Lab # 1

INTRODUCTION TO VVM

# OBJECTive

# 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" (2'nd), 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.

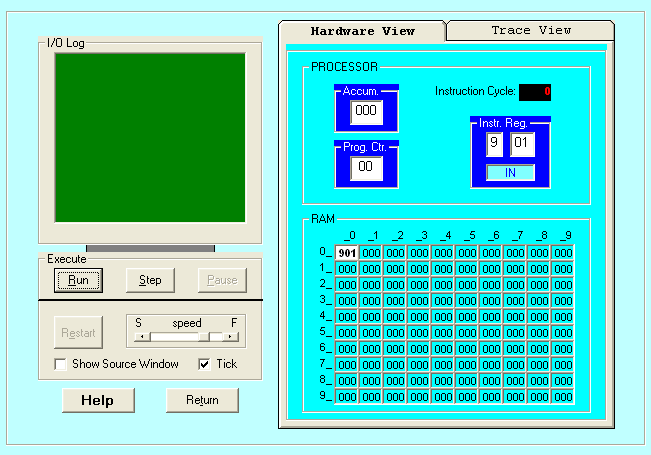
For simplicity, **VVM** works directly with decimal data and addresses rather than with binary values. Furthermore, the virtual machine works with only one form of data: decimal integers in the range ± 999. This design alleviates the need to interpret long binary strings or complex hexadecimal codes.

