Guide to Using the Windows version of the LC-2 Simulator and LC2Edit

by

Kathy Buchheit
The University of Texas at Austin

©
copyright, Kathy Buchheit
2001

Guide to Using the Windows version of the LC-2 Simulator
and LC2Edit

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 the text editor (known as LC2Edit). 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 later in the semester.

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 two sections are meant as reference material, for LC2Edit, and for the simulator itself.

In other words,

		page
Chapter 1	**Creating a program for the simulator**	**2**
Chapter 2	**The simulator: what you see on the screen**	**7**
Chapter 3	**Running a program in the simulator**	**10**
Chapter 4	**Debugging programs in the simulator**	**15**
Chapter 5	**LC2Edit reference**	**24**
Chapter 6	**LC-2 Simulator reference, Windows version**	**28**

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 LC2Edit

If you're using Windows, there's another program in the same folder as the simulator, called LC2Edit.exe. Start that program by double-clicking on its icon, and you'll see a simple text editor with a few special additions.

Entering your program in machine language

You have the option to type your program into LC2Edit 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
```

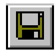
When you type this into LC2Edit, you'll probably be looking at a chart which tells you the format of each instruction, such as the one on page 94 of Chapter 5 of the textbook. So it may be easier for you to read your own code if you leave spaces between the different sections of each instruction. Also, 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 with LC2Edit. It 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 (comments after a semicolon are still an option):

```
3000
5260
5920
192A
E500
6680
14A1
1243
193F
0204
F025
```

 Saving your program

Click on this button or choose "Save" under the File menu. You probably want to make a new folder to save into, because you'll be creating more files in the same place when you turn your program into an object file. Call your program *addnums.bin* if you typed it in 1s and 0s. Call it *addnums.hex* if you typed it in hex.

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 LC2Edit. 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're using LC2Edit, one and only one of these buttons will make this happen:

How will you know which one? It depends whether you entered your program in 1s and 0s (**B** and an arrow), in hex (**X** and an arrow), or in assembly language (**asm** and an arrow).

When you press the appropriate button, a new file will be created in the same folder 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*.

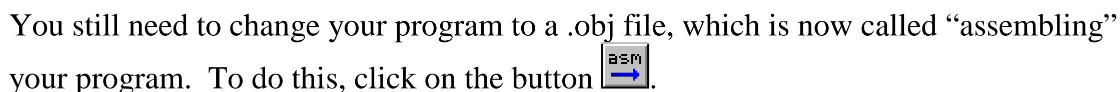
If you typed your program in 1s and 0s, or in hex, only one new file will appear: *addnums.obj*.

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, 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, click on the button .

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 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) 0W          .ORIG x3000
(3000) 5260  0101001001100000 (   2)             AND   R1 R1 #0
(3001) 5920  0101100100100000 (   3)             AND   R4 R4 #0
(3002) 192A  0001100100101010 (   4)             ADD   R4 R4 #10
(3003) E500  1110010100000000 (   5)             LEA   R2 x3100
(3004) 6680  0110011010000000 (   6) LOOP        LDR   R3 R2 #0
(3005) 14A1  0001010010100001 (   7)             ADD   R2 R2 #1
(3006) 1243  0001001001000011 (   8)             ADD   R1 R1 R3
(3007) 193F  0001100100111111 (   9)             ADD   R4 R4 #-1
(3008) 0204  0000001000000100 (  10)             BRP   LOOP
(3009) F025  1111000000100101 (  11)             TRAP  x25
```

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 #0

(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.

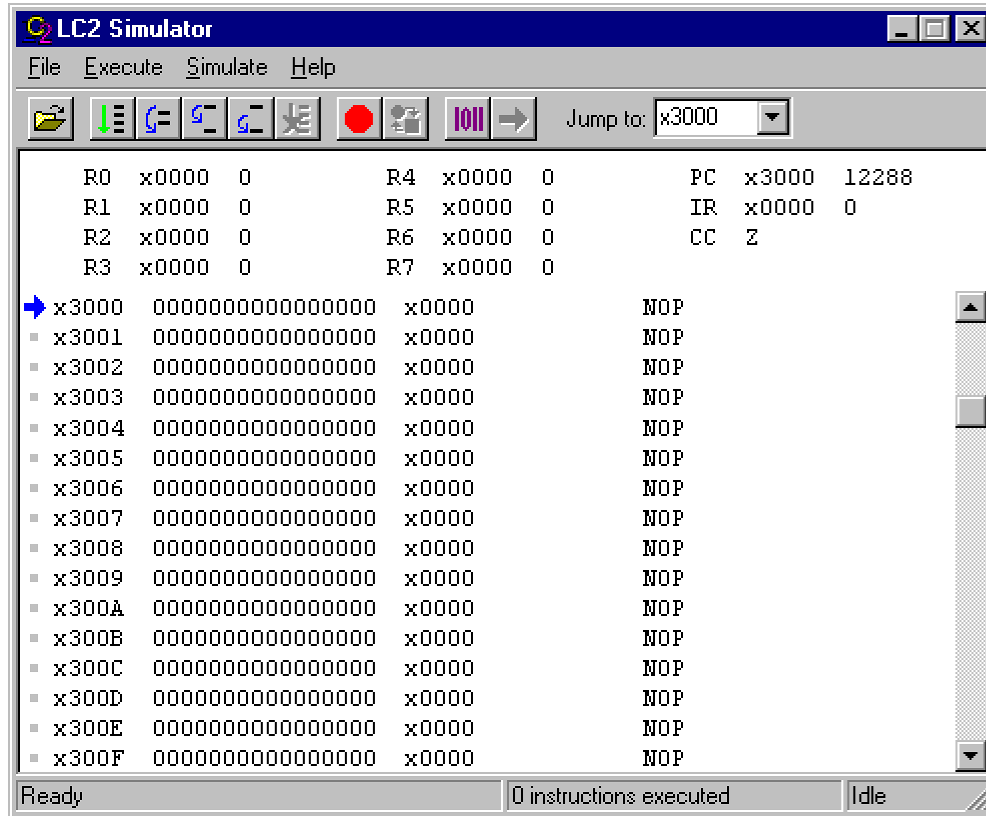
(6)
The instruction is the sixth line of the assembly language program. It will actually be the fifth line of the program once it gets loaded into memory in the simulator, since the line marked (1) just specifies the starting location. But we're counting assembly language lines now, not memory locations,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

LOOP
This is the label associated with the line.

LDR   R3 R2 #0
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

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



Chapter 6 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 menu items and toolbar buttons, notice the list of registers.

```
R0  x0000  0        R4  x0000  0        PC  x3000  12288
R1  x0000  0        R5  x0000  0        IR  x0000  0
R2  x0000  0        R6  x0000  0        CC  Z
R3  x0000  0        R7  x0000  0
```

Starting at the left, you see R0 through R3, and then skipping over to the fourth column, you see R4 through R7. Those are the eight registers that LC-2 instructions use as sources of data and destinations of results. The columns of x0000s and 0s are the contents of those registers, first in hex (that x at the front of the number always means "treat what follows as a hex number"), and then in decimal. When you launch the simulator, the temporary registers always contain zero.

