LAB # 1

**INTRODUCTION TO VVM**

Introduction to Computer Architecture – Lab Component

**OBJECTIVE**

Explore Visible Virtual Machine (VVM).

**TIME BOXING**

|  |  |  |
| --- | --- | --- |
| Activity Name | Activity Time | Total Time |
| Instruments Allocation + Setting up Lab | 10 mints | 10 mints |
| Walk through Theory & Tasks (Lecture) | 60 mints | 60 mints |
| Implementation & Practice time | 90 mints | 80 mints |
| Evaluation Time | 20 mints | 20 mints |
|  | Total Duration | 180 mints |

**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 mousedriven speed control. The program can be paused and subsequently resumed at any point, at the discretion of the user. Alternatively, the user can choose to step through the program one statement at a time. As each program instruction is executed, all relevant hardware components (e.g., internal registers, RAM locations, output devices, etc.) are updated in full view of the user.

# Hardware Components

The VVM machine comprises the following hardware components:

* **I/O Log**. This represents the system console which shows the details of relevant events in the execution of the program. Examples of events are the program begins, the program aborts, or input or output is generated.

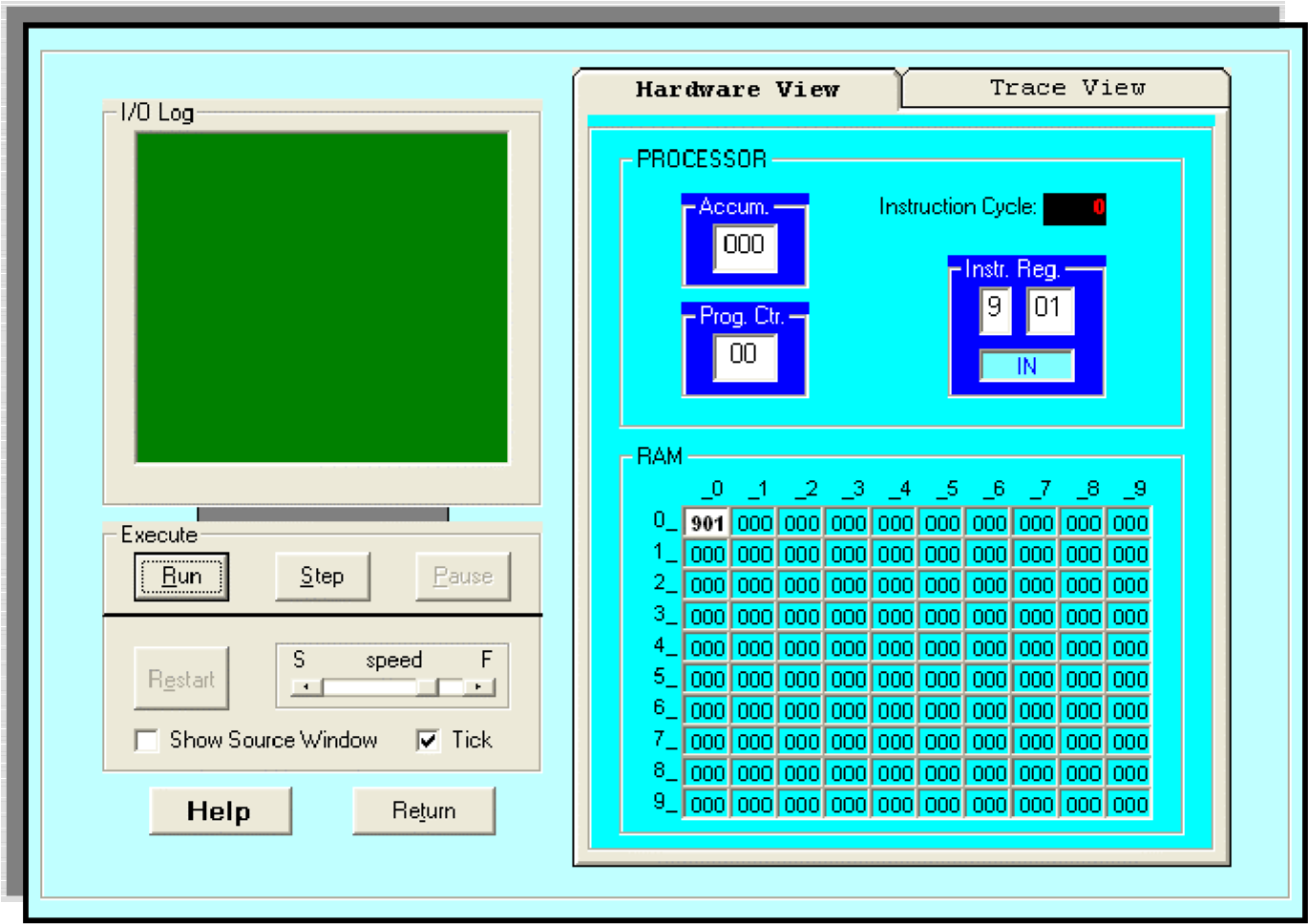
* **Accumulator Register** (Accum). This register holds the values used in arithmetic and logical computations. It also serves as a buffer between input/output and memory. Legitimate values are any integer between -999 and +999. Values outside of this range will cause a fatal VVM Machine error. Non-integer values are converted to integers before being loaded into the register.

