CS 2110 Homework 4
Assembly Programming

Elizabeth Hong, Henry Bui, Nihar Manangi, Ganning Xu, Michael Yi
Jordan Miao, Willson Pan, Andy Garcha, Jessica Liu, Galadriel Cho

Spring 2025

Contents

1 Part 1: Assembly Warmup

1.1 Overview

1.1.1 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.

1.1.2 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. `integerSquareRoot.asm`
2. `reverseString.asm`
3. `binaryStringToInt.asm`
4. `findMinIndex.asm`

1.1.3 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!

1.1.4 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.
- Comment your code in a way that shows how each line implements the pseudocode. For example,
  `ADD R0, R0, 1 ; incrementing counter variable`
  is a good comment while
  `ADD R0, R0, 1 ; add 1 to R0`
  doesn't give a lot of helpful information.
- 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. In the LC3Tools debugger, you can set breakpoints and step through line-by-line to check that your registers and values in memory are being updated as you expect them to.
- 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.
- You may make your own variables (ex. `Integer Square Root` has `N` and `RESULT`) in order to store your own variables (you may find in the later subroutines that registers are a scarce resource). However, please do not change the names of the ones given to you.

1.2 Detailed Instructions

1.2.1 Integer Square Root

This function takes an integer n and finds the integer square root $\lfloor \sqrt{n} \rfloor$.
Assumptions:

- The input N is a non-negative 16-bit integer.
- If $N = 0$, the function should return 0.
- The function only uses addition (no multiplication, division, modulus, or bitwise shifts).

Relevant labels:

- `N`: Contains the value of the non-negative 16-bit integer input.
- `RESULT`: Contains the address in memory at which you must store the result L.

Walkthrough Example
The algorithm provided in the pseudocode uses a linear search approach using only addition. We define the following variables:

1. L, a guess for what the integer square root is
2. a, an accumulator, keeping track of what $(L+1)^2$ is
3. d, a difference, keeping track of the difference between a and $(L+2)^2$

The function calculates the integer square root of $N = 20$:

Iteration	L	a	d
Init	0	1	3
1	1	$1 + 3 = 4$	5
2	2	$4 + 5 = 9$	7
3	3	$9 + 7 = 16$	9
4	4	$16 + 9 = 25$	11

Since $a = 25 > 20$, we stop.
At this point, the function terminates and returns $L = 4$, which is the integer square root of 20.
Notes

- The function works by incrementally computing perfect squares using only addition.
- The value of d starts at 3 and increases by 2 at each step, ensuring that a follows the sequence $1, 4, 9, 16, 25, \ldots$.

Implement your assembly code in `integerSquareRoot.asm`.
Pseudocode on next page for your convenience

Suggested Pseudocode:

```
int L = 0;
int a = 1;
int d = 3;

while (a <= N) {
    a = a + d;  // Compute the next perfect square
    d = d + 2;  // Increase the difference between squares
    L = L + 1;  // Increment L
}

mem[mem[RESULT]] = L;
```

1.2.2 Binary String to Int

Given an unsigned binary number as a string, translate it to its numerical value.
Assumptions:

- '0' ≤ binaryString[i] ≤ '1'
- The provided binary string is unsigned.
- We do not have to worry about overflow.

Relevant labels:

- `BINARYSTRING`: Holds the starting address of the binary string.
- `RESULTADDR`: Holds the **address** of where you will store your numerical value result.
- `ASCIIDIG` holds the value 48.

Say the string at `BINARYSTRING` was "11001". After running your program, mem[mem[`RESULTADDR`]] should equal 25 (base 10). If mem[`RESULTADDR`] = x4000, then this means the value all the way at memory address x4000 should be changed to 25. **The value at `RESULTADDR` should be unchanged.**

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

Implement your assembly code in `binaryStringToInt.asm`.

Suggested Pseudocode:

```
int length = 0;
while (BINARYSTRING[length] != 0) {
    length++;
}

int result = 0;

for (int i = 0; i < length; i++) {
    result = result << 1;
    result += BINARYSTRING[i] - 48;
}

mem[mem[RESULTADDR]] = result;
```

1.2.3 Reverse String

Given a string starting at the address in `STRING`, reverse all the letters in the string, in place.
Assumptions:

- Possible characters in the string: lowercase letters, uppercase letters, and spaces
- If the string is of an odd length the *middle letter* does not change position. For example, “write” becomes “etirw” (the “i” did not move)

Relevant labels:

- `STRING`: Holds the base address of the array containing the string.

