Guide to Using LC3Tools

by

Chirag Sakhuja
(based on Kathy Buchheit's guide)
The University of Texas at Austin

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

How this guide is arranged

The first section gives you a quick introduction to the interface.

The second chapter walks you through entering your first program, in machine language, into the text editor. You'll also find information about writing assembly language programs, but you'll probably skip that part until you've learned the LC-3 assembly language later in the semester. The third section shows you how to watch the effects of the program you just wrote in the simulator.

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

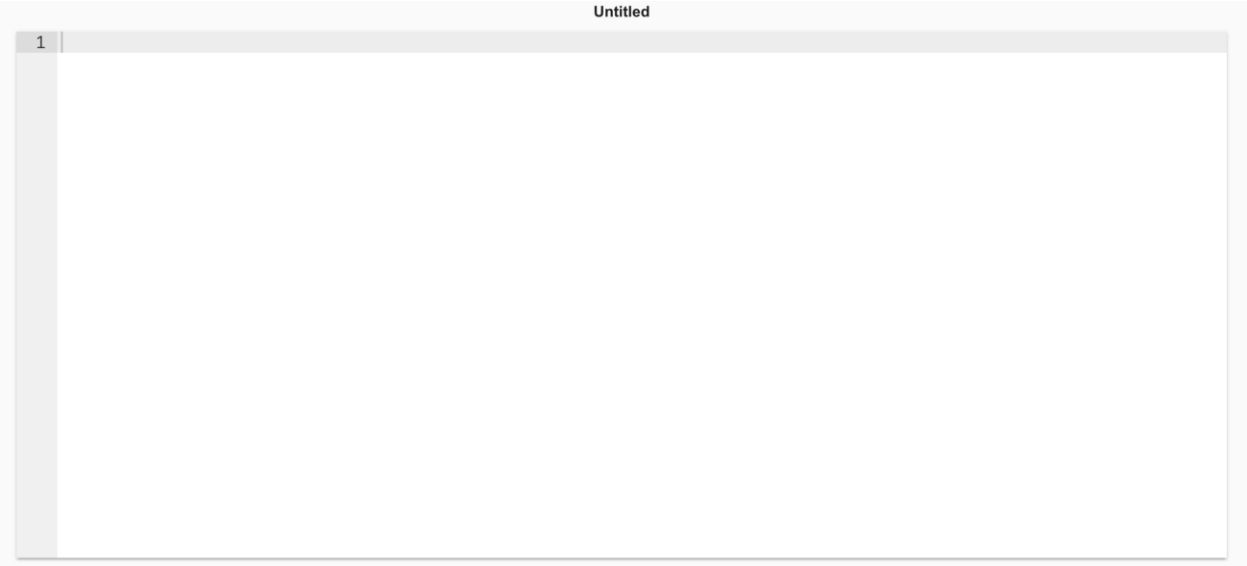
In other words,

Chapter 1: What you see on the screen

This chapter just provides a brief reference to the different parts of LC3Tools. As you continue through the chapters, the usage of each of the buttons and panels will become clearer.

When you launch LC3Tools, you see this.

The main panel in the center of the screen is the text editor. This is where you type out binary or assembly programs. The name of the file being edited is at the top.

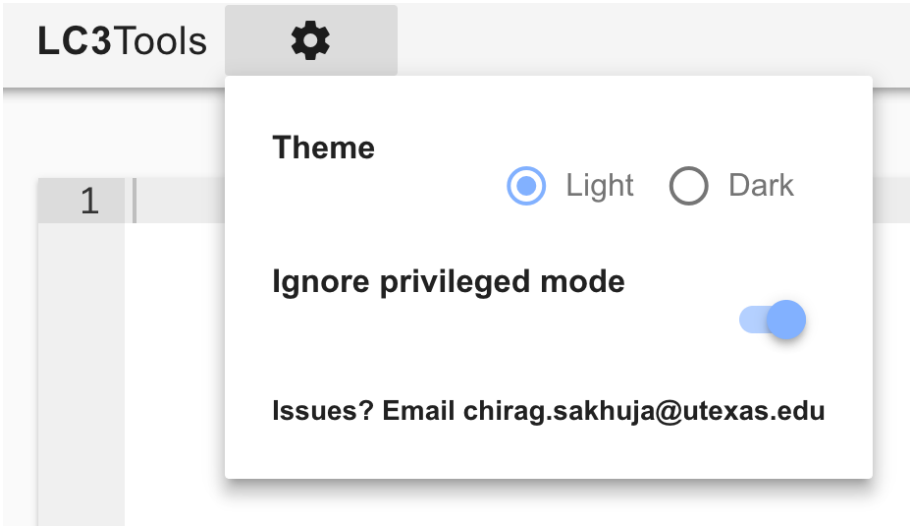


Below the text editor is the console, which shows messages about the Assembly or Conversion process. When you first open LC3Tools, nothing will be displayed.

