

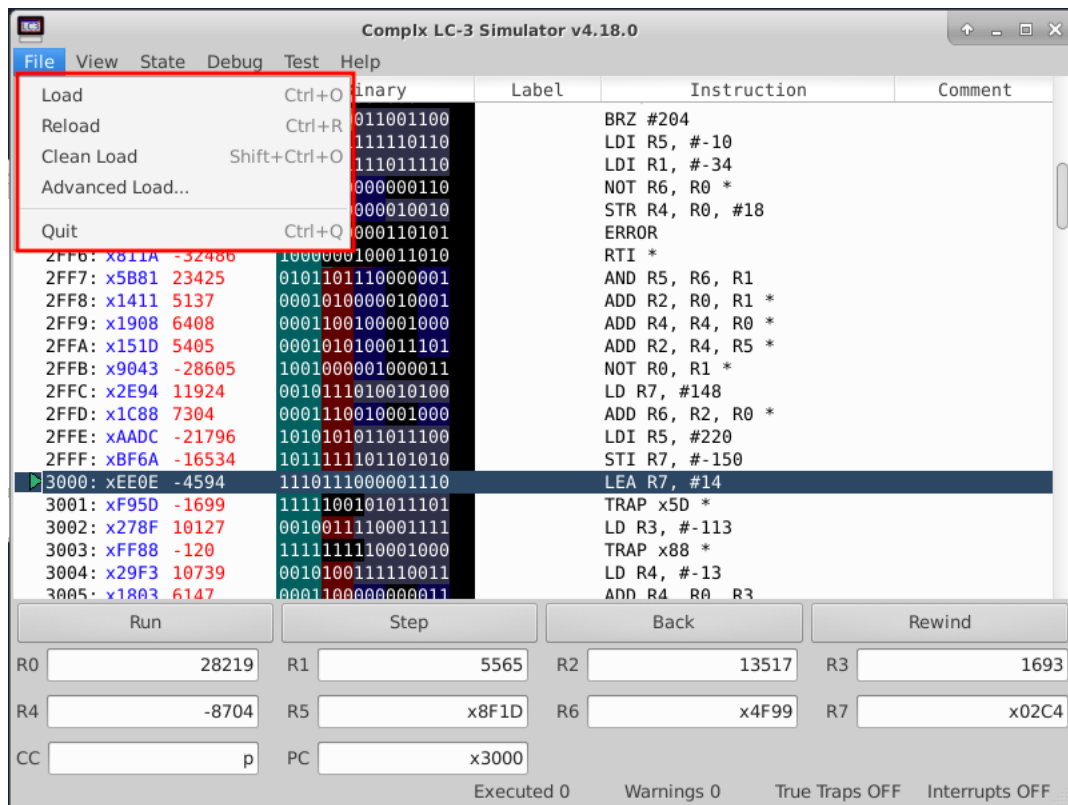
# Introduction to Complx

Your friendly neighborhood 2110 TAs

Have Fun Debugging!

## 1 Components

### 1.1 Loading Files in Complx



Complx comes with 4 different ways to load in your assembly program:

#### 1. Load

This loads a selected assembly program into memory, with the memory and registers assigned random values. It is important that your code works with this, since you can never assume any value in memory or a register.

#### 2. Reload

This reloads the current assembly file loaded into Complx, again with random memory and registers.

### 3. Clean Load

This is load but memory and the registers are initialized to 0. This is helpful when debugging to make what your code does more clear.

### 4. Advanced Load

This allows you to specify starting values for registers and memory as random, 0, or some specific value, specify initial console input if needed, or change the starting value of the PC.

## 1.2 Different Components of Complex

### 1.2.1 Memory View

Addr	Hex	Decimal	Binary	Label	Instruction	Comment
2FFA	x1510	-2495	000101010001101		ADD R4, R4, R5 *	
2FFB	x9043	-28605	100100001000011		NOT R0, R1 *	
2FFC	x2E94	11924	001011010010100		LD R7, #148	
2FFD	x1C88	7304	000110010001000		ADD R6, R2, R0	
2FFE	xAADC	-21796	101010101101100		LDI R5, #220	
2FFF	xBF6A	-16534	101111101101010		STI R7, #-150	
3000	x0000	0	000000000000000		LEA R0, HELLO	Your code here!
3001	x0000	0	000000000000000		OUTS	
3002	x0000	0	000000000000000		R1,	
3003	x0000	0	000000000000000		R2,	
3004	x0000	0	000000000000000		R3,	
3005	x0000	0	000000000000000		ADD R0, R0, R3	
3006	x1003	4099	0001000000000011		OUT	
3007	xF021	-4063	1111000000100001		HALT	
3008	xF025	-4059	1111000000100101		NOP ('T') *	
3009	x0054	84	0000000001010100	HELLO	NOP ('h') *	
300A	x0068	104	0000000001101000		NOP ('e') *	
300B	x0065	101	0000000001100101		NOP (' ') *	
300C	x0020	32	0000000001000000		NOP ('s') *	
300D	x0073	115	0000000001110011		NOP ('u') *	
300E	x0075	117	0000000001110101		NOP ('m') *	
300F	x006D	109	0000000001101101			