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

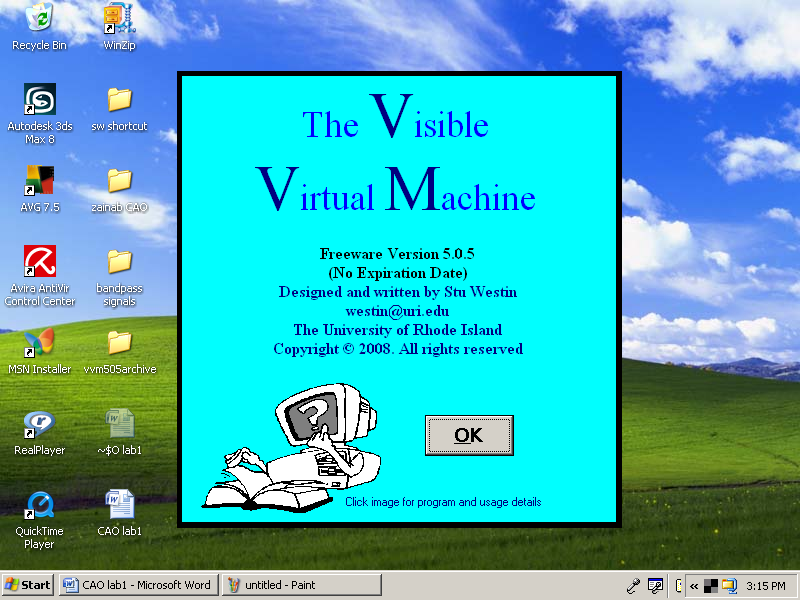
902 Output result

000 Halt (program ends here)

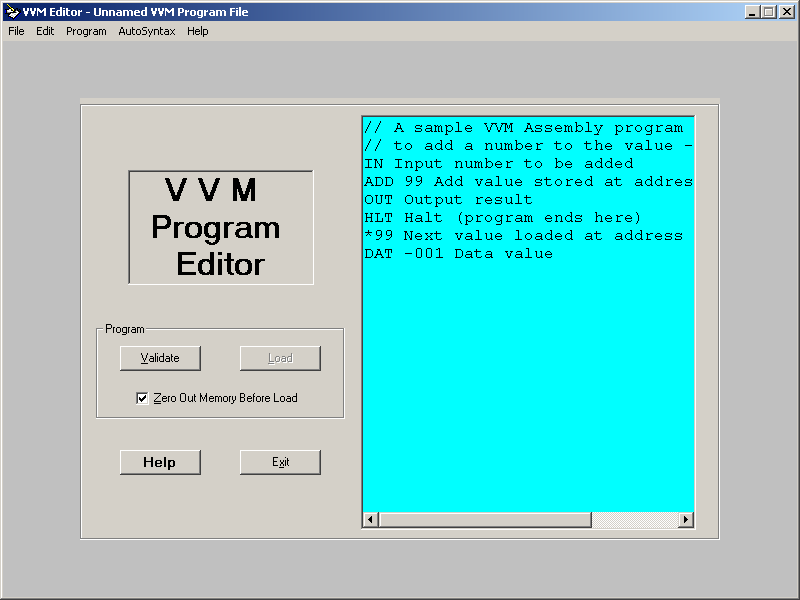
\*99 Next value loaded at address 99

-001 Data value

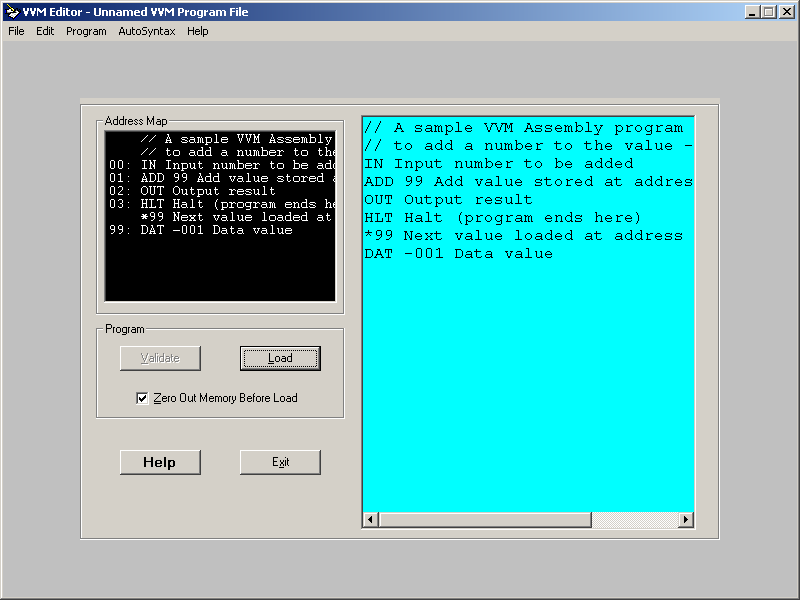
**STEP#1: Load VVM**

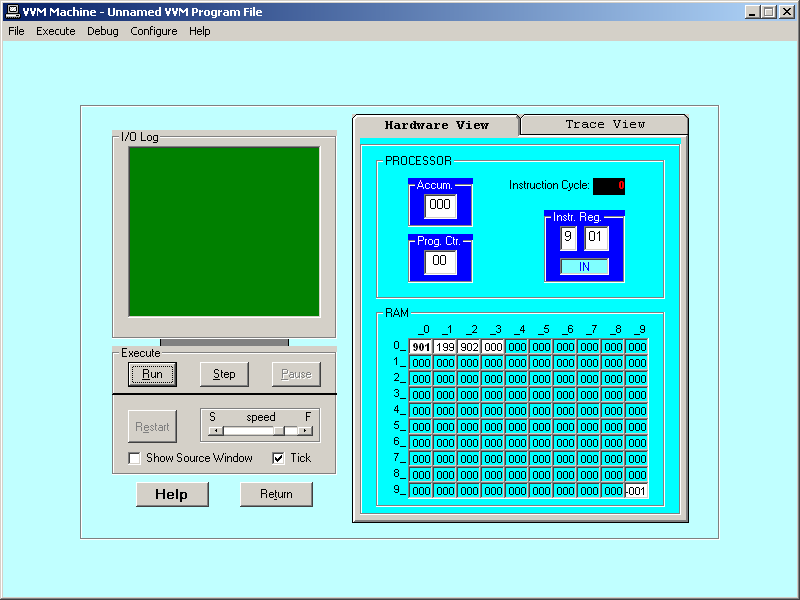
****

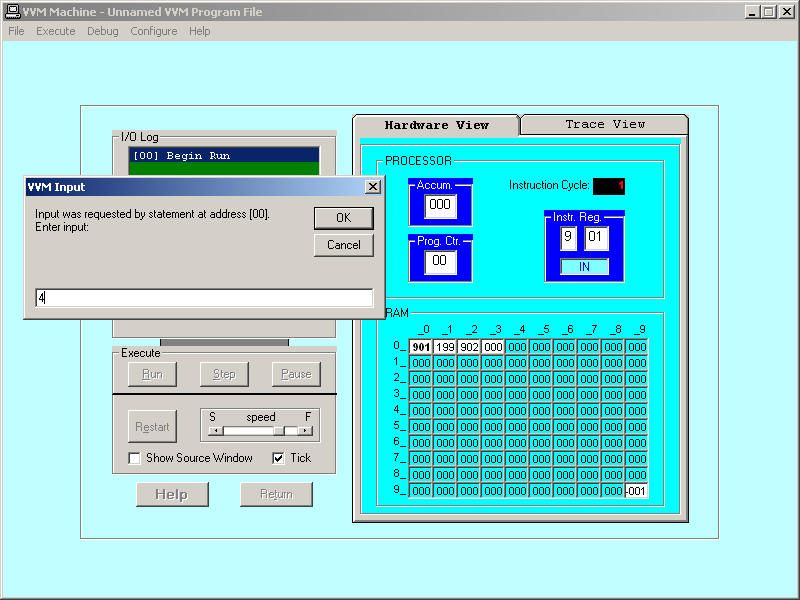
**STEP2: Copy the code and paste in the blue editor window.**

