.

LAB # 5

MIPS ASSEMBLY LANGUAGE ARITHMETIC OPERATIONS

OBJECTIVE

Performing Arithmetic and String functions in MIPS.

Program#1:

PRINTING NAME IN MIPS

```
# Objective: Printing name in MIPS
# Input: Saving name.
# Output: Printing Name.
##### Data segment #####
.data
prompt:      .asciiz  "Ali\n"
##### Code segment #####
.text
.globl main
main:
Li $v0,4
La $a0,message
Syscall
Li $v0,10
syscall
```

Program#2:**ADDING NUMBERS****# Objective: Adding Numbers in MIPS****# Input: Two numbers.****# Output: Printing Sum.****##### Data segment #####****.data****A1: .word 10****A2: .word 10****##### Code segment #####****.text****lw \$t0,A1(\$zero)****lw \$t1,A2(\$zero)****add \$t2,\$t0,\$t1****li \$v0,1****add \$a0,\$zero,\$t2****syscall****LAB TASK**

Task 1: Write an assembly program that Read and Print Hello world.

Task 2: Write an assembly program that add two number within the range of 10.

Task 3: Write an assembly program that subtract two number within the range of 10.

LAB # 6

MIPS ASSEMBLY LANGUAGE USING ADDU INSTRUCTION

OBJECTIVE

Take three-integer input and display sum of three integers using “addu” instruction.

THEORY

```
addu Rdest,Rsrc1,Rsrc2
```

Adds the contents of Rsrc1 and Rsrc2 and stores the result in Rdest. The numbers are treated as unsigned integers. No overflow exception is needed.

Program#1:

Sum of Three Integers

Objective: Computes the sum of three integers.

Input: Requests three numbers.

Output: Outputs the sum.

Data segment

.data

prompt: .asciiz "Please enter three numbers: \n" sum_msg: .asciiz

"The sum is: "

Code segment

.text

.globl main main:

 la \$a0,prompt # display prompt string

li \$v0,4 syscall

 li \$v0,5 # read 1st integer into \$t0 syscall

move \$t0,\$v0

 li \$v0,5 # read 2nd integer into \$t1 syscall

move \$t1,\$v0

 li \$v0,5 # read 3rd integer into \$t2

syscall move \$t2,\$v0

```
addu $t0,$t0,$t1    # accumulate the sum
addu $t0,$t0,$t2    la  $a0,sum_msg
                   # write sum message    li  $v0,4
syscall
move  $a0,$t0 # output sum
li    $v0,1
syscall
li    $v0,10      # exit
syscall
```

LAB TASK

Task 1: Write the same program with small variation i.e. this time the program will ask for 3 integers twice and displays the result for each addition separately;

Output will look like as follows:

Enter 3 integers for 1st addition

2

2

2

Enter 3 integers for 2nd addition

3

3

3

The sum of 1st addition is 6

The sum of 2nd addition is 9

Task 2: Write an assembly program that Multiply two number within the range of 10.

Task 3: Write an assembly program that Divide two number within the range of 10.

LAB # 7

ARITHMETIC INSTRUCTIONS IN MIPS

OBJECTIVE

Write a program using “addi”, “blez” and “bnez” instructions.

THEORY

addi Rdest,Rsrc1, imm

Rdest = Rsrc1 + 16-bit signed immediate. In case of an overflow, an overflow exception is generated.

blez Rs, Label

Branch if Less Than or Equal to Zero.

if ($Rs \leq 0$) go to Label.

bnez Rs, Label

Branch if Not Equal to Zero.

if ($Rs \neq 0$) go to Label

Program#1:

Sum of Positive Integers

Objective: Computes the sum of all positive integers \leq the input number.

Input: Requests input number.

Output: Outputs the sum.