Run Step Back Rewind

R0 1678 R1 -13448 R2 9571 R3 -10001

R4 583 R5 x52C9 R6 xEE09 R7 x3D59

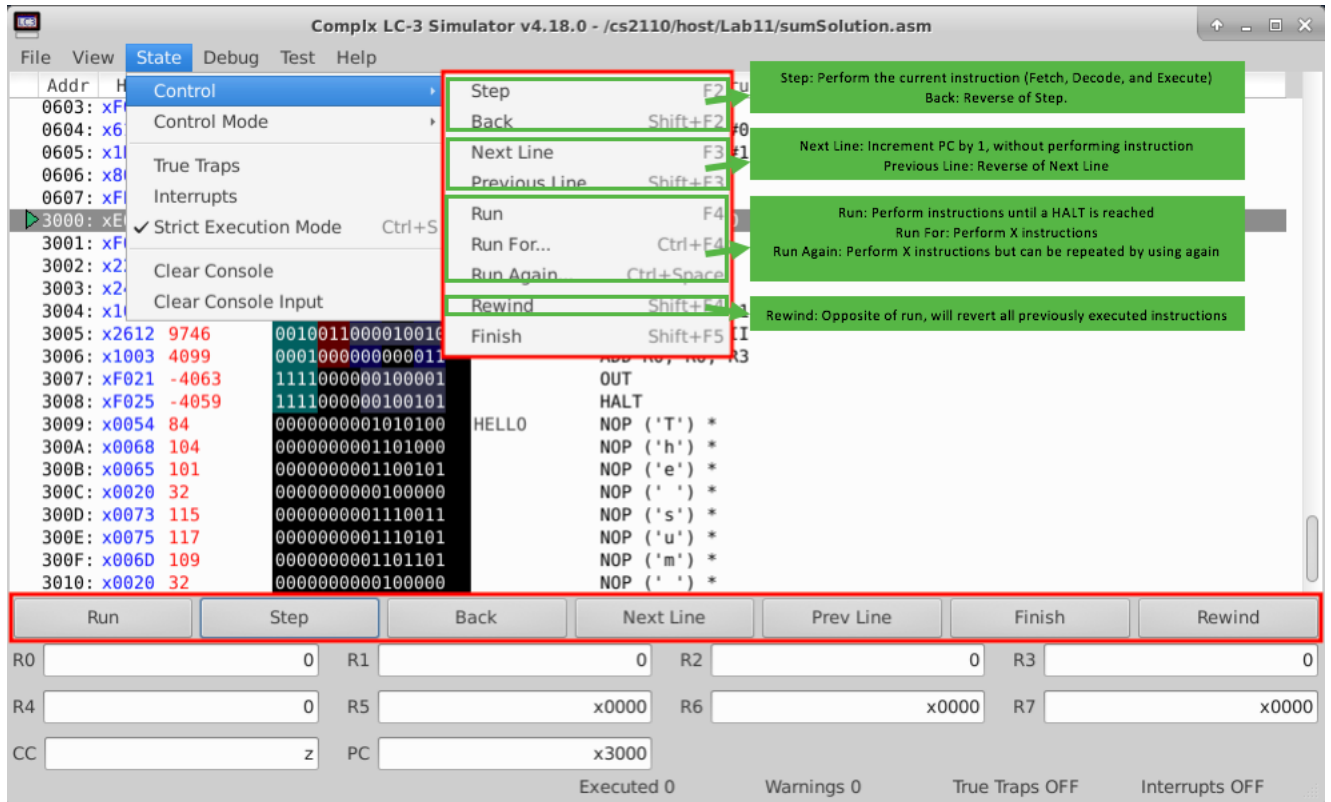
CC p PC x3000

Executed 0 Warnings 0 True Traps OFF Interrupts OFF

### Useful tips!

1. You can go to any memory address by pressing ctrl+g and typing the memory address (for example x3000).
2. If you want to view two different areas of memory simultaneously, you can open another memory view window using view → New View, or ctrl+v.
3. You can use view → Hide Addresses to configure which addresses are displayed in the memory view.
4. You can use view → disassemble to change how the instructions are displayed.
  - (a) Basic: Labels are replaced with calculated offsets and TRAPS are replaced with TRAP and the corresponding trap vector.
  - (b) Normal: Default display of instructions.
  - (c) High Level: Instructions are displayed in C-like syntax.

