CS 2110 Homework 5
Intro to Assembly

Avaneesh Naren, Allison Fain, Alice Zeng, Irene Feijoo

Version 1.0

Contents

1 TLDR

1.1 Debugging Assembly

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

1.2 Summation

Given a value x, calculate the sum $1 + 2 + ... + x - 1 + x$. Store the final sum at the label "ANSWER". The pseudocode can be found in section 3.2.

Below is a list of what we have provided you for this method.

- label named "X" where the value will be stored
- label named "ANSWER" where you will store your final sum

Please note that since x is a label, you will need to load the label to retrieve the data. Additionally, if the value of x is less than or equal to 0, you should store 0 at the "ANSWER" label.

1.3 Build Minimum Array

Given two arrays of the same length, construct a third array that contains the minimum value at each index. Example, $[2, -3, 6], [-2, 5, 2] \rightarrow [-2, -3, 2]$. The pseudocode can be found in section 3.3.

Below is a list of what we have provided you for this method.

- label named "A" that holds the address of the first array
- label named "B" that holds the address of the second array
- label named "C" that will hold the address of your array that contains the minimum values
- label named "LEN" that holds the size of the arrays

1.4 Binary String to Int

Given a 2s-complement binary number, convert it to a decimal integer and store it at the address of the label "RESULTIDX". The pseudocode can be found in section 3.4

Below is a list of what we have provided you for this method.

- label named "BINARYSTRING" that holds the binary string
- label named "LENGTH" that holds the length of BINARYSTRING
- label named "RESULTIDX" that will hold the decimal integer of BINARYSTRING
- label named "ASCII" that holds -48

1.5 Five Character Strings

Given a string sentence, calculate the number of 5 character strings within the sentence. Special characters do not have to be treated differently. For instance, strings like “can't” and “cans,” are both considered 5 character strings. The pseudocode can be found in section 3.5.

Below is a list of what we have provided you for this method.

- the ASCII value of the space character
- label named “STRING” that holds the address of the string that you will evaluate
- label named “ANSWER” that will hold the number of five character strings that you detect

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.

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

1. `summation.asm`
2. `buildMinArray.asm`
3. `binaryStringToInt.asm`
4. `fiveCharacterStrings.asm`

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

- The algorithms presented for each operation are not meant to be the most 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: Summation

To start you off with this homework, we are implementing a function that takes an integer x, and calculates the summation of x. Store the result of the operation at the label `ANSWER`. Argument ”x” is stored in a label, and you will need to load it from there to perform this operation. Similarly, “ANSWER” reserved 1 memory space for you to store the summation of x. Note that if ”x” is less than or equal to 0, return 0. Implement your assembly code in `summation.asm`.

For example, summation of 4 is 4 + 3 + 2 + 1 = 10.

Suggested Pseudocode:

```
int x = 6; (sample integer)
int sum = 0;
while (x > 0) {
    sum += x;
    x--;
}
mem[ANSWER] = sum;
```

Given x = 6, the above code should store 21 at label “ANSWER”, since the given x will yield the summation 6 + 5 + 4 + 3 + 2 + 1 = 21.

3.3 Part 2: Build Minimum Array

For the second assembly program, you are given two integer arrays A and B of the same length. We want you to fill in a third array C of the same length where the i^{th} element of C is the minimum of the i^{th} element in A and the i^{th} element of B.

For instance, if A = [-4, 2, 6] and B = [4, 7, -2], then C[i] = min(A[i], B[i]). Hence, C = [-4, 2, -2].

The label `LEN` will contain the **size** of arrays A and B. The labels `A`, `B`, and `C` will contain the addresses at which the arrays A, B, and C (respectively) **begin** in memory.

Your resulting array should be stored in memory, beginning at the address stored in label `C`.

Implement your assembly code in `buildMinArray.asm`

NOTE: Please do not change the names of any of the labels or the values stored in them as they may cause the autograder to misbehave. The sample values are okay to change though since the autograder will be running multiple test cases with different init-values. So if you want to debug on Complx, feel free to change those sample values!

