

LAB MANUAL

Computer Architecture & Logic Design

Course Code: CEN-221

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



Table of Contents

LAB 01: INTRODUCTION TO VVM.....	1
LAB 02: INTRODUCTION TO VVM PROGRAMMING	10
LAB 03: CONDITIONAL STATEMENT IN VVM PROGRAMMING	14
LAB 04: INTRODUCTION TO MIPS ASSEMBLY LANGUAGE	16
LAB 05: ARITHMETIC INSTRUCTIONS IN MIPS.....	25
LAB 06: BIT MANIPULATION INSTRUCTIONS IN MIPS.....	33
LAB 07: IF THEN ELSE; CONTROL STRUCTURE IN MIPS.....	37
LAB 08: FOR LOOP; CONTROL STRUCTURE IN MIPS.....	40
LAB 09: INTRODUCTION TO BASIC INSTRUMENT AND LOGIC GATES.....	43
LAB 10: MULTISIM AND LOGIC IMPLEMENTATION.....	64
LAB 11: BOOLEAN ALGEBRA AND LOGIC SIMPLIFICATION.....	69
LAB 12: KARNAUGH MAP.....	76
LAB 13: COMBINATIONAL LOGIC USING MUX.....	78
LAB 14: EXPOSURE OF DIFFERENT TYPES OF FLIPS	84

LAB # 1

INTRODUCTION TO VVM Software

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.

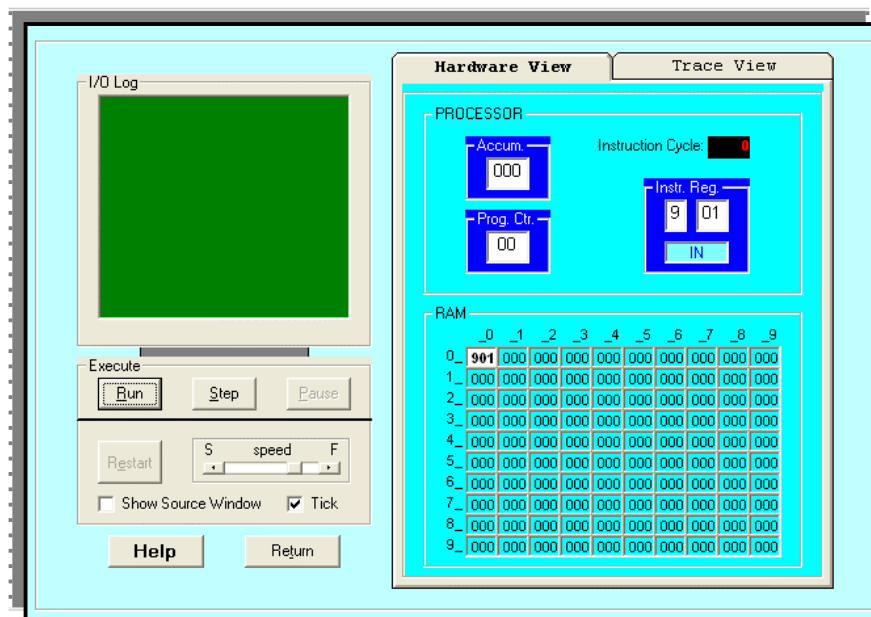
For simplicity, **VVM** works directly with decimal data and addresses rather than with binary values. Furthermore, the virtual machine works with only one form of data: decimal integers in the range ± 999 . This design alleviates the need to interpret long binary strings or complex hexadecimal codes.

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

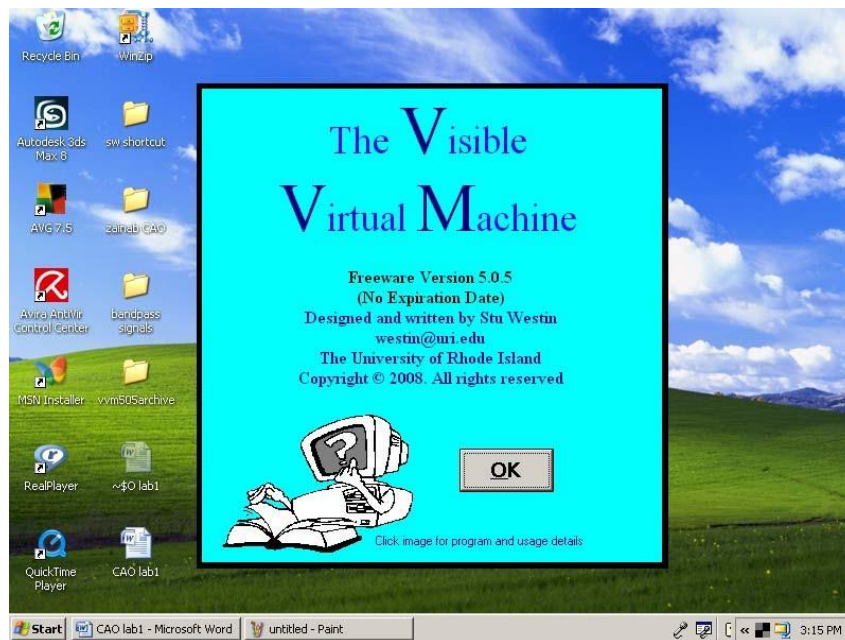
A simple VVM Assembly Language program which adds an input value to the constant value -1 is shown below (note that lines starting with “//” and characters to the right of program statements are considered comments, and are ignored by the VVM machine).

```
// A sample VVM Assembly program
// to add a number to the value -1. IN Input number
to beadedd
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
```

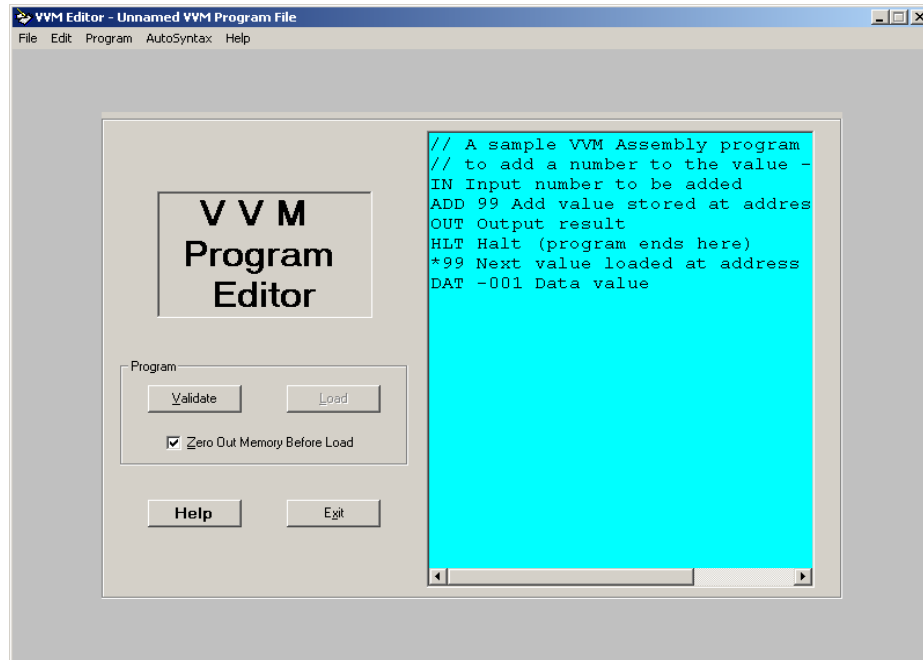
This same program could be written in VVM Machine Language format as follows:

```
// The Machine Language version 901 Input
number to beadedd
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
```

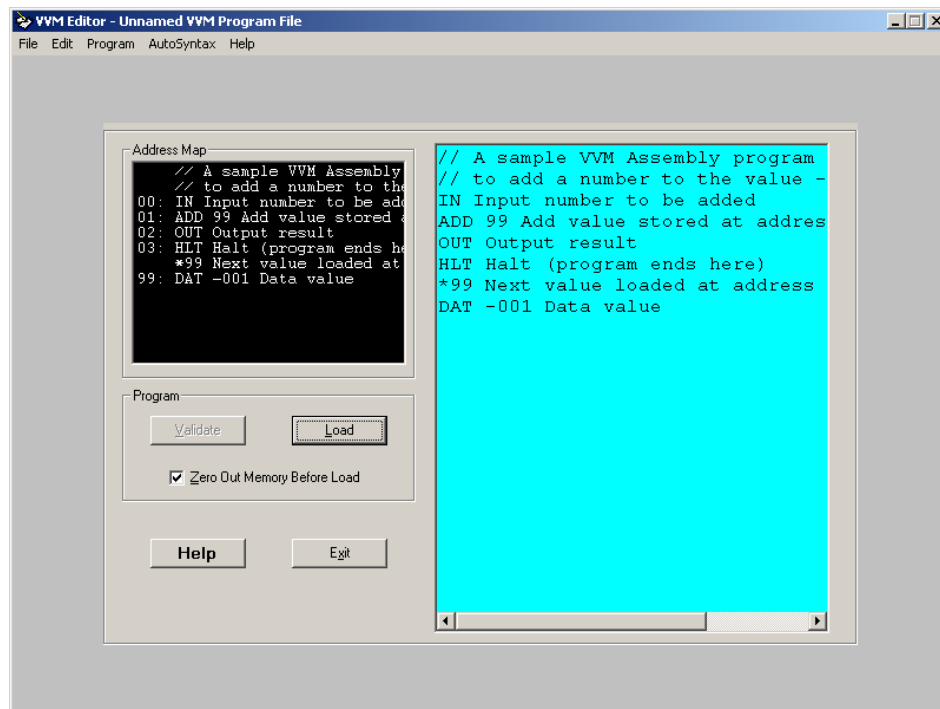
STEP1: Load VVM



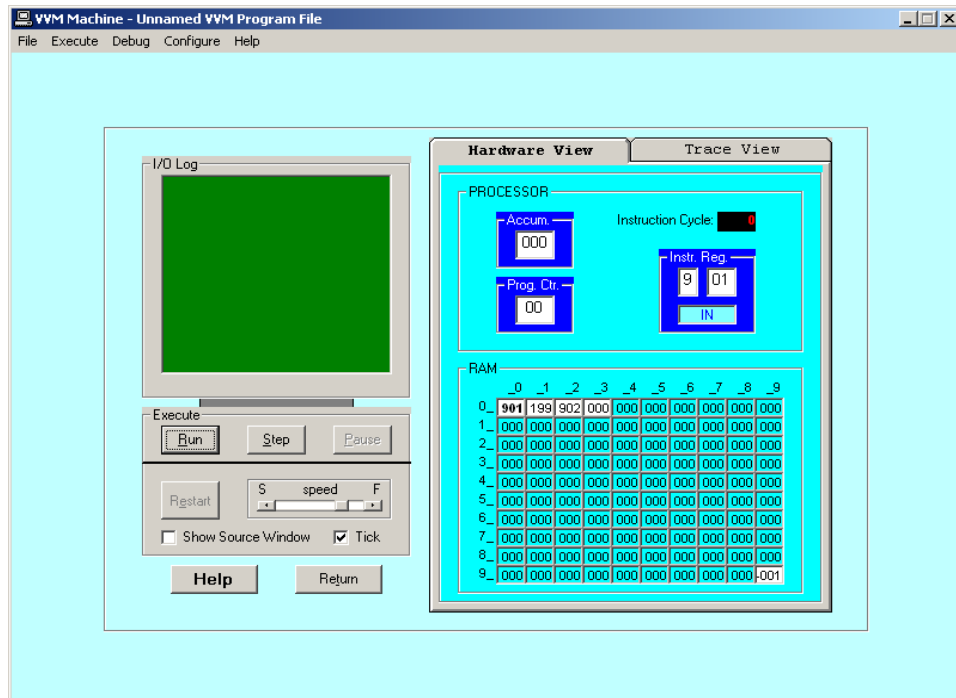
STEP2: Copy the code and paste in the blue editor window



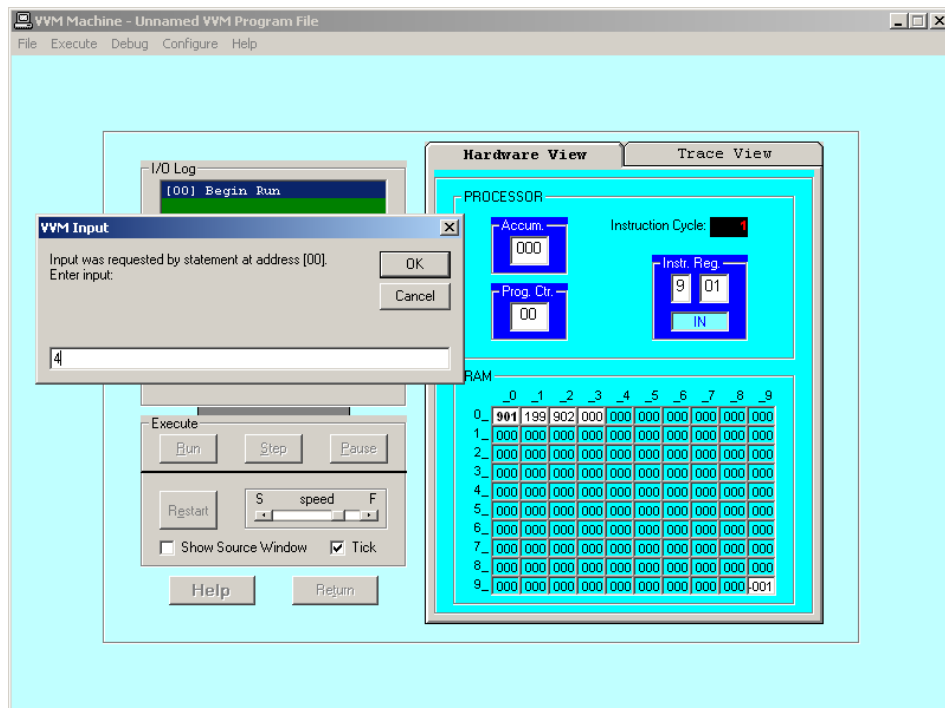
STEP3: Press the Validate button

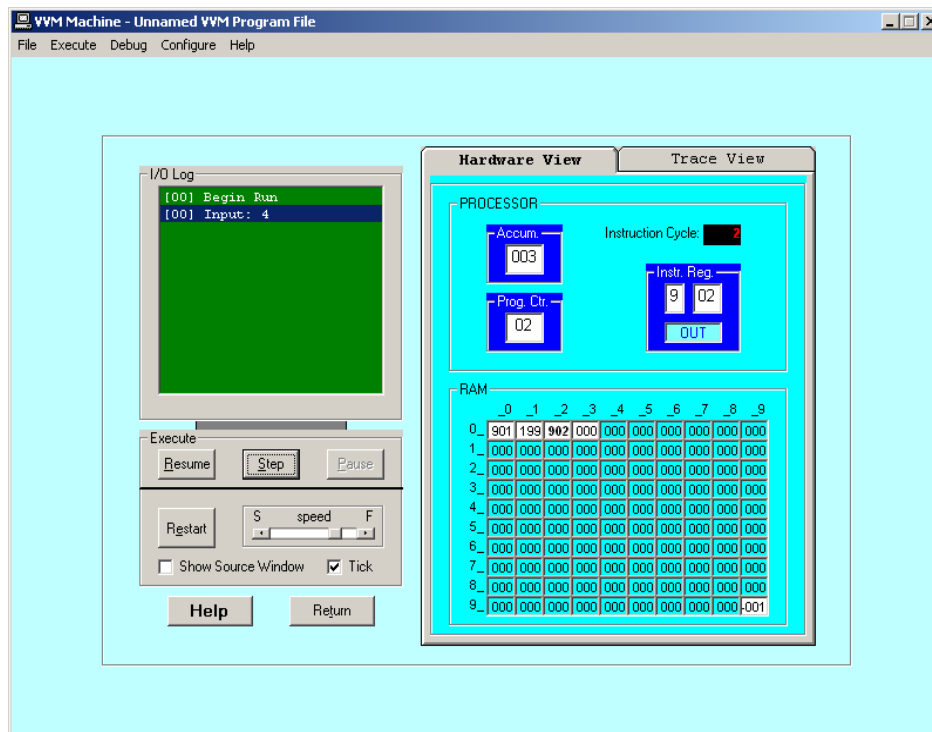
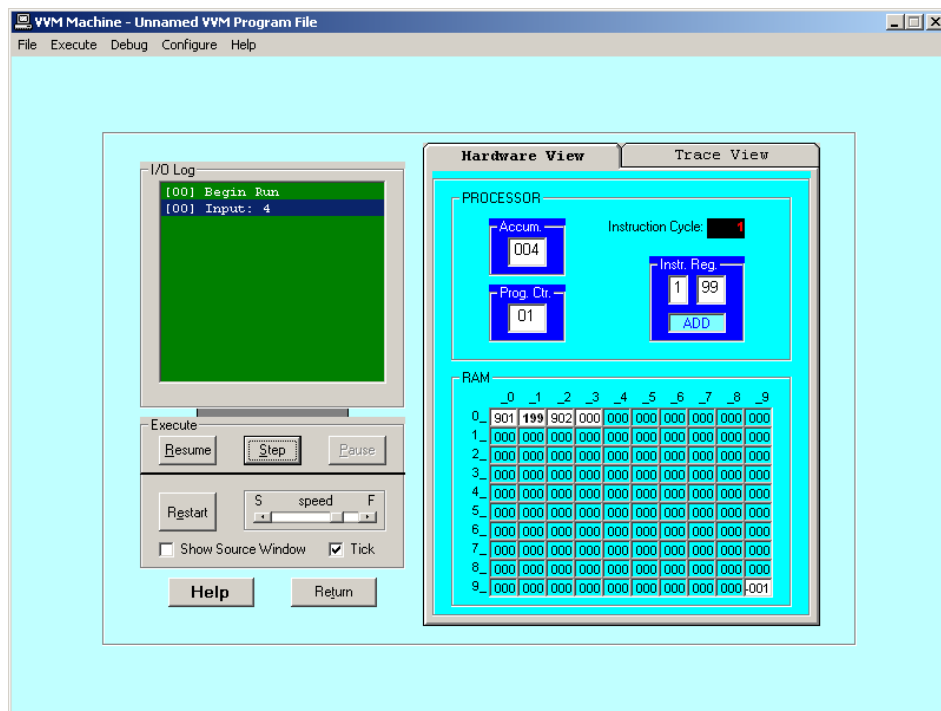