Data segment

.data prompt: .asciiz "\n Please Input a value for N = "

result: .asciiz " The sum of the integers from 1 to N is "

bye: .asciiz "\n **** Have a good day ****"

Code segment

.text

.globl main main:

li \$v0, 4 # system call code for Print String

la \$a0, prompt # load address of prompt into \$a0 syscall

print the prompt message

```

        li $v0, 5          # system call code for Read
Integer  syscall           # reads the value of N into
$v0      blez $v0, end     # branch to end if $v0 <
= 0      li $t0, 0         # clear register $t0 to zero

loop:    add $t0, $t0, $v0  # sum of integers in
register $t0      addi $v0, $v0, -1  # summing
integers in reverse order
bnez $v0, loop      # branch to loop if $v0 is != zero

        li $v0, 4          # system call code for Print String
la $a0, result      # load address of message into $a0
syscall            # print the string

        li $v0, 1          # system call code for Print Integer      move
$a0, $t0           # move value to be printed to $a0      syscall
# print sum of integers

        end:    li $v0, 4          # system call code for
Print String      la $a0, bye      # load address of
message into $a0  syscall          # print the string

        li $v0, 10         # terminate program run and
syscall           # return control to system

```

LAB TASK

Task 1: Write a MIPS assembly language program that calculates the factorials.

Task 2: Write a MIPS assembly language program that Generate table of 5.

LAB # 8

MULTIPLICATION AND DIVISION INSTRUCTIONS IN MIPS

OBJECTIVE

Study Multiplication, Division Instructions.

THEORY

Multiplication

The multiply instruction multiplies two 32-bit binary values and produces a 64-bit product which is stored in two registers named High and Low. The following code segment shows how the lower 32 bits of the product of \$t0 times \$t1 can be moved into \$t3:

```
mult $t0, $t1
mflo $t3    # t3 = Lower 32-bits of product
mfhi $t4    # t4 = Higher 32-bits of product
```

Program#1:

Multiplication of Two Integers

Objective: Computes and display result of the multiplication of two integers.

Data segment

.data

prompt : .ascii "result of multiplication is\n"

Code segment

.text

.globl main # main program entry

main:

 li \$t0,99 #\$t0=99

li \$t1,99 #\$t1=99

 mult \$t0, \$t1 # \$t0*\$t1

 mflo \$t2 # \$t2 = Lower 32-bits of product

 la \$a0,prompt # Print message of prompt

li \$v0,4 syscall

```

move $a0,$t2  #move value of #t2 into $a0
li $v0,1      # Print integer
syscall

li $v0,10     # Exit program
syscall

```

Output:


```

RESULT OF MULTIPLICATION IS
9801
-- PROGRAM IS FINISHED RUNNING --

```

Division

Divide instruction divides the 32-bit binary value in register \$t0 by the 32-bit value in register \$t1. The quotient is stored in the Low register and the remainder is stored in the High register. The following code segment shows how the quotient is moved into \$t2 and the remainder is moved into \$t3:

```

div $t0, $t1
mflo $t2  # here lower register $t2 contain the quotient
mfhi $t3  # here high register $t3 contain the remainder

```

Program#2:**Division of Two Integers**

Objective: Computes and display result of the division of two integers.

Data segment

.data

prompt: .asciiz "Quotient is : "

prompt1: .asciiz "\nRemainder is: "

Code segment

.text

.globl main # main program entry

main:

```

    li $t0,42      #$t0=42
    li $t1,2       #$t1=2

    div $t0,$t1     mflo $t2      # here lower register $t2
    contain the quotient. mfhi $t3 # here high register $t3
    contain the remainder. la $a0,prompt # Print message of
    prompt          li $v0,4
                    syscall

    move $a0,$t2     #move value of #t2 into
    $a0          li $v0,1      # Print value of
    quotient      syscall

    la $a0,prompt1   # Print message of
    prompt1        li $v0,4      syscall

    move $a0,$t3     #move value of #t3 into $a0
    li $v0,1         # Print value of remainder
    syscall

    li $v0,10        # Exit program
    syscall

```

Output:



```

Quotient is : 21
Remainder is : 0
-- program is finished running --

```

CONDITIONAL AND UNCONDITIONAL BRANCH INSTRUCTIONS

Instructions	Description
Branch if greater than or equal to zero bgez rs, L	if ($rs \geq 0$) go to L;
Branch if greater than zero bgtz rs, L	if ($rs > 0$) go to L;

