Guide to Using the Unix version of the LC-3 Simulator

by

Kathy Buchheit
The University of Texas at Austin

Guide to Using the Unix version of the LC-3 Simulator

The LC-3 is a piece of hardware, so you might be wondering why we need a simulator. The reason is that the LC-3 doesn't actually exist (though it might one day). Right now it's just a plan – an ISA and a microarchitecture which would implement that ISA. The simulator lets us watch what would happen in the registers and memory of a "real" LC-3 during the execution of a program.

How this guide is arranged

For those of you who like to dive in and try things out right away, the first section walks you through entering your first program, in machine language, into a text editor (we'll use emacs as an example). You'll also find information about writing assembly language programs, but you'll probably skip that part until you've learned the LC-3 assembly language.

The second section gives you a quick introduction to the simulator's interface, and the third shows you how to use the simulator to watch the effects of the program you just wrote.

The fourth section takes you through a couple of examples of debugging in the simulator.

The last section is meant as reference material for the simulator.

In other words,

Chapter 1
Creating a program for the simulator

This example is also in the textbook, *Introduction to Computing Systems: From Bits and Gates to C and Beyond!* You'll find it in Chapter 6, beginning on page 166. The main difference here is that we're going to examine the program with the error of line x3003 corrected. We'll get to a debugging example once we've seen the "right way" to do things.

The Problem Statement
Our goal is to take the ten numbers which are stored in memory locations x3100 through x3109, and add them together, leaving the result in register 1.

Using emacs
If you know how to use Unix already, and you have a favorite text editor, go ahead and open it, and use it to write and save your program. If you don't have a preferred text editor, try using emacs.

At this point, you should be logged in to a Unix machine, and have a console window open. (I'm assuming basic Unix familiarity, so ask another user in your lab if you don't know how to log in and open a console.)

At the console prompt, type

```
emacs addnums.bin &
```

If you want to name your program something different from "addnums.bin," you could replace that name with the one you like better. The last three letters, "bin," specify that you're going to type your program in binary. If you want to type it in hex instead (I'll explain that momentarily), you could say "addnums.hex." The "&" at the end of the line tells Unix that you want to open the emacs text-editing program in a separate window.

Entering your program in machine language
You have the option to type your program into emacs in one of three ways: binary, hex, or the LC-3 assembly language. Here's what our little program looks like in binary:

```
0011000000000000
0101001001100000
0101100100100000
0001100100101010
1110010011111100
0110011010000000
0001010010100001
0001001001000011
0001100100111111
0000001111111011
```

```
1111000000100101
```

When you type this into emacs, you'll probably be looking at a chart which tells you the format of each instruction, such as the one inside the back cover of the textbook. So it may be easier for you to read your own code if you leave spaces between the different sections of each instruction. Also, you may put a semicolon followed by a comment after any line of code, which will make it simpler for you to remember what you were trying to do. In that case your binary would look like this:

```
0011 0000 0000 0000        ;start the program at location x3000
0101 001 001 1 00000       ;clear R1, to be used for the running sum
0101 100 100 1 00000       ;clear R4, to be used as a counter
0001 100 100 1 01010       ;load R4 with #10, the number of times to add
1110 010 011111100         ;load the starting address of the data
0110 011 010 000000        ;load the next number to be added
0001 010 010 1 00001       ;increment the pointer
0001 001 001 0 00 011      ;add the next number to the running sum
0001 100 100 1 11111       ;decrement the counter
0000 001 111111011         ;do it again if the counter is not yet zero
1111 0000 00100101         ;halt
```

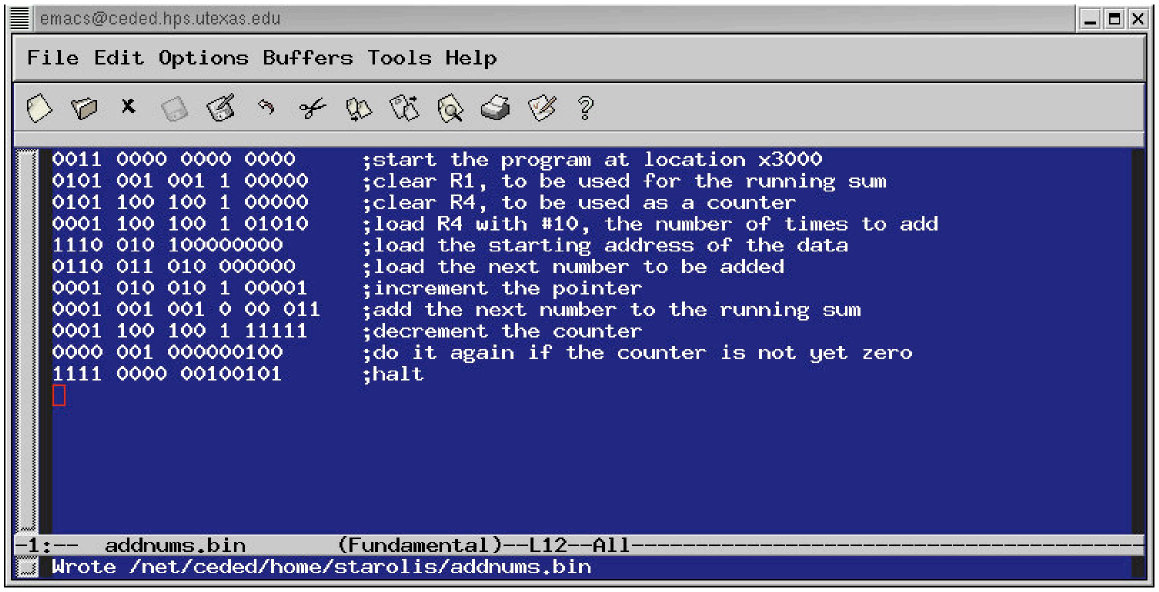
Either way is fine. The program which converts your program to machine language ignores spaces anyway. The second way will just be easier for you to read. Your program could also look like this, if you choose to type it in hex (notice that comments after a semicolon are still an option). In this case, you would have named your program "addnums.hex."

```
3000        ;start the program at location x3000
5260        ;clear R1, to be used for the running sum
5920        ;clear R4, to be used as a counter
192A        ;load R4 with #10, the number of times to add
E4FC        ;load the starting address of the data
6680        ;load the next number to be added
14A1        ;increment the pointer
1243        ;add the next number to the running sum
193F        ;decrement the counter
03FB        ;do it again if the counter is not yet zero
F025        ;halt
```

If you entered your program in binary, with spaces and comments, your emacs window will look something like this:

Saving your program

At the top of the emacs window, you'll see the word **Files**. Click on that now. Then click on **Save Buffer**. This will save your program under the name you gave when you


emacs@ceded.hps.utexas.edu
File Edit Options Buffers Tools Help

```
0011 0000 0000 0000      ;start the program at location x3000
0101 001 001 1 00000     ;clear R1, to be used for the running sum
0101 100 100 1 00000     ;clear R4, to be used as a counter
0001 100 100 1 01010     ;load R4 with #10, the number of times to add
1110 010 100000000       ;load the starting address of the data
0110 011 010 000000      ;load the next number to be added
0001 010 010 1 00001     ;increment the pointer
0001 001 001 0 00 011    ;add the next number to the running sum
0001 100 100 1 11111     ;decrement the counter
0000 001 000000100       ;do it again if the counter is not yet zero
1111 0000 00100101       ;halt

-1:--  addnums.bin       (Fundamental)--L12--All---------------------------------
Wrote /net/ceded/home/starolis/addnums.bin
```

started emacs a little while ago. If you want to save your program under a different name for some reason, click on **Files** and then **Save Buffer As…**. At the bottom of the emacs window, you'll see "Write file: ~/" and then a rectangular cursor. Type your new file name and press **Enter**.

