Table of Contents

Overview

complx is a suite of educational tools for learning lc3 assembly. It includes both a gui and cli based simulator (named complx and comp respectively), an assembler (as2obj), python bindings via pyLC3, and an autograder framework written in python. Complx also be extended with plugins that add additional functionality to the LC3. The tools also come with a C++ interface to the LC3 (liblc3) along with python bindings to it via pyLC3.

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

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] [-hex|-bin|-full] [output_file_prefix]

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
- -hex Generate hex file output
- -bin Generate machine code output
- -full Generate a txt file with instructions in all formats
- output_file_prefix output filename prefix

Note if a parameter is enclosed in []'s it means it is optional. When specifying it you do not include the []'s. Only one (or none) of -hex -bin or -full should be given

For -hex and -bin the format of the file is as follows

x<orig_address>

followed by hexadecimal or binary numbers from assembling the file.

Example:

halt.asm

```
.orig x3000
        HALT
.end
```

option	-hex	-bin	-full
flename	halt.hex	halt.bin	halt.txt
content	x3000 xF025	x3000 1111000000100101	x3000 xF025 -4059 1111000000100101 HALT

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)
- --fill-registers=(random or int) Set initial register fill value strategy
- --fill-memory=(random or int) Set initial memory fill value strategy

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

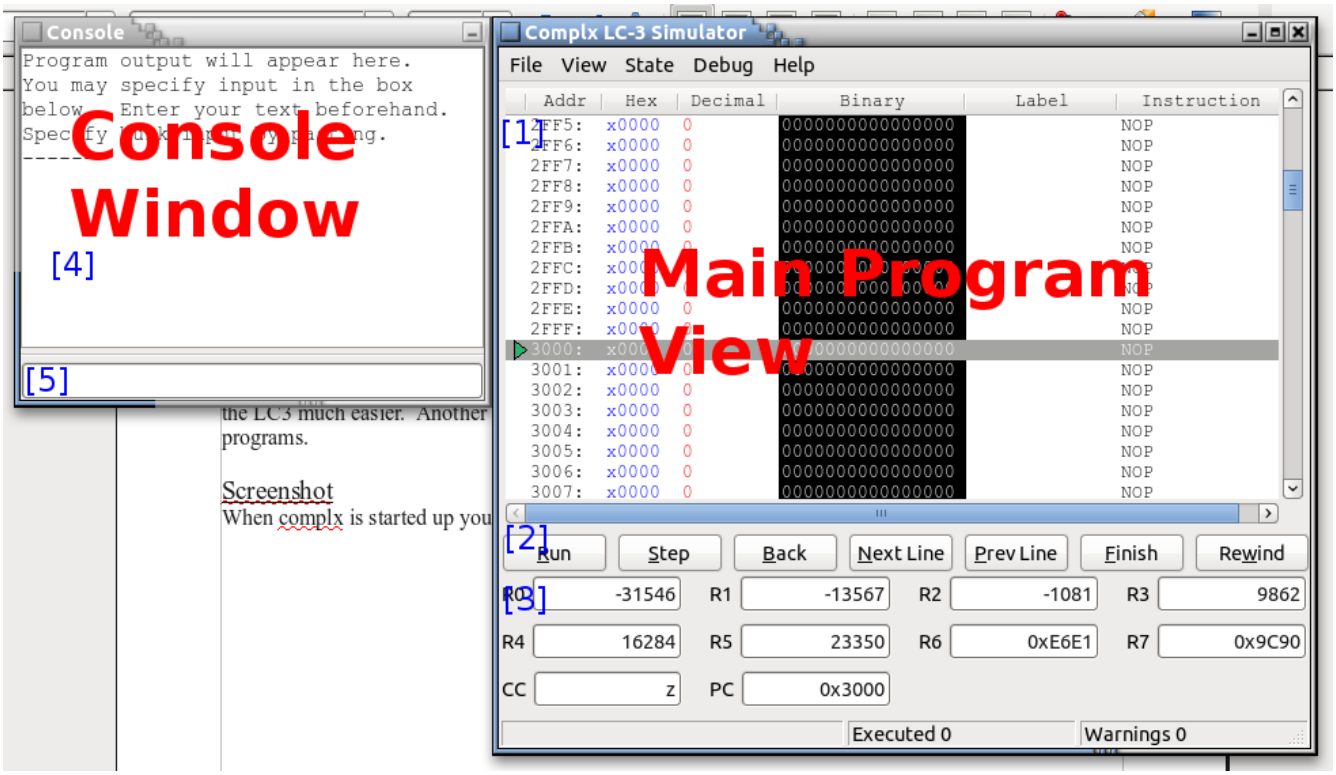
- Load – Ctrl + O
- Reload – Ctrl + R
- Advanced Load –