The panel to the left contains the following buttons from top to bottom:

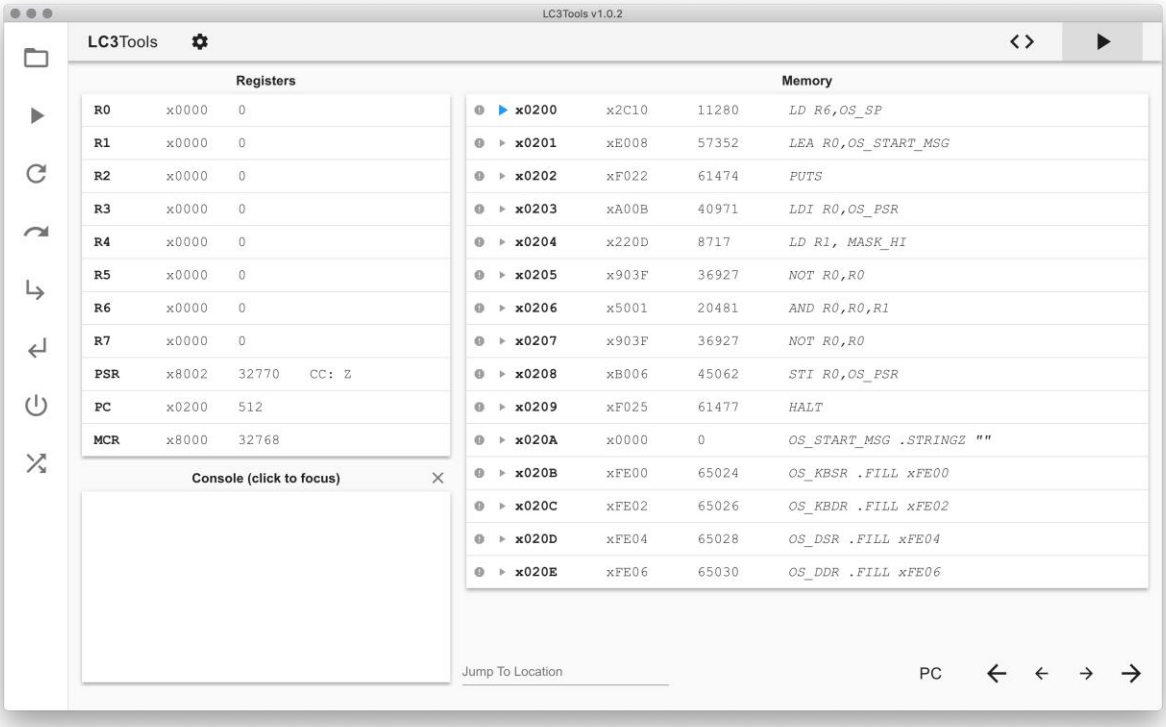
1. Create a new file
2. Save a file
3. Open an existing file
4. Assemble an .asm file or convert a .bin file into an .obj file

The toolbar at the top allows you to change the theme and privilege mode. The privilege mode is only relevant to the simulator section, and it will be addressed in chapter 4.

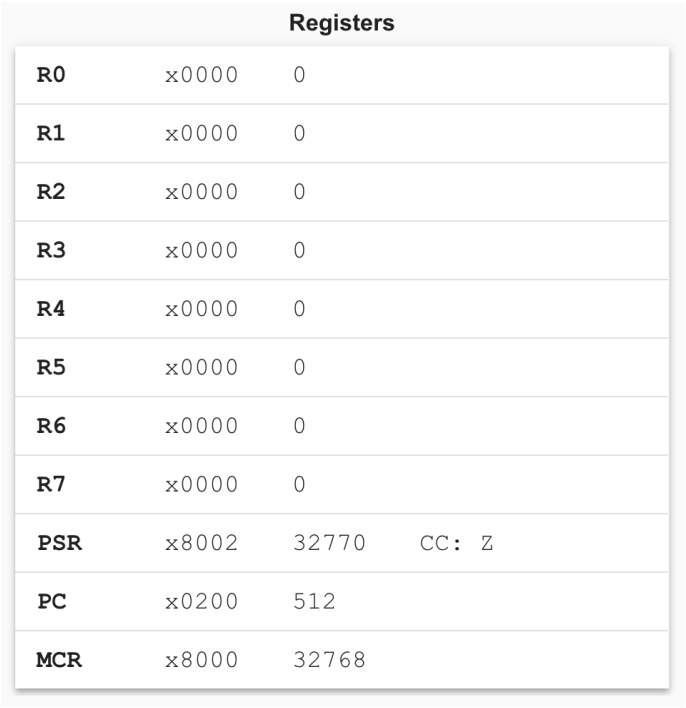


Finally, the top right of the screen allows you to switch between the editor and the simulator tabs.