****

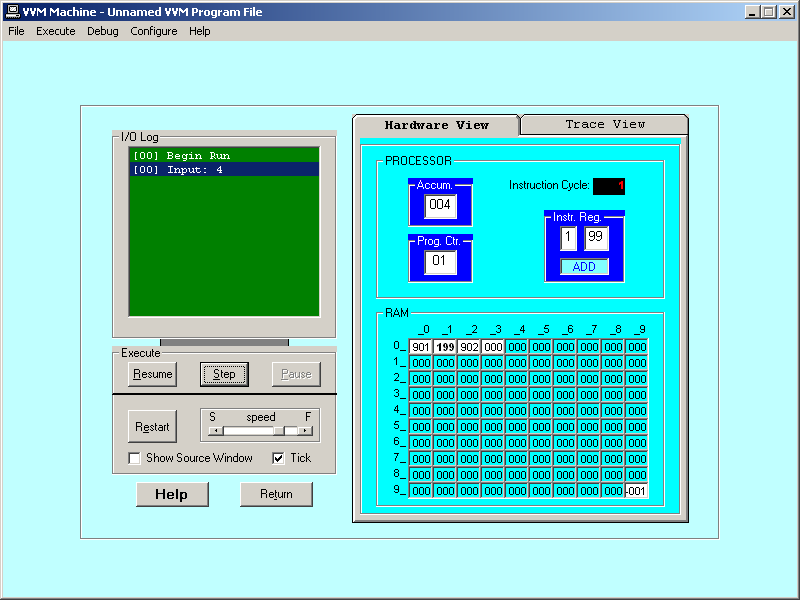
**STEP3: Press the Validate button.**

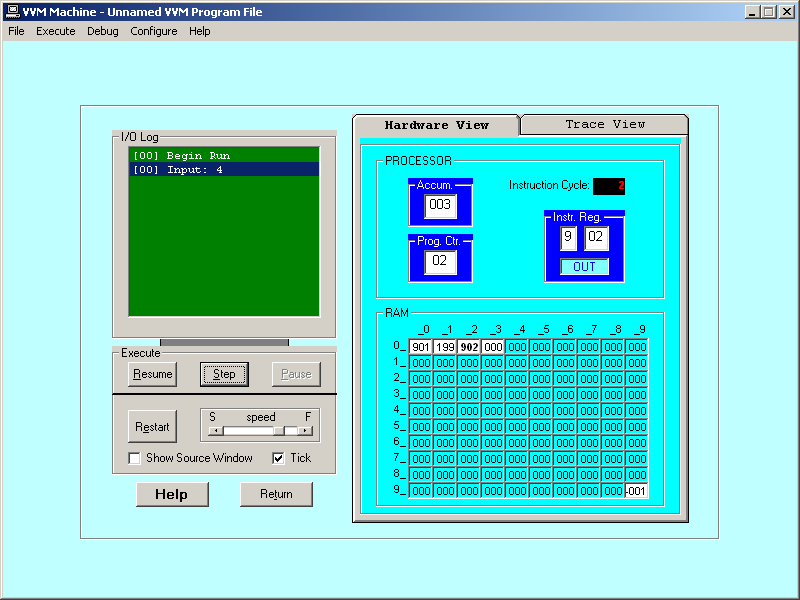
****

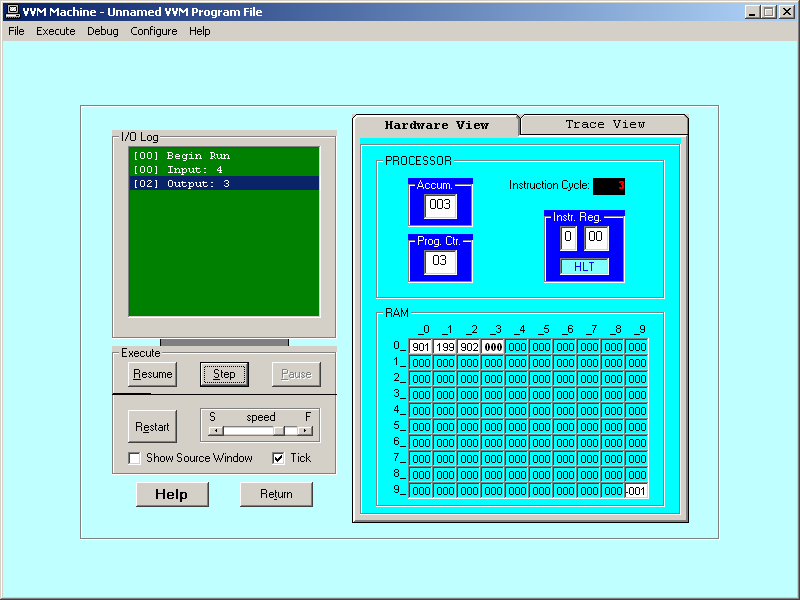
**STEP4: Press the Load button**

****

**STEP#5: Press the Step button**

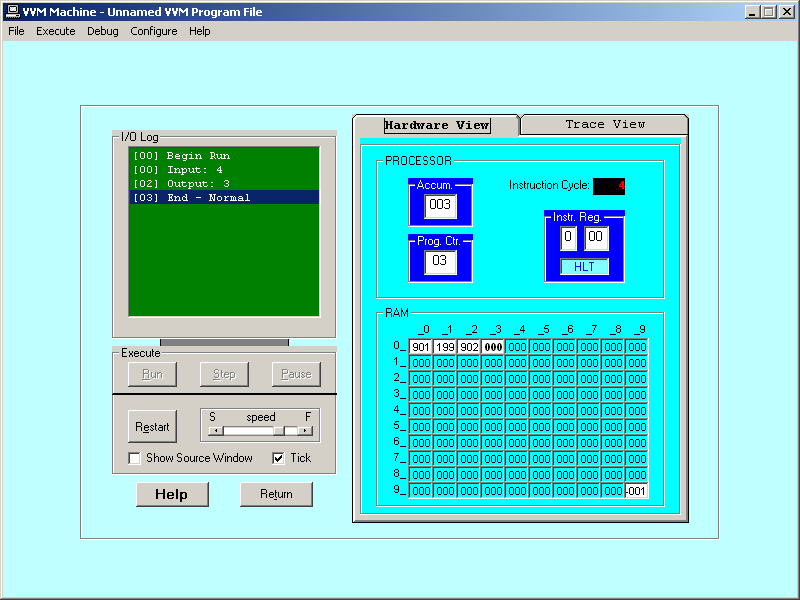
****

****

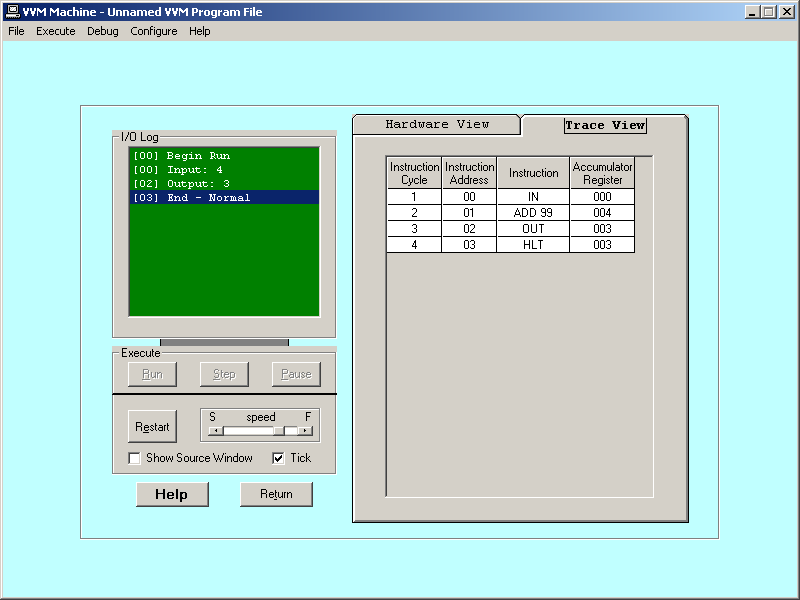
****

**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 (5*nn*) [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 (3*nn*) [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 (1*nn*) [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 (2*nn*) [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 (0*nn*) [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 (7*nn*) [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 <> 0), the Program Counter Register is incremented by one, and the next sequential instruction is executed.
* **Branch if Positive or Zero (8*nn*) [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 (6*nn*) [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 (4*nn*) [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.**

inInput A

sto 98 Store A

inInput B

sto 99 Store B

lda 98 Load value of A

sub 99 Subtract B from A

brp 11 If A >= B, branch to 11A 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)

**Equivalent BASIC**

**program:**

INPUT A

INPUT B

IF A >= B THEN

C = A + B

ELSE

C = A - B

ENDIF

PRINT C

END

add 99 Add B

sta 97 Store C

out [14] Print result

hltDone

**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

1. 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 (8*nn*) [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 (7*nn*) [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 <> 0), the Program Counter Register is incremented by one, and the next sequential instruction is executed.

### BR *nn*

**Branch (6*nn*) [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 # 3**

**Simple looping example.**

In Input A

sto 99 Store A

**Equivalent to the following BASIC program:**

**INPUT A**

**DO WHILE A > 0**

**PRINT A**

**INPUT A**

**LOOP**

**END**

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

**VVM Program example# 4**

**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

# Assignment

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

Lab # 4

INTRODUCTION TO MIPS Assembly Language

# OBJECTive

Introduction to MIPS Assembly language.

Simulating the given MIPS programusing MARS.

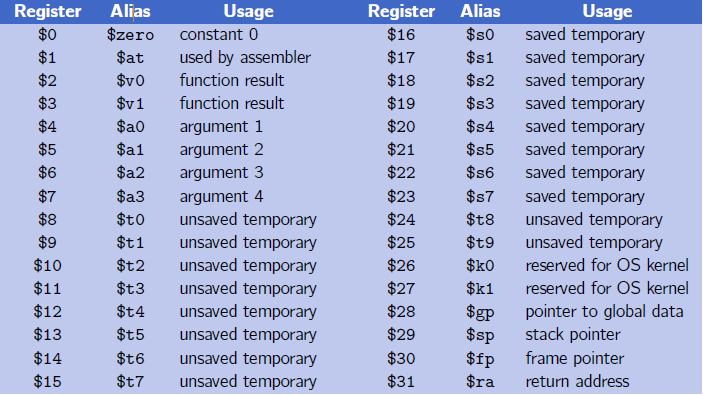
# THEORY

**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 theCPU 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 availablefor 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**

***# Title: Filename:***

***# Author: Date:***

***# Description:***

***# Input:***

**# Output:**

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

**.data**

**. . .**

**. . .**

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

**.text**

**.global main**

**main:**

**. . . # main program entry**

**. . .**

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

**syscall**

**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 mnemonicforaninstructionorpseudo-instructionoradirective;[*operands*]containsa combinationofone,two,orthreeconstants,memoryreferences,andregisterreferences, as required by the particular instruction or directive; [#*comment*] is an optional comment.

**Labels**

Labelsarenothingmore thannames used for referringtonumbers andcharacterstrings ormemorylocationswithinaprogram.Labels letyougivenamestomemoryvariables, values, and the locations of particular instructions.

Thelabel*main*isequivalentto 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. Directivesonlyprovidetheframeworkforanassemblerprogram,though;youalso need lines in your source code thatactually 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 variablesin 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

**.SPACEn** Directive

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

**Pseudo-instructions**

Pseudo-instructionsgiveMIPSarichersetofassemblylanguageinstructionsthan thoseimplementedbythehardware.Forexample, one of the frequent steps needed in programmingistocopythevalueofoneregisterintoanotherregister.Thisactuallycan be solved easily by the instruction:

add $t0, $zero, $t1

However, it is more natural to use the pseudo-instruction

move $t0, $t1.

Theassemblerconvertsthispseudo-instructionintothemachinelanguageequivalent of the prior instruction.

**MIPSINSTRUCTIONS**

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

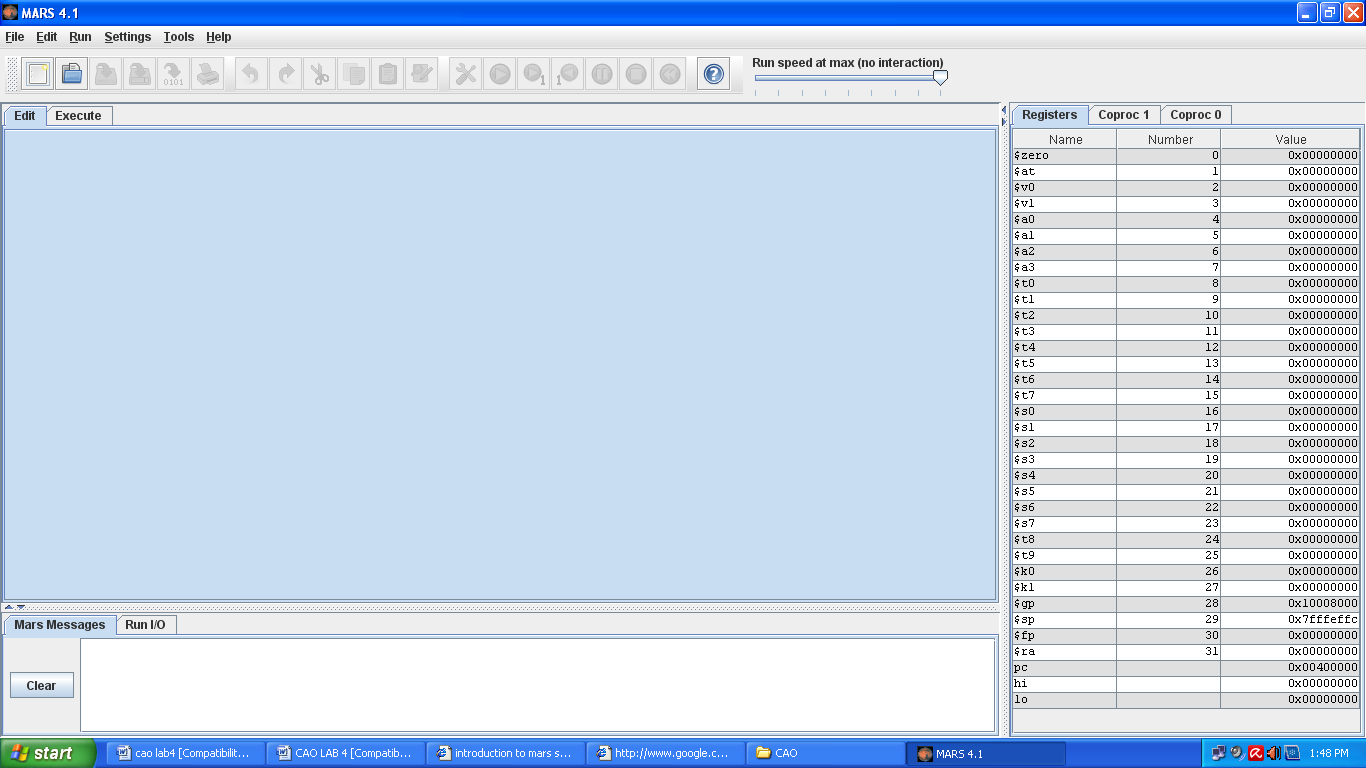
**SYSTEMI/O(INPUT/OUTPUT)**

* 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 are10 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=addressofinputbuffer  inmemory.  $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 Coprocesor 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**

