CS 2110 Project 3: Assembly

Zilong Huang, Jason Ng, Charlie Gunn, Sean Crowley, Johnson Lu

Section A, Fall 2021

Contents

1 General

1.1 Introduction

Hello and welcome to Project 3. In this project, you'll write some simple programs in assembly, and then implement bit shifting, the Collatz conjecture problem, and a dot product calculator in assembly!

The project is due October 29th, 11:59PM EST

Please read through the entire document before starting. Often times we elaborate on instructions later on in the document. **Start early** and if you get stuck, there's always Piazza or office hours.

1.2 General Instructions

- You can create your own labels and fill them with data as you see fit. However, if you have two labels corresponding to the same memory location (e.g., two labels without an instruction between them) this may cause autograder errors.
- You can modify the values stored in some of the helper labels we provide you if you think it'll make it easier for you to program with
- Take a look at Section 8 for details on the LC-3 Assembly Programming requirements that you must adhere to

1.3 Running the autograder

Take a look at section 6 for information on how to run the autograder, and how to debug your code. For this project, we will be using complx in the Docker Container to debug our code.

2 Simple ASM Programs

2.1 Compare

To start off, you will implement the compare function. Comparing two values is something you will do frequently in assembly, so it's important you know how to do this. The two operands are stored in memory with the labels `A` and `B`. You will load them from memory, and store a value in the label `RESULT`. The value that you store to `RESULT` depends on whether `A` is greater than `B`.

- If $A < B$, you should store -1.
- If $A = B$, you should store 0.
- If $A > B$, you should store 1.

You may assume that the memory addresses labeled as `A, B, RESULT` are reachable via a PC-offset instruction with 9-bit offset. Write your code in `comp.asm`.

2.2 Modulus

In this part of the project, you will be implementing the modulus operator that you will find in a lot of programming languages. Since there are different interpretations of modulus for negative numbers, you should take the absolute values of both your operands and then perform the operation. The two operands are stored in memory with the labels `A` and `B`. You will load them from memory, perform the modulus operation, and then store the result in the label `RESULT`.

Suggested Pseudocode:

```
a = mem[A];
b = mem[B];

a = abs(a);
b = abs(b);

while (a >= b) {
    a = a - b;
}

mem[RESULT] = a;
```

You may assume that the memory addresses labeled as `A, B, RESULT` are reachable via a PC-offset instruction with 9-bit offset. You do not need to worry about overflow for negation or subtraction. Write your code in `mod.asm`.

2.3 String Manipulation

Next, you will write a program that takes in a string stored in memory, changes any lowercase letters in the string to uppercase letters, and stores this new string back into memory.

- Strings are essentially a contiguous array of ASCII values. In this case, the first character is stored at the address indicated by the **value** at the address labeled as `STRING`.
- The string continues until the first instance of a null terminator, which has the value of 0.

Here is an example layout:

Address	Label	Value
...	...	...
x3021	STRING	x4000
x3022	RESULT	
...	...	...
x4000		"h"
x4001		"A"
x4002		"h"
x4003		"A"
x4004		0

The string that you receive can contain any characters, and you should change all letters to uppercase, and leave non-letters as is. You may assume that the memory address labeled as `STRING` will be reachable via a PC-offset instruction with 9-bit offset. However, the actual contents of the string might be stored at addressed that can't be reached via a 9-bit offset. We have given you some labels in the assembly file that you might find helpful. Write your code in `toupper.asm`.

3 Bit Shifting

For this section of the product, you will implement bit-shifting in LC3 assembly as subroutines. As such, values for bit shifting will no longer be placed at label values, but rather, **passed-in values will now be passed in through registers. Results will also be stored in registers**.You will be given an original value, and amount of bits to shift by. Note, for shifting, expected behavior involves adding zero's from the left and right. We will not be sign-extending.

The operands will be passed in through 2 registers, R0 for the value and R1 for the shift amount. The result should be stored at R0. **You will not have to handle overflow**. Write your code in `collatzconjecture.asm`, under the appropriate sections and `.orig` tags (`x3100` for `SHIFTLEFT`, `x3200` for `SHIFTRIGHT`).