Branch if less than or equal to zero	
blez rs, L	if ($rs \leq 0$) go to L;
Branch if less than zero	
bltz rs, L	if ($rs < 0$) go to L;
Branch if not equal	
bne rs, rt, L	if ($rs \neq rt$) go to L;
Branch if equal	
beq rs, rt, L	if ($rs == rt$) go to L;
Set less than	
slt rd, rs, rt	if ($rs < rt$) $rd=1$; else $rd=0$; rs and rt are <i>signed</i> integers.
Set less than unsigned	
sltu rd, rs, rt	Same as slt except rs and rt are <i>unsigned</i> integers.
Set less than immediate	
slti rt, rs, immediate	if ($rs < \text{signed immediate}$) $rd=1$; else $rd=0$;
Set less than immediate unsigned	
sltiu rt, rs, immediate	if ($rs < \text{unsigned immediate}$) $rd=1$; else $rd=0$;

LAB TASK

Write a MIPS assembly language program that takes an integer input and multiply that integer with 3 and display the message that is the result is even or odd.

LAB # 9

BIT MANIPULATION INSTRUCTIONS IN MIPS

OBJECTIVE

Learn to use MIPS bit manipulation instructions in assembly language programs.

THEORY

BITWISE LOGICAL INSTRUCTIONS

Instructions		Description
and	rd, rs, rt	$rd = rs \& rt$
andi	rt, rs, immediate	$rt = rs \& \text{immediate}$
or	rd, rs, rt	$rd = rs rt$
ori	rt, rs, immediate	$rd = rs \text{immediate}$
nor	rd, rs, rt	$rd = ! (rs rt)$
xor	rd, rs, rt	To do a bitwise logical Exclusive OR.
xori	rt, rs, immediate	

The main usage of bitwise logical instructions are: *to set, to clear, to invert*, and to *isolate* some selected bits in the destination operand. To do this, a source bit pattern known as a mask is constructed. The Mask bits are chosen based on the following properties of AND, OR, and XOR with Z represents a bit (either 0 or 1):

AND	OR	XOR
$Z \text{ AND } 0 = 0$	$Z \text{ OR } 0 = Z$	$Z \text{ XOR } 0 = Z$
$Z \text{ AND } 1 = Z$	$Z \text{ OR } 1 = 1$	$Z \text{ XOR } 1 = \sim Z$

AND Instruction

The AND instruction can be used to CLEAR specific destination bits while preserving the others. A zero mask bit clears the corresponding destination bit; a one mask bit preserves the corresponding destination bit.

OR Instruction

The OR instruction can be used to SET specific destination bits while preserving the others. A one mask bit sets the corresponding destination bit; a zero mask bit preserves the corresponding destination bit.

XOR Instruction

The XOR instruction can be used to INVERT specific destination bits while preserving the others. A one mask bit inverts the corresponding destination bit; a zero mask bit preserves the corresponding destination bit.

Program#1:

Performing Bitwise AND Instruction with Mask 1

Objective: Performs bitwise AND instruction with Mask 1.

Data segment

**.data input: .asciiz "\n enter an integer value: " # variable
declaration**

result: .asciiz "\n result is: "

Code segment

.text

.globl main main:

li \$t0,0xffffffff # 1 Mask

la \$a0,input # Print input message

li \$v0,4 syscall

li \$v0,5 # user input

syscall

move \$t1,\$v0

and \$t2,\$t1,\$t0 # AND instruction, \$t2 = \$t1 AND \$t0

la \$a0,result # Print result

message li \$v0,4 syscall

move \$a0,\$t2 # Move AND instruction result in \$a0

li \$v0,1 # Print value of \$t2

syscall

li \$v0,10 # Exit program

syscall

Output:

The screenshot shows the Mars Messages window on the left and the MIPS Register window on the right. The Messages window displays 'Enter number:' followed by '12' and a 'Clear' button. The Register window shows the state of MIPS registers. Register \$t0 is highlighted in green and labeled 'MASK'. Register \$t1 is highlighted in green and labeled 'INPUT'. Register \$t2 is highlighted in green and labeled 'RESULT'.

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x0000000a
\$v1	3	0x00000000
\$a0	4	0x0000000c
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0xffffffff
\$t1	9	0x0000000c
\$t2	10	0x0000000c

Program#2:**Performing Bitwise AND Instruction with Mask 0**

Objective: Performs bitwise AND instruction with Mask 0.

