CS 2110 Homework 5
Intro to Assembly

Annelise Lloyd, Zaid Akkawi, Jonathan Marto, Max Everest, Sebastian Jankowski

Version 1.0

Contents

1 TLDR

1.1 Debugging Assembly

For information on debugging or the autograder, jump to section 5. Please note that Docker must be running, but it doesn't need to be within the image.

1.2 Modulus

Given values x and mod, calculate x % mod and store this at the address `ANSWER`. The pseudocode can be found in section 3.2.

Relevant labels:

- `X` and `MOD` where you can load the inputs from.
- `ANSWER` where you will store your result, x mod m.

Say x = 14 and mod = 3. After running your program, mem[`ANSWER`] should equal 2.

1.3 Build Minimum Array

Given two competing arrays A and B of the same length, fill in a third array located C in reverse order. For each index, if A[i] $\leq$ B[i], set C[length - i - 1] = 1. Otherwise, set C[length - i - 1] = 0. Store these values in the array represented by `C`. The pseudocode can be found in section 3.3.

Relevant labels:

- `A` which holds the address of A.
- `B` which holds the address of B.
- `C` which holds the address of C, your solution array.
- `LENGTH` which holds the size of the arrays.

Say A is [-4, 6, 0] and B is [1, 5, 2]. After running your program, C should be [1, 0, 1]. 0

1.4 Octal String to Int

You are given an unsigned octal number encoded as a string. Translate this string to its numerical value. The value stored at `RESULTADDR` represents another different address. Store your numerical value result at this different address. The pseudocode can be found in section 3.4.

Relevant labels:

- `OCTALSTRING` which holds the starting address of the octal string
- `LENGTH` which holds the length of the octal string
- `RESULTADDR` which holds the **address** of where you will store your numerical value result.
- `ASCII` which holds the value -48

Say the string at `OCTALSTRING` was "2023". After running your program, mem[mem[`ANSWERADDR`]] should equal 1043 (base 10).

1.5 Palindrome

Given a string, calculate if it's a palindrome. `ANSWERADDR` should contain another address. At this address, store true if the string is palindrome and false if it is not. Let's agree to interpret a 1 in memory as true, and a 0 in memory as false. The pseudocode can be found in section 3.5.

Relevant labels:

- `STRING` which holds the address of the string that you will evaluate
- `ANSWERADDR` which holds the **address** of where you should store the boolean result.

Say the provided string is "racecar". After running your program, mem[mem[`ANSWERADDR`]] should equal true, which is the same as 1 for our purposes.

Recall that we represent the end of a string with a null terminator, which is the numerical value 0.

1.6 Advice from your TAs

- When converting pseudocode into assembly, become human compilers and translate one line of pseudocode into one (or more) lines of assembly. One line at a time. Don't look at the pseudocode and try to optimize things in your head and write the entire function. Trust us when we say translating one line at a time is the way. So, put on blinders. Ignore previous and future lines of code.
- Don't assume a register still retains a value from a previous LD.
- Don't assume the condition code reflects the register you think it does (NZP).
- Checkout the Appendix for the ASCII table and other helpful resources
- The local autograder is a shell script which should be run in your local terminal (not the docker terminal).

The above strategy of course produces inefficient code. But, it produces correct assembly code, every time. Not clever code. Not optimized code. Not short code. But correct code. So, if you ever find it difficult to write assembly, just go at it one step at a time :) or you could visit office hours/ed discussion :)

2 Overview

2.1 Purpose

So far in this class, you have seen how binary or machine code manipulates our circuits to achieve a goal. However, as you have probably figured out, binary can be hard for us to read and debug, so we need an easier way of telling our computers what to do. This is where assembly comes in. Assembly language is symbolic machine code, meaning that we don't have to write all of the ones and zeros in a program, but rather symbols that translate to ones and zeros. These symbols are translated with something called the assembler. Each assembler is dependent upon the computer architecture on which it was built, so there are many different assembly languages out there. Assembly was widely used before most higher-level languages and is still used today in some cases for direct hardware manipulation.