Clicking on the simulator button pulls up the following display, which has quite a bit more going on. Let's break it down.



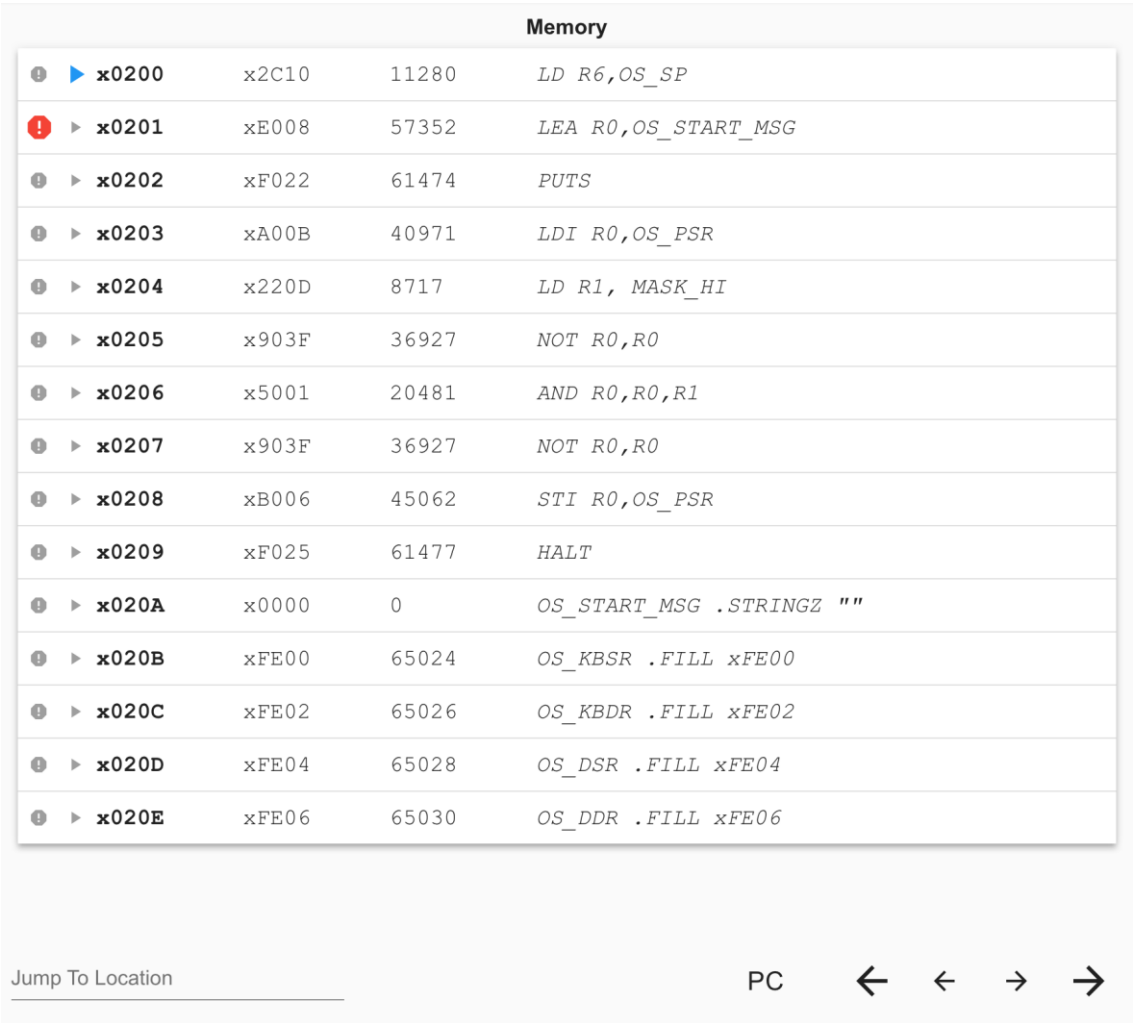
The panel on the top half of the left side shows the register values in hex and decimal. This panel also shows the Processor Status Register (PSR) and Condition Codes (CC), the Program Counter (PC), and the Machine Control Register (MCR).