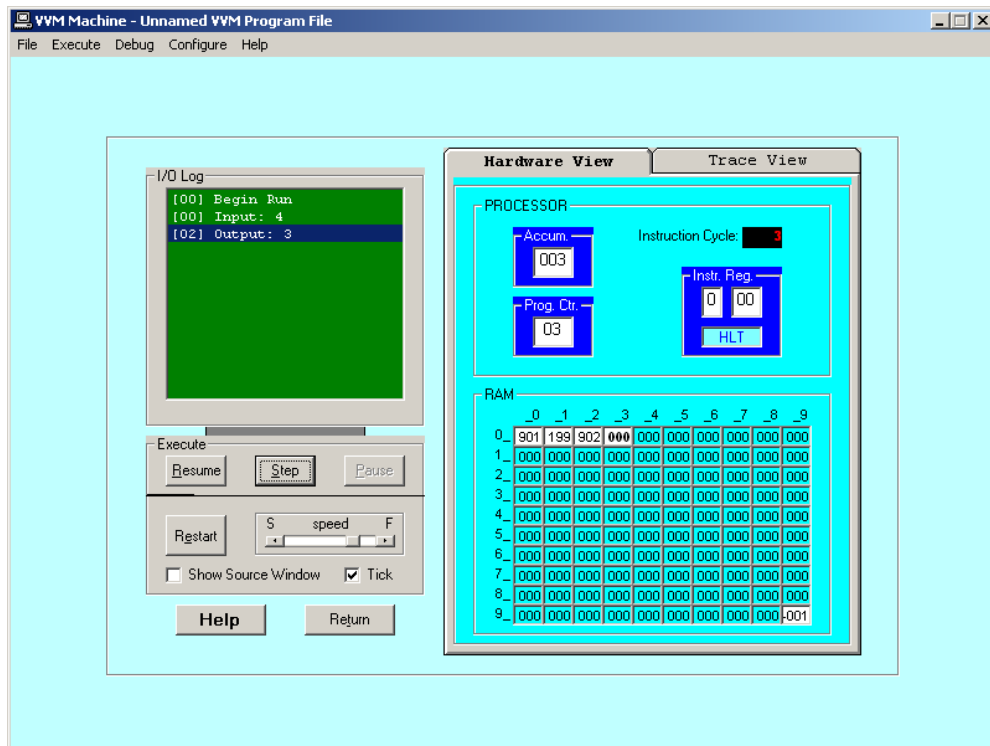
STEP4: Press the Load button



STEP5: Press the Step button

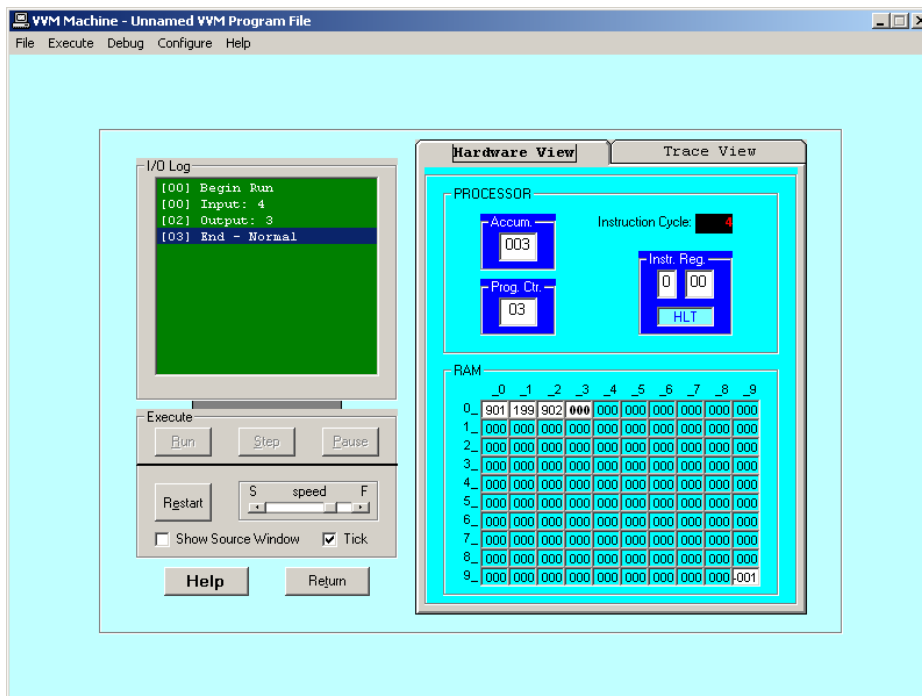




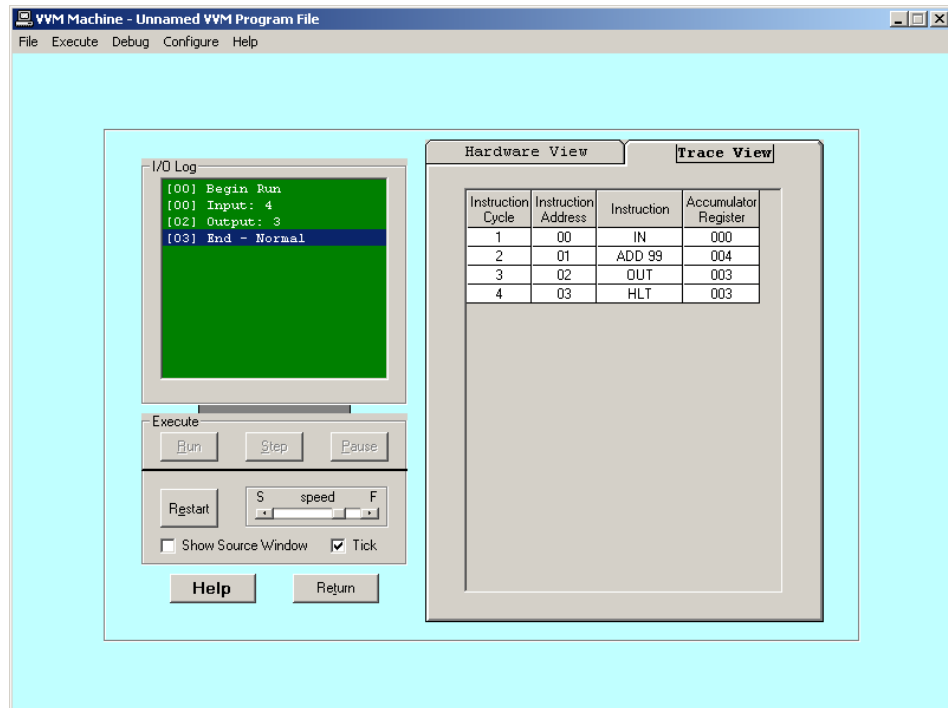


OUTPUT

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
StoreB
lda 98      Load value of A sub 99
            Subtract B fromA
brp 11      If A >= B, branch to 11 A is < B Find difference lda 98  Load value of
A
sub 99      Subtract value ofB 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] Printresult 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

Conditional Statement In assembly Language

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
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 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)

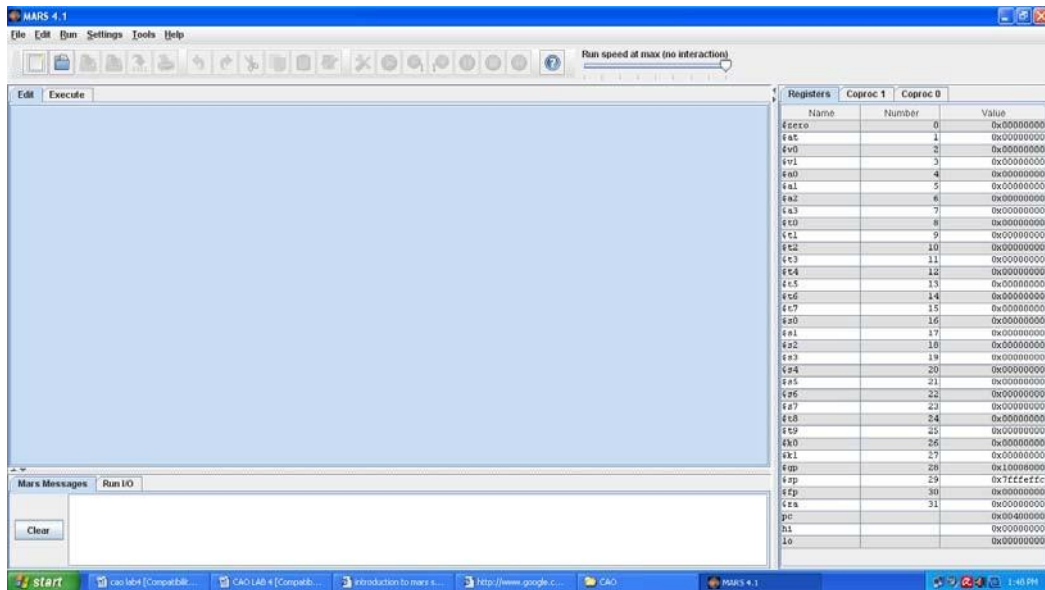
- ❖ Programs do input/output through system calls
- ❖ MIPS provides a special **syscall** instruction
 - To obtain services from the operating system
 - Many services are provided in the MARS simulators
 - There are 10 different services provided.
- ❖ Using the **syscall** system services
 - Load the 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.

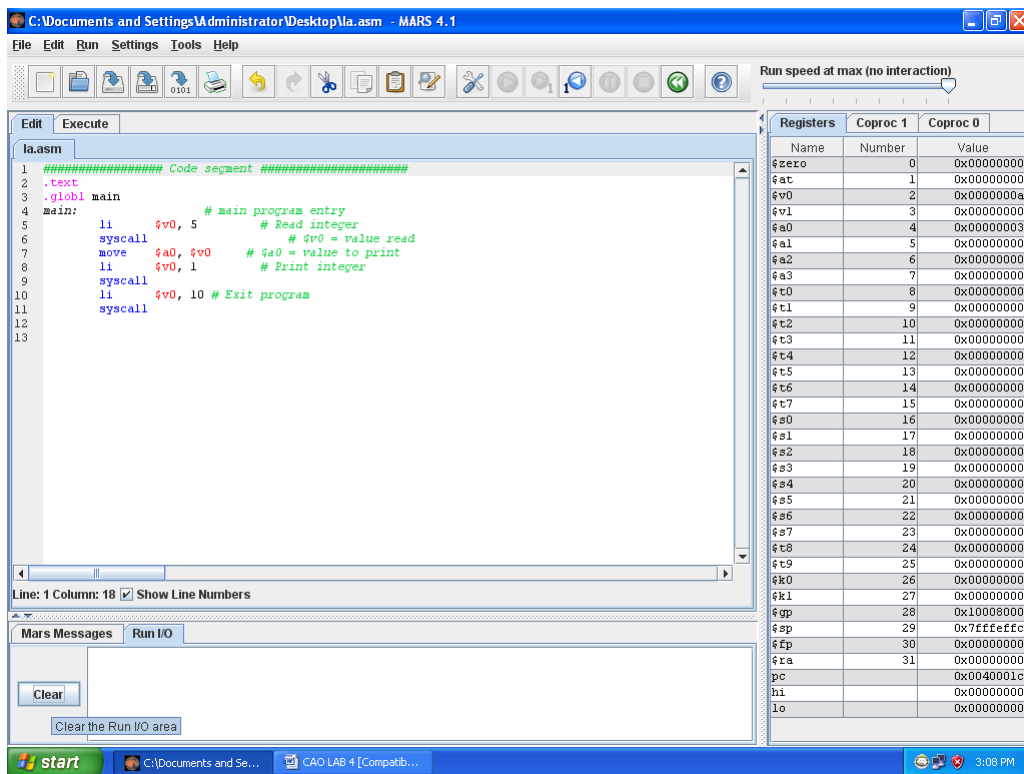
Program#1:

