

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. Your authors 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.

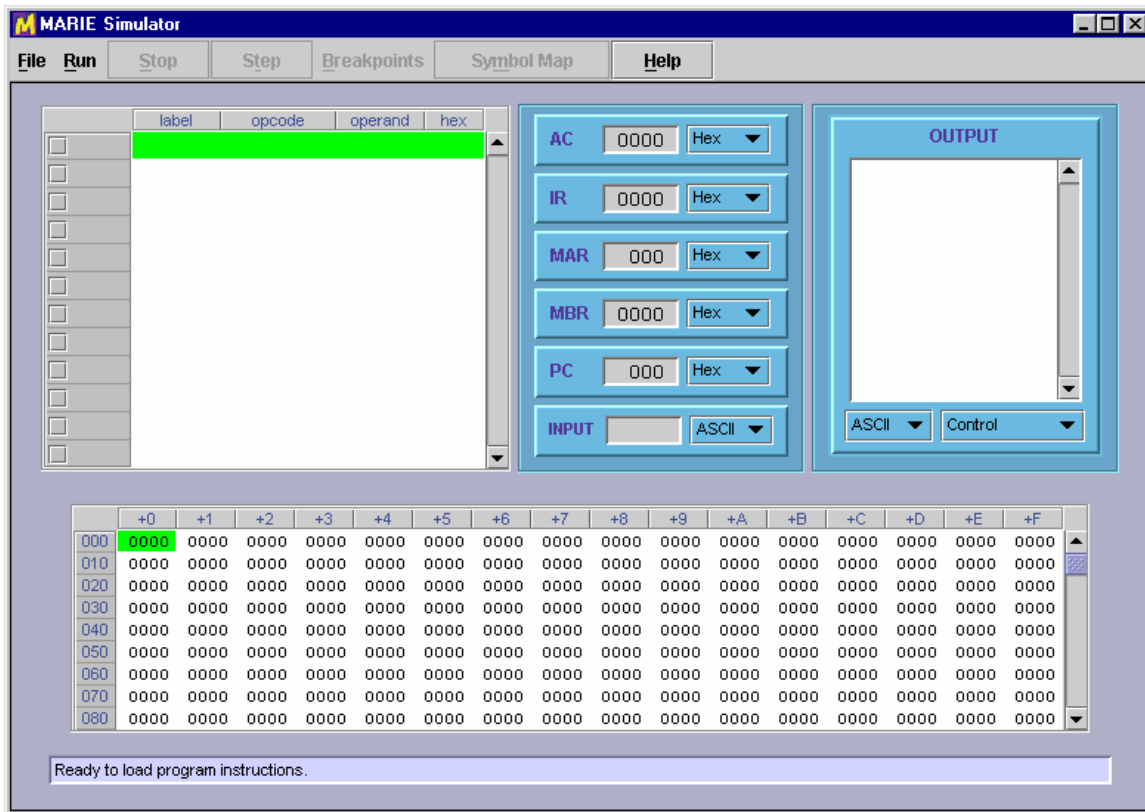


Figure 1: The MarieSim Graphical Environment

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.

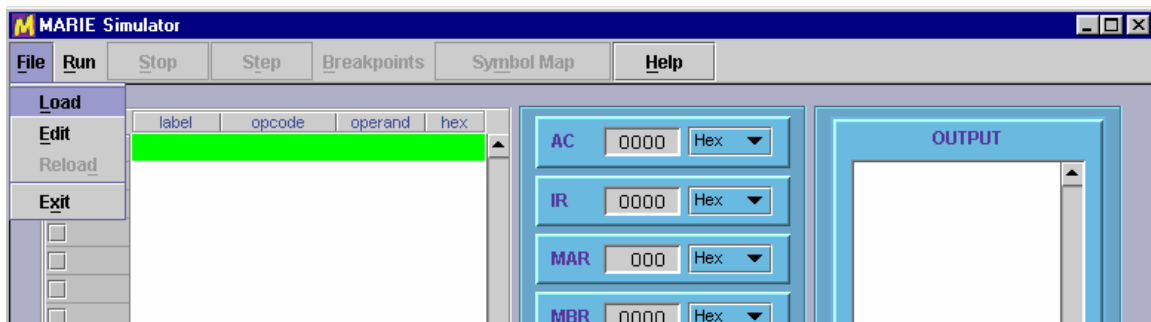


Figure 2: MarieSim File Menu Options

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.

