Guide to Using the Unix version of the LC-2 Simulator

by

Kathy Buchheit
The University of Texas at Austin

Guide to Using the Unix version of the LC-2 Simulator

The LC-2 is a piece of hardware, so you might be wondering why we need a simulator. The reason is that the LC-2 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-2 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-2 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, starting on about page 130. 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-2 assembly language. Here's what our little program looks like in binary:

```
0011000000000000
0101001001100000
0101100100100000
0001100100101010
1110010100000000
0110011010000000
0001010010100001
0001001001000011
0001100100111111
0000001000000100
```

```
1111000000100101
```

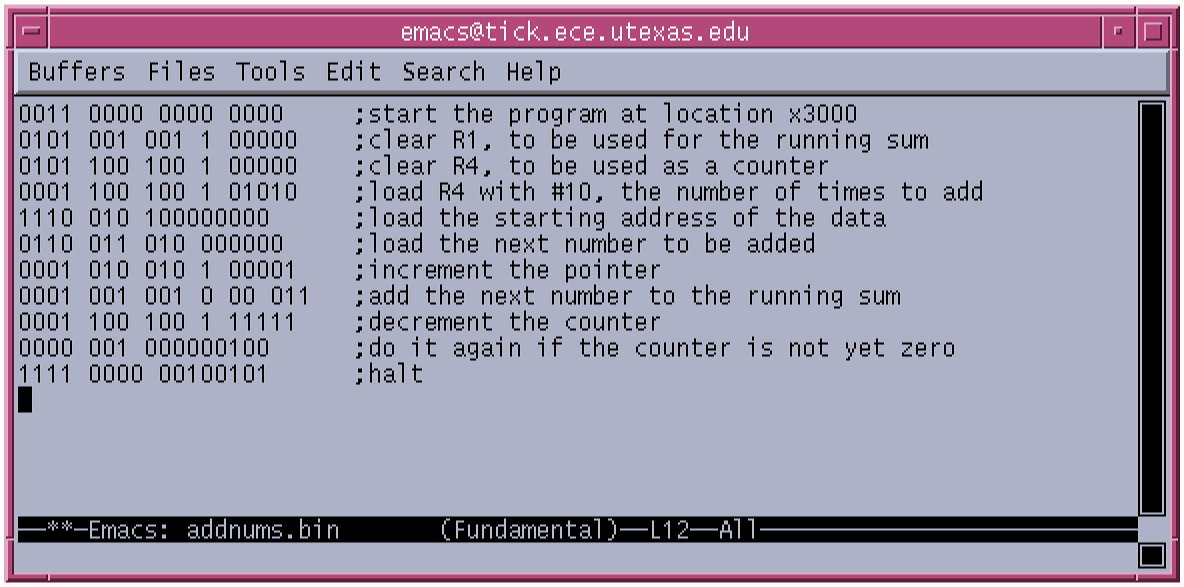
When you type this into emacs, you'll probably be looking at a chart which tells you the format of each instruction, such as the one on page 94 of Chapter 5 of the textbook. So it may be easier for you to read your own code if you leave spaces between the different sections of each instruction. Also, 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 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
```

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
E500        ;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
0204        ;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:


emacs@tick.ece.utexas.edu
Buffers Files Tools Edit Search 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
```

```
--**-Emacs: addnums.bin       (Fundamental)--L12--All-
```

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 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-2 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

```
convert –b2 addnums.bin addnums.obj
```

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

```
convert –b16 addnums.hex addnums.obj
```

* Note to instructors: before your students will be able to use the convert program, they'll need to know how to add the appropriate path in their .cshrc files. If your school uses another program called convert, you may also need to make an alias for the LC-2 convert program.

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-2 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-2 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,x100      ;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

```
assemble addnums.asm addnums
```

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 handful of files:

addnums.obj, as you expected
addnums.bin, your program in ASCII 1s and 0s
addnums.hex, your program in ASCII hex format
addnums.sym, the symbol table created on the assembler's first pass

addnums.lst, the list file for your program

The .bin and .hex files look the same as the ones shown earlier in the chapter (with any comments removed). The last two files are worth looking at.

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 hex value 3004. 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.

addnums.lst
If you open the list file using any text editor, you'll see this:

```
(0000) 3000  0011000000000000  (  1)          .orig  0x3000
(3000) 5260  0101001001100000  (  2)          and    r1 r1 0x0000
(3001) 5920  0101100100100000  (  3)          and    r4 r4 0x0000
(3002) 192A  0001100100101010  (  4)          add    r4 r4 0x000A
(3003) E500  1110010100000000  (  5)          lea    r2 0x0100
(3004) 6680  0110011010000000  (  6) LOOP     ldr    r3 r2 0x0000
(3005) 14A1  0001010010100001  (  7)          add    r2 r2 0x0001
(3006) 1243  0001001001000011  (  8)          add    r1 r1 r3
(3007) 193F  0001100100111111  (  9)          add    r4 r4 0xFFFF
(3008) 0204  0000001000000100  ( 10)                 brp    LOOP
(3009) F025  1111000000100101  ( 11)                 halt
```

Let's pick one line and take it apart. Since the sixth line has a label, that's the most interesting one. So let's look at the pieces.

```
(3004) 6680  0110011010000000 (   6) LOOP        ldr    r3 r2 0x0000
```

(3004)
This is the address where the instruction will be located in memory when your program is loaded into the simulator.

6680
This is the hex value of the instruction itself.

0110011010000000
This is the instruction in binary.

(6)
The instruction is the sixth line of the assembly language program. It will actually be the fifth line of the program once it gets loaded into memory in the simulator, since the line marked (1) just specifies the starting location. But we're counting assembly language lines now, not memory locations, so this is the sixth line.

LOOP
This is the label associated with the line.