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
```

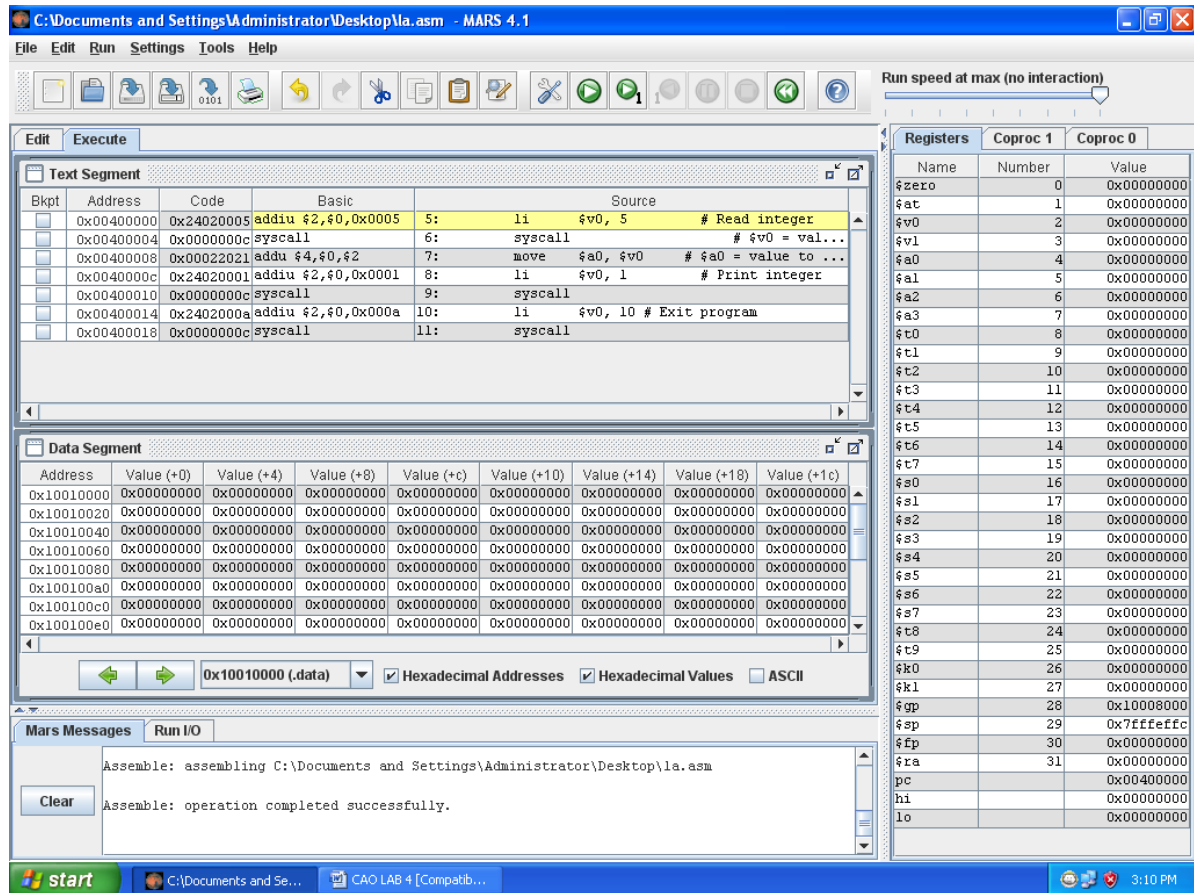
STEP#1

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



STEP# 2

Assemble program by pressing F3.



Text Segment				
Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020005	addiu \$2,\$0,0x0005	5: li \$v0, 5 # Read integer
	0x00400004	0x0000000c	syscall	6: syscall # \$v0 = value read
	0x00400008	0x00022021	addu \$4,\$0,\$2	7: move \$a0, \$v0 # \$a0 = value to print
	0x0040000c	0x24020001	addiu \$2,\$0,0x0001	8: li \$v0, 1 # Print integer
	0x00400010	0x0000000c	syscall	9: syscall
	0x00400014	0x2402000a	addiu \$2,\$0,0x000a	10: li \$v0, 10 # Exit program
	0x00400018	0x0000000c	syscall	11: syscall

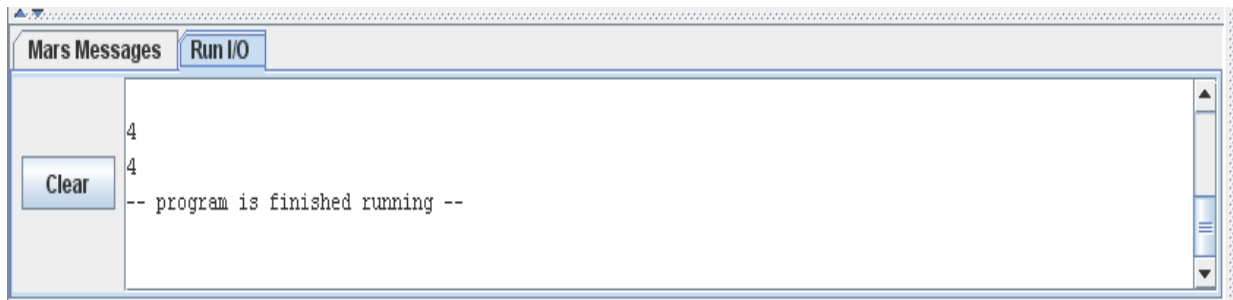
Data Segment									
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)	
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	▲
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	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,myMessage
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

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: .ascii "Please enter three numbers: \n" sum_msg: .ascii

"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.

Task 4: Write an assembly program that Read and Print Hello world.

Task 5: Write an assembly program that add two number within the range of 10.

Task 6: Write an assembly program that subtract two number within the range

LAB # 6

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	$rt = rs \text{immediate}$
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

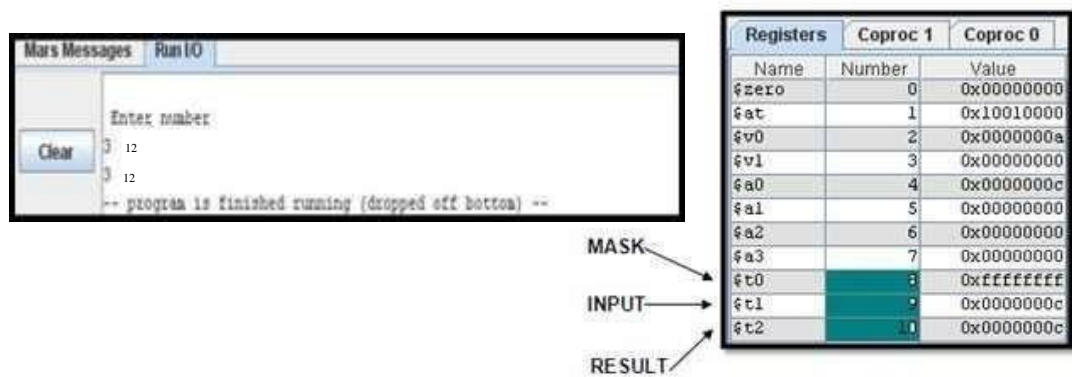
li \$v0,1 # Print value of \$t2

syscall

li \$v0,10 # Exit program

syscall

Output:



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

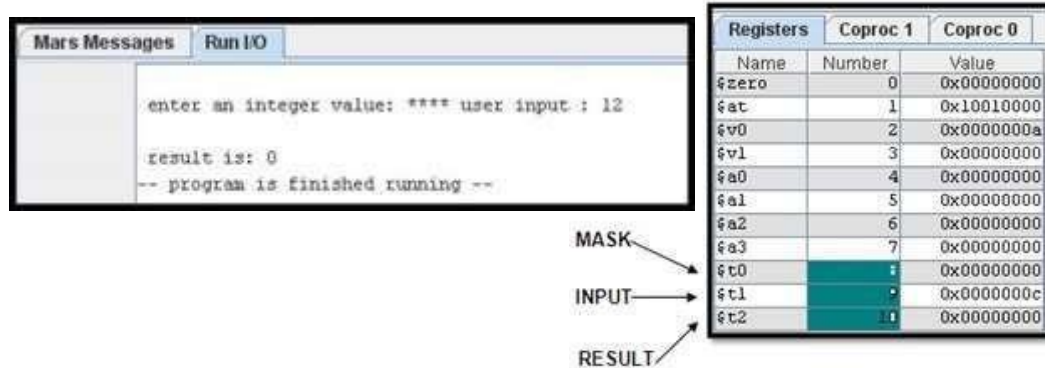
syscall

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 # 7

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

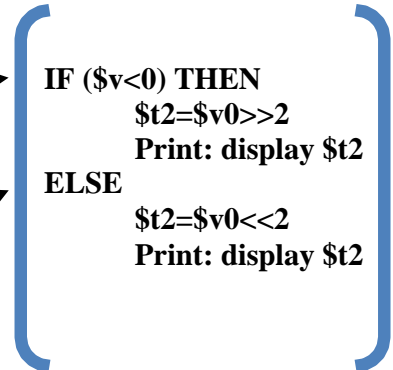
li \$v0,4

syscall

li \$v0,1

syscall

else:



```

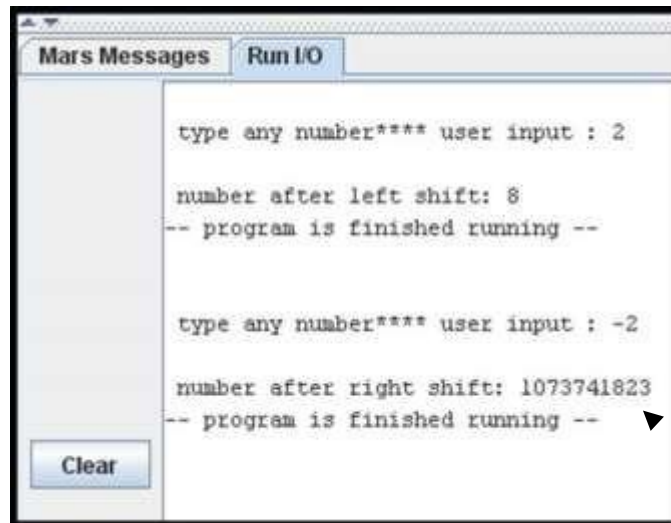
IF ($v<0) THEN
    $t2=$v0>>2
    Print: display $t2
ELSE
    $t2=$v0<<2
    Print: display $t2
  
```

```
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:



\$t2	.10	0x3fffffff
------	-----	------------

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 # 08

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
tab    li $v0,4      syscall

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

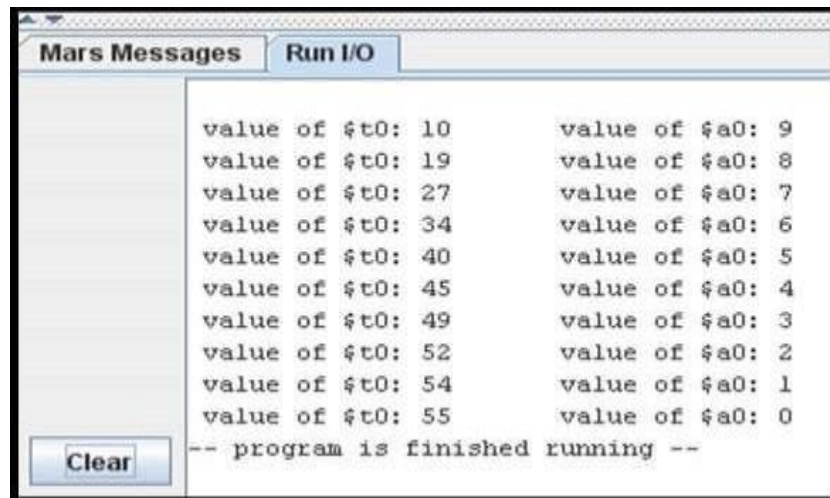
    move $a0,$a2      # print value of $a2
    li $v0,1
syscall