Creating the .obj file for your program *

Before the simulator can run your program, you need to convert the program to a language that the LC-3 simulator can understand. The simulator doesn't understand the ASCII representations of hex or binary that you just typed into emacs. It only understands true binary, so you need to convert your program to actual binary, and save it in a file called *addnums.obj*.

If you saved your program in binary and called it "addnums.bin," go to the Unix prompt and type

```
lc3convert –b2 addnums.bin
```

If you saved your program in hex as "addnums.hex," type this at the Unix prompt:

```
lc3convert –b16 addnums.hex
```

When you type the appropriate line and press **Enter**, a new file will be created in the same directory where you saved your original *addnums* program. It will automatically have the same name, except that its file extension (the part of its name which comes after the ".") will be *.obj*.

If you typed your program in 1s and 0s, or in hex, only one new file will appear: *addnums.obj*.

If you don't know the LC-3 assembly language yet, now you're ready to skip ahead to Chapter 2, and learn about the simulator. Once you do learn the assembly language, a little bit later in the semester (or quarter), you can finish Chapter 1 and learn about the details of entering your program in a much more readable way.

Entering your program in the LC-3 assembly language

So you're partway through the semester, and you've been introduced to assembly language. Now entering your program is going to be quite a bit easier. This is what the program to add ten numbers could look like, making use of pseudo-ops, labels, and comments.

```
        .ORIG x3000
        AND   R1,R1,x0      ;clear R1, to be used for the running sum
        AND   R4,R4,x0      ;clear R4, to be used as a counter
        ADD   R4,R4,xA      ;load R4 with #10, the number of times to add
        LEA   R2,x0FC       ;load the starting address of the data
LOOP    LDR   R3,R2,x0      ;load the next number to be added
        ADD   R2,R2,x1      ;increment the pointer
        ADD   R1,R1,R3      ;add the next number to the running sum
        ADD   R4,R4,x-1     ;decrement the counter
        BRp   LOOP          ;do it again if the counter is not yet zero
        HALT
        .END
```

You still need to change your program to a .obj file, which is now called "assembling" your program. To do this, save your program as "addnums.asm." Then at the command prompt, type

```
lc3as addnums.asm
```

and notice that this time you didn't need to specify "addnums.obj" because your assembled file will automatically get the extension ".obj."

Since you used the fancier assembly language approach, you've been rewarded with not just one, but a couple of files:

addnums.obj, as you expected
addnums.sym, the symbol table created on the assembler's first pass

Let's take a look at the addnums.sym file.

addnums.sym

Here's what this file looks like if you open it in a text editor:

```
// Symbol table
// Scope level 0:
//     Symbol Name     Page Address
//     ----------------  ------------
//      LOOP              3004
```

You only had one label in your program: LOOP. So that's the only entry in the symbol table. 3004 is the address, or memory location, of the label LOOP. In other words, when the assembler was looking at each line one by one during the first pass, it got to the line

```
LOOP  LDR    R3,R2,x0         ;load the next number to be added
```

and saw the label "LOOP," and noticed that the Location Counter held the value x3004 right then, and put that single entry into the symbol table.

So on the second pass, whenever the assembler saw that label referred to, as in the statement

```
BRp    LOOP
```

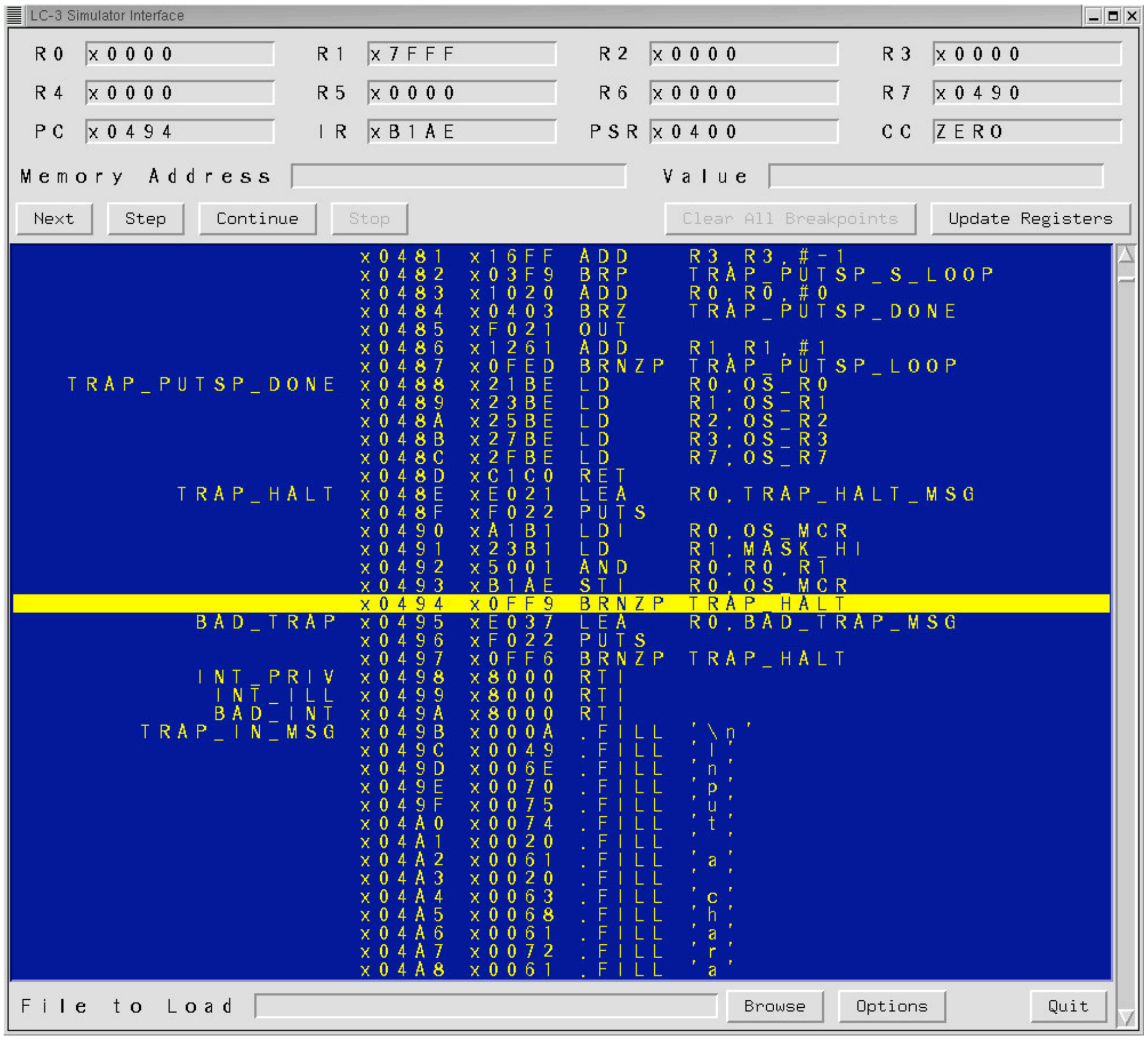
it replaced LOOP with the value that will yield x3004 when added to the PC, namely x1FB. If you'd had more labels in your program, they would have been listed under Symbol Name, and their locations would have been listed under Page Address.