2.2 Task

The goal of this assignment is to introduce you to programming in LC-3 assembly code. This will involve writing small programs, translating conditionals and loops into assembly, modifying memory, manipulating strings, and converting high-level programs into assembly code. LC-3 Assembly code can be converted directly into the 16 bit instructions used by the LC-3, and is a stepping stone between the actual binary and a programming language.

You will be required to complete the four functions listed below with more in-depth instructions on the following pages:

1. `modulus.asm`
2. `buildMinArray.asm`
3. `octalStringToInt.asm`
4. `palindrome.asm`

The assignment is due by 11:59 PM on June 26th, 2023. You could also submit the assignment by 11:59 PM on June 27th, 2023 for a late penalty of 25% of your overall grade.

2.3 Criteria

Your assignment will be graded based on your ability to correctly translate the given pseudocode into LC-3 assembly code. Check the deliverables section for deadlines and other related information. Please use the LC-3 instruction set when writing these programs. More detailed information on each instruction can be found in the Patt/Patel book Appendix A (also on Canvas under "LC-3 Resources"). Please check the rest of this document for some advice on debugging your assembly code, as well some general tips for successfully writing assembly code.

You must obtain the correct values for each function. While we will give partial credit where we can, your code must assemble with **no warnings or errors** (Complx will tell you if there are any). If your code does not assemble, we will not be able to grade that file and you will not receive any points. Each function is in a separate file, so you will not lose all points if one function does not assemble. Good luck and have fun!

3 Detailed Instructions

3.1 Notes

- Assume all inputs will enforce the assumptions. This means you do not need to test whether the assumptions are true within your code.
- The provided pseudocode will account for all assumptions and edge cases.
- Make sure you conceptually understand what labels are and what using them really means behind the scenes. Many problems will revolve around using labels to load inputs and store outputs.
- Be careful changing anything outside the first .orig x3000 and HALT lines in every file. Watch out for the instructions in each file to know what you can and cannot change for testing. Your autograder's functionality will depend on some of the initial code we provided.
- The algorithms presented for each operation are not meant to be the most efficient. You do not need to make them more efficient.
- Be wary of the differences between instructions like `LD` and `LEA`. When you have an answer, make sure you're storing to the correct address. Trace through your code on Complx if you're not sure if you're using the correct instruction.
- Debugging via Complx helps tremendously. Eyeballing assembly code can prove to be very difficult. It helps a lot to be able to trace through your code step-by-step, line-by-line, to see if each assembly instruction does what you expected.
- You can check if far-away addresses contain expected values in Complx by going to View `>>` GoTo Address.

3.2 Part 1: Modulus

Given a values x and mod, calculate x % mod and store this at the address `ANSWER`.

Assumptions:

- $x \geq 0$
- $mod > 0$

Relevant labels:

- `X` and `MOD` where the inputs are stored in memory. You can load the inputs from these locations.
- `ANSWER` where you will store your result, x mod m.

Say $x = 14$ and $mod = 3$. After running your program, mem[`ANSWER`] should equal 2.

Notice this problem involves a while loop, which means your assembly program will need to know how rewind backwards to the top of the loop. Perhaps there are instructions we can use to directly change the PC and help our program shift around our code.

Suggested Pseudocode:

```
int x = 14;
int mod = 3;
while (x >= mod) {
    x -= mod;
}
mem[ANSWER] = x;
```

3.3 Part 2: Build Minimum Array

Given two competing arrays A and B of the same length, fill in a third array C. For each index, if A[i] $\leq$ B[i], set C[length - i - 1] = 1. Otherwise, set C[length - i - 1] = 0.

Assumptions:

- All provided arrays will have the same length. We provide this length at label `LENGTH`.
- Array C has already been reserved sufficient memory at label `C`.

Relevant labels:

- `A` which holds the address of array A.
- `B` which holds the address of array B.
- `C` which holds the address of C, your solution array.
- `LENGTH` which holds the size of the arrays.

Say A is [-4, 6, 0] and B is [1, 5, 2]. After running your program, C should equal [1, 0, 1].

- C[0] should be 1 because A[2] = 0 $\leq$ B[2] = 2
- C[1] should be 0 because A[1] = 6 $\nleq$ B[1] = 5
- C[2] should be 1 because A[0] = -4 $\leq$ B[0] = 1

You'll notice that each array's label stores a value written in hexadecimal. Perhaps these values should be interpreted as memory addresses. Recall how we agreed to represent arrays in memory.