Load Randomizes memory and registers and loads your file.

Reload Loads the last loaded assembly file under the same loading conditions. It will not query you for a file unless one has not been loaded before.

Advanced Load Allows you to set the initial values of registers and memory, console input, and other things.

Interface tour



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.

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 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

E001 - 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.

Common mistakes to get this error include:

- `LD R0, R1` *(LD requires a register and a label or hardcoded PC offset)*
- `LDR R0, R1, R2` *(LDR requires a destination register, a source register and a hardcoded offset)*
- `STR R0, R1, HELLO` (STR requires a destination register, a source register, and a hardcoded offset)
- `ADD R0, R0, HELLO` (ADD instruction’s immediate version requires a hardcoded offset).

E002 - Orig/End Matchup

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

For an .orig statement given in your code you did not have a .end to matchup with it.

E003 - Orig overlap

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

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

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

E004 - Stray .end

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

You got one too many .ends in your file

E005 - Stray data

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

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

E006 - Undefined Symbol

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

You are referring to a symbol that does not exist.

E007 - Duplicate Symbol

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

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

```
.orig x3000
     LOOP
     […]
     BRZP LOOP
     ; some more instructions
     LOOP
     […]
     BRNZP LOOP
.end
```

Created symbols must be unique. A common occurrence of this is when the programmer uses the symbol name loop to refer to a place to branch to. You should give such symbols more descriptive names than just LOOP.

E008 - 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

```
.orig x3000
     HI
     THERE ADD R0, R0, R0
.end
```

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

E009 - 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.

E010 - Invalid Register

“Invalid Register <context>”

The LC3 only has 8 registers R0-R7.

E011 - Invalid Instruction

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

Your opcode name was not correct, see the ISA document for valid instruction names. Note that this includes having any punctuation after the instruction name such as “ADD,”

E012 - Invalid Directive

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

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

E013 - 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. N must come before P. Z must come before P. Any different ordering of NZP results in this error message.

E014 - 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

E015 - Invalid Number

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

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

E016 - Invalid Bytes

Invalid byte <garbage> found in code.\nUse a text editor that can display non ASCII / unprintable characters and remove them.

I recommend SciTE, you can get this error message easily by coping and pasting code from word documents. Editors of this type will sneakily replace characters with another.

E017 - Number Overflow

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

You can't use instructions like `ADD R0, R0, 105`. Each instruction taking in an immediate value has a limited number of bits to specify the immediate value. Using a number too big will overflow that immediate value causing this error message.

E018 - Offset Overflow

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

See E016. This error message is similar for instructions taking in a pc fffset or hardcoded offset there's still a defined number of bits for specifying these values. Examples of wrong instructions:

```
.orig x3000
     LD R0, x3005
     LDI R1, HELLO
     LDR R2, R3, 1000
.end
.orig x4000
     HELLO .fill 3
.end
```

LD – Perhaps you are thinking this will load the value at mem[x3005] however this is wrong as LD uses PC relative addressing.

LDI – The PC offset to get to x4000 here is x4000 – x3002 = x0ffe. xffe is a 13 bit number (12+1 because 2's complement) way more than the 9 bits specifying the offset.

LDR – The hardcoded offset for LDR is a 6 bit 2's complement number. 1000 is too big.