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

    end:    li
    $v0,10
    syscall

```

Output:



The screenshot shows a window titled "Mars Messages" with a "Run I/O" button. The window displays the following output:

Register	Value
\$t0	10
\$a0	9
\$t0	19
\$a0	8
\$t0	27
\$a0	7
\$t0	34
\$a0	6
\$t0	40
\$a0	5
\$t0	45
\$a0	4
\$t0	49
\$a0	3
\$t0	52
\$a0	2
\$t0	54
\$a0	1
\$t0	55
\$a0	0

At the bottom, there is a "Clear" button and the text "-- 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 Loo

LAB # 9

Introduction to Basic Instruments and Study of Logic Gates

OBJECTIVE:

- (a) Familiarization with the Lab Instruments.
- (b) To Study and Verify the Truth Tables of the Logic Gates.

EQUIPMENT:

- 1. Bread Board.
- 2. Digital Design Module.
- 3. Digital Logic Probe.
- 4. Wire Stripper.
- 5. ICs: 74LS00, 74LS02, 74LS04, 74LS08, 74LS32, 74LS86 and 74LS266.
- 6. DC supply (0 and +5V).

Part – I

(Familiarization with Lab Equipments)

THEORY:

Bread Board:

Bread Board is used to design and test different circuits. It is an array that contains the holes where simply the components and the connecting wires are pushed to form a circuit. One of the major advantage of using the Bread Board is that the components are not soldered. So, they can be plugged in / out easily if they are connected incorrectly. The Bread Board with its internal connections / structure is shown in Figure. 1.1.

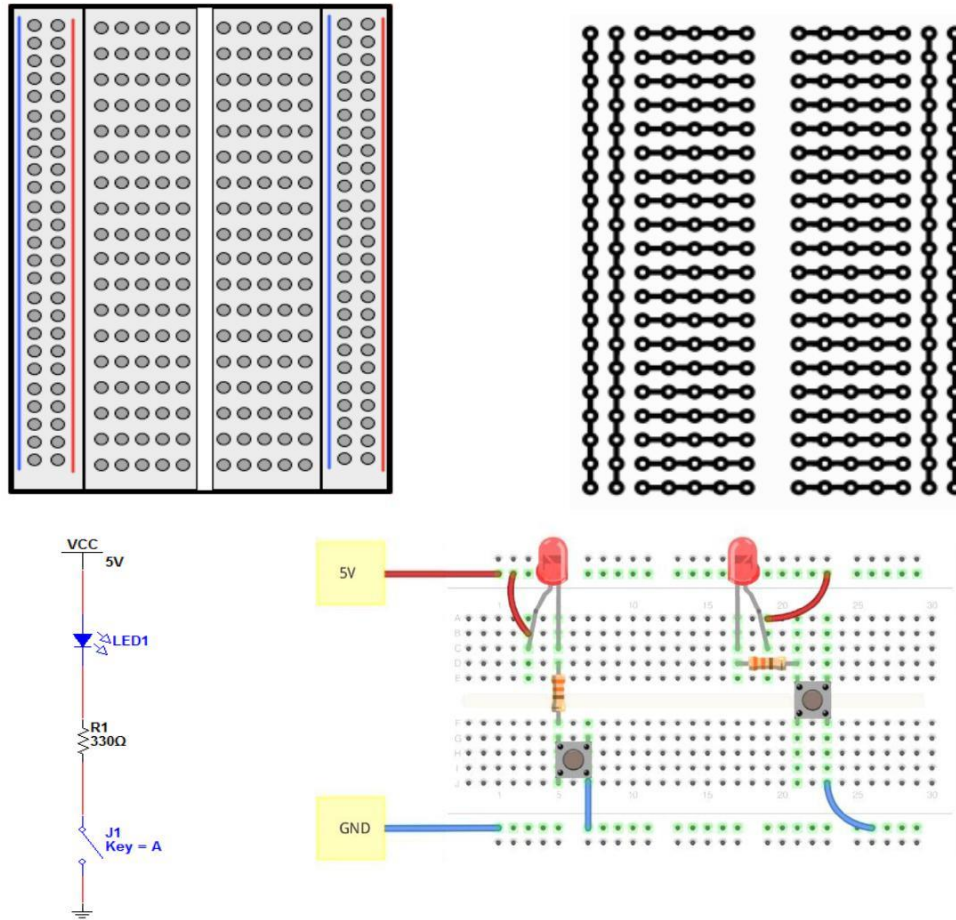


Figure. 1.1: Bread Board and its Internal Connections / Structure.

Digital Design Module (NI myDAQ):

NI myDAQ is a low-cost portable Data Acquisition device that uses the NI LabVIEW based software instruments, allowing the user to measure and analyze the real world signals. NI myDAQ is ideal for the Electronics and taking the sensor measurements. Combined with NI LabVIEW on the PC, one can analyze and process the required signals and control simple processes anytime, anywhere.

myDAQ provides Analog Input (AI), Analog Output (AO), Digital Input and Output (DIO), Audio Input and Output (AIO), DC Power Supplies, and Digital Multimeter (DMM) functions in a compact USB device. The myDAQ connection diagram is shown in Figure. 1.3.



Figure. 1.2: NI myDAQ

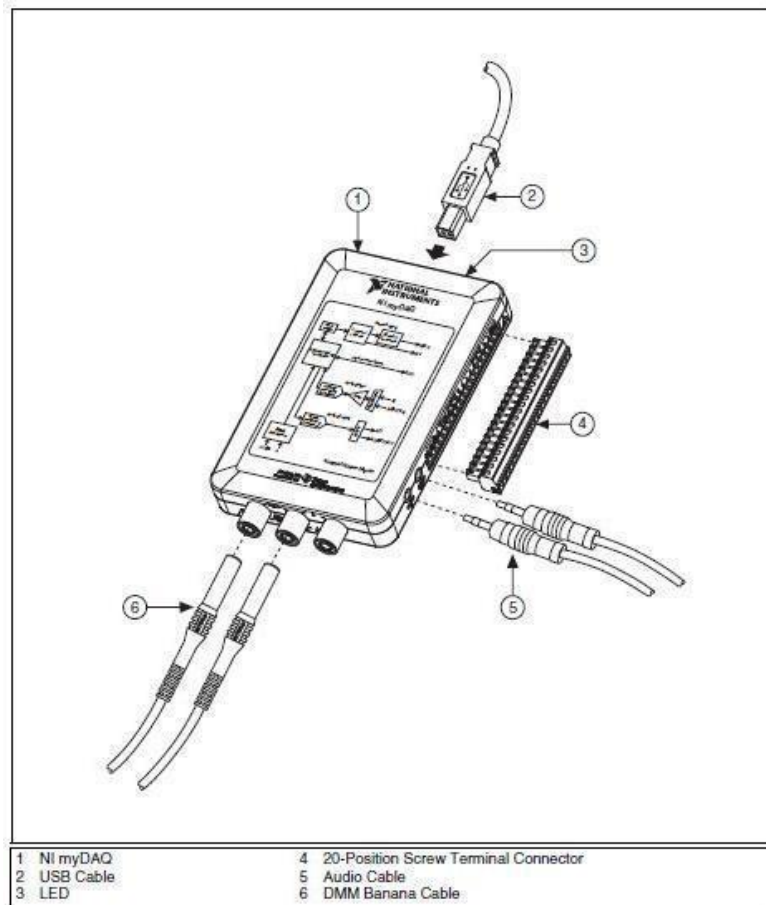


Figure. 1.3: myDAQ Connection Diagram

The myDAQ provides a soft front panel (from NI LabVIEW) to control the functionality of the device, which can launch several instruments including Digital Multimeter (DMM), Oscilloscope, Function Generator, Bode Analyzer, Dynamic Signal Analyzer, Arbitrary Waveform Generator and Digital Reader / Writer

Application Software Details:

The software that is used to interface the myDAQ card with PC is *NI ELVISmx for NI ELVIS NI myDAQ*. Through this software, the different device components of NI myDAQ can be operated which includes Digital Multimeter, Bode Analyzer, Oscilloscope, Function Generator, Impedance Analyzer, etc. etc.

For opening this application software, follow the path as:

Start → *All Programs* → *National Instruments* → *NI ELVISmx for NI ELVIS & NI myDAQ* → *NI ELVISmx Instrument Launcher*. As shown in figure 1.4

All the myDAQ devices are shown in this interface that shows that it can be used for analog applications as well besides the digital. We can access any of the device by just clicking on its icon. For the Digital Lab, we will use just three applications devices. i.e. Digital Multimeter, Digital Reader and Digital Writer. These are shown as in figures Figure 1.5 – 1.7.

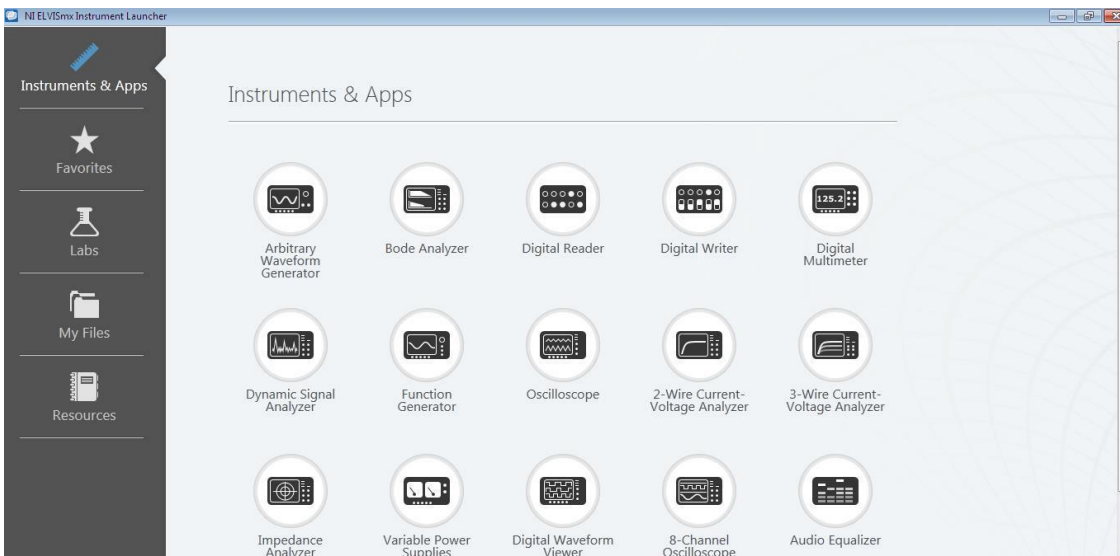


Figure. 1.4: NI Instrument Launcher

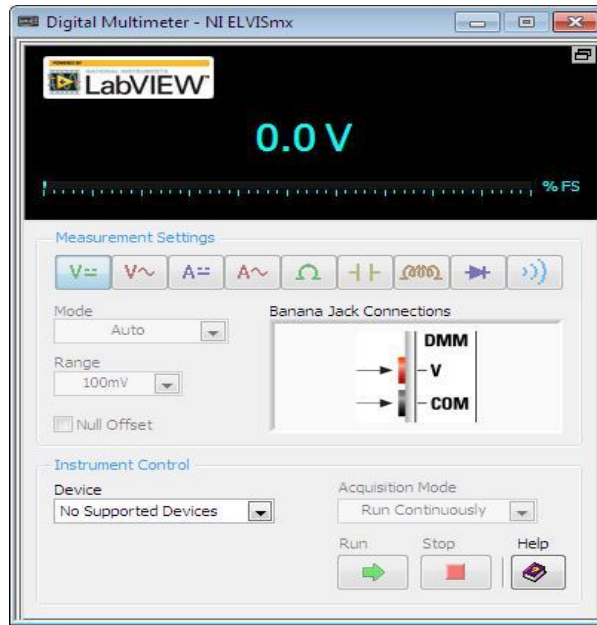


Figure. 1.5: Digital Multimeter Interface

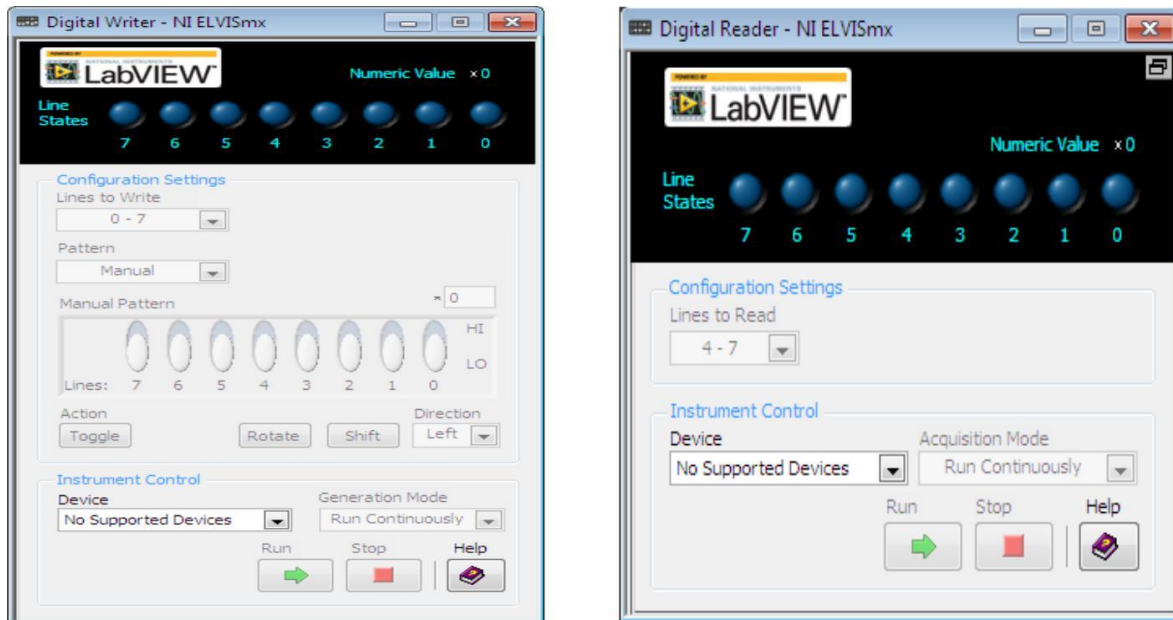


Figure. 1.6: Digital Reader & Digital Writer Interfaces

After making all the necessary connections for the Signal Reading / Writing purpose, the RUN button should be pressed in order to get the readings on interface display.

Digital Logic Probe:

A **logic probe** is a hand-held pen-like test probe used for analyzing and troubleshooting the logical states (Boolean 0 or 1) of a digital circuit. While most are powered by the circuit under test, some devices use batteries. They can be used on either TTL (transistor-transistor logic) or CMOS (complementary metallic oxide semiconductor) integrated circuit devices. There are usually three differently-colored LEDs on the probe's body:

Red and green LEDs indicate high and low states respectively.

An amber LED indicates a pulse.

The pulse-detecting electronics usually has a pulse-stretcher circuit, so that even very short pulses become visible on the amber LED. A control on the logic probe allows either to capture and storage of a single event or continuous running. Probe is shown in Figure. 1.7.

When the logic probe is either connected to an invalid logic level (a fault condition or a tri-stated output) or not connected at all, none of the LEDs lights up. Another control on the logic probe allows selection of either TTL or CMOS family logic. This is required as these families have different thresholds for VIH and VIL. Some logic probes have a separate audible tone for each of the logical states. An oscillating signal causes the probe to alternate between high-state and low-state tones. A logic probe is a cheap, versatile and convenient digital test instrument, but can test only a single signal at a time. When many logic levels need to be observed or recorded simultaneously, a logic analyzer is used.



Figure. 1.7: Digital Logic Probe.

Wire Stripper:

It is a tool that is used to remove the insulation from the wire or cut the wire.



Figure. 1.8: Wire Stripper

Safety:

Following proper safety practices are a must when working with electronic equipment. Not only is there the danger of electrical shock, but the components can explode if not connected properly. Many of today's electronic components are easily damaged by improper handling. The test equipment used in the electronic service industry is expensive and easily damaged if proper operating procedures are not followed. Make sure TEST INSTRUMENTS are set for proper FUNCTION AND RANGE prior to taking a measurement.

1. Always cut wire leads so the clipped wire falls on the table top and not toward others.
2. Avoid skin contact.
3. Only work with powered units when necessary for troubleshooting.
4. Avoid pinching wires when putting equipment back together.
5. Never solder a circuit that has the power applied.
6. Double check circuits for proper connections and polarity prior to applying the power.
7. Observe polarity when connecting polarized components or test equipment into a circuit.

Part – II (Study of Logic Gates)

THEORY:

AND Gate:

The AND function is similar to the multiplication in mathematics. This is the all or nothing operator and it provides a logic 1 output only when all the inputs of the gate are at logic 1, and logic 0 output for all other input combinations. The logic operator for the AND function is a dot (\cdot) sign. The AND function is described in terms of the following “truth table”.

(Boolean Expression is “ $Output = A \cdot B$ ”).

TABLE 1.1: Function of AND Gate