How would you load a specific array value in assembly? If the first value of `A` is located at address x3200, think about where the second or third value of `A` could be located.

Implement your assembly code in `buildMinArray.asm`

Suggested Pseudocode:

```
int A[] = {-4, 6, 0};
int B[] = {1, 5, 2};
int C[3];
int length = 3;

int i = 0;
while (i < length) {
    if (A[i] <= B[i]) {
        C[length - i - 1] = 1;
    }
    else {
        C[length - i - 1] = 0;
    }
    i++;
}
```

3.4 Part 3: Octal String to Int

Given an octal number encoded as a string, translate it to its numerical value. The value stored at `RESULTADDR` represents another different address. Store your numerical value result at this different address.

Assumptions:

- '0' ≤ octalString[i] ≤ '7'
- The provided octal string will be nonnegative.
- We do not have to worry about overflow.

Relevant labels:

- `OCTALSTRING` which holds the starting address of the octal string.
- `LENGTH` which holds the length of the octal string.
- `RESULTADDR` which holds the **address** of where you will store your numerical value result.
- `ASCII` which holds the value -48.

Say the string at `OCTALSTRING` was "2023". After running your program, mem[mem[`RESULTADDR`]] should equal 1043 (base 10). If `RESULTADDR` = x4000, then this means the value all the way at memory address x4000 should be changed to 1096. **The value at `RESULTADDR` should be unchanged.**

Recall how we interpret characters using ASCII (ASCII table listed in Appendix). You might find the `ASCII` label useful.

Implement your assembly code in `octalStringToInt.asm`

Suggested Pseudocode:

```
String octalString = "2023";
int length = 4;
int value = 0;
int i = 0;
while (i < length) {
    int leftShifts = 3;
    while (leftShifts > 0) {
        value += value;
        leftShifts--;
    }
    int digit = octalString[i] - 48;
    value += digit;
    i++;
}
mem[mem[RESULTADDR]] = value;
```

3.5 Part 4: Palindrome

Given a string, calculate if it's a palindrome. ANSWERADDR should contain another address. At this address, store true (represented as 1) if the string is palindrome and false (represented as 0) if it is not.

Assumptions:

- 'a' $\leq$ str[i] $\leq$ 'z'
- We still consider empty strings as palindromes.
- **We agree that a 1 in memory represents true and a 0 represents false.**

Relevant labels:

- STRING which holds the address of the string that you will evaluate.
- ANSWERADDR which holds the **address** of where you should store the boolean result.

Say the provided string is "racecar". After running your program, mem[mem[ANSWERADDR]] should equal true.

Notice we are only given the starting address our string. How do we know which address the string ends? Let's agree that if we ever read a numerical 0 value, then the string ends. If you take a look at an ASCII table (listed in Appendix), you can see the numerical value 0 is interpreted as NULL, which by convention denotes the end of a string.

NOTE: In the pseudocode below, '\0' means the numerical value 0. It is different from '0', which is the numerical value 48.

Implement your assembly code in palindrome.asm

Suggested Pseudocode:

```
String str = "racecar";
boolean isPalindrome = true
int length = 0;
while (str[length] != '\0') {
    length++;
}
int left = 0
int right = length - 1
while(left < right) {
    if (str[left] != str[right]) {
        isPalindrome = false;
        break;
    }
    left++;
    right--;
}
mem[mem[ANSWERADDR]] = isPalindrome;
```

4 Deliverables

Turn in the following files on Gradescope:

1. `modulus.asm`
2. `buildMinArray.asm`
3. `octalStringToInt.asm`
4. `palindrome.asm`

The Gradescope autograder should account for all test cases. Given that you have submitted your own work, on time, you can assume that the autograder will accurately reflect your grade.

