CS 2110 Homework 5
Intro to Assembly

Justin Hu, Varnika Budati, Charlie Gunn, Siddhant Singh

Spring 2021

Contents

1 Overview

1.1 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.

1.2 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. `gcd.asm`
2. `merge.asm`
3. `studlyCaps.asm`
4. `palindrome.asm`

1.3 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!

2 Detailed Instructions

2.1 Part 1: GCD

To start you off with this homework, we are implementing the GCD function! Store the result of the operation in the label `ANSWER`. Arguments A and B are stored in memory, and you will load them from there to perform this operation. Implement your assembly code in `gcd.asm`.

Suggested Pseudocode:

```
ANSWER = 0;
while (a != b) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
ANSWER = a;
```

2.2 Part 2: Array Merge

The second assembly function is to implement the "merge" part of merge-sort into an array in memory. You will be merging two pre-sorted input arrays into a third array, so that the third array is also sorted. The third array will have enough space for the final answer. Use the pseudocode to help plan out your assembly and make sure you are sorting it properly! Implement your assembly code in `merge.asm`.

Suggested Pseudocode:

```
x = 0; // first index of ARR_X
y = 0; // first index of ARR_Y
z = 0; // first index of ARR_RES
while (x < LENGTH_X && y < LENGTH_Y) {
    if (ARR_X[x] <= ARR_Y[y]) {
        ARR_RES[z] = ARR_X[x];
        z++:
        x++;
    } else {
        ARR_RES[z] = ARR_Y[y];
        z++;
        y++;
    }
}
while (x < LENGTH_X) {
    ARR_RES[z] = ARR_X[x];
    z++;
    x++;
}
while (y < LENGTH_Y) {
    ARR_RES[z] = ARR_Y[y];
    z++;
    y++;
}
// the final merged array should be in ARR_RES
```

2.3 Part 3: Studly Caps

The third assembly function is to turn a **null-terminated** string into a string with studly caps. Studly caps, also known as alternating caps, sticky caps, or occasionally "SpongeBob case", consists of alternating the letters in a string between lowercase and uppercase. In this function, the first letter is expected to be lowercase and the subsequent letters alternate between uppercase and lowercase depending on the index. The index starts at 0, if the index is even, the character should be lowercase. If the index is odd, the character is expected to be uppercase.

The label `STRING` will contain the **address** of the first character of the string to be converted. Convert the string in-place, so that the result string is also stored at the same label `STRING`. Remember that strings are just arrays of consecutive characters. You are given four constants to use in your program, `LOWERA` which is the value of the ASCII character 'a', `LOWERZ` which is the value of 'z', `UPPERA` which is the value of 'A', and `UPPERZ` which is the value of 'Z'. Implement your assembly code in `studlyCaps.asm`

Assume that the strings are random: they can contain characters, numbers, spaces and symbols.

To convert a character between lowercase and uppercase, **refer to the ASCII table** and remember that each of these characters are represented by a word (16-bits) in the LC-3's memory. This is a **null-terminated** string, meaning that a 0 will be stored immediately after the final character in memory!

NOTE:

- 0 is the same as '\0'
- 0 is different from '0'

Suggested Pseudocode:

```
string = "TWENTY 1 ten"
i = 0;
while (string[i] != 0) {
    if (i % 2 == 0) {
        // should be lowercase
        if ('A' <= string[i] <= 'Z') {
            string[i] = string[i] | 32;
        }
    } else {
        // should be uppercase
        if ('a' <= string[i] <= 'z') {
            string[i] = string[i] & ~32;
        }
    }
    i++;
}
```

The result here is `tWeNtY 1 TeN`.

2.4 Part 4: Palindromes

For the final problem, your goal is to determine whether a `null-terminated` string is a palindrome. A palindrome is a word, phrase or a sequence of numbers that reads the same backward as forward. For instance, "racecar" is a palindrome because it is "racecar" when read both backward and forward.

The label `STRING` will contain the **address** of the first character of the string to be checked. Remember that this string is `null-terminated`. The result will be stored at the label `RESULT`. If the given string is a palindrome, the result should be set to `1`. Else, it should be set to `0`. Implement your assembly code in `palindrome.asm`

Assume every alphabetic character in the string is in lowercase. The string can contain numbers.

NOTE:

- 0 is the same as '\0'
- 0 is different from '0'

Suggested Pseudocode:

```
string = "racecar";
len = 0;

// to find the length of the string
while (string[len] != '\0') {
    len = len + 1;
}

// to check whether the string is a palindrome
result = 1;
i = 0;
while (i < length) {
    if (string[i] != string[length - i - 1]) {
        result = 0;
        break;
    }
    i = i + 1;
}
```

3 Deliverables

Turn in the following files on Gradescope:

1. `gcd.asm`
2. `merge.asm`
3. `studlyCaps.asm`
4. `palindrome.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.