* **Instruction Cycle Display**. This shows the number of instructions that have been executed since the current program execution began.

* **Instruction Register** (Instr. Reg.). This register holds the next instruction to be executed. The register is divided into two parts: a one-digit *operation code*, and a two-digit *operand*. The Assembly Language mnemonic code for the operation code is displayed below the register.

* **Program Counter Register** (Prog. Ctr.). The two-digit integer value in this register “points” to the next instruction to be fetched from RAM. Most instructions increment this register during the *execute* phase of the instruction cycle. Legitimate values range from 00 to 99. A value beyond this range causes a fatal VVM Machine error.

* **RAM**. The 100 *data-word* Random Access Storage is shown as a matrix of ten rows and ten columns. The two-digit memory addresses increase sequentially across the rows and run from 00 to 99. Each storage location can hold a three-digit integer value between -999 and +999.



**Data and Addresses**

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

# Trace View

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

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

# VVM System Errors

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

* **Data value out of range**. This condition occurs when a data value exceeds the legitimate range -999 - +999. The condition will be detected while the data resides in the *Accumulator Register*. Probable causes are an improper addition or subtraction operation, or invalid user input.

* **Undefined instruction**. This occurs when the machine attempts to execute a three-digit value in the *Instruction Register* which can not be interpreted as a valid instruction code. See the help topic “VVM Language” for valid instruction codes and their meaning. Probable causes of this error are attempting to use a data value as an instruction, an improper *Branch* instruction, or failure to provide a *Halt* instruction in your program.

* **Program counter out of range**. This occurs when the Program Counter Register is incremented beyond the limit of 99. The likely cause is failure to include a *Halt* instruction in your program, or a branch to a high memory address.

* **User cancel**. The user pressed the “Cancel” button during an *Input* or *Output* operation.

# VVM Program Example 1

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

// A sample VVM Assembly program

// to add a number to the value -1.

IN Input number to be added

ADD 99 Add value stored at address 99 to input

OUT Output result

HLT Halt (program ends here)