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

```
R2  x0081  129
```

The last three columns at the top of the simulator show 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  x3000  12288
IR  x0000  0
CC  Z
```

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. It holds a zero when you launch the simulator, since no instruction is "current" yet.

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. Since only one of the three can have the value 1 at any time, the simulator just shows us the name of the register which currently has the value 1. (So when you start the simulator, N=0, Z=1, and P=0 by default.)

The memory

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

```
x3000  0000000000000000  x0000          NOP
x3001  0000000000000000  x0000          NOP
x3002  0000000000000000  x0000          NOP
x3003  0000000000000000  x0000          NOP
x3004  0000000000000000  x0000          NOP
x3005  0000000000000000  x0000          NOP
x3006  0000000000000000  x0000          NOP
```

Use the scrollbar at the right 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 "Jump to" box near the top of the interface, and enter the address (remember the little "x" before an address in hex) where you'd like to go.

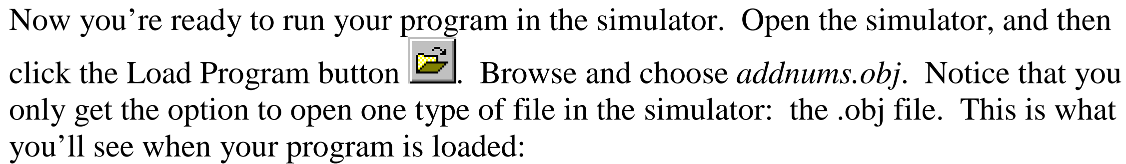
The first column in the long list of memory locations tells you the address of the location. The second column tells you the contents of a location, in binary. The third column also represents the contents, but in hex instead of binary, because that's sometimes easier to interpret. The fourth column is the assembly language interpretation of the contents of a

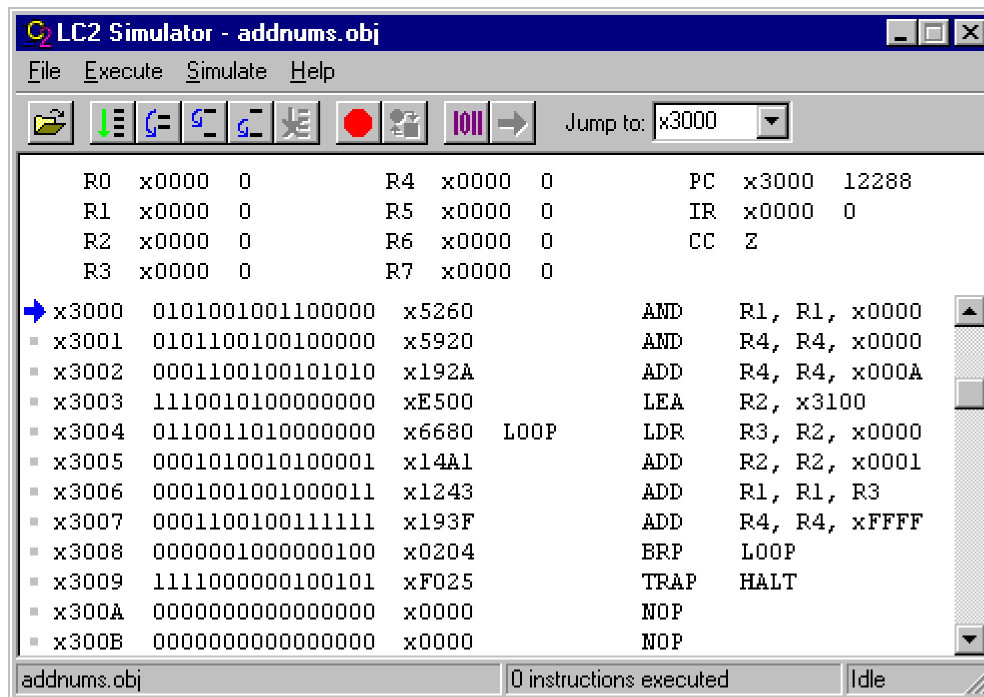
location. If a location contains an instruction, this assembly interpretation will be useful. If a location contains data, just ignore the fourth column entirely.

The Console Window

A second window also appears when you run the simulator. It is rather inconspicuous, and has the vague title “LC2 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.

Chapter 3
Running a program in the simulator

Now you're ready to run your program in the simulator. Open the simulator, and then click the Load Program button . Browse and choose *addnums.obj*. Notice that you only get the option to open one type of file in the simulator: the .obj file. This is what you'll see when your program is loaded:



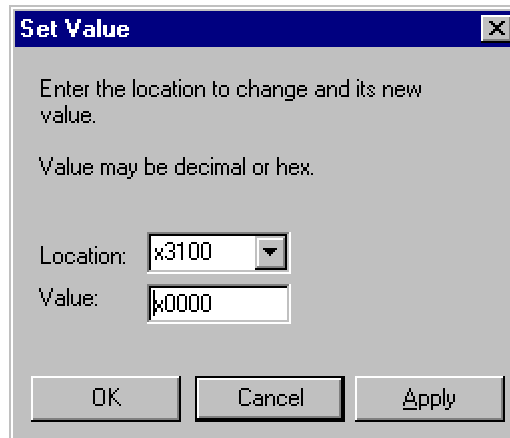
Notice that the first line of your program, no matter what format you originally used, is gone. That line specified where the program should be loaded in memory: x3000. As you can see if you scroll up a line or so, the locations before x3000 are still all 0s. Since nothing has happened yet (you haven't started running or stepping through your program), the temporary registers (R0 through R7) still contain all 0s, the PC is pointing to the first line of your program (as is the blue arrow), and the IR is empty.

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 in the "Jump to" box to the right of the buttons on the toolbar, and type the hex number x3100. When you press return, you'll jump x100 locations ahead in the memory display, so that location x3100 is the first one shown.

Now double-click anywhere on line x3100. You'll get this popup window:



In the "Value" box, type the hex number x3107, and choose OK. Now your first data location will look like this:

```
x3100  0011000100000111  x3107          ST     R0, x3107
```

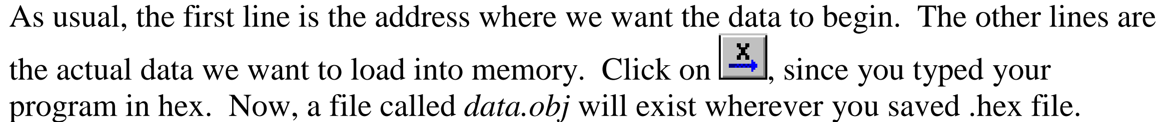
Notice that you get to see the binary representation, the hex representation, and some silly, useless assembly language representation (ST R0, x3107). Of course, this is data, not an instruction, but the LC-2 simulator doesn't know that. In fact, the contents of all memory locations are equal in the 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.

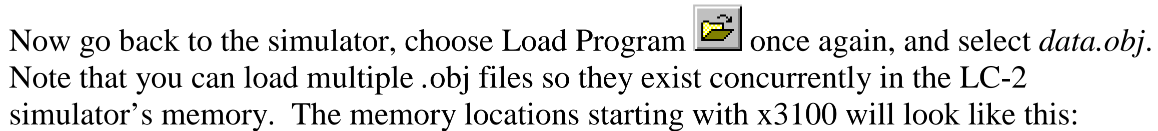
You can double-click on each line in turn and enter the data. If you only want to open the popup window once, keep changing the value of the Location field, enter the next number in the Value field, and click the Apply button. When you're done, click OK.

Second way: go back to the LC2Edit program, and enter this code in hex.