Say the string is `gOOglE`. After running your program, the string starting at the address in STRING should be `ElgOOg`.
Implement your assembly code in `reverseString.asm`.

Suggested Pseudocode:

```
int length = 0;
while (STRING[length] != 0) {
    length++;
}
int start = 0;
int end = length - 1;

while (start < end) {
    char temp = STRING[start];
    STRING[start] = STRING[end];
    STRING[end] = temp;

    start++;
    end--;
}
```

1.2.4 Find Minimum Index

This function scans through an array of signed integers to determine the index of the smallest value in the array. It is assumed that the array contains at least one element. The array is stored in consecutive memory locations.
Assumptions:

- The array contains at least one element (i.e., `length` ≥ 1).

Relevant labels:

- `RESULT`: Holds the address where the final minIndex should be stored.
- `ARRAY`: Holds the base address of the array that holds the signed integers.
- `LENGTH`: Holds the length of the array.

Say `ARRAY` = {`-1, 2, 7, 3, -8`}. After running your program, memory at `RESULT` should hold 4.
Implement your assembly code in `findMinIndex.asm`.

Suggested Pseudocode:

```
int minIndex = 0;
int minValue = ARRAY[0];
for (int i = 1; i < LENGTH; i++) {
    if (ARRAY[i] < minValue) {
        minValue = ARRAY[i];
        minIndex = i;
    }
}
mem[mem[RESULT]] = minIndex;
```

2 Part 2: Programming with Subroutines

2.1 Overview

2.1.1 Purpose

Now that you've been introduced to assembly, think back to some high level languages you know such as Python or Java. When writing code in Python or Java, you typically use functions or methods. Functions and methods are called subroutines in assembly language.

In assembly language, how do we handle jumping around to different parts of memory to execute code from functions or methods? How do we remember where in memory the current function was called from (where to return to)? How do we pass arguments to the subroutine, and then pass the return value back to the caller?

The goal of this assignment is to introduce you to the Stack and the Calling Convention in LC-3 Assembly. This will be accomplished by writing your own subroutines, calling subroutines, and even creating subroutines that call themselves (recursion). By the end of this assignment, you should have a strong understanding of the LC-3 Calling Convention and the Stack Frame, and how subroutines are implemented in assembly language.

2.1.2 Task

You will implement each of the three subroutines (functions) listed below in LC-3 assembly language. Please see the detailed instructions for each subroutine on the following pages. The autograder checks for certain subroutine calls with arguments pushed in the correct order, so we suggest that you follow the provided algorithms when writing your assembly code. Your subroutines must adhere to the LC-3 calling convention.

1. `caesarCipher.asm`
2. `mergeSort.asm`
3. `leapfrog.asm`

Later in this document, you can also find some general tips for successfully writing assembly code.

2.1.3 Criteria

Your assignment will be graded based on your ability to correctly translate the given pseudocode for subroutines (functions) into LC-3 assembly code, following the LC-3 calling convention. Please use the LC-3 instruction set when writing these programs. Check the deliverables section for deadlines and other related information.

You must obtain the correct values for each function. In addition, registers R0-R5 and R7 must be restored from the perspective of the caller, so they contain the same values before and after the caller's JSR call. Your subroutine must return to the correct point in the caller's code, and the caller must find the return value on the stack where it is expected to be. If you follow the LC-3 calling convention correctly, each of these things will happen automatically.

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. Good luck and have fun!

2.2 Detailed Instructions

2.2.1 Caesar Cipher

Invented by Julius Caesar a looooong time ago, the Caesar Cipher is a simple encryption method that shifts each letter in a message by another letter a fixed number (k) of positions away. If going k letters away from current letter causes you to run "past" the last letter, wrap around to the beginning. Additionally, lowercase letters should only be shifted to lowercase letters, and uppercase letters should only be shifted to uppercase letters. You should perform the shift in-place.

