Table of Contents

Overview

complx-tools is a suite of tools for learning lc3 assembly. It includes both a gui and cli based simulator (named complx and comp respectively), an assembler (as2obj), a very simple program that runs lc3 assembly files and spits out whats printed to the console (lc3runner), and a framework for testing lc3 assembly code (lc3test). Complx-tools also be extended with plugins that add additional functionality to the LC3. The tools also come with a C++ interface to the LC3 (liblc3).

These tools are mainly used in CS2110 at Georgia Tech.

Note this document does not explain the lc3 isa, nor does it explain what the instructions do. This document only explains the set of tools at your disposal for running/testing your programs. If you want a document that explains the LC-3 ISA then please refer to Appendix A of the text or the file PattPatelAppA.pdf

Setup

Manual steps to compile the program are as follows

From the root directory of where you have the source code.

1. Ensure you have a C++ compiler (sudo apt-get install build-essential)
2. Ensure you have CMake installed (sudo apt-get install cmake)
3. Install dependency wxWidgets 3.0 (sudo apt-get install libwxgtk3.0-dev)
4. Create a build directory (mkdir build && cd build)
5. Generate a Makefile via cmake (cmake ..)
6. Build the program (make)
7. Install the program (sudo make install)
8. Run ldconfig (sudo ldconfig)

If you'd rather a script to setup the program for you see the following files

To use this make sure you are in the root directory of the source code…

To run the script run the command (sudo *./filename.sh*) replacing filename.sh with the appropriate one below based on your system

install.sh - Performs the above operations

install_script_fedora_21.sh – Install script specifically for Fedora, thanks Thomas Coe

install_script_ubuntu_13_10.sh - Install script for Ubuntu 13.10 and below, thanks Abhijit Murthy

TODO Talk about Arch Linux here... There is a PKGBUILD file in a pull request in the github repo.

Compiling on Windows and Mac is not supported and untested.

General Usage

Assembly Files

An assembly file (with the .asm extension) is just a normal text document. You can create .asm files in any text editor of your choosing. I recommend gedit which should be preinstalled on your linux machine, other text editors are emacs, vim, and nano if you are into those. As a sample assembly program to get you started copy and paste this into a file named helloworld.asm (Be careful to replace the “”'s marks in the program below as it may have changed into another character).

```
.orig x3000
     LEA R0, HELLOWORLD
     PUTS
     HALT
     ; Hello this is a comment
     HELLOWORLD .stringz “Hello World”
.end
```

The next sections will explain the tools you have for running/testing your program

As2obj

This program just assembles files and produces an object file (.obj) and a symbol table file (.sym). I will explain the formats of these two files later. Invoking the assembler is easy (note the []'s indicate optional parameters)

as2obj *filename.asm* [-all_errors] [-disable_plugins] [outfile]

An explanation of the command line parameters:

- -all_errors Report all errors encountered by the assembler if possible
- -disable_plugins Disable use of all lc3 plugins
- outfile output filename

Note if a parameter is enclosed in []'s it means it is optional. When specifying it you do not include the []'s

Lc3runner

This program is a simple command line program that just runs your assembly program until it halts spitting out any output that is written to the console. Running this program is very simple, but it doesn't have many options you can't view the contents of any address unless your program prints it out.

lc3runner filename.asm [-zeroed] [-input=some_text_file] [-count=num]

an explanation of the parameters

- -zeroed if passed in will zero out all memory if not passed in then all memory is randomized.
- -input=some_text_file use this file as stdin any reads from keyboard will read from the file instead
- -count=num from forr num instructions the default is run until HALT is encountered

Lc3test

This program is acommand line program that tests lc3 assembly code. Unit tests are specified in xml explained in the next section. The output of this program is a nice test report detailing what tests passed or failed. This program can also used for auto-grading lc3 assembly code.

lc3test *testfile.xml asmfile.asm* [-random_seed=X]

an explanation of the parameters

- random_seed – Random seed that all test cases will use.

Test Files by Example

An example of the usage of this program. Say you have the following program specification.

Write a program that adds two things.

The input will be located at symbols U and V

You are to store your answer (U + V) at symbol ANSWER

The following assembly code implements the above specification

```
.orig x3000
; Implementation of answer = u + v
LD R0, U
LD R1, V
ADD R2, R0, R1
ST R2, ANSWER
; end program.
HALT
; PARAMS to code are here
U .fill 2000
V .fill 8
; ANSWER is stored here
ANSWER .blkw 1
.end
```

Then this following xml file provides one unit test checking if the params are 2+3 that 5 is stored to ANSWER.

```
<?xml version="1.0"?>

<test-suite>

        <test-case>

                <name>2 + 3 = 5</name>

                <has-max-executions>1</has-max-executions>

                <max-executions>1000000</max-executions>

                <randomize>1</randomize>

                <input>

                        <test-value><address>U</address><value>2</value></test-value>

                        <test-value><address>V</address><value>3</value></test-value>

                </input>

                <output>

                        <test-value><address>ANSWER</address><value>5</value></test-value>

                </output>

        </test-case>

</test-suite>
```

You can specify more test cases by simply including more <test-case> tags in the file.

