

COSC 221: Computer Organization I Winter 2012

LC-3 Tutorial

(Materials obtained from “Guide to Using the Windows version of the LC-3 Simulator and LC3Edit” by Kathy Buchheit, 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.

I Creating a program for the simulator

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

A. 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.

B. Using LC3Edit

If you're using Windows, there's another program in the same folder as the simulator, called LC3Edit.exe. Start that program by double-clicking on its icon, and you'll see a simple text editor with a few special additions.

C. Entering your program in machine language

You have the option to type your program into LC3Edit in one of three ways: binary, hex, or the LC-3 assembly language. Here's what our little program looks like in binary:

```
0011000000000000
0101001001100000
0101100100100000
0001100100101010
1110010011111100
0110011010000000
0001010010100001
0001001001000011
0001100100111111
000000111111011
111100000100101
```

When you type this into LC3Edit, you'll probably be looking at a chart which tells you the format of each instruction, such as the one inside the back cover of the textbook. So it may be easier for you to read your own code if you leave spaces between the different sections of each instruction. Also, you may put a semicolon followed by a comment after any line of code, which will make it simpler for you to remember what you were trying to do. In that case your binary would look like this:

```
0011 0000 0000 0000 ;start the program at location x3000
0101 001 001 1 00000 ;clear R1, to be used for the running sum
0101 100 100 1 00000 ;clear R4, to be used as a counter
0001 100 100 1 01010 ;load R4 with #10, the number of times to add
1110 010 011111100 ;load the starting address of the data
0110 011 010 000000 ;load the next number to be added
0001 010 010 1 00001 ;increment the pointer
0001 001 001 0 00 011 ;add the next number to the running sum
0001 100 100 1 11111 ;decrement the counter
0000 001 11111011 ;do it again if the counter is not yet zero
1111 0000 00100101 ;halt
```

D. 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.

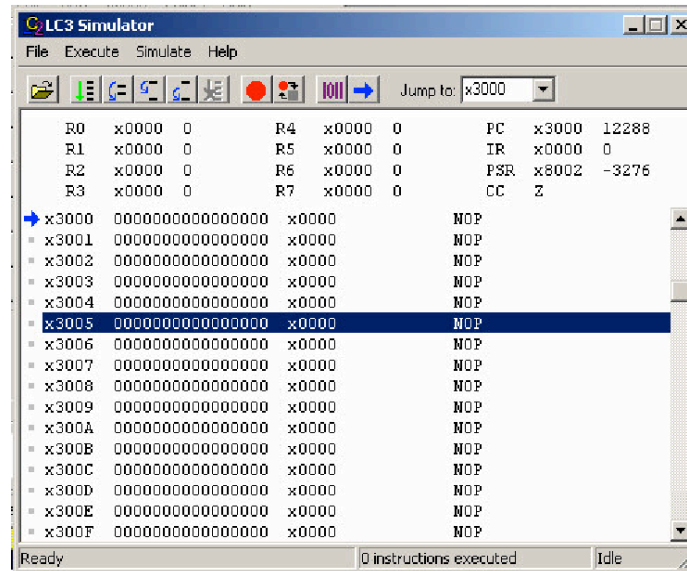
E. Creating the .obj file for your program

Before the simulator can run your program, you need to convert the program to a language that the LC-3 simulator can understand. The simulator doesn't understand the ASCII representations of hex or binary that you just typed into LC3Edit. 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 LC3Edit, 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*.

II The simulator: what you see on the screen



A. 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	PSR	x8002	-3276
R3	x0000	0	R7	x0000	0	CC	Z	

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-3 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-3 control unit. Those registers are the PC, the IR, and the N, Z, and P condition code registers.

PC x3000 12288
IR x0000 0
PSR x8002 -3276
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 PSR, or processor status register, contains the state of the processor, which is either user mode or privileged mode and the value of the condition codes. 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.)

