The Pennsylvania State University
The Graduate School
Capital College

MarieSim: A Simulator for the MARIE Architecture

A Master's Paper in
Computer Science
By
Julia M. Lobur

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Master of Science

April 4, 2003

Abstract

The purpose of this project is to provide computer architecture students with an interactive simulator to deepen their understanding of the workings of a simple computer, MARIE, which is an acronym for a Machine Architecture that is Really Intuitive and Easy. The simulator environment, MarieSim, presents this machine architecture in a detailed, yet visually attractive manner. Through interaction with its graphical environment, students can observe how assembly language statements affect the registers and memory of a computer system. The graphical environment for MarieSim is written in Java Swing. The integrated MARIE assembler is written in Java.

Table of Contents

Acknowledgements

The architecture of the MARIE computer is an outgrowth of Dr. Linda Null's considerable experience in teaching computer organization and architecture to many computer science students. It is also a direct result of her passion for making her lessons clear and accessible. This project would not have been possible without her efforts and guidance, for which I am truly grateful.

I am also grateful for the careful review of this work by the members of the committee: Dr. Thang Bui, Dr. Pavel Naumov, and Dr. Qin Ding.

List of Figures

1 Introduction

Computer users and computer software designers have seemingly insatiable desires for improved features, easier interface usability, and faster execution speeds. Thus, to keep pace with market expectations, microcomputer manufacturers continually increase the power of their systems. In many ways, these systems have surpassed the complexity of yesterday's million dollar mainframe systems. While the exponential growth of computing power has been a boon for both the computer user and the computer manufacturer, this trend has virtually eliminated the possibility of using these systems as pedagogical tools. To do so is to run the risk of students becoming mired in a plethora of proprietary details that serve only to obscure their understanding of the essential functions of computer systems.

While learning the intimate details of today's systems is undoubtedly a worthy endeavor, the complexity of these machines can easily overwhelm the beginning architecture student, particularly if he or she does not first possess a firm understanding of the basics of the von Neumann architecture. The Machine Architecture that is Really Intuitive and Easy, MARIE, was conceived solely to provide this basic understanding. It is an unadorned, yet fully functional von Neumann system complete with an uncomplicated instruction set architecture and a simple register transfer language.

The MARIE architecture is articulated in *The Essentials of Computer Organization and Architecture* [1], an introductory textbook for computer science students. Any textbook description of a machine architecture is necessarily static and two-dimensional. While students may grasp the functions of this machine on an abstract intellectual level, their complete understanding of a system comes only through interacting with it. The MARIE simulator, MarieSim, provides the opportunity for this interaction.

MarieSim is a graphical learning environment that illuminates the operation of the MARIE machine architecture. Within this environment students can: (1) create and edit MARIE assembly language programs; (2) assemble source code into machine object

code; (3) run machine code programs; and (4) observe and debug their programs using various tools provided within the simulator.

MarieSim is not unique in its simplicity or in its animated execution environment. There are many such software simulators, each with its own particular focus. (A survey and taxonomy can be found in [2].) Some simulation systems model obsolete computers, such as the PDP-series computers formerly sold by the Digital Electronics Corporation. Others provide exposure to highly sophisticated (and otherwise quite expensive) computers such as multiprocessor systems and quantum computers [3]. Owing to the plethora of freely available simulators, an educator desiring to incorporate one of these tools into the classroom face a daunting task. Clearly, one must keep in mind the appropriateness of the simulator for the course content and expected sophistication of the students. Thus, a simulator geared toward high-level language compilers would not be the best choice for an introductory computer architecture course. Also, a simulator should provide a gentle learning curve so that students won't become frustrated or spend more time learning simulator operations than the concepts that the simulator is designed to impart.

In this paper, we describe a simulator that provides an animated, graphical visualization of a classical von Neumann systems as presented in a first course in computer organization and architecture for computer science majors. There are many delightful simulators that fall out of this narrow scope, such as those suited to high school students [4] [5] [6], non-computer science majors [7], and more advanced treatments [6] [8]. Among the dozens of graphical simulator environments appropriate for introductory computer architecture students, we examined three that are popular within the computer architecture education community. These simulators are Pep/7, the xComputer, and Ant.

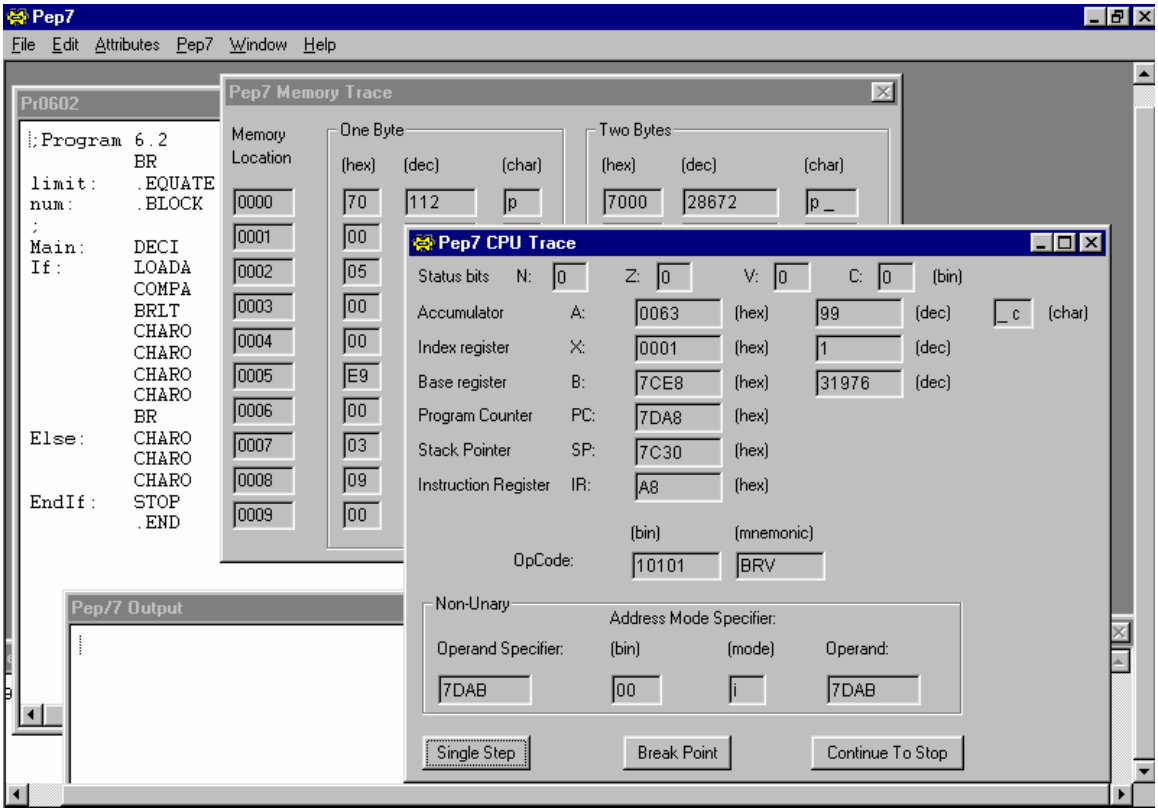


Figure 1: The Pep/7 Simulator

- Pep/7

The Pep/7 simulator, shown in Figure 1, was written as supporting material for J. Stanley Warford's text, *Computer Systems* [9]. This book takes a "top down" approach to computer architecture, starting at the application level, and descending through the computer level hierarchy to the gate level. Because of this, much of Warford's focus is on system software and assembly language programming. The design of Pep/7 is consistent with Warford's pedagogical objectives. The simulator is mature software, incorporating many features that are helpful to the student:

- ◊ A student can step through program, one instruction at a time.
- ◊ Breakpoints can be set in the code, thus allowing fast execution to the location of the instruction of interest.

- ◊ The simulator incorporates a full-featured text processor that includes text emphasis features such typeface coloring, italicizing, bolding, and font selection.
- ◊ Register values are shown in hexadecimal, with binary and character equivalents given for some (not all) registers.
- ◊ The assembly listing is realistic, giving hexadecimal translations of the program instructions on the same line as the mnemonic instruction.
- ◊ A symbol table (without cross-references) is provided along with the assembler listing.
- ◊ Printing facilities are well integrated into the simulator.
- ◊ “Friendly” features include an intuitive interface, and confirmation prompts upon issuance of a program termination command.

Pep/7 is arguably not the best teaching tool for presenting a detailed examination of a von Neumann architecture. In particular:

- ◊ The internal machine operation is viewable only in trace mode.
- ◊ Trace mode provides only a single-instruction "step mode" and a run mode, with no provision for running a program automatically in "slow motion."
- ◊ It is difficult to establish a correlation between source statements and machine events, because the symbolic instructions are presented in a separate window that is encroached upon by the system monitor.
- ◊ To examine the contents of memory, the user must enumerate the addresses to be traced prior to running the program, and only ten addresses can be specified. The difficulty with this is that some beginning students may not know which addresses will be accessed prior to initial program execution.

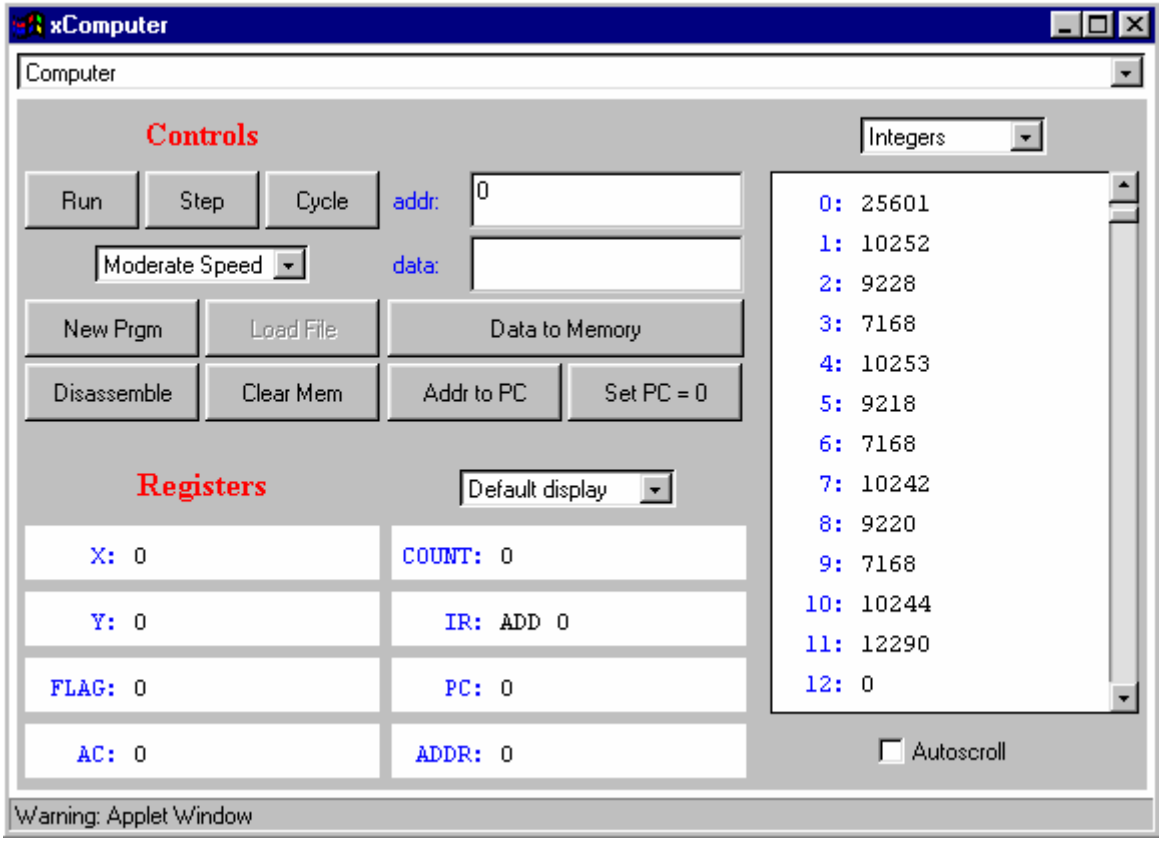


Figure 2: The xComputer

- xComputer

The xComputer, shown in Figure 2, is a Java applet [10] that supports David J. Eck's delightful text, *The Most Complex Machine* [11]. This book introduces computer science majors to computing in general. The text is equally suitable as a basic text for non-majors. The simulator that Eck provides is colorful and inviting to student experimentation.

The xComputer is also available as a standalone Java application, however, we found its use of deprecated (i.e., obsolete) Java classes presented insurmountable difficulties when compiling the programs. (These problems could surely have been resolved with sufficient perseverance, but this is not a task that is consistent with the assumed skills of the simulator's neophyte users.) Thus, only the applet version is readily available to most students. The main difficulty presented by the applet-

only version is that xComputer program files cannot be saved or retrieved through the simulator. (This is a Java applet restriction.) Students must therefore provide their own source file management through other text-based editors. (The source code can be copied and pasted into the simulator's editor.) The simulator does, however, possess some excellent features:

- ◊ Navigation between the main part of the simulator and the editor is simple and intuitive.
- ◊ Two stepping modes are available through control buttons. The "Step" button causes the execution of one machine step of an instruction. Thus, several "steps" are required to complete a single assembly language instruction. A "Cycle" button (named for fetch-decode-execute cycle) executes an entire instruction.
- ◊ Five simulator speeds are available, thus allowing program execution to be observed in "slow motion" without continual interaction by the user.
- ◊ Memory can be displayed as decimal integers (signed or unsigned), binary numbers, ASCII characters, or as graphic symbols. The user may also switch from the memory window display to a control wire signal display.
- ◊ Registers can be displayed in binary, signed decimal or unsigned decimal.
- ◊ By default, the instruction register presents the mnemonic form of the currently executing instruction.
- ◊ The memory observation window can be set to autoscroll, so the student doesn't have to worry about which address should be moved into the window.
- ◊ As instructions are executed, the location of the instruction (or data) is highlighted.