The panel below the registers is the console. The console is used for I/O operations. Click the console box to begin sending in keyboard input. When selected, the console will have a glowing blue border around it. Program output will be displayed whether the console is in focus or not. The button in the top right will clear all the program output.

The right half of the screen is the memory panel. Each entry shows the following information from left to right: a stop sign indicating whether or not a breakpoint is set at that location (currently there is a breakpoint set at x0201), the location of the PC (currently at location x0200), the memory address, the hex and decimal values at the memory location, and the string from the binary or assembly file source code corresponding to the memory location.

The bottom of the memory panel allows you to navigate memory. The leftmost entry lets you jump to a specific spot in memory. The PC button will show the memory starting at the current PC. The arrow buttons allow you to scroll through memory. The large buttons jump by an entire snippet of memory at a time, and the small buttons jump by 5 locations at a time.



Finally, the toolbar on the left contains the following buttons from top to bottom:

1. Open an .obj file
2. Begin simulating
3. Reload object files
4. Step over an instruction
5. Step in an instruction
6. Step out of an instruction
7. Reinitialize the machine
8. Randomize the machine

More details on the specifics of these buttons can be found in chapter 4.

Chapter 2: Creating a program for the simulator

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.

Entering your program in machine language

You have the option to type your program into LC3Tools in either binary or the LC-3 assembly language. Here's what our little program looks like in binary:

```
0011000000000000
0101001001100000
0101100100100000
0001100100101010
1110010011111100
0110011010000000
0001010010100001
0001001001000011
0001100100111111
0000001111111011
1111000000100101
```