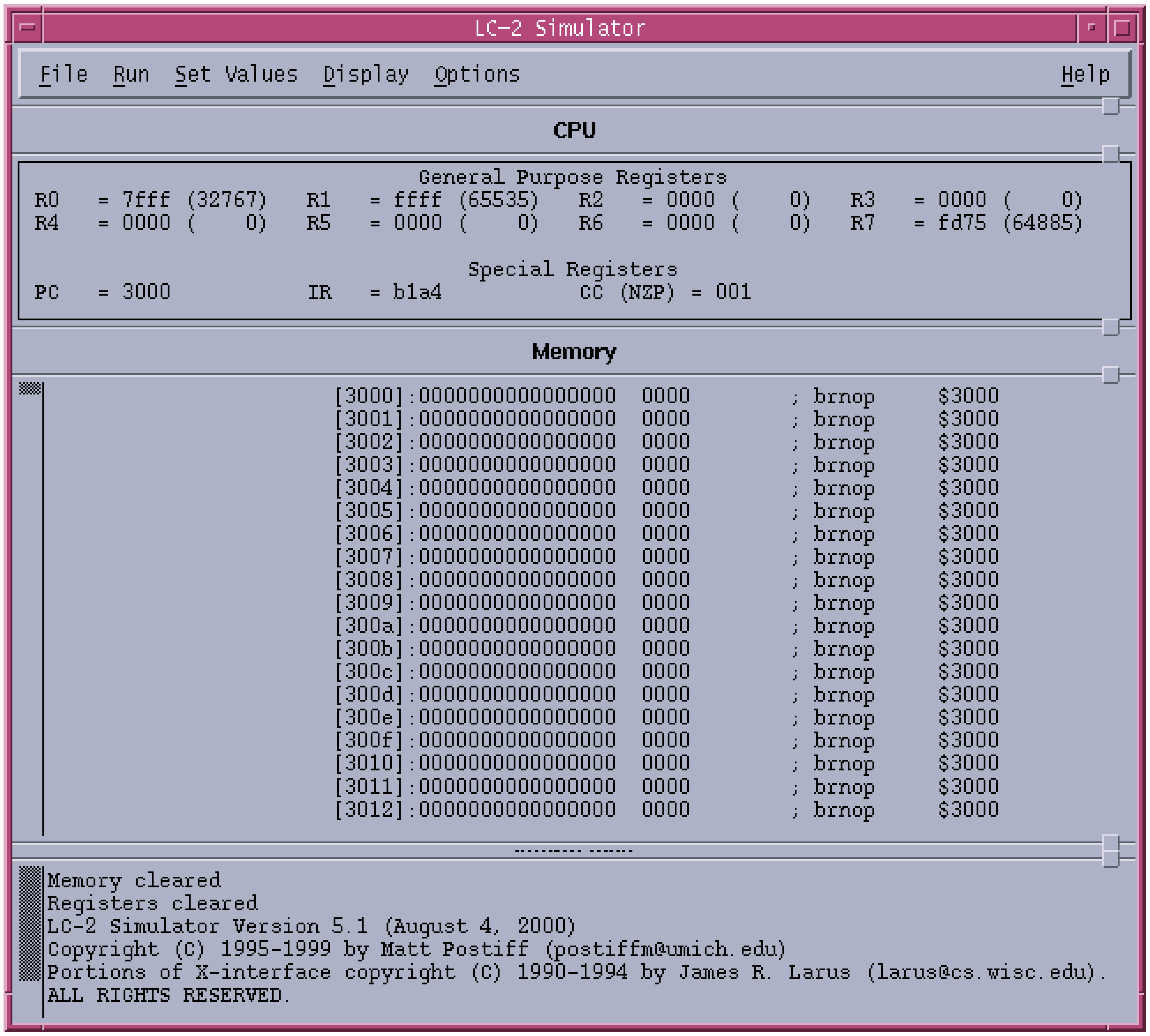
ldr   r3 r2 0x0000
And last, this is the assembly language version of the instruction. Notice that the comments after the instruction are gone now. Those were only for your own (or other programmers') information. The simulator doesn't care about them.

Chapter 2
The simulator: what you see on the screen

To start the simulator, type this at the prompt:

```
simulate &
```

When you launch the Unix version of the LC-2 Simulator, you see this:



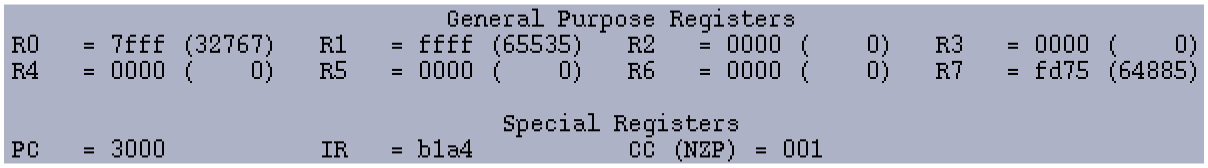
```
                              General Purpose Registers
R0   = 7fff (32767)   R1   = ffff (65535)   R2   = 0000 (    0)   R3   = 0000 (    0)
R4   = 0000 (    0)   R5   = 0000 (    0)   R6   = 0000 (    0)   R7   = fd75 (64885)

                                 Special Registers
PC   = 3000           IR   = b1a4           CC (NZP) = 001
```

```
[3000]:0000000000000000  0000      ; brnop    $3000
[3001]:0000000000000000  0000      ; brnop    $3000
[3002]:0000000000000000  0000      ; brnop    $3000
[3003]:0000000000000000  0000      ; brnop    $3000
[3004]:0000000000000000  0000      ; brnop    $3000
[3005]:0000000000000000  0000      ; brnop    $3000
[3006]:0000000000000000  0000      ; brnop    $3000
[3007]:0000000000000000  0000      ; brnop    $3000
[3008]:0000000000000000  0000      ; brnop    $3000
[3009]:0000000000000000  0000      ; brnop    $3000
[300a]:0000000000000000  0000      ; brnop    $3000
[300b]:0000000000000000  0000      ; brnop    $3000
[300c]:0000000000000000  0000      ; brnop    $3000
[300d]:0000000000000000  0000      ; brnop    $3000
[300e]:0000000000000000  0000      ; brnop    $3000
[300f]:0000000000000000  0000      ; brnop    $3000
[3010]:0000000000000000  0000      ; brnop    $3000
[3011]:0000000000000000  0000      ; brnop    $3000
[3012]:0000000000000000  0000      ; brnop    $3000
```

```
Memory cleared
Registers cleared
LC-2 Simulator Version 5.1 (August 4, 2000)
Copyright (C) 1995-1999 by Matt Postiff (postiffm@umich.edu)
Portions of X-interface copyright (C) 1990-1994 by James R. Larus (larus@cs.wisc.edu).
ALL RIGHTS RESERVED.
```

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.

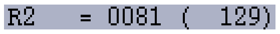
The registers
Below the heading CPU, notice the list of registers.



```
                                General Purpose Registers
R0    = 7fff (32767)    R1    = ffff (65535)    R2    = 0000 (     0)    R3    = 0000 (     0)
R4    = 0000 (     0)    R5    = 0000 (     0)    R6    = 0000 (     0)    R7    = fd75 (64885)

                                    Special Registers
PC    = 3000            IR    = b1a4            CC (NZP) = 001
```

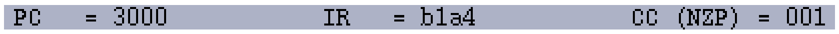
The General Purpose Registers, R0 through R7, are the eight registers that LC-2 instructions use as sources of data and destinations of results. The numbers following the "=" are the contents of those registers, first in hex and then (in parentheses) in decimal.

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



```
R2    = 0081 (   129)
```

The Special Registers section shows the names and contents of five important registers in the LC-2 control unit. Those registers are the PC, the IR, and the N, Z, and P condition code registers.



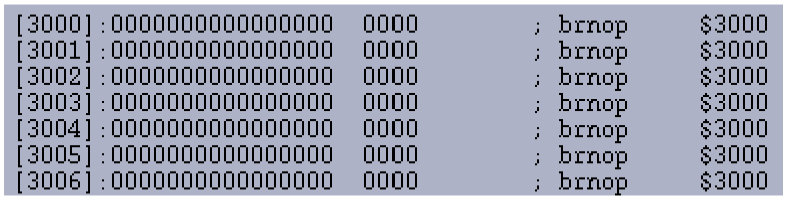
```
PC    = 3000            IR    = b1a4            CC (NZP) = 001
```

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.