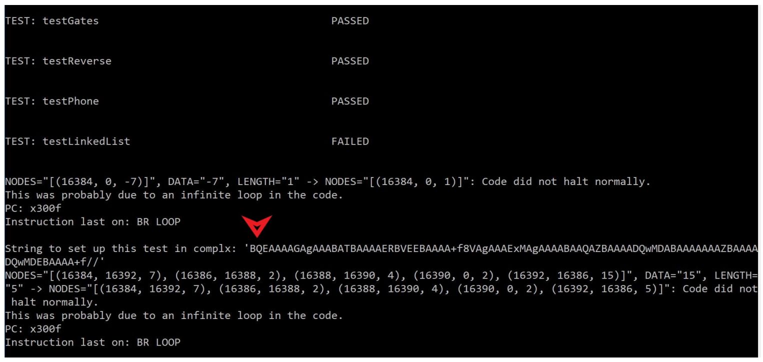
Note: Try to start homeworks early. It will be easier to get help if you get stuck, and last minute turn-ins will result in long queue times for grading on Gradescope.

5 Running the Autograder and Debugging LC-3 Assembly

When you turn in your files on Gradescope for the first time, you may 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 Complx feature called "replay strings".

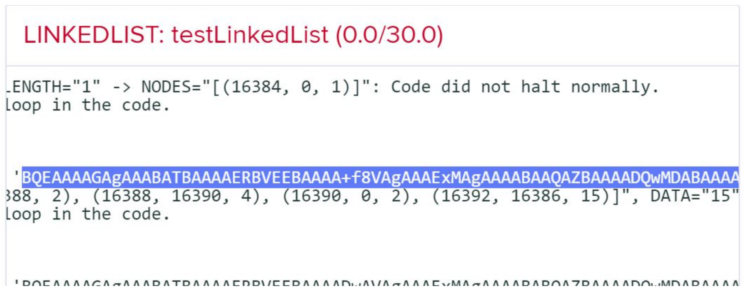
1. First off, we can get these replay strings in two places: the local grader, or off of Gradescope. To run the local grader:
   - Mac/Linux Users:
     (a) Navigate to the directory your homework is in (**in your terminal on your host machine, not in the Docker container via your browser**)
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) In Git Bash (or Docker Quickstart Terminal for legacy Docker installations), navigate to the directory your homework is in
     (b) Run `chmod +x grade.sh`
     (c) 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+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDABAAAAAAAZBAAAA
DQwMDEBAAAA+f//'
NODES="[(16384, 16392, 7), (16386, 16388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 15)]", DATA="15", LENGTH=
"5" -> 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
```

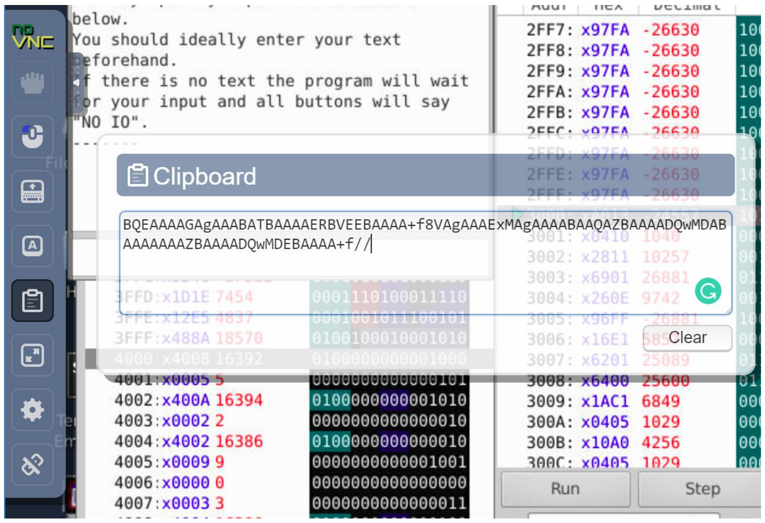
   Copy the string, starting with the leading 'B' and ending with the final backslash. Do not include the quotation marks.



LINKEDLIST: testLinkedList (0.0/30.0)

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


 'BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDABAAAA
388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 15)]", DATA="15"
loop in the code.
```