Suggested Pseudocode (Shift Right):

```
val = R0;
amt = R1;

result = 0;

while (amt < 16) {
    result = result + result;
    if (val < 0) {
        result++;
    }
    val = val + val;
    amt++;
}

R0 = result;
```

Suggested Pseudocode (Shift Left):

```
val = R0;
amt = R1;

while (amt > 0) {
    val += val;
    amt--;
}

R0 = val;
```

4 Collatz Conjecture

For this part of the project, you will be implementing a calculator for iterations of the Collatz Conjecture. This program will use labels as input/output, but will be required to call the subroutines `shiftleft` and `shiftright` that you wrote previously.

The Collatz Conjecture is a famous unsolved math problem, which operates under a few simple rules. We have our Collatz function, $C(n)$, which takes in exclusively positive integers. If n is an odd number, $C(n) = 3n + 1$. If n is an even number, $C(n) = \frac{n}{2}$. From here, we seek to calculate how many times we need to run n (and its subsequent results) through the Collatz function in order for it to reach the value 1.

As an example, let's say we start with the number 5.

```
5 is odd, so we begin by performing (3 * 5) + 1 = 16.
16 is even, so we perform (16/2) = 8.
8 is even, so we perform (8/2) = 4.
4 is even, so we perform (4/2) = 2.
2 is even, so we perform (2/2) = 1.

In this example, our Collatz operation required 5 total iterations to reach 1.
```

The initial value will be passed in through the label `VALUE`. You may assume that the memory addresses labeled as `VALUE` and `RESULT` are reachable via a PC-offset instruction with 9-bit offset. The result (i.e. the number of Collatz iterations it took to reach 1) should be stored at the label `RESULT`. Write your code in `collatzconjecture.asm`, starting at `x3300`. Note that **you are required to call your `shiftleft` and `shiftright` subroutines during the execution of this program. See pseudocode for details**.

Suggested Pseudocode:

```
val = mem[VALUE];
num_of_iterations = 0;
while (val > 1) {
    if (val % 2 != 0) {
        val = shiftleft(val, 1);
        val += val;
        val += 1;
    } else {
        val = shiftright(val, 1);
    }
    num_of_iterations++;
}

mem[RESULT] = num_of_iterations;
```

5 Dot Product

The last assembly program you'll be writing for this project is a dot product calculator. Given two arrays/vectors in memory, your job is to compute the dot product, and return it back into memory. In mathematics, the dot product is calculated as the sum of multiplication between each pair of vector components. If $A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3\}$, the dot product between A and B would be $(a_1 * b_1) + (a_2 * b_2) + (a_3 * b_3)$.

For example,

```
A = {1, 2, 3}
B = {4, -2, -1}

Dot Product of A and B = (1 * 4) + (2 * -2) + (3 * -1) = -3
```

In this program, values inside vectors can be positive or negative integers, and you are given 3 input values at labels `VECTOR_A`, `VECTOR_B`, and .

1. `VECTOR_A` represents the starting location of the first vector in memory.
2. `VECTOR_B` represents the starting location of the second vector in memory.
3. `LENGTH` represents the length of each individual vector.

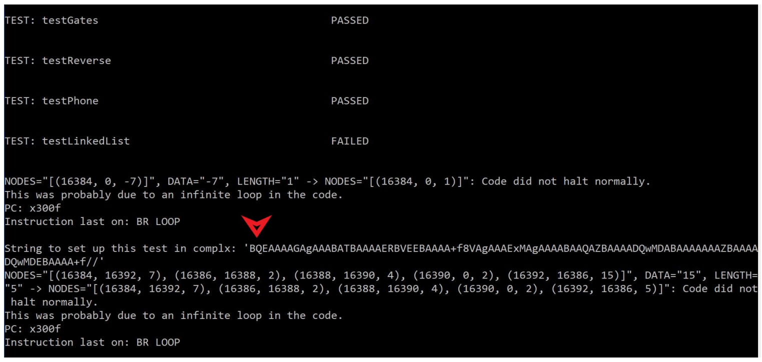
You may assume that the memory addresses labeled as `VECTOR_A`, `VECTOR_B`, `LENGTH` and `RESULT` are reachable via a PC-offset instruction with 9-bit offset. The result should be stored at the label `RESULT`. Write your code in `dotproduct.asm`.