\*99 Next value loaded at address 99

DAT -001 Data value

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

// The Machine Language version

901 Input number to be added

199 Add value stored at address 99 to input

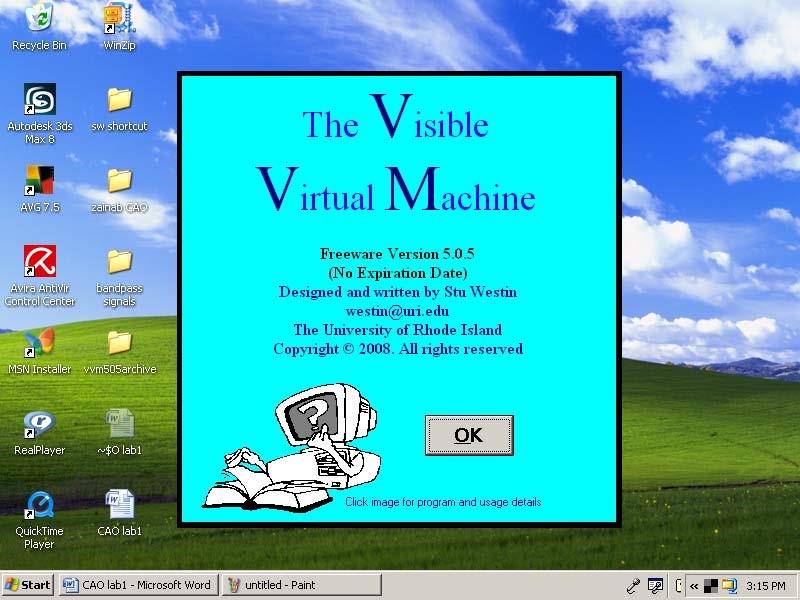
902 Output result

000 Halt (program ends here)

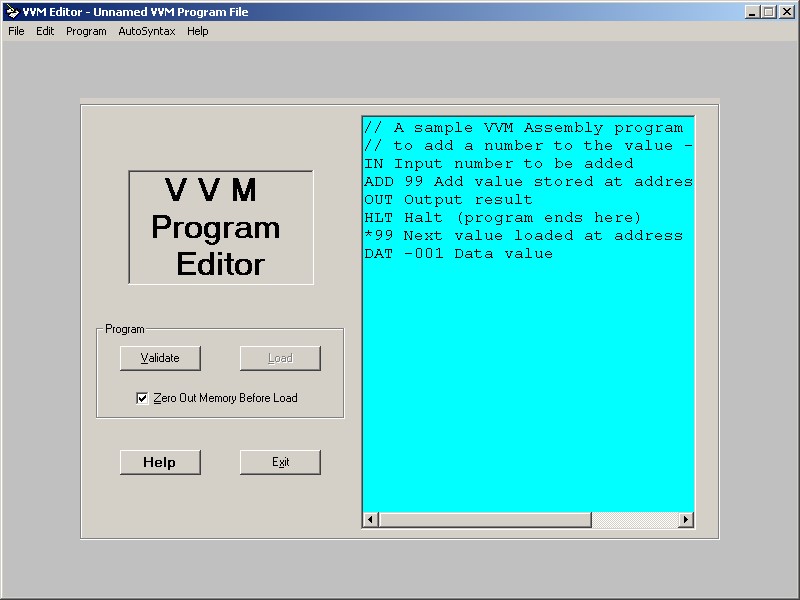
\*99 Next value loaded at address 99

-001 Data value

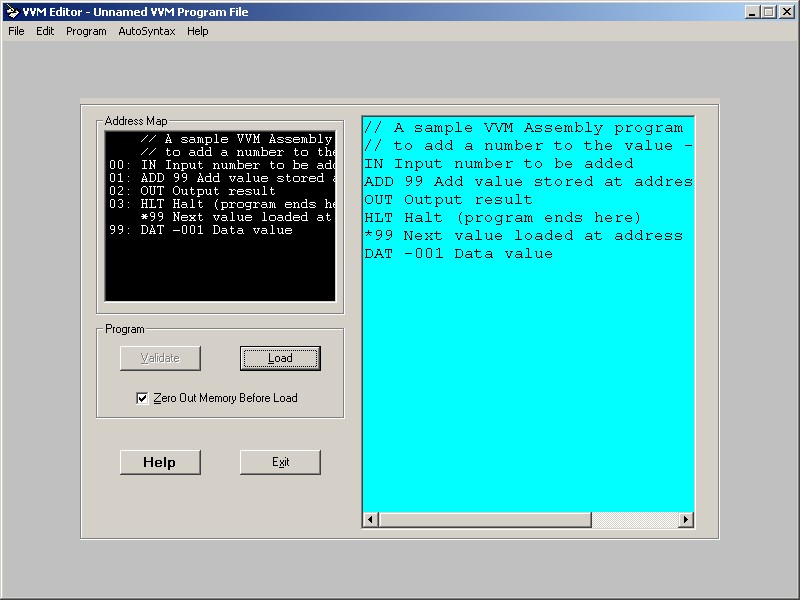
## STEP1: Load VVM



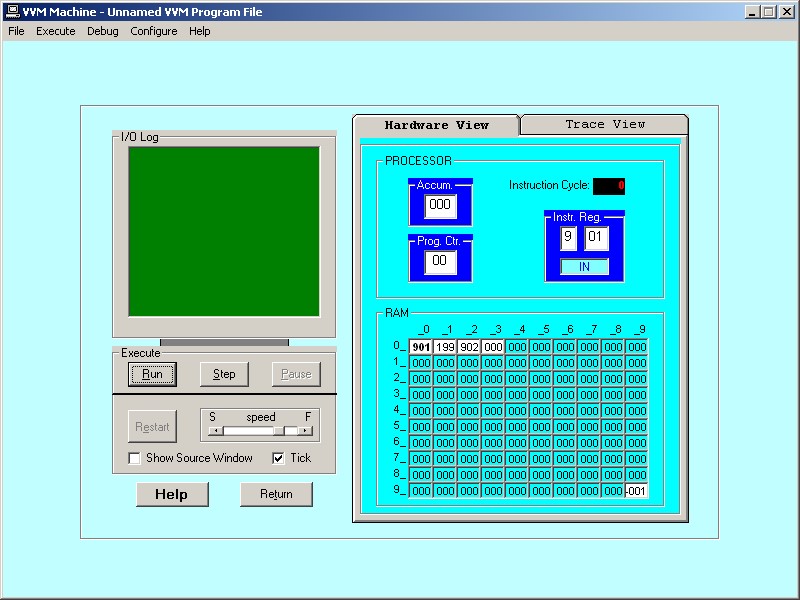
## STEP2: Copy the code and paste in the blue editor window