```
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 .

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. Click on , since you typed your program in hex. Now, a file called *data.obj* will exist wherever you saved .hex file.

Now go back to the simulator, choose 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:

```
▪ x3100  0011000100000111  x3107        ST    R0, x3107
▪ x3101  0010100000011001  x2819        LD    R4, x3019
▪ x3102  0000000100010000  x0110        NOP
▪ x3103  0000001100010000  x0310        BRP   x3110
▪ x3104  0000000100010000  x0110        NOP
▪ x3105  0001000100010000  x1110        ADD   R0, R4, R0
▪ x3106  0001000110110001  x11B1        ADD   R0, R6, xFFF1
▪ x3107  0000000000011001  x0019        NOP
▪ x3108  0000000000000111  x0007        NOP
▪ x3109  0000000000000100  x0004        NOP
▪ x310A  0000000000000000  x0000        NOP
```

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

Running your program

Click on the "Jump to" field, and choose x3000 as the location where you want to go.

This next step is VERY important: double-click on the little grey square in front of the line at address x3009.

```
▪ x3009  1111000000100101  xF025        TRAP  HALT
↑
double-click
here!
```

That sets a breakpoint on that line. 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 double-click, the line will look like this:

```
● x3009  1111000000100101  xF025        TRAP  HALT
```

That red blob is a stop sign. So we'll stop when we get to line x3009, before we run the instruction there.

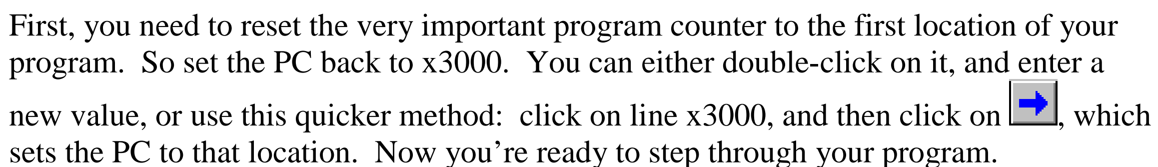
Now you're ready to run your program. Make sure the PC has the value x3000, because that's where the first instruction is. If it doesn't, double-click on the PC value near the top of the interface, and change it to x3000.

Now for the big moment: click on , the Run Program button!

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. (In decimal, the result is –32,459. It's negative because the first bit in x8135 is a 1, which is a negative 2's complement number.) You'll get a popup window telling you that you've reached a breakpoint, which in this case is the event when the PC gets the value x3009.

Stepping through your program

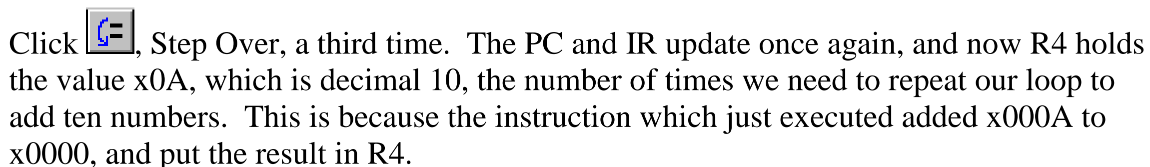
So now that you've seen your program run, you know it works. But that doesn't give you a good sense for what's actually going on in the LC-2 during the execution of each instruction. It's much more interesting to step through the program line by line, and see what happens. You'll need to do this quite 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. You can either double-click on it, and enter a new value, or use this quicker method: click on line x3000, and then click on , which sets the PC to that location. Now you're ready to step through your program.

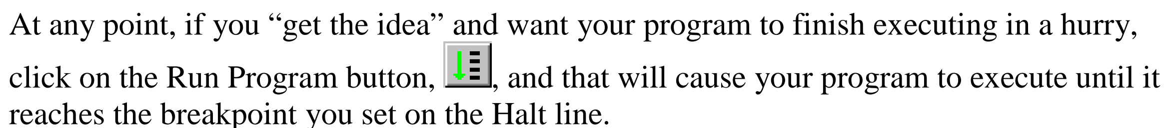
Click , Step Over, one time. A few interesting things just happened:

- R1 got cleared. (If you "cleaned up" by clearing R1 before you started, this won't be an exciting event.)
- The blue arrow, and the PC, both point to location x3001 now, which is the next instruction to run.
- 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 Over, for a second time. Again, notice the new values for the PC and IR. The second instruction clears R4.

Click , Step Over, 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 the Run Program button, , 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 LC2Edit. It should look like this:

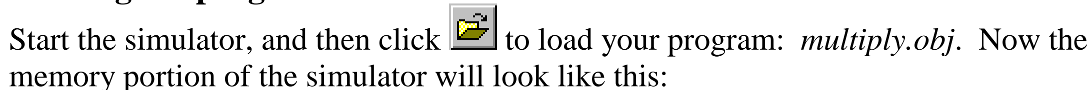
```
0011 0010 0000 0000       ;the address where the program begins: x3200
0101 010 010 1 00000      ;clear R2
0001 010 010 0 00 100     ;add R4 to R2, put result in R2
0001 101 101 1 11111      ;subtract 1 from R5, put result in R5
0000 011 000000001        ;branch to location x3201 if the result is 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 LC2Edit, save it as *multiply.bin* and then click on to convert it to an .obj file.

Loading the program into the simulator



Start the simulator, and then click to load your program: *multiply.obj*. Now the memory portion of the simulator will look like this:

```
x3200  0101010010100000  x54A0    AND   R2, R2, x0000
x3201  0001010010000100  x1484    ADD   R2, R2, R4
x3202  0001101101111111  x1B7F    ADD   R5, R5, xFFFF
x3203  0000011000000001  x0601    BRZP  x3201
x3204  1111000000100101  xF025    TRAP  HALT
x3205  0000000000000000  x0000    NOP
x3206  0000000000000000  x0000    NOP
```

Also notice that the PC contains the value x3200, which corresponds to the blue arrow pointing to that line – 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 double-clicking on the little gray square at the beginning of that line.