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



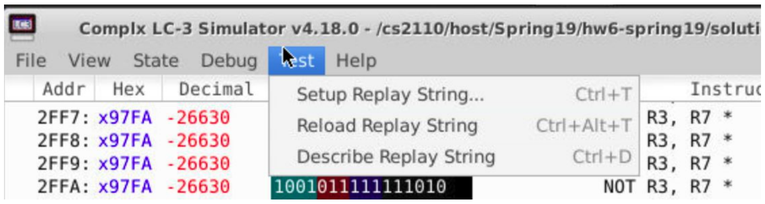
```
below.
You should ideally enter your text
beforehand.
If there is no text the program will wait
for your input and all buttons will say
"NO IO".
```

```
BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDAB
AAAAAAAZBAAAADQwMDEBAAAA+f//
```

```
3FFD:x1D1E 7454   0001110100011110
3FFE:x12E5 4837   0001001011100101
3FFF:x488A 18570  0100100010001010
4000:x4008 16392  0100000000001000
4001:x0005 5      0000000000000101
4002:x400A 16394  0100000000001010
4003:x0002 2      0000000000000010
4004:x4002 16386  0100000000000010
4005:x0009 9      0000000000001001
4006:x0000 0      0000000000000000
4007:x0003 3      0000000000000011
```

```
2FF7: x97FA -26630
2FF8: x97FA -26630
2FF9: x97FA -26630
2FFA: x97FA -26630
2FFB: x97FA -26630
2FFC: x97FA -26630
2FFD: x97FA -26630
2FFE: x97FA -26630
2FFF: x97FA -26630
3001: x0410 1040
3002: x2811 10257
3003: x6901 26881
3004: x260E 9742
3005: x96FF -26881
3006: x16E1 5857
3007: x6201 25089
3008: x6400 25600
3009: x1AC1 6849
300A: x0405 1029
300B: x10A0 4256
300C: x0405 1029
```

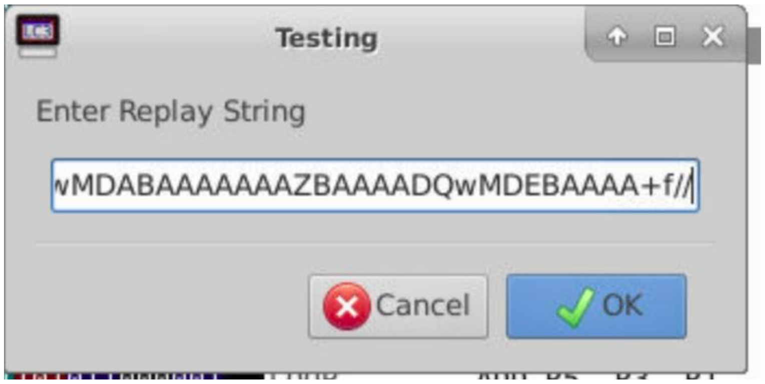
3. Next, go to the Test Tab and click Setup Replay String



Complx LC-3 Simulator v4.18.0 - /cs2110/host/Spring19/hw6-spring19/soluti
File
View
State
Debug
Test
Help
Setup Replay String... Ctrl+T
Reload Replay String Ctrl+Alt+T
Describe Replay String Ctrl+D

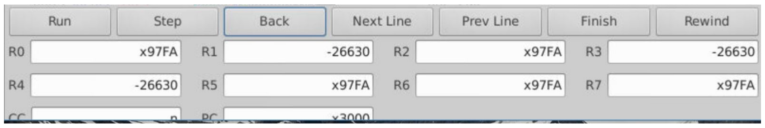
```
Addr  Hex    Decimal
2FF7: x97FA  -26630                    R3, R7 *
2FF8: x97FA  -26630                    R3, R7 *
2FF9: x97FA  -26630                    R3, R7 *
2FFA: x97FA  -26630  1001011111111010  NOT R3, R7 *
```

4. Now, paste your tester string in the box!



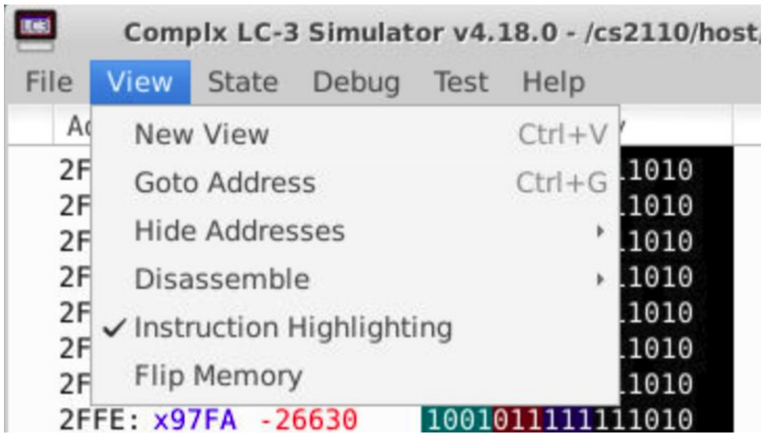
```
wMDABAAAAAAAZBAAAADQwMDEBAAAA+f//
```

5. 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.