Input: A	Input: B	Output $Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

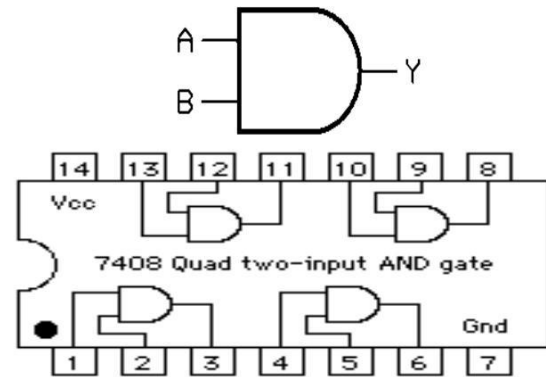


Figure. 1.9: AND Gate Symbol & TTL IC

OR Gate:

The OR function is similar to the mathematical function of addition and the output for the OR gate may be analyzed using the laws of addition. The logic operator for the OR function is a plus (+) sign. The output will be logic 0 only if all the inputs are logic 0, and the output will be logic 1 anytime any input is at logic 1.

(Boolean Expression is as “ $Output = A + B$ ”).

TABLE 1.2: Function of OR Gate

Input: A	Input: B	Output $Y = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

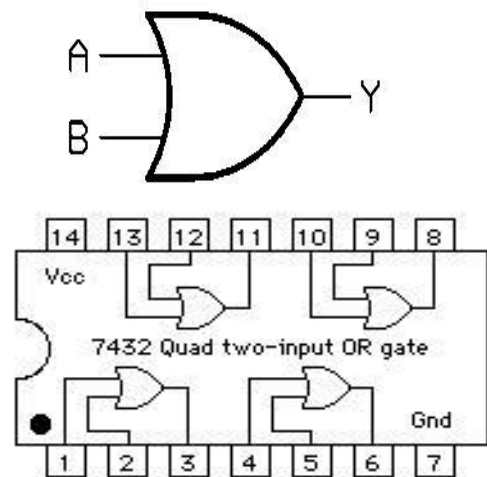


Figure. 1.10: OR Gate Symbol & TTL IC

NOT Gate:

The NOT circuit or inverter performs the basic logic function of complementation. It may be identified by the presence of a bubble on the input or the output of the traditional logic symbol. The output of NOT gate is the inverse of the input. Unlike the others it only has one input and one output.

(Boolean Expression is as “ $Output = A'$ ”).

TABLE 1.3: Function of NOT Gate

Input: A	Output: $Y = A'$
0	1
1	0

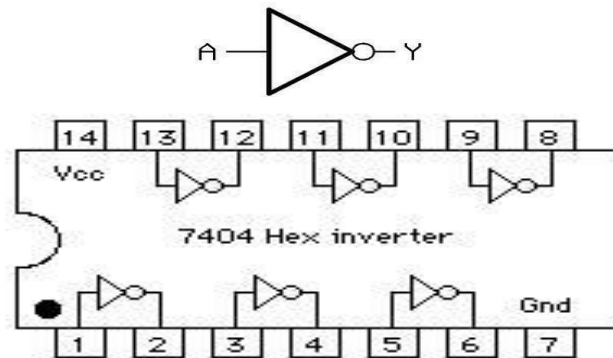


Figure. 1.11: NOT Gate Symbol & TTL IC

NAND Gate:

The NAND function is the complement of the AND function and the logic symbols have the inversion on the output. NAND gate is constructed by adding an inverter after AND operator. The NAND function provides logic 0 on the output only when all inputs are logic 1, and logic 1 output for all other combinations.

(Boolean Expression is as “ $Output = (A \cdot B)'$ ”).

TABLE 1.4: Function of NAND Gate

Input: A	Input: B	Output: $Y = (A \cdot B)'$
0	0	1
0	1	1
1	0	1
1	1	0

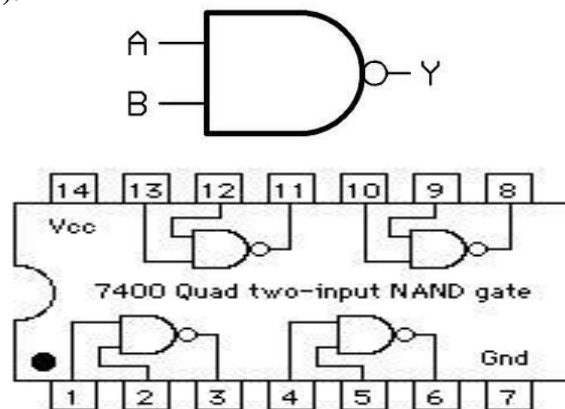


Figure. 1.12: NAND Gate Symbol & TTL IC

NOR Gate:

The complement of the OR function is the NOR function and the logic symbol has the inversion present on the output. NOR gate is constructed by adding an inverter after OR operator. The NOR function provide logic 1 when all input are 0 and 0 for all other combinations.

TABLE 1.5: Function of NOR Gate

Input:A	Input: B	Output: $Y = (A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0

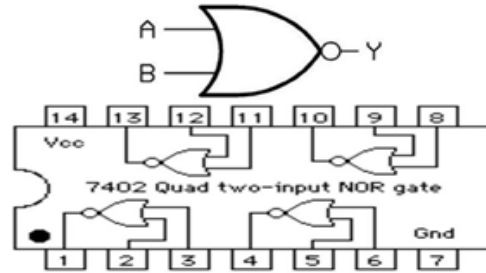


Figure. 1.13: NOR Gate Symbol & TTL IC

Exclusive OR Gate:

The Exclusive OR (also known as XOR) gate is a logic function that results true if one and only one input to the gate is 1. In other words, when the inputs are at different logic levels, the output is 1 and when the inputs are at the same logic levels, the output is 0.

(Boolean Expression is as " $Output = A \oplus B$ ").

TABLE 1.6: Function of XOR Gate

Input: A	Input: B	Output: $(A \oplus B)$
0	0	0
0	1	1
1	0	1
1	1	0

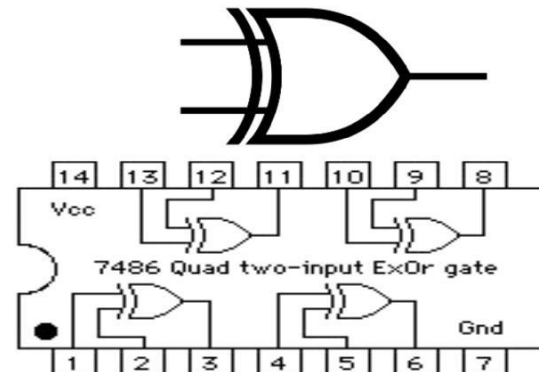


Figure. 1.14: XOR Gate Symbol & TTL IC

Exclusive NOR Gate:

The Exclusive NOR (also known as XNOR) is a function that is the complement of the XOR Function that results in 1 when the inputs are same and results 0 when the inputs are different.

(Boolean Expression is as " $Output = (A \oplus B)'$ ").

TABLE 1.7: Function of XNOR Gate

Input:A	Input: B	Output: $Y = (A \oplus B)'$
0	0	1
0	1	0
1	0	0
1	1	1

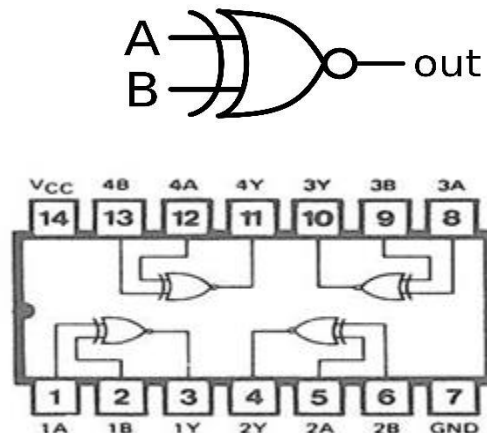


Figure. 1.15: XNOR Gate Symbol & TTL IC

PROCEDURE:

1. Place the respective ICs (74LS400, 74LS02, 74LS04, 74LS08, 74LS32, 74LS86 and 74LS266) on the trainer board.
2. Connect VCC and ground to the respective pins on the trainer board.
3. Connect the inputs to the input switches provided in the trainer board.
4. Connect the outputs to the switches of output LEDs and apply various combinations of inputs as shown in the truth table and observe the conditions of LEDs.

OBSERVATIONS / RESULTS & DISCUSSION:

Input: A	Input: B	7400 NAND	7402 NOR	A•B 7408 AND	A+B 7432 OR	(A⊕B) 7486 XOR	(A⊕B)' 74266 XNOR
0	0						
0	1						
1	0						
1	1						
0	Not Connected						
1	Not Connected						
Not Connected	0						
Not Connected	1						
Not Connected	Not Connected						

CONCLUSION:

1. The output of AND Gate is only high when all inputs are high.
2. The output of OR Gate is low when all inputs are low.
3. The output of NOT Gate is inverse of input.
4. The output of NAND Gate is low when all inputs are high.
5. The output of NOR Gate is high when all inputs are low.
6. The output of XOR Gate is high when the inputs are at different logic levels.
7. The output of XNOR Gate is high when the inputs are at same logic levels.

QUESTIONS:

1. How do these gate treat when inputs are unconnected (Low or High)? Briefly explain why?
2. Do swapping A and B make any difference?
3. What is difference between an Inverter and a **NOT** gate?
4. How can you make an inverter using a **NOR** gat? And using **NAND** gate?
5. Implement XOR gate function using basic gates?
6. Implement XNOR gate function using basic gates?

Lab 10

MultiSim Tutorial and Logic Implementation

OBJECTIVE:

- (a) To understand the MultiSim.
- (b) To Implement the Digital Logic Equation.

THEORY:

Multisim is the schematic capture and simulation application of National Instruments Circuit Design Suite, a suite of EDA (Electronic Design Automation) tools that assists you in carrying out the major steps in the circuit design flow. Multisim is designed for schematic entry, simulation, and feeding to downstage steps, such as PCB layout.

In order to start Multisim

Click to Start —————> All Programs —————> National Instruments —————> Circuit
Design Suite —————> Multisim

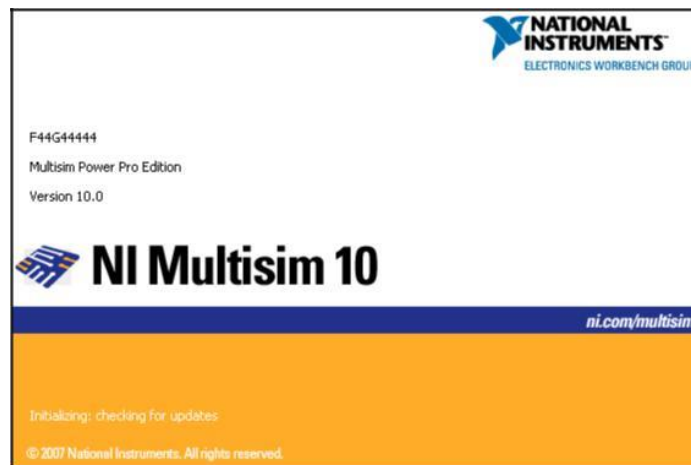


Figure. 2.1: MultiSim 10.0 Runtime Process Window

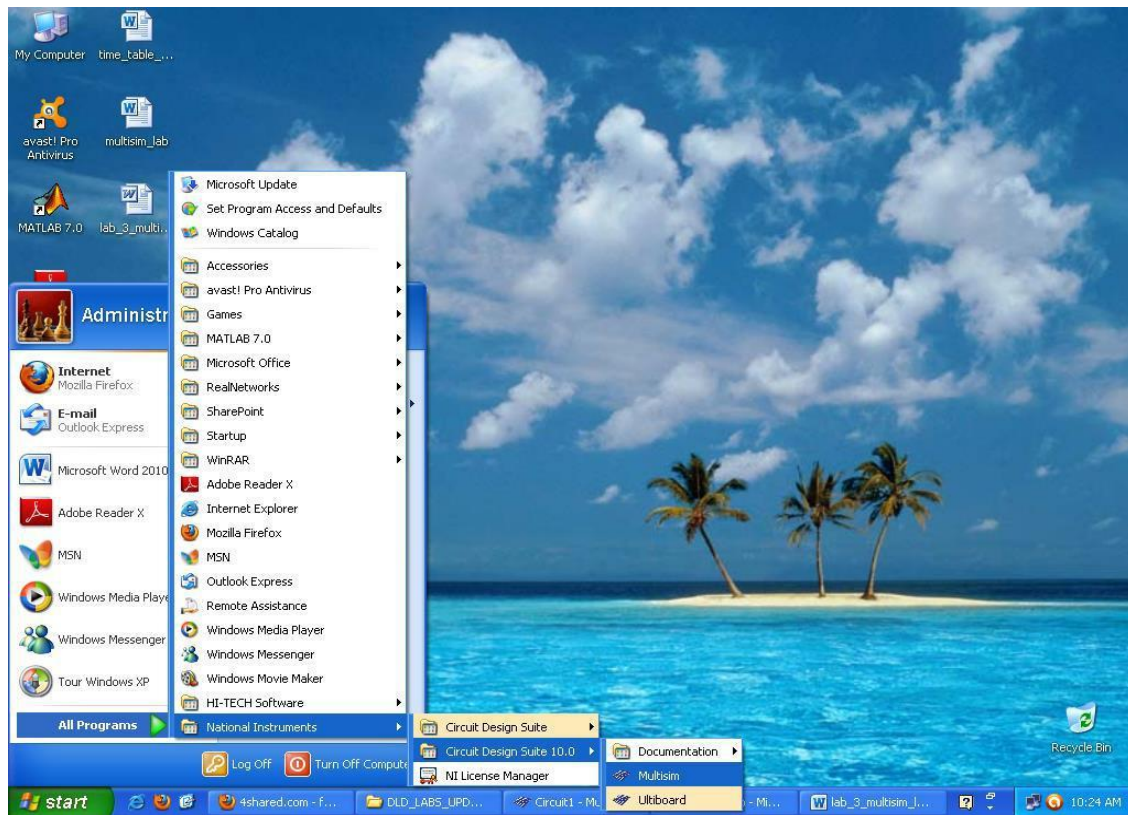


Figure. 2.2: Start Menu Path for MultiSim 10.0 (Circuit Design Suite)

Menus & Toolbars:

7. **Menus** are where you find commands for all functions.
8. The **Standard** toolbar contains buttons for commonly-performed functions.
9. The **Simulation** toolbar contains buttons for starting, stopping, and other simulation functions.
10. The **Instruments** toolbar contains buttons for each instrument.
11. The **Component** toolbar contains buttons that let you select components from the Multisim databases for placement in your schematic.
12. The **Circuit Window** (or workspace) is where you build your circuit designs.
13. The **Design Toolbox** lets you navigate through the different types of files in a project (Schematics, PCBs, reports), view a schematic's hierarchy and show or hide different layers.
14. The **Spreadsheet View** allows fast advanced viewing and editing of parameters including component details such as footprints, RefDes, attributes and design constraints. Users can change parameters for some or all components in one step and perform a number of other functions.

From the above toolbar the most commonly used toolbar is component toolbar:

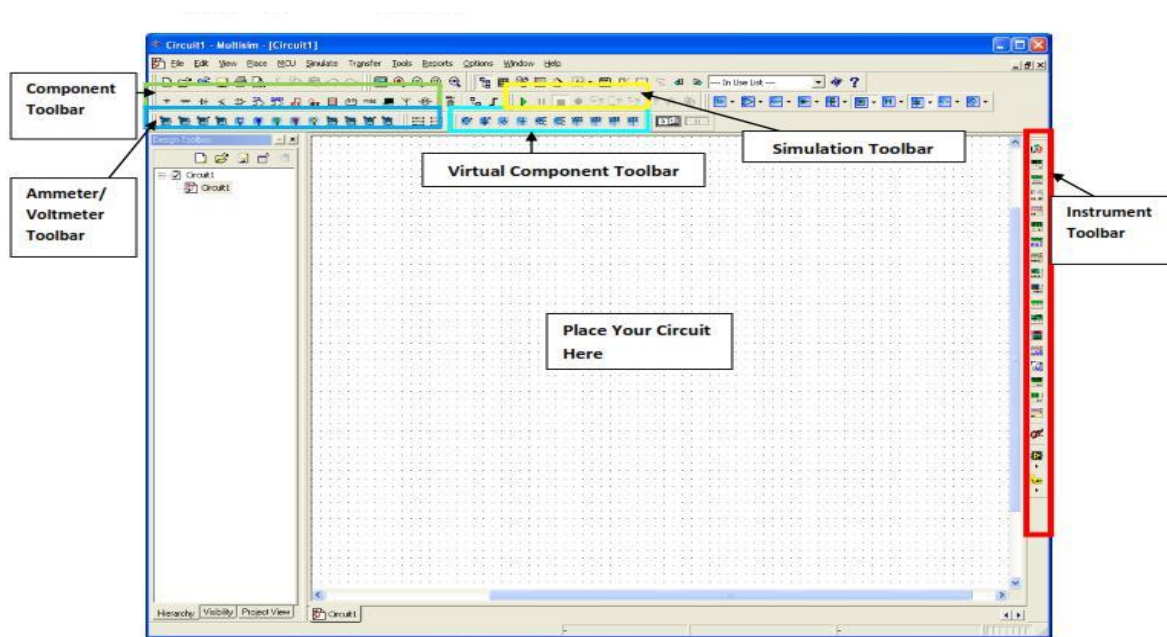


Figure. 2.3: MultiSim – Working Environment

Components Toolbar:

The buttons in the **Components** toolbar are described below. Each button will launch the Place component browser (**Select a Component browser**) with the group specified on the Button pre-selected.

Button	Description
	Source button. Selects the Source components group in the browser.
	Basic button. Selects the Basic components group in the browser.