Data segment

.data input: .asciiz "\n enter an integer value: " # variable
declaration

result: .asciiz "\n result is: "

Code segment

.text

.globl main

main:

li \$t0,0x00000000 # 0 Mask

la \$a0,input # Print input message

li \$v0,4

syscall

li \$v0,5 # user

input

syscall

move \$t1,\$v0

and \$t2,\$t1,\$t0 # AND instruction, \$t2 = \$t1 AND \$t0

la \$a0,result # Print result message

li \$v0,4

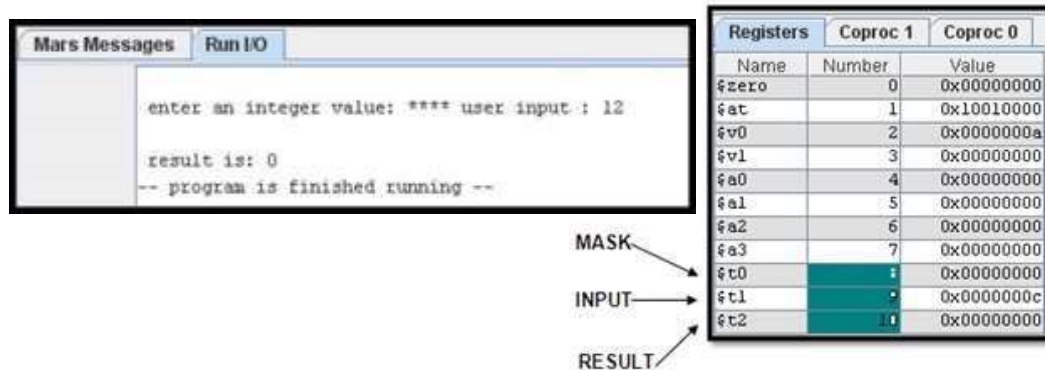
syscall

move \$a0,\$t2 # Move AND instruction result in \$a0

```
li $v0,1          # Print value of $t2
syscall
```

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

Output:



Lab Task:

Complete the table by solving the bitwise instruction of all Logical gates. Add the code and output of the logical gates to show solution of MASK BITS given in the table.

Logic	Mask Bits	
	0	1
AND		
OR		
NOT		
XOR		
XNOR		
NOR		
NAND		

LAB # 10

IF THEN ELSE; CONTROL STRUCTURE IN MIPS

OBJECTIVE

Study how to implement translation of an “if then else” control structure in MIPS assembly language.

THEORY

Translation of an “IF THEN ELSE” Control Structure

The “**If** (condition) **then** do {this block of code} **else** do {that block of code}” control structure is probably the most widely used by programmers. Let us suppose that a programmer initially developed an algorithm containing the following pseudo code.

```
if ($t8 < 0) then
    { $s0 = 0 - $t8;
    $t1 = $t1 + 1 } else
    { $s0 = $t8;
    $t2 = $t2 + 1 }
```

When the time comes to translate this pseudo code to MIPS assembly language the results could appear as shown below. In MIPS assembly language, anything on a line following the number sign (#) is a comment. Notice how the comments in the code below help to make the connection back to the original pseudo code.

```
        bgez $t8, else      # if ($t8 is > or = zero) branch to
else      sub $s0, $zero, $t8  # $s0 gets the negative of $t8
addi $t1, $t1, 1           # increment $t1 by 1      b next
# branch around the else code else:      ori $s0, $t8, 0
# $s0 gets a copy of $t8      addi $t2, $t2, 1      #
increment $t2 by 1 next:
```

BRANCH IF GREATER THAN ZERO

```
bgez $v0,else # branch to else if $v0>=0
```

SHIFT LEFT LOGICAL

```
sll $t2,$v0,2 # set $t2 to result of shifting $v0 left by number specified by
              immediate
```

SHIFT RIGHT LOGICAL

```
srl $t2,$v0,2 # set $t2 to result of shifting $v0 right by number specified by
              immediate
```

Program#1:**Translation of an IF THEN ELSE Control Structure**

Objective: Translates an IF THEN ELSE control structure.