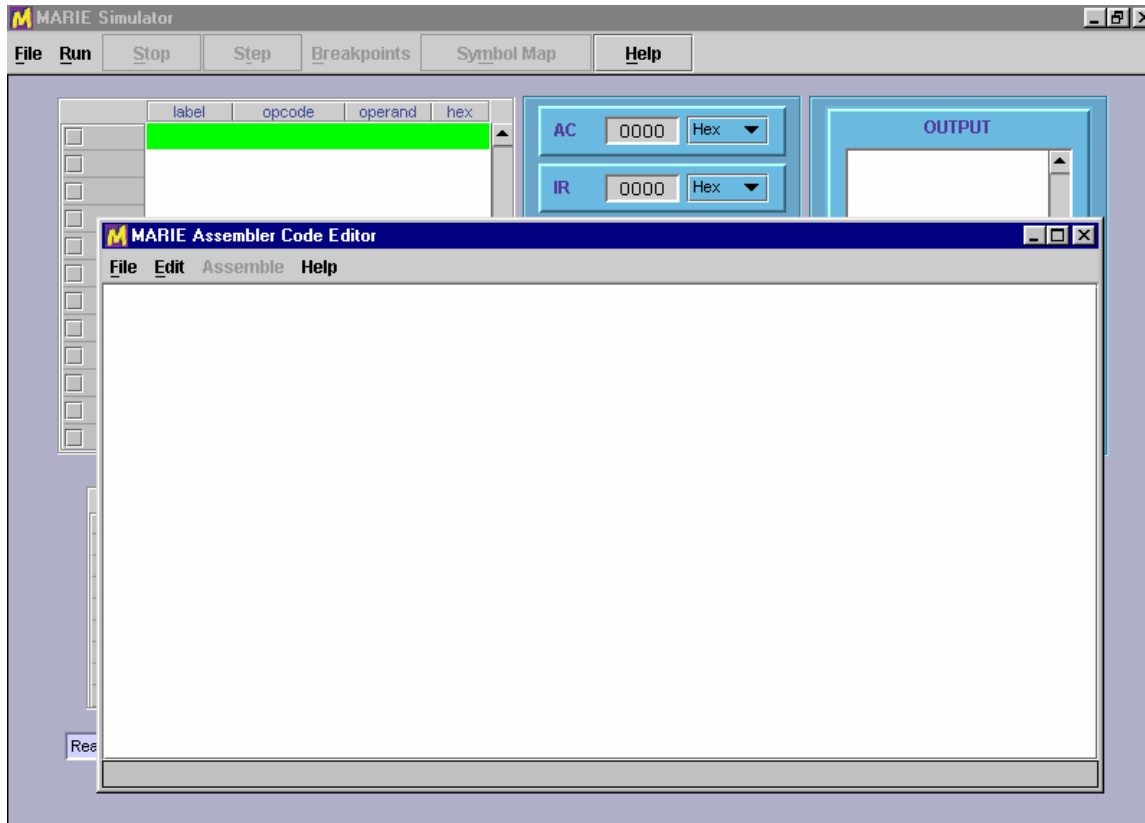


Figure 3: The MarieSim Editor

MARIE assembly code source files must have a ".mas" extension, for MARIE Assembler. Both the editor and the assembler recognize files of this type. Once you have saved a file with a ".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 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.