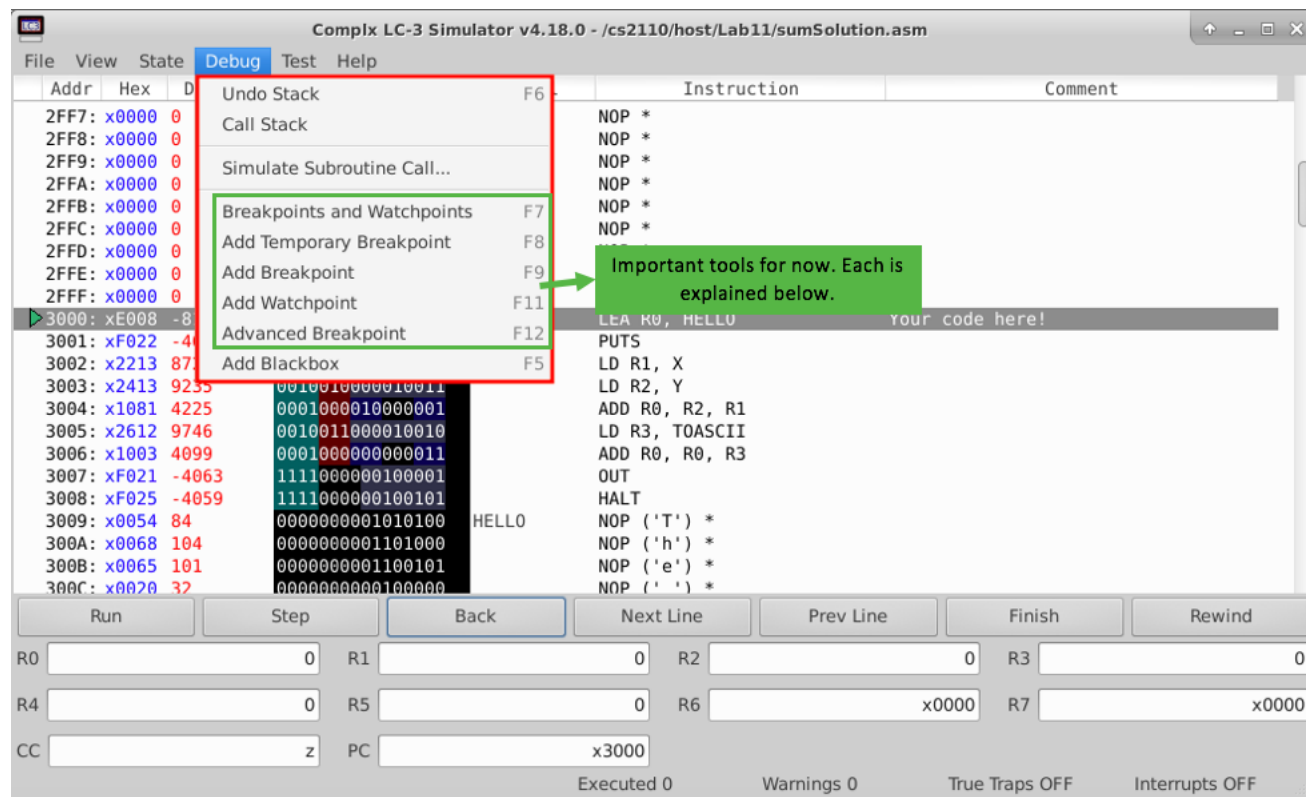
### 1.2.2 Controls



### 1.2.3 Registers

The bottom of Complx displays the values in all 8 registers, as well as the PC and CC. You can double-click on a register to cycle through displaying it as decimal, binary, and hexadecimal!