```
▪ x3204   1111000000100101   xF025          TRAP   HALT
↑
double-click
here!
```

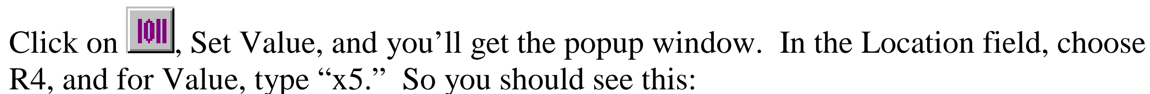
Now line x3204 should look like this:

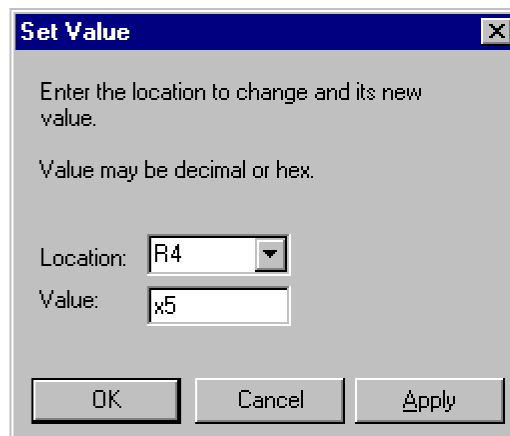
```
● x3204   1111000000100101   xF025          TRAP   HALT
```

That red stop sign 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

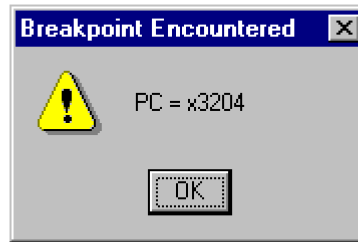
Before you run your program for the first time, you need to put some values in R4 and R5 so that they'll be multiplied (or not, in this case!). How should you choose values to test? Common sense will help you here. 0 and 1 are probably bad choices to start with, since they'll be rather boring. (It would 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 , Set Value, and you'll get the popup window. In the Location field, choose R4, and for Value, type "x5." So you should see this:



Click Apply (which means "set the location I just picked to the value I just entered, but leave this popup window open so I can do more stuff"), and now choose R5 in the Location field, and type "x3" in the Value field. Click OK to close the popup window. Now your registers are set, and you're ready to try running your program.

To run your program until the breakpoint, click on [icon], Run Program. In a fraction of a second, you'll get a popup window that looks like this:

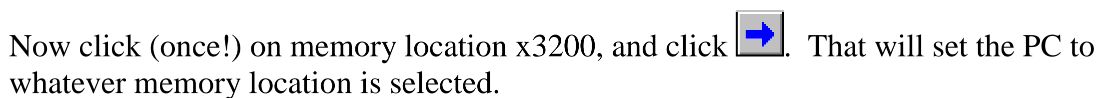


That happened because you set a breakpoint at your halt line. Click OK to close the popup, and then take a look at R2, which is supposed to contain your result now that you've run all but the last line of the program. As you realize, 3 * 5 = 15 in decimal, but R2 now contains 20 in decimal (which is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

Stepping through the multiply program

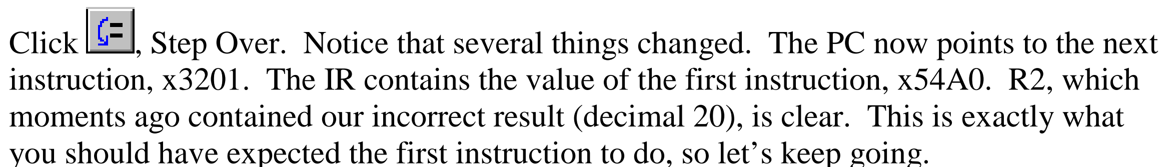
One option for debugging your program is to step through the entire multiply program from beginning to end. Since you have a loop, let's approach debugging a different way. First, let's try stepping through one iteration of the loop to make sure that each instruction does what you think it should.

Double-click on the R5, near the top of the interface. That will bring up a "Set Value" popup window. Set the value of R5 to x3, and click OK to close the window.

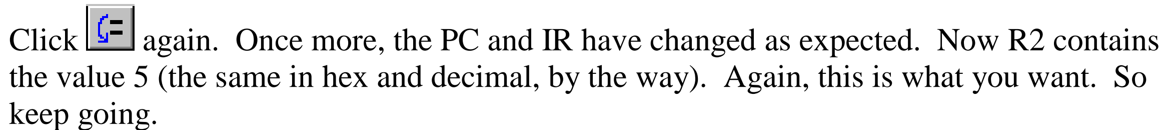


Now click (once!) on memory location x3200, and click [icon]. That will set the PC to whatever memory location is selected.

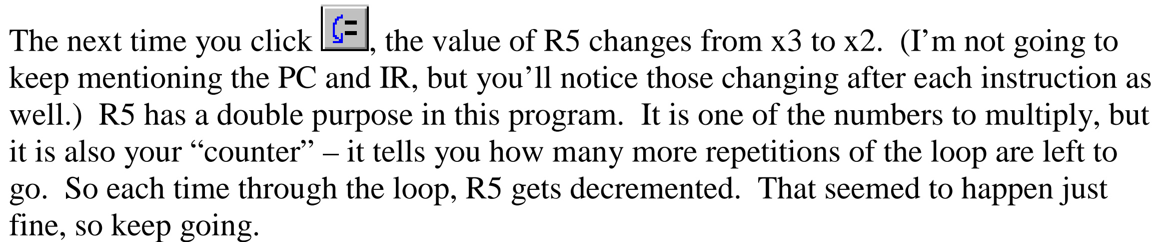
The blue arrow is now pointing to the first line, and the two registers you'll need are initialized to the values you want. So you're ready to step through the program, in your first attempt to figure out what's going wrong.



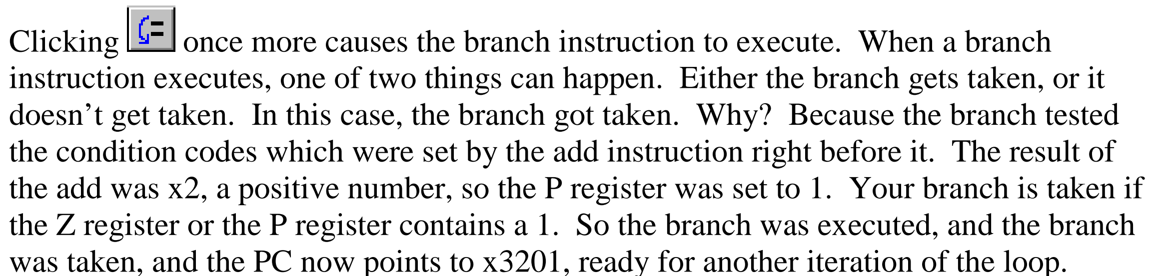
Click [icon], Step Over. Notice that several things changed. The PC now points to the next instruction, x3201. The IR contains the value of the first instruction, x54A0. R2, which moments ago contained our incorrect result (decimal 20), is clear. This is exactly what you should have expected the first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



Click 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 , 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.

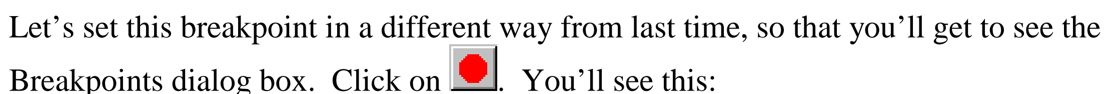