Suggested Pseudocode:

```
int A[] = {-4, 2, 6}; (sample array)
int B[] = {4, 7, -2}; (sample array)
int C[3]; (sample array)
int length = 3; (sample length of above arrays)

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

3.4 Part 3: Binary String to Int

For the third assembly program, you are given a twos-complement binary string. We want you to convert the twos-complement binary string into an integer value. Please note that we give you a label that contains the ASCII value of 0 to help you with the conversion.

For instance,
"11111100" should convert to a -4
"00100001" should convert to a 33

Your converted value should be stored in memory at the address stored in label `RESULTIDX`.

Implement your assembly code in `BinaryStringToInt.asm`

Suggested Pseudocode:

```
String binaryString = "00010101"; (sample binary string)
int length = 8; (sample length of the above binary string)
int base = 1;
int value = 0;
int i = length - 1;
while (i >= 0) {
    int x = binaryString[i] - 48;
    if (x == 1) {
        if (i == 0) {
            value -= base;
        } else {
            value += base;
        }
    }
    base += base;
    i--;
}
mem[mem[RESULTIDX]] = value;
```

Given a binary string("00010101"), its length(8), and a location to store (x4000), this code should store 21 in memory address x4000.

3.5 Part 4: Five Character Strings in a paragraph

In the final part of this assignment, we want you to find the number of five letter words in a given `null terminated` string.

The label `STRING` will contain the **address** of the first character in our string. Keep in mind that a string is just an array of characters that end with a null-termination character ('\0'). For your program, you are given one constant to help you called `SPACE`, which is the negative value of 'space' in ASCII (-32).

Store the result of the operation at the label `ANSWER`.

IMPORTANT

- Special characters do not have to be treated differently. For instance, strings like "can't" and "sure," are considered 5 character strings.

Implement your assembly code in `fiveCharacterStrings.asm`

Assume that the last character of every string is a space.

NOTE:

- 0 is the same as '\0'
- 0 is different from '0'
- " " in ASCII is 32 (the space character)

Suggested Pseudocode:

```
int count = 0; (keep count of number of 5-letter words)
int chars = 0; (keep track of length of each word)
int i = 0; (indexer into each word)
String str = "I really love CS 2110 and using wires and code!"; (sample string)
while (str[i] != '\0') {
    if (str[i] != ' ') {
        chars++;
    }
    else {
        if (chars == 5) {
            count++;
        }
        chars = 0;
    }
    i++;
}
mem[ANSWER] = count;
```

The above code should store 3 at the label `ANSWER` from the 5 character strings: "using", "wires", and "code!".

4 Deliverables

Turn in the following files on Gradescope:

1. `summation.asm`
2. `buildMinArray.asm`
3. `binaryStringToInt.asm`
4. `fiveCharacterStrings.asm`

Note: Please do not wait until the last minute to run/test your homework. Last minute turn-ins will result in long queue times for grading on Gradescope. You have been warned.

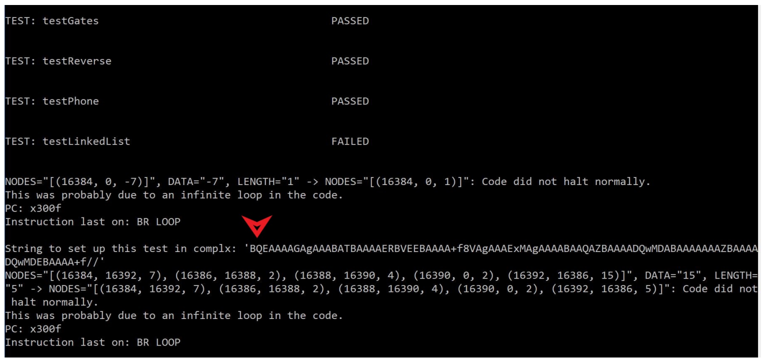
5 Running the Autograder and Debugging LC-3 Assembly

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:

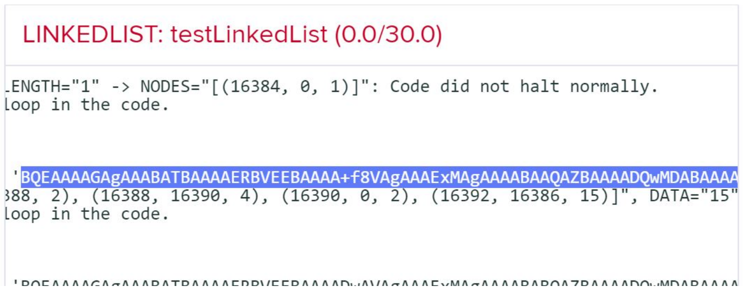
   - Mac/Linux Users:
     (a) Navigate to the directory your homework is in (**in your terminal on your host machine, not in the Docker container via your browser**)
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) In Git Bash (or Docker Quickstart 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
```