	Diode button. Selects the Diode components group in the browser.
	Transistor button. Selects the Transistor components group in the browser.
	Analog button. Selects the Analog components group in the browser.
	TTL button. Selects the TTL components group in the browser.
	CMOS button. Selects the CMOS component group in the browser.
	Miscellaneous Digital button. Selects the Miscellaneous Digital component group in the browser.
	Mixed button. Selects the Mixed component group in the browser.
	Power Components button. Selects the Power component group in the browser.
	Indicator button. Selects the Indicator component group in the browser.
	Miscellaneous button. Selects the Miscellaneous component group in the browser.
	Electromechanical button. Selects the Electromechanical component group in the browser.
	RF button. Selects the RF component group in the browser.
	Place Advanced Peripherals button. Selects the Advanced Peripherals component group in the browser.
	Place MCU Module button. Selects the MCU Module component group in the browser.

Figure. 2.4: MultiSim – Component Toolbar

PROCEDURE:

Open/Create Schematic:

1. A new schematic circuit 1 is automatically created.
2. To create new schematic circuits click on **File- New- Schematic Capture**.
3. To save the schematic click on **File/Save As**.
4. To open an existing file click on **File/Open** in the toolbar.

Place Components:

Now place the components which are necessary to perform the experiment.

Steps to place the component:

1. To Place Components right click on circuit window and then select **Place Components, (Ctrl + w)**.

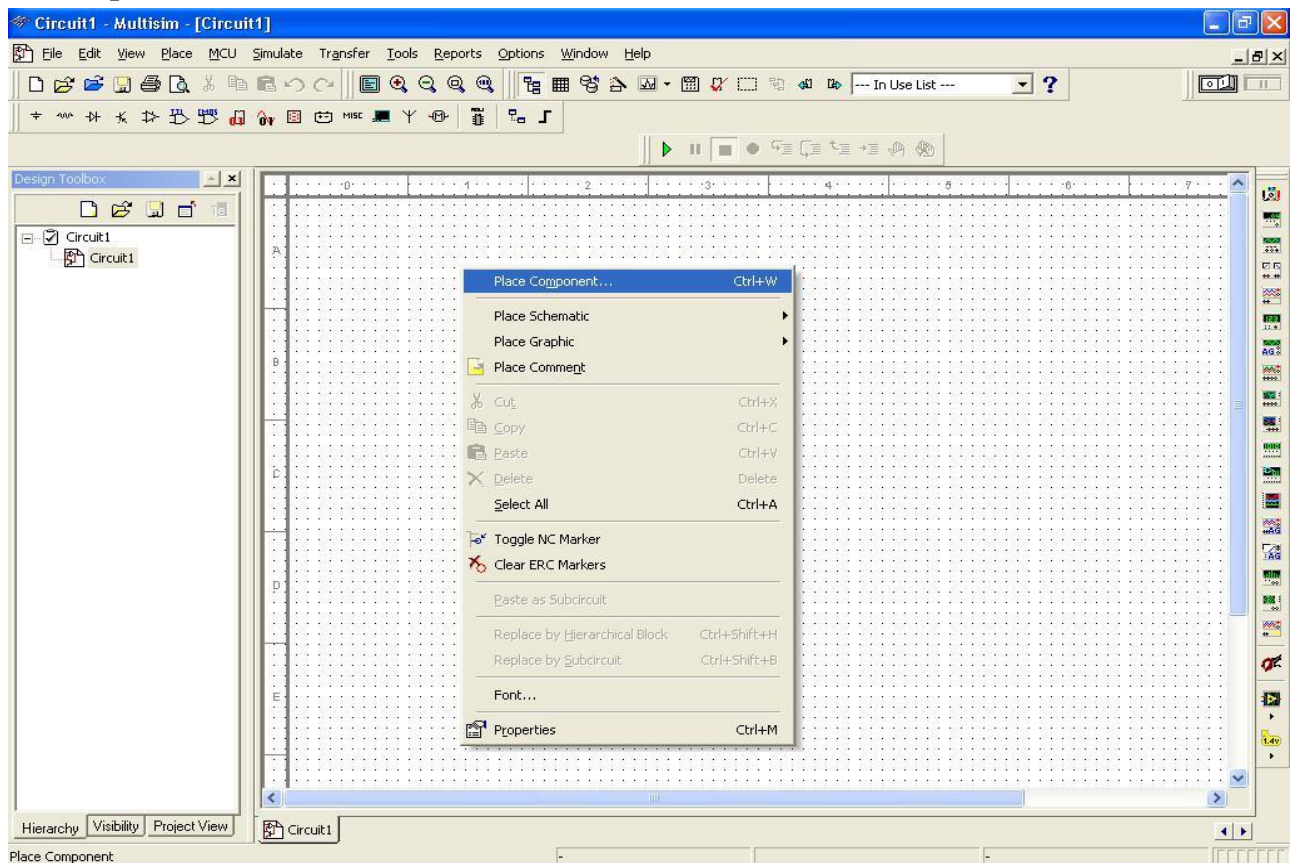


Figure 2.5: MultiSim Environment depicting “How to Place Component”

2. On the Select Component window click on **Group** and then select the Family in which your component is. Let suppose I want to place the VCC select the following:
 - ☐ Group: Sources
 - ☐ Family: Power Sources
 - ☐ Component: VCC

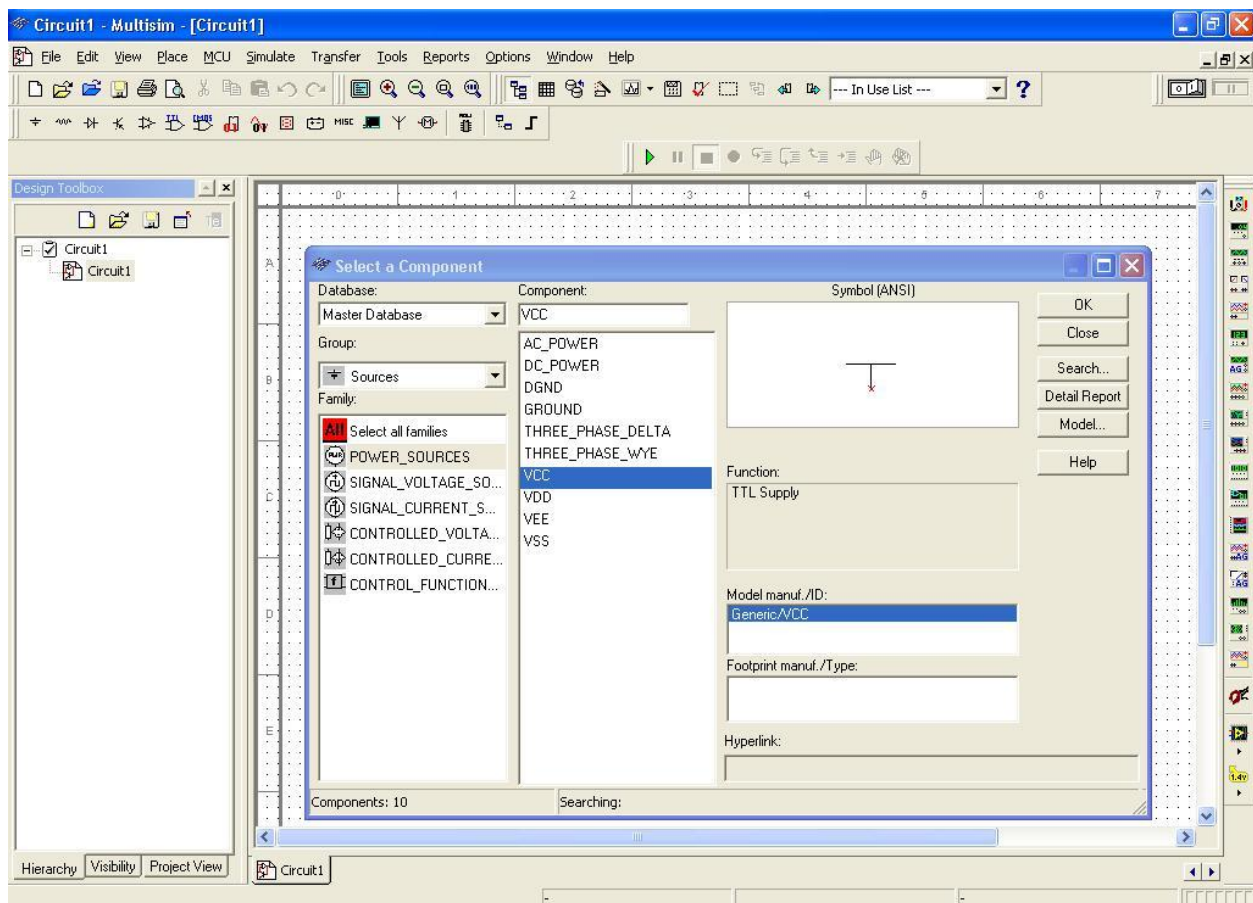


Figure 2.6: MultiSim – Component Library

8. Then Click OK, now component is on your mouse tip clip any where on circuit window to place the component on the schematic.

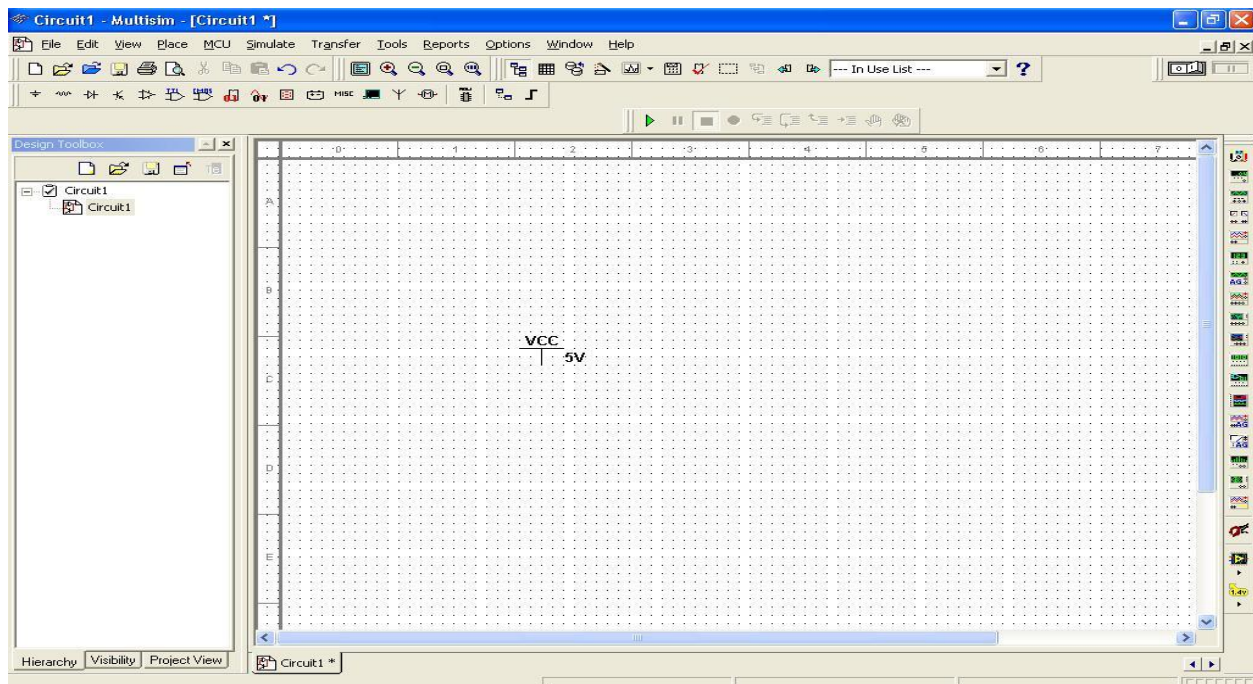


Figure 2.7: MultiSim Environment depicting component placed on Grid

How To Make Your First Circuit And Perform Simulation:

1. Now I want to see the simulation of basic gates on multisim...like AND gate...
2. First of all place the components which are necessary to perform the experiment.
3. Place AND-gate, supply (VCC), switch (SPDT), and indicator (PROBE or LED).

	GROUP	FAMILY	COMPONENT
SUPPLY	SOURCES	POWER SOURCES	VCC
SWITCH	BASIC	SWITCH	SPDT
AND GATE	TTL	SELECT ALL FAMILIES	7408N
AND GATE	MISC DIGITAL	TIL	AND2
PROBE	INDICATOR	PROBE	PROBE

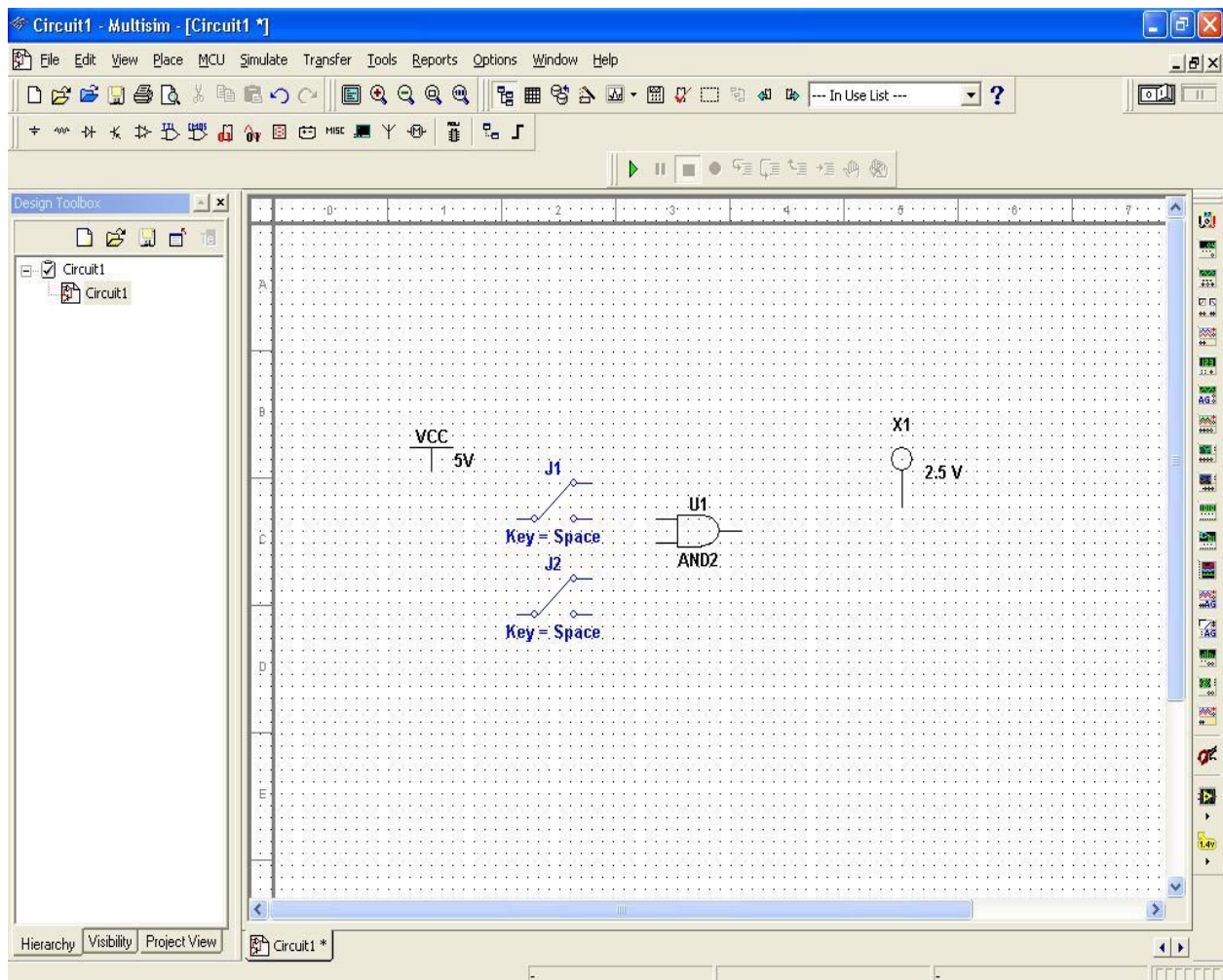


Figure. 2.8: MultiSim Environment depicting components placed on Grid

Now connect all the component and then starts simulation.

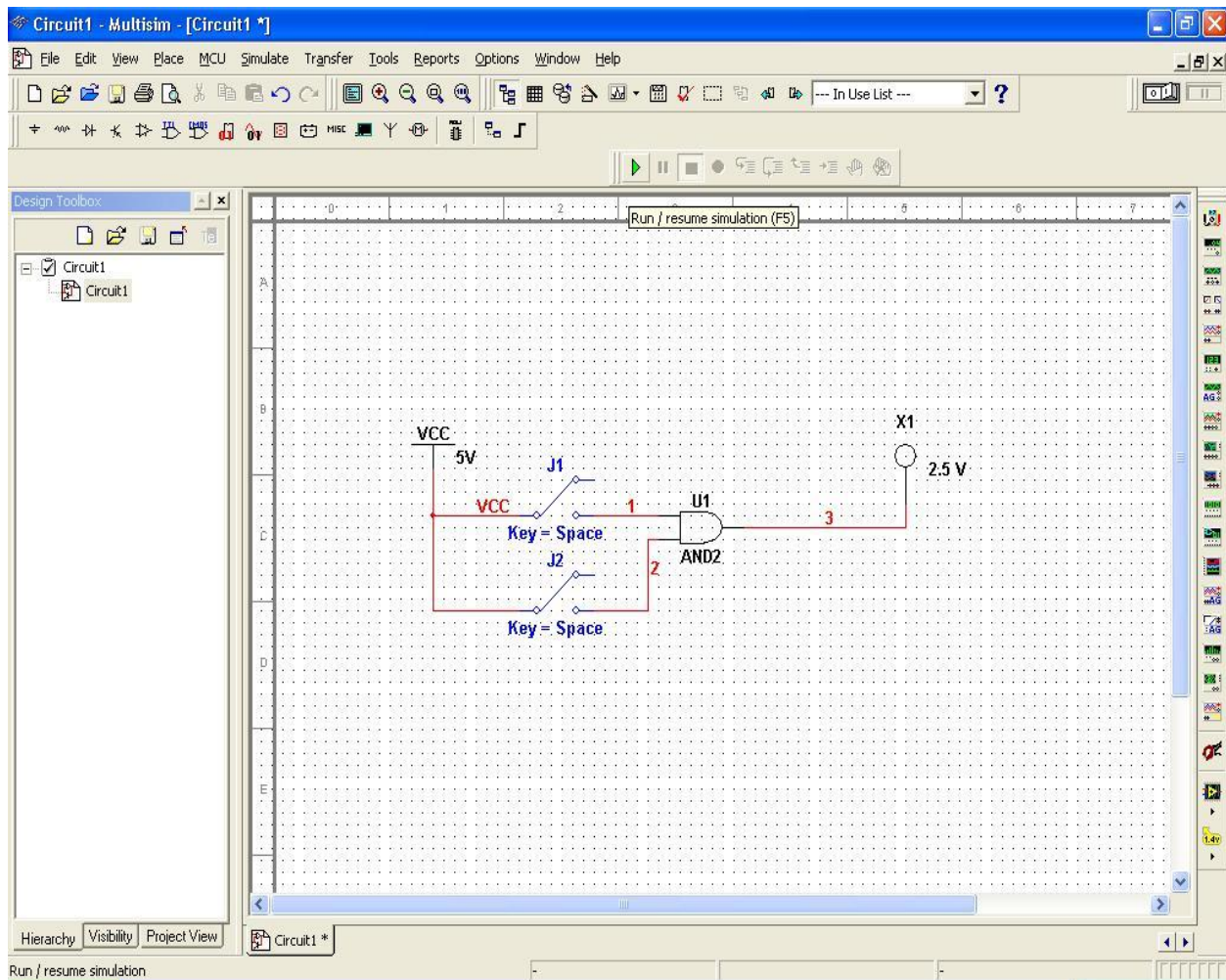


Figure 2.9: MultiSim Environment depicting designed Circuit

Simulation:

To simulate the completed circuit Click on **Simulate/Run** or **F5**. This feature can also be accessed from the toolbar as shown in the Figure 10 below.



Figure. 2.10: MultiSim Simulation Toolbar

Digital Logic Equation Implementation:

Logic Equation is a combination of basic, universal and / or exclusive gates that results in a specific functionality. Each digital circuit is represented by the equation that can be verified by a verification table known as Truth Table. So, in order to implement the equation, the appropriate gates needs to be connected in the way as the equation describes. The implemented circuit should verify the Truth Table associated with the equation.

Example 1: $(A + B)(C) + A' \cdot (B + C') + ABC$

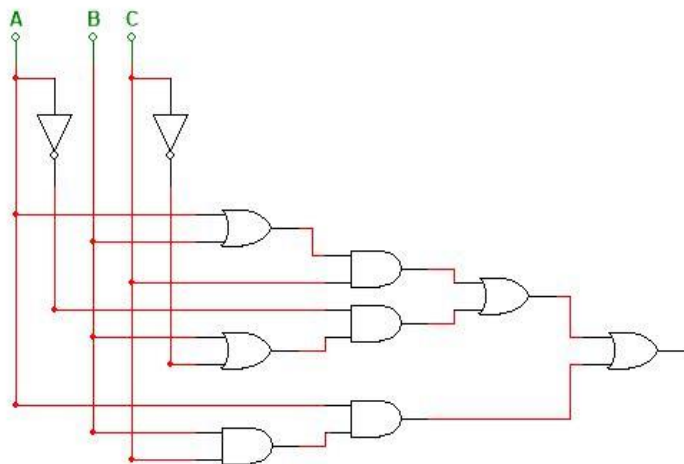


Figure. 2.11: Circuit Diagram – Example 1

TABLE 2.1: Truth Table – Example 1

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Example 2: $A'BC + AB'C + A'B'(B + C)B'C$

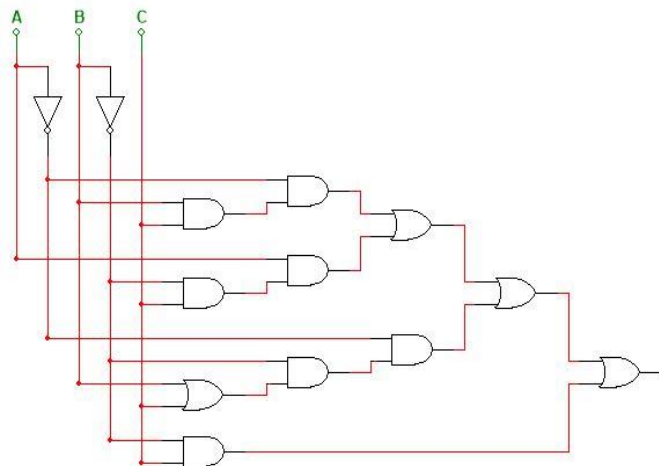


Figure 2.12: Circuit Diagram – Example 2

TABLE 2.2: Truth Table – Example 2

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

CONCLUSION:

1. Easy way to implement the circuits.
2. Easy to troubleshoot.
3. Less time required.
4. Complex circuit can be implemented very easily.
5. No need to purchase expensive component to implement and verify the operation of a circuit.
6. Digital Logic can be represented in the form of digital equations.
7. Digital Equation is a combination of gates.
8. Logic Circuit can be verified by the Verification / Truth Table.

QUESTIONS:

1. Verify the operation of NOT, AND, OR, NAND and NOR Gates using MultiSim and attach the circuit snapshots?
2. Implement the following equations on MultiSim and verify through Truth Table.
(Attach the circuit snapshots)

a. $\bar{A}\bar{B}CD + A(B + C) + B\bar{C}\bar{D} + AB\bar{D} + BC\bar{D}$

b. $(A + \bar{B} + C) \cdot (\bar{A} + \bar{C} + D) \cdot (B + C + \bar{D})$

LAB #11

Boolean algebra and Logic Simplification

OBJECTIVE:

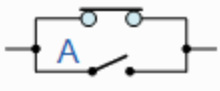
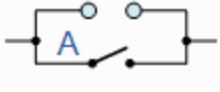
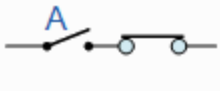
- (a) Introduction to Boolean algebra and Familiarization of Boolean Laws.
- (b) Simplification of Boolean Logic Equation.

EQUIPMENT:

- 15. IC: 7400LS, 7404LS, 7408LS and 7432LS.
- 16. Bread board.
- 17. Connection Wires.
- 18. Digital Logic Probe.
- 19. DC supply (0 and +5V).

THEORY:

Boolean Algebra is the mathematics which is used to analyze and simplify the digital logic circuits. It uses only the binary numbers. i.e. 0 and 1. It is also called as Binary Algebra or Logical Algebra. On comparison with the Elementary Algebra where the main operations are Addition and Multiplication, the main operations of Boolean Algebra are AND, OR and NOT. When the number of gates are increased in the circuit, it is difficult to manage the connections, this becomes a challenge. Boolean Algebra provides certain rules to simplify the logic equation in order to reduce the number of gates to the least possible by keeping the circuit overall same functionality. Boolean Algebra was introduced by George Boole in his first book "*The Mathematical Analysis of Logic (1847)*". The laws of Boolean Algebra are as shown in Table 3.1:

Boolean Expression	Description	Equivalent Switching Circuit	Boolean Algebra Law or Rule
$A + 1 = 1$	A in parallel with closed = "CLOSED"		Annulment
$A + 0 = A$	A in parallel with open = "A"		Identity
$A \cdot 1 = A$	A in series with closed = "A"		Identity

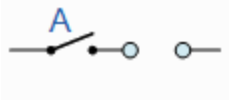
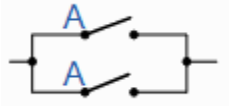
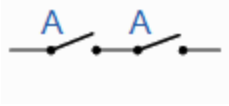
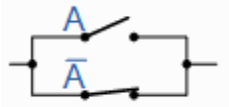
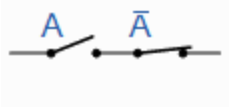
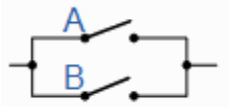
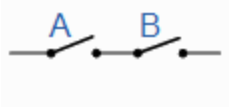
$A \cdot 0 = 0$	A in series with open = "OPEN"		Annulment
$A + A = A$	A in parallel with A = "A"		Idempotent
$A \cdot A = A$	A in series with A = "A"		Idempotent
$\text{NOT } \bar{A} = A$	NOT A (double negative) = "A"		Double Negation
$\bar{A} + A = 1$	A in parallel with NOT A = "CLOSED"		Complement