E019 - Memory Overflow

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

Example:

```
.orig xFFF0
     .blkw 100
.end
```

xFFF0-xFFFF consists of 10 addresses, blkw is trying to allocate 1000 addressses at the end of memory.

E020 - 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
```

The second instruction would be at x10000, which is an invalid address.

E021 - File Error

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

Are you sure you are in the correct directory? The output of the `ls` (Linux/Mac) or `dir` (Windows) command should show your assembly file.

E022 - Unterminated String

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

You forgot a ' or “

```
LABEL .fill ‘A
STRING .stringz ”Unterminated
```

E023 - Malformed String

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

You either gave a string that has 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 from **"**.

E024 - Extra Input

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

You can get this error message from specifying too many parameters to an instruction. Example:

```
ADD R0, R0, R0, R0
```

E025 - 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.

E026 - Invalid LC3 Version

Invalid LC3 Version <version>

Valid LC-3 versions at the time this document was updated are 0 (initial version) and 1 (2019 revision).

Runtime Warning Messages

W000 - 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.

W001 - 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 privilege 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 privileged mode.

W002 - Reading from reserved memory at <address>

Same as above

W003 - 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.

W004 - Unsupported Instruction <hex>. Halting

Your code executed data whose first four bits was 0xD which is an invalid opcode.

W005 - Malformed Instruction <hex>. Halting

You executed a bad instruction. The only way to get this warning message is if you execute data or random memory addresses. Note that malformed instructions will appear in the simulator in the instruction column followed by an asterisk.

For a complete enumeration of all instructions that this warning can be caused on is listed below.

instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	1	x	x	x	x	x	x	0	**0**	**0**	x	x	x
AND	0	1	0	1	x	x	x	x	x	x	0	**0**	**0**	x	x	x
BR	0	0	0	0	**0**	**0**	**0**	x	x	x	x	x	x	x	x	x
JMP	1	1	0	0	**0**	**0**	**0**	x	x	x	**0**	**0**	**0**	**0**	**0**	**0**
JSRR	0	1	0	0	0	**0**	**0**	x	x	x	**0**	**0**	**0**	**0**	**0**	**0**
NOT	1	0	0	1	x	x	x	x	x	x	**1**	**1**	**1**	**1**	**1**	**1**
RTI	1	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
TRAP	1	1	1	1	**0**	**0**	**0**	**0**	x	x	x	x	x	x	x	x

Bits in shaded regions must be set to that bit pattern in order to be considered valid.

Note that all ADD/AND instructions with an immediate value are valid, since all bits are devoted to expressing that form of instruction.

W006 - RTI executed in user mode. Halting.

Your code executed an instruction whose opcode was 0x8 for RTI (return from trap or interrupt). This instruction can only be called from within an interrupt handler or trap.

W007 - Trying to write character x%04x

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

W008 - PUTS called with invalid address x%04x

See W002 Reading from reserved memory.

W009 - 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.

W010 - Trying to read from the keyboard when its not ready

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

W011 - 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 privelege mode and is responsible for the implementation of the HALT instruction.

W012 - PUTSP called with invalid address x%04x

See W002 - Reading from reserved memory.

W013 - PUTSP found an unexpected NUL byte at address x%04x

Putsp found a nul byte while writing characters, it is expected that the following address contain the value x0000 which wasn't found. Or a NUL byte was found in the least significant 8 bytes and the most significant 8 bits was not x00.

W014 - Invalid value x%04x loaded into the PSR

This means that when an RTI instruction was executed a value was written to the PSR which that multiple of the N,Z,P (or none) bits was set, leading to an invalid state for the PSR.

W015 - Executing trap vector table address x%04x

This means the PC ended up at address x0000-x00FF where the TRAP vector table lives. Generally this means that your code loaded an incorrect value into the PC, usually due to bad handling of the stack for a subroutine, trap, or interrupt.

W016 - Executing interrupt vector table address x%04x

Similar to W015. This means the PC ended up at address x0100-x01FF where the Interrupt vector table lives. Generally this means that your code loaded an incorrect value into the PC, usually due to bad handling of the stack for a subroutine, trap, or interrupt.