An explanation of each tag used here

- <name> specifies the name of the test case for the report
- <has-max-executions>/<max-executions> Specifies a limit on the maximum number of instructions executed in case the code infinitely loops.
- <randomize> Randomizes LC3 memory before running the test.
- <input> Specifies preconditions that will be met before the code is ran. Such as setting a value at a label, or specifying console input
- <test-value> Here we are simply setting the values at labels U and V to 2 and 3 respectively.
- <output> Specifies postconditions that will be checked after the code is ran.
- <test-value> Here we are just checking if the ANSWER label is 5.

Specifying Preconditions

In the example above we simply loaded the values from the locations U, V and stored the result at another location ANSWER. These locations simply contained a value that we access and mutate directly.

test-value

The format of <test-value> is as follows:

<test-value><address>*ADDR*</address><value>*EXPR*</value></test- value>

where:

ADDR is an expression for calculating the address (can be a symbol, an addess specified in hex ex xABCD)

EXPR is an expressing calculating the value to store at that addresses

When the test case is ran, before executing the code MEM[ADDR] will be set to EXPR.

So now I explain how to specify preconditions for locations with other types of data.

test-pointer

This is used when the value at the address does not contain a value, but another address. As a hint for if you need to use this is if you are using LDI with a symbol (or a combination with LD with LDR with offset 0).

The format of <test-pointer> is similar to <test-value>:

<test-pointer><address>*ADDR*</address><value>*EXPR*</value></test- pointer>

When the test case is ran, before executing the code MEM[MEM[ADDR]] will be set to EXPR.

test-register

If your input is specified via a register rather than a memory address then test-register should be used instead. By default all of the register's values are randomized here you can specify a value to put into a register.

The format of <test-register> is as follows:

<test-register><register>R0-R7</register><value>VALUE</value></test- register>

where VALUE is an expression.

When the test case is ran REG = VALUE

test-pc

To move the PC from the default x3000 address test-pc should be used

The format for <test-pc> is as follows:

<test-pc><value>VALUE</value></test-pc>

where VALUE is an expression.

test-array

This is used to set an array The address of where the array is will be pointed to by an address. That is the arrays location will be contained at an memory address.

The format for <test-array> is as follows:

<test-array><address>ADDR</address><value>EXPR0,EXPR1,...,EXPRN</value></test-array>

Note that at least one item is required to use test-array, otherwise it will be rejected since it doesn't actually set anything, or check anything.

ADDR is an expression such that MEM[MEM[ADDR]] contains the first value in the array, that is, the value contained in MEM[ADDR] must point to where the array is.

When the test case is ran the following addresses are set

MEM[MEM[ADDR]] = EXPR0

MEM[MEM[ADDR] + 1] = EXPR1

MEM[MEM[ADDR] + 2] = EXPR2

…

MEM[MEM[ADDR] + N] = EXPRN

test-string

This is used to set a string its use is similar to that of test-array except there is an additional value for the nul terminator.

The format of test-string is as follows:

<test-string><address>ADDR</address><value>HELLOWORLD</value></test-string>

ADDR is an expression such that MEM[MEM[ADDR]] contains the first character of the string, that is, the value contained in MEM[ADDR] must point to where the string is.

HELLO_WORLD is a string without the “”'s

When the test case is ran the following is done (for the string HELLOWORLD)

MEM[MEM[ADDR]] = H

MEM[MEM[ADDR] + 1] = E

...

MEM[MEM[ADDR] + 10] = D

MEM[MEM[ADDR] + 11] = 0

test-stdin

And to specify console input for a specific test test-stdin should be used

The format for test-stdin is as follows:

<test-stdin><value>STRING</value></test-stdin>

where STRING is any string without the quotes.

When the test is ran STRING is used for the input.

Specifying Postconditions

The same XML tags and syntax are used for specifying postconditions. The only difference is that test-stdin is now test-stdout

You can also change the way the test runner checks your output, by default the test runner will compare the value computed by the test against the expected value using == to change this behavior from this default you can specify a condition in the <test-XXX> element.

Example:

<test-value condition=”!=”><address>ANSWER</address><value>0</value></test-value>

And this will check if MEM[ANSWER] is not equal to 0.

For a complete enumeration for what you can specify for conditions

For test-value, test-pointer, test-register, and test-pc:

- equals, ==, =
- notEquals, !=
- less, <
- greater, >
- lessOrEquals, <=
- greaterOrEquals, >=

For test-string and test-stdout

- equals, ==, =
- notEquals, !=
- equalsIgnoreCase
- notEqualsIgnoreCase
- contains, c
- notContains, !c
- containsIgnoreCase
- notContainsIgnoreCase

For test-array:

- equals
- notEquals

Caveats

test-array

For the array comparisons the length of the array tested from the code will be equal to the length of the array given in the test /

That is, if the actual array the code generated is [2, 3, 4, 6, 3, 4, 2, 3] and the expected array given in the test is [2, 3, 4] then it will say the test passed.