Data segment

.data

input: .asciiz "\n type any number" rshift:

.asciiz "\n number after right shift: "

lshift: .asciiz "\n number after left shift: "

Code segment

.text

.globl main

main:

la \$a0,input # print input message

li \$v0,4

syscall

la \$v0,5 # read integer

syscall

bgez \$v0,else # if-else equivalent statement

srl \$t2,\$v0,2 # shift right logical

la \$a0,rshift # print value of rshift

move \$a0,\$t2 # print value after right shift

b end # branch to statement at end unconditionally

IF (\$v<0) THEN
\$t2=\$v0>>2
Print: display \$t2
ELSE
\$t2=\$v0<<2
Print: display \$t2

li \$v0,4

syscall

li \$v0,1

syscall

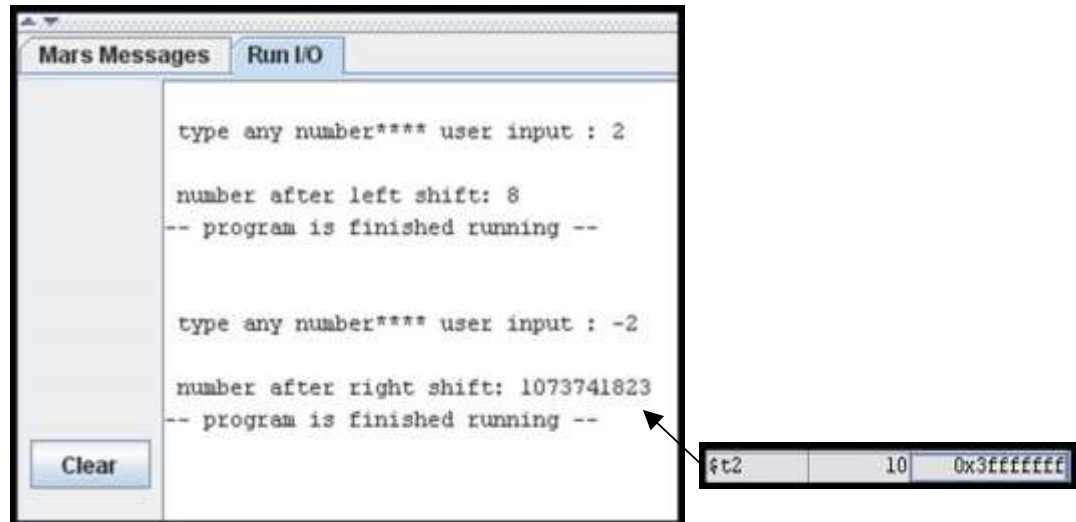
else:

```
    sll $t2,$v0,2    # shift left logical
    la $a0,lshift    # print value of lshift
    li $v0,4
    syscall

    move $a0,$t2     # print value after left shift
    li $v0,1
    syscall

end:   li $v0,10      # terminate
program syscall
```

Output:



LAB TASK

Write a program in MIPS assembly language that takes input from user and print whether the input is greater or less than 10 and also shift input left and right 4 bits.

LAB # 11

FOR LOOP; CONTROL STRUCTURE IN MIPS

OBJECTIVE

Study how to implement translation of “for loop” control structure in MIPS assembly language.

THEORY

Translation of a “FOR LOOP” Control Structure

Obviously a “for loop” control structure is very useful. Let us suppose that a programmer initially developed an algorithm containing the following pseudo code. In one sentence, can you describe what this algorithm accomplishes?

```
$a0 = 0; for ( $t0 =10; $t0 > 0; $t0
= $t0 -1)
do { $a0 = $a0 + $t0 }
```

The following is a translation of the above “for loop” pseudo code to MIPS assembly language code.

```
li $a0, 0      # $a0 = 0 li $t0, 10    # Initialize loop
counter to 10 loop:  add $a0, $a0, $t0  addi $t0,
$t0, -1      # Decrement loop counter
      bgtz $t0, loop      # If ($t0 > 0) Branch to
loop
```

BRANCH IF GREATER THAN ZERO

```
bgtz $t0,loop    # branch to loop if $t0>0
```

Program#1:

