CS 2110 Homework 5
Intro to Assembly

Prabhav Gupta, Saloni Bedi, Justin Almas,
Richard So, Gilbert Mao, Rohan Bafna

Spring 2024

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. `fibonacci.asm`
2. `uppercaseInRange.asm`
3. `hexStringToInt.asm`
4. `palindromeWithSkips.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. Your code must assemble with **no warnings or errors** (LC3Tools 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 Notes

- The provided pseudocode is fully correct and will naturally account for all assumptions and edge cases. It might not be the most efficient solution, and this is OK.
- Make sure you conceptually understand what labels are and what using them really means behind the scenes. All 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.
- 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 LC3Tools if you're not sure if you're using the correct instruction.
- Debugging via LC3Tools 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 LC3Tools by entering an address in the `Jump To Location` input and pressing Enter.

2.2 Part 1: Fibonacci

Given N, calculate and store the first N Fibonacci numbers in an array starting at `RESULT`.
Assumptions:

- The first Fibonacci number is 0.
- N can be 0 or positive (it will not be negative).

Relevant labels:

- `N`: Holds the value of N.
- `RESULT`: Holds the base address of the array that will hold the Fibonacci numbers.

Say N = 5. After running your program, memory starting at `RESULT` should hold 0, 1, 1, 2, 3.
Implement your assembly code in `fibonacci.asm`

Suggested Pseudocode:

```
n = mem[N];
resAddr = mem[RESULT]

if (n == 1) {
   mem[resAddr] = 0;
} else if (n > 1) {
    mem[resAddr] = 0;
    mem[resAddr + 1] = 1;
    for (i = 2; i < n; i++) {
        x = mem[resAddr + i - 1];
        y = mem[resAddr + i - 2];
        mem[resAddr + i] = x + y;
    }
}
```

2.3 Part 2: Uppercase In Range

Given a string starting at the address in `STRING`, convert the characters that are in the range between `START` and `END` to uppercase.

Assumptions:

- Possible characters in the string: lowercase letters, uppercase letters, and spaces
- If a character in the range is not lowercase, leave it as it is.
- The range should include `START` and not include `END`. So, change all the characters to uppercase in range [`START`, `END`).
- `END` may be greater than the length of the string. If that is the case, apply the conversion to all of the characters in the string from `START` onwards.

Relevant labels:

- `STRING`: Holds the base address of the array containing the string.
- `LENGTH`: Holds the length of the string.
- `START`: Holds the index of the first character you want to uppercase.
- `END`: Holds the index following that of the last character you want to uppercase.
- `ASCII_A`: Holds the ASCII value of lowercase a.

Say the string is gOOglE, START is 1, and END is 4. After running your program, the string starting at the address in STRING should be gOOGlE.

Implement your assembly code in `uppercaseInRange.asm`

Suggested Pseudocode:

```
String str = "touppERcase";
int start = mem[START];
int end = mem[END];
int length = mem[LENGTH];
if (end > length) {
    end = length;
}

for (int x = start; x < end; x++) {
    if (str[x] >= 'a') {
        str[x] = str[x] - 32;
    }
}
```

2.4 Part 3: Hex String to Int

Given an hex 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' $\leq$ hexString[i] $\leq$ '9' and 'A' $\leq$ hexString[i] $\leq$ 'F'
- The provided hex string will be nonnegative.
- We do not have to worry about overflow.

Relevant labels:

- `HEXSTRING`: Holds the starting address of the hex string.
- `LENGTH`: Holds the length of the hex string. **This will always be 4.**
- `RESULTADDR`: Holds the **address** of where you will store your numerical value result.
- `ASCIIDIG`: Holds the value 48.
- `ASCIICHAR`: Holds the value 55.
- `SIXTYFIVE`: Holds the value 65.