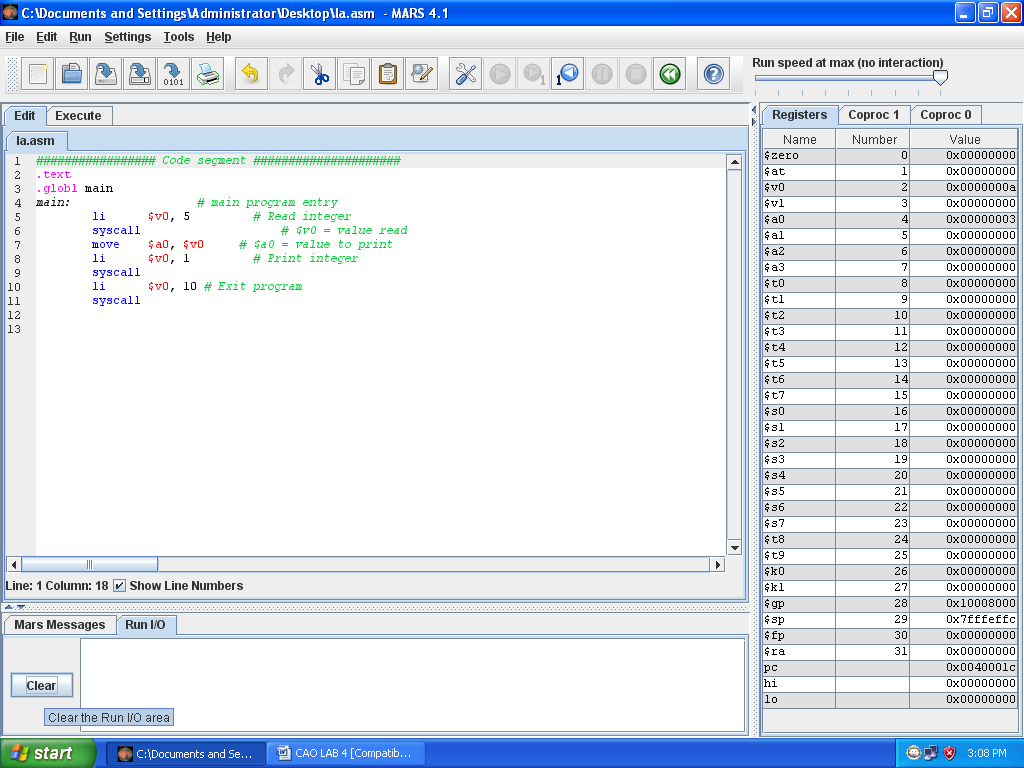
**syscall**

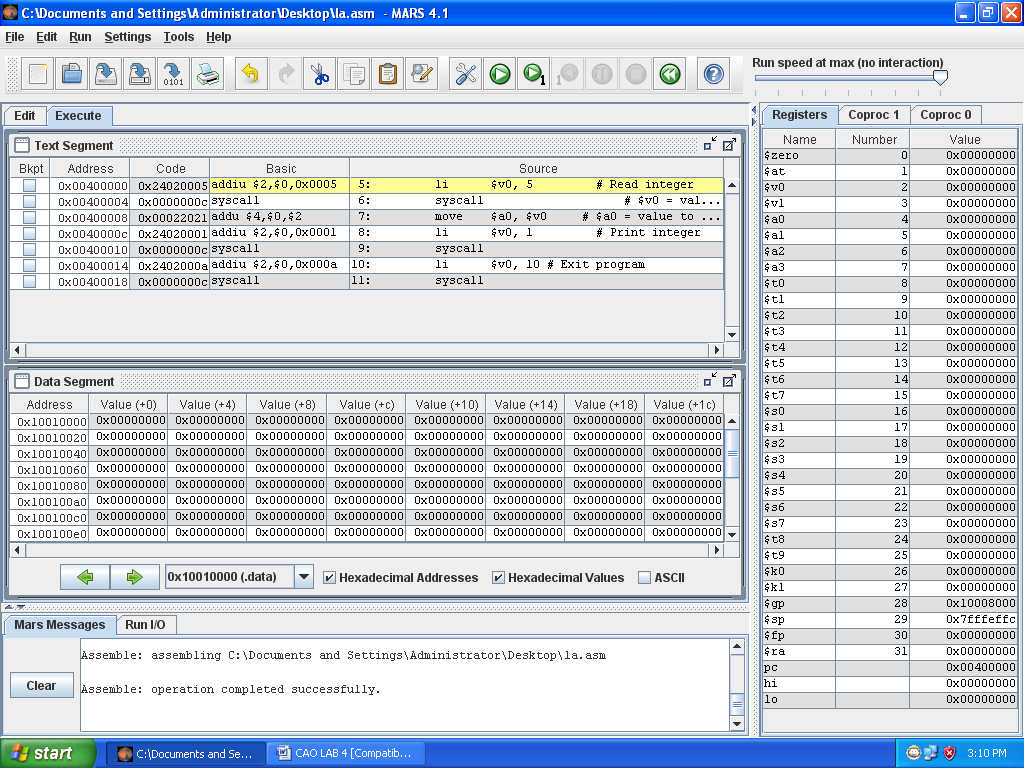
**li $v0, 10 # Exit program**

**syscall**

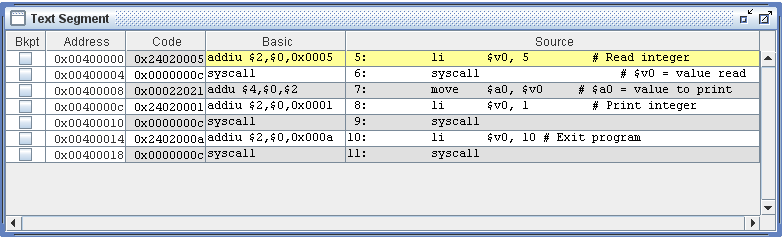
**STEP#1**

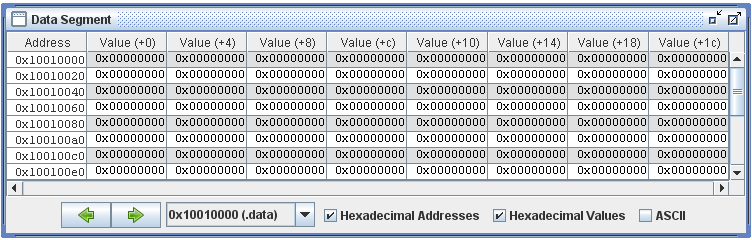
Load mars simulator, copy this code to the editor and save file with .asm extension.



**STEP# 2**

Assemble program by pressing F3.

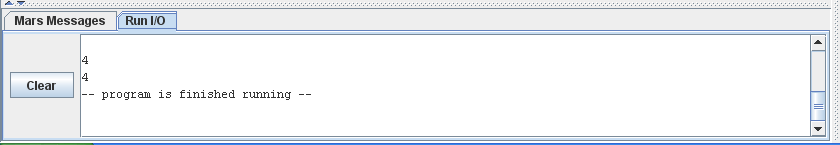




**STEP# 3**

Execute program by pressing F5.