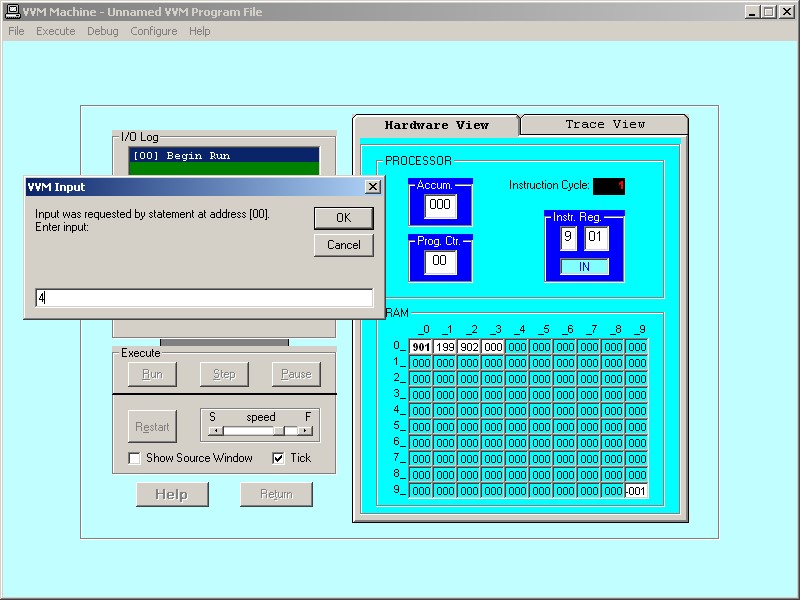
## STEP3: Press the Validate button

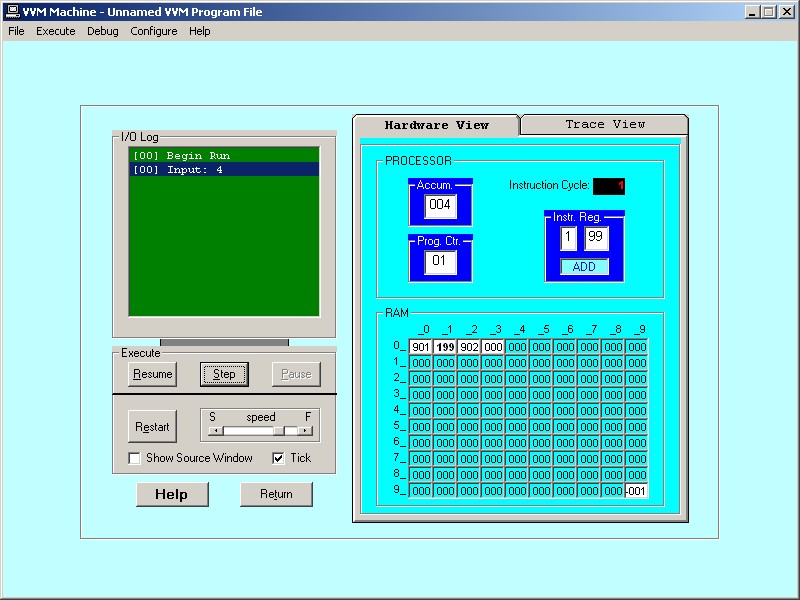


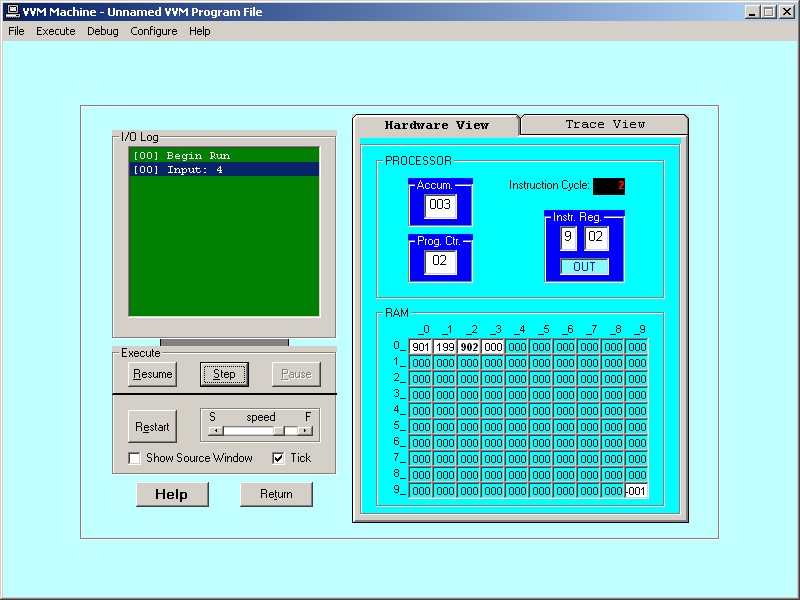
## STEP4: Press the Load button

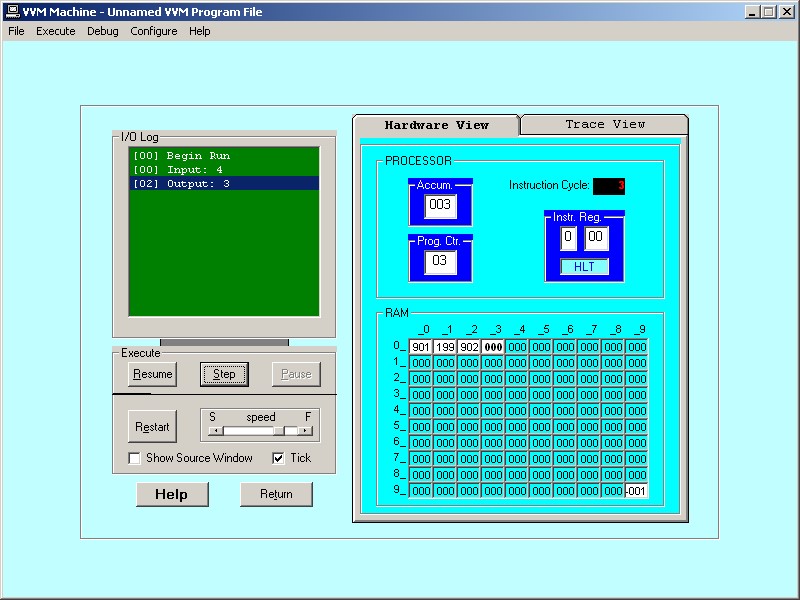