Say the string at `HEXSTRING` was "211F". After running your program, mem[mem[`RESULTADDR`]] should equal 8479 (base 10). If mem[`RESULTADDR`] = x4000, then this means the value all the way at memory address x4000 should be changed to 8479. **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 `hexStringToInt.asm`

Suggested Pseudocode:

```
String hexString = "F1ED";
int length = 4;
int value = 0;
int i = 0;
while (i < length) {
    int leftShifts = mem[LENGTH];
    while (leftShifts > 0) {
        value += value;
        leftShifts--;
    }
    if (hexString[i] >= 65) {
        value += hexString[i] - 55;
    } else {
        value += hexString[i] - 48;
    }
    i++;
}
mem[mem[RESULTADDR]] = value;
```

2.5 Part 4: Palindrome With Skips

Given a string, calculate if it's a palindrome if you don't consider any instances of the character located at the address in `CHARADDR`. `ANSWERADDR` should contain another address. At this address, store true if the string is palindrome and false if it is not.

Assumptions:

- The characters in the string and the character to remove can be lowercase, uppercase, spaces, numbers, or special characters.
- We still consider empty strings as palindromes.
- **We agree that a 1 in memory represents true and a 0 represents false.**

Relevant labels:

- `CHARADDR`: Holds the address of the character you don't consider when deciding if the string is a palindrome.
- `STRING`: Holds the address of the string that you will evaluate.
- `ANSWERADDR`: Holds the **address** of where you should store the boolean result.

Say the provided string is "aibohphooobia" and the given character is "o". 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 `palindromeWithSkips.asm`

Suggested Pseudocode:

```
String str = "aibohphobia";
char skipChar = mem[mem[CHARADDR]];
int length = 0;
while (str[length] != '\0') {
    length++;
}

int left = 0;
int right = length - 1;
boolean isPalindrome = true;
while(left < right) {
    if (str[left] == skipChar) {
        left++;
        continue;
    }
    if (str[right] == skipChar) {
        right--;
        continue;
```

```
        }
        if (str[left] != str[right]) {
            isPalindrome = false;
            break;
        }

        left++;
        right--;
    }
    mem[mem[ANSWERADDR]] = isPalindrome;
```

3 Deliverables

Turn in the following files on Gradescope:

1. `fibonacci.asm`
2. `uppercaseInRange.asm`
3. `hexStringToInt.asm`
4. `palindromeWithSkips.asm`

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.

4 Running the Autograder and Debugging LC-3 Assembly

For this homework, there are a variety of test cases associated with each assembly file you implement. Additionally, there are two ways to grade your submission: locally, or through Gradescope.

To run the local grader:

1. Ensure that you have Docker running.
2. Navigate to the directory your homework is in.
3. If you are on MacOS or Linux, run the command `sudo chmod +x grade.sh`
4. Now run `./grade.sh` if you are on MacOS/Linux, or `.\grade.bat` on Windows.

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!

Each test case details exactly how we are testing your assembly code (e.g. writing to labels, writing strings, expecting a certain answer, etc.). Here is an example:

```
Writing word "touppercase" at MEM[MEM[WORD]]
Writing start argument '2' at MEM[START]
Writing end argument '9' at MEM[END]
Running student assembly...

--Outputs correct answer: Expected: toUPPERCAse, Got: touppercase
```

You can use this information to replicate such tests yourself locally on LC3Tools.