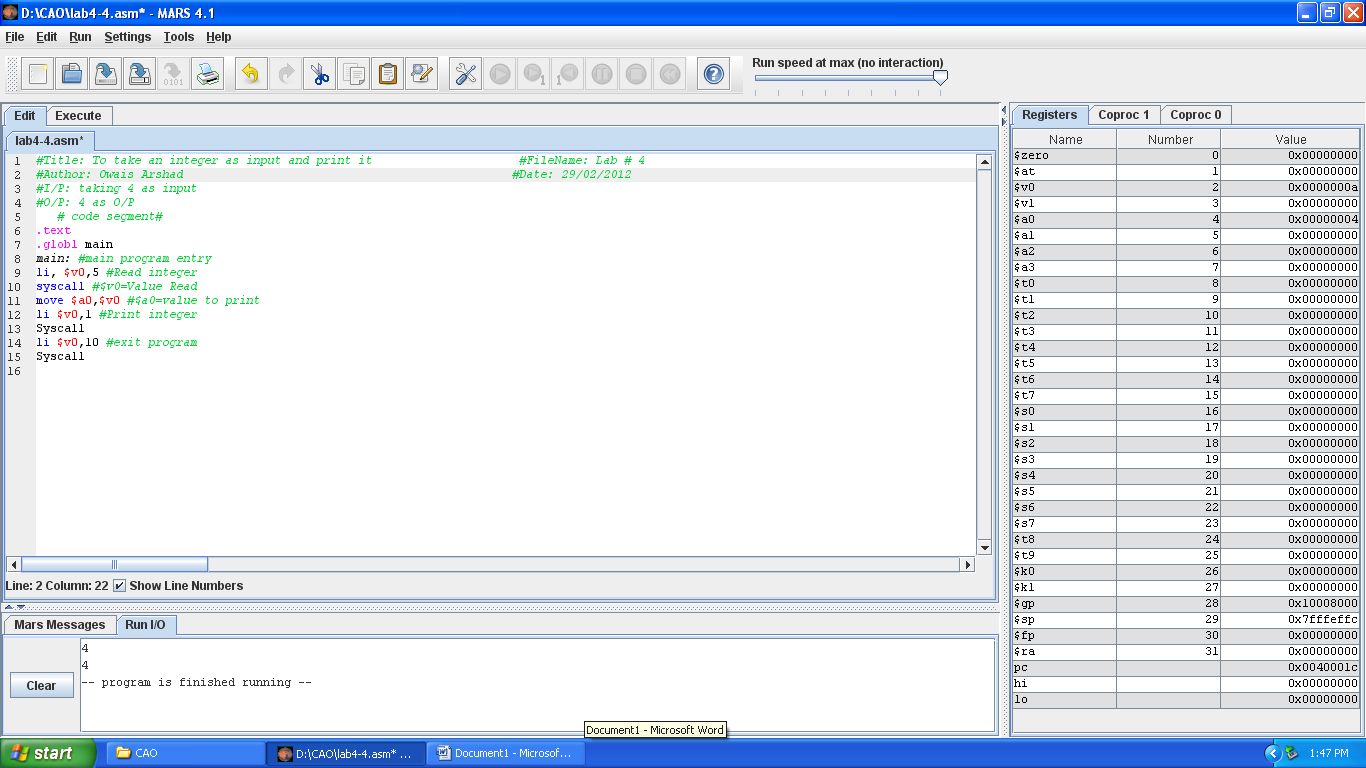
Type any integer number for input.



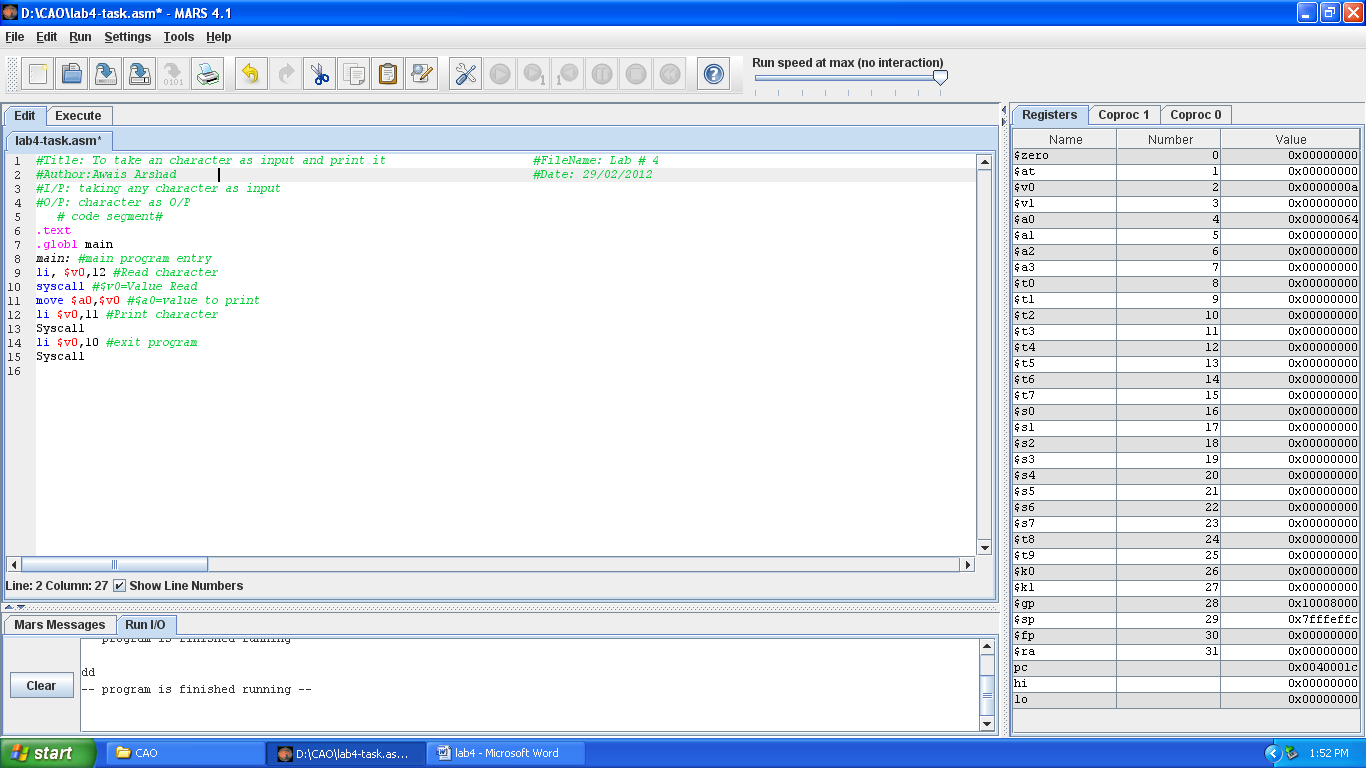
**LAB TASK**

1. Write an assembly program that Read and Print character.

**TASK 1**



**TASK 2**

****

Lab # 5

### LOAD & CONSTANT MANIPULATING

### INSTRUCTIONS IN MIPS

# OBJECTive

Take three integer input and display Sum of Three Integers using “addu” instruction.

# Theory

la Rdest, var

**Load Address**. Loads the address of var into Rdest.

li Rdest, imm

**Load Immediate**. Loadsthe immediate value imm intoRdest.

addu Rdest,Rsrc1,Rsrc2

AddsthecontentsofRsrc1andRsrc2andstorestheresultinRdest.Thenumbersaretreatedasunsignedintegers. No overflow exception is needed.

# Program #2

**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**

Write the same program with small variationi.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 integer for 1st addition

2

2

2

Enter 3 integer for 2nd addition

3

3

3

The sum of 1st addition is 6

The sum of 2nd addition is 9

Lab # 6

aRITHMETIC INSTRUCTIONS IN MIPS

# OBJECTive

Study Addition, Multiplication, Division Instructions.

# Theory

**addu** Rdest,Rsrc1,Rsrc2

AddsthecontentsofRsrc1andRsrc2andstorestheresultinRdest.Thenumbersaretreatedasunsignedintegers. No overflow exception is needed.

**addi** Rdest,Rsrc1, imm

Rdest=Rsrc1+16-bitsignedimmediate.Incaseofanoverflow, an overflow exception is generated.