Chapter 2
The simulator: what you see on the screen

To start the simulator, type this at the prompt:

```
lc3sim-tk &
```

When you launch the Unix version of the LC-3 Simulator, you see this:

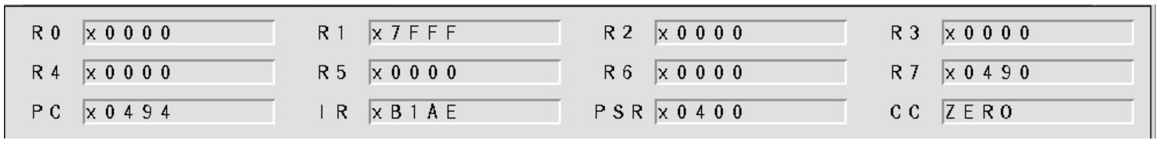


```
R0 x0000    R1 x7FFF    R2  x0000    R3 x0000
R4 x0000    R5 x0000    R6  x0000    R7 x0490
PC x0494    IR xB1AE    PSR x0400    CC ZERO

Memory Address                       Value

                   x0481  x16FF  ADD    R3,R3,#-1
                   x0482  x03F9  BRP    TRAP_PUTSP_S_LOOP
                   x0483  x1020  ADD    R0,R0,#0
                   x0484  x0403  BRZ    TRAP_PUTSP_DONE
                   x0485  xF021  OUT
                   x0486  x1261  ADD    R1,R1,#1
                   x0487  x0FED  BRNZP  TRAP_PUTSP_LOOP
 TRAP_PUTSP_DONE   x0488  x21BE  LD     R0,OS_R0
                   x0489  x23BE  LD     R1,OS_R1
                   x048A  x25BE  LD     R2,OS_R2
                   x048B  x27BE  LD     R3,OS_R3
                   x048C  x2FBE  LD     R7,OS_R7
                   x048D  xC1C0  RET
       TRAP_HALT   x048E  xE021  LEA    R0,TRAP_HALT_MSG
                   x048F  xF022  PUTS
                   x0490  xA1B1  LDI    R0,OS_MCR
                   x0491  x23B1  LD     R1,MASK_HI
                   x0492  x5001  AND    R0,R0,R1
                   x0493  xB1AE  STI    R0,OS_MCR
                   x0494  x0FF9  BRNZP  TRAP_HALT
        BAD_TRAP   x0495  xE037  LEA    R0,BAD_TRAP_MSG
                   x0496  xF022  PUTS
                   x0497  x0FF6  BRNZP  TRAP_HALT
        INT_PRIV   x0498  x8000  RTI
         INT_ILL   x0499  x8000  RTI
         BAD_INT   x049A  x8000  RTI
     TRAP_IN_MSG   x049B  x000A  .FILL  '\n'
                   x049C  x0049  .FILL  'I'
                   x049D  x006E  .FILL  'n'
                   x049E  x0070  .FILL  'p'
                   x049F  x0075  .FILL  'u'
                   x04A0  x0074  .FILL  't'
                   x04A1  x0020  .FILL  ' '
                   x04A2  x0061  .FILL  'a'
                   x04A3  x0020  .FILL  ' '
                   x04A4  x0063  .FILL  'c'
                   x04A5  x0068  .FILL  'h'
                   x04A6  x0061  .FILL  'a'
                   x04A7  x0072  .FILL  'r'
                   x04A8  x0061  .FILL  'a'

File to Load
```

Chapter 5 of this guide is a more complete reference to all the parts of this interface. If you want all the details, look there. If you want just enough details to be able to continue the step-by-step example, keep reading.

The registers

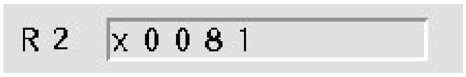
Below the heading CPU, notice the list of registers.



R0	x0000	R1	x7FFF	R2	x0000	R3	x0000
R4	x0000	R5	x0000	R6	x0000	R7	x0490
PC	x0494	IR	xB1AE	PSR	x0400	CC	ZERO

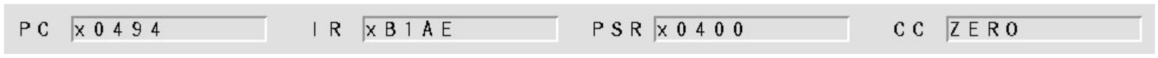
The General Purpose Registers, R0 through R7, are the eight registers that LC-3 instructions use as sources of data and destinations of results. The numbers following the "=" are the contents of those registers, first in hex and then (in parentheses) in decimal.

If, during the execution of a program, R2 contained the decimal value 129, you would see this:



R2 x0081

The Special Registers section shows the names and contents of five important registers in the LC-3 control unit. Those registers are the PC, the IR, and the N, Z, and P condition code registers.



PC	x0494	IR	xB1AE	PSR	x0400	CC	ZERO

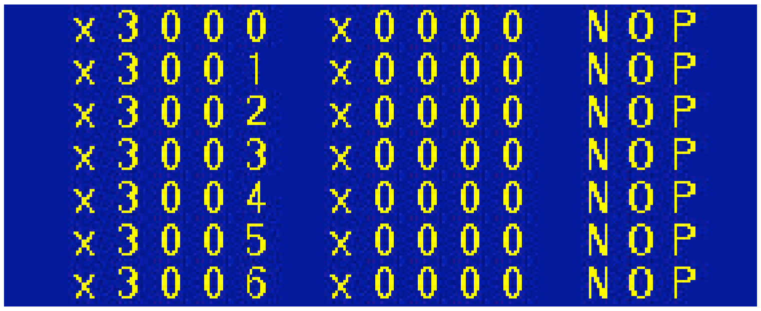
The PC, or program counter, points to the next instruction to be run. When you load your program, it will contain the address of your first instruction. The default value is x3000.

The IR, or instruction register, contains the value of the current instruction.

The CC, or condition codes, are set by certain instructions (ADD, AND, OR, LEA, LD, LDI, and LDR). They consist of three registers: N, Z, and P. Remember that only one of the three can have the value 1 at any time. The simulator shows us the values of the three registers all lumped together. The above situation, NZP = 001, means that N=0, Z=0, and P=1.

The memory

Below the registers, you see a long, dense list of numbers which begins like this:



```
x3000  x0000  NOP
x3001  x0000  NOP
x3002  x0000  NOP
x3003  x0000  NOP
x3004  x0000  NOP
x3005  x0000  NOP
x3006  x0000  NOP
```