In `caesarCipher.asm`, we want you to implement two subroutines: `MOD` and `ENCRYPT` (see the following pseudocodes for more details). As a reminder, please do **not** change the names of provided subroutines. Otherwise, your submission will not pass the autograder.

Remember to get your arguments from the stack!

For example:

```
ENCRYPT("hello", 7) should return "olssv"
ENCRYPT("hello", 33) should return "olssv"
ENCRYPT("jAckEts", 1) should return "kBdlFut"
ENCRYPT("Yellow", 0) should return "Yellow"
```

Relevant labels

- `STACK_PTR` contains the memory address corresponding to the top of the stack
- `STRING` contains the starting address of the first letter in the string
- `SHIFT` contains an integer representing how far each letter in STRING should be shifted by
- `ASCIIUPPERA` contains ASCII value of "A"
- `ASCIILOWERA` contains ASCII value of "a"
- `ALPHABETLEN` contains 26

Assumptions

- `STRING` will only contain uppercase and lowercase characters from the english alphabet

Pseudocode on next page for your convenience

Suggested Pseudocode: Here are the pseudocodes for these subroutines:

```
MOD(int a, int b) {
    while (a >= b) {
        a -= b;
    }
    return a;
}

ENCRYPT(String str, int k) {
    int length = 0;
    while (str[length] != 0) {
        length++;
    }
    for (int i = 0; i < length; i++) {
        char = str[i];
        if (char >= ‘a’ && char <= ‘z’) {
            char = char - ‘a’;
            char = MOD(char + k, 26);
            char = char + ‘a’;
        } else if (char >= ‘A’ && char <= ‘Z’) {
            char = char - ‘A’;
            char = MOD(char + k, 26);
            char = char + ‘A’;
        }

        str[i] = char;
    }
}
```

2.2.2 Merge Sort

Now, you'll be implementing merge sort, recursively. Here's a quick refresher on how merge sort works. As a reminder, please do **not** change the names of provided subroutines. Otherwise, your submission will not pass the autograder.

In essence, you will divide an array into two halves, sort each half by merge sort, and merge them with the "merge" operation. You will implement 3 subroutines:

- `DIVIDE(a, b)`: Divides `a` by `b`, useful for dividing in half.
- `MERGE(arr, buf, start, mid, end)`: Applies the merge operation on array, assuming there exist two "sorted" halves of the array, `arr[start:mid]` and `arr[mid:end]`, where `arr[mid]` is included in the second half, not the first. This operation uses a temporary buffer (`buf`) to merge the subarrays together and then copies the data back into `arr`.
- `MERGESORT(arr, buf, start, end)`: Sorts `arr[start:end]` using merge sort. This operation uses a temporary buffer (`buf`) for merging.

Assumptions

- the sorted array should be placed at `ARRAY`
- `ARRAY` is 0-indexed
- When specifying the "start" and "end" indices, "start" is inclusive and "end" is exclusive. For example, if `arr = [0, 1, 2, 3, 4, 5]`, then `arr[0:3] == [0, 1, 2]`.

Relevant Labels

- `STACK_PTR` contains the memory address corresponding to the top of the stack
- `LENGTH` contains length of `ARRAY`
- `ARRAY` contains the address of the first element in the array
- `BUF` contains the address of the first element in a temporary array, with same length as `ARRAY`

Examples

- `MERGE([2, 3, 1, 8, 4], buffer, 0, 2, 4)` should update the array to [$\underline{1, 2, 3, 8}$, 4].
- `MERGE([5, 6, 7, 1, 2, 3, 4], buffer, 0, 3, 7)` should update the array to [$\underline{1, 2, 3, 4, 5, 6, 7}$].
- `MERGESORT([5, 2, 3, 1], buffer, 0, 4)` should update the array to `[1, 2, 3, 5]`.
- `MERGESORT([5, 1, 1, 2, 0, 0], buffer, 0, 6)` should update the array to `[0, 0, 1, 1, 2, 5]`.

Pseudocode on next page for your convenience.