If you really need to check if two arrays are identical then you will need to also output the size of it. Or alternatively write a value at the past the end of the array and see if it was changed.

test-string

A nul terminator must be written at the end of the string so that the test runner knows where the string ends.

If this doesn't happen the test-runner will terminate the string as soon as a value outside the range (0, 255] is found.

LC3 Calling Convention Checker

The calling convention checker is just another thing you can use in input/output tags. However, it is a little involved and performs a lot of operations and checks a lot of things for output.

The syntax (within test-input)

<test-subr><name>*subroutine*</name><stack>*stack location*</stack><r7>*some value*</r7><r5>*some other value*</r5><params>*expressions*</params></test-subr>

Where:

subroutine is the name of the subroutine under test

stack location is the starting location of the stack

some (other) value is a dummy value to be placed in r7, r5 as a canary

expressions is a comma separated list of expressions

What happens when this is tested? The following happens:

```
PC = SUBROUTINE
R6 = STACK - num_params
MEM[R6] = EXPR0
MEM[R6 + 1] = EXPR1
… for each param
R7 = R7_VALUE
R5 = R5_VALUE
MEM[R7] = xF025
```

To explain execution starts off directly at the subroutine. This is done to simulate a call to the subroutine, once ret is executed it will return to whatever is in r7. R7 gets a dummy address and at that dummy address there is xF025 which is HALT. This is done to ensure the machine halts after the subroutine is executed.

Next the stack is loaded with the parameters and is set to point at the first parameter.

A dummy value is placed in r5 and will be checked after the halt is reached.

This is all done to ensure your subroutine works if it was called from anywhere.

Syntax (within output)

```
<test-subr>
        <answer>answer</answer>
        <locals>locals</locals>
        <points>
                <answer>1</answer>
                <params>1</params>
                <r7>1</r7>
                <r6>1</r6>
                <r5>1</r5>
                <locals>1</locals>
        </points>
        <deductions-per-mistake>1</deductions-per-mistake>
</test-subr>
```

Where

answer is the answer for the subroutine call

locals is the values of the local variables in declaration order.

What does the test runner do to check the output?

1. The test runner grabs the stack frame of your function and compares it with what it should be
2. The answer is searched for in the stack.
3. R6's value is checked. It should be one less than what it was when the test started.
4. R7's value is searched for in the stack, R7 is checked if it is not clobbered
5. R5's value is searched for in the stack, R5 is checked if it is not clobbered
6. Parameters is searched for in the stack
7. Local variables are each searched for in the stack.
8. Then the structure of the stacks is checked. Your stack is tested against the stack generated by the test. The number of structural mistakes is noted. Essentially here it is checking the number of edits (additions, deletions, and modifications) necessary to transform your stack to the expected stack.

Output

```
1 (P 8 / 8) Answer 1 calling convention followed
 expected (right): 0x1 0x2 0xcafe 0x5000 0x1 0x1 0x2  r5: 0xcafe r6: 0xeffd r7: 0x5001
    actual  (left): 0x1 0x2 0xcafe 0x5000 0x1 0x1 0x2  r5: 0xcafe r6: 0xeffd r7: 0x5001
     [✓] answer found on stack +1
     [✓] r6 points to r6-1 (answer location) +1
     [✓] r7 found on stack and r7 not clobbered +1
     [✓] r5 found on stack and r5 not clobbered +1
     [✓] 1 param found on stack +1
     [✓] 2 param found on stack +1
     [✓] 1 local found on stack +1
     [✓] 2 local found on stack +1
     Found no structural mistakes in the stack.  No changes needed.
```

So to interpret this output. You read the stack from right to left.

The first num_params items are the parameters (as soon as the first underlined items from the right).

The next value from the right is the return value 0x1 as soon as the second item underlined.

Then comes the return address and old frame pointer.

Then afterwards come the local variables.

If any registers was saved then the test runner will effectively perform some operations to ignore them. However the local variables *MUST* appear after the old frame pointer, if not then it will be marked as a structural error as they were stored in the wrong location.

If any expected value was not found on the stack, then it will not also be counted as a structural mistake.

Other Features

These features can be used within the <test-case> tag.

True Traps Setting

<true-traps>0 or 1</true-traps>

Enable true-traps the default for this is 0 for false.

Setting maximum executions

    <has-max-executions>0 or 1</has-max-executions>

    <max-executions>NUM</max-executions>

By default the test case runner will run your code infinitely, this should always be used in case the code has an infinite loop.

Randomization

```
<randomize>0 or 1</randomize>
```

By default memory is not randomized. If set to 1 then all addresses will be randomized before the code is loaded.

Interrupts Settting

```
<interupt-enabled>0 or 1</interupt-enabled>
```

By default interrupts are not enabled. If set to 1 then interrupts will be generated.

Disable Plugins Setting

- `<disable-plugins>0 or 1</disable-plugins>`

Disable the use of lc3 plugins for this test. Default is false.