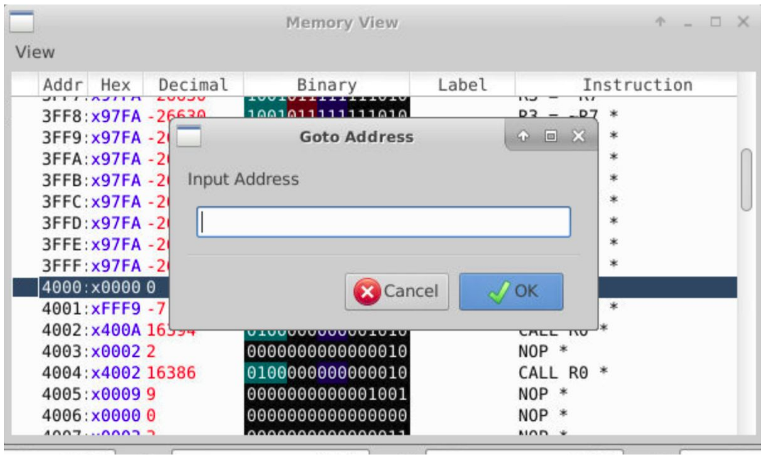
R0	x97FA	R1	-26630	R2	x97FA	R3	-26630
R4	-26630	R5	x97FA	R6	x97FA	R7	x97FA
CC	n	PC	x3000				

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



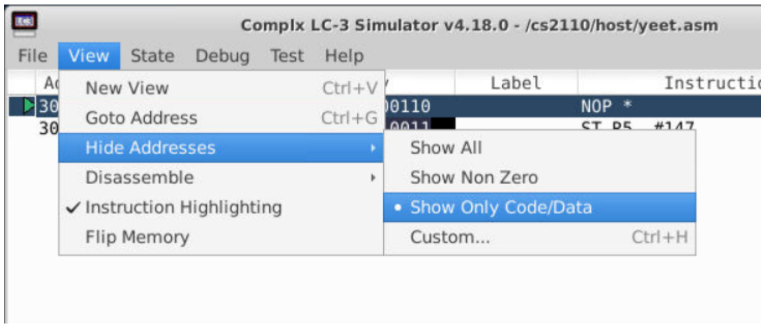
```
2FFE: x97FA -26630 1001011111111010
```