The xComputer simulator provides a clear depiction of a simple von Neumann architecture that bears much resemblance to MARIE. Two significant limitations of this simulator are:

- ◊ The xComputer is (easily) runnable only as an applet, causing difficulties with loading, saving, and printing assembly language files.
- ◊ No hexadecimal translations are available for registers or memory.

◊ The assembly "listing" provides only mnemonic code, with no hexadecomal equivalents, and, accordingly, no symbol table is produced as an aid to tracing and debugging.

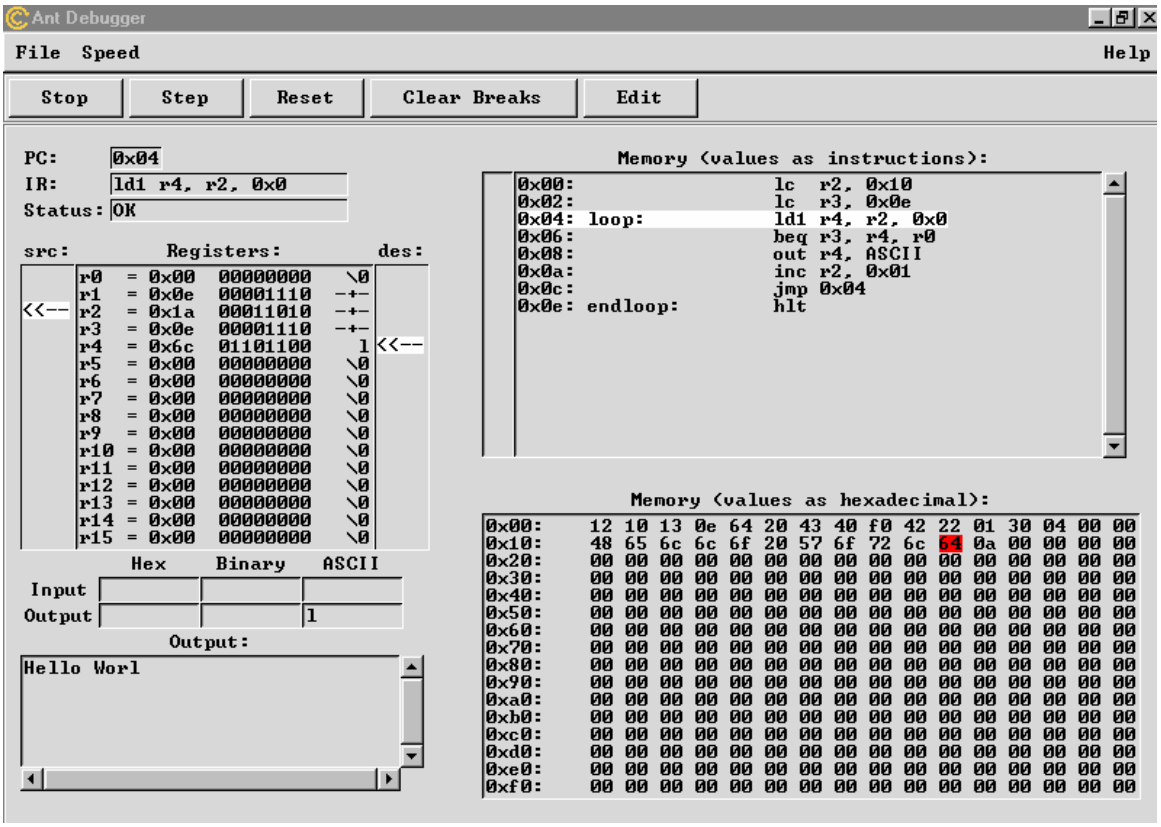


Figure 3: The Ant Debugger

- Ant

The Ant machine is a von Neumann architecture invented for general pedagogical purposes by a group of Harvard University researchers [12][13]. Like the xComputer, Ant is aimed at reinforcing concepts of computer organization and architecture as it is taught to beginning computer science and engineering students. At this writing, we are aware of no textbooks that expressly incorporate the Ant simulator architecture into a discussion of the von Neumann architecture.

The Ant architecture and its debugger (simulator) are useful teaching tools. There are two versions of Ant, Ant-8 and Ant-32. The 32-bit version contains advanced features (virtual memory and system calls, for example) that are aimed at advanced architecture and programming language courses. Ant-8 closely resembles MARIE in that it has a limited fixed-length instruction set and only a few registers. Like MARIE, Ant is a simple system designed to illuminate basic computer concepts for beginning computer science majors.

Ant's debugger provides a simulation environment analogous to MarieSim. The Ant debugger, shown in Figure 3, provides a number of features that compare favorably with the xComputer and Pep/7 simulator:

- ◊ The layout and presentation are clear in their purpose and meaning.
- ◊ A simple integrated editor is provided.
- ◊ Input and output can be in binary hex, or ASCII. (Like many real systems, program instructions determine the format of the output and expected input.)
- ◊ The source code panel scrolls automatically, keeping the currently executing instruction always within easy sight.
- ◊ Instructions and memory locations are highlighted as they are accessed, thus maintaining a connection between the instructions and their effects upon the machine.
- ◊ Register and memory contents are shown in hexadecimal, rather than in decimal.
- ◊ Ant provides code breakpoint assembly instructions and also allows breakpoints to be set during debugger execution. .
- ◊ A step mode is available.
- ◊ Three execution speeds are available.

There are, however, ways in which its usability could be improved: Ant's user interface could be more intuitive. Specifically:

- ◊ Register values are given only in hexadecimal. No radix translations to binary, decimal or character are provided.

- ◊ The assembler provides no symbol map or assembly listing.
- ◊ There is no direct printing facility.
- ◊ The simulator fails to ask the user for confirmation upon terminating the program. It is too easy to inadvertently close the program's window.

We feel that MarieSim incorporates the best features offered by the Pep/7, xComputer, and Ant simulators. We have added features to this baseline functionality that improve usability and realism without increasing complexity or raising the student's learning curve. As a result, MarieSim provides an easy-to-use, visually-appealing environment that provides maximum learning support for the student. It illuminates and reinforces the simple architecture of the MARIE computer, which we describe in detail in the next section.

2 MARIE: A Simple Machine Architecture

The MARIE system organization, shown graphically in Figure 4, includes the essential components of a one-address, von Neumann machine with the following characteristics:

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 4K words of main memory (12 bits per address).
- 16-bit data words.
- 16-bit instructions, 4 for the opcode and 12 for the address.
- A 16-bit arithmetic logic unit (ALU).
- Seven registers for control and data movement.

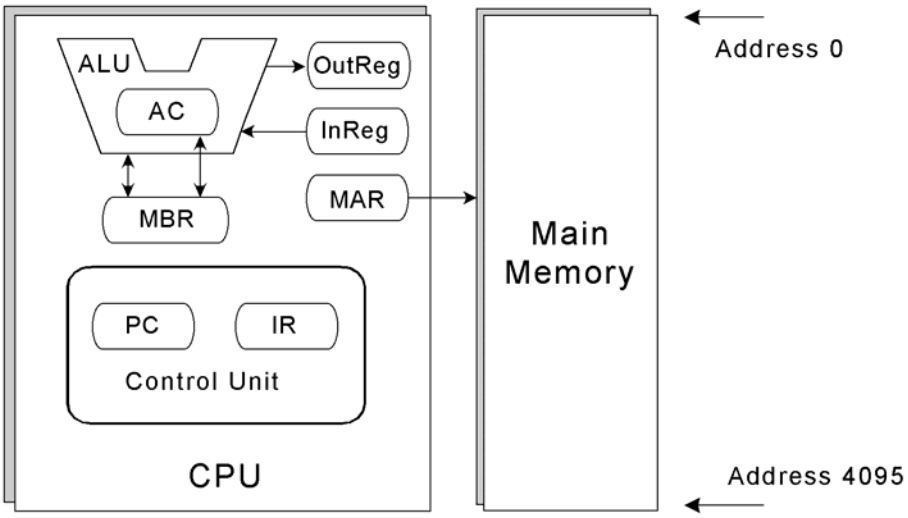


Figure 4: The Organization of MARIE

2.1 Registers and Buses

MARIE's seven registers are the minimal set required to support this simple architecture. These registers are the:

- Accumulator, AC, a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
- Memory address register, MAR, a 14-bit register that holds the memory address of an instruction or the operand of an instruction.

- Memory buffer register, MBR, a 16-bit register that holds the data after its retrieval from, or before its placement in memory.
- Program counter, PC, a 14-bit register that holds the address of the next program instruction to be executed.
- Instruction register, IR, which holds an instruction immediately preceding its execution.
- Input register, InREG, an 8-bit register that holds data read from an input device.
- Output register, OutREG, an 8-bit register, that holds data that is ready for the output device

Unlike most general-purpose systems, MARIE contains no user-addressable registers. For instructions requiring two operands, such as the ADD instruction, one operand must be in the accumulator, and the other in memory. The accumulator stores the results of all arithmetic operations.

The input and output registers "automatically" convert input and output between human-readable numerals or characters and machine-readable binary and vice versa. Computer system input and output operations are nontrivial, requiring I/O handlers and hardware interfaces. To maintain utmost simplicity, however, we consciously chose to omit the requisite details of this process. This design choice is also made clear in [1].

2.2 The Data Path

The MARIE machine contains a single 16-bit bus that conveys bytes to and from the registers and memory. No bus arbitration protocol is required owing to MARIE's single-thread design. All data movement occurs under the direct control of a single program running in memory. Each register connected to the bus is identified by a number, from 1 through 7, as shown in Figure 5. Main memory is located at Bus Address 0. Thus, three control lines are required for device addressing, and a fourth to indicate direction. A register or main memory may place data on the bus only when its number is enabled on the control lines.

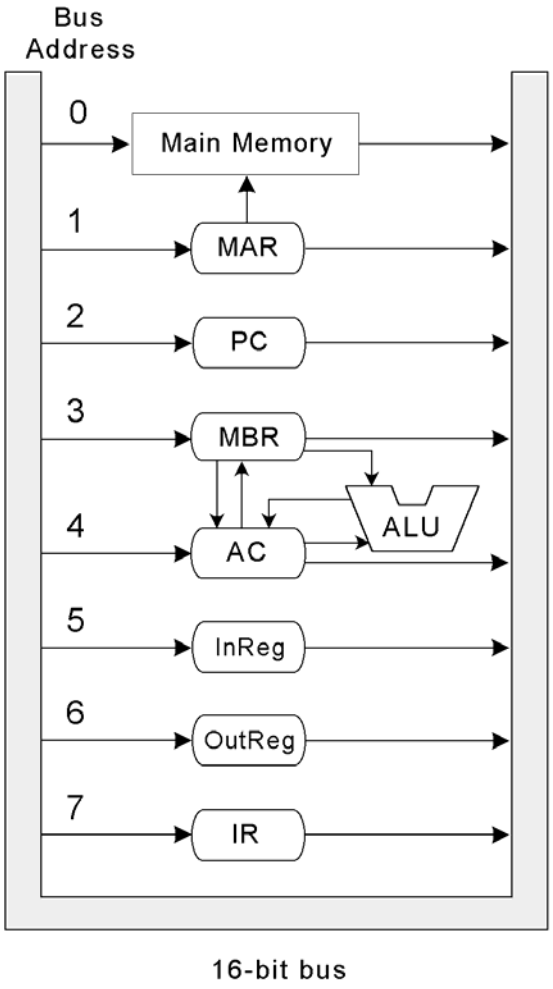


Figure 5: The MARIE Data Path

To speed up execution, separate pathways are provided between the MAR and memory; from the MBR to the AC; and from the MBR to the ALU. Information can also flow from the AC, through the ALU and back into the AC without being put on the common bus. The advantage gained using these additional pathways is that information can be put on the common bus in the same clock cycle that data is put on these other pathways, allowing these events can take place in parallel. For pedagogical purposes, these separate pathways also provide a more interesting and somewhat realistic register transfer language.

2.3 The Instruction Set Architecture

MARIE supports 16-bit instructions. The most significant 4 bits, bits 12-15, comprise the opcode that specifies the instruction to be executed (which allows for a total of 16 instructions). The least significant 12 bits, bits 0-11, form an address, which allows for a maximum memory address of 2^{12} - 1. The instruction format for MARIE is shown in Figure 6. Figure 7 depicts the MARIE instructions, along with the register transfer language corresponding to the instruction.

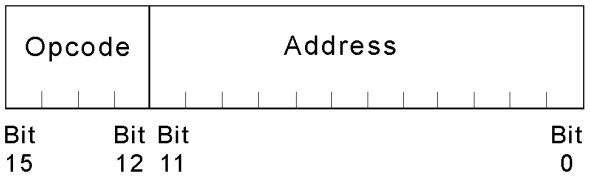


Figure 6: The MARIE Instruction Format

The MARIE instruction set includes memory access (`LOAD` and `STORE`) instructions, arithmetic (`ADD` and `SUBT`ract) instructions, input and output (`INPUT` and `OUTPUT`), execution flow control (`JUMP` and `HALT`), and an instruction to set the accumulator to zero (`CLEAR`). Binary operations expect to find one operand in the accumulator and the other at the memory address specified within the instruction. Conditional branching is controlled by the `SKIPCOND` instruction. `SKIPCOND` employs a binary conditional operand in the address field of the instruction: If the first two bits in the address field are zeroes, and the accumulator is negative, the instruction that follows the `SKIPCOND` statement is skipped. Likewise, if the first two bits of the address field are 01, and the accumulator is zero, or if the first two bits of the address field are 10 and the accumulator is positive. Otherwise, execution continues with the next program statement.