When you type this into LC3Tools, you'll probably be looking at a chart which tells you the format of each instruction, such as the one inside the back cover of the textbook. 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 returning the sum
0101 100 100 1 00000   ; clear R4, to be used as a counter
0001 100 100 1 01010   ; load R4 with #10, the number of times to add
1110 010 011111100     ; load the starting address of the data
0110 011 010 000000    ; load the next number to be added
0001 010 010 1 00001   ; increment the pointer
0001 001 001 0 00 011  ; add the next number to the running sum
0001 100 100 111111    ; decrement the counter
0000 001 111111011     ; do it again if the counter is not yet zero
1111 0000 00100101     ; halt
```

Either way is fine with LC3Tools. It ignores spaces anyway. The second way will just be easier for you to read.

Saving your program

Click on the Save  button in the toolbar on the left. 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*.

Creating the .obj file for your program

Before the simulator can run your program, you need to convert the program to a language that the LC-3 simulator can understand. The simulator doesn't understand the ASCII representations of binary that you just typed into LC3Tools. It only understands true binary, so you need to convert your program to actual binary, and save it in a file called *addnums.obj*.

When you press the Convert  button in the toolbar on the left, the *addnums.bin* file will try to be converted into a real binary file, *addnums.obj*. The console at the bottom of the window will report whether the conversion was a success or if there were any errors.

If you don't know the LC-3 assembly language yet, now you're ready to skip ahead to Chapter 3, 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-3 assembly language

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

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

Save this file as *addnums.asm* using the Save  button in the left panel. You still need to change your program to a real binary file, *addnums.obj*, which is now called "assembling" your program. To do this, press the Assemble  button in the left panel.

Chapter 3: Running a program in the simulator

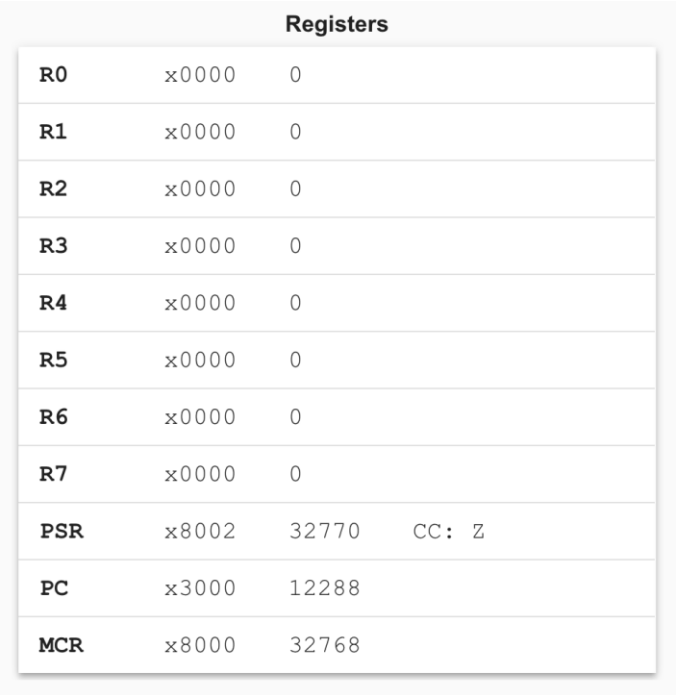
Now you're ready to run your program from the previous chapter in the simulator. Click on the simulator tab and then press the Open  button. Browse to and select *addnums.obj*. Assuming you originally wrote the assembly version of the program, this is what you will see in the memory panel.

Memory

x3000	x5260	21088	*AND R1, R1, x0*
x3001	x5920	22816	*AND R4, R4, x0*
x3002	x192A	6442	*ADD R4, R4, #10*
x3003	xE4FC	58620	*LEA R2, x0FC*
x3004	x6680	26240	*LOOP LDR R3, R2, x0*
x3005	x14A1	5281	*ADD R2, R2, x1*
x3006	x1243	4675	*ADD R1, R1, R3*
x3007	x193F	6463	*ADD R4, R4, x-1*
x3008	x03FB	1019	*BRp LOOP*
x3009	xF025	61477	*HALT*
x300A	x0000	0	
x300B	x0000	0	
x300C	x0000	0	
x300D	x0000	0	
x300E	x0000	0	

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, and the PC is pointing to the first line of your program (as is the blue arrow). Bit 15 of the PSR is 1, which means the machine is in user mode. Bit 15 of the MCR is set, which means the machine is powered on.



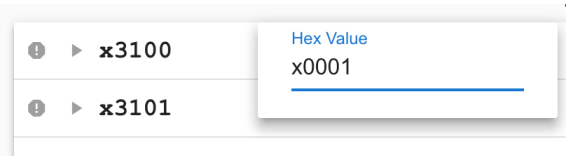
Loading the data (ten numbers) into memory

There are multiple ways to get the data into locations x3100 to x3109, and we show two of them below.

Method 1: Manually entering values

To add numbers to locations x3100 to x3109, type x3100 into the 'Jump to Location' text box and press enter. The memory panel will jump to location x3100.

Click on either the hex value or the decimal value and a text box will pop up. You may enter a hex value or decimal value, respectively, to set the memory location.



Continue doing so until locations x3100 through x310A have been set as such:

x3100	x0001	1
x3101	x0002	2
x3102	x0003	3
x3103	x0004	4
x3104	x0005	5
x3105	x0006	6
x3106	x0007	7
x3107	x0008	8
x3108	x0009	9
x3109	x000A	10

Method 2: Adding the data directly to the assembly file

Switch back to the editor tab and add the following lines after the .END in *addnums.asm*.

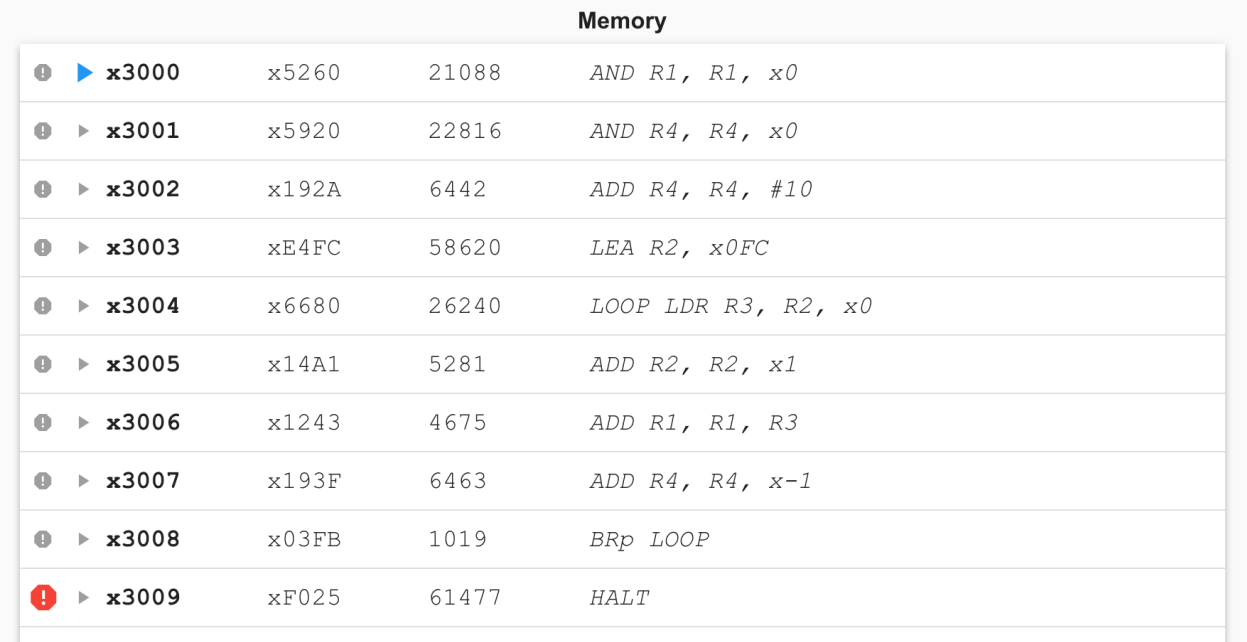
```
.ORIG x3100      ; start the data at location x3100
.FILL #1         ; the value of the first data element
.FILL #2
.FILL #3
.FILL #4
.FILL #5
.FILL #6
.FILL #7
.FILL #8
.FILL #9
.END
```

Assemble the program again using the Assemble  button. Switch back over to the simulator tab and press the Reload Object Files  button.

Running your program

Type x3000 into the 'Jump to Location' text box and press enter. Click the gray arrow to the left of location x3000 to set the PC to that location (it may already be set correctly there). Click the gray stop sign to the left of location x3009 to set a breakpoint at this location. Breakpoints will be discussed in further detail in Chapter 4, but basically simulation will pause as soon as the PC encounters a breakpoint. By setting a breakpoint at the HALT instruction, the simulator will pause before entering the HALT trap service routine. This is important because the HALT trap service routine changes R1, which means if it executes we'll never actually see the sum in R1!

The memory panel should look like this:



Press the Run  button. Simulation will pause as soon as the PC encounters location x3009.

If you've already added up the ten numbers you put into the data section of your program, you know that 55 is the answer to expect. That's what you should see in R1 when the program stops at the breakpoint (in hex, the result is x0037).

Stepping through your program

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

First, you need to reset the very important PC to the first location of your program. Set the PC back to x3000 by clicking the gray arrow to the left of location x3000. Alternatively, you can change the value of the PC by clicking either the hex or decimal value in the register panel. Now you're ready to step through your program.

Click the Step Over  button. A couple of interesting things just happened:

1. R1 got cleared (you won't notice a difference if it was already set to 0).
2. The blue arrow representing the PC moved to location x3001.

Click the Step Over  button again. Now R4 got cleared (you won't notice a difference if it was already set to 0) and the PC moved to location x3002.

Click the Step Over  button again. Now R4 got set to 10 and the PC moved to location x3003.

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  button, and that will cause your program to execute until it reaches the breakpoint you set on the HALT.

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. 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.

The following are a series of steps that are common while debugging any program, but we will work with a single example so it is easier to follow along.

Example 1: Debugging a program to multiply without a multiply instruction

The Problem Statement

Our goal is to multiply the values in R4 and R5 and store the result in R2.

Entering your program in machine language

Enter the following program into the editor.