6 Debugging

When you turn in your files on Gradescope for the first time, 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 a handy tool known as tester strings.

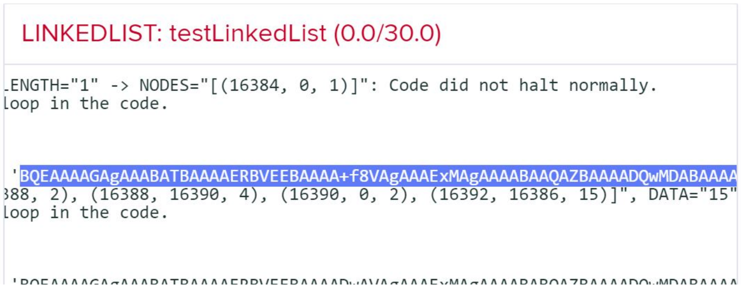
1. First off, we can get these tester strings in two places: the local grader or off of Gradescope. To run the local grader:
   - Mac/Linux Users:
     (a) Navigate to the directory your project is in. **In your terminal, not in your browser**
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) On **docker quickstart**, navigate to the directory your project is in
     (b) 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
```

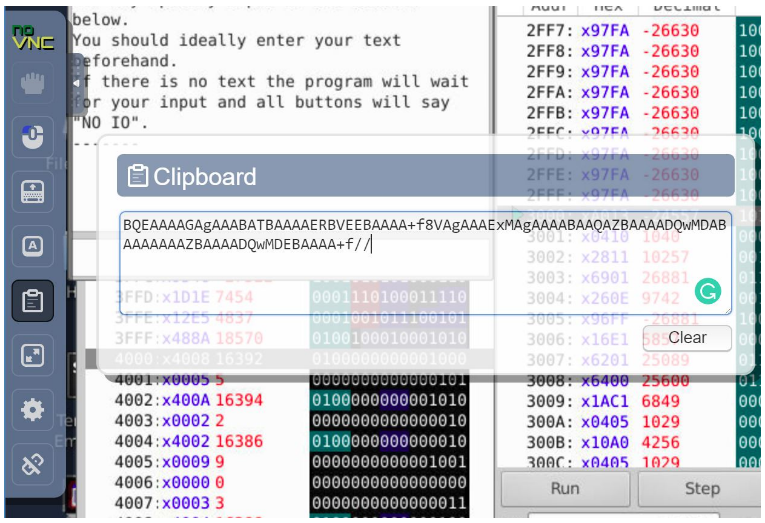
Copy the string, starting with the leading 'B' and ending with the final backslace. Do not include the quotations.
Side Note: If you do not have docker installed, you can still use the tester strings to debug your assembly code. In your Gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslace and again, don't copy the quotations.



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.



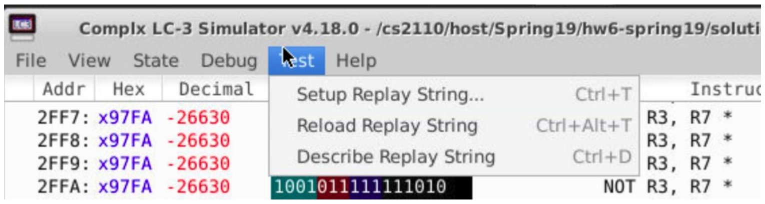
```
below.
You should ideally enter your text
beforehand.
If there is no text the program will wait
for your input and all buttons will say
"NO IO".
```

Clipboard

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

```
2FF7: x97FA -26630
2FF8: x97FA -26630
2FF9: x97FA -26630
2FFA: x97FA -26630
2FFB: x97FA -26630
3008: x6400 25600
3009: x1AC1 6849
300A: x0405 1029
300B: x10A0 4256
300C: x0405 1029