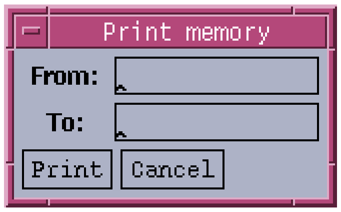
The CC, or condition codes, are set by certain instructions (ADD, AND, OR, LEA, LD, LDI, and LDR). They consist of three registers: N, Z, and P. Remember that only one of the three can have the value 1 at any time. The simulator shows us the values of the three registers all lumped together. The above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



```
[3000]:0000000000000000  0000        ; brnop     $3000
[3001]:0000000000000000  0000        ; brnop     $3000
[3002]:0000000000000000  0000        ; brnop     $3000
[3003]:0000000000000000  0000        ; brnop     $3000
[3004]:0000000000000000  0000        ; brnop     $3000
[3005]:0000000000000000  0000        ; brnop     $3000
[3006]:0000000000000000  0000        ; brnop     $3000
```

Use the scrollbar at the left (with the middle mouse button) to scroll up and down through the memory of the LC-2. Remember that the LC-2 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 **Display** menu at the top of the window, and choose **Memory**. You'll get this popup window:

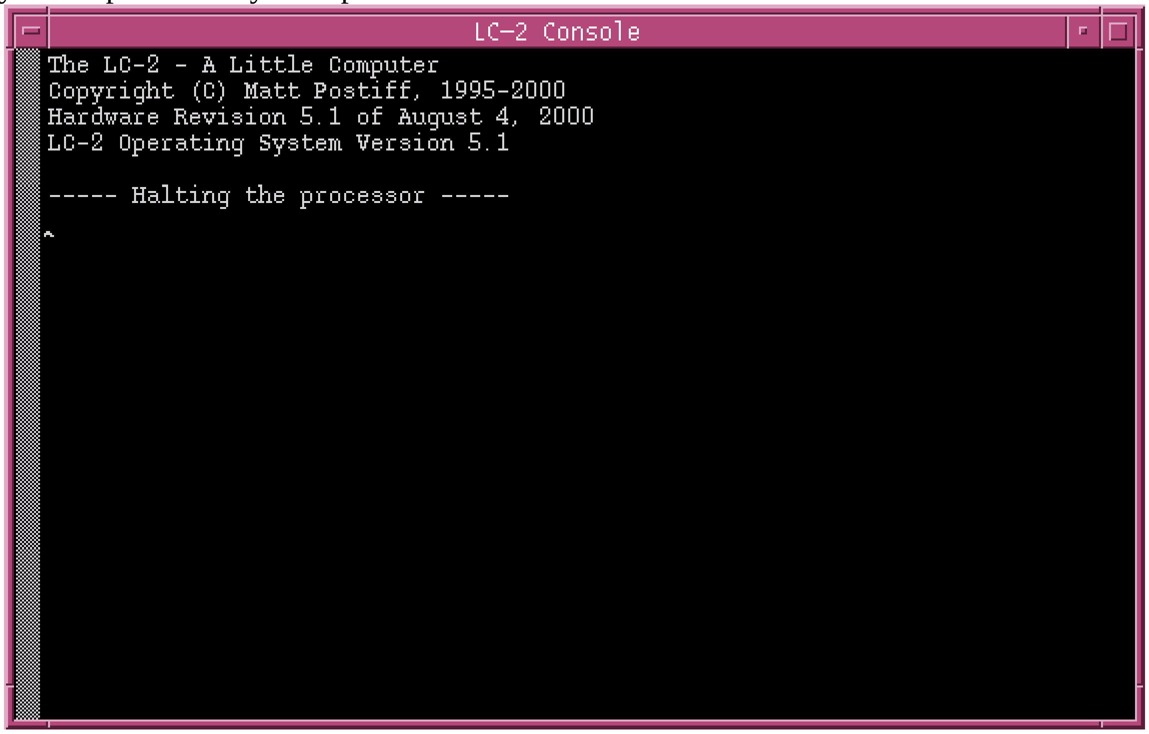


In the **From** field, type the memory location you want to start with, such as 3000. In the **To** field, type the memory location to display through, such as 3020. Then click **Print**, and those x20 locations will appear in the Memory part of the simulator.

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 binary. The third column also represents the contents, but in hex instead of binary, because that's sometimes easier to interpret. The columns after the semicolon 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-2 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.



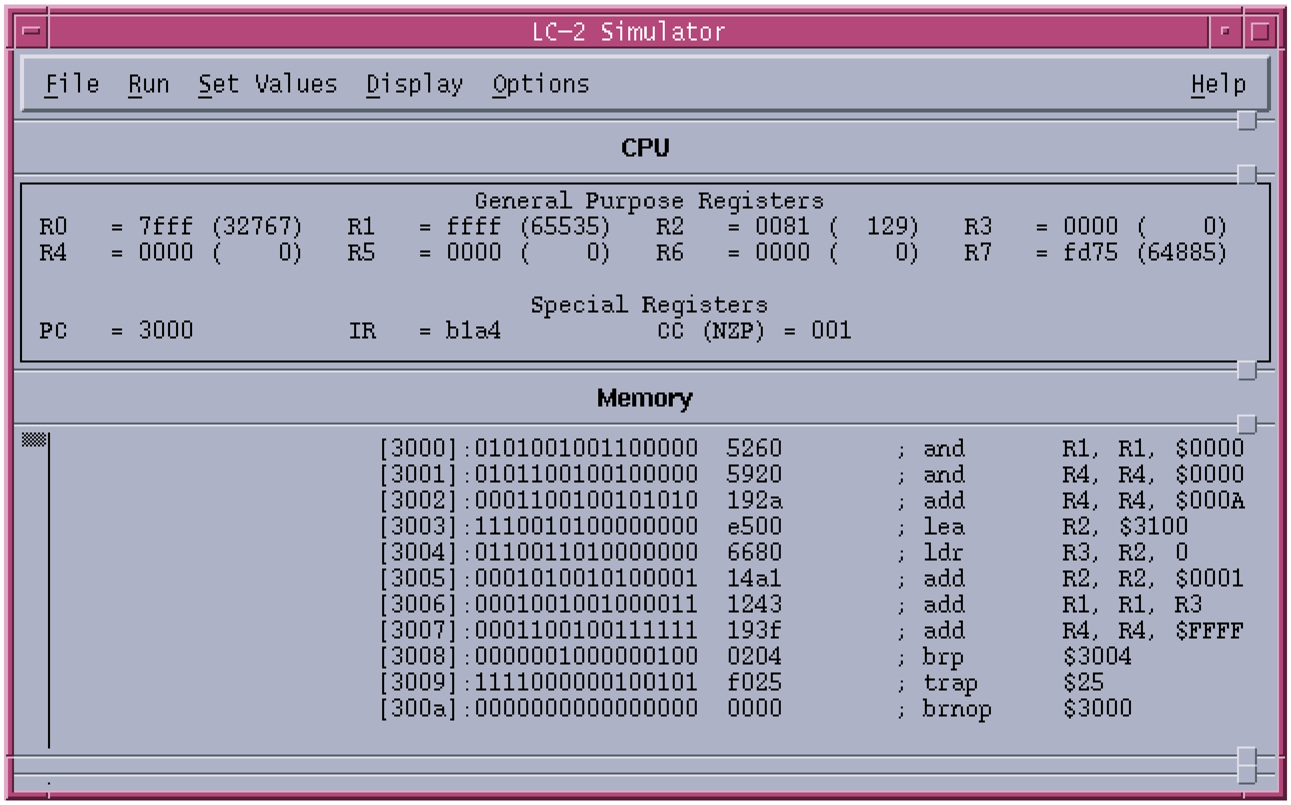
```
The LC-2 - A Little Computer
Copyright (C) Matt Postiff, 1995-2000
Hardware Revision 5.1 of August 4, 2000
LC-2 Operating System Version 5.1

----- Halting the processor -----
```