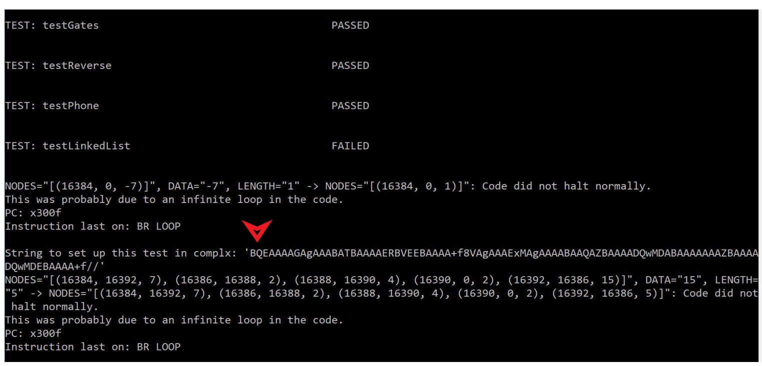
4 Running the Autograder and Debugging LC-3 Assembly

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:

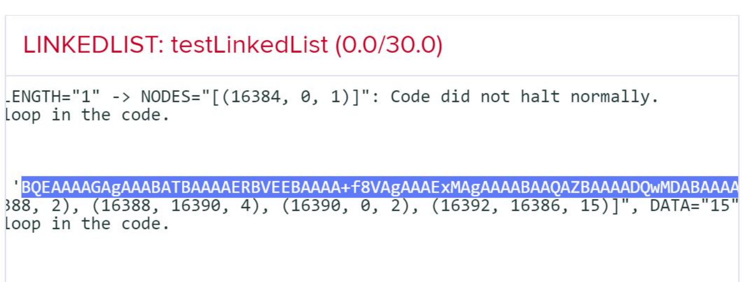
   - Mac/Linux Users:
     (a) Navigate to the directory your homework is in (**in your terminal on your host machine, not in the Docker container via your browser**)
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) In Git Bash (or Docker Quickstart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

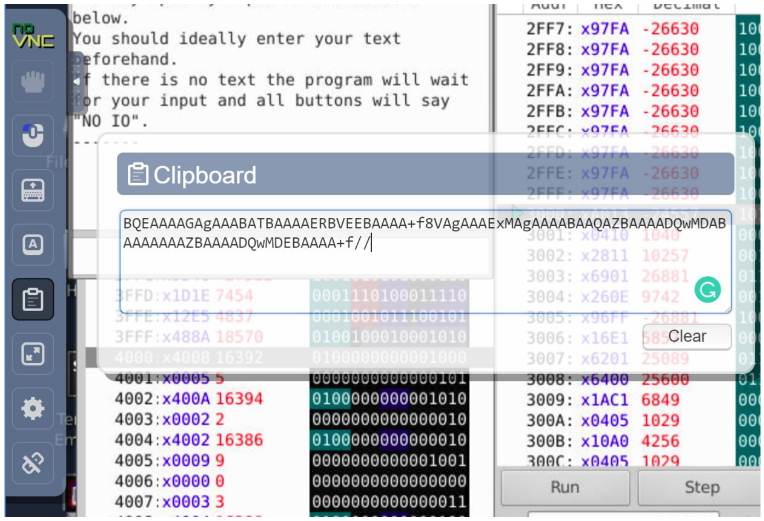


   Copy the string, starting with the leading ’B’ and ending with the final backslash. Do not include the quotation marks.

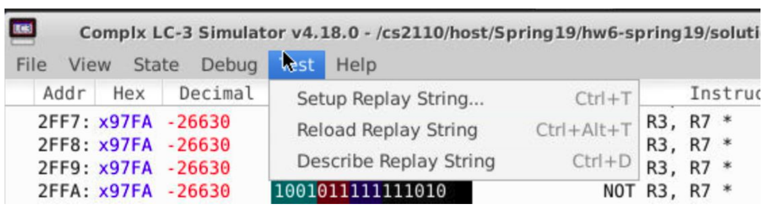
   Side Note: If you do not have Docker installed, you can still use the tester strings to debug your assembly code. In your Gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslash and again, don’t copy the quotations.