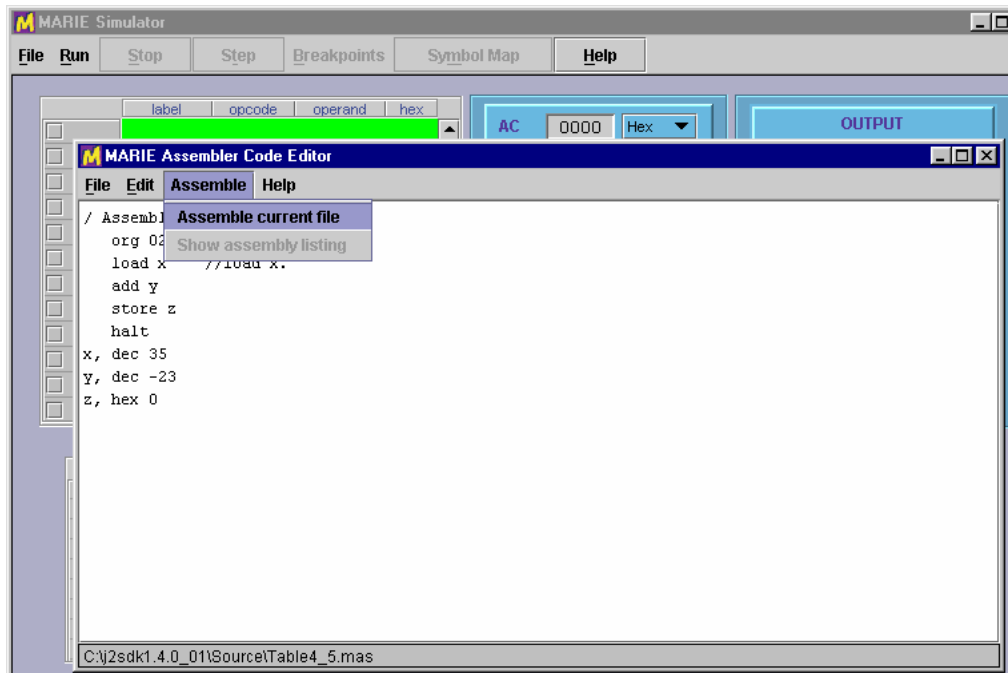


Figure 4: Preparing to Assemble Source Code

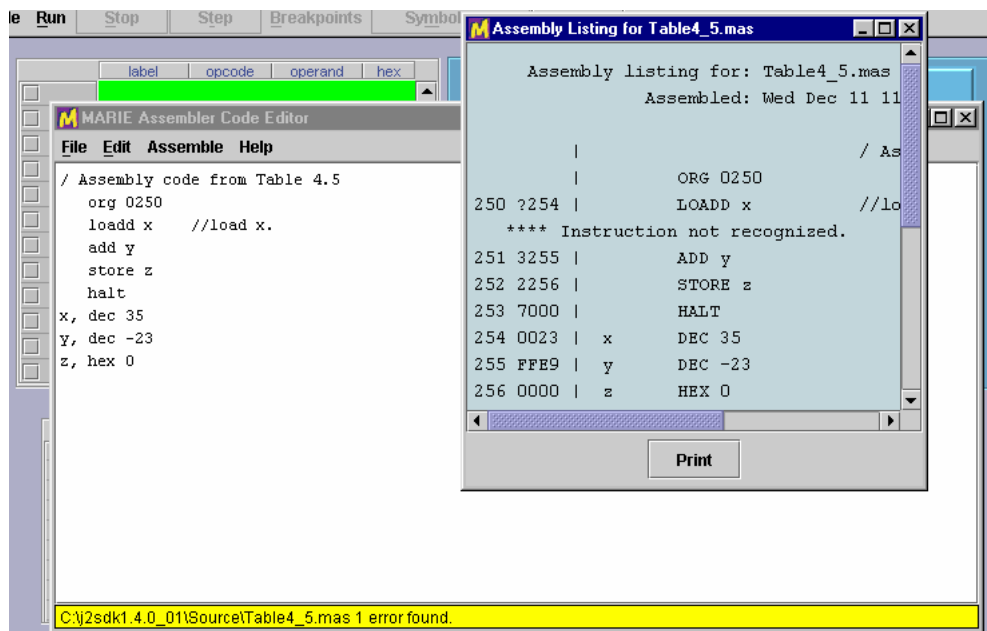


Figure 5: An Unsuccessful Assembly

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.