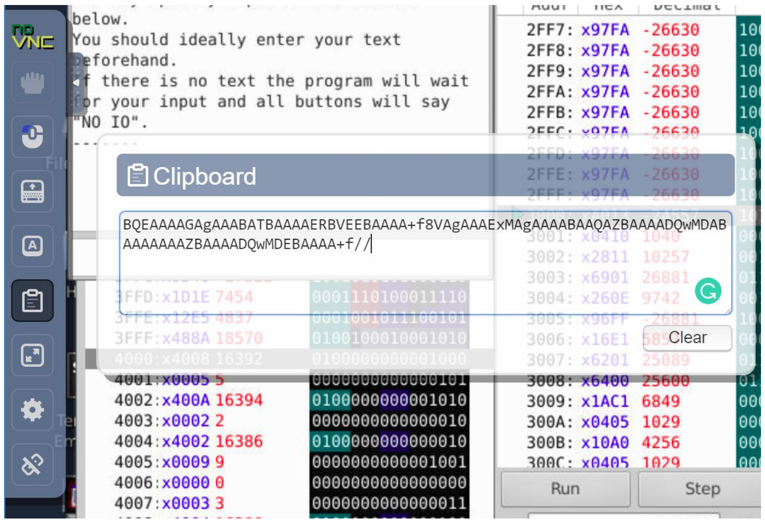
   Copy the string, starting with the leading 'B' and ending with the final backslash. Do not include the quotation marks.

   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.



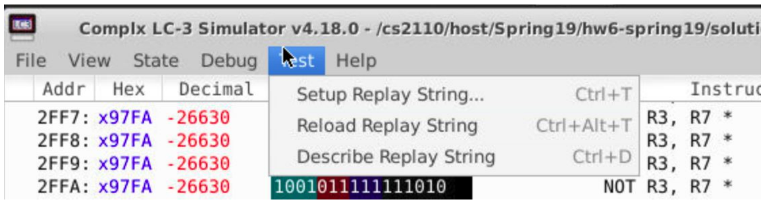
```
below.
You should ideally enter your text
beforehand.
If there is no text the program will wait
for your input and all buttons will say
"NO IO".
```

```
BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDAB
AAAAAAAAZBAAAADQwMDEBAAAA+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 585
3007: x6201 25089
3008: x6400 25600
3009: x1AC1 6849
300A: x0405 1029
300B: x10A0 4256
300C: x0405 1029
```

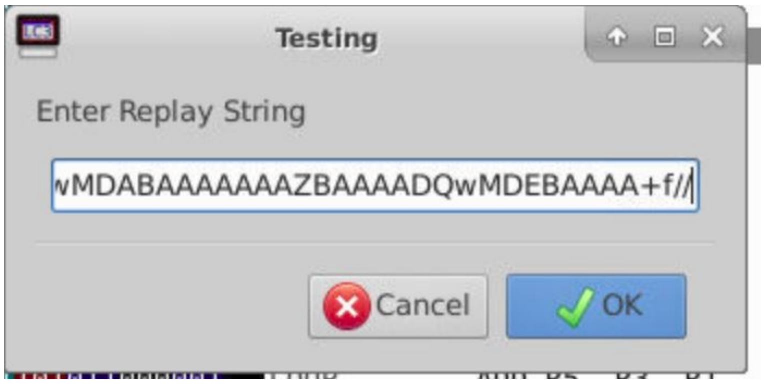
3. Next, go to the Test Tab and click Setup Replay String



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

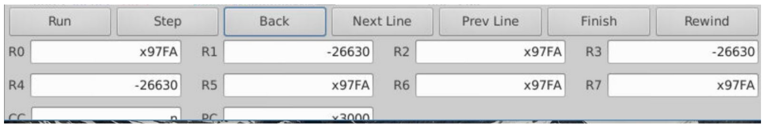
```
Addr   Hex    Decimal                          Instruc
2FF7: x97FA  -26630                       R3, R7 *
2FF8: x97FA  -26630                       R3, R7 *
2FF9: x97FA  -26630                       R3, R7 *
2FFA: x97FA  -26630  1001011111111010  NOT R3, R7 *
```

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



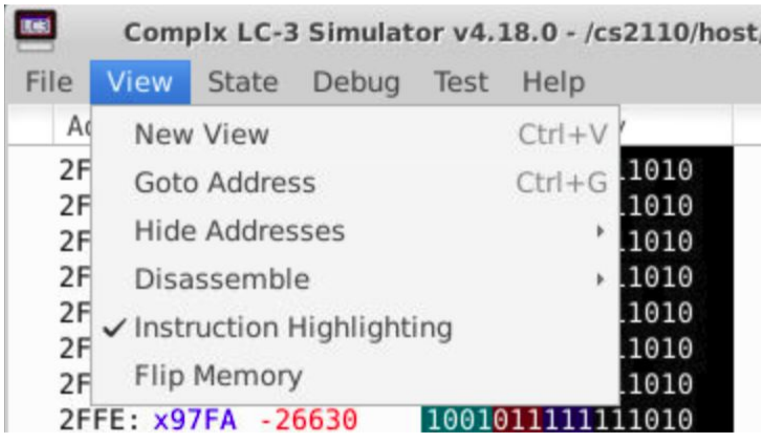
```
wMDABAAAAAAAZBAAAADQwMDEBAAAA+f//
```

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.