Opcode	Instruction	Register Transfer Language
0000	JnS X	MBR ← PC MAR ← X M[MAR] ← MBR MBR ← X AC ← 1 AC ← AC + MBR PC ← AC
0001	Load X	MAR ← X MBR ← M[MAR], AC ← MBR
0010	Store X	MAR ← X, MBR ← AC M[MAR] ← MBR
0011	Add X	MAR ← X MBR ← M[MAR] AC ← AC + MBR
0100	Subt X	MAR ← X MBR ← M[MAR] AC ← AC – MBR
0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	

1000	Skipcond	If IR[11-10] = 00 then
		  If AC < 0 then PC ← PC+1
		Else If IR[11-10] = 01 then
		  If AC = 0 then PC ← PC + 1
		Else If IR[11-10] = 10 then
		  If AC > 0 then PC ← PC + 1

1001	Jump X	PC ← IR[11-0]
1010	Clear	AC ← 0
1011	AddI X	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← AC + MBR
1100	JumpI X	MAR ← X MBR ← M[MAR] PC ← MBR

Figure 7: The Complete MARIE Instruction Set with Register Transfer Language

There are three indirect addressing mode instructions in MARIE. These instructions were included in the instruction set for the sole purpose of demonstrating the concept of indirect addressing and to permit limited subroutine functionality. The ADDI (add indirect) instruction assumes that the address of the address of the second operand. The JUMPI (jump indirect) instruction operates similarly using a branch instruction. Limited

subroutine functionality is provided by the `JNS` (jump and store) instruction. The return address is stored in the memory location specified by the operand. Execution proceeds with the instruction immediately following the address specified. When the "subroutine" completes, a `JUMPI` instruction (that points to this stored address) is issued to provide the "return" from the subroutine.

This limited instruction set is sufficient to allow students to experience assembly language programming, without overwhelming them with the complexity of instruction set architectures characteristic of real machines.

3 Significance for Computer Science Students

The value of computer architecture simulation and visualization software is well documented in the literature. (See [14], [15], and [16].) Wolffe, et. al. [16] state that the use of architecture simulators is gaining popularity for the following reasons:

1. Students can see the computer through various levels of abstraction.
2. Simulation software is more readily available to students—non-traditional students, in particular—than conventional campus-based computer labs.
3. Simulators are available for numerous advanced topics.
4. Many textbooks incorporate simulators for topical reinforcement and motivation.
5. Much simulation software is freely available.

To this list, we would add that simulation software is a product of known behavior over which the instructor has control. Installation of revisions to simulator software can be managed in accordance with the academic calendar. This is not always the case with physical systems. Upgrades to the hardware and software can occur without an instructor's knowledge, causing hasty, "last minute" adjustments to course lectures and materials that could confuse and frustrate students.

Ultimately, simulators provide a safe and inviting environment within which students can experience a computer system. Learning is by doing, thus providing a vehicle for all learning styles. MarieSim provides an intuitive graphical environment within which students can create their own real, runnable, programs. Through their own efforts, students can observe the effects of their instructions upon the state of the machine. Through hands-on experience with an uncomplicated machine, the lessons become clear.

4 MarieSim, The MARIE Simulator

MarieSim, is a Java-based, graphical software machine that illustrates the operation of the MARIE architecture described in Section 3. Its goal is to provide an intuitive and visually appealing environment that will encourage student understanding and experimentation.

4.1 Description

The principal component of MarieSim is the virtual machine environment with which students interact. All components of the system, registers, instructions, and memory, are visible on the screen simultaneously. Furthermore, source instructions are displayed in a separate panel within the simulator allowing the student to see the symbolic instructions along with their hexadecimal memory equivalent. No additional keystrokes (or mouse clicks) are required to show memory versus instructions.

Execution speed is controllable through menu options, and a single-step mode is available. Breakpoints can be set by clicking checkboxes within the instruction display window. For ease of understanding, all register displays, as well as input and output, can be read or displayed in hexadecimal, decimal, or as the ASCII character equivalents of the numeric values. The output format of each register can be set independently of the others. When the simulator is running a program, the relevant program instruction is highlighted in the instruction window, as is the memory location corresponding to the value displayed in the memory address register.

By highlighting each instruction as it executes, and its memory location, MarieSim draws an animated and graphic correspondence between an instruction and its effects upon the state of the machine.

4.2 Design Decisions

MarieSim is written in Java and Java Swing. The primary reason for this choice is that Java Virtual Machines (Java bytecode interpreters) have been written for every type of

real machine that one could reasonably expect to be available to university students. A second consideration is that Java includes a rich set of features and functions that are not provided by most other popular languages. Third, the use of Java makes MarieSim extensible in many ways owing to the popularity of the language. (See Section 5.)

The overarching design goal of the MARIE simulator was to provide an inviting and intuitive environment for students to interact with a simple, hypothetical machine. The purpose of the simulator is to reinforce von Neumann machine concepts. Thus, every effort was made to reduce the learning curve with regard to interaction with the simulator itself. Virtually all operational features are explained through help texts, or are congruent with operation of other software with which the student is expected to be familiar. (For example, if one desires to load a file, one does so by pressing the "File" button on the menu bar at the top of the simulator.) Simulator controls are enabled and disabled in accordance with the state of the simulator. For example, the "Stop" button is enabled only when the machine is running, and the "Run" button is enabled only when the machine is halted and an executable program is loaded.

MarieSim's various panel backgrounds consist of soft colors that are easy on the eyes. Meaningful content (e.g., register values) is shown in dark, bold colors. Source code and memory contents are traced with bright highlights, thus providing sufficient contrast for users who may have color vision impairments [17], or who may be using monitors with limited color resolution.

The simulator layout includes as much of the machine on the screen as possible. Supplementary functions, such as text editing, code assembly, and execution speed controls are provided through popup windows or drop down menus. To make room for all of this, the view of the instructions and memory had to be limited. However, memory contents and mnemonic instructions are both displayed in separate scrollable panes. Thus, while the student cannot see the entire contents of memory, or the entire program in its mnemonic form, the simulator will automatically scroll to each instruction as it executes, and to each memory location as it is accessed.

The most vexing design decision involved the format of the assembled, executable MARIE program files. As currently implemented, the MARIE assembler produces a Java class file as output. When this class file is created by an assembler that is compiled on one machine, it is not directly executable in the simulator on a second machine. If two users wish to share a MARIE program, the source code must be compiled through the simulator that will run it.

The advantage in the present approach is that the simulator is assured of the integrity of the assembled code when it is loaded. It cannot be tampered with, or corrupted, in any way. Also by using a class file, as opposed to the pure binary output of other assemblers, the source code, in its symbolic form is easily stored along with the executable code. The symbolic code includes instruction mnemonics, user-defined variable names, and code labels. Certainly to do all of this, it is possible to devise a binary file format that is not a Java class file, but in so doing, greater error checking would be required at program load time within the simulator. Thus, in order to provide integrity and functionality, class file output was selected. If the source code is error-free, only a few mouse clicks (issued from within the simulator) are required to produce an executable program file from the source code. After successful assembly, we are assured that the binary object is runnable and complete.

4.3 Functionality

As stated above, the simulator consists of a collection of programs written in Java and Java Swing. There are two main components in the system: the MARIE assembler and the MARIE graphical simulator environment, MarieSim.

The MarieSim environment, shown in Figure 8, consists of three main parts: A menu bar that controls operation of the simulator, the main pane that contains the components of the MARIE machine, and a message area at the bottom of the screen to display status and error messages.

The menu bar controls the overall operation of the simulator. Through the menu bar, the user loads files, runs programs, sets the execution speed, and starts and stops program execution. Options are also available for displaying help text or the symbol table of a loaded program. These options, and their effects upon the simulator, are shown in detail in the user's guide supplied as Appendix A.

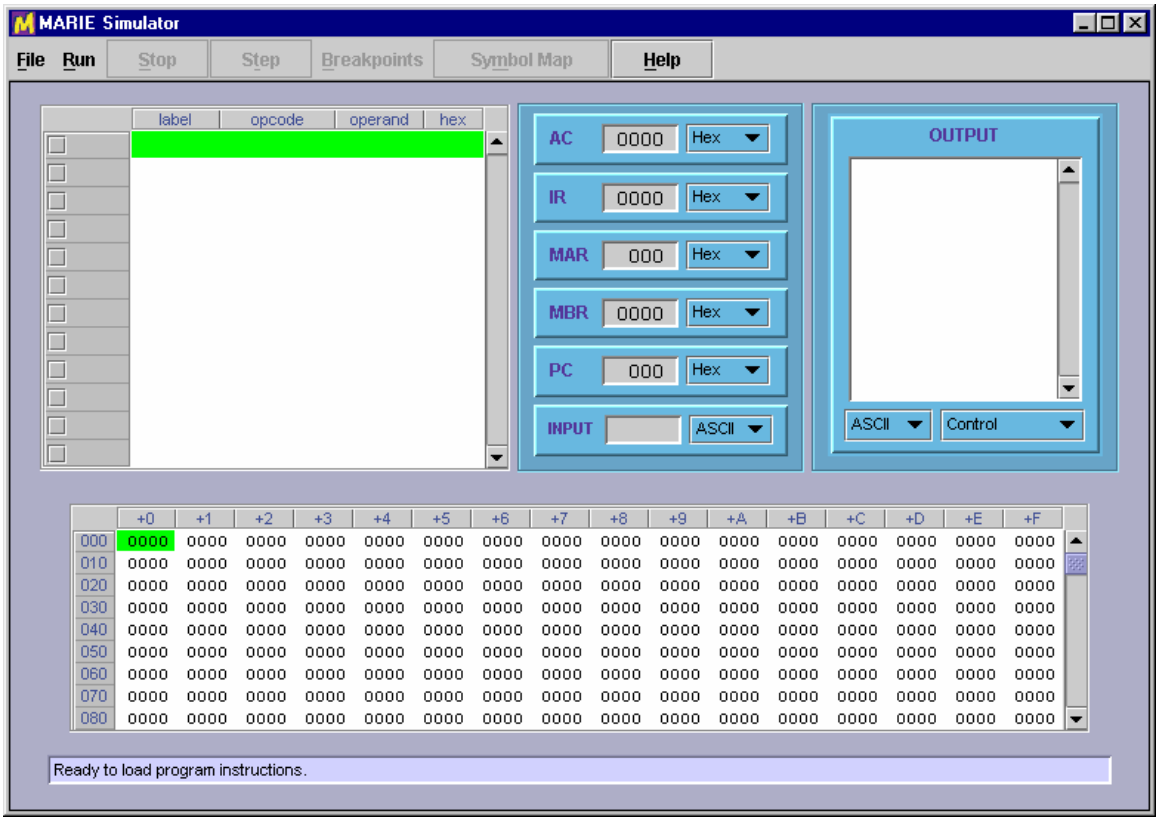


Figure 8: The MarieSim Graphical Environment

MarieSim runs programs written in MARIE assembly language. For the convenience of the user, MarieSim provides an integrated text editor from which the assembler can be invoked. The MARIE assembler is written in Java. It can be used as a standalone program, or as a callable module. In either mode, it expects to be given the name of a MARIE assembly code source file, which is a plain text file containing MARIE assembly code having a `.MAS` file extension. The assembler conducts two passes over the source code. During the first pass, source statements from the text file are parsed to create a symbol table containing labels and other program symbols. An object stream (file)

containing partially assembled MARIE code, is produced at the completion of this first pass. During the second pass of the assembler, addresses of the program symbols (labels and variables) are retrieved from the symbol table and supplied to the code that was partially assembled during the first pass. Upon successful assembly, three files will be present: a .MEX (MARIE Executable) class file for the program, a .LST plain text program listing (with mnemonics, symbols and addresses), and a .MAP symbol table (also plain text) that can be displayed as an aid to the user during program execution.

When errors are found in the code, a message displays in the message area of the editor and the assembly listing automatically pops up in a small window. Figure 9 shows the integrated editor and assembler with an unsuccessful assembly effort. Upon receiving this feedback, the student can correct the program error and request reassembly through the integrated editor. When assembly is successful, the editor also displays a completion message.