Clicking once more causes the branch instruction to execute. When a branch instruction executes, one of two things can happen. Either the branch gets taken, or it doesn't get taken. In this case, the branch got taken. Why? Because the branch tested the condition codes which were set by the add instruction 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.

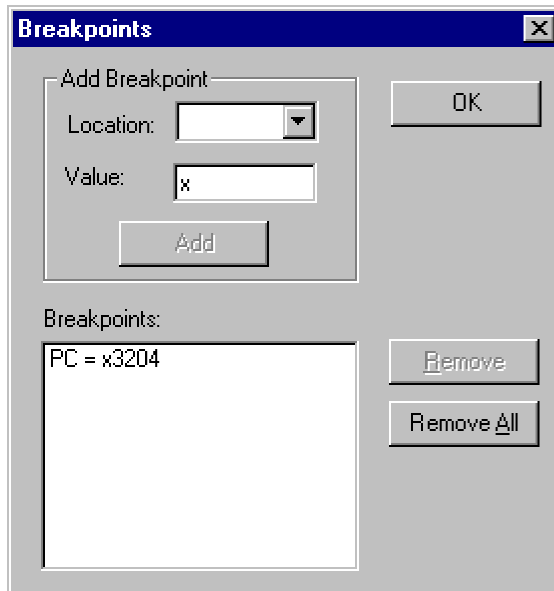
Stepping through the program for one repetition of the loop has shown that there's nothing wrong with any of the individual instructions in the loop. Maybe the problem lies in the way the loop is set up instead, so let's try another approach.

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).



Let's set this breakpoint in a different way from last time, so that you'll get to see the Breakpoints dialog box. Click on . You'll see this:



If you click on the drop-down arrow next to the Location field, you'll see that you have all sorts of choices:

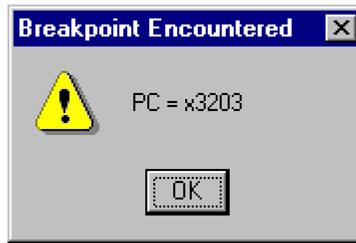
PC
x
IR
CC
R0
R1
R2
R3
R4
R5
R6
R7

Basically, you can have the simulator pause when it notices that any one of those registers or memory locations has the value that you specify. So if you want, you could set a breakpoint so that the simulator would pause when R0 contains x00FF (choose R0 for Location, and choose x00FF for Value), or when the condition codes have Z = 0 (choose CC for Location, and choose Z for Value), or when memory location x4000 contains x1234 (choose x4000 for Location, and choose x1234 for Value).

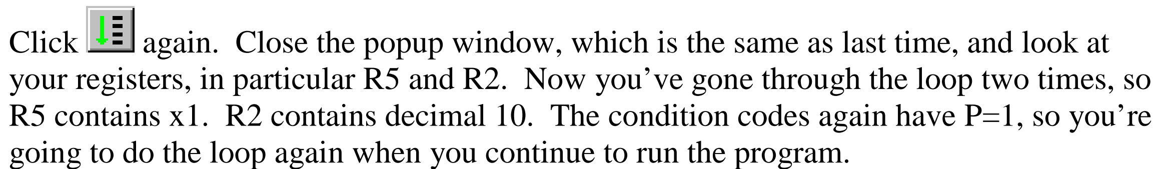
In this case, you want to choose PC for your Location, and type x3203 for your Value. Click Add. Now you have two items in your list of breakpoints, both having to do with the PC. The simulator will pause whenever the PC gets the value x3203, or the value x3204. Click OK to close this window.

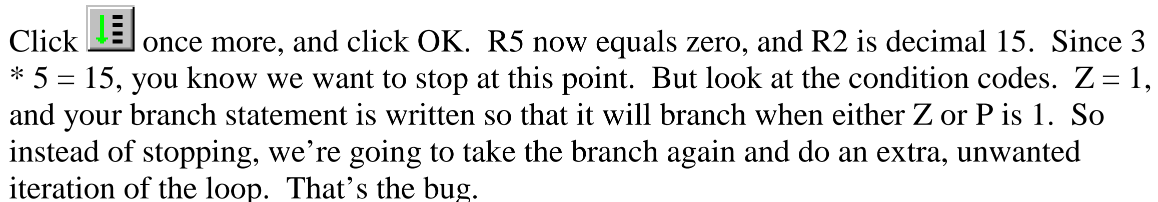
Now you'll need to set the PC to x3200, and R5 to x3 the same way you did earlier.

Click , Run Program. Almost immediately, you'll see this:



Click OK, and notice what has changed in your registers. The blue arrow and the PC both point 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.

Click  again. Close the popup window, which is the same as last time, and 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.

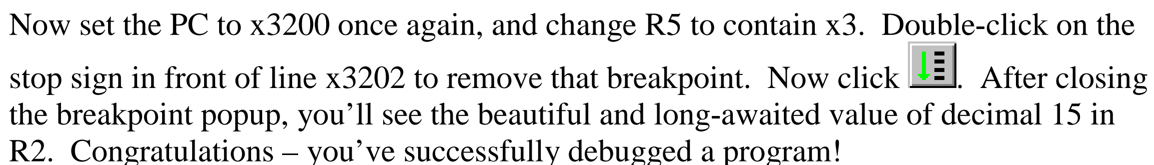
Click  once more, and click OK. R5 now equals zero, and R2 is decimal 15. Since 3 * 5 = 15, you know we want to stop at this point. But look at the condition codes. Z = 1, and your branch statement is written so that it will branch when either Z or P is 1. So instead of stopping, we're going to take 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 LC2Edit, 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, and you want to just change it quickly in the simulator, double-click on line x3203. When the Set Value popup window appears, change the Value from x0601 to x0201. (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. Double-click on the stop sign in front of line x3202 to remove that breakpoint. Now click . After closing the breakpoint 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 LC2Edit

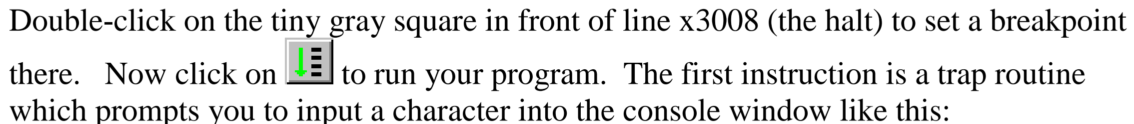
The purpose of this program is to request two integers (between 0 and 9) from the user, add them, and display the result (which must also be between 0 and 9) in the console window. Our buggy version of the program looks like this:

```
        .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 LC2Edit, and assemble it by clicking on .

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.

Double-click on the tiny gray square in front of line x3008 (the halt) to set a breakpoint there. Now click on  to run your program. 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. (Notice the "_____ instructions executed" message at the bottom of the simulator, where the _____ is a quickly growing number. That will grow indefinitely, until you go ahead and input a character.) 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"

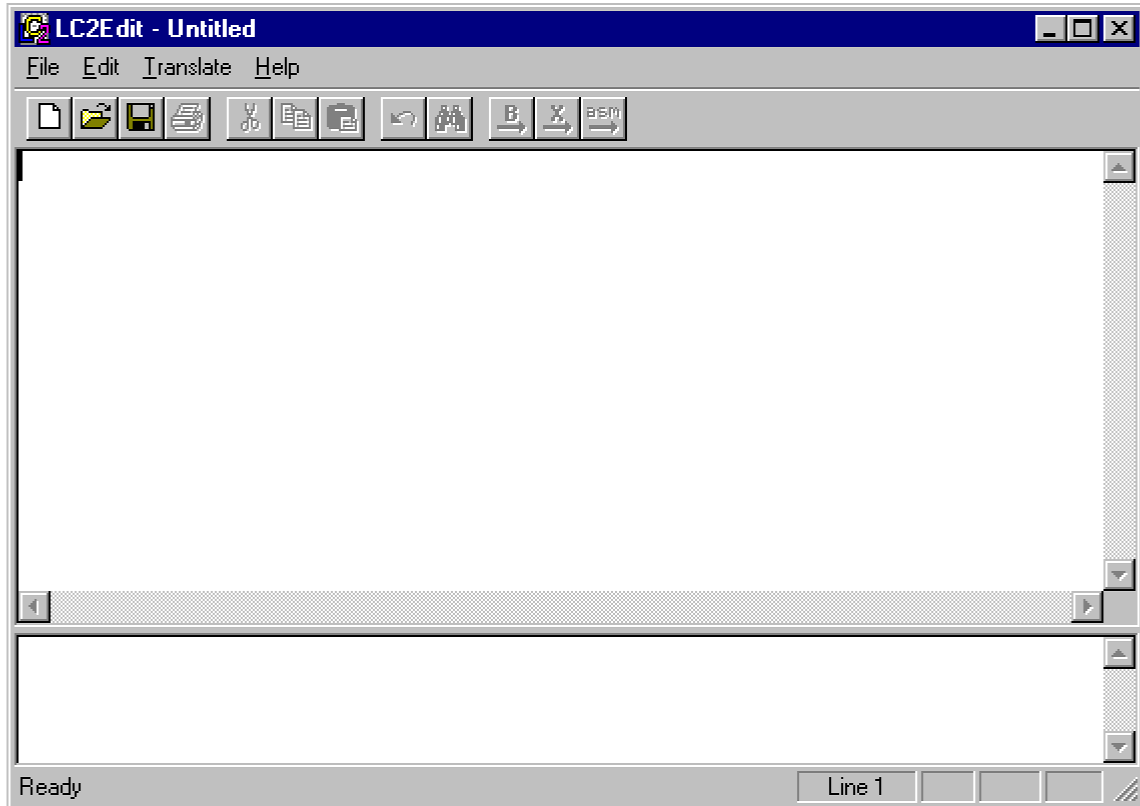
```
        ADD       R0,R0,R5    ;convert next ASCII number to numerical value
        ADD       R2,R0,R1    ;add the two integers
        ADD       R2,R2,R6    ;convert the sum to its ASCII representation
        LEA       R0,MESG     ;load the address of the message string
        TRAP      x22         ;"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 LC2Edit, 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
LC2Edit reference

Here is what you see when you open LC2Edit:



The upper and lower windows

The upper window, with the flashing cursor, is where you'll type your program, either in machine language (binary or hex), or LC-2 assembly language. For examples, see Chapters 3 and 4.

The lower window will give you messages when you convert your program to an object file. If your program has no errors, you'll see a message like this:

```
Assembling C:\WINNT\Profiles\kathy\Desktop\test.asm...
Starting Pass 1...
Pass 1 - 0 error(s)
Starting Pass 2...
Pass 2 - 0 error(s)
```

If you're not so lucky, you may see something like this instead:

```
Assembling C:\WINNT\Profiles\kathy\Desktop\test.asm...
Starting Pass 1...
Line 8: Unrecognized opcode or syntax error at or before 'r0'
Pass 1 - 1 error(s)
```

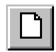
in which case you know you have some debugging to do even before you get to run your program in the simulator.

Figuring out what your error messages mean

Go to the Help menu of LC2Edit, and choose "Contents…." Click the tab that says "Index." You'll see a list of topics. Double-click on "Error Messages." This will bring up a list, with descriptions, of all possible errors that you could get during your conversion (or assembly) to an .obj file.

The buttons on the toolbar

Many of the commands in the menus are easier to use by just clicking on a button on the toolbar. While you're using LC2Edit, you can rest your cursor over any button, and a "tool tip" will appear to tell you what the button does.



New

This clears the upper window so that you can start fresh. If you haven't saved what you're working on, you'll be prompted to save your file first.

Open

By clicking this, you'll be able to browse the computer for text files. The default file extensions which Open looks for are .bin, .hex, and .asm, but you can also choose "all files" in case you named something with a different file extension.

Save

If you click this and you haven't saved your file yet, you'll get a popup window called "Save Source Code As." Otherwise, this button will automatically replace the previous version of your file.

Print

This brings up the typical popup window to print your file, so that you can set the Properties before printing, if you wish to.

Cut

This button will copy the highlighted text onto the clipboard, and remove it.

Copy

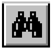
This button will copy the highlighted text onto the clipboard without removing it.

Paste

This inserts the text from the clipboard to the location of your cursor.

Undo

You can undo your last action (within reason of course … no undoing a save or a print!) and then by clicking again, redo what you just undid.



Find

This brings up a popup window in which you can specify whether to search up or down from the location of your cursor, and whether you'd like to match the case or not. (Matching the case means that if you type "add," that will match "add" but not "Add" or "ADD" or "aDd."

Convert from base 2

When you've finished typing your program in 1's and 0's, and you'd like to (attempt to) convert it to an .obj file, click this button. If you have errors, you'll see a report of them in the lower window of the interface.

Convert from base 16

When you've finished typing your program in hex, and you'd like to (attempt to) convert it to an .obj file, click this button. If you have errors, you'll see a report of them in the lower window of the interface.

Assemble

When you've finished typing your program in the LC-2 assembly language, and you'd like to (attempt to) convert it to an .obj file, also known as assembling your file, click this button. If you have errors, you'll see a report of them in the lower window of the interface.

The menus

Most of the items on the menus do the same things as previously mentioned buttons. Some options, however, are only available through menus.

File menu

"New" does the same thing as the button mentioned above.

"Open…" does the same thing as the button mentioned above.

"Save" does the same thing as the button mentioned above.

"Save As…" gives you the option of saving your file under a different name than its current name.

“Print…” does the same thing as the button mentioned above.

“Print Setup…” brings up a popup window where you can change properties of printing, such as which printer to use, what size paper you like, etc.

“Exit” closes LC2Edit.

Edit menu
“Undo” does the same thing as the button mentioned above.

“Cut” does the same thing as the button mentioned above.

“Copy” does the same thing as the button mentioned above.

“Paste” does the same thing as the button mentioned above.

“Find…” does the same thing as the button mentioned above.

“Find Next” searches for the same word which you mostly recently specified under “Find.”

“Replace…” brings up a popup window in which you can say which word(s) to look for, and what to replace that with. You get the option of replacing just one instance, or every instance in your file.

Translate menu
“Convert Base 2” does the same thing as the “Convert from base 2” button mentioned above.

“Convert Base 16” does the same thing as the “Convert from base 16” button mentioned above.

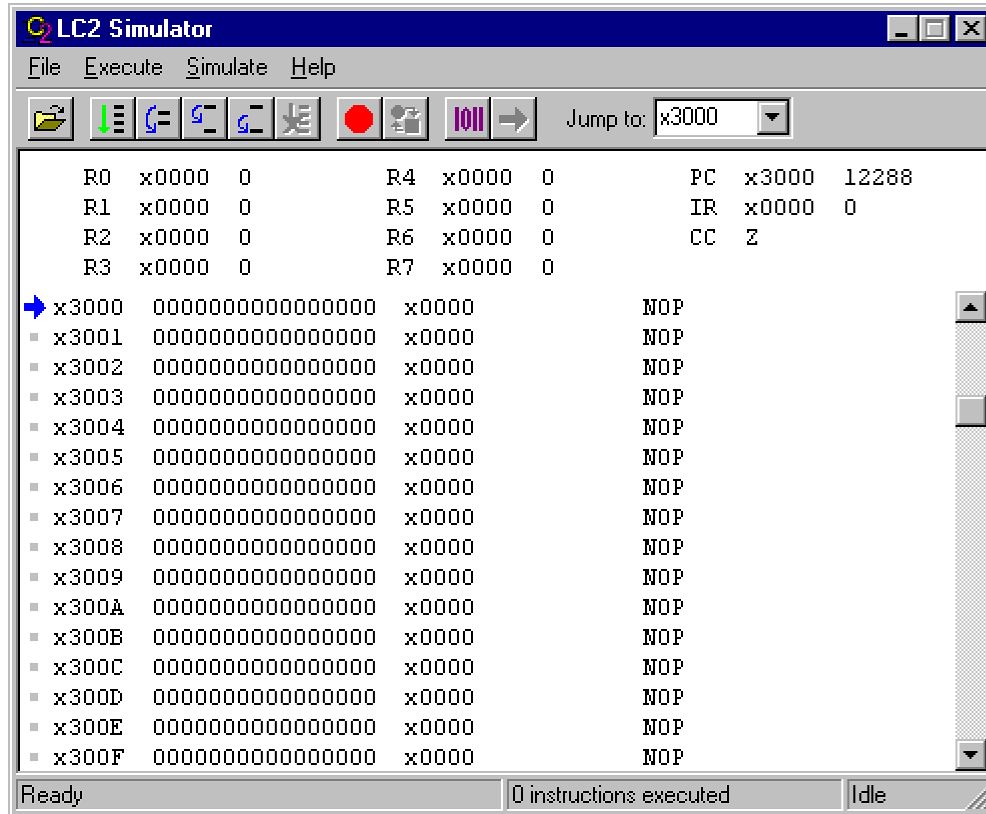
“Assemble” does the same thing as the button mentioned above.

Help menu
“Contents…” brings up a help window with an overview, and various topics. If you need more specific details on some topic, look here.

“About LC2Edit…” gives you some information about the author and copyright of LC2Edit.

Chapter 6
LC-2 Simulator reference, Windows version

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



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.