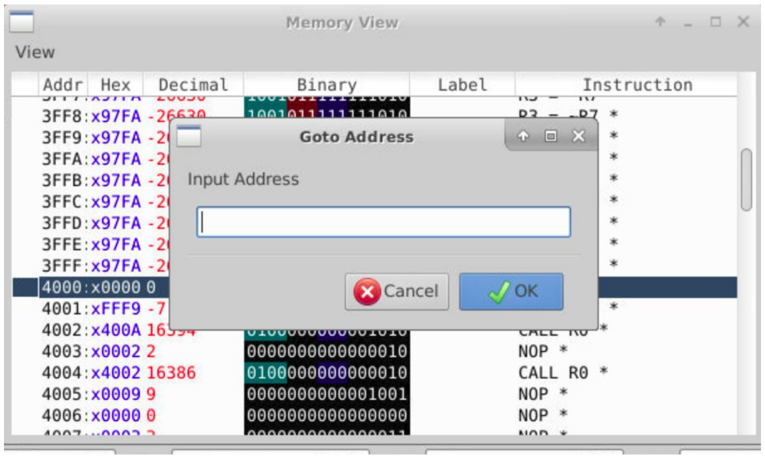
R0	x97FA	R1	-26630	R2	x97FA	R3	-26630
R4	-26630	R5	x97FA	R6	x97FA	R7	x97FA
CC	n	PC	x3000				

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



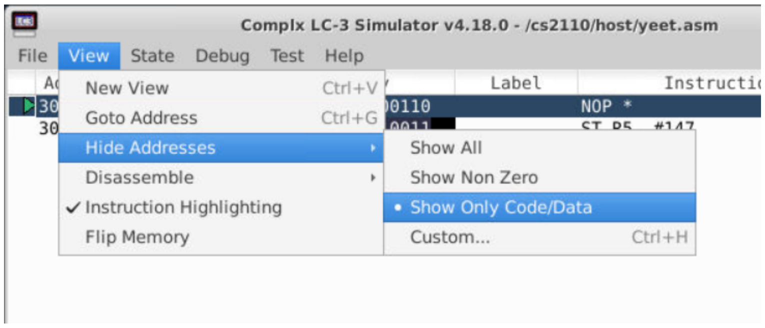
```
2FFE: x97FA -26630 1001011111111010
```