```
0011 0010 0000 0000    ; start the program at x3200
0101 010 010 1 00000   ; clear R2
0001 010 010 0 00 100  ; add R4 to R2, put result in R2
0001 101 101 1 11111   ; subtract 1 from R5, put result in R5
0000 011 111111101     ; branch to location x3201 if 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.

Click on the Save  button and call the file *multiply.bin*. Press the Convert  button in the toolbar on the left to create *multiply.obj*.

Loading the program in the simulator

Switch to the simulator tab and click the Open  button. Browse to *multiply.obj* and open it.

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 clicking the gray stop sign to the left of the address.

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. Let's start with two reasonably small, but different numbers, like 5 and 3.

Click on the decimal value to the right of R4 in the register panel. Type in 5 in the text box that pops up and press enter. Do the same for R5, except enter in 3.

Ensure that the value of the PC is x3200. This can be confirmed by checking the PC register in the register panel or by seeing if there is a blue arrow to the left of location x3200 in the memory panel.

Run your program by pressing the Run  button. When the PC hits the breakpoint, R2 should contain the value 15. However, R2 contains 20! Something went wrong.

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.

To begin simulating again, you need to reset a couple of things. Click the gray arrow to the left of location x3200 in the memory panel to set the PC back to the beginning of the program. Click the decimal value to the right of R5 in the register panel and set the value to 3 again.

Click the Step Over  button. The PC changes to x3201 and R2 is set to 0. This is what you expected the first instruction to do, so you can move on to the next instruction.

Click the Step Over  button. Now PC contains x3202 and R2 contains the value 5. Again, this is what you want, so keep going.

The next time you click the Step Over  button, R5 changes from 3 to 2 (I'm not going to keep mentioning the PC, but you'll notice it 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. Each time through the loop, R5 gets decremented. That seemed to happen just fine, so keep going.