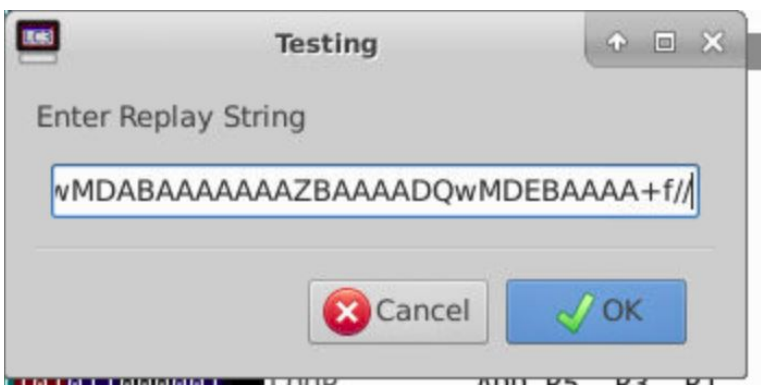
2. Secondly, navigate to the clipboard in your Docker image and paste in the string.



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

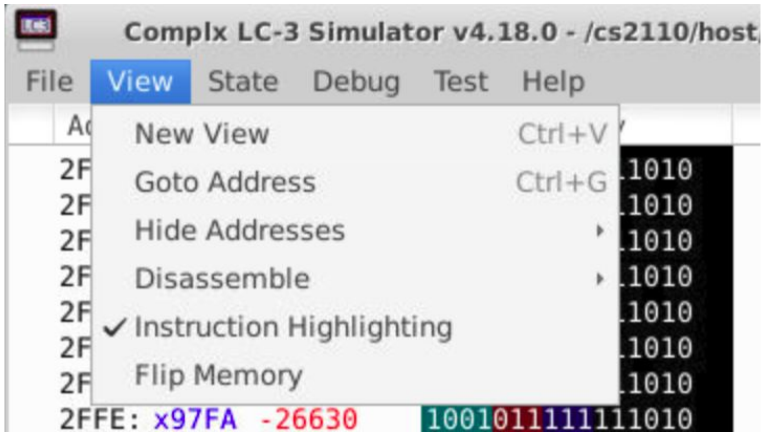


4. Now, paste your tester string in the box!

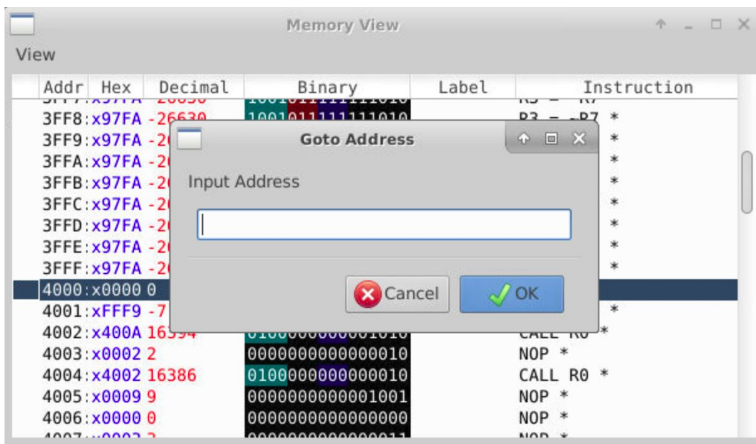


5. Now, Complx is set up with the test that you failed! The nicest part of Complx is the ability to step through each instruction and see how they change register values. To do so, click the step button. To change the number representation of the registers, double click inside the register box.

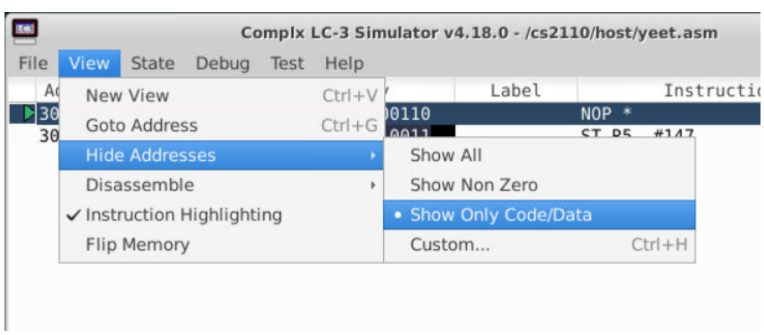
6. If you are interested in looking how your code changes different portions of memory, click the view tab and indicate 'New View'



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



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.



5 Appendix

5.1 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				

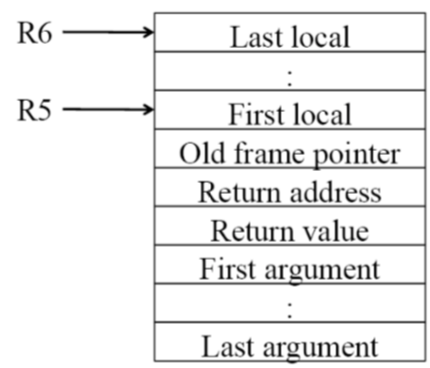
Figure 1: ASCII Table — Very Cool and Useful!

5.2 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



5.3 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.

6 Rules and Regulations

6.1 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.

6.2 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.

6.3 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.

6.4 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

6.5 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.