Comp

This section is a work in progress as this program is not finalized.

comp is a text based command line front-end for complx-tools, contributed by a student in 2013. If you wish to test it out you can uncomment its targets from the Makefile in the top level directory of complx-tools.

Complx

This program is the main program you will probably want to run. It is an lc-3 simulator, that allows you to play through instructions, undo instructions, and modify the state of the lc3 while a program is running. Also supports a plethora of debugging tools. To start the program you can either find it in the “Start Menu” (Should be under programming), or start it through the terminal,

```
complx filename.asm
```

Command line parameters

The command line parameters as shown below are all optional.

- --unsigned=bool Sets if decimal representations are displayed in unsigned (default 0)
- --disassemble=num Sets the disassemble level 0: basic 1: normal 2: high level (default 1)
- --stack_size=num Sets the Undo stack size (default 65536 instructions)
- --call_stack_size=num Sets the Call stack size (default 10000 subroutine/trap calls)
- --address=hex Sets the PC's starting address (default 0x3000)
- --true_traps=num Enable true traps (see True Traps Mode) (default 0)
- --interrupts_enable=num Enable interrupts (default 0)
- --highlight=num Enable instruction highlighting (default 1)

Getting Started

As stated above you can start complx via the command line to load a file immediately by just passing in the path to the assembly file you want to load.

To load a file via the GUI you can use one of the menu options under the File menu

- Randomize and Load – Ctrl + Alt + O
- Randomize and Reload – Ctrl + Alt + R
- Load – Ctrl + O
- Reload – Ctrl + R
- Load Over – Ctrl + Shift + O
- Reload Over – Ctrl + Shift + R

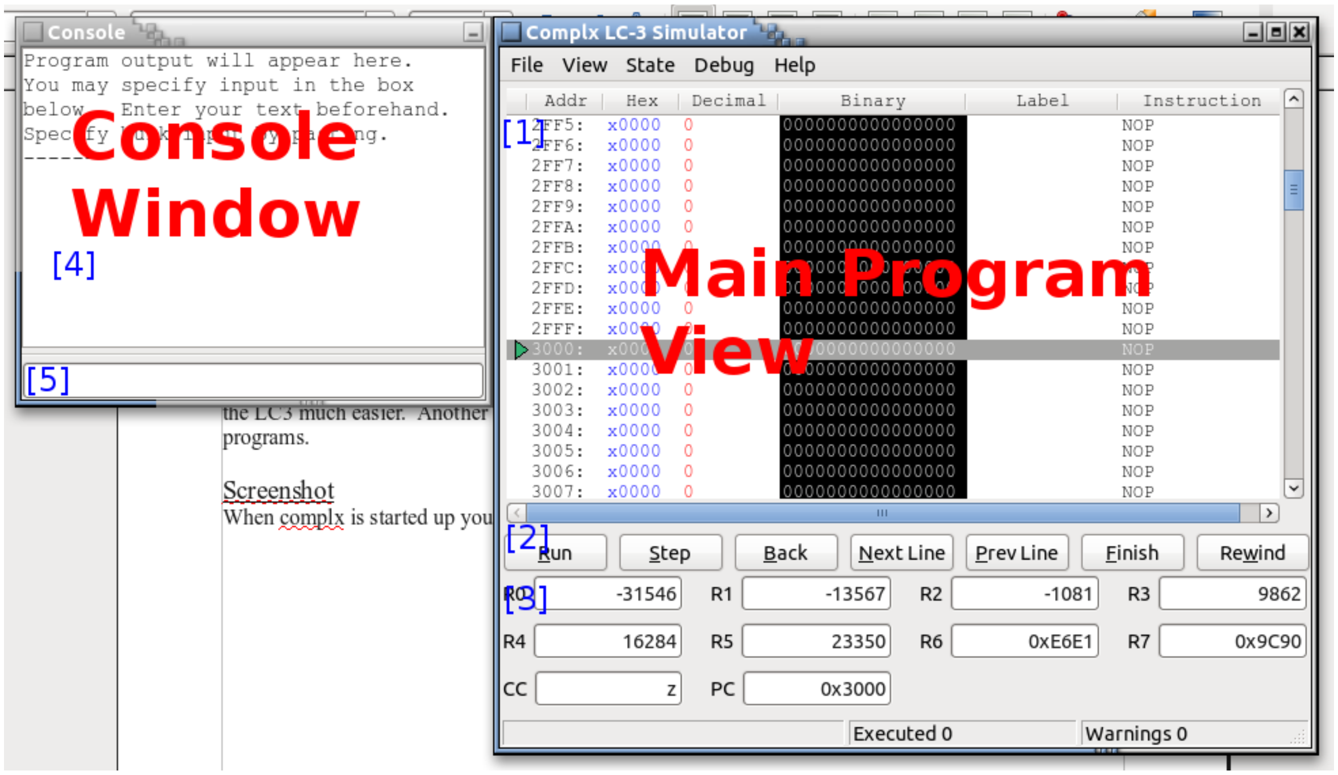
Randomize and Load does the following randomizes every address, then loads your assembly file over the randomized memory, that is, any addresses modified by your program will overwrite memory.