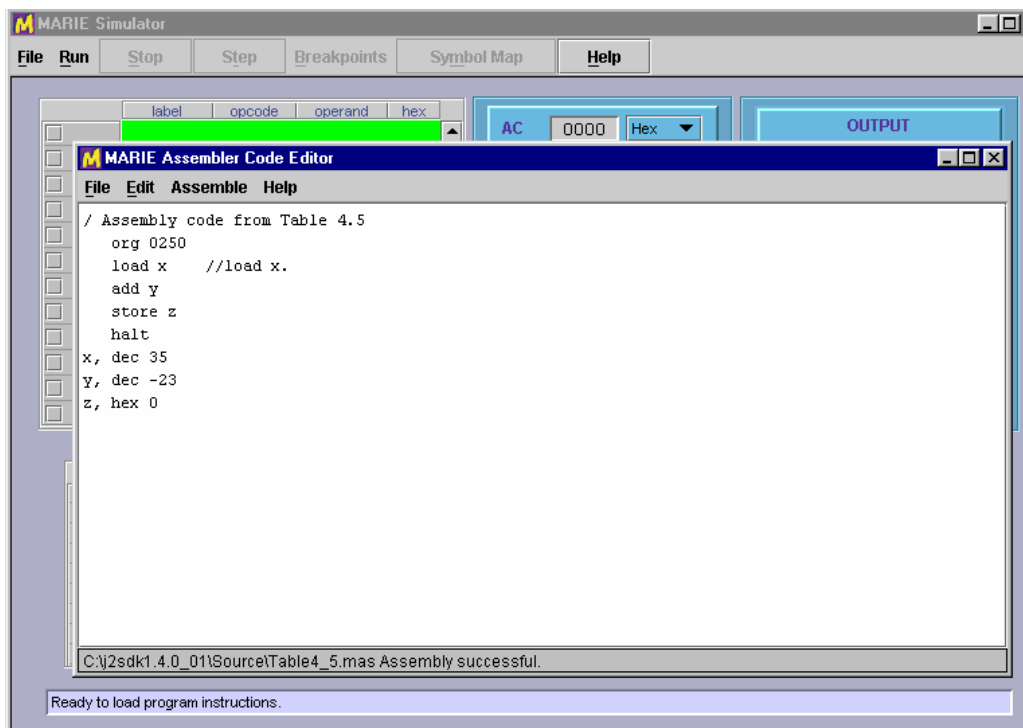


Figure 6: A Successful Assembly

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.*

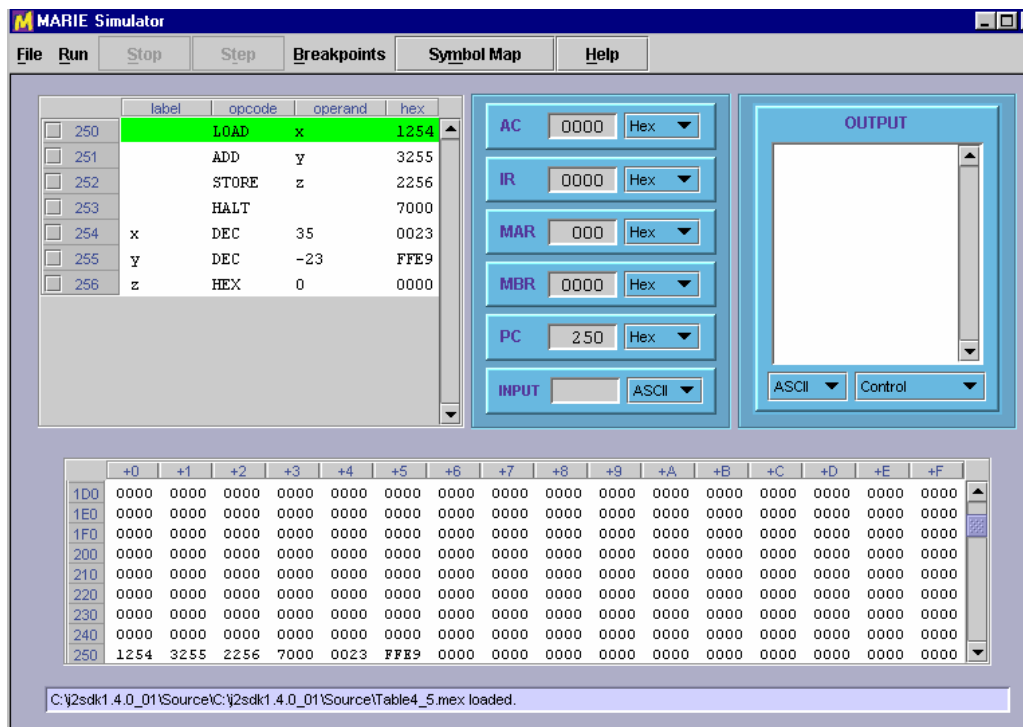


Figure 7: A Program Ready to Run