**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

**Example:**

**Multiply two numbers and display the result.**

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

.DATA

Prompt : .asciiz"resuit 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:**

# resuit 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.**

**Example:**

**Divide two numbers and display the result.**

**################# 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 --

**CONDITIONALANDUNCONDITIONALBRANCHINSTRUCTIONS**

|  |  |
| --- | --- |
| Instructions | Description |
| **Branch if greater than or equal to zero**  **bgez rs, L** | **if** ( rs ≥ 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 ≤ 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 != 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*signed* integers. |
| **Set less than unsigned**  **sltu rd, rs, rt** | Same as **slt**except rs and rt are *unsigned* integers. |
| **Set less than immediate**  **slti rt, rs,immediate** | **if** ( rs<*signed* immediate ) rd=1; **else** rd=0; |
| **Set less than immediate unsigned**  **sltiu rt, rs,immediate** | **if** ( rs<*unsigned*immediate ) rd=1; **else** rd=0; |

**LAB TASK**

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

**Lab # 7**

**CONTROL INSTRUCTIONS IN MIPS**

**OBJECTive**

Write a MIPS assembly language program that calculates the sum of all positive integers less than or equal to N and displays the result using ‘blez’ and ‘bnez’ instructions.

**Theory**

**blez**Rs, Label

Branch if Less Than or Equal to Zero.

**if**( Rs ≤ 0 ) go to Label.

**bnezRs, Label**

Branch if Not Equal to Zero.

**if**(Rs != 0) go to Label

**Program #1**

Write a MIPS assembly language programthat calculates the sumof all positive integerslessthanorequaltoNanddisplaystheresult.

################### 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 msg. into $a0

syscall # print the string

li $v0, 10 # terminate program run and

syscall # return control to system

**LAB TASK**

Write a MIPS assembly language programthat calculates the sumof all positive even integerslessthanorequaltoNanddisplaystheresult.

Lab # 8

BIT MANIPULATION INSTRUCTIONS IN MIPS

# OBJECTive

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

# Theory

**BITWISE LOGICAL INSTRUCTIONS**

|  |  |
| --- | --- |
| Instructions | Description |
| and rd, rs, rt | rd = rs & rt |
| andi rt, rs, immediate | rt = rs & immediate |
| or rd, rs, rt | rd = rs | rt |
| ori rt, rs, immediate | rd = rs | immediate |

The main usage of bitwise logical instructions are: ***to set, to clear*** 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 |
| Z AND 0 = 0 | Z OR 0 = Z |
| Z AND 1 = Z | Z OR 1 = 1 |

**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

# Program #1

Write a MIPS assembly program that take input value and perform bitwise AND instruction with mask 1.

**############# DATA SEGMENT ##################**

.data

input: .asciiz "\n enter an integer value: " **# VARIABLE DECLARATION**

result: .asciiz "\n result is: "

**############# TEXT 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

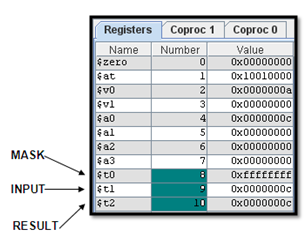
move $a0,$t2 **# MOVE AND INSTRUCTION RESULT IN $a0**

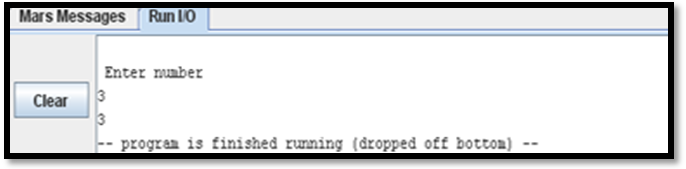
li $v0,1 **# PRINT VALUE OF $t2**

syscall

li $v0,10 **# EXIT PROGRAM**

syscall

**OUTPUT:**



12

12

# Program #2

Write a MIPS assembly program that take input value and perform bitwise AND instruction with mask 0.

**############# DATA SEGMENT ##################**

.data

input: .asciiz "\n enter an integer value: " **# VARIABLE DECLARATION**

result: .asciiz "\n result is: "

**############# TEXT 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

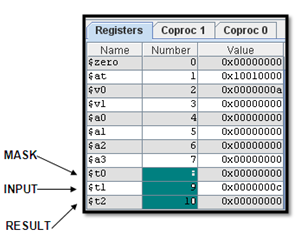
move $a0,$t2 **# MOVE AND INSTRUCTION RESULT IN $a0**

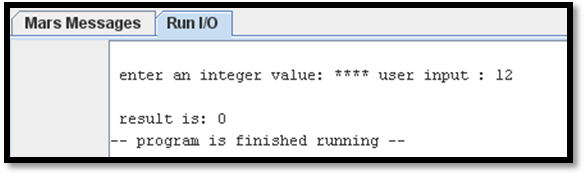
li $v0,1 **# PRINT VALUE OF $t2**

syscall

li $v0,10 **# EXIT PROGRAM**

syscall

**OUTPUT:**

****

# Program #3

Write a MIPS assembly program that take input value and perform bitwise OR instruction with mask 1.

**############# DATA SEGMENT ##################**

.data

input: .asciiz "\n enter an integer value: " **# VARIABLE DECLARATION**

result: .asciiz "\n result is: "

**############# TEXT 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

OR $t2,$t1,$t0 **# OR INSTRUCTION, $t2 = $t1 OR $t0**

la $a0,result **# PRINT RESULT MESSAGE**

li $v0,4

syscall

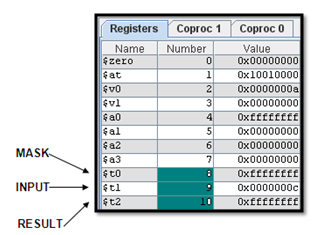
move $a0,$t2 **# MOVE AND INSTRUCTION RESULT IN $a0**

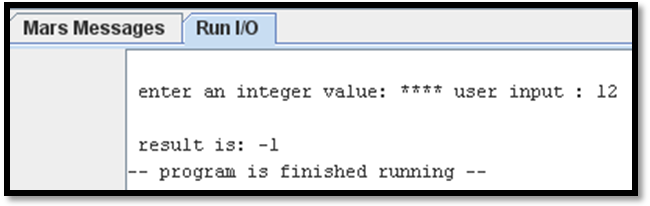
li $v0,1 **# PRINT VALUE OF $t2**

syscall

li $v0,10 **# EXIT PROGRAM**

syscall

**OUTPUT:**

****

# Program #4

Write a MIPS assembly program that take input value and perform bitwise OR instruction with mask 0.

**############# DATA SEGMENT ##################**

.data

input: .asciiz "\n enter an integer value: " **# VARIABLE DECLARATION**

result: .asciiz "\n result is: "

**############# TEXT 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

OR $t2,$t1,$t0 **# OR INSTRUCTION, $t2 = $t1 OR $t0**

la $a0,result **# PRINT RESULT MESSAGE**

li $v0,4

syscall

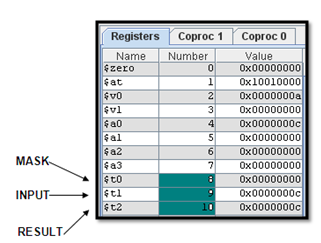
move $a0,$t2 **# MOVE AND INSTRUCTION RESULT IN $a0**

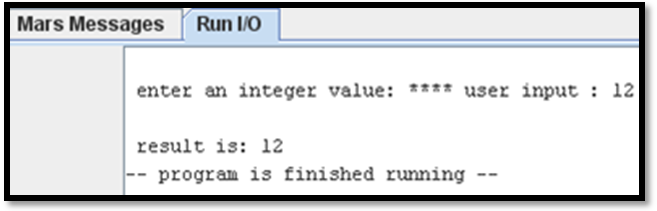
li $v0,1 **# PRINT VALUE OF $t2**

syscall

li $v0,10 **# EXIT PROGRAM**

syscall

**OUTPUT:**



**LAB TASK**

**LAB TASK:**

**Task#1:**

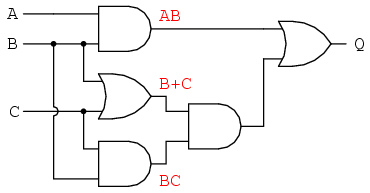
Clear the sign bit of $t0 while leaving the other bits unchanged?

**Task#2:**

Set the most significant and least significant bits of $t0 while preserving the other bits?

**Task#3:**

Implement following circuit into MIPS Assembly Language using AND,OR instructions.