B. The memory

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

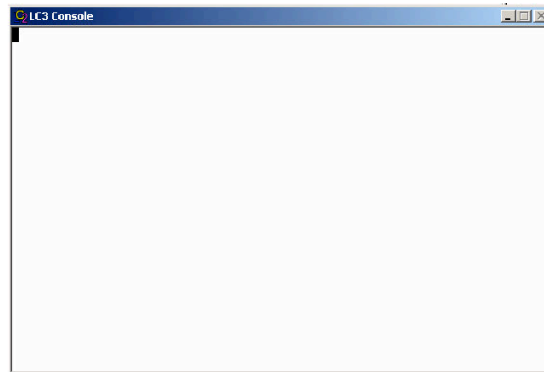
x3000	0000000000000000	x0000	NOP
x3001	0000000000000000	x0000	NOP
x3002	0000000000000000	x0000	NOP
x3003	0000000000000000	x0000	NOP
x3004	0000000000000000	x0000	NOP
x3005	0000000000000000	x0000	NOP
x3006	0000000000000000	x0000	NOP

Use the scrollbar at the right to scroll up and down through the memory of the LC-3. Remember that the LC-3 has an address space of 216, 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.

Jump to: x3000


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.

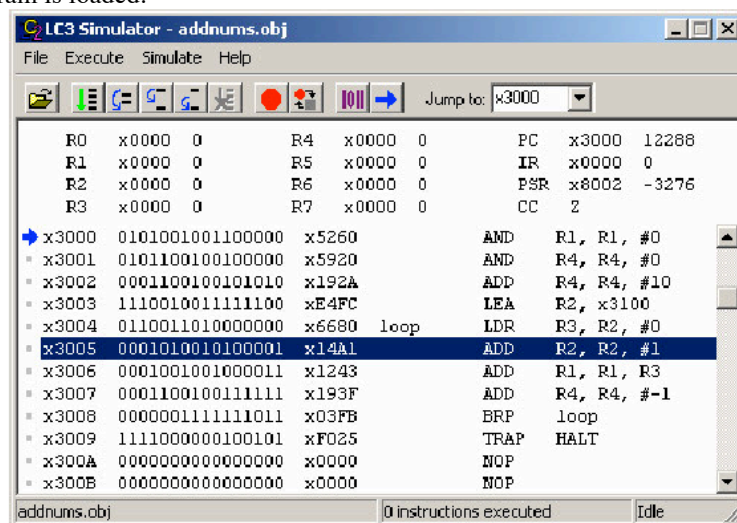
C. The Console Window



A second window also appears when you run the simulator. It is rather inconspicuous, and has the vague title "LC3 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.

III Running a program in the simulator

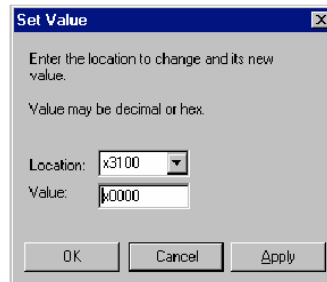
Now you're ready to run your program in the simulator. Open the simulator, and then click the Load Program button . Browse and choose *addnums.obj*. Notice that you only get the option to open one type of file in the simulator: the .obj file. This is what you'll see when your program is loaded:




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.

A. Loading the data (ten numbers) into memory

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



The "Set Value" dialog box has a title bar with a close button. Inside, it says "Enter the location to change and its new value." and "Value may be decimal or hex:". There are two input fields: "Location:" with a dropdown menu showing "x3100" and "Value:" with a text box containing "x0000". At the bottom are three buttons: "OK", "Cancel", and "Apply".

In the "Value" box, type the hex number x3107, and choose OK. (You can also get this window by clicking on the , Set Value, button.) 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-3 simulator doesn't know that. In fact, the contents of all memory locations are equal in the eyes of the LC-3, 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 LC3Edit 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-3 simulator's memory. The memory locations starting with x3100 will look like this:

▪ x3100	0011000100000111	x3107	ST	R0, x3008
▪ x3101	0010100000011001	x2819	LD	R4, x311B
▪ x3102	0000000100010000	x0110	NOP	
▪ x3103	00000001100010000	x0310	BRP	x3014
▪ x3104	0000000100010000	x0110	NOP	
▪ x3105	0001000100010000	x1110	ADD	R0, R4, R0
▪ x3106	0001000110110001	x11B1	ADD	R0, R6, #-15
▪ x3107	00000000000011001	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.