Load does the following cleans out the memory bring the lc3 back to the initial state, then loads your assembly file.

Load Over just loads your assembly file over the current state of the memory

The reload versions do exactly the same as above except it does not query you for a file. It will use the last file loaded (or if there isn't one it will query you for one.

Interface tour



```
Program output will appear here.
You may specify input in the box
below.  Enter your text beforehand.
Spec[illegible]ng.
-----
```

Addr	Hex	Decimal	Binary	Label	Instruction
2FF5:	x0000	0	0000000000000000		NOP
2FF6:	x0000	0	0000000000000000		NOP
2FF7:	x0000	0	0000000000000000		NOP
2FF8:	x0000	0	0000000000000000		NOP
2FF9:	x0000	0	0000000000000000		NOP
2FFA:	x0000	0	0000000000000000		NOP
2FFB:	x0000	0	0000000000000000		NOP
2FFC:	x0000	0	0000000000000000		NOP
2FFD:	x0000	0	0000000000000000		NOP
2FFE:	x0000	0	0000000000000000		NOP
2FFF:	x0000	0	0000000000000000		NOP
3000:	x0000	0	0000000000000000		NOP
3001:	x0000	0	0000000000000000		NOP
3002:	x0000	0	0000000000000000		NOP
3003:	x0000	0	0000000000000000		NOP
3004:	x0000	0	0000000000000000		NOP
3005:	x0000	0	0000000000000000		NOP
3006:	x0000	0	0000000000000000		NOP
3007:	x0000	0	0000000000000000		NOP

R0	-31546	R1	-13567	R2	-1081	R3	9862
R4	16284	R5	23350	R6	0xE6E1	R7	0x9C90
CC	z	PC	0x3000				

Executed 0   Warnings 0

the LC3 much easier. Another
programs.

Screenshot
When complx is started up you

As a brief tour of the interface

[1] Is the main memory view. This shows the value each address contains in the LC-3 interpreted in hexadecimal, decimal, binary, and as an LC-3 Instruction. Remember that all instructions can be represented as a 16 bit value.

So if I had the instruction ADD R0, R0, R0 then I should see the following:

- Hex column I should see x1000
- Decimal column I should see 4096
- Binary column I should see 0001000000000000
- Instruction column I should see ADD R0, R0, R0 (if the disassemble level is set to normal).

[2] The main control buttons. You will use these buttons to run your program (Run) or step through your program one instruction at a time (Step), you may also undo instructions using Back or rewind the entire program undoing everything in the undo stack (Rewind).

[3] The current state of all registers. You can enter values in binary, hexadecimal, or decimal. You can also change the base the registers contents is displayed in by either double clicking the text box or right clicking and selecting a new base for the register. By default R5-R7 are displayed in hexadecimal since

these registers usually contain an address.

[4] Console window output will be displayed here. Also any warnings generated from your program will also be spit out here.

[5] Console window input will be typed in here. Its best to type your text before the program is ran.

Running Programs

The control buttons from the previous section should be used to run your assembly program. I will explain what each button does in this section

Step – F2 Executes exactly one instruction

Back – Shift + F2 Undoes exactly one instruction

Next Line – F3 Performs one instruction, if used on a subroutine or trap it will execute the entirety of it, that is, if used on a statement the PC shall point to the next line in the program.

Prev Line – Shift + F3 Undoes one instruction, but if backed into a subroutine or trap will undo the entirety of it, that is, if used on a statement the PC shall point to the previous line in the program

Run – F4 Runs your program until it HALTs or you stop it manually.

Run For... – Ctrl + F4 Runs for X instructions (will always query you for X)

Rewind – Shift + F4 Repeatedly undoes instructions until the undo stack is exhausted

Finish – Shift + F5 Executes enough instructions to step out of the current subroutine or trap you are in

Run Again – (only available as a menu option) Ctrl + Space Same as Run For but will not query you for X after the first time it is used

Example

This program does nothing interesting, but is designed to show where the PC will end up if you use each of these buttons. The end result is that 2 should be loaded into R0 by the time HALT is executed

```
.orig x3000
JSR FAKE_SUBR                 ;x3000 [1]
HALT                          ;x3001 [2]
FAKE_SUBR ST R7, SAVE         ;x3002 [3]
AND R0, R0, 0                 ;x3003
JSR ADD2                      ;x3004
LD R7, SAVE                   ;x3005 [4]
RET                           ;x3006
ADD2 ADD R0, R0, 2            ;x3007 [5]
RET                           ;x3008
SAVE .blkw 1                  ;x3009
.end
```

Note the flow of this program is the following instructions at these addresses get executed

x3000, x3002, x3003, x3004, x3007, x3008, x3005, x3006, x3001

control \ point	[1]	[2]	[3]	[4]	[5]
step	x3002	x3001	x3003	x3006	x3008
back	x3000	x3008	x3000	x3004	x3004
next line	x3001	x3001	x3003	x3006	x3008
prev line	x3000	x3000	x3000	x3004	x3004
run	x3001	x3001	x3001	x3001	x3001
finish	Error	Error	x3001	x3001	x3005

To read this table look at the instruction with the comment [#] in the header of the column the address in the cell is the result of using that operation. From the flow of the program above look where the PC ended up relative to the starting address.