Chapter 3
Running a program in the simulator

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

```
simulate &
```

at the prompt. To load your program, click **File** and then **Load Program**. When the File Selection Dialog_popup appears, choose *addnums.obj* and click **OK**. This is what you'll see when your program is loaded:



```
                        General Purpose Registers
R0   = 7fff (32767)   R1   = ffff (65535)   R2   = 0081 (  129)   R3   = 0000 (    0)
R4   = 0000 (    0)   R5   = 0000 (    0)   R6   = 0000 (    0)   R7   = fd75 (64885)

                        Special Registers
PC   = 3000           IR   = b1a4           CC (NZP) = 001
```

```
[3000]:0101001001100000  5260     ; and      R1, R1, $0000
[3001]:0101100100100000  5920     ; and      R4, R4, $0000
[3002]:0001100100101010  192a     ; add      R4, R4, $000A
[3003]:1110010100000000  e500     ; lea      R2, $3100
[3004]:0110011010000000  6680     ; ldr      R3, R2, 0
[3005]:0001010010100001  14a1     ; add      R2, R2, $0001
[3006]:0001001001000011  1243     ; add      R1, R1, R3
[3007]:0001100100111111  193f     ; add      R4, R4, $FFFF
[3008]:0000001000000100  0204     ; brp      $3004
[3009]:1111000000100101  f025     ; trap     $25
[300a]:0000000000000000  0000     ; brnop    $3000
```

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).

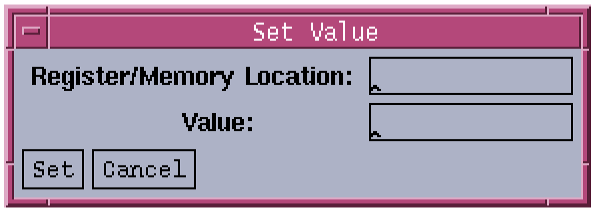
Loading the data (ten numbers) into memory

There are several ways to get the ten numbers that you're planning to add into the memory of the LC-2 simulator. You want them to begin at location x3100.

First way: click on the **Display** menu at the top of the simulator, and then choose **Memory**. When the popup window appears, choose 3100 as your **From** amount, and

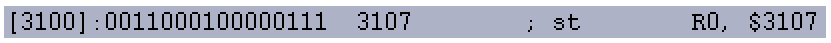
310f as your **To** amount. Click **Print**. The locations displayed under Memory will now go from 3100 to 310f.

Now click the **Set Values** menu, and choose **Set Register or memory**. You will see this window:



In the **Register/Memory Location** field, type 3100, and for the **Value**, type 3107. Then click **Set**.

Now your first data location will look like this:



```
[3100]:0011000100000111   3107         ; st        R0, $3107
```

Notice that you get to see the binary representation, the hex representation, and some silly, useless assembly language representation (st R0, $3107). Of course, this is data, not an instruction, but the LC-2 simulator doesn't know that. In fact, the contents of all memory locations are equal in the eyes of the LC-2, until they get run as instructions or loaded as data. Since you have a halt instruction far before the place where you're putting this data, it will never be treated as an instruction. So ignore that assembly language interpretation.

You can set each line in turn by choosing **Set Values** and **Set Register or memory**. This method is rather tedious! But 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
```

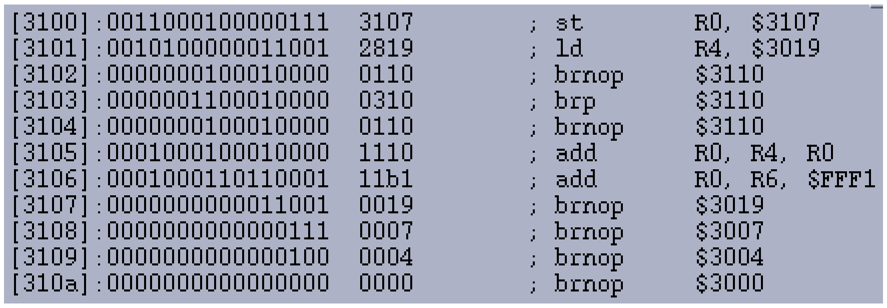
0007
0004

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 data.obj
```

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

Now go back to the simulator, choose **File** and **Load Program…** once again, and select *data.obj*. Note that you can load multiple .obj files so they exist concurrently in the LC-2 simulator's memory. The memory locations starting with x3100 will look like this:



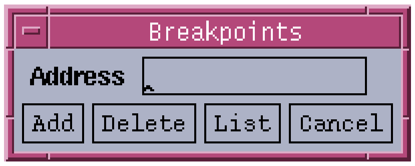
```
[3100]:0011000100000111  3107      ; st       R0, $3107
[3101]:0010100000011001  2819      ; ld       R4, $3019
[3102]:0000000100010000  0110      ; brnop    $3110
[3103]:0000001100010000  0310      ; brp      $3110
[3104]:0000000100010000  0110      ; brnop    $3110
[3105]:0001000100010000  1110      ; add      R0, R4, R0
[3106]:0001000110110001  11b1      ; add      R0, R6, $FFF1
[3107]:0000000000011001  0019      ; brnop    $3019
[3108]:0000000000000111  0007      ; brnop    $3007
[3109]:0000000000000100  0004      ; brnop    $3004
[310a]:0000000000000000  0000      ; brnop    $3000
```

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

Running your program

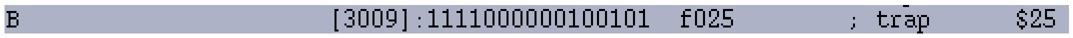
Click on the **Display** field, and choose **Memory**. Specify **From** as 3000 and **To** as 310a. Now when you scroll through memory, you'll be able to see your program in locations 3000 through 3009, and your data in locations 3100 through 3109.

This next step is VERY important: click on the **Run** menu and then choose **Breakpoints….** You'll see this popup window:



In the **Address** field, type 3009, and then click **Add**.