## 1.3 Breakpoints and Watchpoints



### 1.3.1 Breakpoints

Breakpoints are lines in our assembly code that we specify Complix to stop at **before** executing it. So if we were to use the Run command, it would stop at a breakpoint instead of a HALT instruction. This is really useful if we want to figure out the status of our program sometime during its execution! Each breakpoint has the following properties:

1. Name: A name for the breakpoint, which can be anything.
2. Address: The address where the breakpoint is being added. This is the instruction we are breaking at.
3. Condition: A condition for when the breakpoint actually stops execution upon being hit. This can be 1 (true) if you always want it to stop execution, or checking something like:  $R1 == 0$ , for example, or  $\text{mem}[x3000] == 5$ .
4. Times: How many times it should be hit before stopping. Use -1 for an indefinite amount.
5. Hits: Not modifiable, but the number of times the breakpoint has been reached.
6. Enabled: Whether or not the breakpoint will stop execution.

To create a new breakpoint you can select an instruction in Memory View and use any of the following (Note: the selected line is highlighted, the green arrow refers to where the PC is pointing):

1. Add Temporary Breakpoint: This creates a single breakpoint that will be destroyed after being hit once.

2. Add Breakpoint: This creates a single breakpoint with Times = -1.
3. Advanced Breakpoint: This creates a single breakpoint and lets you specify the different properties.

You can also create a breakpoint directly in your assembly code! You would add one as a comment to a line with the following syntax:

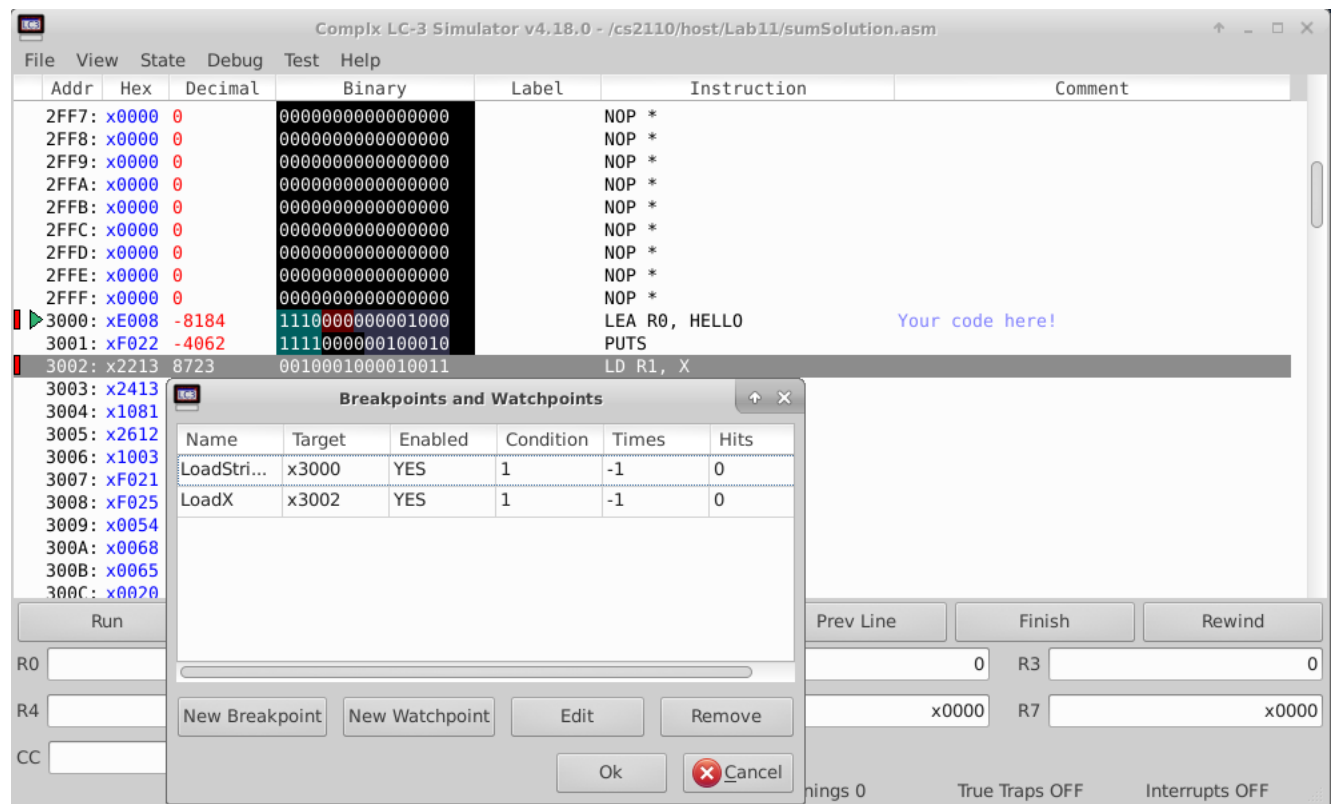
```
;@break name= condition=
```

For example:

```
LEA R0, HELLO ;@break name=LoadStringAddr, condition=1
```

would create a new breakpoint for the instruction LEA R0, HELLO.