For example see using Finish at point [5] in the program.

Point [5] is address x3007 and using Finish at this point will go to x3005

Therefore the instructions at x3007 and x3008 get executed to perform the Finish operation.

Debugging

So what tools does complx provide for people struggling to get a program working. At a basic level the step and back buttons along with viewing the state of the memory and registers would suffice for simple programs; however, for bigger programs stepping one instruction at a time may not be enough or too time consuming. So here are more ways to get more control over a running program.

All of the following can be created from the Debug menu or by right clicking the address and selecting what you want from the context menu popup.

Breakpoints

A breakpoint should be familiar to anyone who has used a debugger before. A breakpoint simply stops execution of the program when it is encountered. If a breakpoint is at an address then the program will stop once the instruction is executed.

Breakpoints are customizable, you can mark the number of times a breakpoint is triggered before it is inactive. You can also have conditional breakpoints, that is, breakpoints that only stop the program when a certain condition is met.

Watchpoints

A watchpoint is similar to a breakpoint in that it stops the program the difference is that it isn't tied to an instruction. A watchpoint is tied to the action of storing to an address or register. The address or register the watchpoint is interested in is called its target. Once the target is written to a condition is evaluated and if it is true then the program will stop.

Again these are customizable, you can mark the number of times the watchpoint is triggered before it is inactive.

Blackboxes

Marking something as a blackbox will never step into it if the step button is used. This should be used on subroutines/traps you know are correct and don't want to switch between the step and next line buttons.

Other features of complx

Console Input

Input and output is specified in the console window which pops up as a separate window when complx is loaded. It is best to specify the console input before your program is ran. Any output (and warnings) will also appear in the window. If there is no input in the console and any instruction that requests the state of the keyboard input is executed then the program will stop and wait for you to type in input. All of the buttons at this time will read “NO IO” to mean there is no input. At this time you may not advance the program or load a new program. To exit this state enter some characters into the console.

Memory View

So now a list of other things the Memory View widget can do.

The memory viewer in the main window of complx will always follow the PC, if you want a memory view tied to a specific area of memory this can be achieved by first 1) Creating a new memory view (View > New View or Ctrl+V) then 2) Scrolling the new memory view to the address you are interested in (View > Goto Address or Ctrl+G) note that the dialog prompt can take an Expression, a Symbol, or Any Memory address in the format xABCD.

The Memory view is editable. You can modify a memory address in hexadecimal, decimal, binary, and even via the instruction column. As a reminder do not use the instruction column to type out your programs as there is no way to save your program via this method. You can also modify the symbol name bound to an address by modifying the Label column.

Mousing over any instruction in the instruction column will bring up a tooltip with the comments for that line of code.

If you would like to freely scroll the memory view without scrolling too far, then you should hide any memory addresses you don't care about. You can do this via View > Hide Addresses > *. This setting can be configured for each MemoryView you create. A quick explanation of the options in this submenu

Show all Addresses

Self explanatory reverts the memory view back to showing all addresses.

Show only Code/Data

Shows only addresses modified when your program was loaded into the assembler.

```
Example
.orig x3000
.blkw 16
.end
.orig x4000
.fill 1
.end
```

Upon using show only code/data the following addresses will only be viewable

x3000-x3010 and x4000

all other addresses will be hidden.

Show Nonzero Addresses

This option scans memory and hides all addresses that contains 0.

Custom

You are allowed to specify what regions of memory you want to be viewable.

The format of this is a list of ranges delimited by a comma in the format start-end_inclusive

example

x3000-x4000, xEF00-xEFFF

Will show x3000-x4000 and xEF00-xEFFF

True Traps Mode

By default any traps executed (including HALT) are only emulated and executing any user defined traps will print out a warning. To disable the trap emulation enable True Traps Mode via State > True Traps.

Interrupts Mode

By default interrupts will not be issued and if any device generates an interrupt it will not be handled. You can enable interupts via State > Interrupts.

Call Stack Viewer

A viewer for the activation stack is also kept track of given your subroutines are well formed. To access this feature Debug > Call Stack. You can improve the view of this dialog by annotating your subroutines (see Subroutine Annotations) so that information is available for processing the activation stack.

In addition to viewing the call stack you can also use this dialog to rewind back to a particular function call in the call stack and viewing that function calls particular stack frame.

Expressions

For any prompt requiring a memory address or a symbol you can give it an expression instead. Expressions are also used to determine when a watchpoint stops the program. The condition of a watchpoint (or breakpoint and blackbox for that matter) is an expression that gets evaluated to determine whether it temporarily stops the program. Note that if an expression evaluates to a non zero value it is considered to be true, otherwise it is false.