```
R0  x0000  0        R4  x0000  0        PC  x3000  12288
R1  x0000  0        R5  x0000  0        IR  x0000  0
R2  x0000  0        R6  x0000  0        CC  Z
R3  x0000  0        R7  x0000  0
```

R0 through R7 are the eight registers where LC-2 instructions can obtain their sources or store their results. The columns of x0000s and 0s are the contents of those registers, first in hex, and then in decimal.

The last three columns show 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. It will always contain the value zero when you start the simulator, because until you start running a program, there is no such thing as the "current instruction." 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. You don't see all three listed, because the author of the simulator was clever, and realized that only one of the three registers can have the value 1 at any time. The other two are guaranteed to be zero. So next to CC, you'll only see one letter: N, Z, or P. When you launch the simulator, the Z register is set to 1, and N and P are set to 0. So you see a Z listed for the condition codes. This will change when you execute any instruction that changes the condition codes (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 right 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. (That's why we have a "Jump to" option at the top, but we'll get to that.)

```
x3000  0000000000000000  x0000          NOP
x3001  0000000000000000  x0000          NOP
x3002  0000000000000000  x0000          NOP
x3003  0000000000000000  x0000          NOP
x3004  0000000000000000  x0000          NOP
x3005  0000000000000000  x0000          NOP
x3006  0000000000000000  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 last column in the long list of memory contains 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:

```
x0020  0000010000000000  x0400        BRZ    x0000
x0021  0000010000110000  x0430        BRZ    x0030
x0022  0000010001010000  x0450        BRZ    x0050
x0023  0000010010100000  x04A0        BRZ    x00A0
x0024  0000010011100000  x04E0        BRZ    x00E0
x0025  1111110101110000  xFD70        TRAP   x70
x0026  1111110100000000  xFD00        TRAP   x00
x0027  1111110100000000  xFD00        TRAP   x00
x0028  1111110100000000  xFD00        TRAP   x00
```

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.

The blue arrow

When you launch the simulator, and during the execution of your program, you see a blue arrow which points to one location in memory.

```
➡ x3000  0000000000000000  x0000        NOP
```

Here's a hint as to why the blue arrow is currently pointing to location x3000.

```
PC  x3000  12288
```

The arrow lets you know which instruction is next in your program. Since the program counter is currently telling us that the instruction at location x3000 is next, the arrow points to that line in memory.

The "jump to:" box

At the top right corner of the simulator, you'll see an extremely useful element of the interface.

Since memory is so huge (2^{16} locations), and the scrollbar doesn't give very precise control over how you can move through that long list, you'll find yourself using this trick quite often. Click on the hex number (currently x3000), and type a new location. As soon as you press the Enter key, your memory list will adjust so that the address you typed will be the first one listed. If you click on the drop-down arrow, you'll see a list of the recent locations to which you have jumped.

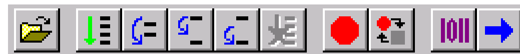
One reminder: the simulator understands both decimal and hex values, but you nearly always want to refer to a location in hex. So don't forget to put the "x" before your number. Otherwise it will be interpreted as a decimal value, and you'll go to who-knows-where.

The Console Window

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 "LC2 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 buttons on the toolbar

Let's go through the function of each button on the simulator's toolbar. (If you rest the cursor on top of any button for a moment, a "tool tip" will appear to tell you its name in case you forget or don't know.)

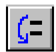


Load Program

This lets you browse and open a file of one type only: an .obj file. These files can be created in LC2Edit (see the section discussing this program for more details).

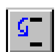
Run Program

This will cause your program to execute until one of two things stops it: the HALT routine (which stops the clock), or a breakpoint that you've set.



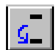
Step Over

This will execute one line of your program, and then wait with the blue arrow pointing to the following instruction. If the line to be executed is a JSR or JSRR or TRAP instruction, the simulator will execute everything contained within the subroutine without stopping, and then wait with the blue arrow pointing to the following instruction. So it "steps over" the subroutine without making you wade through it.



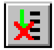
Step Into

This will execute one line of your program, with a major difference from the Step Over command. If the line to be executed is a JSR or JSRR or TRAP instruction, you will "step into" the subroutine. The blue arrow will end up pointing to the first instruction of the subroutine. If you did this accidentally, read on.



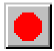
Step Out

If you're in a subroutine (caused by following a JSR or JSRR or TRAP instruction), and you would like to hop to the end of it and return to the section of the program that called the subroutine, click this button. You'll end up with the blue arrow pointing to the line of your program directly following the JSR or JSRR or TRAP instruction. More accurately, you'll end up at the line whose address was put in R7 when the subroutine was called.