****

Lab # 9

BIT MANIPULATION INSTRUCTIONS IN MIPS

# OBJECTive

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

# Theory

**BITWISELOGICALINSTRUCTIONS**

|  |  |
| --- | --- |
| Instructions | Description |
| and rd, rs, rt | rd = rs&rt |
| andi rt, rs, immediate | rt = rs& immediate |
| or rd, rs, rt | rd = rs | rt |
| ori rt, rs, immediate | rd = rs | 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*** someselectedbitsinthe destinationoperand.Todothis,asourcebitpatternknownasa 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 AND 0 = 0 | Z OR 0 = Z | Z XOR 0 = Z |
| Z AND 1 = Z | Z OR 1 = 1 | Z XOR 1 = ~Z |

**XOR instruction:**

The XOR instruction can be used to INVERT specificdestinationbits whilepreserving the others. A one mask bit inverts the corresponding destination bit; a zero mask bit preserves the corresponding destination bit.

1. **Example 1:**

Change the sign bit of $t0?

**Solution:**

Use the XOR instruction with a mask of 80000000h.

xori $t1,$t0,0x80000000

# Program #1

Write a MIPS assembly program that take input value and perform bitwise XOR instruction with mask 1.

**############# DATA SEGMENT ##################**

.data

input: .asciiz "\n enter an integer value: " **# VARIABLE DECLARATION**

result: .asciiz "\n result is: "

**############# TEXT 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

XOR $t2,$t1,$t0 **# XOR INSTRUCTION, $t2 = $t1XOR $t0**

la $a0,result **# PRINT RESULT MESSAGE**

li $v0,4

syscall

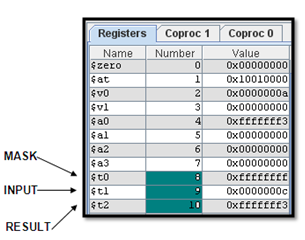
move $a0,$t2**# MOVE AND INSTRUCTION RESULT IN $a0**

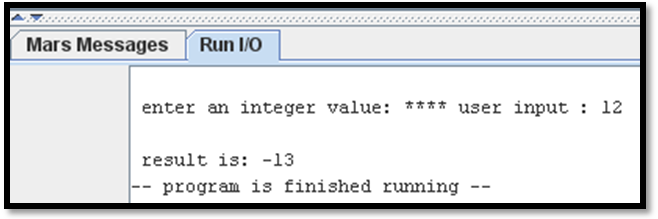
li $v0,1 **# PRINT VALUE OF $t2**

syscall

li $v0,10 **# EXIT PROGRAM**

syscall

**OUTPUT:**

****

# Program #2

Write a MIPS assembly program that take input value and perform bitwise XOR instruction with mask 0.

**############# DATA SEGMENT ##################**

.data

input: .asciiz "\n enter an integer value: " **# VARIABLE DECLARATION**

result: .asciiz "\n result is: "

**############# TEXT 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

XOR $t2,$t1,$t0 **# XOR INSTRUCTION, $t2 = $t1XOR $t0**

la $a0,result **# PRINT RESULT MESSAGE**

li $v0,4

syscall

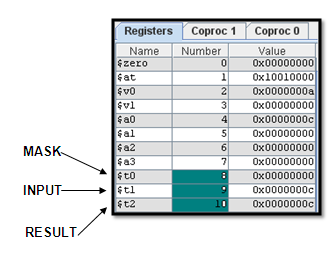
move $a0,$t1 **# MOVE AND INSTRUCTION RESULT IN $a0**

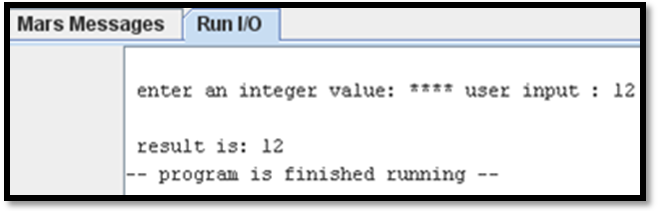
li $v0,1 **# PRINT VALUE OF $t2**

syscall

li $v0,10 **# EXIT PROGRAM**

syscall

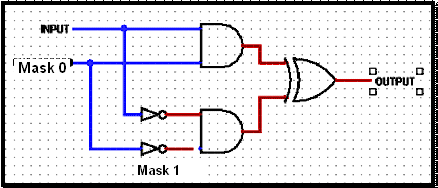
**OUTPUT:**

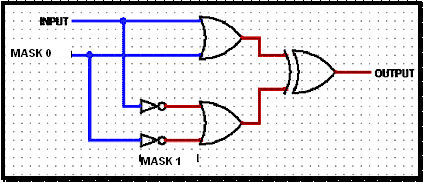


**LAB TASK:**

Implement following circuit into MIPS Assembly Language using AND,OR& XOR instructions..

**Diagram # 1**

****



Lab # 10

IF THEN ELSE,Control Structure in MIPS

# OBJECTive

# Study how to implement translation of“if then else” control structures in MIPS assembly langage.

# 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 pseudocode.

**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

resultscould 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 belowhelp to make the connection back to the original pseudocode.

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 b ……………….immediate

**SHIFT RIGHT LOGICAL**

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

**PROGRAM#1:**

Write a program in MIPS assembly language that translateIF-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: "

**################## TEXT SEGMENT ########################**

.text

.globl main

main:

la $a0,input **# print input message**

li $v0,4

syscall

la $v0,5 **# read integer**

syscall

**IF ($v<0) THEN**

**$t2=$v0>>2**

**Print: display $t2**

**ELSE**

**$t2=$v0<<2**

**Print: display $t2**

bgez $v0,else **# if-else equilavent statement**

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

la $a0,rshift **# print value of rshift**

li $v0,4

syscall

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

li $v0,1

syscall

b end **#branch to statement at end unconditionally**

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

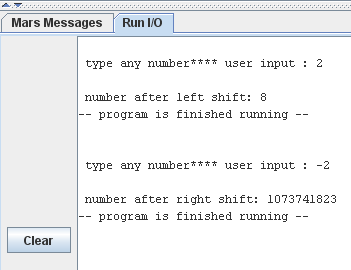
syscall

end:

li $v0,10 **# terminate program**

syscall

**OUTPUT:**



****

**LAB TASK**

Write a program in MIPS assembly language that take input from user and print whelther 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 structures in MIPS assembly langage.

# 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 pseudocode.

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” pseudocode 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:**

Write a program in MIPS assembly language that translateFOR LOOP control structure.

**################## DATA SEGMENT ###################**

.data

counter: .asciiz"\n value of count, $t0: "

total: .asciiz"value of sum, $a0: "

tab: .asciiz"\t"

**################## TEXT SEGMENT ###################**

.text

.globl main

main:

li $a2,0 **# $a2=0**

li $t0,10 **# initiallize 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 tab**

li $v0,4

syscall

la $a0,total **# print message of total**

li $v0,4

syscall

move $a0,$a2**# print value of $a2**

li $v0,1

syscall

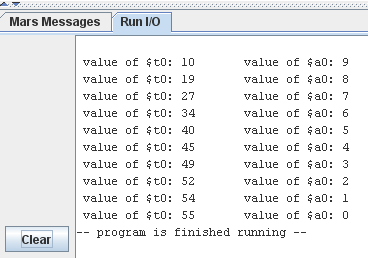
bgtz $t0,loop**# if($t0>0) branch to loop**

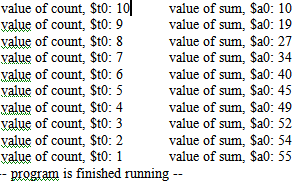
**end:**

li $v0,10

syscall

**OUTPUT:**

****



**LAB TASK**

Write a program in MIPS assembly language that take input and display whether number is prime or not.

Lab # 12

MIPS Assembly Language

# OBJECTive

To introduce how to implement array as an abstract data structure in MIPS assembly language.

# Theory

Not like high level languages, Assembly language has no notion of an array at all. Arrays like variables are treated as a block of memory that could be allocated with a single directive, where the first element is given a label.

The array as the most important and most general data structure has the following properties:

1. All elements must be the same size. The array is a homogeneous data structure.
2. The size of an array is fixed. The number of elements is fixed.
3. A label (address) is tied to the first element of the array.
4. Traversing each element of an array needs an index or indices and the label as the array's name.

