

4 Larc Machine Programming

This section describes how to write, run, and debug Larc machine programs.

4.1 Writing Programs

As described in the previous section, a Larc program consists of several lines with each line containing 16 0's and 1's. If we want to execute a program on a Larc machine, then it must be in the format described in the previous section. However, it is difficult for most humans to write programs using a binary representation (in fact, it is difficult for many humans to program at all). It is often easier to write programs using a textual representation and then convert this representation into binary once the program is completed. The textual representation we will use is known as assembly language and is the topic of the next section. We will have much more to say about assembly language in the following section, but for now, let's take a brief look at how to write instructions and data values in a text-based format, as well as how to document programs.

Table 5 shows the 16 Larc instructions and how they are written in assembly language. The format of an instruction is **<opname> <operands>** where **<opname>** is replaced with an operation name (e.g., **add**, **sub**) and **<operands>** is replaced with a list of operands, which include registers and immediates. The operands as well as the operation name are separated by either spaces or tabs. Registers in assembly language are written using the format: **\$<id>** where **<id>** is replaced with an identifier from 0 to 15. The **\$** is a special symbol used to identify registers. Immediates are written as decimal numbers without a leading **\$**.

Data values can also be expressed in a textual form. The format is **.word <value>** where **<value>** is replaced with a decimal value. (Note: there are other ways to express data values in assembly language, but we will defer looking at these until the next section.) For example, the value -10 would be written as **.word -10** rather than **111111111110110**.

It is also convenient to be able to comment the textual program, which we cannot do in the binary version. Comments will start with the '#' character and will continue until the end of the current line. These characters are not a part of the actual program. In addition, arbitrary spacing, tabbing, and newlines can be used as the programmer sees fit. Like comments, these are not a part of the program.

Figure 8(a) shows the "Hello, world!" program written in assembly language. Note that the

Type	Operation	Format	Semantics
ALU	addition	add \$a \$b \$c	Reg[a] = Reg[b] + Reg[c]
	subtraction	sub \$a \$b \$c	Reg[a] = Reg[b] - Reg[c]
	multiplication	mul \$a \$b \$c	Reg[a] = Reg[b] * Reg[c]
	division	div \$a \$b \$c	Reg[a] = Reg[b] / Reg[c]
	shift left logical	sll \$a \$b \$c	Reg[a] = Reg[b] << Reg[c]
	shift right logical	srl \$a \$b \$c	Reg[a] = Reg[b] >> Reg[c]
	bitwise AND	and \$a \$b \$c	Reg[a] = Reg[b] & Reg[c]
	bitwise NOR	nor \$a \$b \$c	Reg[a] = ~(Reg[b] Reg[c])
Long immediate	load immediate	li \$a imm	Reg[a] = sext(limm)
	load upper immediate	lui \$a imm	Reg[a] = limm << 8
	branch equal to 0	beqz \$a imm	if (Reg[a] == 0) PC=PC+sext(limm)
	branch less than 0	bltz \$a imm	if (Reg[a] < 0) PC=PC+sext(limm)
Short immediate	memory load	load \$a simm(\$b)	Reg[a] = Mem[Reg[b]+sext(simm)]
	memory store	store \$a simm(\$b)	Mem[Reg[b]+sext(simm)] = Reg[a]
Jump	jump and link register	jalr \$a \$b	old_pc=PC, PC=Reg[b], Reg[a]=old_pc+1
System call	system call	syscall	perform system call (type in Reg[1])

Table 5: Larc (base) assembly instructions. The operators in the semantics column are expressed using C syntax. \$a, \$b, and \$c are register identifiers (\$0-\$15). limm and simm refer to a long immediate and a short immediate, respectively (limm ranges from -128 to 127, simm ranges from -8 to 7). sext() is a function for sign extending a short or long immediate into a 16-bit value.

markers **.text** and **.data** indicate the start of the code and data sections, respectively. Once the program is written in assembly language, we can then convert each line into the corresponding machine language program as shown in Figure 8(b).

In general, it is easier to write programs in a form such as assembly language, and then translate the program into machine language once it is completed. As we will see in Section 5, we can write a translation tool called an assembler that automatically translates an assembly program into a machine program. With an assembler, we can avoid writing machine language programs altogether.