Here is a list of variables you can use in expressions:

Registers - R0-R7

Program Counter – PC

Value in memory address – MEM[ADDR]

Any symbol (will resolve to the address where the symbol lives).

Examples

R0 * 5

MEM[x5000] – MEM[x5001]

MEM[MEM[x7000]]

MEM[R4]

PC == HELLOWORLD

For a full list of operators that are supported

- Arithmetic (*, /, %, +, -)
- Bitwise operators &, |, ^
- Logical operators (&&, ||, !&, !|). That is logical and, or, nand, and nor.
- Equality operators ==, !=
- Shifts <<, >>
- Relational operators < > <= >=.
- Parenthesis ()

Assembly Files Extended

Debugging Comments

These are special smart comments that will automatically set up debugging breakpoints and watchpoints within any simulator in complx-tools. Note that use of these comments will not affect the testing environment in lc3-test nor any autograders. Please see Debugging for an overview of what breakpoints, watchpoints, and blackboxes are.

Breakpoints

To specify in a comment to create a breakpoint at a specified address you can use one of two ways to create it.

```
;@break address=address/symbol/expression name=label condition=1 times=-1
;@break address name condition times
```

In the first form any parameter can be omitted the default values are given after the equal sign.

In the second the parameters must be given sequentially, that is, if you want to define “times” then you must specify address, name, and condition. However, if you only want to specify the address in the second form you may omit the rest of the parameters.

Default values and expected types for the parameters are as follows:

Parameter	Type	Default
address	Expression, Address, or Symbol	The address below where the comment is placed.
name	String	An empty string
condition	Integer	1 (always break when breakpoint is encountered)
times	Integer	-1 (the breakpoint will never expire)

Watchpoints

To specify a watchpoint as a comment the syntax is very similar to that of a breakpoint.

```
;@watch target=address condition="0" name=label times=-1
;@watch target condition name times
```

The parameter condition is required omitting it will cause the watchpoint to never trigger

Default values and expected types for the parameters are as follows:

Parameter	Type	Default
target	Address, Symbol, or Register	The address below where the comment is placed.
condition	Integer	0 (Watchpoint will never trigger)
name	String	An empty string
times	Integer	-1 (the watchpoint will never expire)

Blackboxes

And to specify blackboxes the syntax is again similar.

```
;@blackbox address=address name=label condition=1
;@blackbox address name condition
```

Default values and expected types for the parameters are as follows:

Parameter	Type	Default
target	Address or Symbol	The address below where the comment is placed.
name	String	An empty string
condition	Integer	1 (You will always skip over the blackbox)

Subroutine Annotations

You can give the simulator more information about your subroutines (that follow the lc3 calling convention) and this will improve the output of the view call stack function in complx (see Call Stack Viewer).

The syntax for this is as follows:

;@subroutine address=address name=label num_params=0

;@subroutine address name num_params

Default values and expected types for the parameters are as follows:

Parameter	Type	Default
address	Address or Symbol	The address below where the comment is placed.
name	String	An empty string
num_params	Integer	0 (Subroutine takes no parameters)

Plugins

And lastly you can extend the simulator and assembler through use of plugins. With plugins you may add new devices, traps, and a new instruction for the LC-3.

To include a plugin with your assembly file the following syntax must be used

```
;@plugin filename=??? vector=??? address=??? interrupt=???
```

The arguments vary by plugin but for a minimum the filename must be given. For plugins introducing new traps vector must be specified, it will be the entry in the trap vector table where the trap lives. For plugins introducing devices address must be specified. This will be the address where the device register lives. If the plugin generates interrupts then the interrupt parameter must be specified.

Parameter	Type	Description
filename	Filename	Must be specified. Name of file without extension and without lib in the name. So a file named liblc3_udiv.so will be specified here as lc3_udiv
vector	Address	Address to install Trap plugin into trap vector table.
address	Address	Address where Device Register plugin will live.
interrupt	Address	If plugin generates interrupts the interrupt vector that is sent

Troubleshooting

Assembly File Errors

Syntax Error

“Syntax Error on line <lineno>: <line>”

This means that you did not follow the syntax for an instruction. Look in PattPatelAppA.pdf or Appendix A of the text for the instruction format.

Orig/End Matchup

“No matching .end found on line <lineno> for <line>”

Means what it says for the .orig statement given you did not have a .end to matchup with it.

Orig overlap

“Code sections <section1> and <section2> overlap”

You have two .orig statements whose contents overlap with a another code section.

All .orig/.end pairs must be disjoint areas of memory.

Stray .end

“No matching .orig found for .end on line <lineno>”

You got one too many .ends in your file

Stray data

“Stray data found on line <lineno>: <line>"

Some data or instructions was found outside a .orig/.end block.