**ARRAY DECLARATION:**

With reference to the above properties, in assembly language to declare an array it requires:

1. A label name,
2. The number of elements,
3. The size of each element,
4. The initial value of each element.

**EXAMPLE:**

.data

A01: **.**byte 'a', 'k', 'p', 5 # A01 is an array of 4 bytes: {'a', 'k', 'p', 5}

A02: **.**word 5, 6, -9, 7 # A02 is an array of 4 words: {5, 6, -9, 7}

B02: **.**space 40 #allocate 40 consecutive bytes, with storage uninitialized

# could be used as a 40-element character array, or a

# 10-element integer array;

# a comment should indicate which!

var1: **.**half 3 # create a single short integer variable with initial value 3

B03: **.**word -1**:**30 # allocate 40 consecutive words with each element

# initialized with -1.

**TRAVERSING SINGLE-DIMENSIONAL ARRAY:**

To access every element of an array, we have to know the address of that element. Because all elements have the same size, the address of an element of the array can be formulated as:

The address of ith-element (in byte) = starting address + ***size-of-element*** \* i

1. The first element of the array is indexed 0.
2. The ***size-of-element*** is the number of bytes in a single array element.
3. The ***size-of-element*** either is one byte, 2 bytes, 4 bytes, or 8 bytes.

**EXAMPLE:**

The following code fragment is to access the sixth element of table1:

.data

table1: .word 4, 5, 6, 7, 8, 9, 10, 21

.text

la $t0, table1

lw $t1, 20($t0)

addiu $t2, $t0, 20

lw $t1, 0($t2)

Lab # 13

MIPS Assembly Language

# OBJECTive

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

# 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**

Write a program in MIPS assembly language that translate SWITCH control structure.

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SWITCH TRANSLATION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#####################DATA SEGMENT #########################

.data

.align 2

varword: .word main,case1,case2,case3

input: .asciiz "\nType a va;ue from 1 to 3 "

msg\_1: .asciiz "\n you are in case1"

msg\_2: .asciiz "\n you are in case2"

msg\_3: .asciiz "\n you are in case3"

################ ##TEXT SEGMENT ######################

.text

.globl main

main:

li $v0,4 # print input message

la $a0,input

syscall

li $v0,5 # read an 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

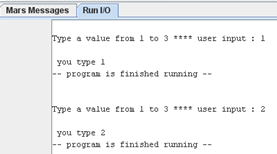
la $a0,msg\_3

syscall

end:

li $v0,10

syscall

**OUTPUT:**

Lab # 14

pROCEDURES IN MIPS

# OBJECTive

# Study how to implement “Procedures” in MIPS assembly langage.

# Theory

**INTRODUCTION**

So far, we've only looked at programs consisting of a single long chunk of code. Each program has started at the top of the code, executed each instruction in turn (with an occasional detour for looping or decision-making), and then ended at the bottom of the code. That's fine for small programs, but larger programs require a programming construct known as a subroutine/procedure.

You've probably familiar with subroutines from a high-level language. In C, subroutines are known as functions, and in Pascal and Basic, they're known as procedures and functions. Subroutines, procedures, and functions all amount to the same thing – a separate section of code that optionally accepts well-defined inputs, promptly performs a certain action, and optionally returns a specific result value.

Subroutines let you build programs in a modular fashion, with the subroutines hiding the details of specific tasks so you can focus on the overall flow of the program. Subroutines can also make programs far more compact, since a single subroutine can be called from many places in a program, and can even perform different functions when passed different values. In a large program (whether written in assembler, C, Pascal, or some other language), subroutines are essential to creating orderly, maintainable code.

**HOW PROCEDURES WORK**

There are six steps that need to be accomplished in order to call and return from a procedure.

1. Place parameters in a place where the procedure can access them.

2. Transfer control to the procedure

3. Acquire the storage resources needed for the procedure.

4. Execute the procedure

5. Place the result value in a place where the calling program can access it.

6. Return control to the point of origin.

MIPS software follows the following convention in allocating its 32 registers for procedure calling:

 **$a0 – $a3**: four argument registers in which to pass parameters

 **$v0 – $v1**: two value registers in which to return values

 **$ra**: one return address register to return to the point of origin

The code that calls the procedure executes a **jal** instruction, which saves the address of the following instruction (PC+4) in register **$ra** and then loads PC with the address of the desired subroutine, thereby branching to the subroutine. The subroutine then executes just as any other code would. Procedures can – and often do – contain calls to other procedures; in fact, properly designed subroutines can even call themselves, a practice known as recursion.

When the subroutine has finished its task, it executes a **jr $ra** instruction, which jumps to the address stored in register $ra. This causes execution of the calling routine to resume at the instruction following the **jal X** instruction.

However, since the procedure may utilize any registers needed by the caller, those registers must be preserved before the procedure called and then be restored back after the procedure completed the tasks.The ideal data structure for spilling registers is a *stack*. MIPS software allocates **$sp** to track as the top of stack (TOS). The stack grows from higher address to lower address.

**STACK MANIPULATION**

The MIPS architecture does not explicitly support stack operations. In MIPS, we have to manipulate the stack pointer register to implement the stack.

**A. PUSH operation**

We have to decrement **$sp** to make room for the value being pushed onto the stack. For example, if we want to push the contents of **$a0**, we have to reserve four bytes of

stack space and use the **sw** instruction to push the value as shown below:

|  |  |  |
| --- | --- | --- |
| addiu | $sp,$sp,-4 | # reserve 4 bytes of stack |
| sw | $a0,0($sp) | # save the register |

**B. POP operation**

The operation can be implemented by using the load and add instructions. For example, to restore the value of $a0 from the stack, we use the lw instruction to copy the value and

increment $sp by 4 as shown below:

|  |  |  |
| --- | --- | --- |
| lw | $a0,0($sp) | # restore the two registers |
| addiu | $sp,$sp,-4 | # clear 4 bytes of stack |

**HOW TO PRESERVE REGISTERS**

To preserve registers efficiently, MIPS software separates 18 of the registers into two groups:

 **$t0 – $t9**: 10 temporary registers that are NOT preserved by the called procedure on a procedure call. It is the caller's responsibility to preserve any of them.

 **$s0 – $s7**: 8 saved registers that must be preserved on a procedure call (if used, the called procedure saves and restores them).

**NESTED PROCEDURES**

Procedures that do not call others are called *leaf* procedures. Non-leaf procedures must push all necessary registers before calling other procedures.

# INSTRUCTIONS USED:

**jal label**

**(Jump and link)**

Copies the address of the next instruction into the register $ra(register 31) and then jumps to the address label.

**jr $ra**

Jump register unconditionally : Jump to statement whose address is in $ra

**lw $t2,0($t1)**

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

**j label**

Jump Unconditrionally: jump to statement at label unconditionally

**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

**PROGRAM#1**

Write a program in MIPS assembly language that display sum of positive and negative values using procedure.

**##################DATA SEGMENT ###################**

.data

array: .word -4, 5, 8, -1 # 4,8,12,16

msg1: .asciiz "\n The sum of positive values= "

msg2: .asciiz "\n The sum of negative values= "

**##################TEXT SEGMENT ###################**

.text

.globl main

main:

li $v0, 4 #Print msg1

la $a0, msg1

syscall

la $a0, array # Initialize address parameter

li $a1, 4 # Initialize length parameter

jal Sum # Call sum function

addu $a0, $v0, $0 # sum of positive returned in $v0

li $v0, 1 #Print integer

syscall

li $v0, 4 #Print msg2

la $a0, msg2

syscall

addu $a0, $v1, $0 #sum of negative returned in $v1

li $v0, 1

syscall

end:

li $v0, 10 #Terminate Program

syscall

Sum:

addu $v0, $0, $0

addu $v0, $0, $0

Loop:

blez $a1,Return # Branch if less than or equal to zero

addi $a1, $a1, -1

lw $t0, 0($a0) #Load Word

addi $a0, $a0, 4

blez $t0, negative

add $v0, $v0, $t0

j Loop

negative:

add $v1, $v1, $t0

j Loop

Return:

jr $ra

**OUTPUT:**