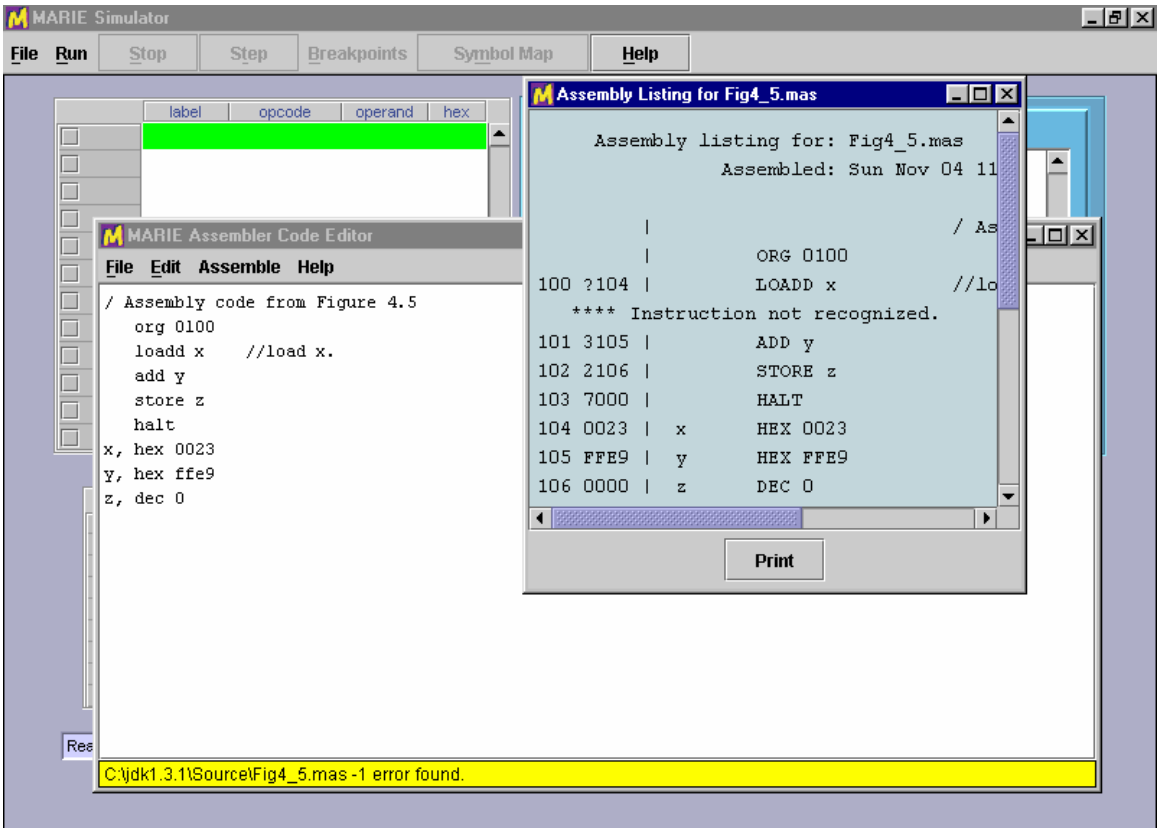


Figure 9: An Unsuccessful Program Assembly

To load a MARIE executable file into the simulator, the student selects the `File|Load` option from the menu bar. This action evokes a popup window that displays a listing of all of the MARIE executable programs in the current directory. The directory may be changed through Java functionality included in the file selection window component.

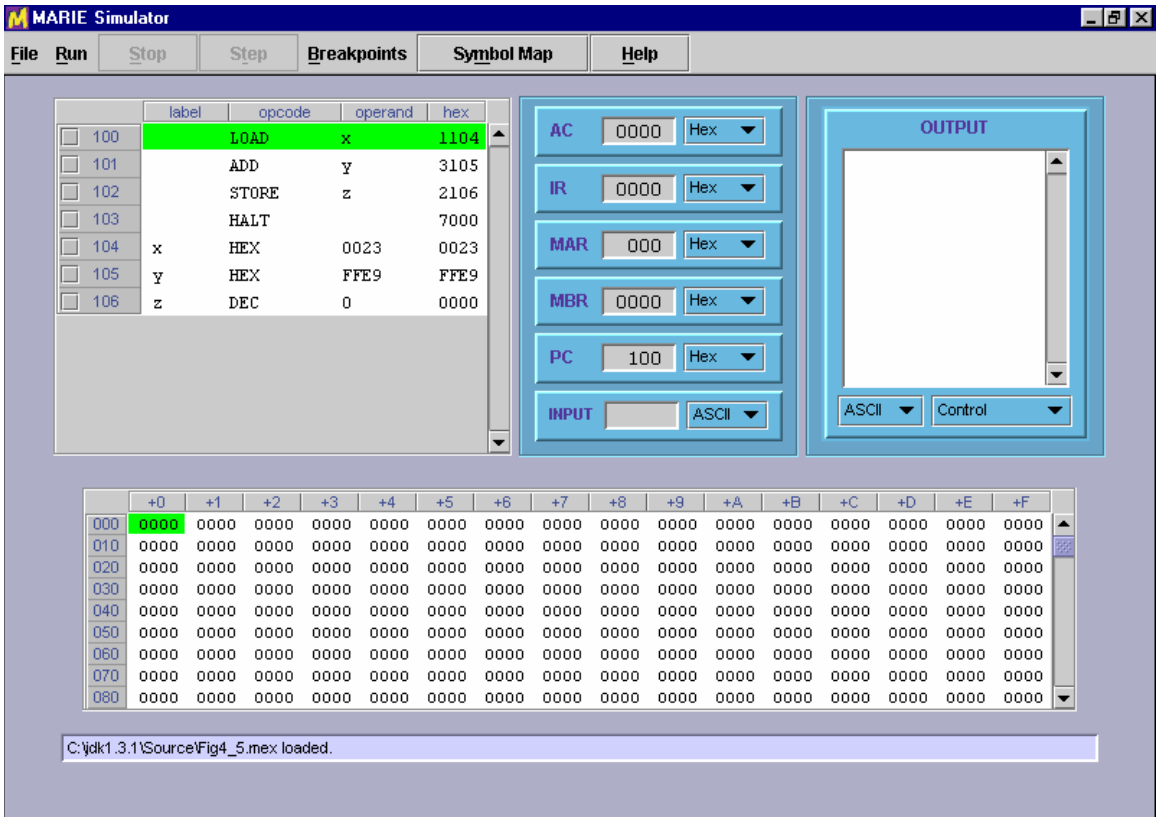
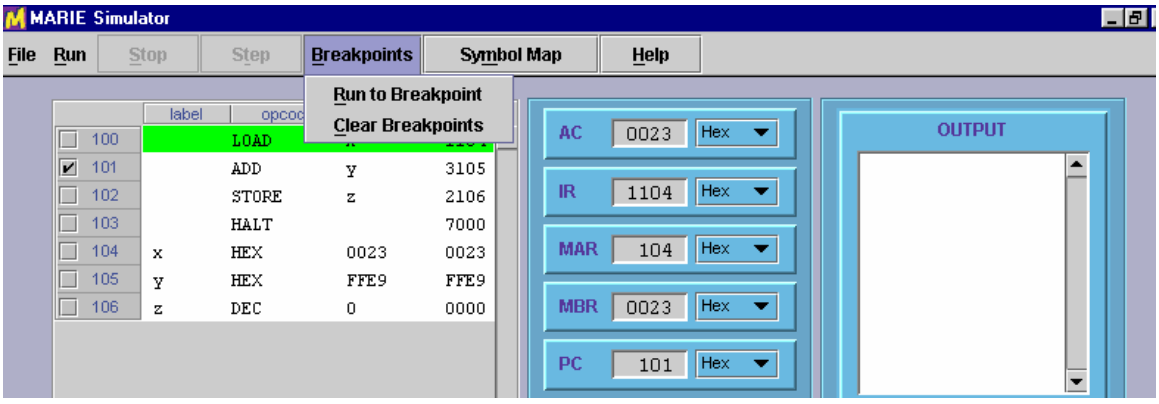


Figure 10: A Program Loaded and Ready to Run

As the MARIE executable class file is loaded into the simulator, a window containing the mnemonic code is populated while its hexadecimal equivalent is loaded into the memory table at the bottom of the simulator. If the mnemonic version of the program contains more than 12 lines, a vertical slider bar will appear on the right-hand side of the mnemonic code window. Thus, the student can scroll through the entire program at any time. All 4096 MARIE memory locations are displayed in a similar sliding display at the bottom of the simulator. (As an aid to the student, memory is initialized to all zeroes each time a new program is loaded.) Figure 10 shows a seven line program that is loaded

and ready for execution. The first program instruction was loaded at memory location 100h in accordance with an ORiGination instruction placed in the source code[1]. The MarieSim loader always assumes that the first line of executable code is the first line of the program and sets the program counter accordingly.

A stepping mode can be set in the simulator to allow the student to carefully observe the effect that each program instruction has upon the state of the machine. Selection of these modes is shown in Figure 11. For long programs or programs with ponderous looping behavior, the single-step mode may prove too tedious, so two additional execution mode features have been provided. One of these is an option that allows control of the delay between executing consecutive program instructions. The delay feature enables the student to see his or her program running in "slow motion." If the slow mode proves wearisome, the student may instead set breakpoints in the code simply by checking a box adjacent to the code line of interest in the mnemonic program instruction display window, as shown in Figure 12. If the student chooses to just run the program (not stepping through it), the "Stop" button at the top of the simulator is enabled, allowing for a "panic mode" halt to the program. This feature will come in handy should an infinite loop condition arise. (The simulator might not halt instantaneously, because it can only issue a signal requesting a Java thread process to stop. This event scheduling is not under the direct control of the simulator.)



[1] The ORiGination directive is not specified in the textual description of the assembler. It was added to the functionality of the simulator to amplify the concept of code relocation and its effects upon symbol addresses.

Figure 11: Program Execution Modes

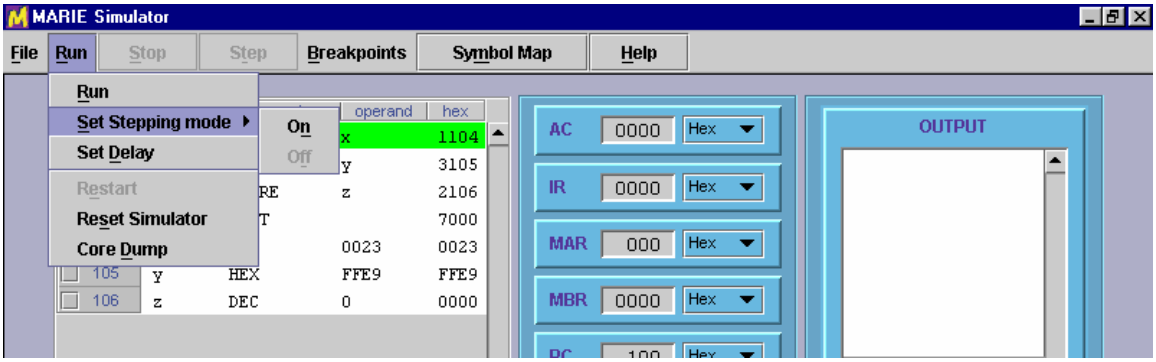


Figure 12: Breakpoint Program Control

While a program is running, the instruction corresponding to the value of the program counter is highlighted, along with the memory location corresponding to the value of the memory address register. These highlighted cells are scrolled within their respective display panels so that they are visible at all times while the program executes. Thus, the student can sit back and carefully observe the execution of his or her program without any concern for the operation of the simulator itself.

A student observing or debugging a MarieSim program also has the option of displaying register values in hexadecimal, decimal, or as an ASCII character equivalent (modulo 128). This feature is particularly useful with respect to input and output registers, because most students are more comfortable with decimals and characters than they are with the hexadecimal numbering system. Character input and output is particularly helpful to students writing programs that perform any kind of string handling or that solicit confirmation prompts from the user.

The simulator requests confirmation prior to performing any “destructive” operations, such as closing the simulator. This feedback helps to avoid grave errors such as eradicating the machine state in the middle of a lengthy execution trace.

In summary, the MarieSim environment provides a safe, usable, graphic environment for learning von Neumann machine concepts. The simulator has been designed so that beginning computer users will have little trouble with its operation, allowing them to focus their efforts on writing and debugging MARIE assembly language programs. Observing the execution of these instructions promotes a deep understanding of the operation of a von Neumann machine.

5 Future Enhancements and Expansions

The Essentials of Computer Organization and Architecture text outlines the requirements for instruction set architectures that differ from the one given for MARIE. In particular, various instruction address architectures are given along with the requisite changes in the instruction formats. Instruction architectures for two-address machines and zero address (stack) machines are outlined. Within these descriptions lies an opportunity for students to experience programming within different machine environments, thus imparting a real appreciation for the differences in these architectures. Accordingly, the MARIE machine could be recast into two additional and equally simple architectures that support zero- and two-address instruction sets. Creation of simulators for both of these environments would enlarge a student's comprehension of the differences and similarities between various instruction set architectures.

With MarieSim, MARIE program assembly occurs in the background, as a black box process. Students may benefit by observing real time construction of the symbol table and two-pass assembler operation. This process could also be illuminated through graphical software that is integrated into the simulator.

MarieSim, at this writing, does not offer visibility to MARIE's data path. While register-to-register and register-to-memory data flows are not particularly difficult concepts to master, animation software written for thus purpose would provide completeness to the simulation environment.

Surely, as MarieSim gains use in the educational environment, additional enhancement opportunities will present themselves. It is hoped that all future work will continue in the spirit of making the simulator as easy to use as possible, with an unwavering focus on providing the student the clearest possible view of the machine and its internal operations.

6 Conclusion

This paper has presented a graphical environment that simulates the operation of the simple von Neumann machine, MARIE, as described in *The Essentials of Computer Organization and Architecture*. This simulator, MarieSim, animates and reinforces concepts imparted by the text within a safe and inviting environment. MarieSim provides a host of features to assist the student in writing and debugging MARIE assembly language programs. These features include those offered by many other graphical simulation systems: register and memory contents displays, program execution speed control, and stepping mode execution, among others. MarieSim augments this essential functionality by providing an assortment of user tools including: an integrated assembler that produces realistic assembler listings with symbol maps, breakpoints that can be set and cleared in real time within the execution monitor, execution monitoring that establishes the correspondence between symbolic instructions and their effects upon the machine, as well as register and output displays that can be set independently to show contents in hexadecimal, binary, or character mode. Above all, MarieSim's graphical environment is visually appealing and its operation is straightforward.

MarieSim offers another option for the many educators who wish to incorporate simulation and visualization software into their curricula. The reasons for employing virtual systems over real systems are clear and compelling. Thus, the use of system simulators as tools for teaching introductory computer organization and architecture is becoming increasingly accepted, if not expected. Owing to this trend, it would seem that we are witness to the beginning of a new era in computer science education, where simulation software becomes the focus and foundation for instructional delivery. By harnessing and channeling the complexity of today's machines, we can provide the simplicity and abstraction required to master them.