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
```

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

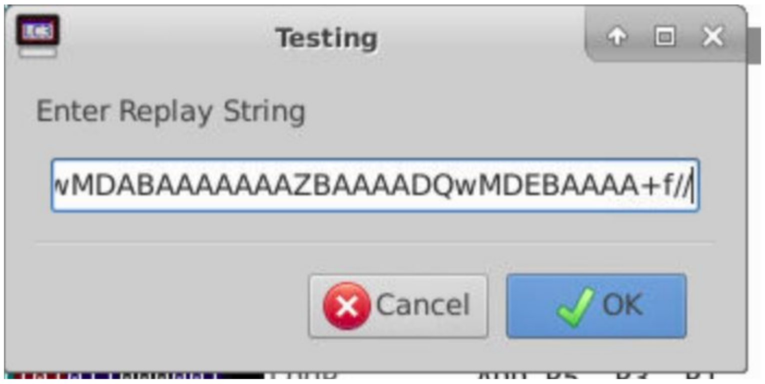


Complx LC-3 Simulator v4.18.0 - /cs2110/host/Spring19/hw6-spring19/soluti

File View State Debug Test Help
Setup Replay String... Ctrl+T
Reload Replay String Ctrl+Alt+T
Describe Replay String Ctrl+D

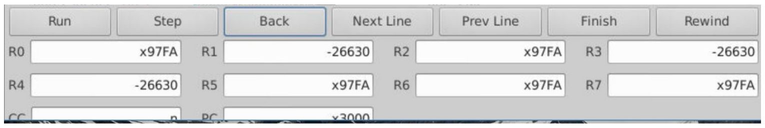
```
Addr  Hex    Decimal
2FF7: x97FA  -26630
2FF8: x97FA  -26630
2FF9: x97FA  -26630
2FFA: x97FA  -26630  1001011111111010  NOT R3, R7 *
```

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



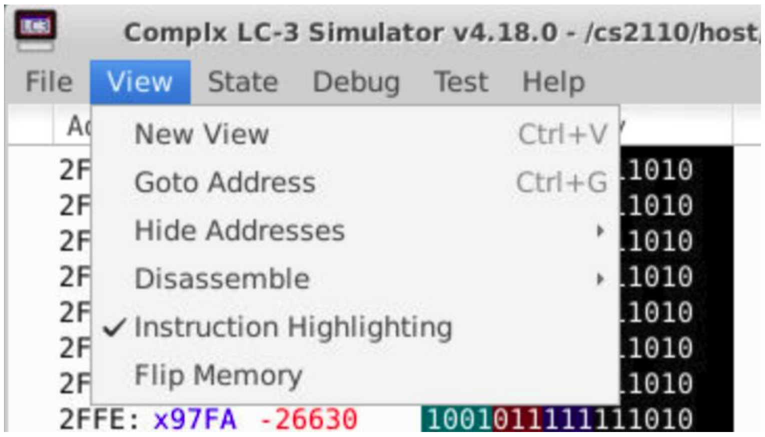
vMDABAAAAAAAZBAAAADQwMDEBAAAA+f//

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.



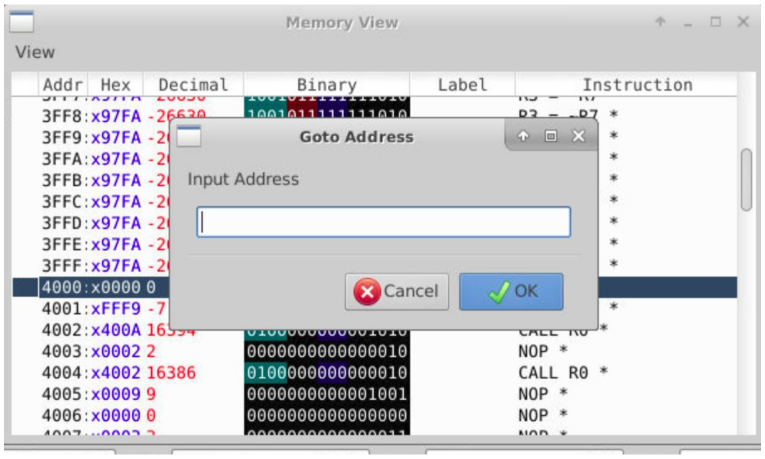
R0 x97FA R1 -26630 R2 x97FA R3 -26630
R4 -26630 R5 x97FA R6 x97FA R7 x97FA

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



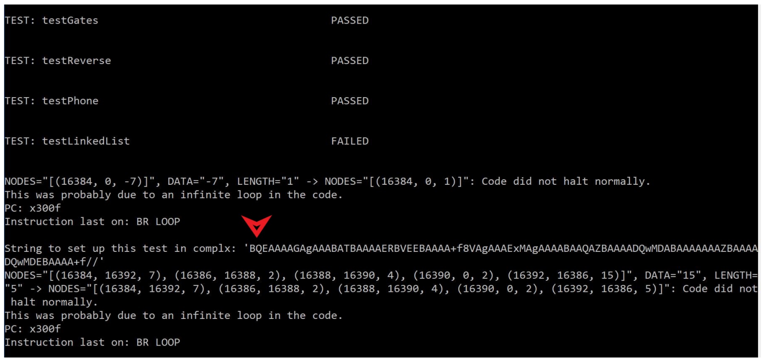
2FFE: x97FA -26630 1001011111111010

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