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.

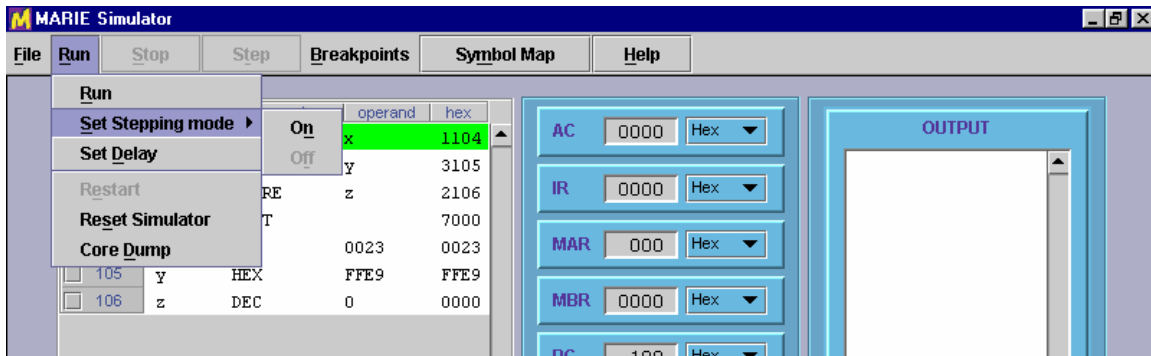


Figure 8: The Run Menu

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.