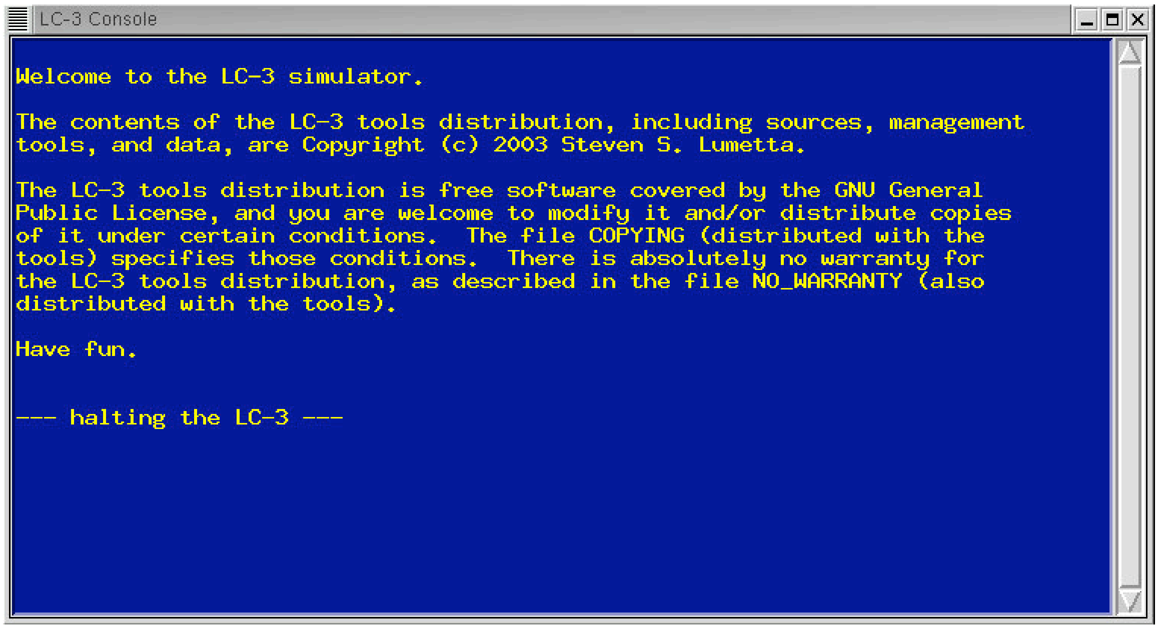
Use the scrollbar at the right (with the middle mouse button) to scroll up and down through the memory of the LC-3. Remember that the LC-3 has an address space of 2^{16}, or 65536 memory locations in all. That's a very long list to scroll through. You're likely to get lost. If you do, go to the **Memory Address** box and manually enter a memory location and press enter. You will then be taken directly to that memory location.

In the **From** field, type the memory location you want to start with, such as 3000. In the **To** field, type the memory location to display through, such as 3020. Then click **Print**, and those x20 locations will appear in the Memory part of the simulator.

The first column in the long list of memory locations tells you the address of the location. The second column tells you the contents of a location, in hex. The columns after the first two are the assembly language interpretation of the contents of a location. If a location contains an instruction, this assembly interpretation will be useful. If a location contains data, just ignore these columns entirely.

The Console Window

A second window also appears when you run the simulator. It is rather inconspicuous, and has the vague title "LC-3 Console." This window will give you messages such as "Halting the processor." If you use input and output routines in your program, you'll see your output and do your input in this window.



LC-3 Console

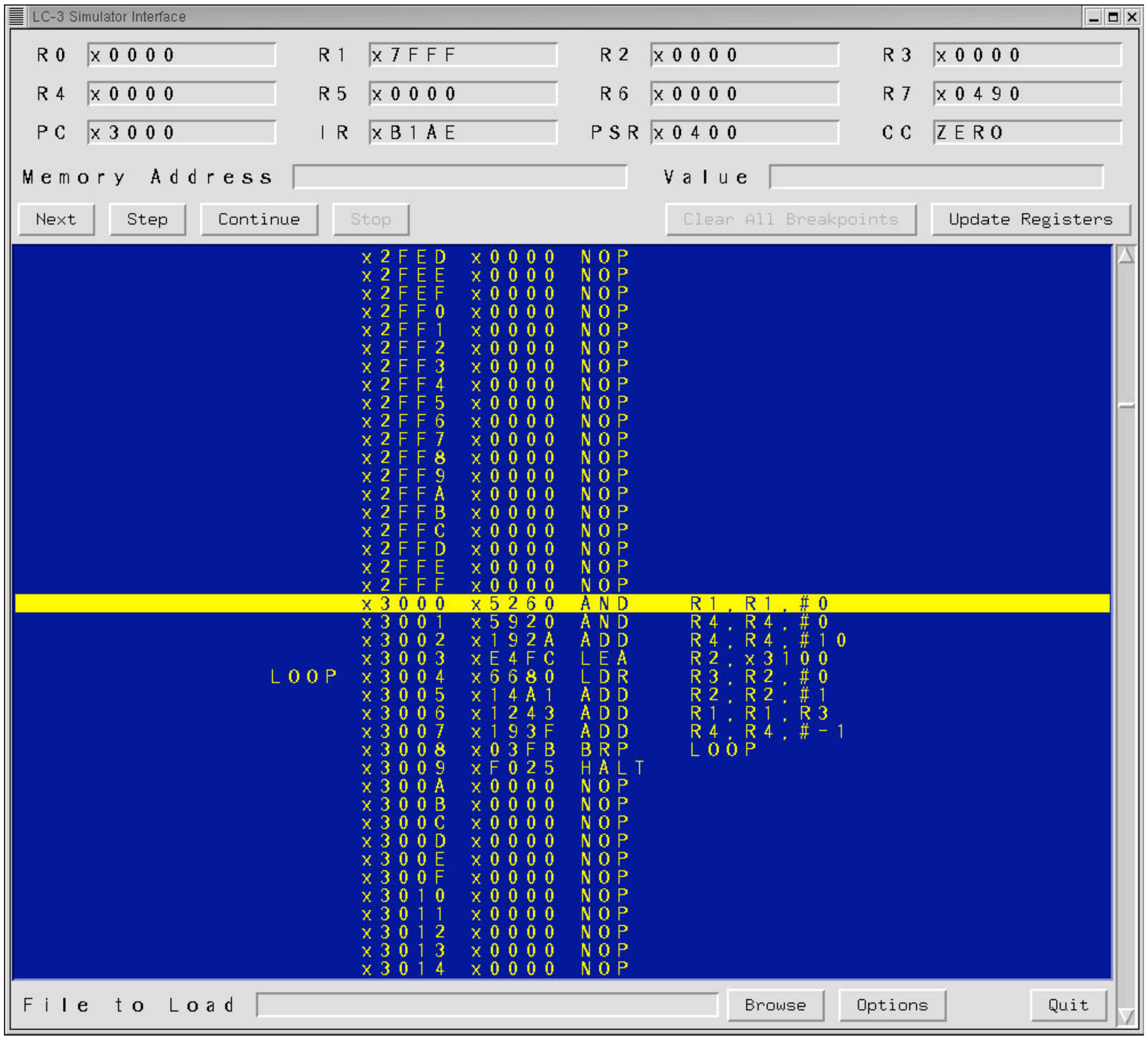
```
Welcome to the LC-3 simulator.

The contents of the LC-3 tools distribution, including sources, management
tools, and data, are Copyright (c) 2003 Steven S. Lumetta.

The LC-3 tools distribution is free software covered by the GNU General
Public License, and you are welcome to modify it and/or distribute copies
of it under certain conditions.  The file COPYING (distributed with the
tools) specifies those conditions.  There is absolutely no warranty for
the LC-3 tools distribution, as described in the file NO_WARRANTY (also
distributed with the tools).

Have fun.


--- halting the LC-3 ---
```