## STEP5: Press the Step button



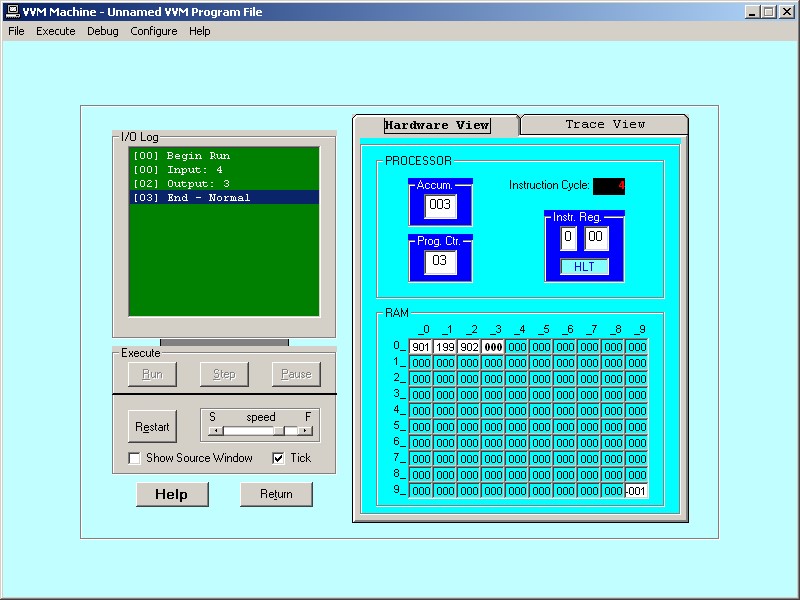




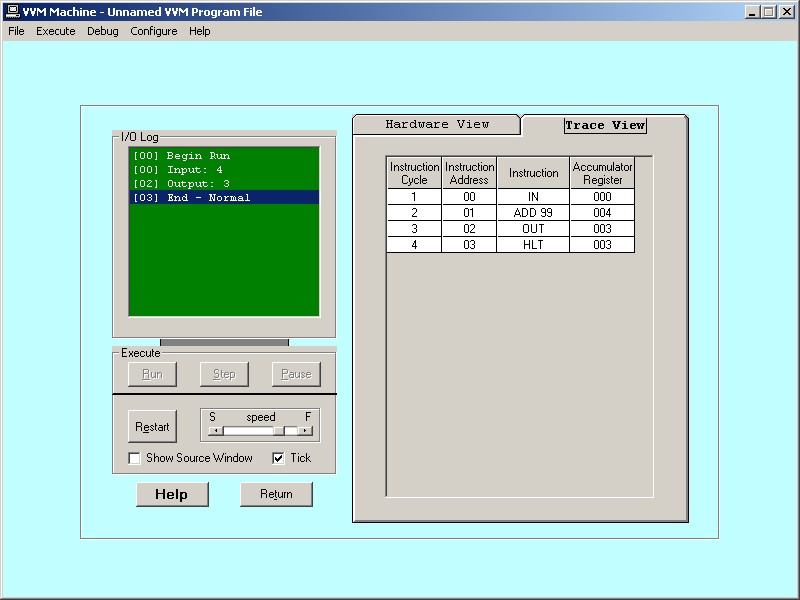


# OUTPUT

**Hardware View**



**Trace View**



# LAB TASK

1. Take two inputs and Subtract.

in

sto 98

in