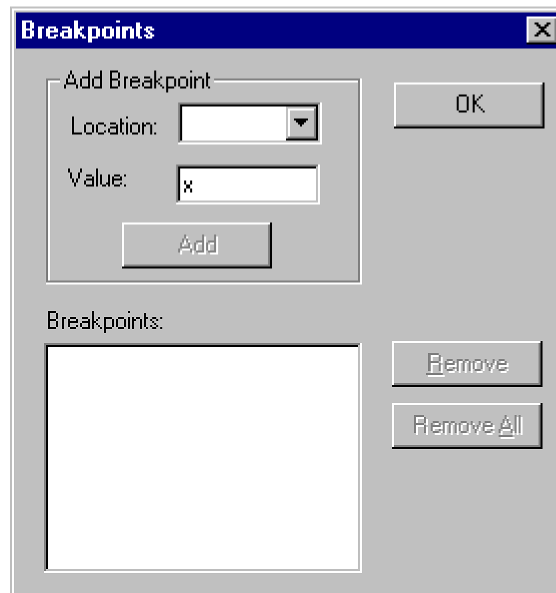
Stop Execution

If you ever get caught in an endless loop, you're more likely to panic than to read this guide. So remember this button. It will stop everything.



Breakpoints

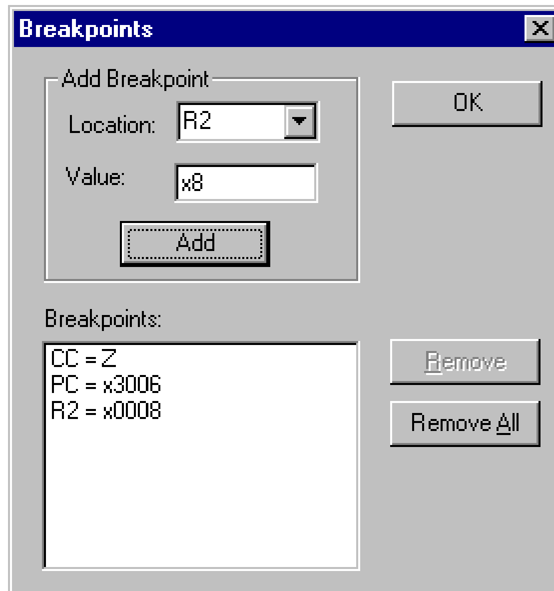
This button brings up the following dialog box. Here you have the option to set breakpoints depending on the values of the PC, IR, CC, R0 through R7, or any memory location. Breakpoints and their uses are discussed in the chapter about debugging.



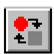
Click on the drop-down arrow next to Location. You'll see a list of those registers, along with the mysterious option "x." This "x" is the option you want if you're going to specify a memory location in hex. For example, you want the Location option to read "x3008" if you want the simulator to stop and wait for your input when the contents of memory location x3008 are the value you specified in the Value text field.

The Value field should specify a hex (starting with "x") or decimal (not starting with "x") value which can be represented with 16 bits. There is one exception to this. If you choose "CC" as your Location, you should specify "N," "Z," or "P" as your Value.

To add only the breakpoint you specified, and then close the popup window, click OK. To add more than one breakpoint in one sitting, fill in a new Location and Value, and then click Add. Let's say you've added a few breakpoints: one to pause the simulator when the CC have the value Z, one to pause the simulator when it gets to line x3006, and one to pause the simulator when R2 gets the value x8. Your dialog box would look like this.



Now you could choose one of the breakpoints listed in the box, and choose Remove to delete it. If you want to delete all breakpoints from the list, choose Remove All.

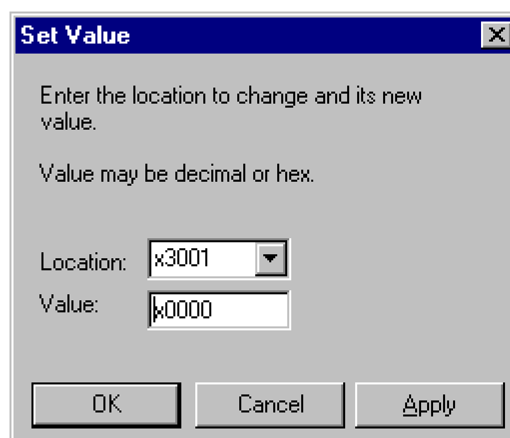


Toggle Breakpoint

If you click on any line of your program, and then this button, you will have a breakpoint set on that line, or cleared from that line, depending on whether there was a breakpoint on that line to begin with.

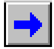
Set Value

This button brings up a popup window which lets you change the values of any register or memory location.



The drop-down arrow lists the registers, along with that mysterious "x," which you now know is the hex symbol before an address location. As the window tells you, the value can be entered in hex or decimal. Remember the "x" before a hex value. No "x" means that your number will be interpreted as decimal.

When you click the Apply button, the value you've entered in the Value box goes into the location you've entered in the Location box, but the window stays open so that you can easily enter more values into more locations. When you're done and you want the window to go away, click OK.



Set PC to selected location
This button is extremely useful. Let's say you're looking through your program, and you'd like to step through one particular section. The program counter is pointing to somewhere else entirely. If you click on the memory location where you'd like to begin, and then this button, the PC will hold the value of the memory location you just selected.

The menus

Most of the items on the menus do the same things as previously mentioned buttons. Some options, however, are only available through menus.

File menu

"Load Program…" does the same thing as the button mentioned above.

"Reinitialize Machine" resets all registers and memory locations to their default values. If you've made a major mess of things and want to get a clean start, this is a good option.

"Reload Program" loads the program that you're currently working on, and resets the PC to its beginning. If you edited your program and made a new .obj file, do this to load the new version. If you don't have a program loaded at the moment, this option won't be available.

"Randomize Machine" is one of my personal favorites. This sets all the registers, and all the memory locations which aren't holding important values (like TRAP routines) to random values. It's a good idea to do this before loading your program, when you think you have all your bugs worked out. That way, if you're assuming that some location is going to start with the value zero (which you should never assume), a new bug will present itself.

"Clear Console" erases all text from the console window.

"Exit" is, I hope, self-explanatory!

Execute menu

"Run" does the same thing as the "Run Program" button (see above).

"Stop" does the same thing as the "Stop Execution" button (see above).

"Step Over," "Step Into," and "Step Out" do the same things as their respective buttons (see above).

Simulate menu
"Set Value" does the same as its button (see above).

"Set PC to selection location" does the same thing as the button called "Set PC to selected location" (see above).

"Breakpoints" and "Toggle Breakpoint" do the same things as their identically titled buttons (see above).

"Display Follows PC" is a toggle which can be checkmarked or not. If it is checked, the memory location that contains the currently running instruction will always be showing (the memory section will scroll to follow the code). If it is unchecked, you can scroll to a particular location in memory and stay there no matter what instructions are executing. This could be helpful if you need to watch some data locations change, and your code isn't nearby in memory.

Help menu
"Contents…" brings up an index of topics. Choose one to get an explanation of that topic. If you find this guide cumbersome or lacking in any way, the help menu is a good place to find a second opinion or additional explanation.

"About Simulate" brings up copyright information and the name and email address of its author.