Suggested Pseudocode:

Here is the pseudocode for these subroutines:

```
DIVIDE(int a, int b) {
    if (b == 0) {
        return 0;
    }
    int quotient = 0;
    while (a >= b) {
        a -= b;
        quotient++;
    }
    return quotient;
}


MERGESORT(int arr[], int buf[], int start, int end) {
    if (start >= end) {
        return;
    }
    int mid = DIVIDE(start + end, 2);
    MERGESORT(arr, buf, start, mid);
    MERGESORT(arr, buf, mid + 1, end);
    MERGE(arr, buf, start, mid, end);
}

MERGE(int arr[], int buf[], int start, int mid, int end) {
    int i = start;
    int j = mid;
    int k = start;
    while (i < mid && j < end) {
        if arr[i] <= arr[j] {
            buf[k] = arr[i];
            k++;
            i++;
        } else {
            buf[k] = arr[j];
            k++;
            j++;
        }
    }
    while (i < mid) {
        buf[k] = arr[i];
        k++;
        i++;
    }
    while (j < end) {
        buf[k] = arr[j];
        k++;
        j++;
    }
    for (i = start; i < end; i++) {
        arr[i] = buf[i];
    }
}
```

2.2.3 Leapfrog

For this exercise, while it is useful to know the end goal of the program, you do not necessarily have to understand the pseudocode before implementing it. After you have completed it, you can step through it to see exactly how it works. All you need to know to begin though, is that Leapfrog reads (loads) and writes (stores) its assembled code `x0100` forward in memory, jumps to it, and repeats until address `xFE00` (trying to go beyond this point would result in an access violation).

Suggested Pseudocode:

```
inc = x0100
func_addr = starting address of function
copy_addr = func_addr + inc
start_copy = copy_addr
stop_addr = xFE00
if (start_copy - stop_addr == 0):
    HALT
curr = mem[func_addr]
while (curr != 0) {
    curr = mem[func_addr]
    mem[copy_addr] = curr
    func_addr++
    copy_addr++
}
mem[copy_addr] = 0
PC = start_copy
.fill 0
```

3 Deliverables

Turn in the following files on Gradescope:

1. `integerSquareRoot.asm`
2. `reverseString.asm`
3. `binaryStringToInt.asm`
4. `findMinIndex.asm`
5. `caesarCipher.asm`
6. `mergeSort.asm`
7. `leapfrog.asm`

4 Debugging

When you turn in your files, you might not receive a perfect score. Does this mean you change one line and spam Gradescope until you get a 100? No! You can use LC3Tools to step through each line in your assembly program to see if the program is behaving the way it's supposed to!

Each failed test case lists the arguments used and the expected return value. For example, here is a failed test case for *Reverse String*:

```
self = <test_123_reverse_string.RevStrSuite testMethod=test_reverse_string_0>
string = 'gOOglE'

    @parameterized.expand([
        "gOOglE", # Example test case
        "Reverse This",
        "CS Twenty One tEN",
        "hELlo",
        "gORGIa tECh",
        "WolleY StekcaJ",
        ''.join(random.choices(string.ascii_letters + ' ', k=100))
    ],
        name_func=lambda fn, n, param: f"{fn.__name__}_{n}"
    )
    def test_reverse_string(self, string):
        """
        (1.2.3) Reverse String Test
        """
        string_addr = self.readMemValue("STRING")
        self.writeString(string_addr, string)

        self.runCode()
        self.assertHalted()

        reversed_str = string[::-1]

>       self.assertString(string_addr, reversed_str)
E       AssertionError: 103 != 45 : String starting at mem[mem[STRING]] did not match expected
E       expected: ElgOOg [69, 108, 103, 79, 79, 103, 0]
E       actual:   El-OOg [69, 108, 45, 79, 79, 103, 0]

tests/test_123_reverse_string.py:38: AssertionError
```

Here we can see:

- The arguments (`string = g00g1E`)
- The expected value (`expected: E1g00g`)
- The actual value from your program (`actual: E1-00g`)

To test your program, edit the arguments in your ASM program in LC3Tools to the arguments from the failed test case, run the program, and ensure the resulting value from the program aligns with the expected value.
Some useful tips for LC3Tools:

- You can add breakpoints to pause the program at certain points. Hit the octagon next to the memory row to add a breakpoint.
- Use the Step Over, Step In, and Step Out buttons (on the left-hand side)! The *Step Over* button executes one instruction at a time (without jumping into a subroutine), and *Step In* button executes one instruction at a time (while jumping into subroutines if a call is present).
- You can jump to an address stored in memory by right-clicking the memory row and pressing "Jump to Address". This is useful to read values (such as the values in the array).