B. 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.




C. Stepping through your program


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

1. 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.

2. 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.
3. Click , Step Over, for a second time. Again, notice the new values for the PC and IR. The second instruction clears R4.
4. 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.
5. 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.

LC3Edit reference

(Materials obtained from Chapter 5 of the “Guide to Using the Windows version of the LC-3 Simulator and LC3Edit” by Kathy Buchheit, The University of Texas at Austin)

Here is what you see when you open LC3Edit:

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-3 assembly language. 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:

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

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 LC3Edit, 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 LC3Edit, 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-3 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 LC3Edit.

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 LC3Edit...” gives you some information about the author and copyright of LC3Edit.

LC-3 Simulator reference, Windows version

(Materials obtained from Chapter 6 of the “Guide to Using the Windows version of the LC-3 Simulator and LC3Edit” by Kathy Buchheit, The University of Texas at Austin)

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-3, along with their contents. R0 through R7 are the eight registers where LC-3 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 four columns show the names and contents of the PC, the IR, the PSR, 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-3 during the six phases that make up one instruction cycle, you would notice that the value of the IR would be the same during all six phases. This simulator doesn’t let us “see inside” an instruction in this way. So when you are stepping through your program one instruction at a time, the IR will actually contain the instruction that just finished executing. It is still considered “current” until the next instruction begins, and the first phase of its execution – the fetch phase – brings a new value into the IR.

The PSR, or processor status register, contains information regarding the current state of the processor. In particular, which mode the processor is in, user or privileged as well as the current condition code set.

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-3. Remember that the LC-3 has an address space of 216, 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.)

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-3. When you load a program, these translations will be extremely helpful because you’ll be able to quickly look through your program and know what is happening. Sometimes, however, these assembly language interpretations make no sense. Remember that computers, and likewise the simulator, are stupid. They don’t understand your intentions.

So the data section of your program will always be interpreted into some sort of assembly language in this last column. An important aspect of the Von Neumann model is that both instructions and data are stored in the computer’s memory. The only way we can tell them apart is by how we use them. If the program counter loads the data at a particular location, that data will be interpreted as an instruction. If not, it won’t. So ignore the last column of information when it tries to give some nonsense interpretation to the data in your program. You may notice, if you browse around in memory sometime, that not every memory location is set to all 0’s even though you didn’t put anything there. Certain sections of memory are reserved for instructions and data that the operating system needs. For instance, locations x20 through xFF are reserved for addresses of trap routines. Here’s a small piece of that section:

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

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 (216 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 whoknowswhere.

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 “LC3 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 LC3Edit (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, PSR, 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.

LC-3 Assembler Quick Reference

(Materials obtained from Chapter 7 of the “Guide to Using the Windows version of the LC-3 Simulator and LC3Edit” by Kathy Buchheit, The University of Texas at Austin)

Labels

Labels are case sensitive and should begin with a letter or underscore followed by no more than 9 alphanumeric characters. They may not be a reserved word, such as an opcode or pseudo-op. Labels may or may not be terminated by a colon. If terminated by a colon, the colon should not be included in later references to the label. There may only be one label per memory location.

Instruction syntax

Instructions must consist of the opcode and operands. The operands may or may not be separated by commas. Although the assembler is case-sensitive to labels, this is not the case with regard to instructions.

Pseudo-ops

There may only be one .orig and one .end statement per assembly file. Labels may be used in conjunction with .FILLs to fill a memory location with an address of a label. Pseudo-ops may appear anywhere in the assembly file between the .orig and .end statements.

Constant Formats

The assembler accepts constants preceded by either a # for decimal or x for hexadecimal. For hexadecimal constants they should not be preceded by a “-“. It is acceptable to do so with a decimal constant.