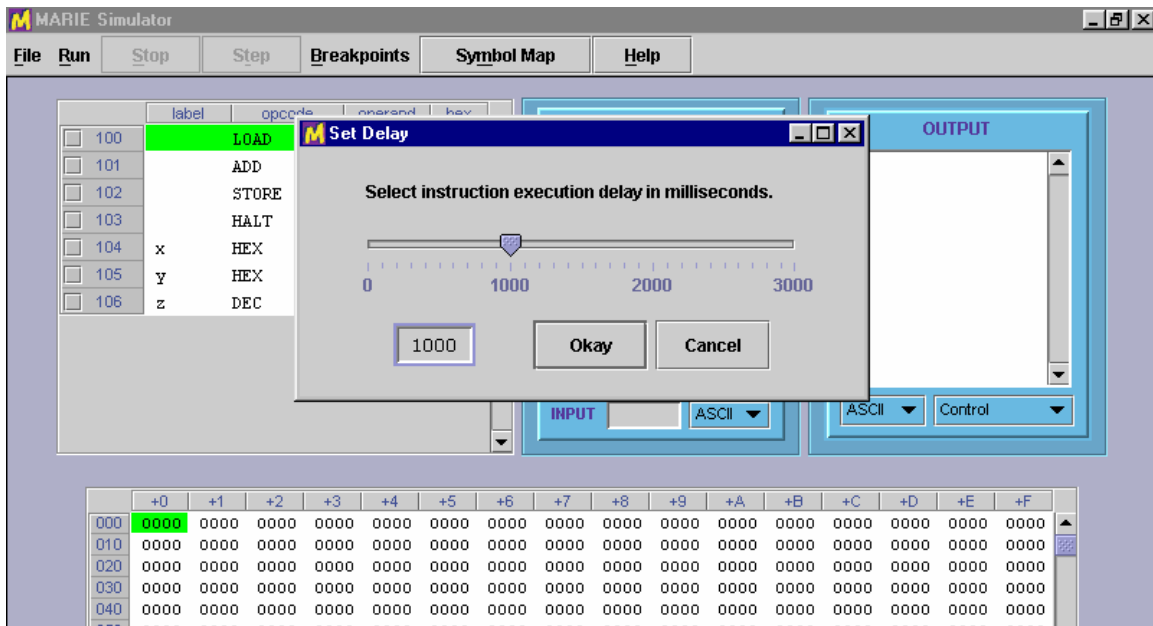


Figure 9: Setting Execution Delay

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.

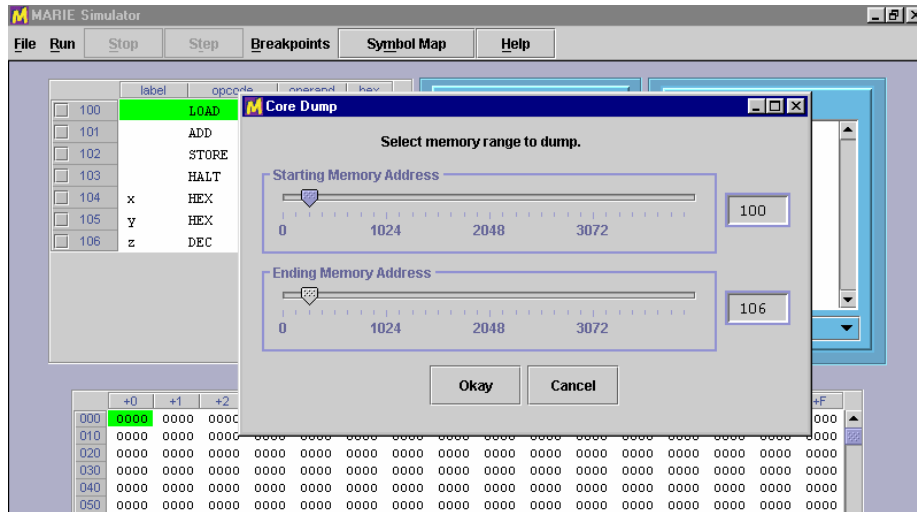


Figure 10: Setting Memory Core Dump Range

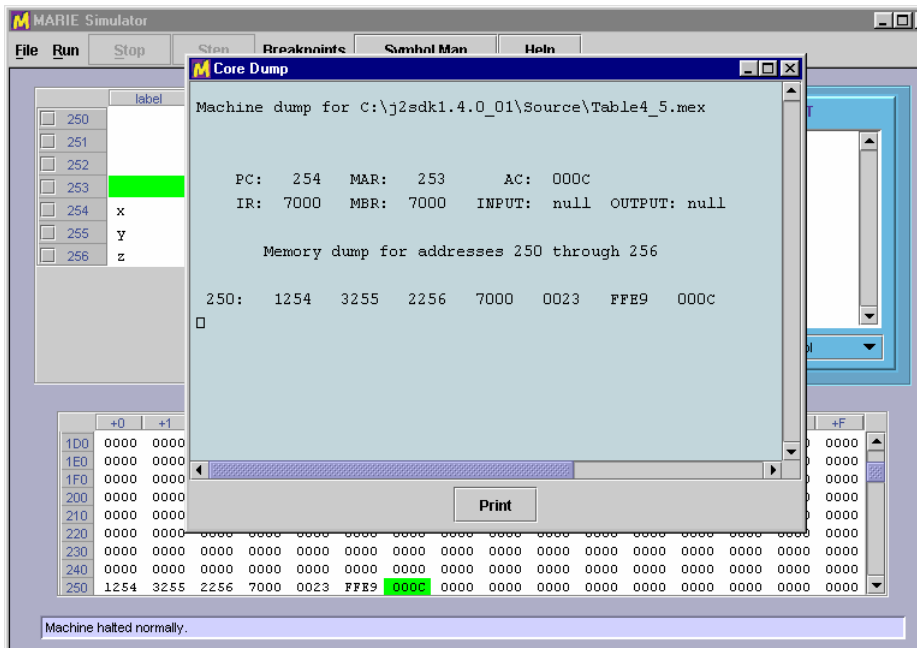


Figure 11: A MARIE Core Dump

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.