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



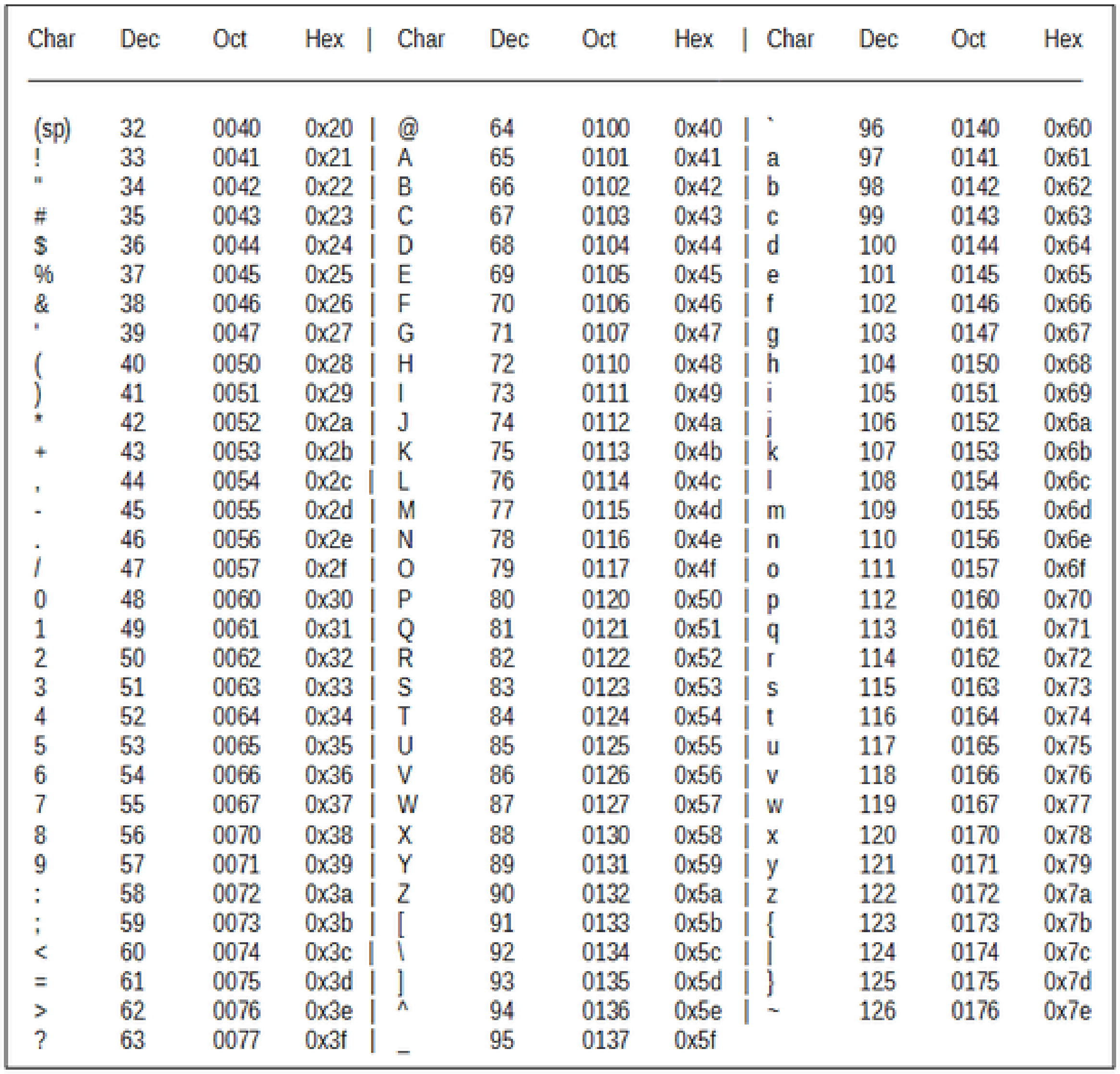
```
Addr  Hex   Decimal  Binary            Label  Instruction
3FF8:x97FA -26630   1001011111111010          R3 = ~R7 *
3FF9:x97FA -2
3FFA:x97FA -2
3FFB:x97FA -2
3FFC:x97FA -2
3FFD:x97FA -2
3FFE:x97FA -2
3FFF:x97FA -2
4000:x0000 0
4001:xFFF9 -7
4002:x400A 16394                              CALL R0 *
4003:x0002 2        0000000000000010          NOP *
4004:x4002 16386    0100000000000010          CALL R0 *
4005:x0009 9        0000000000001001          NOP *
4006:x0000 0        0000000000000000          NOP *
```

8. One final tip: to automatically shrink your view down to only those parts of memory that you care about (instructions and data), you can use View Tab → Hide Addresses → Show Only Code/Data.



6 Appendix

6.1 Appendix A: ASCII Table



Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

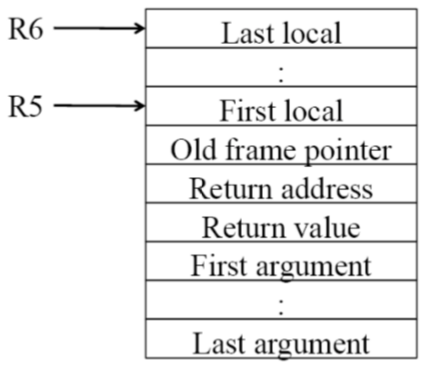
Figure 1: ASCII Table — Very Cool and Useful!