References

[1] Null, L. and Lobur, J. 2003. *The Essentials of Computer Organization and Architecture*. Jones and Bartlett, Sudbury, MA.

[2] Yurcik, W., Wolffe, G. S., and Holiday, M. A. 2001. "A Survey of Simulators Used in Computer Organization / Architecture Courses." *Summer Conference on Computer Simulation*, Orlando, FL. Society for Computer Simulation.

[3] A compendium of computer simulators (for theoretical, obsolete, and real systems) can be found at: http://www.sosresearch.org/caale/caalesimulators.html.

[4] Braught, G. and Reed, D. 2001. "The Knob & Switch Computer: A Computer Architecture Simulator for Introductory Computer Science." *ACM Journal of Educational Resources in Computing 1,* 4, 31 – 45.

[5] Yurcik, W, and Brumbaugh, L. 2001. "A Web-Based Little Man Computer Simulator," *Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE), Charlotte NC USA Feb. 21-25.* 204-208.

[6] Yehezke, C., Yurcik, W., Pearson, M., and Armstrong, D. 2001. "Three Simulator Tools for Teaching Computer Architecture: EasyCPU, Little Man Computer, and RTLSim." *ACM Journal of Educational Resources in Computing 1,* 4, 60 – 80.

[7] Menczer, P. and Segre, A. M., 2001. "OAMulator: A Teaching Resource to Introduce Computer Architecture Concepts." *ACM Journal of Educational Resources in Computing 1,* 4, 18 – 30.

[8] Grünbacher , H. 1998. "Teaching Computer Architecture/Organization Using Simulators" *Proceedings of the IEEE Frontiers in Education (FIE) Conference.* Tempe, Arizona. 1107 – 1112.

[9] Warford, J. S. 2002. *Computer Systems, 2/e*. Jones and Bartlett, Sudbury, MA.

[10] The xComputer applet can be launched from:
http://math.hws.edu/TMCM/java/labs/xComputerLab1.html.

[11] Eck, D. J. 1995. *The Most Complex Machine: A Survey of Computers and Computing*. A K Peters, Ltd. Wellesley, MA.

[12] Ellard, D., Holland, D., Murphy, N., Seltzer, M. 2002. "On the Design of a New CPU Architecture for Pedagogical Purposes." *Proceedings of the Workshop on Computer Architecture Education,* Anchorage, AK. 28-34.
(http://www.csc.ncsu.edu/eos/users/e/efg/wcae/2002/submissions/ellard.pdf)

[13] Ellard, D., Ellard, P., Megquier, J., Chen, J. B., Seltzer, M. 1999. “The Ant Architecture - An Architecture for CS1.” *The IEEE Computer Society Technical Committee on Computer Architecture Newsletter.* 25-27. (http://www.ant.harvard.edu/Papers/ellard-tcca99.pdf)

[14] Bruschi, S. M, Santana, R. H.., and Santana, M. J. 1999 “Simulation as a Tool for Computer Architecture Teaching,” *Proceedings of the Summer Computer Simulation Conference*, Chicago, IL.

[15] Cassel, L., Kumar, D., Bolding, K. Davies, J., Holliday, M., Impagliazzo, J., Pearson, M., Wolfe, G. S., Yurcik, W. 2000. “Distributed Expertise for Teaching Computer Organization and Architecture..” *ACM SIGCSE Bulleti*n , 33, 2. 111-126.

[16] Wolffe, G.A., Yurcik, W., Osborne, H., and Holliday, M.A. 2002. "Teaching Computer Organization/Architecture With Limited Resources Using Simulators", *Proceedings of the 33rd ACM Symposium on Computer Science Education (SIGCSE),Technical Symposium*, Covington, KY.

[17] *Color on the Web*. http://www.fc.peachnet.edu/facultystaff/irc/webdesign/color.html.

Appendix – User's Guide for MarieSim

A Guide to the MARIE Machine Simulator Environment

Accompanying *The Essentials of Computer Organization and Architecture*

by

Linda Null and Julia Lobur

Version 1.0 – January 2003

Introduction

Your authors have made every effort to create a MARIE machine simulator that is as *Really Intuitive* and *Easy* to use as the MARIE architecture is to understand. We believe that the best way to gain a deep understanding of the MARIE machine—or any computer system for that matter—is to write programs for it. Toward our goal of helping you to understand how computers really work, we have created the Marie machine simulator, *MarieSim*. MarieSim is an environment within which you can write your own programs and watch how they would run on a real "von Neumann architecture" computer system. By running programs on this simulator, not only will you see your programs in action, but you will also get a taste of assembler language programming without learning any particular assembly language beyond the simple instructions that your authors have presented.

MarieSim was written in the Java language so that the system would be portable to any platform for which a Java Virtual Machine (JVM) is available. Students of Java may wish to look at the simulator's source code, and perhaps even supply improvements or enhancements to its simple functions.

Installation

The MARIE machine simulator requires Sun's Java SDK 1.4.0 or later. This software is available at no charge from the java.sun.com Web site. After this package is installed, the Java archive file MarieSim.jar (case sensitive) can be placed in the directory of your choosing. The following command will uncompress the archive:

```
jar xvf MarieSim.jar
```

If the archive uncompresses correctly, you will have the main MARIE simulator class file, MarieSim1.class and two MARIE code example files in your directory. Jar will also create two subdirectories, Meta-inf, and MarieSimulator. The MarieSimulator subdirector contains all of the (many) other classes required for simulator operation. (The Meta-inf subdirectory is created by jar.) Note: The MARIE simulator can be run directly from the jar file; however, "Help" and other text files will not display.

If you also wish to see MARIE's source code, you can obtain the MarieSource.jar file that contains all of the Java source for the Marie simulator. This file is uncompressed in the same way as the simulator jar file:

```
jar xvf MarieSource.jar
```

The java source will be uncompressed into the same directories as the class files.

To run the MARIE machine environment, the java classpath must be set to point to the directory where the MarieSim1.class file is located. For example, if your classpath is `C:\j2sdk1.4.0_01`, and you have located the MarieSim1.class file in a directory named

`C:\j2sdk1.4.0_01\marie`, you must change your classpath to `C:\j2sdk1.4.0_01\marie`. Within a Windows environment, you do this by using the SET command (`set CLASSPATH=C:\j2sdk1.4.0_01\marie`). In a Linux/Unix environment, the `.cshrc` file contains the classpath. (Check with your system support staff if you are unsure as to how to change this file.)

The MARIE simulator environment is invoked using the command:

```
java MarieSim1
```

within the directory that contains the MarieSim1.class file. (Note: This command is case sensitive!)

The MarieSim Environment

Figure 1 shows the graphical environment of the MARIE machine simulator. The screen consists of four parts: a menu bar, a central monitor area, a memory monitor and a message area.

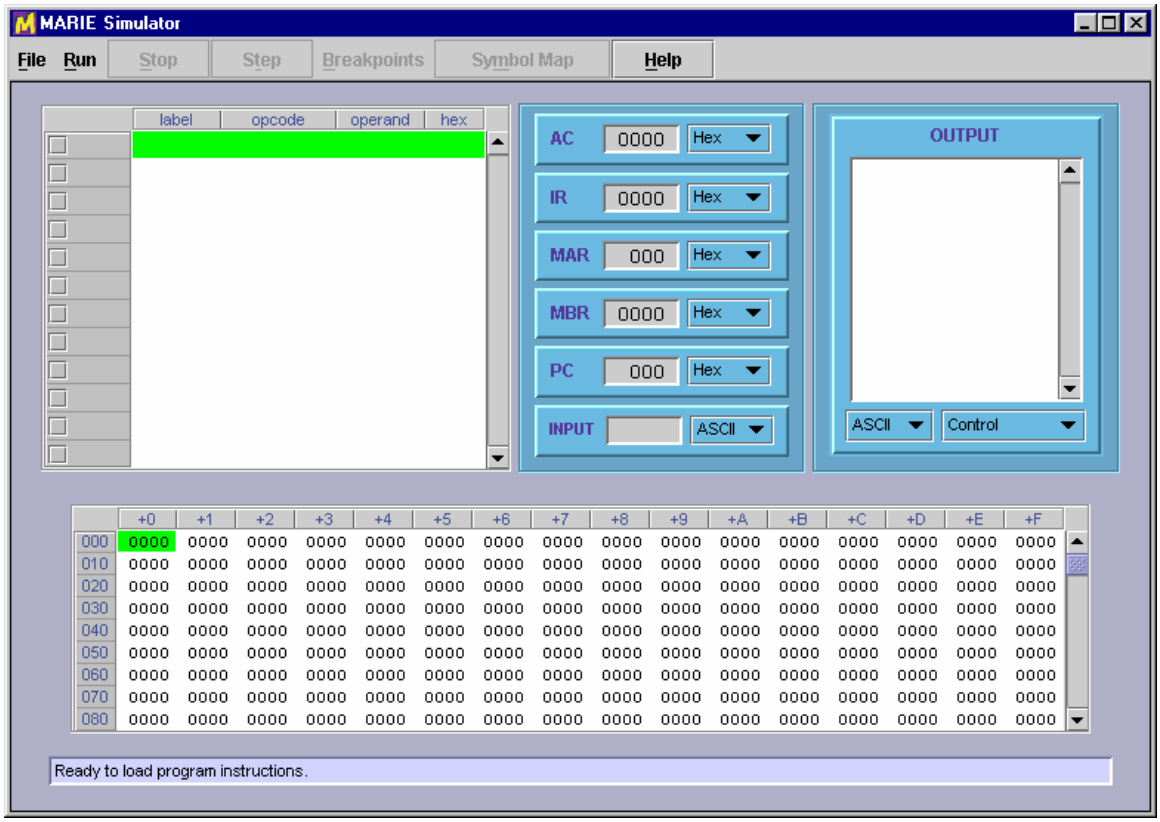


Figure 1: The MarieSim Graphical Environment

The central monitor area contains a *program monitor* area, six of MARIE's seven registers, and an *output area*, representing MARIE's seventh register.
The memory monitor area displays the contents of all 4096 addresses of MARIE's memory. Each horizontal row of the memory area contains 16 memory addresses.

Therefore, the address labels at the left side of the memory area are given in increments of 16, with titles above each column indicating the offset from the memory address of each row of memory. For example, the memory address DE8 is found in the column labeled +8 of the row labeled DE0. All addresses are given in hexadecimal.

As the simulator executes your program, the instructions in the program monitor area are highlighted along with any memory in the memory area that the instruction is accessing. These highlights are most visible (on a fairly "fast" system) when you set a 500 millisecond (or greater) delay between instructions. (See below). With a little experimentation, you will find an optimal value for your system.

During the course of executing your program instructions, status messages may appear in the message area at the bottom of the screen. When your program ends, you will see either a "Program halted normally" or "Program halted abnormally" message. If you never see this message, either your program hasn't started running yet, or it is in a loop and you'll need to halt it manually.

The MarieSim Controls Menu

The menu at the top of the simulator gives you control over the actions and behavior of the MARIE machine Simulator system.

The File Menu

The features available through the File menu are shown in Figure 2 If you already have an assembled MARIE program at your disposal, all you need to do is load it and run it. If you want to write a program from scratch, you should select the File | Edit option. The Edit option gives you a simple way to write and assemble programs in MARIE assembly language.

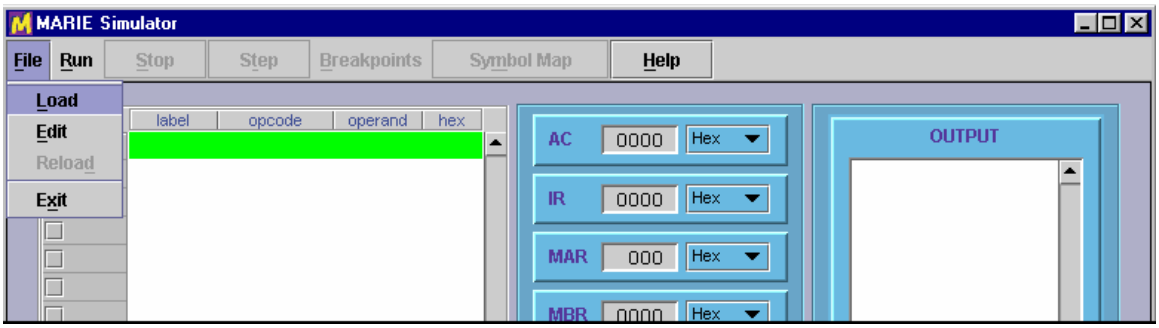


Figure 2: MarieSim File Menu Options

Although you can use any plain text editor (perhaps one with fancier features) to create your source code, the simulator's built-in editor gives you one-button access to the assembler. The MARIE editor frame is shown in Figure 3.

The MARIE Editor

Once you select File | Edit, and if you do not have a file loaded in the simulator (as shown in Figure 3), the editor frame is displayed with a blank text area. If, however, you have already loaded an assembled file into the simulator, the source code for that file is automatically brought into the editor if the editor can locate it.