4.2 Running Programs

As described in Section 3, a functional simulator is used to run Larc programs. The simulator will simulate the input/output behavior, such as keyboard and monitor events, that occur when running the program on a real machine. It will not simulate the timing of the machine, *i.e.*, how long it takes to execute a program. Figure 9(a) shows the simulation of the “Hello, world!” program using the Larc simulator, which is run with the shell.

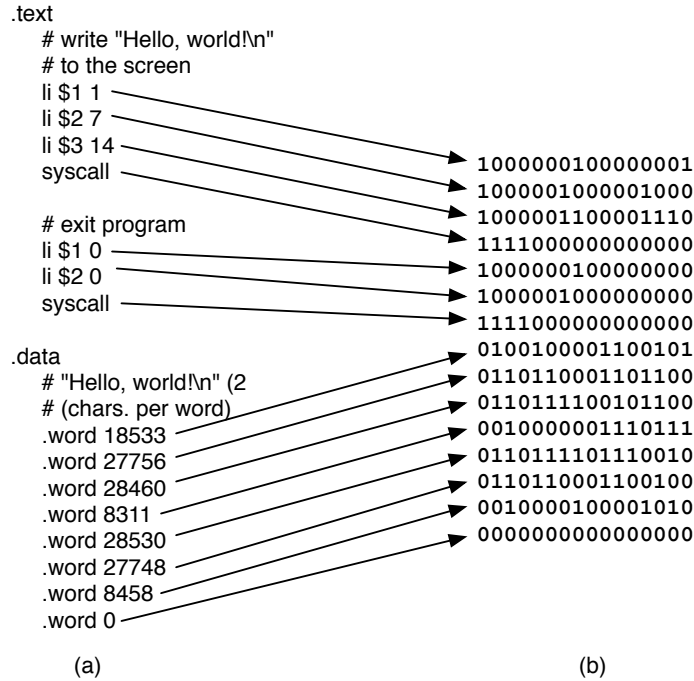


Figure 8: The “Hello, world!” program written in (a) assembly language and (b) machine language.

For simple programs such as the “Hello, world!” program, the simulator will simulate the program in very little time. However, for other programs that may not be the case. Simulation is slow compared to executing a program on real machine implemented in hardware. Most simulators take orders of magnitude more time to execute a program than a real machine. However, the performance of the simulator can be improved by using higher-performance languages, compilers, and algorithms (although it will still be much slower than a real machine). For this reason, the Larc toolset provides two simulators. The first, which must be completed by the student in a lab assignment, is written in Java and run using the Java Virtual Machine. As you might expect, this simulator is fairly slow (if you think about it, there is a simulator running in a simulator). The second, which the student does not write, is written in C and compiled to run natively. Aggressive optimization is enabled during compilation of the simulator. As a result, it is much faster than the Java simulator. For simple, small programs, the Java simulator will suffice, but when running more complex, longer-running programs, the C simulator should be used. Figure 9(b) shows the simulation of the “Hello, world!” program using the fast C simulator. The output is obviously identical as with the Java simulator, but this simulator runs much faster.

There are a few differences between a simulator (both fast and slow) and a real machine. First,

```
bash$ ./sim hello-world.out
Hello, world!
bash$
```

(a)

```
bash$ ./sim-fast hello-world.out
Hello, world!
bash$
```

(b)

Figure 9: Running the “Hello, world!” program using the (a) standard, Java simulator and (b) the faster, C simulator.

the Larc simulator is not always listening for keyboard input as in a real machine. Instead, the simulator will wait for input only when the program performs a read system call or the program reads the keyboard control register (**kbc**r). In the former case, the simulator reads the string from the user and copies it to memory as directed by the system call (recall that system calls are performed atomically by the simulator). In the latter case, the simulator reads a string from the user when **kbc**r is first read by the program. The simulator will then simulate the I/O delay by waiting a random amount of time before setting the ready bit in **kbc**r. When the simulator sets the ready bit it will also copy the first character into **kbc**r. The simulator continues this process until the entire string has been read by the program.

Similarly, the program will also simulate I/O delay with regard to the display monitor. The simulator will set the ready bit in **dcr** only after waiting a random amount of time since the last character was printed to the screen. At that point, the simulator will set the ready bit in **dcr** and simultaneously print the character in **ddr** to the screen.