6.2 Appendix B: LC-3 Instruction Set Architecture

ADD	0001	DR	SR1	0	00	SR2
ADD	0001	DR	SR1	1	imm5	
AND	0101	DR	SR1	0	00	SR2
AND	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD	0010	DR	PCoffset9			
LDI	1010	DR	PCoffset9			
LDR	0110	DR	BaseR	offset6		
LEA	1110	DR	PCoffset9			
NOT	1001	DR	SR	111111		
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			

Trap Vector	***Assembler Name***
x20	GETC
x21	OUT
x22	PUTS
x23	IN
x25	HALT

Device Register	***Address***
Keybd Status Reg	xFE00
Keybd Data Reg	xFE02
Display Status Reg	xFE04
Display Data Reg	xFE06



6.3 Appendix C: LC-3 Assembly Programming Requirements and Tips

1. Your code must assemble with **NO WARNINGS OR ERRORS**. To assemble your program, open the file with Complx. It will complain if there are any issues. **If your code does not assemble you WILL get a zero for that file.**

2. **Comment your code!** This is especially important in assembly, because it's much harder to interpret what is happening later, and you'll be glad you left yourself notes on what certain instructions are contributing to the code. Comment things like what registers are being used for and what less intuitive lines of code are actually doing. To comment code in LC-3 assembly just type a semicolon (;), and the rest of that line will be a comment.

3. Avoid stating the obvious in your comments, it doesn't help in understanding what the code is doing.

   Good Comment

   ```
   ADD R3, R3, -1          ; counter--
   BRp LOOP                ; if counter == 0 don't loop again
   ```

   Bad Comment

   ```
   ADD R3, R3, -1          ; Decrement R3
   BRp LOOP                ; Branch to LOOP if positive
   ```

4. **DO NOT assume that ANYTHING in the LC-3 is already zero.** Treat the machine as if your program was loaded into a machine with random values stored in the memory and register file.

5. Following from 3, you can load the file with randomized memory by selecting "File" ⇒ "Advanced Load" and selecting randomized registers/memory.

6. Do NOT execute any data as if it were an instruction (meaning you should put `.fills` after `HALT` or `RET`). All your program does is interpret the values RAM as instructions until it reaches HALT. If you use .fill before your program HALTS, your program might interpret what you filled as an instruction and try to execute it!

7. Do not add any comments beginning with `@plugin` or change any comments of this kind.

8. **Test your assembly.** Don't just assume it works and turn it in.

9. When translating pseudocode into assembly, don't skip over the closing brackets! Even though they're only one character long, perhaps they also might need to be translated into assembly...

7 Rules and Regulations

7.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. As such, please start assignments early, and ask for help early. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
2. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

7.2 Submission Conventions

1. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends. You must submit all files listed in the **Deliverables** section individually to Gradescope as separate files.

7.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

7.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

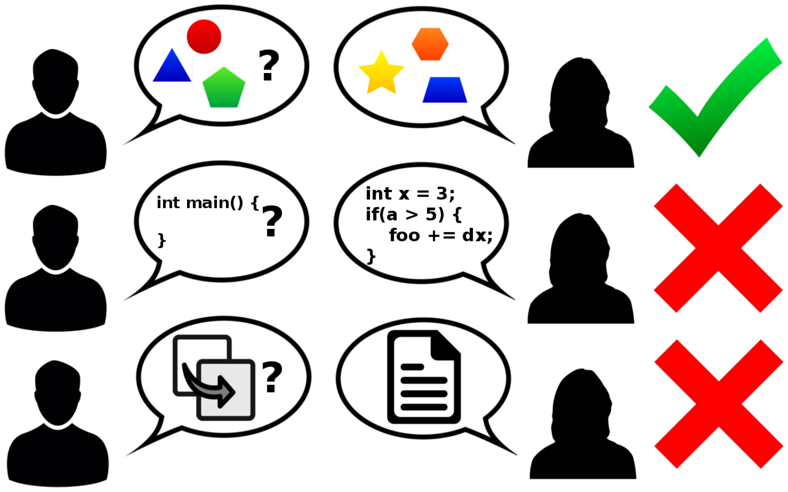
Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

7.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.


?
int main() {
}
?
int x = 3;
if(a > 5) {
    foo += dx;
}
?