Breakpoints show up in Complx denoted by a red rectangle on the line of the breakpoint, but you can also see them listed in Debug→Breakpoints and Watchpoints.



### 1.3.2 Watchpoints

Watchpoints are very similar to breakpoints except you specify a memory address or register, and they trigger when the value at the memory address or register changes. Just like breakpoints they also have properties for: name, condition, times, hits, and enabled. You can create these with Debug → New Watchpoint or from within the Breakpoints and Watchpoints table.

## 2 Debugging in Complx/Docker

When you turn in your files on gradescope for the first time, you might not receive a perfect score. Does this mean you change one line and spam gradescope until you get a 100? No! You can use a handy tool known as tester strings.

1. First off, we can get these tester strings in two places: the local grader or off of gradescope. To run the local grader:
  - Mac/Linux Users:
    - (a) Navigate to the directory your homework is in. **On your computer, not the docker image**
    - (b) Run the command `sudo chmod +x grade.sh`
    - (c) Now run `./grade.sh`
  - Windows Users:
    - (a) On **docker quickstart**, navigate to the directory your homework is in
    - (b) Run `./grade.sh`

When you run the script, you should see an output like this:

```
TEST: testGates PASSED
TEST: testReverse PASSED
TEST: testPhone PASSED
TEST: testLinkedList FAILED

NODES="[(16384, 0, -7)]", DATA="-7", LENGTH="1" -> NODES="[(16384, 0, 1)]": Code did not halt normally.
This was probably due to an infinite loop in the code.
PC: x300f
Instruction last on: BR LOOP

String to set up this test in complx: 'BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAAABAAQAZBAAAADQwMDABAAAA
DQwMDEBAAAA+f/'
NODES="[(16384, 16392, 7), (16386, 16388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 15)]", DATA="15", LENGTH=
"8" -> NODES="[(16384, 16392, 7), (16386, 16388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 5)]": Code did not
halt normally.
This was probably due to an infinite loop in the code.
PC: x300f
Instruction last on: BR LOOP
```

Copy the string, starting with the leading 'B' and ending with the final backslash. Do not include the quotations.

**Side Note:** If you do not have docker installed, you can still use the tester strings to debug your assembly code. In your gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslash and again, don't copy the quotations.

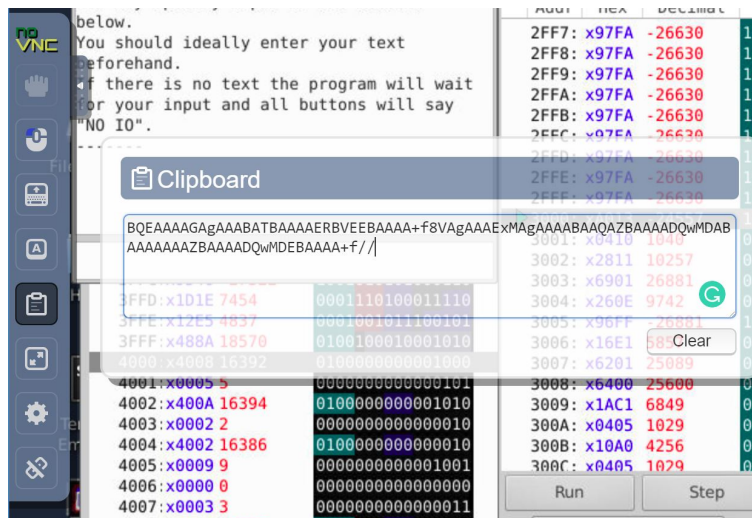
```
LINKEDLIST: testLinkedList (0.0/30.0)

LENGTH="1" -> NODES="[(16384, 0, 1)]": Code did not halt normally.
loop in the code.

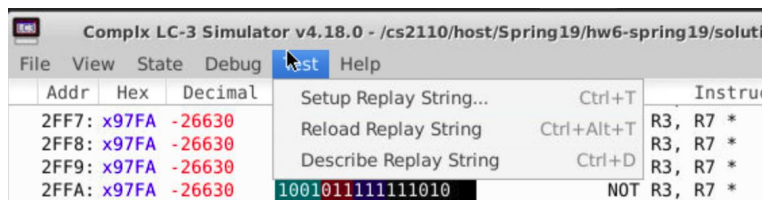
'BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAAABAAQAZBAAAADQwMDABAAAA
16388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 15)]", DATA="15"
loop in the code.

'BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAAABAAQAZBAAAADQwMDABAAAA
```