Translation of a FOR LOOP Control Structure

Objective: Translates a FOR LOOP control structure.

```
##### Data segment #####
.data counter: .asciiz  "\n value of count,
$t0: " total:  .asciiz  "value of sum,
$a0: " tab:    .asciiz  "\t"
##### Code segment #####
```

```
.text
.globl main
main:
    li $a2,0          # $a2=0      li $t0,10          #
    # initialize loop variable counter $t0=10

loop:
    add $a2,$a2,$t0

    la $a0,counter     # print message of counter
    li $v0,4
    syscall

    move $a0,$t0       # print value of $t0
    li $v0,1
    syscall

    addi $t0,$t0,-1    # decrement loop variable counter

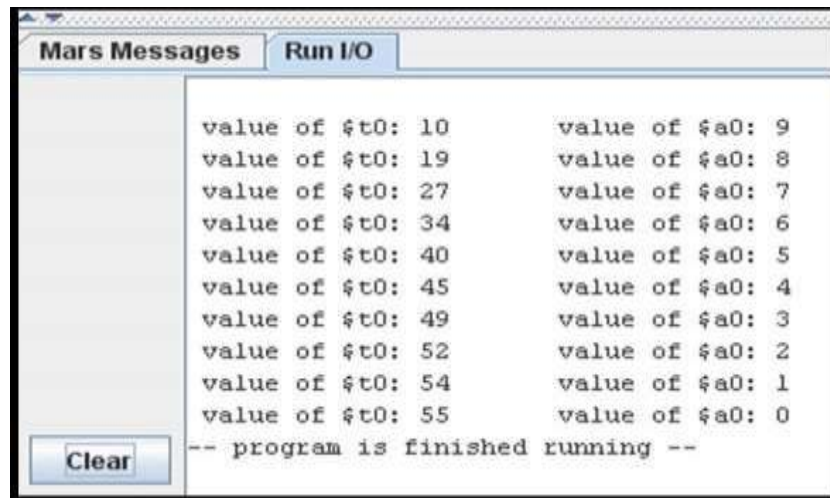
    la $a0,tab         # print
    li $v0,4
    tab               syscall

    la $a0,total       # print message of total
    li $v0,4
    syscall

    move $a0,$a2       # print value of $a2
    li $v0,1
    syscall

    bgtz $t0,loop      # if($t0>0) branch to loop

end:
    li $v0,10
    syscall
```

Output:

```
value of $t0: 10      value of $a0: 9
value of $t0: 19      value of $a0: 8
value of $t0: 27      value of $a0: 7
value of $t0: 34      value of $a0: 6
value of $t0: 40      value of $a0: 5
value of $t0: 45      value of $a0: 4
value of $t0: 49      value of $a0: 3
value of $t0: 52      value of $a0: 2
value of $t0: 54      value of $a0: 1
value of $t0: 55      value of $a0: 0
-- program is finished running --
```

LAB TASK

Task 1: Write a program in MIPS assembly language that takes input and display whether number is prime or not.

Task 2: Write a program in MIPS assembly language that provide the sum from 1 to 99 using for Loop

LAB # 12

SWITCH; CONTROL STRUCTURE IN MIPS

OBJECTIVE

Study how to implement translation of a “switch” control structure in MIPS assembly language.

THEORY

.align n

Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)

blez \$v0, label

Branch if less than or equal to zero: Branch to statement at label's address if \$t1 is less than or equal to zero

bgt \$v0,\$t3, label

Branch if Greater Than: Branch to statement at label if \$t1 is greater than \$t2

lw \$t2,0(\$t1)

Load Word: Set \$t2 to contents of effective memory word address

jr \$t2

jr \$t2 Jump register unconditionally: Jump to statement whose address is in \$t2

b label

Branch: Branch to statement at label unconditionally

Program#1:

Translation of a SWITCH Control Structure

Objective: Translates a SWITCH control structure.

Data segment

.data

```

.align 2 varword: .word
main,case1,case2,case3 input: .ascii
"\nType a value from 1 to 3 " msg_1: .ascii
"\n you are in case1" msg_2: .ascii "\n
you are in case2" msg_3: .ascii "\n you
are in case3"
##### Code segment #####
.text
.globl main
main:
    li $v0,4          # print input message
    la $a0,input
    syscall

    li $v0,5          # read integer
    syscall

    blez $v0,main     # default for less than 1
    li
    $t3,3

    bgt $v0,$t3,main  # default for greater than 3

    la $a1,varword    # load address of varword
    sll $t0,$v0,2      # compute word offset    add
    $t1,$a1,$t0       # form a pointer into variable    lw
    $t2,0($t1)        # load an address from varword
    jr $t2            # jump specific case "switch"
case1:    li $v0,4     la $a0,msg_1    syscall
        b end

    case2:    li
    $v0,4     la
    $a0,msg_2
    syscall
    b end

    case3:    li
    $v0,4     la
    $a0,msg_3
    syscall