Chapter 3
Running a program in the simulator

Now you're ready to run your program in the simulator. Open the simulator by typing

```
lc3sim-tk &
```

at the prompt. To load your program, click **Browse** and then locate your program on disk. This is what you'll see when your program is loaded:



```
R0  x0000     R1  x7FFF     R2   x0000     R3  x0000
R4  x0000     R5  x0000     R6   x0000     R7  x0490
PC  x3000     IR  xB1AE     PSR  x0400     CC  ZERO
Memory Address                  Value

        x2FED  x0000  NOP
        x2FEE  x0000  NOP
        x2FEF  x0000  NOP
        x2FF0  x0000  NOP
        x2FF1  x0000  NOP
        x2FF2  x0000  NOP
        x2FF3  x0000  NOP
        x2FF4  x0000  NOP
        x2FF5  x0000  NOP
        x2FF6  x0000  NOP
        x2FF7  x0000  NOP
        x2FF8  x0000  NOP
        x2FF9  x0000  NOP
        x2FFA  x0000  NOP
        x2FFB  x0000  NOP
        x2FFC  x0000  NOP
        x2FFD  x0000  NOP
        x2FFE  x0000  NOP
        x2FFF  x0000  NOP
        x3000  x5260  AND   R1, R1, #0
        x3001  x5920  AND   R4, R4, #0
        x3002  x192A  ADD   R4, R4, #10
        x3003  xE4FC  LEA   R2, x3100
LOOP    x3004  x6680  LDR   R3, R2, #0
        x3005  x14A1  ADD   R2, R2, #1
        x3006  x1243  ADD   R1, R1, R3
        x3007  x193F  ADD   R4, R4, #-1
        x3008  x03FB  BRP   LOOP
        x3009  xF025  HALT
        x300A  x0000  NOP
        x300B  x0000  NOP
        x300C  x0000  NOP
        x300D  x0000  NOP
        x300E  x0000  NOP
        x300F  x0000  NOP
        x3010  x0000  NOP
        x3011  x0000  NOP
        x3012  x0000  NOP
        x3013  x0000  NOP
        x3014  x0000  NOP

File to Load
```

Notice that the first line of your program, no matter what format you originally used, is gone. That line specified where the program should be loaded in memory: x3000. Since nothing has happened yet (you haven't started running or stepping through your program), the PC is pointing to the first line of your program (x3000).

Loading the data (ten numbers) into memory

There are several ways to get the ten numbers that you're planning to add into the memory of the LC-3 simulator. You want them to begin at location x3100.

First way: Manually enter each data value in the memory location by left-clicking on the memory location and then entering a value in the **Value** field. Do this for all ten locations. This method is rather tedious! Do not dispair, there's another way to accomplish the same thing.

Second way: go back to emacs, and enter the data as code in hex. First start emacs at the command prompt:

```
emacs data.hex &
```

and then enter this in your file:

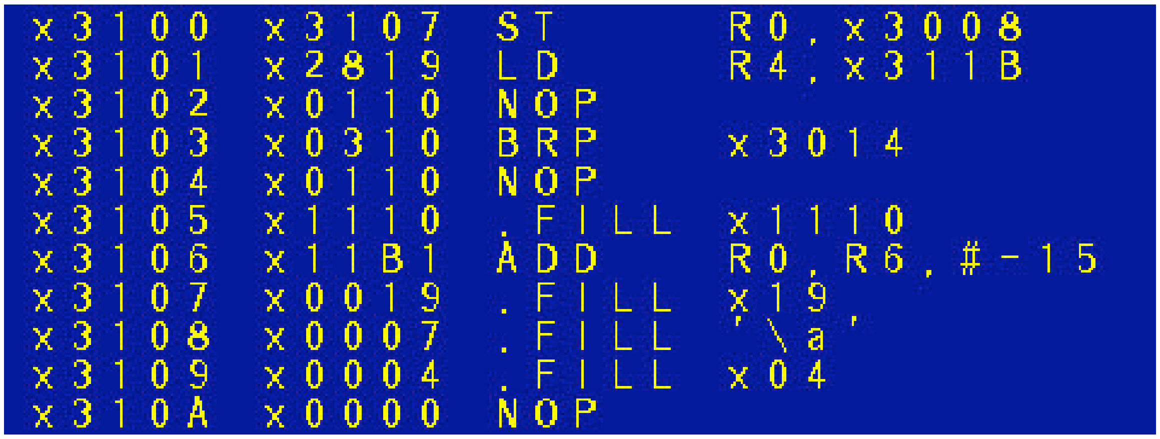
```
3100            ;data starts at memory location x3100
3107            ;the ten numbers we want to add begin here
2819
0110
0310
0110
1110
11B1
0019
0007
0004
```

Save this code as *data.hex* by clicking on **Files** and choosing **Save Buffer**. As usual, the first line is the address where we want the data to begin. The other lines are the actual data we want to load into memory. To convert your program to an .obj file, type this at the prompt:

```
convert –b16 data.hex
```

Now, a file called *data.obj* will exist wherever you saved your .hex file.

Now go back to the simulator, choose to load a program**…** once again, and select *data.obj*. Note that you can load multiple .obj files so they exist concurrently in the LC-3 simulator's memory. The memory locations starting with x3100 will look like this:



```
x3100  x3107  ST     R0, x3008
x3101  x2819  LD     R4, x311B
x3102  x0110  NOP
x3103  x0310  BRP    x3014
x3104  x0110  NOP
x3105  x1110  .FILL  x1110
x3106  x11B1  ADD    R0, R6, #-15
x3107  x0019  .FILL  x19
x3108  x0007  .FILL  '\a'
x3109  x0004  .FILL  x04
x310A  x0000  NOP
```

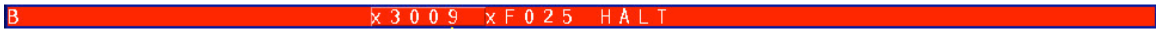
Now your data is in place, and you're ready to run your program.

Running your program

Enter memory address x3000 in **Memory Address** so that you may see your program. Be sure that **PC** contains x3000 as well so that the LC3 will know where to begin executing your program.

Now, we'll set a breakpoint. This will cause the LC-3 to stop when the PC reaches this address. Setting and unsetting breakpoints is very simple in the LC-3. All you have to do is double click on the memory location where you would like the breakpoint set. Double click on location x3009.

That sets a breakpoint on line x3009. If you don't follow this suggestion, you'll never see your result in R1, because we'll do the trap routine for HALT, which changes R1 before it halts the simulator. (I'll explain breakpoints in more detail in the next chapter.) After you set your breakpoint, the line will look like this:



```
B                       x3009  xF025  HALT
```

That B at the beginning of the line shows that you've set a breakpoint on that line. So we'll stop when we get to line x3009, before we execute the instruction there.

Now for the big moment: click on **Continue**. If you've already added up the ten numbers you put into the data section of your program, you know that x8135 is the answer to expect. That's what you should see in R1 when the program stops at the breakpoint.

Stepping through your program

So now that you've seen your program run, you know it works. But that doesn't give you a good sense for what's actually going on in the LC-3 during the execution of each instruction. It's much more interesting to step through the program line by line, and see what happens. You'll need to do this quite a bit to debug less perfect code, so let's try it.

First, you need to reset the very important program counter to the first location of your program. So set the PC back to x3000. (Enter x3000 in the PC window)

Click on **Next**. This will execute exactly one LC-3 instruction.

A couple of interesting things just happened:

- R1 got cleared. (If you "cleaned up" by clearing R1 before you started, this won't be an exciting event.)
- The IR has the value x5260. Look at the hex value of location x3000. That is also x5260. The IR holds the value of the "current" instruction. Since we finished the first instruction, and have not yet run the second, the first instruction is still the current one.

Click **Next** for a second time. Again, notice the new values for the PC and IR. The second instruction clears R4.

Click **Next** a third time. The PC and IR update once again, and now R4 holds the value x0a, which is decimal 10, the number of times we need to repeat our loop to add ten numbers. This is because the instruction which just executed added x000a to x0000, and put the result in R4.

Continue to step through your program, watching the results of each instruction, and making sure they are what you expect them to be.