```
Memory View
View
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                                          *
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. Just be careful: if you misclick and select Show Non Zero, it *may* make the window freeze (it's a known Complx bug).



```
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
```

7 Deliverables

Turn in the files `comp.asm`, `mod.asm`, `toupper.asm`, `collatzconjecture.asm`, and `dotproduct.asm` to Gradescope by the due date.

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

8 LC-3 Assembly Programming Requirements

8.1 Overview

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 randomize the memory and load your program by doing File - Randomize and Load using complx.
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.

9 Demos

This project will be demoed. Demos are designed to make sure that you understand the content of the project and related topics. They may include technical and/or conceptual questions.

- Sign up for a demo time slot via Canvas **before** the beginning of the first demo slot. This is the only way you can ensure you will have a slot.
- If you cannot attend any of the predetermined demo time slots, e-mail the Head TA **before** the beginning of the first demo slot.
- If you know you are going to miss your demo, you can cancel your slot on Canvas with no penalty. However, you are **not** guaranteed another time slot. You cannot cancel your demo within 24 hours or else it will be counted as a missed demo.
- Your overall project score will be `((autograder_score * 0.5) + (demo_score * 0.5))`, meaning if you received a 90% on your autograder, but a 30% on the demo you would receive an overall score of 60%. **If you miss your demo you will not receive any of these points and the maximum you can receive on the project is 50%.**
- You will be able to makeup one of your demos at the end of the semester for half credit.

10 Rules and Regulations

10.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. 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. Please read the assignment in its entirety before asking questions.

3. Please start assignments early, and ask for help early. Do not email us a few hours before the assignment is due with questions.

4. 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.

10.2 Submission Conventions

1. When preparing your submission you may either submit the files individually to Canvas/Gradescope or you may submit an archive (zip or tar.gz only please) of the files. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.

2. 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.

10.3 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.

3. You are additionally responsible for ensuring that the collaborators list you have provided in your submission is accurate.

4. Projects turned in late receive partial credit within the first 48 hours. We will take off 30% of the points for a project submitted between 0 and 24 hours late, and we will take off 50% of the points for a project submitted between 24 and 48 hours late. We will not accept projects turned in over 48 hours late. This late policy is also in the syllabus.

5. You alone are responsible for submitting your project before the assignment is due; neither Canvas/Gradescope, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until the deadline.

10.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class.

1. Students are expected to have read and agreed to the Georgia Tech Honor Code, see http://osi.gatech.edu/content/honor-code.
2. Suspected plagiarism will be reported to the Division of Student Life office. It will be prosecuted to the full extent of Institute policies.
3. A student must submit an assignment or project as his/her own work (this is what is expected of the students).
4. Using code from GitHub, via Googling, from Stack Overflow, etc., is plagiarism and is not permitted. Do not publish your assignments on public repositories (i.e., accessible to other students). This is also a punishable offense.
5. Although discussion among the students through piazza and other means are encouraged, the sharing of work is plagiarism. If you are not sure about it, please ask a TA or stop by the instructor's office during the office hours.
6. You must list any student with whom you have collaborated in your submission. Failure to list collaborators is a punishable offense.
7. TAs and Instructor determine whether the project is plagiarized. Trust us, it is really easy to determine this....

10.5 Is collaboration allowed?

From the syllabus:

- You must submit an assignment or project as your own work. No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct.
- Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "submission of material that is wholly or substantially identical to that created or published by another person").
- Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.

Any of your peers with whom you collaborate in a conceptual manner with must be properly added to a **collaborators.txt** file. Collaborating with another student without listing that student as a collaborator is considered plagiarism.