The Larc simulator can be used to run and debug Larc machine programs. It will catch and identify all syntactic bugs. For example, if a line of the program file does not contain exactly 16 0’s and 1’s, the simulator will print out an error message. The simulator also catches and identifies many semantic bugs. For example, if the Larc program attempts to divide by zero, this bug will be caught and reported by the simulator. (When the student implements their own simulator, they should try to catch and identify as many bugs as possible.) Still, for more subtle bugs, the simulator is less helpful. A debugger is provided for these cases.

4.3 Debugging Programs

The Larc toolset includes a debugger for identifying and fixing errors within a Larc machine program (it can also be used for assembly programs as discussed in Section 6). The Larc debugger effectively allows the student to run a Larc machine program, an instruction at a time and watch

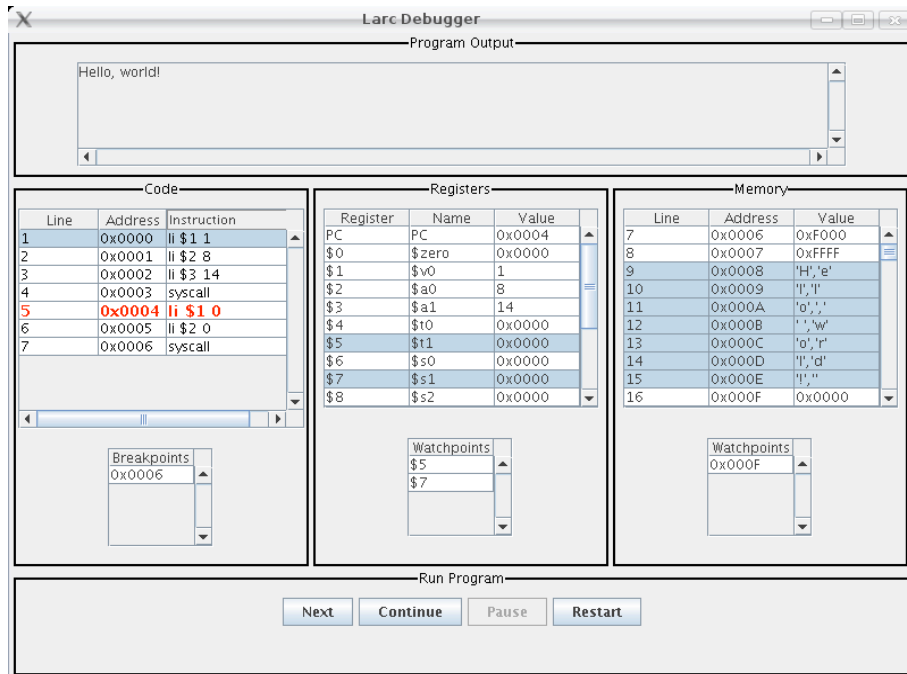


Figure 10: A screenshot of the Larc debugger running the “Hello, world!” program.

how the contents of the PC, registers, and memory change. The debugger is a graphical application, written in Java. It is slower than the Java simulator, so it should probably only be used when testing.

To run the debugger on the “Hello, world!” program, the student would type “./db hello-world.out” into the shell. “./db” is a shell script for running the Java debugger and “hello-world.out” is the program to be debugged. This command opens a new window, which will allow the student to run the program and watch the machine state change as the program is run. The Larc debugger can only run a single program at a time. To run a different program, the student would need to rerun “./db” using a different program name.

Figure 10 shows a screenshot of the debugger. The debugger window consists of five panels. The top panel displays the program input and output as it would appear in the shell. The middle three panels display the code (instructions), registers, and memory. The bottom panel is a panel for running the program. It contains four buttons for running or stopping the program. Below the buttons is also an area where information and error messages are displayed.

Running the program. The program is not initially run when the debugger is first started. To run the program, the user can push either the “next” or “continue” buttons. The “next” button will execute the next instruction (the first one), stop, and wait for further input from the user. The

“continue” button will execute the entire program. However, it can be stopped by pushing the the “pause” button. This will stop the program at the current instruction and allow the user to inspect the machine state. The program can also be paused without the button if the program requests keyboard input via a system call. In this case, the program is paused and a message is displayed on the button of the run panel to alert the user that they must provide keyboard input. If the user enters some input and does not click the “pause” button, then the program will continue running until the “pause” button is entered, keyboard input is requested again, or the program exits. A fourth button, “restart”, allows the programmer to restart the program from the beginning.