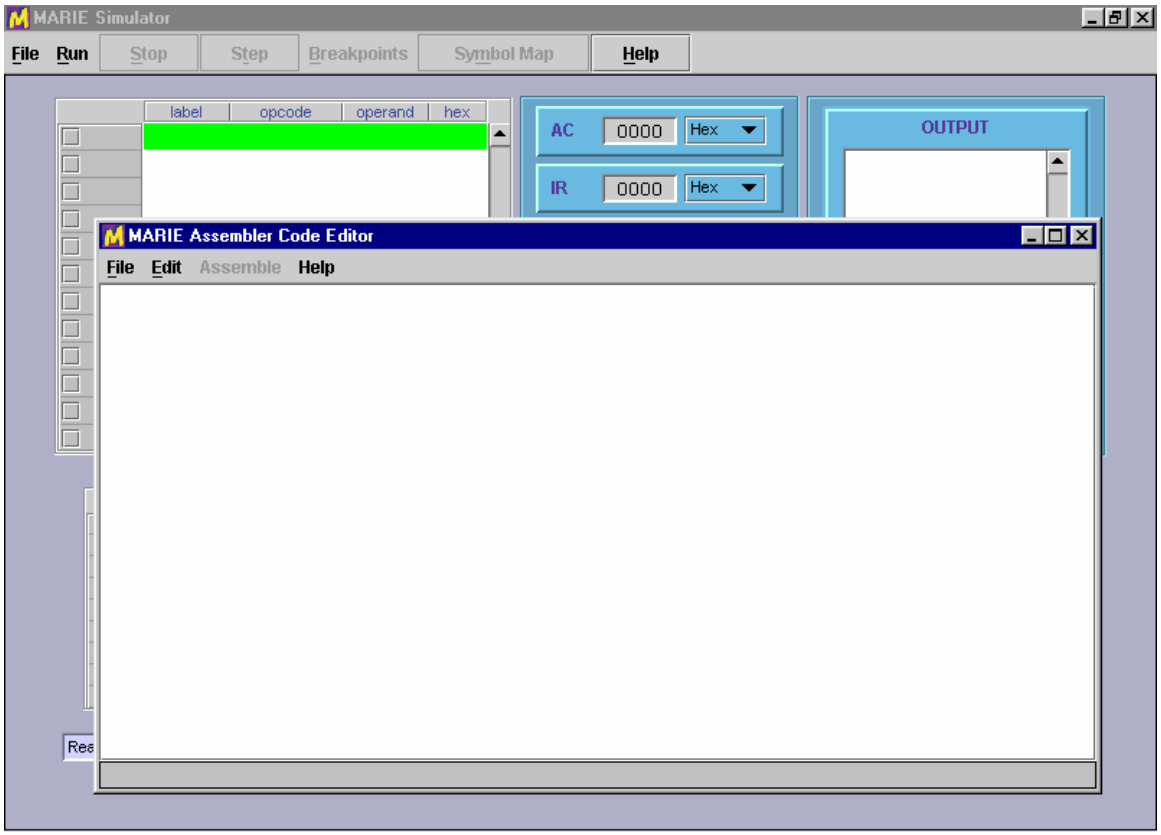


Figure 3: The MarieSim Editor

MARIE assembly code source files must have an ".mas" extension, for *M*ARIE *A*ssembler. Both the editor and the assembler recognize files of this type. Once you have saved a file with an ".mas" extension, the Assemble menu option becomes enabled and you can assemble your program by selecting the Assemble current file menu pick. If you load an existing ".mas" file, the Assemble button is automatically enabled. Any modifications that you have made to your to your assembly-language file are automatically saved by the editor prior to its invoking the assembler. This process is shown in Figure 4, using the example from Table 4.5 in the text.

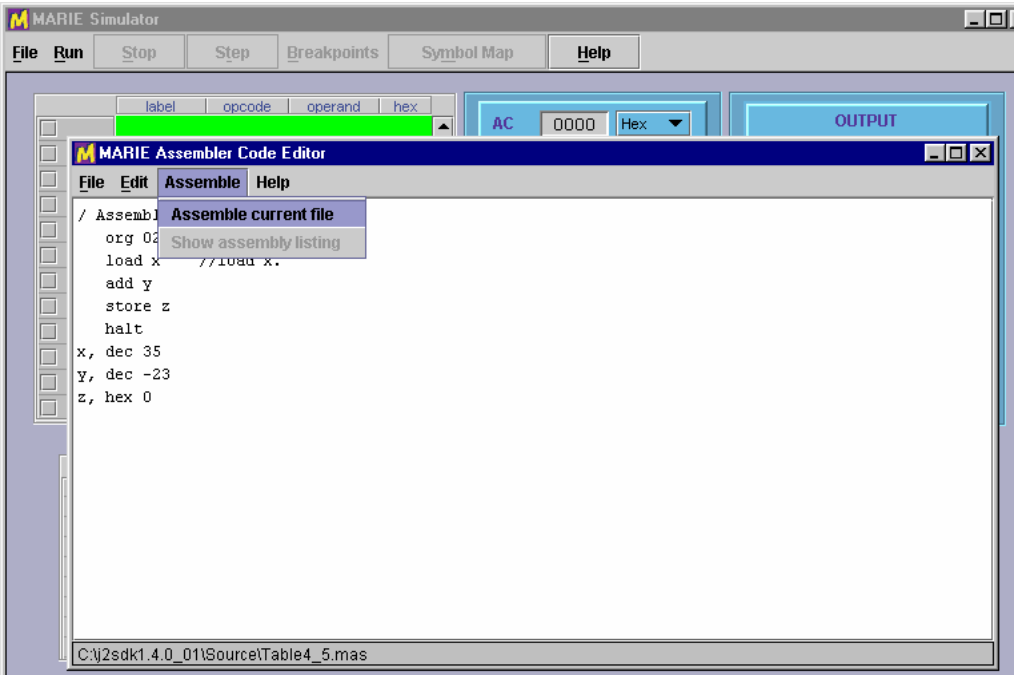


Figure 4: Preparing to Assemble Source Code

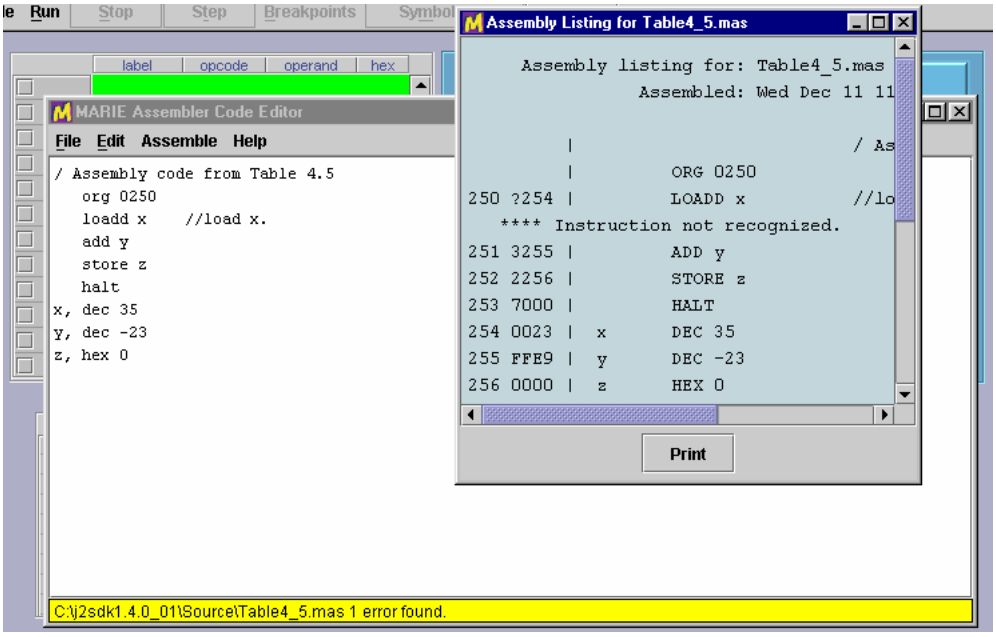


Figure 5: An Unsuccessful Assembly

If the assembler detects errors in your program, the editor sends you a message and the assembly listing file appears in a popup frame as shown in Figure 5. All that you need to do is correct your program and press the Assemble current file button once more. If the file contains no other assembler errors, you will see the screen shown in Figure 6. If

you wish, you can display or print the assembly listing file, by using the editor or any text-processing program.

The listing file will be placed in the currently-logged directory, along with the "MARIE machine code" file, if assembly was successful. The listing file is a plain-text file with an ".lst" extension. For example, when Fig4_5.mas is assembled, the assembler produces Fig4-5.lst. You may view it, print it, or incorporate it into another document as you would any plain text file. If assembly is error-free, a ".mex" or MARIE EXecutable file will also be placed in the same directory as the source and listing files. This is a binary file (actually a serialized Java object) that is executable by the simulator.

For example, if your assembly source code is called MyProg.mas, the listing file will be called MyProg.lst and the executable will be called MyProg.mex.

Once you have achieved a "clean" assembly of your program, you will see the message shown in Figure 6. If you are satisfied with your program, you can exit the editor by closing its window or selecting File | Exit from the menu.

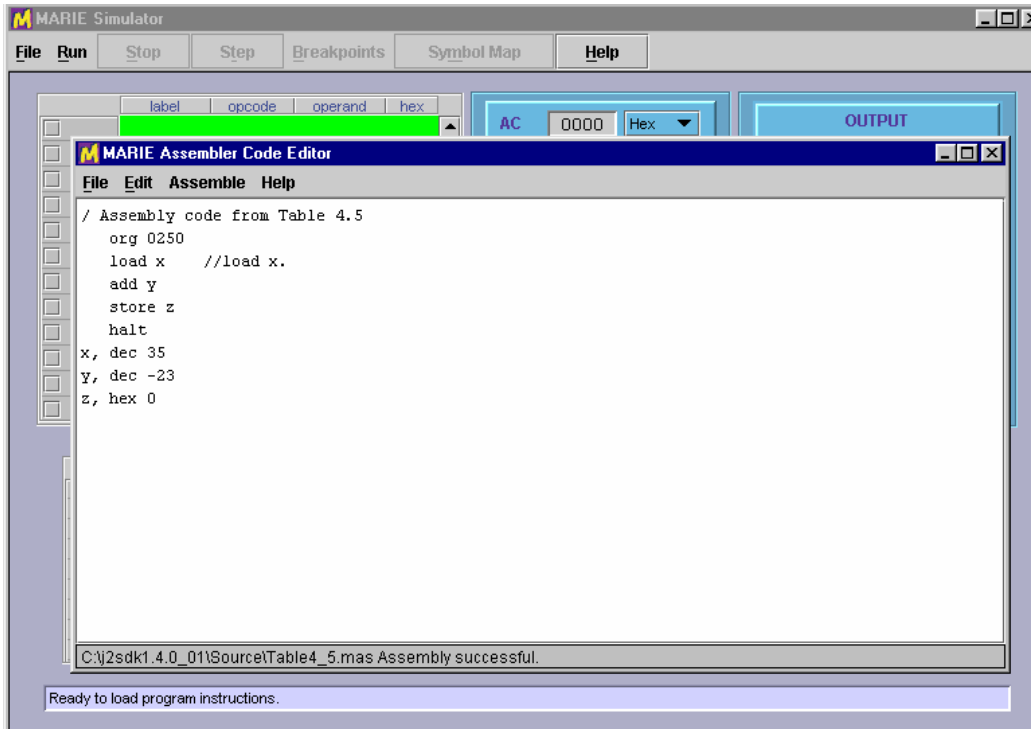


Figure 6: A Successful Assembly

As implied above, the MARIE editor provides only the most basic text-editing functions, but it is tailored to the MarieSim environment. The Help button provides you with some general help, as well as an instruction set "cheat sheet" that you can use for reference as you write your programs.

The frame in which the editor appears may look a little different on your system, but you can manipulate it as you can with any frame, that is: you can maximize it,

minimize it, hide it or close it. This is true of all frames spawned by the editor as it responds to your commands.

Loading Your Program

After you have successfully assembled your program, you must load it into the simulator by selecting the File | Load menu option from the simulator. This option brings up a file chooser panel that lists all of the MARIE executable files in your current directory, and the names of other directories that are available to you. All you need to do is highlight or type the name of the file that you wish to run.

Note: *Each time you reassemble a file, you must reload it.*

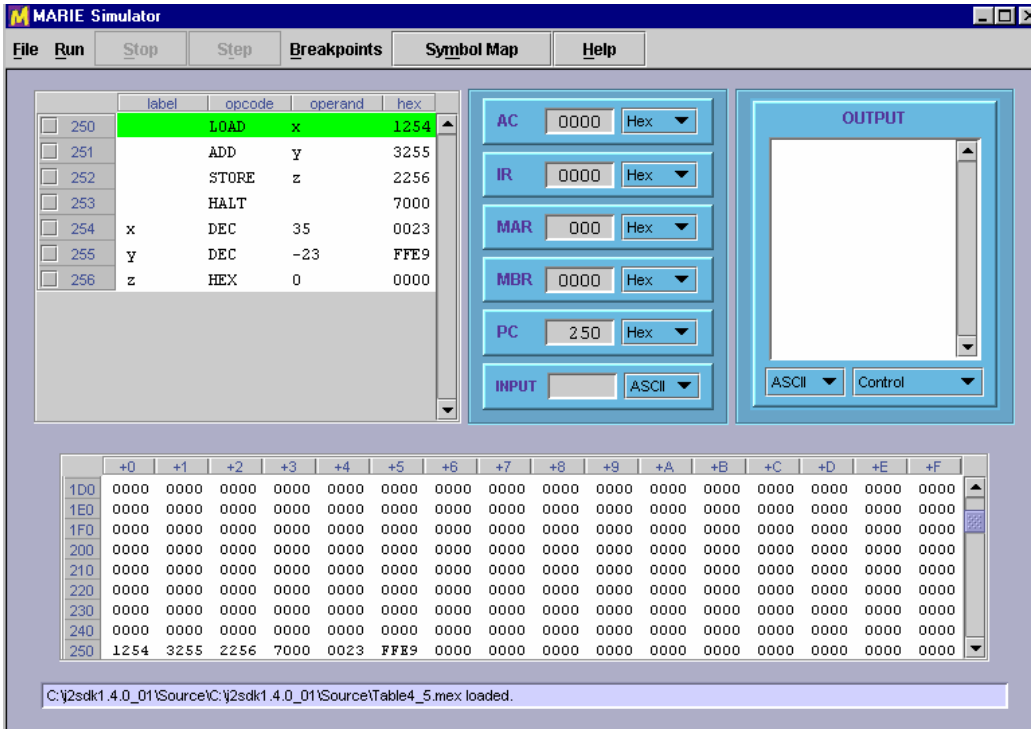


Figure 7: A Program Ready to Run

Figure 7 shows the MARIE simulator after an executable file has been loaded. The program monitor window shows the assembly language statements as they were written, along with their hexadecimal equivalents. At the left-hand side of the program monitor, you will see the addresses of the program statements. The statement that has just been executed by the simulator is shown in green highlight, so that you can see the effect that the instruction has had upon the state of the machine. Of course, when the program is first loaded, the green highlight will be on the statement at the first address of your program. You will also notice that the PC register is set at the address of the first statement in your program, indicating that this is the *next* statement that will be run.