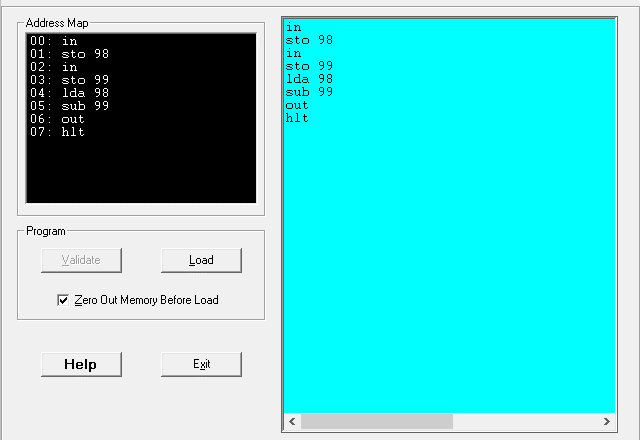
sto 99

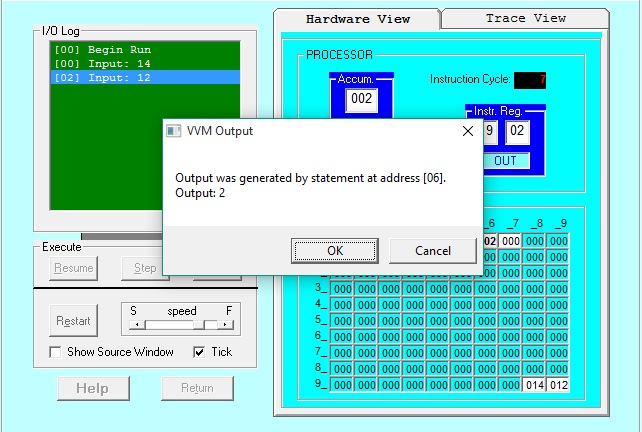
lda 98

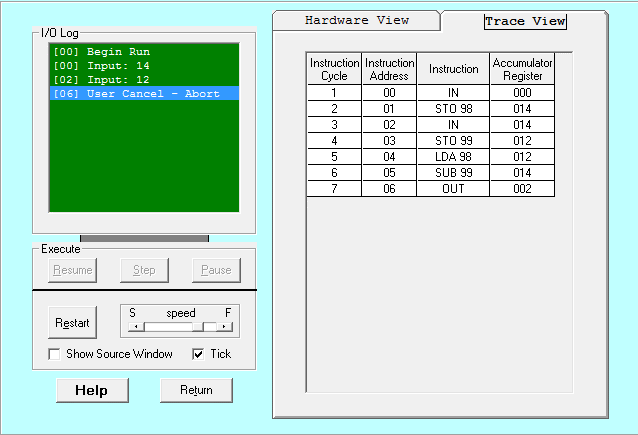
sub 99

out

hlt







1. To take two hardcoded input and Add them.

lda 98

add 99

out

hlt

\*98

dat 010

dat 015