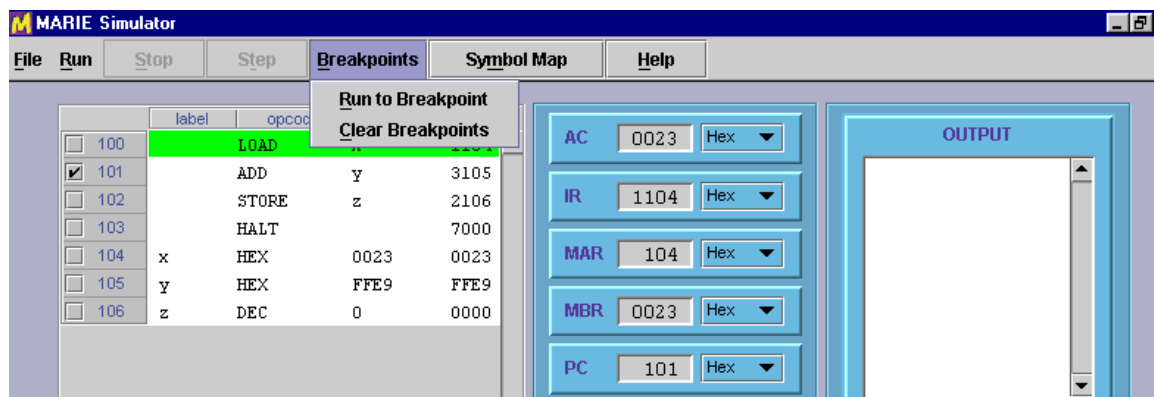


Figure 12: Breakpoint set at 101h and the Breakpoint Menu

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.