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                    [3009]:1111000000100101  f025        ; trap      $25
```

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 you're ready to run your program. Make sure the PC has the value x3000, because that's where the first instruction is. If it doesn't, click on the **Set Values** menu and choose **Set Register or memory**. In the **Register/Memory Location** field, type PC, and in the **Value** field specify the starting location of your program, which is 3000. Click **Set**.

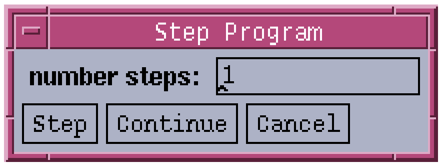
Now for the big moment: click on the **Run** menu and choose **Run Program…**! When you get a popup window asking you to verify the starting address, click **Run**. When you get another popup window telling you that the breakpoint has been encountered, click **Cancel**. If you've already added up the 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-2 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. (Click on the **Set Values** menu, choose **Set Register or Memory**, and then type PC for your **Location** and 3000 for your **Value**, and click **Set**.) Now you're ready to step through your program.

Click on the **Run** menu, and choose **Step Program….** A popup window will appear:



First, move this window to the side so that it doesn't block your view of the simulator. You want the number of steps (or number of instructions executed at a time) to be 1, so click Step.

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

Click **Step** 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 1:
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 emacs (or your favorite text editor). 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 000000001         ;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:

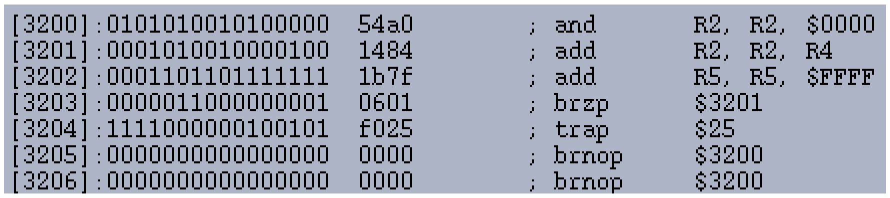
```
convert –b2 multiply.bin multiply.obj
```

Loading the program into the simulator

Start the simulator by typing

```
simulate &
```

at the prompt, and then click the **File** menu, and **Load Program…** to load your program: *multiply.obj*. Now the memory portion of the simulator will look like this:



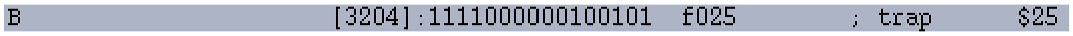
```
[3200]:0101010010100000  54a0      ; and      R2, R2, $0000
[3201]:0001010010000100  1484      ; add      R2, R2, R4
[3202]:0001101101111111  1b7f      ; add      R5, R5, $FFFF
[3203]:0000011000000001  0601      ; brzp     $3201
[3204]:1111000000100101  f025      ; trap     $25
[3205]:0000000000000000  0000      ; brnop    $3200
[3206]:0000000000000000  0000      ; brnop    $3200
```

Also notice that the PC contains the value x3200, which corresponds to the next instruction to be run, which happens to be the first instruction of your program, since you haven't started yet.

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**.

Now line x3204 should look like this:



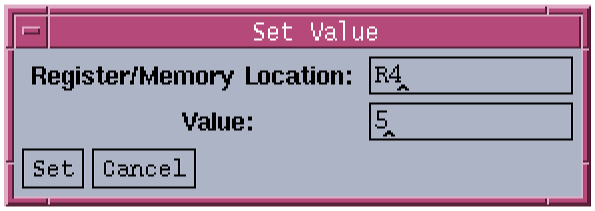
```
B                         [3204]:1111000000100101  f025        ; trap      $25
```

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.

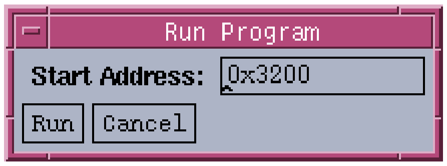
Click on the **Set Values** menu, and choose **Set Register or memory**. You'll get a popup window. Now specify R4 in the **Register/Memory Location** field, and specify 5 in the **Value** field. So you should see this:



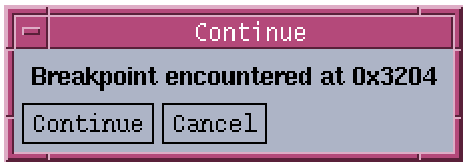
Register/Memory Location: R4
Value: 5

Click **Set**. Now repeat the process, but specify R5 in the **Register/Memory Location** 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, click on the **Run** menu, and choose **Run Program…**. In a fraction of a second, you'll get a popup window that looks like this:



Click **Run**. In another moment, you'll get another popup window like this:


Breakpoint encountered at 0x3204

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, click on the **Set Values** menu and choose **Set Register or memory**. Set the value of R5 to x3, and click **Set** to close the window.

Now choose **Set Values** and **Set Register or memory** once more. This time, set the PC to 3200, and click **Set**.

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 the **Run** menu and choose **Step Program…**. When the popup window appears, 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.

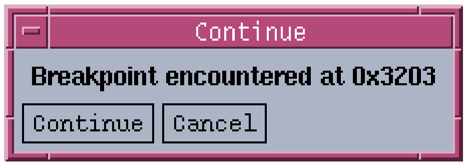
Debugging the loop with a breakpoint

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, choose the **Run** menu and then **Breakpoints…**. Type x3203 for your **Address**. Click **Add**. 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 **Run** and **Run Program…**. When the Run Program popup window appears, click **Run**. Almost immediately, you'll see this:



Breakpoint encountered at 0x3203

Move this window to one side, and 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** in the popup window, 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 000000001          ;branch to location x3201 if the result is positive
```

and save, convert to .obj format, and load the new version into the simulator. Or if you don't want to change the source code yet in case you find more bugs later, you can just change it quickly in the simulator. Click on the **Set Values** menu and choose **Set Register or memory**. In the **Register/Memory Location** field, type 3203. In the **Value** field, type 0201, which is the corrected instruction: 0000 001 000000001. Click **Set**. (But remember that next time you load *multiply.obj*, the bug will still be there unless you go back and fix the original binary version of the program and reconvert it.)

It works!

Now set the PC to x3200 once again, and change R5 to contain x3. To remove the breakpoint at line x3203, click the **Run** menu and choose **Breakpoints…**. Specify 3203 for your **Address**, and click **Delete**.

Now run your program by choosing **Run** and **Run Program…**. Click **Run** in the first popup window. After choosing **Cancel** in the Continue popup, you'll see the beautiful and long-awaited value of decimal 15 in R2. Congratulations – you've successfully debugged a program!

Example 2:
Debugging the program to input numbers and add them

This program is in assembly language, so if you haven't learned the LC-2 assembly language, you might want to wait until you do before you read this section.

Entering the program in emacs

The purpose of this program is to request two integers (between 0 and 9) from the user, add them, and display the result (which must also be between 0 and 9) in the console window. Our buggy version of the program looks like this:

```
        .ORIG       x3000
        TRAP        x23         ;the trap instruction which is also known as "IN"
        ADD         R1,R0,x0    ;move the first integer to register 1
        TRAP        x23         ;another "IN"
        ADD         R2,R0,R1    ;add the two integers
        LEA         R0,MESG     ;load the address of the message string
        TRAP        x22         ;"PUTS" outputs a string
        ADD         R0,R2,x0    ;move the sum to R0, to be output
        TRAP        x21         ;display the sum
        HALT
MESG    .STRINGZ    "The sum of those two numbers is "
        .END
```

Save this program in emacs, and assemble it by typing this at the command prompt:

```
assemble sum.asm sum
```

Notice that the output file will have an *.obj* extension added onto whatever you give for the name, so in this case your file will be called *sum.obj*.