At any point, if you "get the idea" and want your program to finish executing in a hurry, click on **Continue**, and that will cause your program to execute until it reaches the breakpoint you set on the Halt line.

So now you know how it feels to write a program perfectly the very first time, and see it run successfully. Savor this moment, because usually it's not so easy to attain. But maybe programming wouldn't be as fun if you always got it right immediately. So let's pretend we didn't. The next chapter will walk you through debugging some programs in the simulator.

Chapter 4
Debugging programs in the simulator

Now that you've experienced the ideal situation of seeing a program work perfectly the first time, you're ready for a more realistic challenge – realizing that a program has a problem, and trying to track down that problem and fix it.

Example:
Debugging the program to multiply without a multiply instruction

This example is taken from the textbook, and is discussed on pages 129 and 130. The program is supposed to multiply two positive numbers, and leave the result in R2.

Typing in the program

First you'll need to enter the program in LC3Edit. It should look like this:

```
0011 0010 0000 0000        ;the address where the program begins: x3200
0101 010 010 1 00000       ;clear R2
0001 010 010 0 00 100      ;add R4 to R2, put result in R2
0001 101 101 1 11111       ;subtract 1 from R5, put result in R5
0000 011 111111101         ;branch to location x3201 if the result is zero or positive
1111 0000 00100101         ;halt
```

As you can tell by studying this program, the contents of R4 and R5 will be "multiplied" by adding the value in R4 to itself some number of times, specified by the contents of R5. For instance, if R4 contains the value 7, and R5 contains the value 6, we want to add 0+7 the first time through, then 7+7 the second time through, then 14+7 the third time through, then …, then 35+7 the sixth time through, ending up with the value 42 in R2 when the program finishes.

Converting the program to .obj format

Once you've typed your program into emacs, save it as *multiply.bin* and then type this at the command prompt to convert it to an .obj file:

```
lc3convert –b2 multiply.bin
```

Loading the program into the simulator

Start the simulator by typing

```
lc3sim-tk &
```

at the prompt, load your object file.

Setting a breakpoint at the halt instruction

Breakpoints are extremely useful in many ways, and we'll get to a few of those soon. You should make it a habit to set a breakpoint on your "halt" line, because if you run your program without that, you'll end up in the halt subroutine, which will change some of your registers before halting the machine. So first, set a breakpoint on line x3204 by choosing the **Run** menu and then **Breakpoints…**. In the Breakpoints popup window, specify the line 3204, and click **Add**.

That B tells you that a breakpoint is set on that line, so if the program is running, and the PC gets the value x3204, the simulator will pause and wait for you to do something.

Running the buggy multiply program

Before you run your program for the first time, you need to put some values in R4 and R5 so that they'll be multiplied (or not, in this case!). How should you choose values to test? Common sense will help you here. 0 and 1 are probably bad choices to start with, since they'll be rather boring. (It would be good to test those later though.) If you choose a large number for R5, you'll have to watch the loop repeat that large number of times. So let's start with two reasonably small, but different numbers, like 5 and 3.

In the R4 register enter the number 5 and hit enter. Now repeat the process, but type into the R5 field, and specify 3 in the **Value** field. Now your registers are set, and you're ready to try running your program.

To run your program until the breakpoint, first make sure x3200 is in the PC field. Then hit continue and you should encounter your breakpoint at x3204.

That happened because you set a breakpoint at your halt line. Click **Cancel** to close the popup, and then take a look at R2, which is supposed to contain your result now that you've run all but the last line of the program. As you realize, 3 * 5 = 15 in decimal, but R2 now contains 20 in decimal (which is x14 in hex). Something went wrong. Now you need to figure out what that something was.

Stepping through the multiply program

One option for debugging your program is to step through the entire multiply program from beginning to end. Since you have a loop, let's approach debugging a different way. First, let's try stepping through one iteration of the loop to make sure that each instruction does what you think it should.

First you'll have to reset the initial values in your registers, and set the PC to the first instruction. R4 still contains x5, so you can leave it alone. To reset R5, set the value of R5 to x3, and hit enter.

The PC is now pointing to the first line, and the two registers you'll need are initialized to the values you want. So you're ready to step through the program, in your first attempt to figure out what's going wrong.

Click **Step**. Notice that several things changed. The PC now points to the next instruction, x3201. The IR contains the value of the first instruction, x54A0. R2, which moments ago contained our incorrect result (decimal 20), is clear. This is exactly what you should have expected the first instruction to do, so let's keep going.

Click **Step** again. Once more, the PC and IR have changed as expected. Now R2 contains the value 5 (the same in hex and decimal, by the way). Again, this is what you want. So keep going.

The next time you click **Step**, the value of R5 changes from x3 to x2. (I'm not going to keep mentioning the PC and IR, but you'll notice those changing after each instruction as well.) R5 has a double purpose in this program. It is one of the numbers to multiply, but it is also your "counter" – it tells you how many more repetitions of the loop are left to go. So each time through the loop, R5 gets decremented. That seemed to happen just fine, so keep going.

Clicking **Step** once more causes the branch instruction to execute. When a branch instruction executes, one of two things can happen. Either the branch gets taken, or it doesn't get taken. In this case, the branch got taken. Why? Because the branch tested the condition codes which were set by the add instruction right before it. The result of the add was x2, a positive number, so the P register was set to 1. Your branch is taken if the Z register or the P register contains a 1. So the branch was executed, and the branch was taken, and the PC now points to x3201, ready for another iteration of the loop.

Stepping through the program for one repetition of the loop has shown that there's nothing wrong with any of the individual instructions in the loop. Maybe the problem lies in the way the loop is set up instead, so let's try another approach.

Debugging the loop with a breakpoint

One good technique for discovering whether a loop is being executed too many times, is to put a breakpoint at the branch instruction. That way, you can pause once at the end of each iteration, and check out the state of various registers (and memory locations, in more complicated programs).

To set this breakpoint, double-click on memory location x3203. Now you have two breakpoints. The simulator will pause whenever the PC gets the value x3203, or the value x3204.

Now you'll need to set R5 to x3 and the PC to x3200, the same way you did earlier.

Click **Continue** to run your program, notice what has changed in your registers. The PC points to line x3203. R4 is unchanged – it contains x5. R5 has changed – it contains x2.

R2 has changed – it contains x5. The condition codes have P=1. That tells you that when you continue to run the program, the branch will be taken.

Now click **Continue** again, which causes the program to keep running until it hits the breakpoint at line x3203 once again. Look at your registers, in particular R5 and R2. Now you've gone through the loop two times, so R5 contains x1. R2 contains decimal 10. The condition codes again have P=1, so you're going to do the loop again when you continue to run the program.

Click **Continue** once more. R5 now equals zero, and R2 is decimal 15. Since 3 * 5 = 15, you know we want to stop at this point. But look at the condition codes. Z = 1, and your branch statement is written so that it will branch when either Z or P is 1. So instead of stopping, we're going to take the branch again and do an extra, unwanted iteration of the loop. That's the bug.

By changing the branch instruction to only be taken when P = 1, the loop will happen the correct number of times. To prove this to yourself, you can edit the program in emacs, and change the branch line to this:

```
0000 001 111111101          ;branch to location x3201 if the result is positive
```

and save, convert to .obj format, and load the new version into the simulator.