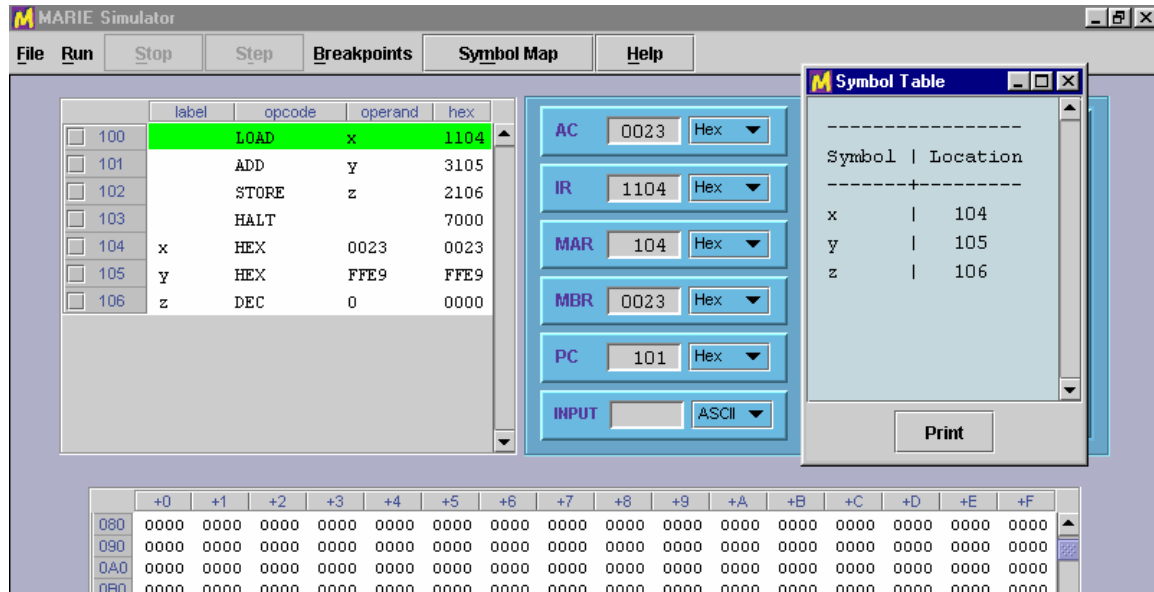


Figure 13: Viewing a Symbol Table

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,

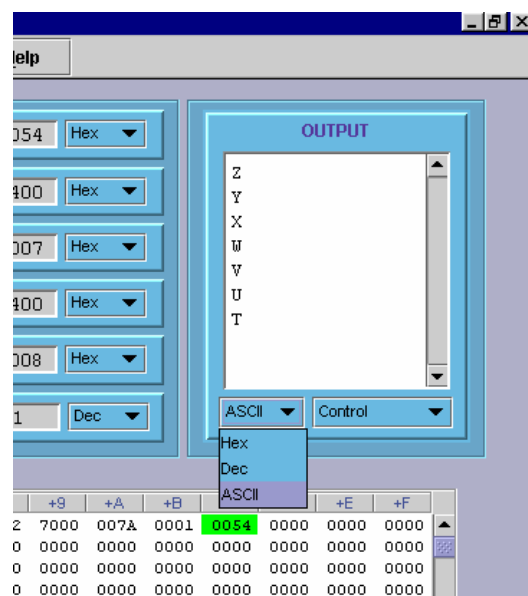


Figure 14: Setting a Register's Mode

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.

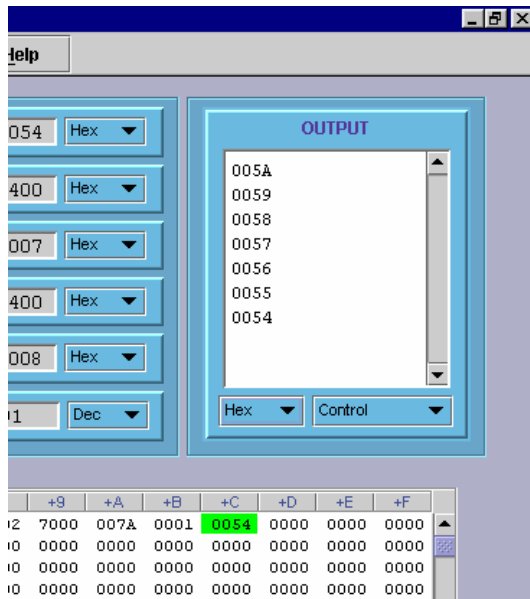


Figure 15: Reformatted Output

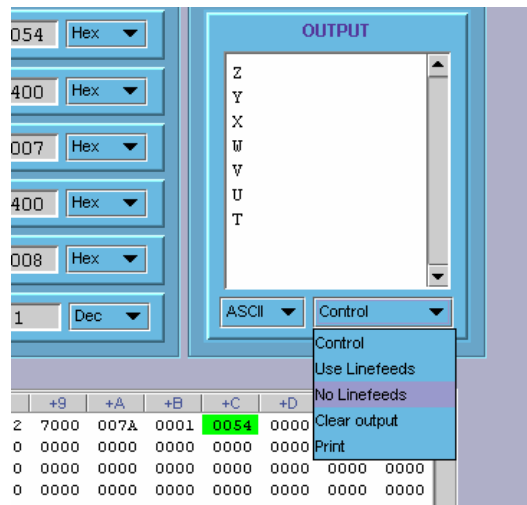
If you select the No Linefeeds option, as shown in Figure 16, your output will be placed horizontally in the output pane, from left to right—with no automatic wrapping. When the No Linefeeds option is turned on, you will need to programmatically provide your own linefeeds order to advance the output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

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.

Figure 16: Setting Linefeeds



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: ECOJA@jpub.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.