Running the buggy program in the simulator

Launch the simulator, and load the program. Notice that the halt is at line x3008, and that starting at line x3009, you see one ASCII value per line. At line x3009, you see x54, which is the ASCII code for “T.” At line x300A, you see x68, the ASCII code for “h.” The whole string, “The sum of those two number is “ is represented in the memory locations from x3009 to x3028, the last of which contains the final space before the end quotes.

Set a breakpoint on line x3008 (the halt) by clicking the **Run** menu and **Breakpoints…**. Type 3008 for your **Address**, and click **Add**.

Now run your program by clicking the **Run** menu and **Run Program…**. When the popup window appears, click **Run**. The first instruction is a trap routine which prompts you to input a character into the console window like this:

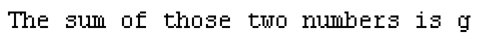
```
Input a character>
```

The simulator will just “wait” in the trap routine. You’ll need to make the console window the active window by clicking on it, and then press an integer between 0 and 9. Try “4.”

Notice that the “4” you typed actually appears as x34 in R0. (R0 is the register which ends up with your keyboard input during the IN trap routine.) If you look at the ASCII chart in Appendix E of your book, you’ll see that the integer 4 is indeed represented by the ASCII code x34.

The second instruction of your program moves this x34 into R1, and then you’re prompted again to enter a character. Because this is a very, very simple program, you have to specify another integer which, when added to the first, creates a sum of 9 or less. So click on the console window again, and enter “3.”

As soon as you enter the second number, you’ll see a message in the console window:



```
The sum of those two numbers is g
```

You know that 3 + 4 is 7. What went wrong?

Debugging the program

The big hint about why this program gives the wrong result is a few paragraphs above this. Remember that the “4” that you typed in the console window actually ended up in R0 as the value x34. Then the “3” you typed in appeared as x33. We added those values, and ended up with x67. Looking in that ASCII chart, the code x67 represents “g.” So your output makes sense, but it isn’t what you want!

You’re going to need to add a few lines to your program, along with two pieces of data, in order for it to work correctly. The trick has to do with the ASCII codes for the digits 0 through 9. “0” is represented by x30. “1” is represented by x31. This pattern continues until “9” is represented by x39. So how about subtracting x30 from the ASCII value of your integer, to get its numerical value?

You’re going to need the following pieces of data:

```
ASCII       .FILL  x30        ;the mask to add to a digit to convert it to ASCII
NEGASCII    .FILL  xFFD0      ;the negative version of the ASCII mask (-x30)
```

You’re also going to need to add five instructions: two to load the two masks, one to add the negative version of the mask to the first number, one to do the same to the second number, and then one to add the positive version of the mask to the result before outputting it. Your program should now look like this (the new lines are in bold):

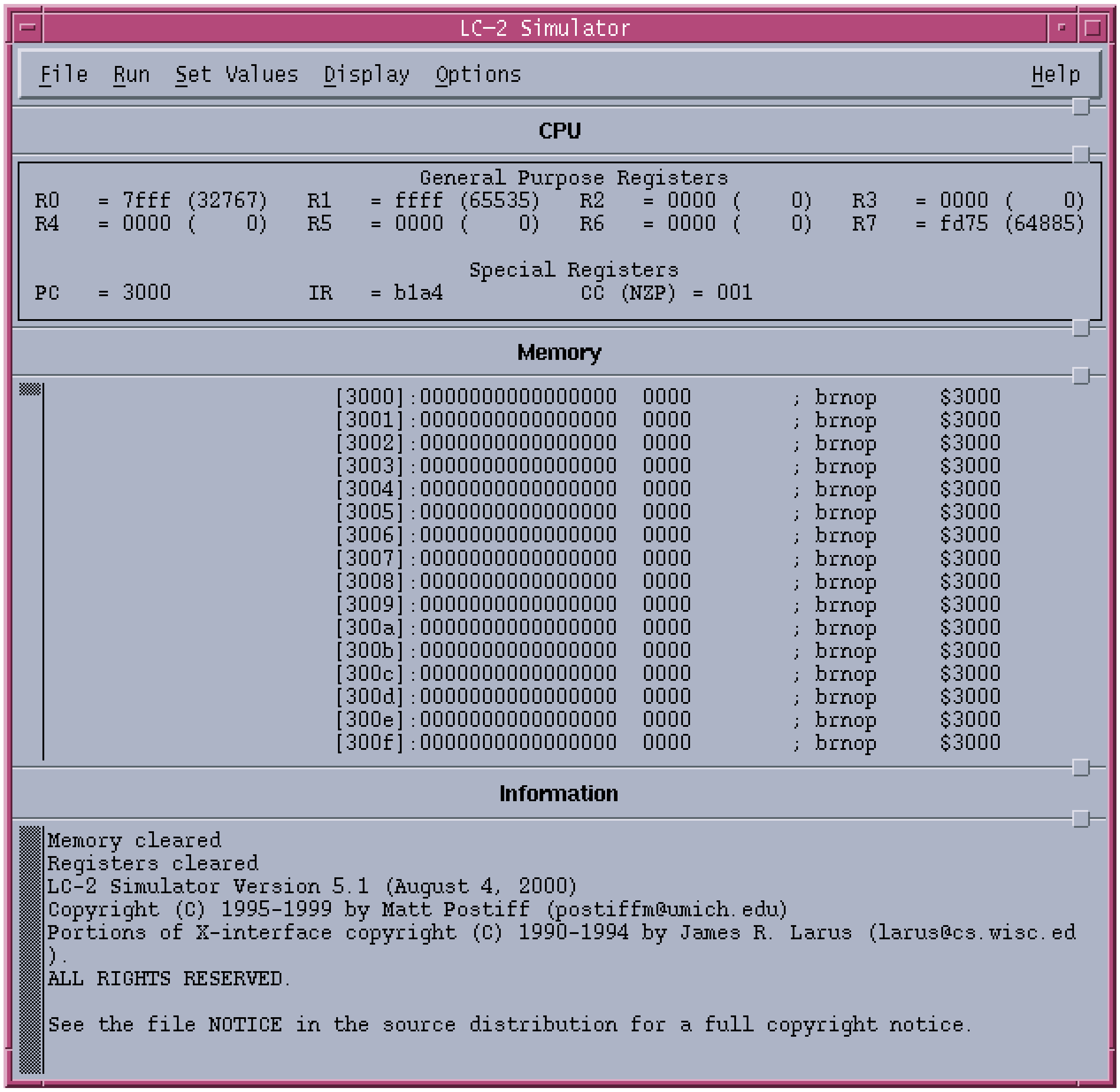
```
         .ORIG        x3000
         LD           R6, ASCII
         LD           R5, NEGASCII
         TRAP         x23          ;the trap instruction which is also known as "IN"
         ADD          R1,R0,x0     ;move the first integer to register 1
         ADD          R1,R1,R5     ;convert first ASCII number to numerical value
         TRAP         x23          ;another "IN"
         ADD          R0,R0,R5     ;convert next ASCII number to numerical value
         ADD          R2,R0,R1     ;add the two integers
         ADD          R2,R2,R6     ;convert the sum to its ASCII representation
         LEA          R0,MESG      ;load the address of the message string
         TRAP         x22          ;"PUTS" outputs a string
         ADD          R0,R2,x0     ;move the sum to R0, to be output
         TRAP         x21          ;display the sum
         HALT
ASCII    .FILL        x30          ;the mask to add to a digit to convert it to ASCII
NEGASCII .FILL        xFFD0        ;the negative version of the ASCII mask
MESG     .STRINGZ     "The sum of those two numbers is "
         .END
```

Save your program in emacs, and reassemble. When you try this version in the simulator, it should work just the way you expect.