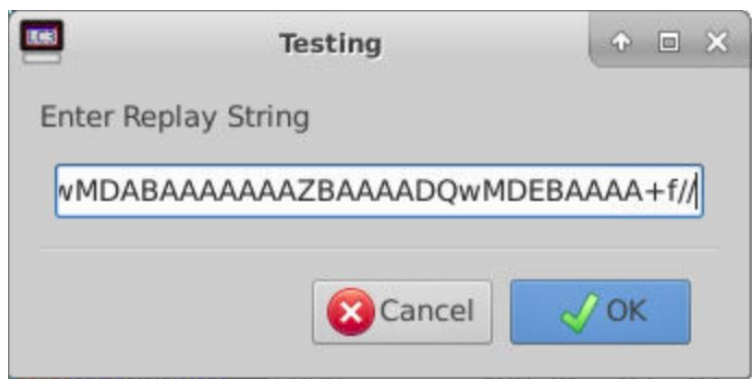
2. Secondly, navigate to the clipboard in your docker image and paste in the string.



- Next, go to the Test Tab and click Setup Replay String



- Now, paste your tester string in the box!

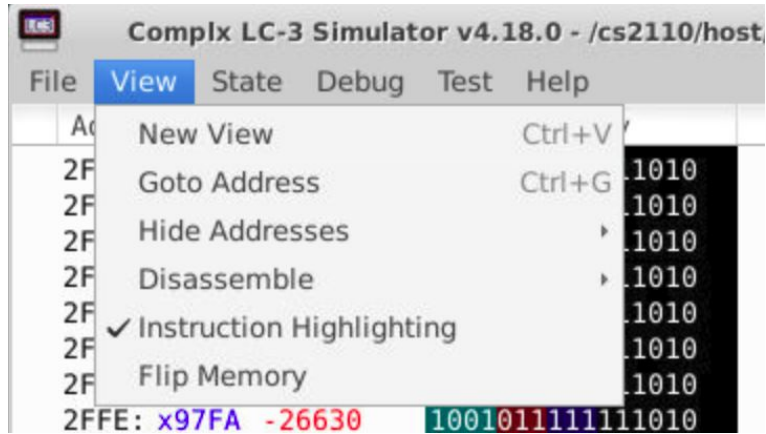


- Now, complx is set up with the test that you failed! The nicest part of complx is the ability to step through each instruction and see how they change register values. To do so, click the step button. To change the number representation of the registers, double click inside the register box.

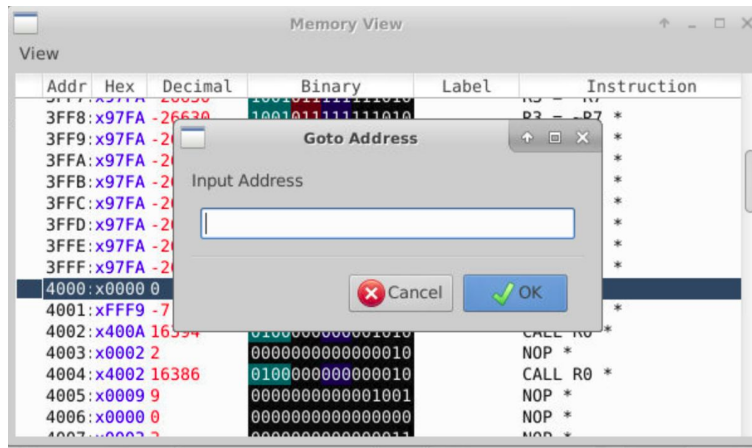


- If you are interested in looking how your code changes different portions of memory, click the view tab and indicate 'New View'





7. Now in your new view, go to the area of memory where your data is stored by CTRL+G and insert the address



Ta-Da! You now have a handy-dandy way to debug your assembly code.

**Please Note:** If you ask your TAs to debug your assembly code, we will ask to see if you have already stepped through your code using complx. If you have not, we will show you how to debug and let you debug your code yourself.