4.1 Running the Autograder

To run the autograder, run `./grade.sh` (Linux, macOS) or `.\grade.bat` (Windows) **outside of the Docker container**.
After executing the autograder, if you want to only run tests that previously failed, run:

```
./grade.sh --last-failed (Linux, macOS)
.\grade.bat --last-failed (Windows)
```

If you want to execute the autograder on a specific set of test cases, add the file name or the test ID. For example, to run only *Merge Sort* test cases, run:

```
./grade.sh tests/test_222_merge_sort.py (Linux, macOS)
.\grade.bat tests/test_222_merge_sort.py (Windows)
```

To run only the divide calling convention test case for *Merge Sort*, run:

```
./grade.sh tests/test_222_merge_sort.py::MergeSortSuite::test_divide_calling_convention
(Linux, macOS)
.\grade.bat tests/test_222_merge_sort.py::MergeSortSuite::test_divide_calling_convention (Windows)
```

(Note that these test IDs can be found in the test report.)
Note that these test cases are fully written in Python and are intended to be customized. If you want more control in testing your programs, you can make your own test cases by adding a `test_custom.py` to the `tests/` folder in your directory. Refer to `docs/API.md` at `https://pypi.org/project/lc3-ensemble-test/` for more information about creating your own test cases.
Also note that making your own test cases is *not* required! The project can be completed without changing the test cases; they are simply provided for your personal use.

5 Tips

1. Start early. This homework can be difficult to debug and can take up a significant amount of time
2. **COMMENT YOUR CODE**. Commenting helps you to debug and helps the TAs if you choose to ask for help in office hours or on Piazza
3. Debug using different test cases. The autograder and pdf provide test cases and expected outputs that you can use in your own assembly file.
4. Use version control. Sometimes during debugging you can make negative progress and it's useful to have a previous version to work in order to avoid having to start over or waste time restoring something you already had.