Inspecting the state of the machine. Whenever the program has been paused, either with the “pause” button, after an instruction is executed using the “next” button, or while the debugger is waiting for keyboard input, the user can inspect the state of the machine. The state is shown in the upper table of the three middle panels (the bottom table in each panel is discussed later). On the far left, the code panel shows the instructions in the program. It highlights the current instruction. For each instruction, the debugger displays its line in the program file, its address in memory, and the assembly representation of the instruction.

The careful reader might be wondering how the debugger knows where the instructions reside within the program file. The debugger knows the first line of the program is the first instruction. It looks for a special marker to determine the end of the text segment. The marker is all 1’s, which is not a valid instruction. If no marker is found, then each line in the program file will be interpreted as an instruction, even if it is actually a data value. So to debug a Larc machine program, the user should make sure that the text segment is only contained in the first part of the program file and that a marker of all 1’s indicates the start of the static data section. Note that the debugger will not work accurately for programs that do not follow the conventions discussed in Section 3.6. For example, the debugger will incorrectly display the instructions if the program generates some of them dynamically (as in a Just-In-Time compiler).

The far right panel, shows the contents of memory. These include instructions as well as data values. For each word in memory, the debugger shows the line from the program file corresponding to that memory word (“N/A” if there is no such line), the address of the word, and the value of the word. The value is initially shown in hexadecimal although the value can also be viewed as a signed int, unsigned int, or as two ASCII characters. To change the value format, right click inside the memory panel, click on “Format of values”, and select the appropriate format. Note: the format

cannot be changed for instructions in the code panel, although each instruction is also contained in memory and this value can be viewed with a given format.

Memory values can also be modified by clicking inside the value field for a particular memory word. The current format of the value must be followed when modifying the value. For example, if the memory word at address 2 is “0x0100”, *i.e.*, the format is hexadecimal, then it cannot be set to “5”. It can be set to “0xA2B1”, however. Modifying the value will impact the behavior of the program if that value is used in the future, so use care when modifying values. But this feature can be helpful, to see if modifying a particular value, does in fact eliminate the error. The line or address fields cannot be modified. Note: the instructions cannot be directly modified in the code panel, however, the corresponding value in memory can be modified within the memory panel.

The middle panel shows the contents of registers. Notice that the PC is also included in the list of registers. For each register, the identifier (*e.g.*, “\$0”), name (*e.g.*, “\$zero”), and value is shown (register names are discussed in Section 5, they can be ignored for now). Like a memory value, the value of a register can be displayed in hexadecimal, signed decimal, unsigned decimal, and as two ASCII characters. Also like a memory value, a register value can be modified by clicking within the value field and modifying the contents, following the current format.

Breakpoints and watchpoints. It is often helpful to be able to pause the program whenever a particular instruction is executed. For example, the user might want to pause the debugger every time the first instruction within some loop is executed. This feature is called a breakpoint and is supported in the Larc debugger. The lower table within the code panel lists memory addresses of the current breakpoints. To add a breakpoint, the user can right click within the code panel, select “Add/remove breakpoints”, and then select “Add selected breakpoints”. A breakpoint is added for each instruction that has been selected by the mouse (note: the user can use the control and shift buttons to select more than one instruction at a time). To remove breakpoints, the user can right click within the code panel, select “Add/remove breakpoints”, and then select either “Remove selected breakpoints” or “Clear breakpoints”. “Remove selected breakpoints” removes each breakpoint that is selected within the breakpoint table. “Clear breakpoints” removes all breakpoints within the breakpoint table.

Another helpful feature, which the Larc debugger provides, is watchpoints. These also pause the program when a particular event occurs. However, watchpoints stop the program when a particular data value is written rather than when an instruction is executed. The user can watch

values in either a register or in memory. For example, the user could set a watchpoint to stop the program whenever register 5 is written. Often, in programming at the machine language level, programmers will find that a particular region of memory is unintentionally getting corrupted by the program. Watchpoints allow the programmer to find the instruction or instructions that are causing this corruption.

Adding and removing watchpoints works similarly to adding and removing breakpoints. The only difference is that to set a register watchpoint the user must right click within the register panel and to select a memory watchpoint the user must right click within the memory panel.