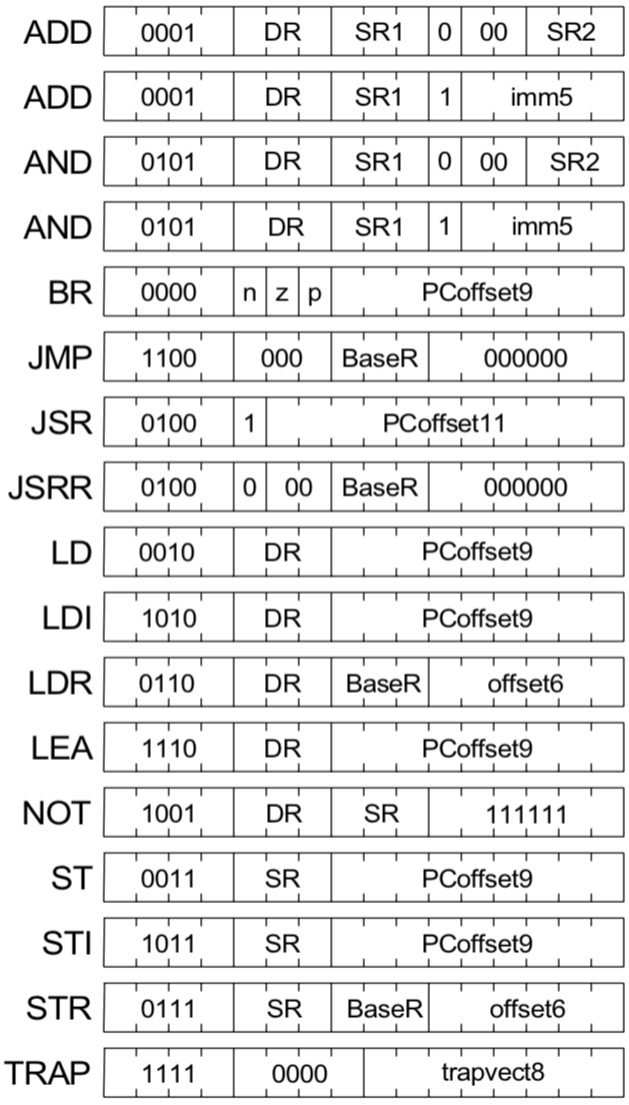
5 Appendix

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				

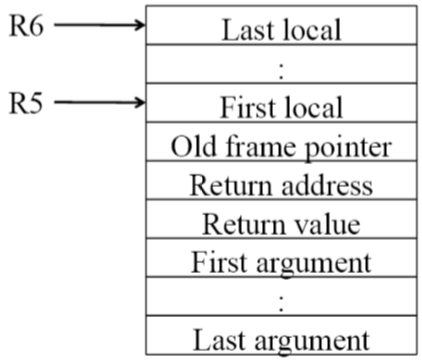
Figure 1: ASCII Table — Very Cool and Useful!

5.2 Appendix B: LC-3 Instruction Set Architecture



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 LC3Tools. 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.

3. **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. The autograder will do the same.

4. As you step through your assembled code in LC3Tools, registers or memory locations that have updated after one step will be highlighted to you. Use this to your advantage to figure out bugs in your assembly if you are failing a test case.

5. 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!

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

7. 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...

6 Rules and Regulations

1. Please read the assignment in its entirety before asking questions.
2. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
3. If you find any problems with the assignment, please report them to the TA team. Announcements will be posted if the assignment changes.
4. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency please reach out to your instructor and the head TAs **IN ADVANCE** of the due date with documentation (i.e. note from the dean, doctor's note, etc).
5. You are responsible for ensuring that what you turned in is what you meant to turn in. 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 Gradescope. Email submissions will not be accepted.
6. See the syllabus for information regarding late submissions; any penalties associated with unexcused late submissions are non-negotiable.

6.1 Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work. Homework assignments will be examined using cheat detection programs to find evidence of unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student. If you supply a 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 platform which would allow other parties to it (public repositories, pastebin, etc). If you would like to use version control, use a private repository on github.gatech.edu

Homework collaboration is limited to high-level 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.

High-level collaboration means 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, or providing other students any part of your code.

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.

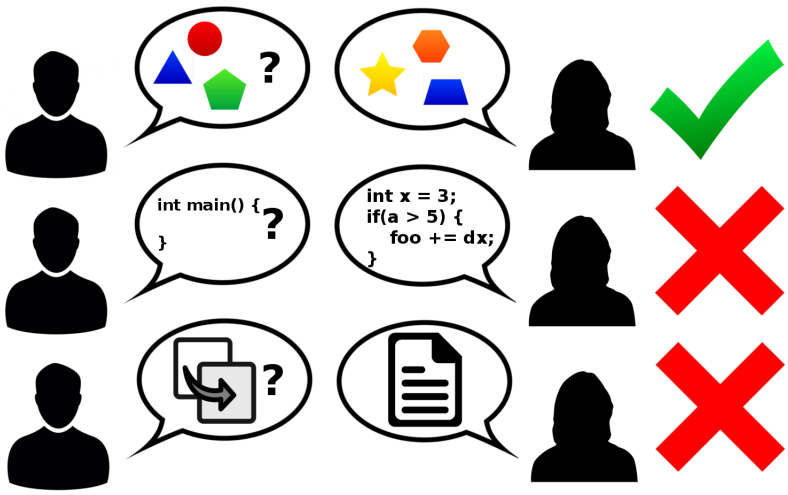


Figure 2: Collaboration rules, explained colorfully