$A \cdot \bar{A} = 0$	A in series with NOT A = "OPEN"		Complement
$A + B = B + A$	A in parallel with B = B in parallel with A		Commutative
$A \cdot B = B \cdot A$	A in series with B = B in series with A		Commutative
$\overline{A + B} = \bar{A} \cdot \bar{B}$	invert and replace OR with AND		de Morgan's Theorem
$\overline{A \cdot B} = \bar{A} + \bar{B}$	invert and replace AND with OR		de Morgan's Theorem

TABLE 3.1: Boolean Laws

Associative property of addition:

$$A + (B + C) = (A + B) + C$$

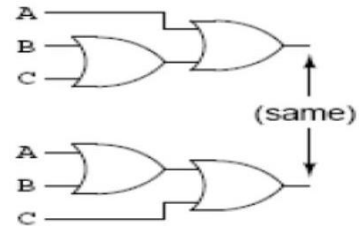


Figure 3.1 Circuit of Associate Property w.r.t. "Addition"

Associative property of Multiplication:

$$A \bullet (B \bullet C) = (A \bullet B) \bullet C$$

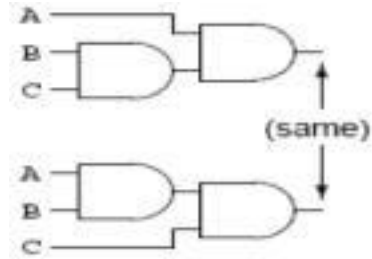
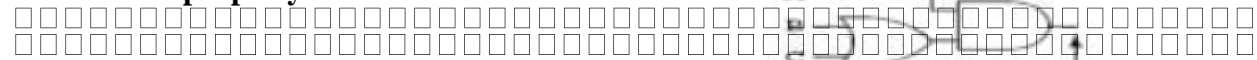


Figure. 3.2: Circuit of Associate Property w.r.t. "Multiplication"

Distributive property:



$$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$$

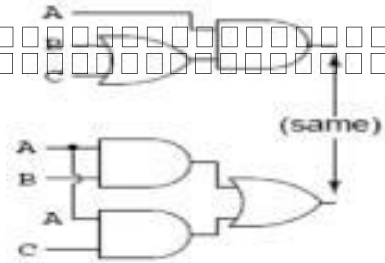


Figure. 3.3: Circuit depicting Distributive Proper

Logic Simplification:

1. Example # 01:

$$\begin{aligned}
 F &= (A + B) \cdot (A + C) \\
 &= A \cdot A + A \cdot C + A \cdot B + B \cdot C \\
 &= A + A \cdot C + A \cdot B + B \cdot C \\
 &= A (1 + C) + A \cdot B + B \cdot C \\
 &= A \cdot 1 + A \cdot B + B \cdot C \\
 &= A + A \cdot B + B \cdot C \\
 &= A (1 + B) + B \cdot C \\
 &= A \cdot 1 + B \cdot C \\
 &= A + B \cdot C
 \end{aligned}$$

Distributive Law
 Idempotent AND Law
 Distributive Law
 Identity OR Law
 Identity AND Law
 Distributive Law
 Identity OR Law
 Identity AND Law

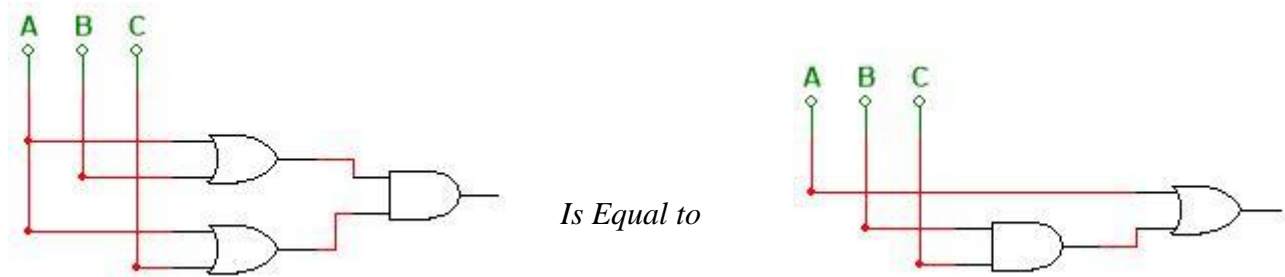


Fig. 3.4: Circuit Diagrams depicting Same Logic after Boolean Simplification – Example 01

2. Example # 02:

$$\begin{aligned}
 F &= A \cdot B + B \cdot C \cdot (B + C) \\
 &= A \cdot B + B \cdot C \cdot B + B \cdot C \cdot C \\
 &= A \cdot B + B \cdot B \cdot C + B \cdot C \cdot C \\
 &= A \cdot B + B \cdot C + B \cdot C \\
 &= A \cdot B + B \cdot C \\
 &= B \cdot (A + C)
 \end{aligned}$$

Distributive Law
 Commutative Law
 Idempotent Law
 Idempotent Law
 Distributive Law

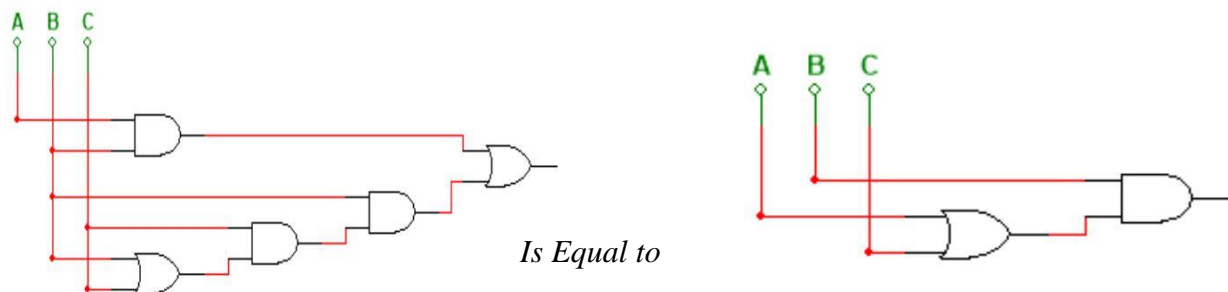


Fig. 3.5: Circuit Diagrams depicting Same Logic after Boolean Simplification – Example 02

3. Example # 03:

$$\begin{aligned}
 F &= (\bar{A})(A + B) + (B + A)(A + \bar{B}) \\
 &= (\bar{A})A + (\bar{A})B + (B + A)A + (B + A)\bar{B} \\
 &= (\bar{A})B + (B + A)A + (B + A)\bar{B} \\
 &= (\bar{A})B + BA + AA + B\bar{B} + A\bar{B} \\
 &= (\bar{A})B + BA + A + A\bar{B} \\
 &= B(\bar{A} + A) + A(1 + \bar{B}) \\
 &= B.1 + A.1 \\
 &= B + A \\
 &= A + B
 \end{aligned}$$

Distributive Law
 Complement Law
 Distributive Law
 Idempotent Law
 Distributive Law
 Idempotent & Annulment Law
 Identity
 Commutative Law

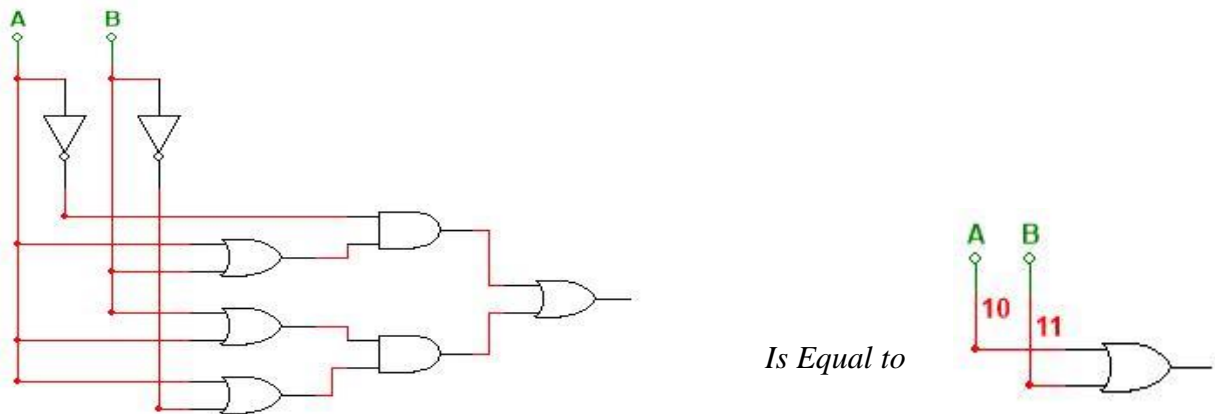


Fig. 3.6: Circuit Diagrams depicting Same Logic after Boolean Simplification – Example 03

Morgan's Theorem:

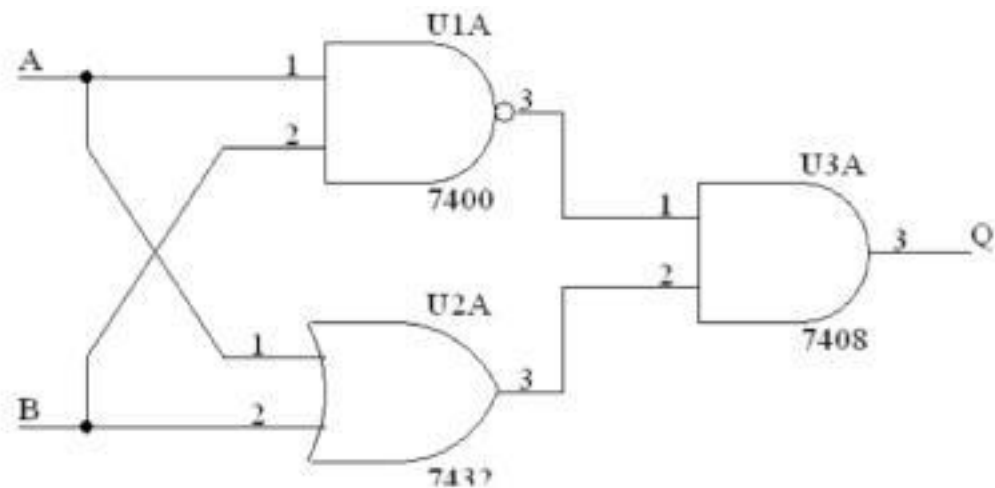
De Morgan developed a theorem that allows conversion between logic expressions that has inversions on the output to a different logic expression with the inversions on each of the inputs. This may allow for the simplification of a Boolean Expression by the cancellation of some redundant inversions. There are two Boolean Equations that represent De Morgan's Theorem:

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

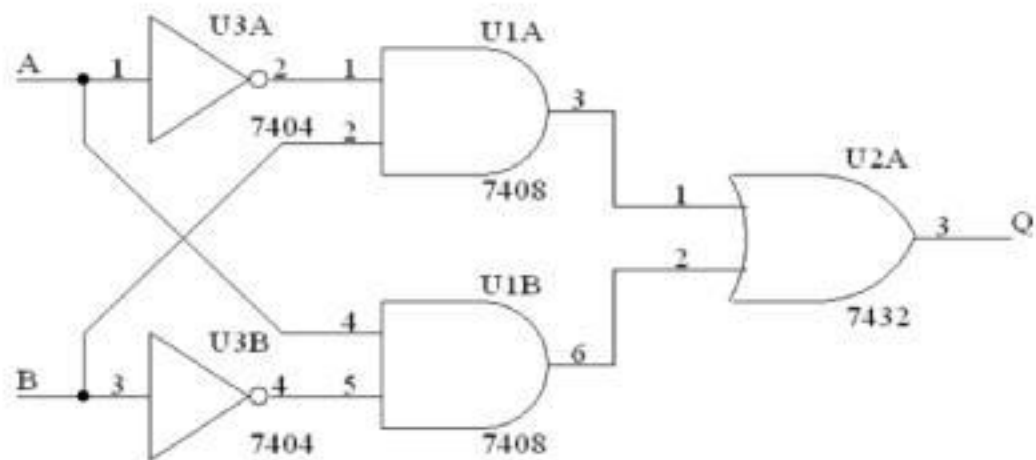
Or

$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

4. Example (A) – Circuit Diagram:-



(a)



(b)

Fig. 3.7: Circuit Diagram (a) and Circuit Diagram (b) depicts Same Logic (De – Morgan's Law)

PROCEDURE:

1. At first construct the circuits represented by the original and simplified equations of each example.
2. Construct the Truth Tables of both the original and the simplified equation in each case.
3. Check if the two circuits give the same result.

OBSERVATION / RESULTS and DISCUSSION:

9. Do the original and simplified equation shows same functionality in each case (Example 01, Example 02 and Example 03)? _____.
10. Do the circuits of Fig. 3.7(a) and Fig. 3.7(b) gives the similar result? _____.
11. The logical expression for the Fig. 3.7(a) is _____.
12. Manipulate the equation obtained in observation (3) to reduce the equation that represents the circuit diagram of Fig. 3.7(b)?

CONCLUSION:

3. A circuit can easily represent in the form equation
4. It is easy to simplify the circuit using Boolean algebra.
5. After Simplification Hardware Implementation is quite easy.
6. Truth table can easily be converted to Boolean expression and vice-versa.
7. Reduces the complexity.

EXERCISES:

1. Simplify the following equation using Boolean Laws. Construct the Truth Tables to verify that the simplified equation gives the same result as that of the original equation.

$$F = (A + C)(AD + \overline{A}D) + \overline{\overline{A}C} + C$$

LAB #12

Karnaugh Map or K-MAP Minimization

OBJECTIVE:

- (a) The aim of this lab is to minimize the digital circuit using K-Map
- (b) To implement the reduced circuit on hardware / software.

EQUIPMENT:

- 20. IC: 7408LS, 7432LS, 7404LS, 7400LS and 7402LS.
- 21. Bread board.
- 22. Connection Wires.
- 23. Digital Logic Probe.
- 24. DC supply (0 and +5V).

THEORY:

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, when it is expressed algebraically it can appear in many different, but equivalent, forms. Boolean expressions may be simplified by Boolean rules. However, this procedure of minimization is awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the Karnaugh map or K-map.

Logic Implementation:

Example # 01:

Implement the following function

$$F_1 = ABC + AB\bar{C} + A\bar{B}C$$

On the breadboard and fill in the truth table F_1 by applying all possible inputs.

TABLE 4.1: Observation Table – Example 01

Input: A	Input: B	Input: C	F ₁	F ₂
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

1. Simplify the above Boolean equation (F₁) using K-Map – Find Simplified Expression F₂?
2. Simplify the above Boolean equation (F₁) using the multisim instrument “LOGIC CONVERTER”.

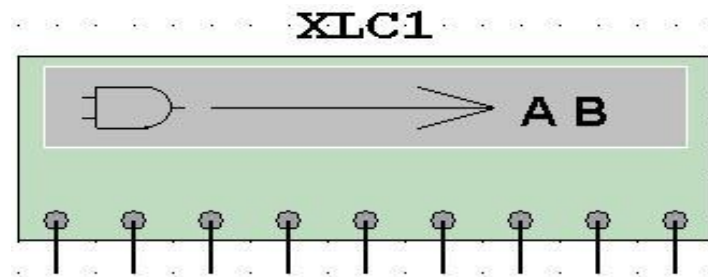


Figure 4.1: MultiSim Logic Simplification Tool – Logic Converter

3. Implement the simplified expression on the bread board and fill the in above truth table F₂?

Example # 02:

Consider the Boolean Function as,

$$F(x, y, z) = \sum (1, 2, 4, 5, 6)$$

On simplification using K-Map, the simplified equation is,

$$F = A\bar{B} + \bar{B}C + B\bar{C}$$

Verify both equations by Truth Table and by designing Circuit on Hardware.

Example # 03:

Consider the Boolean Function as,

$$Y = FFFE \quad (\text{Hex})$$

On simplification using K –Map, the simplified equation is,

$$Y = A + B + C + D$$

Verify both equations by Truth Table and by designing Circuit on Hardware.

Example # 04:

Consider the Boolean Function as,

$$F (x, y, z) = \sum (0,1,4,5)$$

On simplification using K –Map, the simplified equation is,

$$F = \bar{B}$$

Verify both equations by Truth Table and by designing Circuit on Hardware

CONCLUSION:

3. It is easy to simplify the circuit using K-Map.
4. After Simplification Hardware implantation is quite easy.
5. Reduces the complexity.

QUESTIONS:

9. Simplify the following function using K-Map in

Sum of Product. (SOP)