Keep in mind that the program monitor window is there only to help you visualize what is going on. The program instructions are, in reality, pulled from the memory panel at

the bottom of the screen. Once you have loaded your program, you will notice that the memory monitor contains the hexadecimal program instructions in the addresses corresponding to those in the program monitor window. A green highlight will move to different memory location as your program runs, accessing various storage locations.

Once loaded, your program can be executed using any of three different run options.

The Run Menu

The Run menu offers a number of features that allow you to have control over how your program is executed by the simulator. As shown in Figure 8, the first option on this menu is Run | Run, which executes the statements in your program in sequence to termination. When you select Run | Run, the Stop button becomes enabled, giving you the chance to halt your program should it get stuck in a loop or simply is taking too long to run.

Figure 8 shows the Run menu option that you would use to put your program into *step mode*. Step mode allows you to execute your program one statement at a time. After executing each statement, the simulator pauses until you press the Step button (or the program terminates).

Note: If the simulator is in step mode, and you subsequently select Run | Run, the simulator automatically terminates step mode and enters run mode.

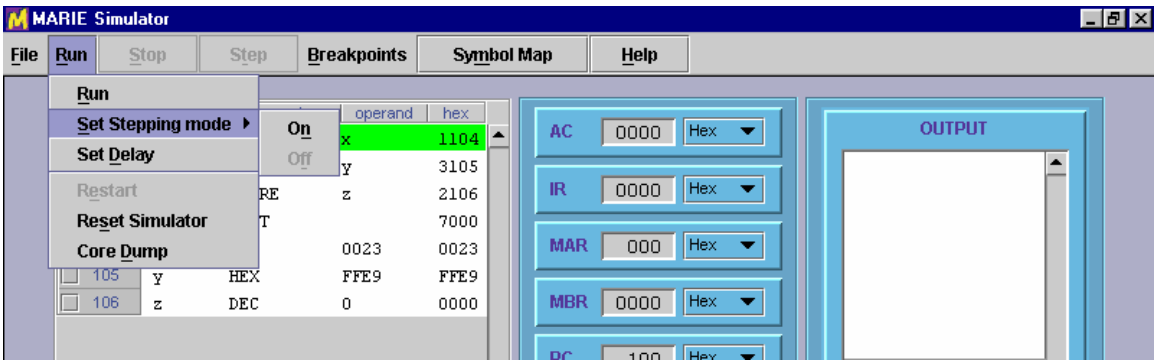


Figure 8: The Run Menu

The Run | Set Delay Option

By default, the MARIE simulator pauses for approximately 10 milliseconds between subsequent executions of program statements when it is in run mode. The main purpose for this delay is to allow you to halt execution of your program, should you desire to do so. The delay feature may also be used to allow slow-motion viewing of your program statements as the simulator executes them. You can put the simulator in this slow-motion mode by setting the delay to 500 milliseconds or longer.

The delay-setting screen is shown in Figure 9. To change the execution delay, just move the slider bar to the desired number of milliseconds and press the Okay button.

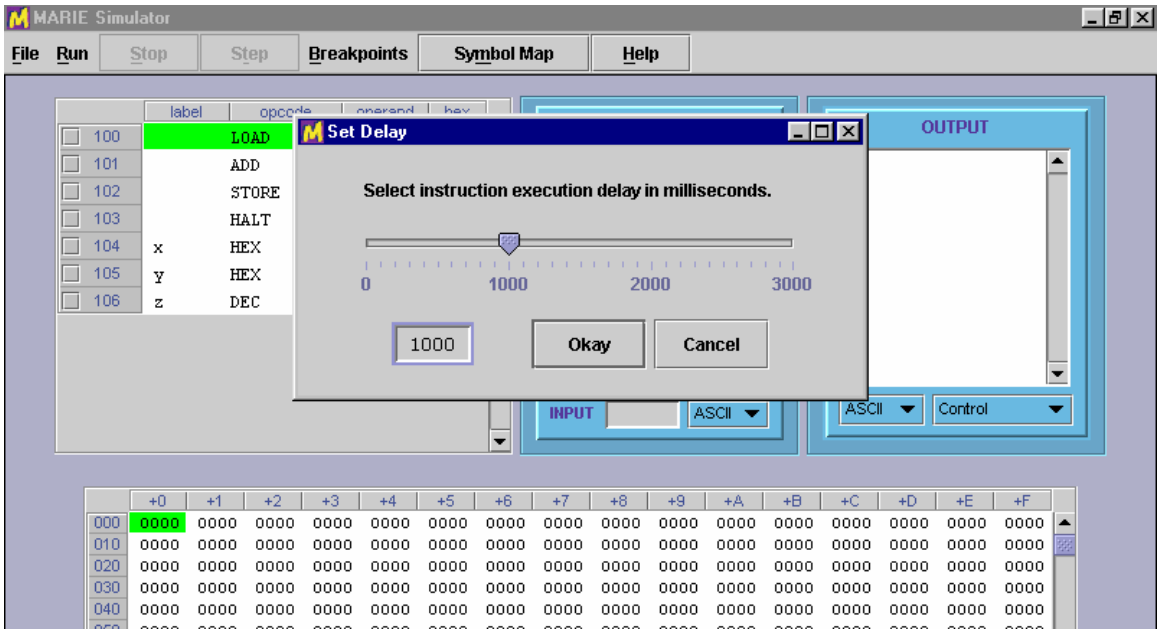


Figure 9: Setting Execution Delay

Setting the slider: As you would expect, you can move the slider pointer by clicking and dragging it with your mouse. You can also move it using the cursor-movement keys on your keyboard. Page Up and Page Down move the slider by large increments, and your left and right arrows move it by smaller increments, allowing for precision setting of the slider bar.

The Run | Restart Option

The next option under the Run menu is the option to Restart the simulator. This option simply resets the program counter to the first address of the program that is loaded in the simulator. Any changes that your program may have made to the simulator's memory stay in place.

The Run | Reset Option

To completely start over from scratch, use the Run | Reset option. Selecting this option has the same effect as pressing the reset key on a personal computer. All memory is cleared and the registers are reset to zero. Because this option eradicates everything in the simulator, you will be asked for confirmation before proceeding with the reset (unlike pressing the reset button on most PCs!).

The Run | Core Dump Option

Many computers dump the entire contents of their memory after they encounter certain severe errors. If you have used Windows NT, you may have experienced the famous "blue screen of death" which appears while the system is dumping its memory to a dump file. When it encounters a fatal error, the MARIE simulator does not automatically provide a core dump, you need to request one through the Run | Core Dump menu option. This menu option provides you with the popup frame shown in Figure 10.

As you can see from the figure, the popup shows two sliders (which, like the delay slider, can be controlled using cursor keys). The scales on the sliders range from 0 to 4095 and are given in decimal. The hexadecimal translation of the slider values appears in the text boxes to the right of the sliders. The initial values of the core dump are set to the memory address space occupied by your program. In the illustration, our program occupies memory addresses 100h through 106h. You can move these sliders to any addresses that you wish.

If you press the Okay button from the core dump selection frame, another frame soon appears, containing the contents of MARIE's registers and the memory addresses that you selected. This screen is shown in Figure 11.

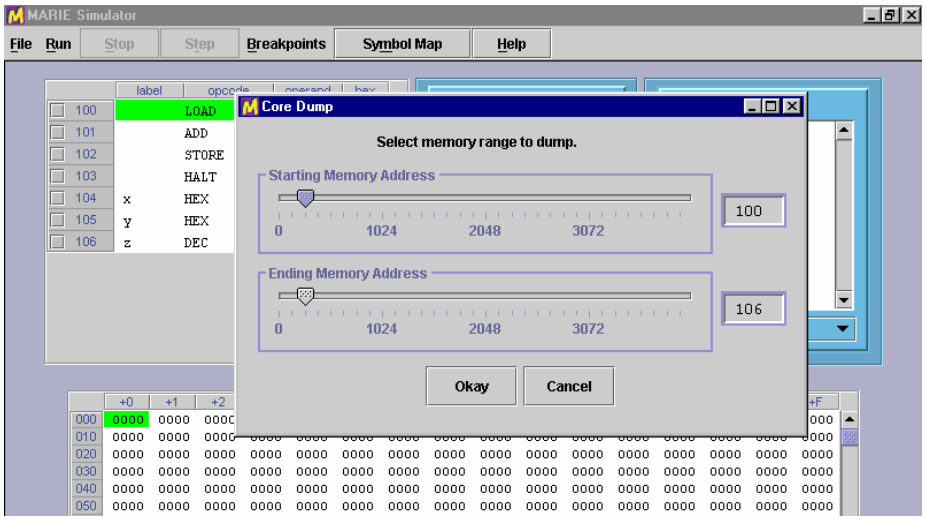


Figure 10: Setting Memory Core Dump Range

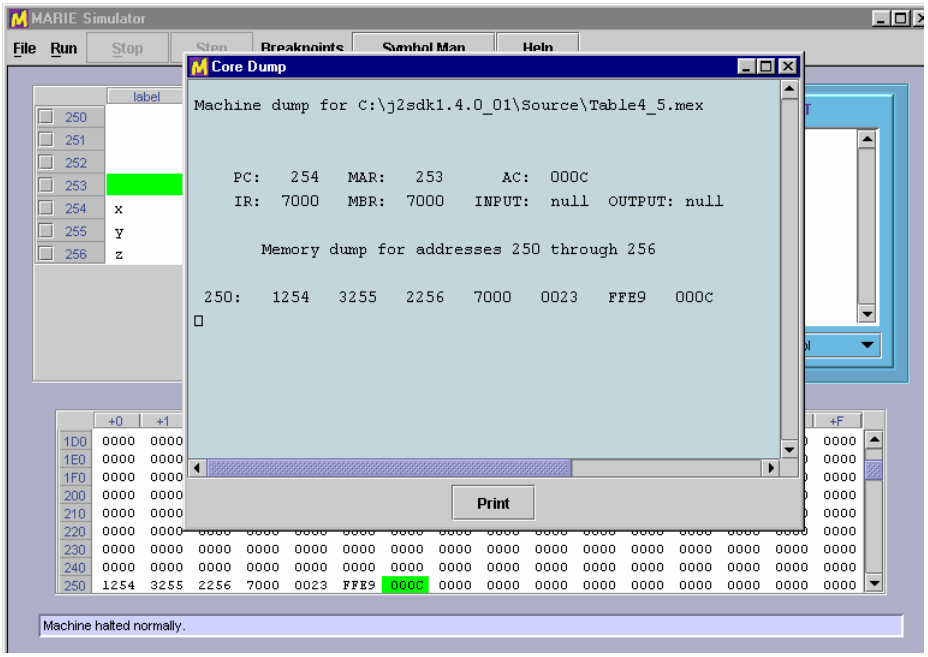


Figure 11: A MARIE Core Dump

Before showing it to you on the screen, the simulator writes your core dump to a disk file with a ".dmp" extension. So if the name of your program is MyProg.mas, its dump file will be named MyProg.dmp. It is a plain text file that you can edit using any plain text editor, if you so desire.

Breakpoints

Breakpoints are flags placed on computer instructions that tell the system to pause execution at the instruction where the breakpoint is set. Breakpoints are useful because a long series of instructions can be executed quickly (as in initializing an array, for example) before the system pauses to allow you to inspect the contents of registers and memory before proceeding. In the MARIE simulator, breakpoints are set by clicking on the square to the left of the instruction number in the program monitor area. One such breakpoint has been entered at instruction 101h in the program shown in Figure 12. There is no limit to the number of breakpoints that can be set in a program.