One comment about the revised version of the program: it’s actually possible to make this program somewhat shorter. The way it is now, all steps are spelled out clearly and done separately. If you want to work on your programming efficiency skills, try to rewrite this program so that it behaves in exactly the same way, but is about four lines shorter.

Chapter 5
LC-2 Simulator reference, Unix version

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



```
                    General Purpose Registers
R0   = 7fff (32767)   R1   = ffff (65535)   R2   = 0000 (     0)   R3   = 0000 (     0)
R4   = 0000 (     0)   R5   = 0000 (     0)   R6   = 0000 (     0)   R7   = fd75 (64885)

                        Special Registers
PC   = 3000             IR   = b1a4             CC (NZP) = 001
```

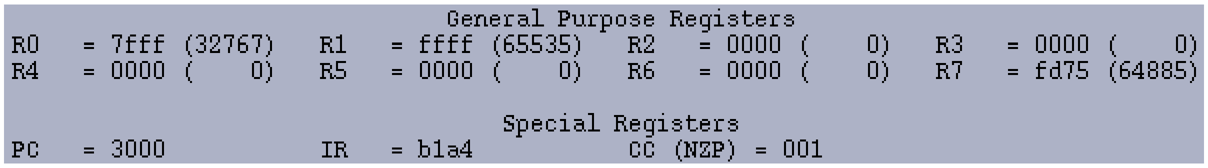
```
[3000]:0000000000000000  0000       ; brnop     $3000
[3001]:0000000000000000  0000       ; brnop     $3000
[3002]:0000000000000000  0000       ; brnop     $3000
[3003]:0000000000000000  0000       ; brnop     $3000
[3004]:0000000000000000  0000       ; brnop     $3000
[3005]:0000000000000000  0000       ; brnop     $3000
[3006]:0000000000000000  0000       ; brnop     $3000
[3007]:0000000000000000  0000       ; brnop     $3000
[3008]:0000000000000000  0000       ; brnop     $3000
[3009]:0000000000000000  0000       ; brnop     $3000
[300a]:0000000000000000  0000       ; brnop     $3000
[300b]:0000000000000000  0000       ; brnop     $3000
[300c]:0000000000000000  0000       ; brnop     $3000
[300d]:0000000000000000  0000       ; brnop     $3000
[300e]:0000000000000000  0000       ; brnop     $3000
[300f]:0000000000000000  0000       ; brnop     $3000
```

```
Memory cleared
Registers cleared
LC-2 Simulator Version 5.1 (August 4, 2000)
Copyright (C) 1995-1999 by Matt Postiff (postiffm@umich.edu)
Portions of X-interface copyright (C) 1990-1994 by James R. Larus (larus@cs.wisc.ed
).
ALL RIGHTS RESERVED.

See the file NOTICE in the source distribution for a full copyright notice.
```

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-2, along with their contents.



```
                              General Purpose Registers
R0   = 7fff (32767)   R1   = ffff (65535)   R2   = 0000 (    0)   R3   = 0000 (    0)
R4   = 0000 (    0)   R5   = 0000 (    0)   R6   = 0000 (    0)   R7   = fd75 (64885)

                                  Special Registers
PC   = 3000           IR   = b1a4           CC (NZP) = 001
```

The General Purpose Registers, R0 through R7, are the eight registers that LC-2 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.

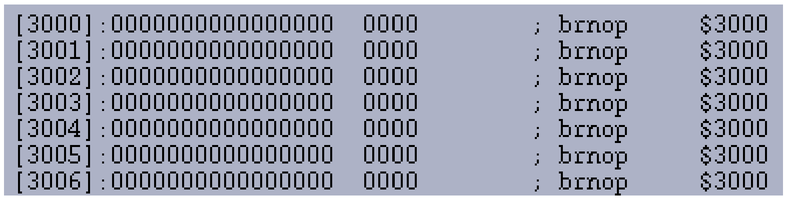
As you know, the PC, or program counter, points to the next instruction which will be executed when the current one finishes. When you launch the simulator, the PC's value will always be x3000 (12288 in decimal, but we never refer to address locations in decimal). For some reason, professors of assembly 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-2 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-2. Remember that the LC-2 has an address space of 2^{16}, or 65536 memory locations in all. That's a very long list to scroll through.

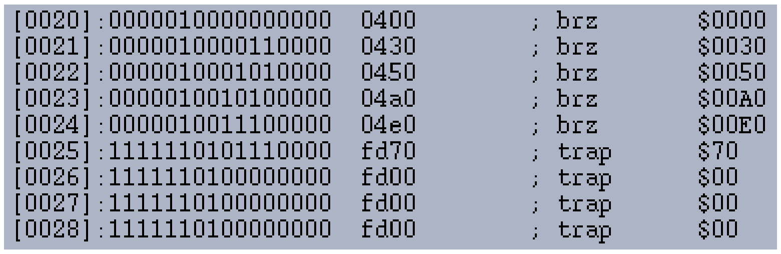


```
[3000]:0000000000000000  0000        ; brnop     $3000
[3001]:0000000000000000  0000        ; brnop     $3000
[3002]:0000000000000000  0000        ; brnop     $3000
[3003]:0000000000000000  0000        ; brnop     $3000
[3004]:0000000000000000  0000        ; brnop     $3000
[3005]:0000000000000000  0000        ; brnop     $3000
[3006]:0000000000000000  0000        ; brnop     $3000
```

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-2. 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. Here's a small piece of that section:

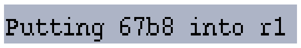


```
[0020]:0000010000000000  0400        ; brz      $0000
[0021]:0000010000110000  0430        ; brz      $0030
[0022]:0000010001010000  0450        ; brz      $0050
[0023]:0000010010100000  04a0        ; brz      $00A0
[0024]:0000010011100000  04e0        ; brz      $00E0
[0025]:1111110101110000  fd70        ; trap     $70
[0026]:1111110100000000  fd00        ; trap     $00
[0027]:1111110100000000  fd00        ; trap     $00
[0028]:1111110100000000  fd00        ; trap     $00
```

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.