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



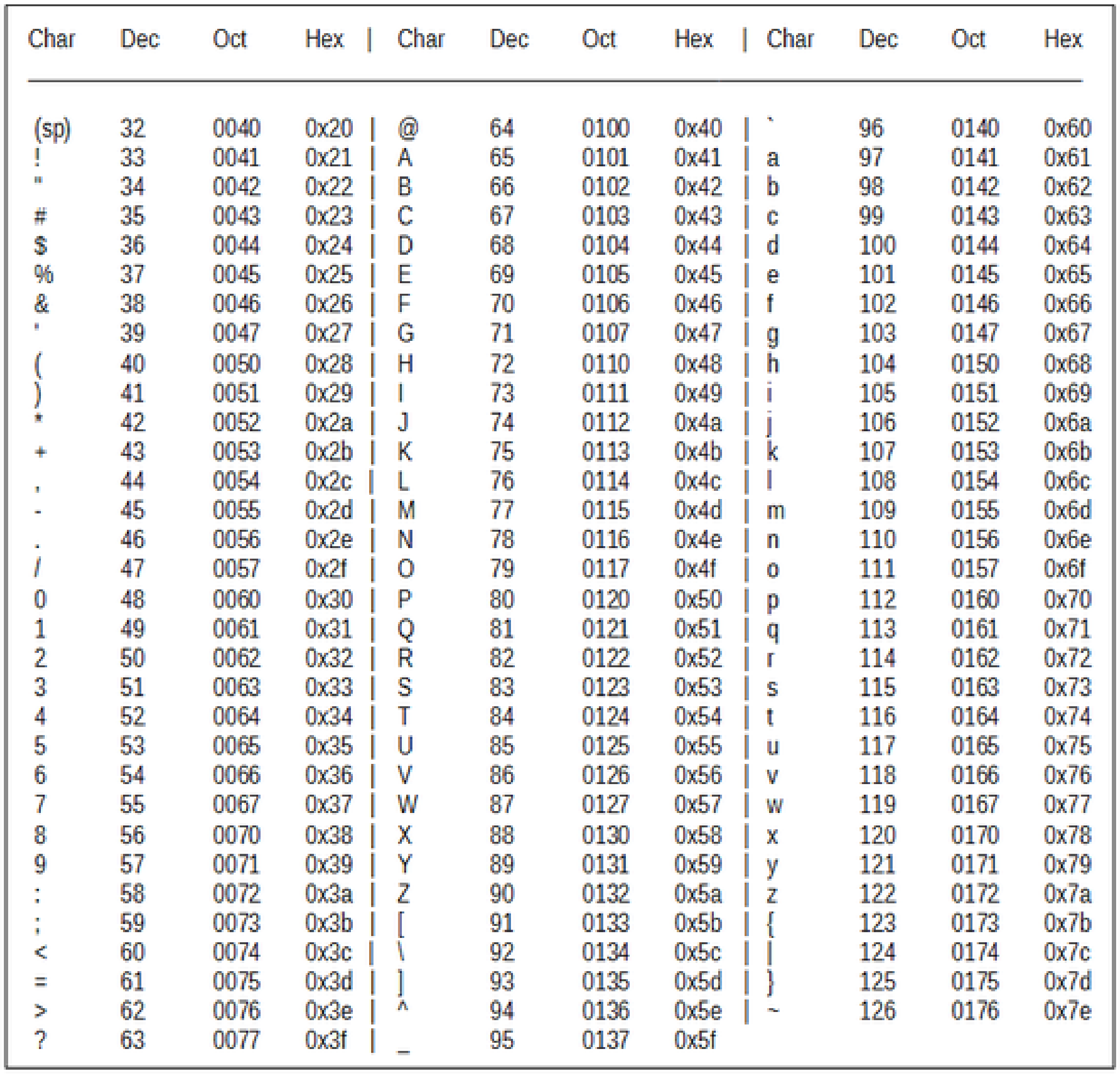
```
Addr Hex    Decimal  Binary            Label  Instruction
3FF8:x97FA -26630    1001011111111010         R3 = ~R7 *
3FF9:x97FA -2                                 *
3FFA:x97FA -2                                 *
3FFB:x97FA -2                                 *
3FFC:x97FA -2                                 *
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				

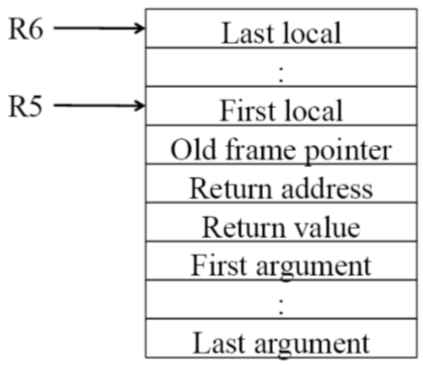
Figure 1: ASCII Table — Very Cool and Useful!

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`).

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.

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.

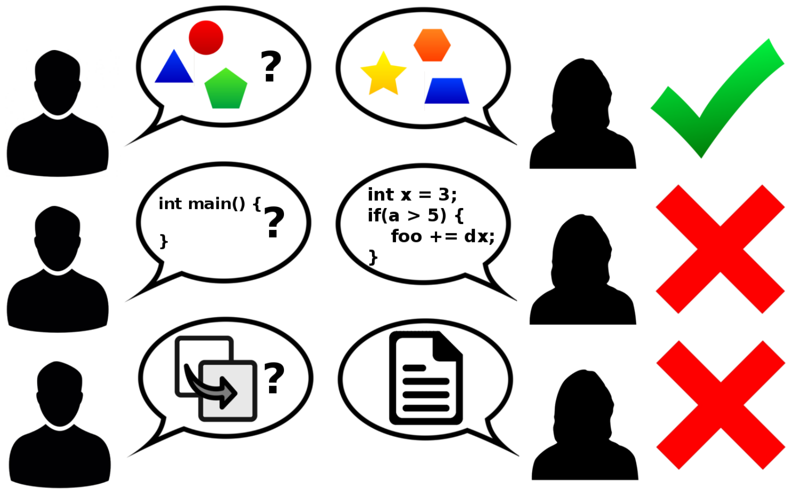
Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to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