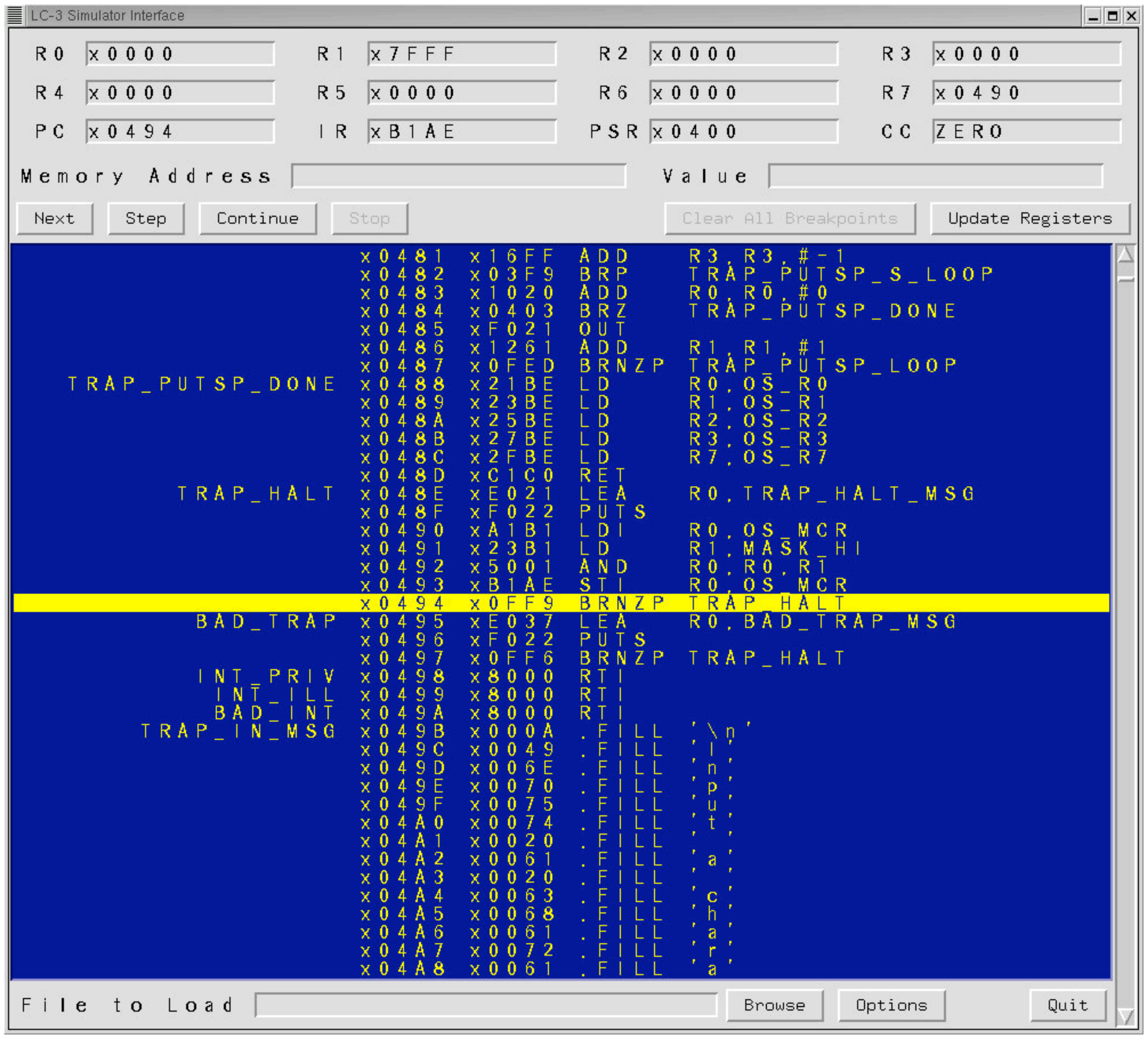
It works!

Now set the PC to x3200 once again, and change R5 to contain x3. To remove the breakpoint at line x3203, simply double-click that line of memory in the simulator.

Now run your program by setting the PC to x3200 and hitting continue. Congratulations – you've successfully debugged a program!

Chapter 5
LC-3 Simulator reference, Unix version

Here is what you see when you launch the Unix version of the simulator:



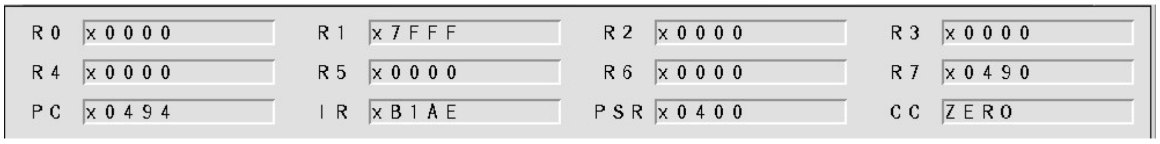
```
R0  x0000     R1  x7FFF     R2  x0000     R3  x0000
R4  x0000     R5  x0000     R6  x0000     R7  x0490
PC  x0494     IR  xB1AE     PSR x0400     CC  ZERO

Memory Address [            ]   Value [            ]
```

```
                  x0481  x16FF  ADD    R3,R3,#-1
                  x0482  x03F9  BRP    TRAP_PUTSP_S_LOOP
                  x0483  x1020  ADD    R0,R0,#0
                  x0484  x0403  BRZ    TRAP_PUTSP_DONE
                  x0485  xF021  OUT
                  x0486  x1261  ADD    R1,R1,#1
                  x0487  x0FED  BRNZP  TRAP_PUTSP_LOOP
TRAP_PUTSP_DONE   x0488  x21BE  LD     R0,OS_R0
                  x0489  x23BE  LD     R1,OS_R1
                  x048A  x25BE  LD     R2,OS_R2
                  x048B  x27BE  LD     R3,OS_R3
                  x048C  x2FBE  LD     R7,OS_R7
                  x048D  xC1C0  RET
      TRAP_HALT   x048E  xE021  LEA    R0,TRAP_HALT_MSG
                  x048F  xF022  PUTS
                  x0490  xA1B1  LDI    R0,OS_MCR
                  x0491  x23B1  LD     R1,MASK_HI
                  x0492  x5001  AND    R0,R0,R1
                  x0493  xB1AE  STI    R0,OS_MCR
                  x0494  x0FF9  BRNZP  TRAP_HALT
       BAD_TRAP   x0495  xE037  LEA    R0,BAD_TRAP_MSG
                  x0496  xF022  PUTS
                  x0497  x0FF6  BRNZP  TRAP_HALT
       INT_PRIV   x0498  x8000  RTI
        INT_ILL   x0499  x8000  RTI
        BAD_INT   x049A  x8000  RTI
    TRAP_IN_MSG   x049B  x000A  .FILL  '\n'
                  x049C  x0049  .FILL  'I'
                  x049D  x006E  .FILL  'n'
                  x049E  x0070  .FILL  'p'
                  x049F  x0075  .FILL  'u'
                  x04A0  x0074  .FILL  't'
                  x04A1  x0020  .FILL  ' '
                  x04A2  x0061  .FILL  'a'
                  x04A3  x0020  .FILL  ' '
                  x04A4  x0063  .FILL  'c'
                  x04A5  x0068  .FILL  'h'
                  x04A6  x0061  .FILL  'a'
                  x04A7  x0072  .FILL  'r'
                  x04A8  x0061  .FILL  'a'

File to Load [                    ]
```

The interface has several parts, so let's look at each one in turn.