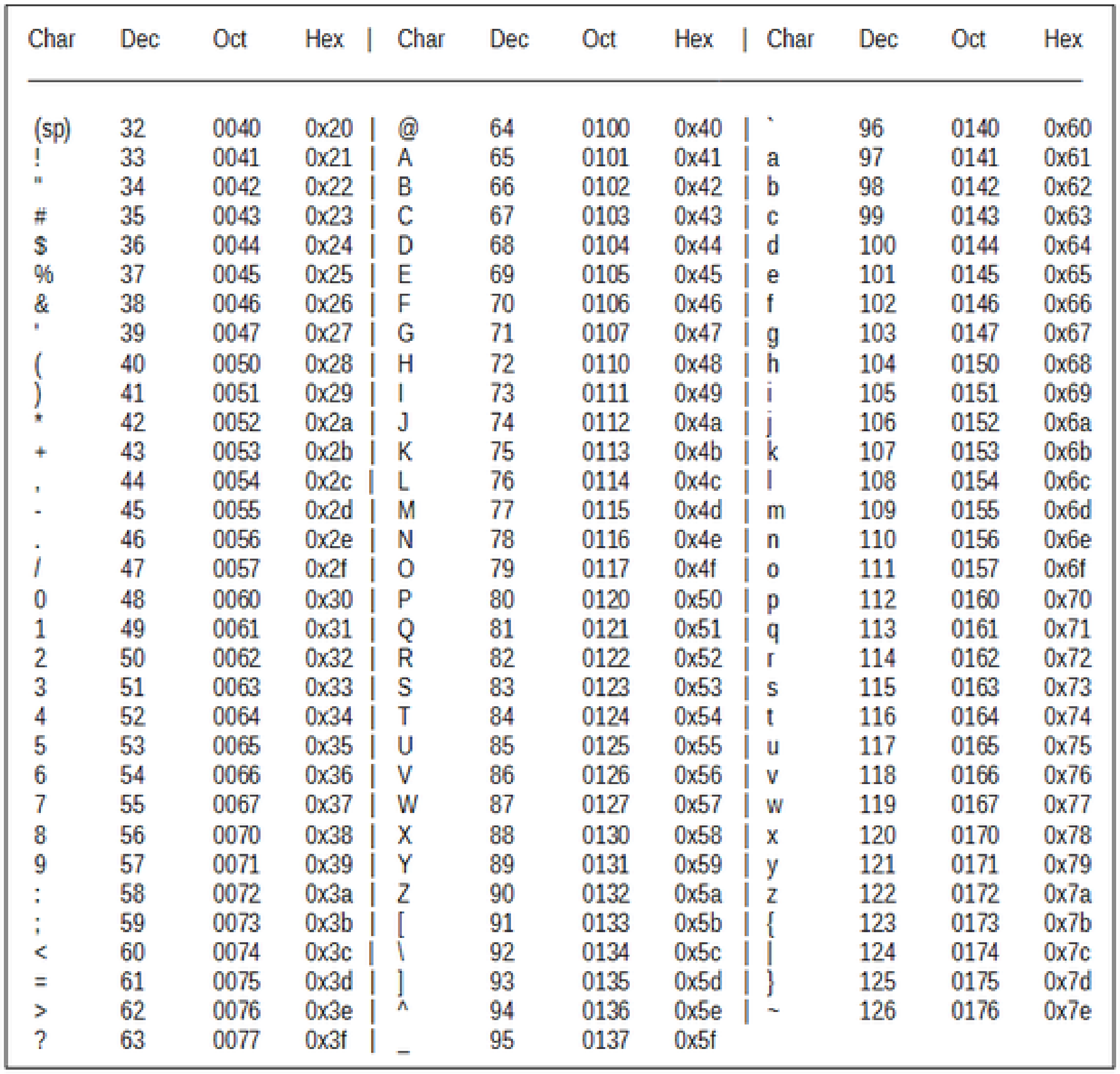
6 Demos

This homework will be demoed. The demos will be 10 minutes long and will occur **IN PERSON**. Stay tuned for details as the due date approaches.

Please note: Your grade for this assignment is split into 2 parts. 75% of your grade is based upon how well you do with the autograder. The other 25% is determined by how well you do in the demo. **This means that when you submit to Gradescope intially, your score will be out of 75.** Again, more details to come soon.

7 Appendix

7.1 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				

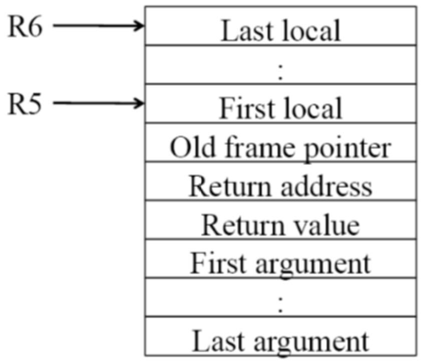
Figure 1: ASCII Table — Very Cool and Useful!

7.2 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



8 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.

8.1 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.

More details, along with this course's policy on AI assistants, can be found in the syllabus.

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.

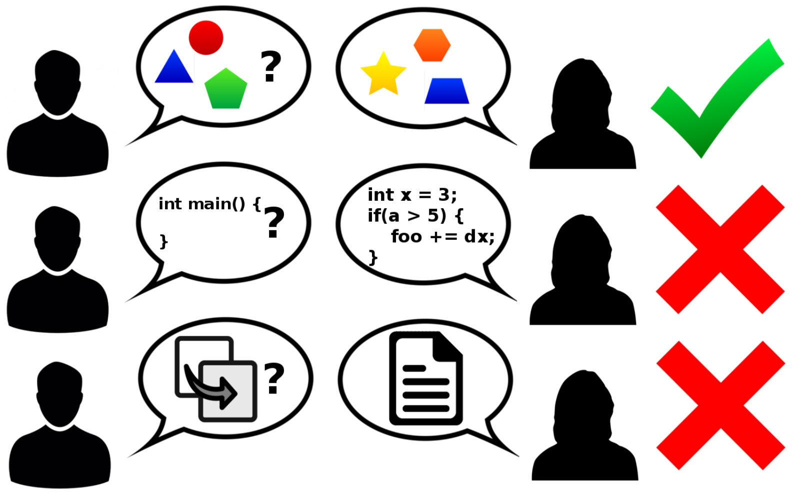


Figure 2: Collaboration rules, explained colorfully