Clicking the Step Over  button once more cause 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 2, a positive number, so the P condition code was set to 1. Your branch is taken if the Z condition code or the P condition code contains a 1. The branch was executed taken, so the PC now points to x3201, and you're 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).

Set an additional breakpoint at the branch at location x3203 by clicking the gray stop sign to the left of the address. Now there should be breakpoints at both x3203 and the HALT at x3204. Simulation will pause if the PC encounters either breakpoint.

Run the program by clicking the Run  button. Notice the register values when you hit the breakpoint. R4 is unchanged, it still contains 5. R5 has changed to 2. The condition code is set to P. This tells you that the branch will be taken.

Click the Run  button again and look at the registers. R4 is unchanged, R5 contains 1, and R2 contains 10. The condition code is set to P, so the branch will be taken.

Click the Run  button again and look at the registers. R4 is unchanged, R5 is now 0, and R2 contains 15. The condition code is set to Z, so the branch will be taken. At this point the program should stop, but 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 the condition code is set to P, the loop will happen the correct number of times. To prove this to yourself, you can edit the program in the editor tab by changing the line with the branch to:

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

saving, converting to .obj format, and reloading the new version into the simulator by pressing the Reload Object Files  button. Alternatively, if you don't want to change the source code yet in case you find more bugs later, you can directly change the instruction in the simulator by clicking on the hex value next to address x3203 in the memory panel. Enter x0201 (it should be x0601 currently) in the text box that pops up and press enter. Remember that the 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 3. Click the red stop sign at location x3203 to disable the breakpoint. Click the Run button and you should see the value 15 in R2. Congratulations – you've successfully debugged the program!

Example 2: Debugging a program to inputs numbers and add them

This program is in assembly language, so if you haven't learned the LC-3 assembly language, you might want to wait until you do before you read this section.

The Problem Statement

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 panel.

Writing and assembling the buggy program

Our buggy version of the program looks like this:

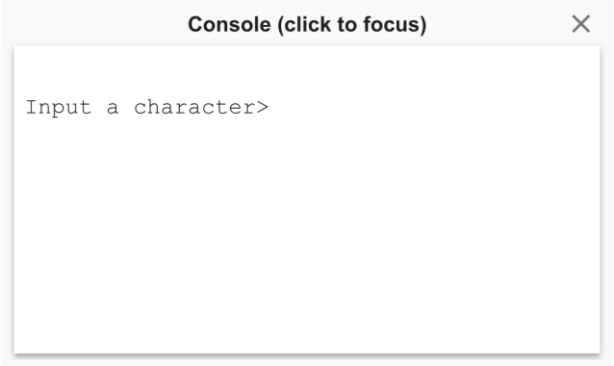
```
        .ORIG x3000
        TRAP x23                ; trap instruction known as "IN"
        ADD R1, R0, x0          ; move the first integer to R1
        TRAP x23                ; another "IN"
        ADD R2, R0, R1          ; add the two integers
        LEA R0, MESG            ; load the address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

Save this program as *addinput.asm* and assemble it.

Running the buggy program in the simulator

Switch to the simulator tab and load *addinput.obj*. 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, which is the ASCII code for 'h'. The whole string, "The sum is " is represented in the memory locations x3009 to x3014.

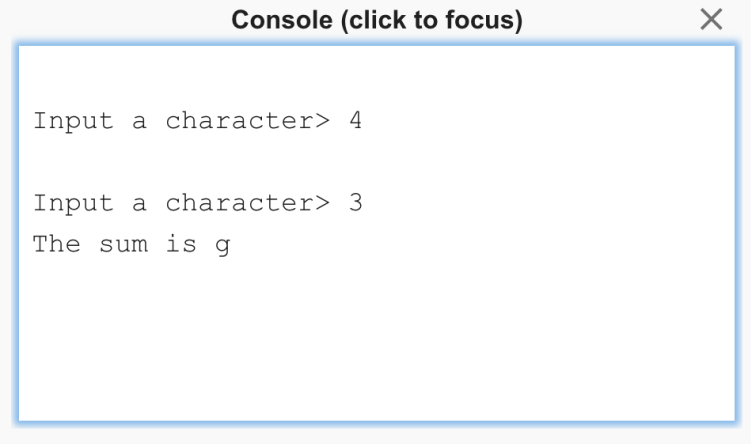
Click the gray stop sign to the left of location x3008 (the HALT) in the memory panel to set a breakpoint. Now click the Run button. The first instruction is a trap routine which prompts you to input a character into the console panel like this:



The simulator will just “wait” in the trap routine. You’ll need to click the console panel and then press an integer between 0 and 9. Try 4.

The console will show the 4 and then prompt for the next integer, meaning the simulator is in the TRAP x34 at location x3002. The value x34, which is the ASCII value representing 4, shows up in R1. TRAP x23 saves the ASCII value in R0, which is then copied to R1 by the instruction at x3001. R0 currently contains the value 0 because the program is already in the the “IN” trap routine invoked by TRAP x23 at location x3002, which must have cleared R0 at some point.

The simulator is still waiting on the second integer, so click the console again and enter 3. This is what the console looks like after the second integer is entered.



You know that 3 + 4 is not equal to g, so what went wrong?

Debugging the program

The big hint about why this program gives the wrong result is a couple paragraphs above this. Remember that the “4” that you typed in the console window actually ended up in R1 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?

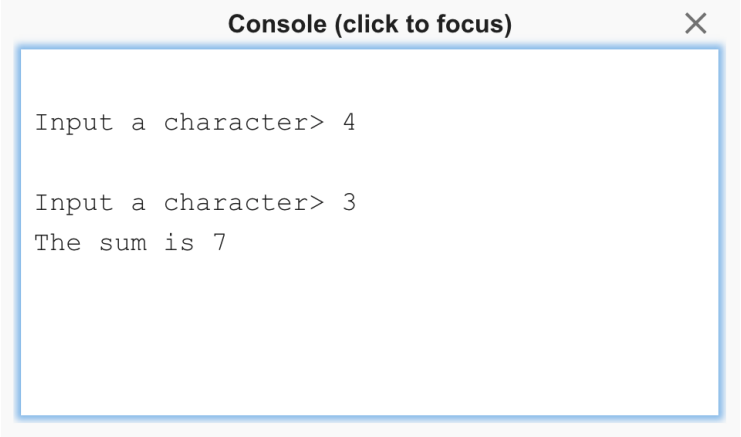
You're going to need the following pieces of data:

```
ASCII      .FILL x30     ; mask to add to convert to ASCII
NEGASCII   .FILL xFFD0   ; negated ASCII mask (x-30)
```

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                ; trap instruction known as "IN"
           ADD R1, R0, x0          ; move the first integer to R1
           ADD R1, R1, R5          ; convert ASCII number to integer
           TRAP x23                ; another "IN"
           ADD R0, R0, R5          ; convert ASCII number to integer
           ADD R2, R0, R1          ; add the two integers
           ADD R2, R2, R6          ; convert sum to ASCII
           LEA R0, MESG            ; load the address of the message
           TRAP x22                ; trap known as "PUTS"
           ADD R0, R2, x0          ; move the sum to R0, to be output
           TRAP x21                ; display the sum
           HALT
ASCII      .FILL x30               ; mask to add to convert to ASCII
NEGASCII   .FILL xFFD0             ; negated ASCII mask (-x30)
MESG       .STRINGZ "The sum is "
           .END
```

Save your program in the editor and reassemble. Reload the object files in the simulator. You should see the following in the console.



Other debugging tools

Here's a description of the debugging features provided by the simulator, some of which we've already seen.

Reload Object Files: reloads all files that have been opened and resets the PC to the first .ORIG in the file.

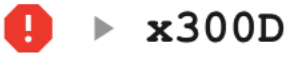
Step Over: for non-control instructions (ADD, LD, etc.), execute a single instruction. For control instructions (BR, JSR, TRAP, etc.), execute the entire routine and then pause at the next adjacent instruction in memory.

Step In: for non-control instructions (ADD, LD, etc.), execute a single instruction. For control instructions (BR, JSR, TRAP, etc.), execute the single instruction so the PC is at the beginning of the routine.

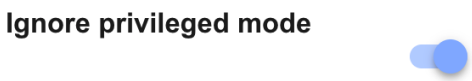
Step Out: execute instructions until an RTI or RET is reached.

Reinitialize machine: reset registers and memory locations to x0000. Reset PSR and MCR as well. Note that you will need to reload object files after reinitializing the machine.

Randomize machine: randomize registers and memory locations. Useful for verifying that programs are initializing registers and memory properly instead of relying on them being initialized to x0000. Note that you will need to reload object files after randomizing the machine.



The red stop sign indicates a breakpoint is set at location x300D. When the PC encounters x300D during execution, simulation will pause.



Found in the settings panel in the top toolbar. The PSR is initialized to user mode (bit 15 is a 1), so typically programs you write will not be able to access protected memory. However, you may want to access protected regions, such as memory-mapped I/O and the vector table. Make sure this option is selected so that you do not run into access violation exceptions.