The registers

Near the top of the interface, you'll see the most important registers (for our purposes, anyway) in the LC-3, along with their contents.



```
R0  x0000     R1  x7FFF     R2  x0000     R3  x0000
R4  x0000     R5  x0000     R6  x0000     R7  x0490
PC  x0494     IR  xB1AE     PSR x0400     CC  ZERO
```

The General Purpose Registers, R0 through R7, are the eight registers that LC-3 instructions use as sources of data and destinations of results. The numbers following the "=" are the contents of those registers, first in hex and then (in parentheses) in decimal.

The Special Registers section shows the names and contents of the PC, the IR, and the N, Z, and P condition code registers.

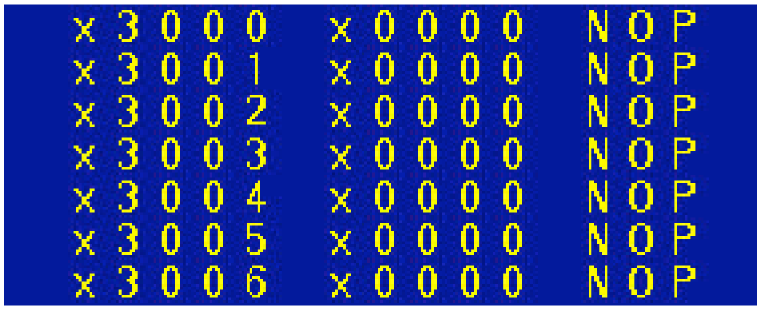
As you know, the PC, or program counter, points to the next instruction which will be executed when the current one finishes. When you launch the simulator, the PC's value will always be x3000 (12288 in decimal, but we never refer to address locations in decimal). For some reason, professors of assembly language have a special fondness for location x3000, and they love to begin programs there. Maybe one day someone will discover why.

The IR, or instruction register, holds the current instruction being executed. If there were a way to see what was happening within the LC-3 during the six phases that make up one instruction cycle, you would notice that the value of the IR would be the same during all six phases. This simulator doesn't let us "see inside" an instruction in this way. So when you are stepping through your program one instruction at a time, the IR will actually contain the instruction that just finished executing. It is still considered "current" until the next instruction begins, and the first phase of its execution – the fetch phase – brings a new value into the IR.

The CC, or condition codes, consist of three registers: N, Z, and P. Only one of the three registers can have the value 1 at any time. The other two are guaranteed to be zero. The values of the condition codes will change when you execute certain instructions (ADD, AND, OR, LEA, LD, LDI, or LDR).

The memory

Below the registers, you see a long, dense list of numbers. Use the scrollbar at the left to scroll up and down through the memory of the LC-3. Remember that the LC-3 has an address space of 2^{16}, or 65536 memory locations in all. That's a very long list to scroll through.


```
x3000  x0000  NOP
x3001  x0000  NOP
x3002  x0000  NOP
x3003  x0000  NOP
x3004  x0000  NOP
x3005  x0000  NOP
x3006  x0000  NOP
```

Most of memory is "empty" when you launch the simulator. By that I mean that most locations contain the value zero. You notice that after the address of each location are 16 0s and/or 1s. That is the 16-bit binary value which the location contains. After the binary representation you see the hex representation, which is often useful simply because it's easier to read.

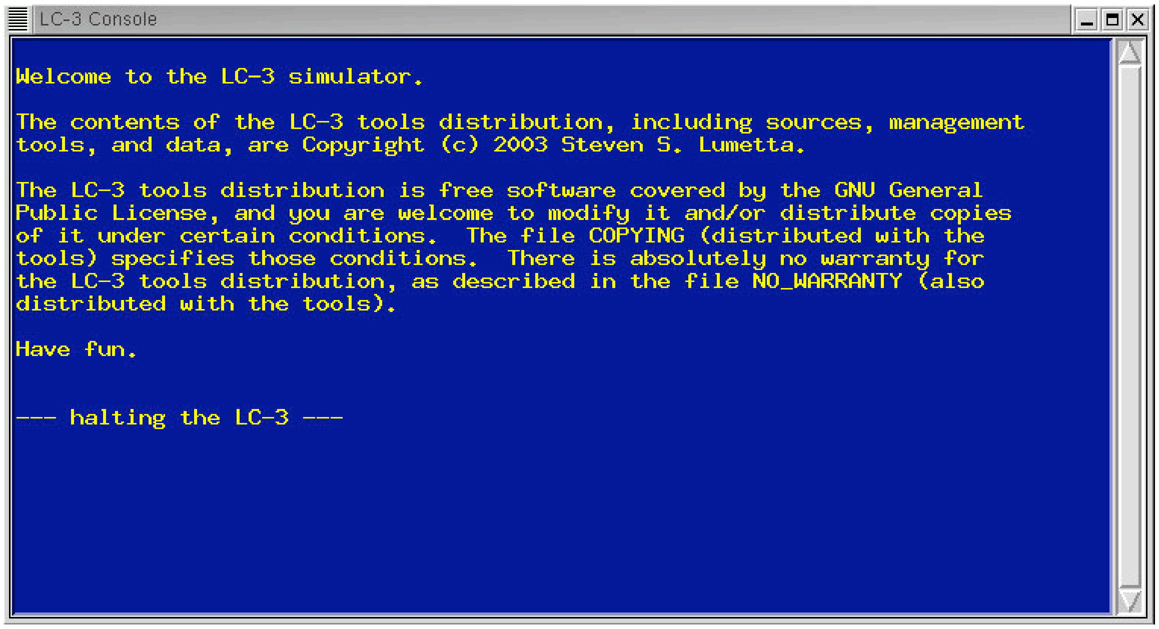
The columns after the semicolon contain words, or mnemonics. That is the simulator's interpretation of that line, translated into the assembly language of the LC-3. When you load a program, these translations will be extremely helpful because you'll be able to quickly look through your program and know what is happening. Sometimes, however, these assembly language interpretations make no sense. Remember that computers, and likewise the simulator, are stupid. They don't understand your intentions. So the data section of your program will always be interpreted into some sort of assembly language in this last column. An important aspect of the Von Neumann model is that both instructions and data are stored in the computer's memory. The only way we can tell them apart is by how we use them. If the program counter loads the data at a particular location, that data will be interpreted as an instruction. If not, it won't. So ignore the last column of information when it tries to give some nonsense interpretation to the data in your program.

You may notice, if you browse around in memory sometime, that not every memory location is set to all 0's even though you didn't put anything there. Certain sections of memory are reserved for instructions and data that the operating system needs. For instance, locations x20 through xFF are reserved for addresses of trap routines.

Other places in memory hold the instructions that carry out those routines. Don't replace the values in these locations, or strange behaviors may happen at unexpected times when you're running your programs. Maybe this information will give you a hint as to why professors are so fond of memory location x3000 as a place to start your program. It's far from any operating system section of memory, so you're not likely to replace crucial instructions or data accidentally.

Information

The Console Window



```
Welcome to the LC-3 simulator.

The contents of the LC-3 tools distribution, including sources, management
tools, and data, are Copyright (c) 2003 Steven S. Lumetta.

The LC-3 tools distribution is free software covered by the GNU General
Public License, and you are welcome to modify it and/or distribute copies
of it under certain conditions.  The file COPYING (distributed with the
tools) specifies those conditions.  There is absolutely no warranty for
the LC-3 tools distribution, as described in the file NO_WARRANTY (also
distributed with the tools).

Have fun.


--- halting the LC-3 ---
```

You should notice that a second window also appears when you run the simulator. It is rather inconspicuous, and has the vague title "LC-3 Console." This window will give you messages such as "Halting the processor." If you use input and output routines in your program, you'll see your output and do your input in this window.