Product of sum. (POS)

$$F (A, B, C, D) = \sum(0, 1, 2, 5, 8, 9, 10)$$

Verify Boolean function ‘F’ before and after K-Map simplification using truth table and design circuit on hardware / software.

10. Simplify the Boolean Function using K-Map

$$F (A, B, C, D) = \sum(1, 3, 7, 11, 15)$$

Which has the don’t care conditions

$$D (A, B, C, D) = \sum(0, 2, 5)$$

Verify Boolean function ‘F’ before and after K-Map simplification using truth table and

design circuit on hardware / software.

11. Simplify the Boolean Function using K-Map

$$F = E8 \quad (\text{Hex})$$

Verify Boolean function 'F' before and after K-Map simplification using truth table and design circuit on hardware / software.

LAB 13

Binary and Decade Counters (IC 7490) Circuits

OBJECTIVE:

- (a) The objective of this experiment is to study the operating conditions of Binary and Decade Counter IC 7490 and designing MOD 10, MOD 60 and MOD 100, MOD 200 and MOD 1000 counters by using Seven Segment Display.

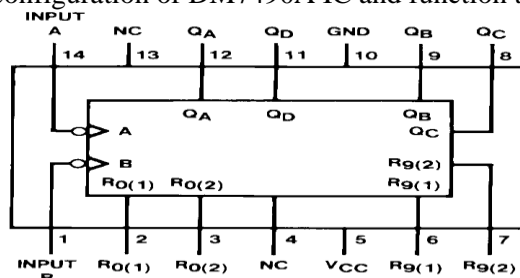
EQUIPMENT:

1. IC 74LS47, IC DM7490A, IC 74LS08.
2. Bread Board.
3. Digital Logic Probe.
4. DC Supply (+5V) and Seven Segment Display.
5. Few LEDs and Resistors.

THEORY:

A decade counter is one that counts in decimal digits, rather than binary. A decade counter may have each digit binary encoded (that is, it may count in binary-coded decimal, as the 7490 integrated circuit did) or other binary encodings (such as the bi-quinary encoding of the 7490 integrated circuit). Alternatively, it may have a "fully decoded" or one-hot output code in which each output goes high in turn (the 4017 is such a circuit). The latter type of circuit finds applications in multiplexers and demultiplexers, or wherever a scanning type of behavior is useful. Similar counters with different numbers of outputs are also common. The decade counter is also known as a mod-counter.

The DM7490A monolithic counter contains four master slave flip-flops and additional gating to provide a divide-by two counter and a three-stage binary counter for which the count cycle length is divide-by-five. The counter has a gated zero reset and also has gated set to- nine inputs for use in BCD nine's complement applications. To use the maximum count length (decade or four-bit binary), the B input is connected to the QA output. The input count pulses are applied to input A and the outputs are as described in the appropriate Function Table. A symmetrical divide-by-ten count can be obtained from the counters by connecting the QD output to the A input and applying the input count to the B input which gives a divide-by-ten square wave at output QA. The pin configuration of DM7490A IC and function tables is as shown in Figure 11.1.



Function Tables

BCD Count Sequence (Note 1)

Count	Outputs			
	Q _D	Q _C	Q _B	Q _A
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	L	H	L	H
6	L	H	H	L
7	L	H	H	H
8	H	L	L	L
9	H	L	L	H

BCD Bi-Quinary (5-2) (Note 2)

Count	Outputs			
	Q _A	Q _D	Q _C	Q _B
0	L	L	L	L
1	L	L	L	H
2	L	L	H	L
3	L	L	H	H
4	L	H	L	L
5	H	L	L	L
6	H	L	L	H
7	H	L	H	L
8	H	L	H	H
9	H	H	L	L

Reset/Count Function Table

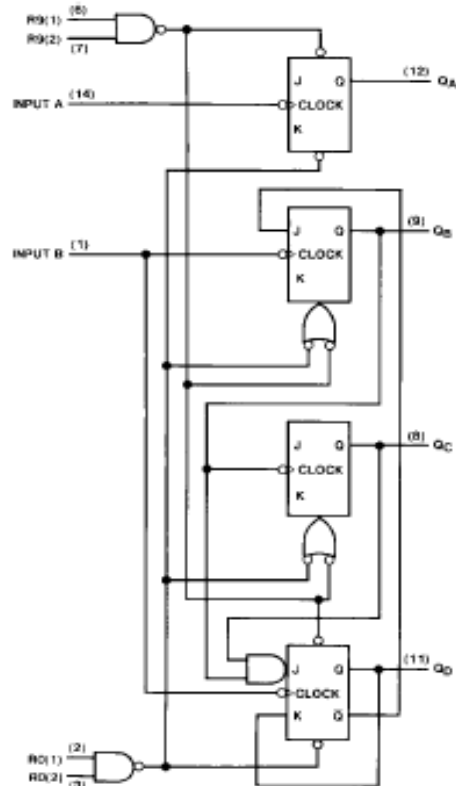
Reset Inputs				Outputs			
R0(1)	R0(2)	R3(1)	R3(2)	Q _D	Q _C	Q _B	Q _A
H	H	L	X	L	L	L	L
H	H	X	L	L	L	L	L
X	X	H	H	H	L	L	H
X	L	X	L	COUNT			
L	X	L	X	COUNT			
L	X	X	L	COUNT			
X	L	L	X	COUNT			

H = HIGH Level
L = LOW Level
X = Don't Care

Note 1: Output Q_A is connected to input B for BCD count.

Note 2: Output Q_D is connected to input A for bi-quinary count.

Logic Diagram



The J and K inputs shown without connection are for reference only and are functionally at a HIGH level.

Figure 11.1: IC DM7490A Pin Configuration and Function Tables

Decade Counter (Circuit Diagram):

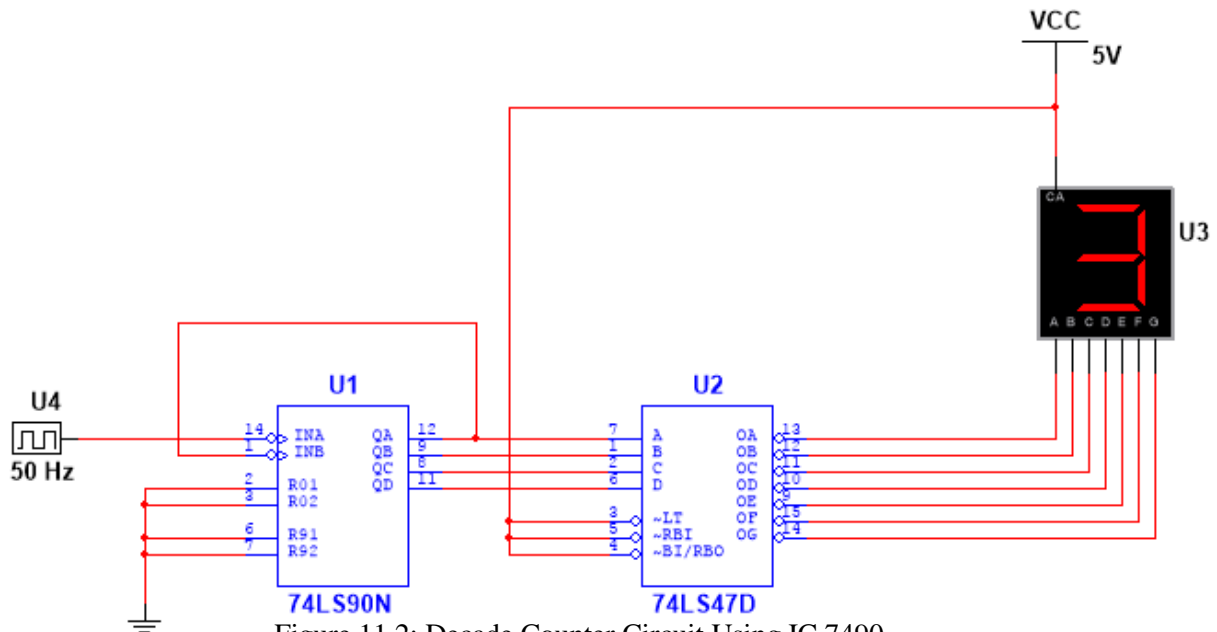


Figure 11.2: Decade Counter Circuit Using IC 7490

MOD 100 Counter (Circuit Diagram):

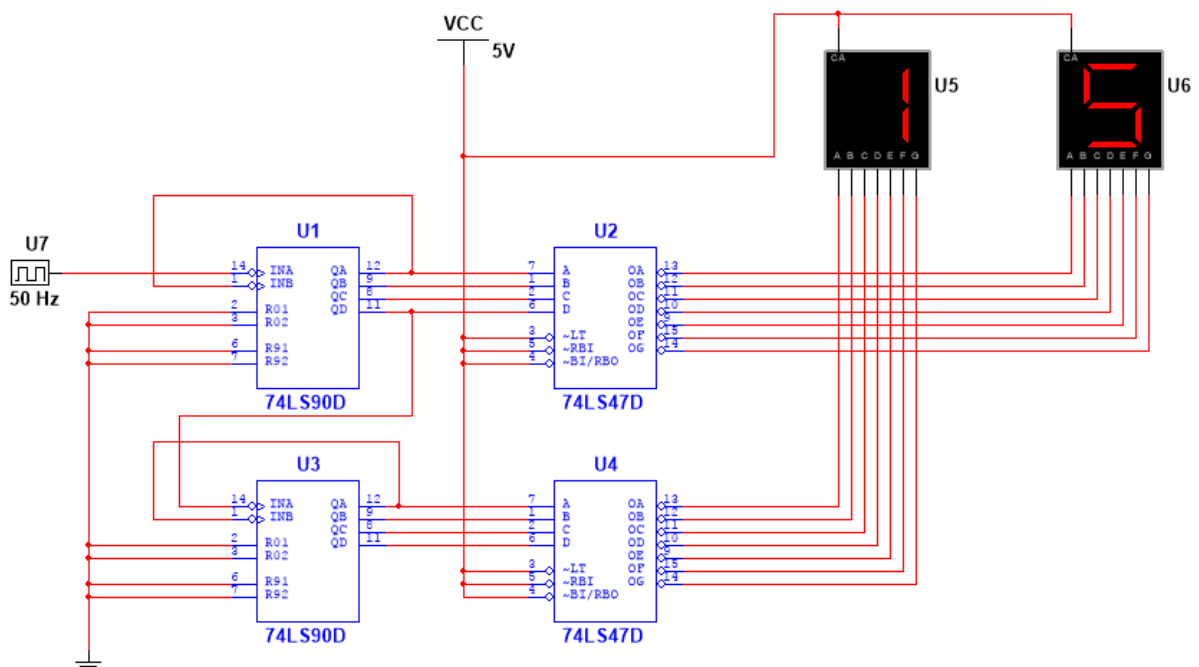


Figure 11.3: MOD 100 (0-99) Counter Circuit Using IC 7490

PROCEDURE:

1. Construct the circuits as shown in Figure 11.2 and Figure 11.3.
2. Then connect the BCD to seven segment decoder to display the count from 0 - 9.
3. Verify the function table of DM7490A as shown in Figure 1.

QUESTIONS:

1. Design the MOD 1000 (0-999) Counter using IC DM7490A and Seven Segment Display.
2. Design the MOD 60 (0-59) and MOD 200 (0 – 199) Counter using IC DM7490A and Seven Segment Display.
3. What does the pins $R0(1)$, $R0(2)$, $R9(1)$ and $R9(2)$ of IC DM7490A specify?

$R0(1)$: _____

$R0(2)$: _____

$R9(1)$: _____

$R9(2)$: _____

LAB 14

EXPOSURE OF DIFFERENT FLIP FLOPS

EQUIPMENT: -

1. IC 7400, IC 7404, IC 7476.
2. NI Multisim

THEORY: -

SR flip-flop is also called Synchronous flip-flop. That means that this flip-flop is concerned with time. Digital circuits can have a concept of time using a clock signal. The clock signal simply goes from low-to-high and high-to-low in a short period of time.

C	S	R	Next Stage of Q
0	x	x	No change
1	0	0	No change
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; Set State
1	1	1	Indeterminate

Figure 1: SR Flip Flop and its truth table

In case of D flip-flop, The Q output always takes on the state of the D input at the moment of a rising clock edge. (Or falling edge if the clock input is active low). It is called the D flip-flop for this reason, since the output takes the value of the D input or Data input, and Delays it by one clock count. The D flip-flop can be interpreted as a primitive memory cell, zero-order hold, or delay line.

C	D	Next Stage of Q
0	x	No change
1	0	Q = 0; Reset state
0	1	Q = 1; Set State

Figure 1: D Flip Flop and its truth table

In case of T flip-flop, if the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobe. If the T input is low, the flip-flop holds the previous value. The truth table is as follows:

Characteristics Table				Excitation Table			
T	Q	Q _{next}	Comment	Q	Q _{next}	T	comment
0	0	0	Hold on state (no clock)	0	0	0	No change
0	1	1	Hold on state (no clock)	1	1	0	No change
1	0	1	toggle	0	1	1	Complement
1	1	0	toggle	1	0	1	Complement

The JK flip-flop augments the behavior of the SR flip-flop (J=Set, K=Reset) by interpreting the S = R = 1 condition as a "flip" or toggle command. Specifically, the combination J = 1, K = 0 is a command to set the flip-flop; the combination J = 0, K = 1 is a command to reset the flip-flop; and the combination J = K = 1 is a command to toggle the flip-flop, i.e., change its output to the logical current complement of its value.

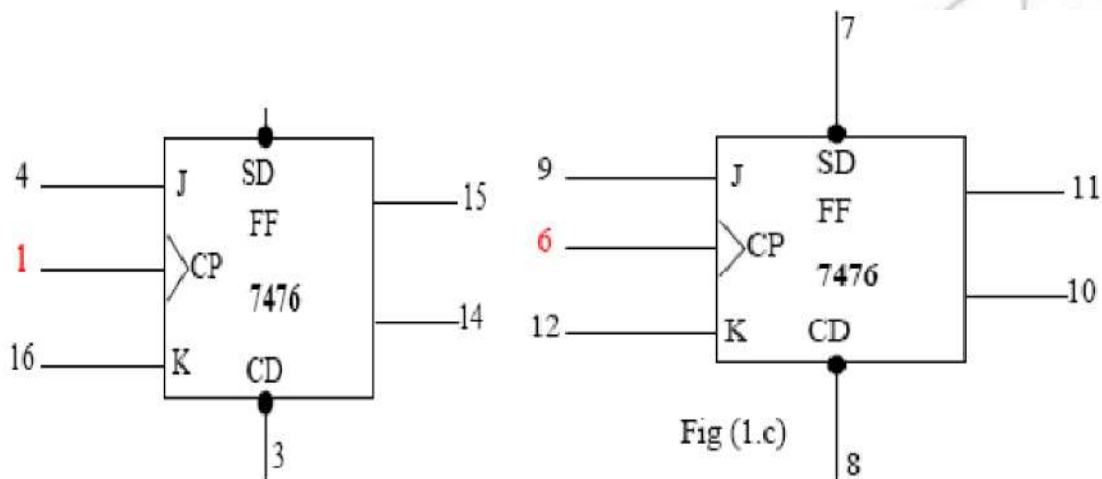


Fig (1.c)

Figure: IC 7476 M/S JK flip-flop

SD Preset	SD Clear	Clock	J	K	OUTPUST	
					Q	\bar{Q}
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H*	H*
H	H	↓	L	L	Q _o	\bar{Q}_o
H	H	↓	H	L	H	L
H	H	↓	L	H	L	H
H	H	↓	H	H	TOGGLE	
H	H	H	X	X	Q _o	\bar{Q}_o

Figure: Truth table for JK flip-flop (IC 7476)

PROCEDURE:

- Make connections as shown in the diagrams.
- Verify the truth tables for various combinations of inputs.

QUESTIONS:

1. What is the difference between Flip-Flop and latch?
2. What is the difference between synchronous and asynchronous inputs?
3. What are the applications of different Flip-Flops?
4. What is the difference of Edge triggering over level triggering?