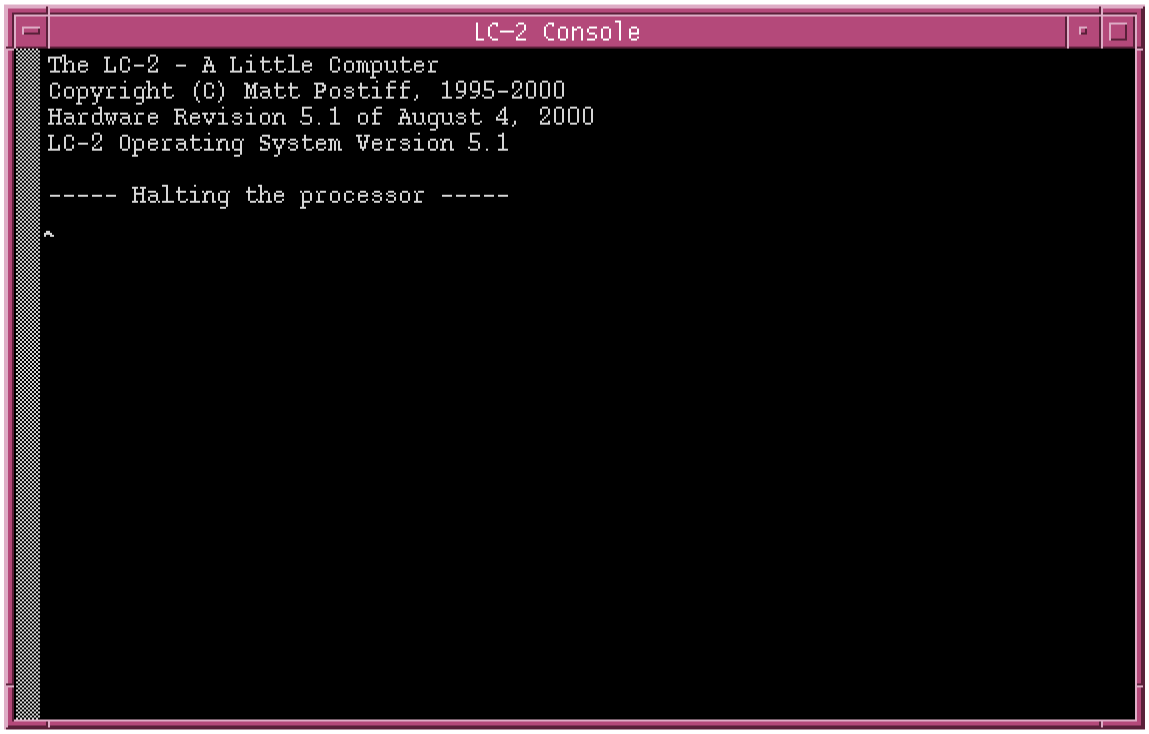
Information

The bottom section of the simulator may at times give you helpful information about what you've just done, such as



```
Putting 67b8 into r1
```

The Console Window



```
The LC-2 - A Little Computer
Copyright (C) Matt Postiff, 1995-2000
Hardware Revision 5.1 of August 4, 2000
LC-2 Operating System Version 5.1

----- Halting the processor -----

^
```

Before we go through the details of the simulator window, you should notice that a second window also appears when you run the simulator. It is rather inconspicuous, and has the vague title "LC-2 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.

The menus

File menu:

Load Program… brings up a popup window in which you can browse and choose the file you'd like to open.

Load&Run Script…* reads a script file and performs the commands in that file.

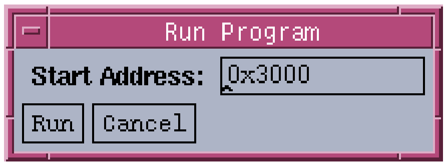
Run Script Command…* performs one script command.

Exit… closes the simulator.

Run menu:

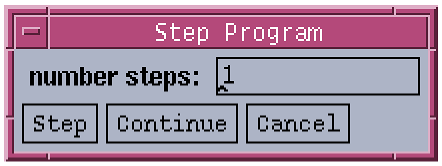
Run Program brings up this popup window:

* If you're a student, don't worry about these menu items. They're for your professor.



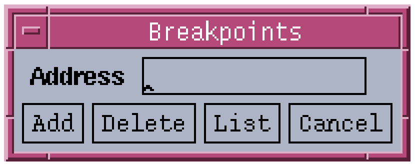
After you click **Run**, the program executes, starting from the **Start Address**, until it reaches a breakpoint or a Halt instruction is carried out.

Step Program… brings up this popup window:



Here, you can specify how many instructions to execute at a time. By clicking **Step**, you'll cause that number of instructions to execute. By clicking **Continue**, your program will run until a breakpoint or a halt.

Breakpoints… brings up this popup window:



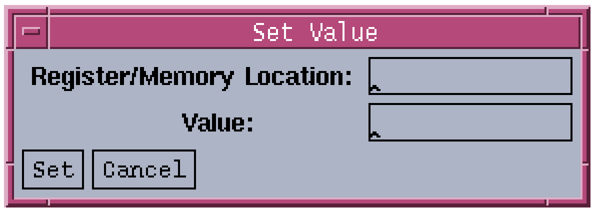
In the **Address** field, you should specify the address where you want to add or delete a breakpoint. If you click **List**, a summary of all breakpoints will appear in the Information part of the simulator window.

Set Values menu:

Reinitialize Machine clears all registers (with the exception of the PC, which is set to x3000) and all areas of memory that are available to be used by you.

Clear Registers sets all registers to zero, except the PC which is set to x3000.

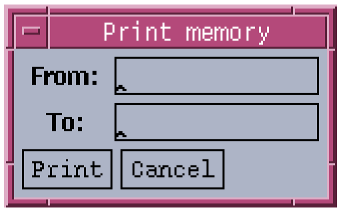
Set Register or memory brings up this popup window:



Here, you may specify any register or memory location address, and the 16-bit value you want it to contain. When you click **Set**, the new value will be applied.

Display menu:

Memory… brings up this popup window:



You can enter values into the **From** and **To** fields, and when you click **Print**, the range of addresses starting with the first and ending with the second will be displayed in the Memory section of the simulator window.

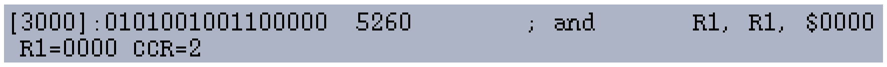
Registers causes a list of all the registers named in the CPU section of the simulator window, to be displayed along with their contents. The list appears in the Information section of the simulator window.

Refresh updates the simulator window with current information.

Options menu:

Step into traps is a toggle. When it's on, and you do **Run** and **Step Program…**, you'll execute each line of a trap routine if you encounter such a routine when stepping through your program. When it's off, you'll execute the trap routine all at once (not instruction by instruction) if you encounter one while stepping through your program.

Dump Instruction Trace is a toggle. When it's on, each instruction that gets executed by the simulator puts something like this into the Information part of the window:



```
[3000]:0101001001100000  5260        ; and       R1, R1, $0000
 R1=0000 CCR=2
```

This tells us that the instruction which just executed was at memory location x3000. It had the value x5260, which translated to "AND R1, R1, x0" in the LC-2 assembly language. It caused R1 to get a new value which was zero. It caused the condition codes to get the value 2, also known as 010, also known as N=0, Z=1, P=0. When the toggle is off, this information doesn't appear in the Information part of the window.

Right-clicking

If you click the right mouse button in the simulator interface, you'll get the following common options:

> **Load Program…**
> **Run Program…**
> **Step Program…**
> **Breakpoints…**
> **Run Script Command…**
> **Exit…**

These commands have the same effect as their corresponding entries in the menus mentioned earlier. For example, instead of going to the **File** menu and then choosing **Load Program…**, you can simply right-click somewhere in the simulator window, and choose **Load Program…**.