```



```
end:  
li $v0,10  
syscall
```

Output:



The screenshot shows a window titled "Mars Messages" with a "Run I/O" button. The output text is as follows:

```
Type a value from 1 to 3 **** user input : 1  
  
you type 1  
-- program is finished running --  
  
Type a value from 1 to 3 **** user input : 2  
  
you type 2  
-- program is finished running --
```

LAB TASK

Write a program in MIPS assembly language by which you can show a total of 10 cases from case0 to case9.

LAB # 13

ARRAY; CONTROL STRUCTURE IN MIPS

OBJECTIVE

Study how to implement translation of a “Array” control structure in MIPS assembly language.

THEORY

.align n

Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)

beq \$v0, value, label

Branch on equal : Branch if the two registers are equal

lw \$t2,0(\$t1)

Load Word: Set \$t2 to contents of effective memory word address

sw \$s0, MyArray (\$t0)

store word in source register into RAM destination

b label

Branch: Branch to statement at label unconditionally

Program#1:

Translation of a ARRAY Control Structure

Objective: Translates a ARRAY control structure.

Data segment

.data

myArray: .space 12

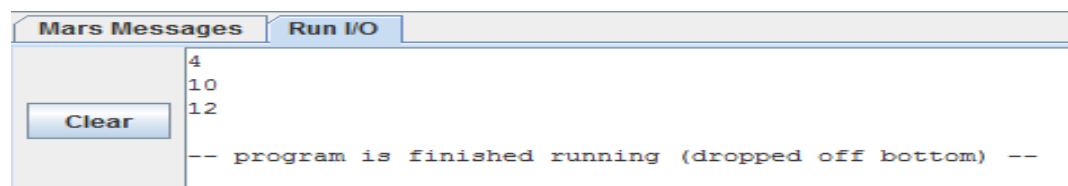
newline: .asciiz “\n”

Code segment

.text

```
.globl main
main:
addi $s0, $zero, 4
addi $s1, $zero
,10
addi $s2, $zero
,12
#Index= $t0
addi $t0, $zero, 0
Sw
$s0,myArray($t0
)
addi $t0, $t0, 4
Sw
$s1,myArray($t0
)
addi $t0, $t0, 4
Sw
$s2,myArray($t0
)
addi $t0, $zero, 0
While:
beq $t0, 12, exit
lw $t6,myArray(
$t0)
addi $t0, $t0,4
li $v0,1
move $a0, $t6
syscall
li $v0, 4
la $a0, newline
syscall
j While
exit:
li $v0, 10
```

Output:



LAB TASK

Write a program in MIPS assembly language which can read and print string by using array.

LAB # 14

DESIGNING GAME; TOWER OF HANOI

OBJECTIVE

The objective of this lab is to implement the Game of Tower of Hanoi in MIPS Assembly Language.

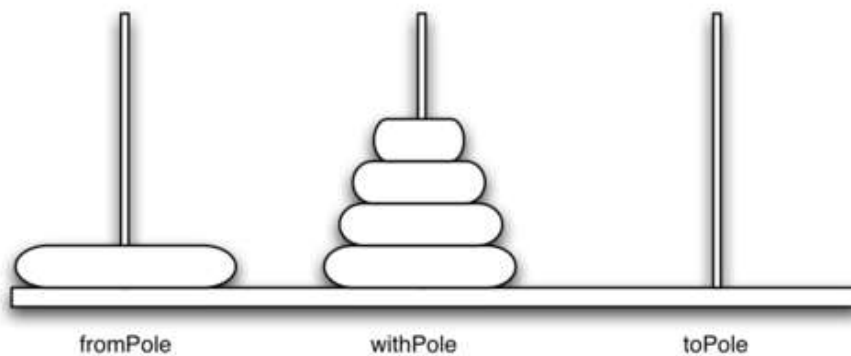
THEORY

The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower,[1] and sometimes pluralized) is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.



TASK

Using MIPS Assembly language implement the Tower of Hanoi Game in MARS.