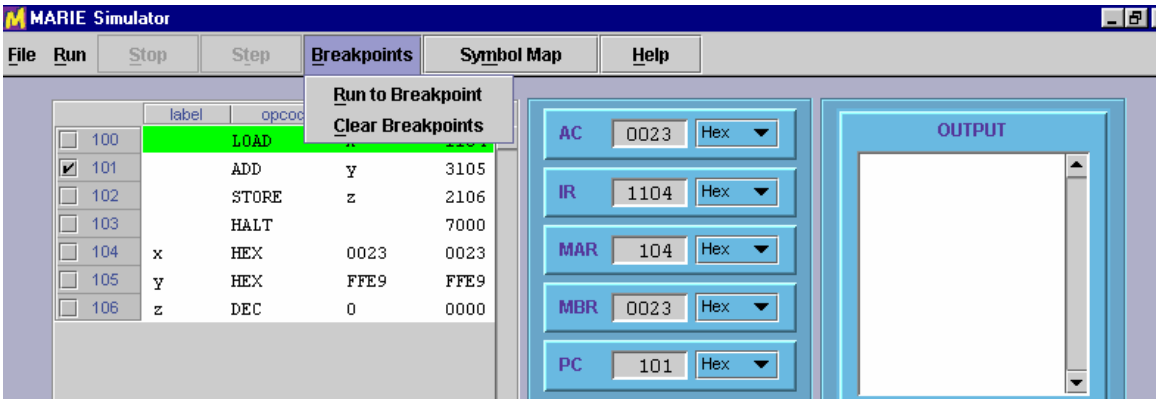


Figure 12: Breakpoint set at 101h and the Breakpoint Menu

When you select the Breakpoints | Run to Breakpoint menu option, the program starts execution at the current value of the program counter and proceeds until the breakpoint is encountered. Pressing Breakpoints | Run to Breakpoint once more resumes execution until the next breakpoint is encountered or the program terminates. If you select Breakpoints | Run to Breakpoint when the program counter is pointing to a halt instruction, the simulator performs an automatic restart and executes your program beginning with the first instruction.

You may notice that when your program is in Run to Breakpoint mode, the Stop button becomes enabled. This allows you to halt your program if you need to do so. The Run to Breakpoint option is also mindful of the execution delay that you have set in the program. So if you have set this delay at 500 ms, the simulator will wait approximately one-half second between each statement.

The Breakpoints | Clear Breakpoints menu option does exactly what you think it would do: It removes all breakpoints from the program instruction monitor. You can also remove a breakpoint by clicking on its checkmark.

The Symbol Table

From the discussion in your text, you know that one of the principal data structures involved in the assembly process is the symbol table. MARIE's assembler is no different. You will see this table at the end of your assembly listing, whether or not assembly was successful. The MARIE simulator also makes this table available to you right in the simulator environment through the Symbol Map button. If you are trying to debug a large program, or a program that contains a considerable number of symbols, you may want to refer to this table from the simulator environment. Figure 13 shows a symbol table display frame for the small program loaded in the simulator.

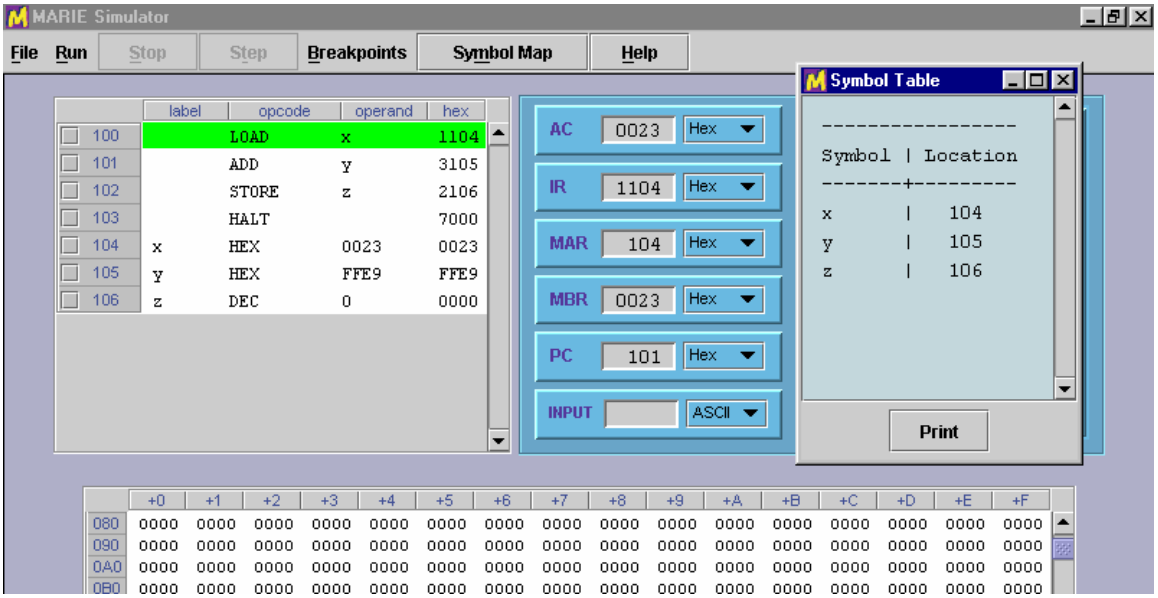


Figure 13: Viewing a Symbol Table

Input and Output Modes

You will notice in the figures that each register, as well as the OUTPUT pane, is provided with a pull-down (combo box) that lets you select the display mode of the register, either in hexadecimal, decimal or as an ASCII character. If you request a register to display its contents in ASCII character mode, the contents of the register will be displayed in a standard 7-bit ASCII modulus over the contents of the register. For example, if the register contains a value of 00C1h, the ASCII character displayed will be a capital A. Figure 14 illustrates register mode selection .

The mode selection available for the output register (or output pane) will be useful when you want to see character output. By default,

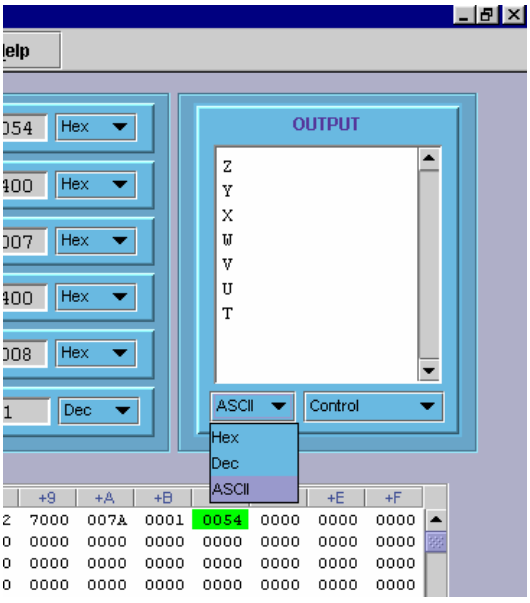
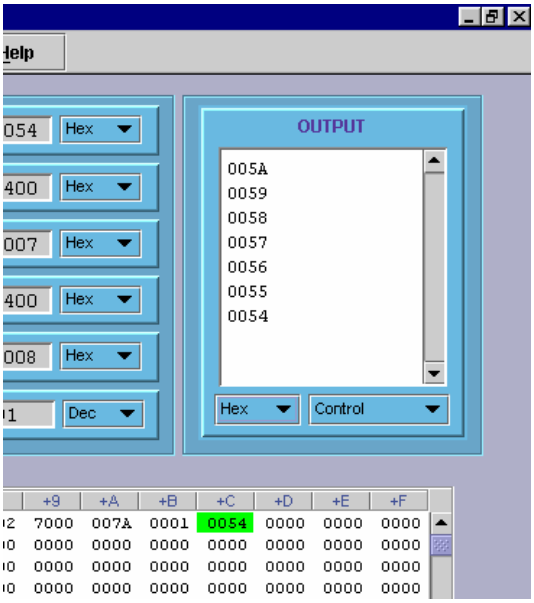


Figure 14: Setting a Register's Mode

both the input and output registers are in ASCII mode. So any values that you type in the input register will be interpreted as ASCII characters. For example, if your program is asking for input and you enter 29 while the input register is in ASCII mode, *only the first character of your input will be recognized* so the value stored in the accumulator will be 32h, which is the ASCII value for the decimal numeral 2. If you wanted to place a value of 2 into your program, you must set the register to decimal or hexadecimal mode. The values you enter must be appropriate to the register mode, or else you'll get a fatal error message in the simulator, as you would on a real system.



You will notice that when you change the radix mode of the output, all of your output immediately changes to the mode that you have selected, just as it does for the other registers. Figure 15 shows Figure 14 after the hexadecimal mode has been selected.

Another useful feature of the output pane is the ability to control how the output will be placed in the pane. By default, when a value is output, it is automatically followed by a linefeed so that the output will be a single column of characters, similar to how characters print on an adding machine tape.

Figure 15: Reformatted Output

If you select the No Linefeeds option, as shown in Figure 16, your output will be placed horizontally in the output pane, from left to right—with no automatic wrapping. When the No Linefeeds option is turned on, you will need to programmatically provide your own linefeeds order to advance the output to the next line.

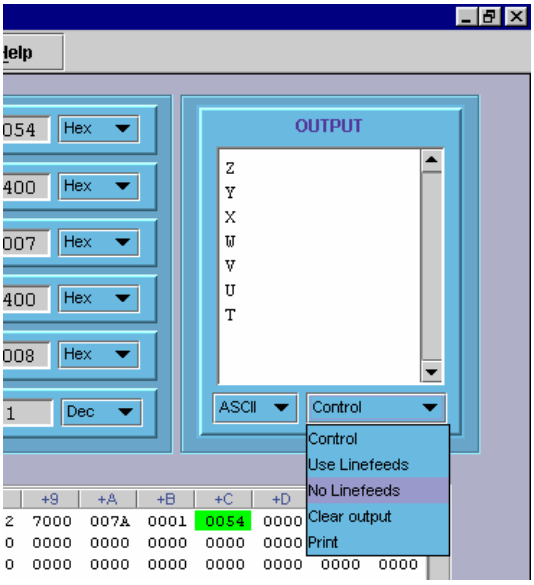


Figure 16: Setting Linefeeds

Implementation note: Although we have tried to make the MARIE Simulator behave as much like a real machine as possible, in the area of output and input we have deviated from this ideal. Handling input and output at the machine level is an enormously tedious task. Your authors feel that it is counterproductive to be wrestling with these issues when you're first learning assembly language programming. When you're up for a challenge, try using only hexadecimal values for input and no linefeeds on your output!

Another output control option available to you enables you to clear the contents of the output pane. Unlike the other six registers, the contents of the output register (pane) are retained until you explicitly clear it or load a different program.

You can also print the contents of the output register, but you may simply want to copy its contents into a text-handling program. On most systems, you can copy the contents of the output area into your system clipboard. The text can then be pasted into any program that recognizes the clipboard.

A Few Words About the MARIE Assembler

Your book discusses each of MARIE's instructions in detail, so we will not restate them here. There are, however, a few things particular to the assembler that you'll need to know before you begin writing your first program.

Assembler Directives

There are four directives that the MARIE assembler recognizes. The first of these is the origination directive, ORG. As stated in Chapter 4, the ORG directive controls the starting address of your program. If you do not include an ORG directive in your code, the first address of your program is automatically 000h. (Note: On a real machine, you could not count on this, which is why origination directives are used.) If you want the first address of your program to be 010h, you would place the directive, ORG 010 at the beginning of your program. The ORG directive must be the first statement of your program, otherwise, the assembler will give you an error.

The other three directives enable you to put constants in your program as decimal (DEC), octal (OCT), and hexadecimal (HEX) numbers. These constants must be valid for the radix stated in the directive. For example, the statement, OCT 0900, will give you an error.

Recall that MARIE's word size is 16 bits, and all constants are assumed to be signed numbers. Therefore, the valid range of decimal constants is decimal –32,768 to 32,767 (8000h to 7FFFh).

Operands

As you learned in Chapter 4, all MARIE operands are addresses. If you use address literals in your program, the valid values are 000h through FFFh, since MARIE has 4096 memory locations. These addresses must be given in hexadecimal, preceded by a zero. So, if you wish to add the value at address F00 to the accumulator, you would use the instruction, ADD 0F00. If you instead use the instruction, ADD F00, the assembler will expect to find the label F00 somewhere in your program. Unless you have such an address in your program, the assembler will give you an error.

You may prefer to use a symbolic labels instead of address literals. MARIE's labels are case sensitive and can be of virtually any length (up to the limits placed on string variables by the Java language). Be advised, however, that only the first 24 characters of the symbol will print on the assembly listing and only the first 7 characters will be visible in MARIESim's program monitor panel.

Code Construction

Assembly language source code files can be of any size, but you can have only as many program statements as will fit in the MARIE memory. The program statement count does not include comments or blank lines. We encourage you to use comments and blank lines to make your code understandable to humans as well as the machine.

The only other restriction on your code is that it can begin with only the following types of statements:

1. A comment followed by an origination directive and at least one imperative statement
2. An origination directive and at least one imperative statement, or
3. At least one imperative statement.

In other words, your program cannot begin with a DEC, OCT or HEX directive. If it does, neither the assembler nor the simulator will care, but the literal values defined by those directives will be interpreted as instructions by the machine, giving you results that you may not have expected.

A similar problem will arise if you fail to include a HALT statement in your program, or your HALT statement does not execute, and the program counter ends up pointing to data. (How will the machine react if it tries to execute the statement HEX 0000? What about HEX 0700?)

A Summary of File Types

Throughout this guide, we have mentioned the various file types that MarieSim uses for various functions. For your reference, we have included a summary of these file types in the table below.

File extension	Description	Type	Created by	Used by
.mas	Assembly code source	plain text	MarieEditor, or any plain text editor	MarieEditor and Assembler
.lst	Assembly listing file	plain text	Assembler	MarieEditor (TextFileViewer)
.map	Symbol table	plain text	Assembler	MarieSim (TextFileViewer)
.mex	Executable code	serialized Java object	Assembler	MarieSim
.dmp	Core dump	plain text	MarieSim	MarieSim (TextFileViewer)

Comments, Bugs, and Suggestions

The authors of your text and this simulator invite you to send us comments and suggestions. We would also like to correct any bugs that you find. You may send your findings and observations to: ECOA@jbpub.com

If you are reporting a bug, please supply as much information as you can with regard to what the simulator was doing when the problem occurred. Also, attach the .mas file, if the error occurred during program execution. Your comments will enable us to improve this simulator so that all students can have a positive learning experience.