Undefined Symbol

"Undefined symbol <symbol> found on line <lineno>"

You are referring to a symbol that does not exist

Duplicate Symbol

"Duplicate symbol <symbol> found on line <lineno>"

You can't define a symbol twice. You have code somewhere of this form

```
MY_SYMBOL ADD R0, R0, R0
[code here]
MY_SYMBOL ADD R0, R0, R0
```

Created symbols must be unique

Multiple Symbol

"Multiple symbol <symbol1> and <symbol2> found for address <address> on line <lineno>"

You can't associate more than one symbol to an address. You have code of the form

```
HI
THERE ADD R0, R0, R0
```

HI and THERE will refer to the same address which is not allowed.

Invalid Symbol

"Invalid symbol <symbol> found on line <lineno>"

Symbols can only contain alphanumeric characters and _, must start with a letter, must be less than 20 characters, and can not be an instruction or register name.

Invalid Register

“Invalid Register <context>”

The LC3 only has 8 registers R0-R7.

Invalid Instruction

"Invalid instruction <context> found on line <lineno>"

Your opcode name was not correct, see the ISA document for valid instruction names.

Invalid Directive

"Invalid assembler directive <context> found on line <lineno>"

The valid assembler directives are .orig .end .blkw .fill .stringz

Invalid Flags

"Invalid condition code flags <flags> found on line <lineno>"

The flags for BR must be in the order nzp. N must come before Z which comes before P.

Invalid Character

"Invalid character constant found on line <lineno>: <line>"

You gave an invalid chracter that is you either did one of the following

'67' – The ' specifies a character and only one character must be contained in it unless you are using an escape sequence, that is, '\n' is valid.

.fill “Hello World'. This is invalid .fill will expect a character as it is filling one location. If you want to add a string then use .stringz

Invalid Number

"Found signed number expecting unsigned number on line <lineno>: <line>"

Example: You pass a negative number to .blkw or something like TRAP -3

Number Overflow

"<number> is too big for an immediate value expected <x> bits got <y> bits found on line <lineno>"

Sorry but you can't do things like ADD R0, R0, 105. Please reread the ISA document, ADD requires the immediate value to fit in 5 bits and is a 2's complement number therefore -16 to 15 are valid values.

Offset Overflow

"<offset> is too far away for an offset expected <x> bits got <y> bits found on line <lineno>"

See above LD/LDI/ST/STI has a max of 9 bits LDR/STR has a max of 6 bits. You may want to restructure your code.

And writing code like LD x3005 is just plain wrong, LD and friends work based on offsets not absolute addressing

Memory Overflow

"Can't add by <num> found on line <lineno>"

Example

```
.orig xFFF0

.blkw 100

.end
```

No.

Scan Overflow

"I'm at the end of memory (xFFFF) and I refuse to wrap around! found on line <lineno>"

Example

```
.orig xFFFF

ADD R0, R0, R0

ADD R0, R0, R1

.end
```

Again No.

File Error

"Could not open <file> for reading\nAre you sure the file is in your current working directory?"

Means what it says, might want to use ls to see if your file you are trying to run is in the directory you are in, if not then use cd to change the directory to where it is.

Unterminated String

"Unterminated string on line <lineno>: <line>"

Means what it says, you forgot a ' or “

Malformed String

"Malformed string on line <lineno>: <line>"

You either gave a string that had bad escape sequences, you did not enclose the string in “”'s. Be careful with copying code from documents as the “”s are a different character.

Extra Input

"Extra input found at end of line <lineno>: <line>"

Means what it says, you can get this from specifying too many paramters to an instruction. Remove them.

Plugin Failed to load

"Plugin <file> failed to load at line <lineno>”

The assembler could not find your plugin or it was unable to load it due to a difference in version or some other error. Please see Plugins on how to specify this.

Runtime Warning Messages

Reading beyond end of input. Halting

The LC3 ran out of input, you should never get this warning as the simulator will wait for your input.

Writing <data> to reserved memory at <address>

x0000-x2FFF and xFE00-xFFFF is considered reserved memory. In the first section the trap vector table, interrupt vector table and lc3os code is located there. In the second is special device registers.

You are only allowed to write to the lower memory if you are in kernel mode (that is within a trap handler).

You are only allowed to write to the higher memory if there exists a device there or you are in kernel mode.

Reading from reserved memory at <address>

Same as above

Unsupported Trap <vector>. Assuming Halt

Your code executed a trap instruction that isn't one of the predefined traps or you are writing your own trap and you did not enable true traps mode.

Unsupported Instruction x%04x. Halting

Your code executed data whose first four bits was 0xD for the invalid opcode instruction.

RTI executed in user mode. Halting.

Means what it says, your code executed data whose first four bits was 0x8 for RTI (return from interrupt). This instruction can only be called within an interrupt handler.

Trying to write character x%04x

You are trying to write an invalid character (character > 255).

PUTS called with invalid address x%04x

See Reading from reserved memory.

Trying to write to the display when its not ready

You must poll the DSR before seeing if it is okay to write to the DDR.

Trying to read from the keyboard when its not ready

Same as above you must poll the KBSR before writing to KBDR

Turning off machine via the MCR register

Your code wrote to address xFFFE which is the MCR (Machine Control Register). This register can only be written to in kernel mode and is responsible for the implementation of the HALT instruction.