CS 2110 Project 3: Assembly

Julian Gu, Charlie Gunn, Todd Hayes, Justin Hu, Elton Pinto, Ammar Ratnani

Section A, Fall 2020

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 build your very own Reverse Polish Notation calculator in assembly!

The project is due October 21, 12:00PM 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.
- 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
- You are free to create your own helper subroutines (although you are not required to for this project).
- Take a look at Section 6 for details on the LC-3 Assembly Programming requirements that you must adhere to

1.3 Running the autograder

Take a look at section 4 for information on how to run the autograder, and how to debug your 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 To Uppercase

Next, you will write a program that prints out an all uppercase version of an input string. You do not need to change the string store in memory, but instead just print the uppercase string to console.

- 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"

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. As a hint, take a look at the trap vectors to see how you might print characters to the console. Write your code in toupper.asm.

2.4 Alternating array sum

For this section, you will iterate through an array and either add or subtract from a sum depending on the index of the array. For every element with an even index, you should add its value to the sum. For every element with an odd index, you should subtract its value from the sum. The sum should start from 0, and we are using 0-indexing here. The array length will be stored at the label LENGTH and the **address** of the first element will be stored at the label ARRAY. Note that ARRAY stores the address of the first element instead of the first element itself. An example memory layout might look something like this:

Address	Label	Value
...	...	...
x3020	LENGTH	4
x3021	ARRAY	x7000
x3022	RESULT	
...	...	...
x7000		1
x7001		2
x7002		3
x7003		4

After you compute the sum, you should store the value in RESULT. You can assume that the labels LENGTH, ARRAY, RESULT will be reachable via a PC-offset instruction with a 9-bit offset, but you cannot make the same assumption about the starting address of the array stored in ARRAY. You might notice that the memory layout is very similar to the previous question, so maybe consider resuing your code. Write your code in sum.asm.

3 Building an RPN Calculator

For this part of the project, you will be implementing a Reverse Polish Notation calculator (RPN) from scratch in LC3 assembly.

Your calculator must support the following operators:

1. Addition (+)
2. Subtraction (−)
3. Multiplication (∗)
4. Division (/)

The operands will be integers in 2's complement. You don't have to support floating point numbers.

The part of the project is further subdivided into two sub-parts. For the first sub-part, you will be implementing all of the supported operands (+, -, *, /) as subroutines. For the second sub-part, you will use these subroutines to build the calculator.

Before we lay out the specification for each part, we will discuss some background material.

3.1 Background

We are used to calculating expressions like $(3 + 4)/10$ and $5 * 8 - 20/10$ using PEMDAS, which helps us discern the order of precedence, i.e., the order in which we should carry out calculations.

Expressions of this form, in which the operator is in between the operands, are called infix expressions. This notation is commonly used in our day-to-day life because it's easy to read and parse by humans.

However, for computers, computing infix expressions poses some challenges. For simple expressions like $5 + 1 - 10$, the task is relatively straightforward: accumulate the results from left to right.

Things get more interesting when we start bringing parentheses and operators with different precedence into the picture. Let's take $5 * 8 - 20/10$. If we were to compute it from left to right, accumulating results as we go, we'll get 2 as the final answer, which is incorrect. This is because $*$ and $/$ have a higher precedence that $-$ (as per PEMDAS), and therefore stick to its operands more strongly. As a result, we must first perform $5 * 8$, then $20/10$, and only then subtract the results. Doing this gives us 38, which is the correct answer.

This is a simple example. It gets even more complicated when we have deep levels of nesting. A standard way to compute infix expressions is by writing a Pratt parser. Making you write such a parser, and then evaluate the resulting tree, in LC3 assembly, would be very mean, and I would like to think that I have a heart `<3`.

So, how do we solve this problem?

3.1.1 Leveraging the Stack

Lucky for us, there exists another notation for arithmetic and logical expressions. It's the postfix notation.

In this notation, the operators come after their operands. For example, $3 + 4$ in infix notation would be $3\ 4\ +$ in postfix notation. A very nice property of postfix notation is that we can compute the resulting expression using a stack. Allow me to blow your mind `:)`.

Let's say we want to compute $3+4$. It's corresponding postfix form is $3\ 4\ +$. To begin calculating the result, we will parse tokens (numbers or operators) starting from the left. If we encounter a number, we push it onto the stack. If we encounter an operator, we'll pop values off the stack, calculate the result of applying the operator on the values we just popped, and the push the result back onto the stack. The final result will be the value that's on top of the stack.

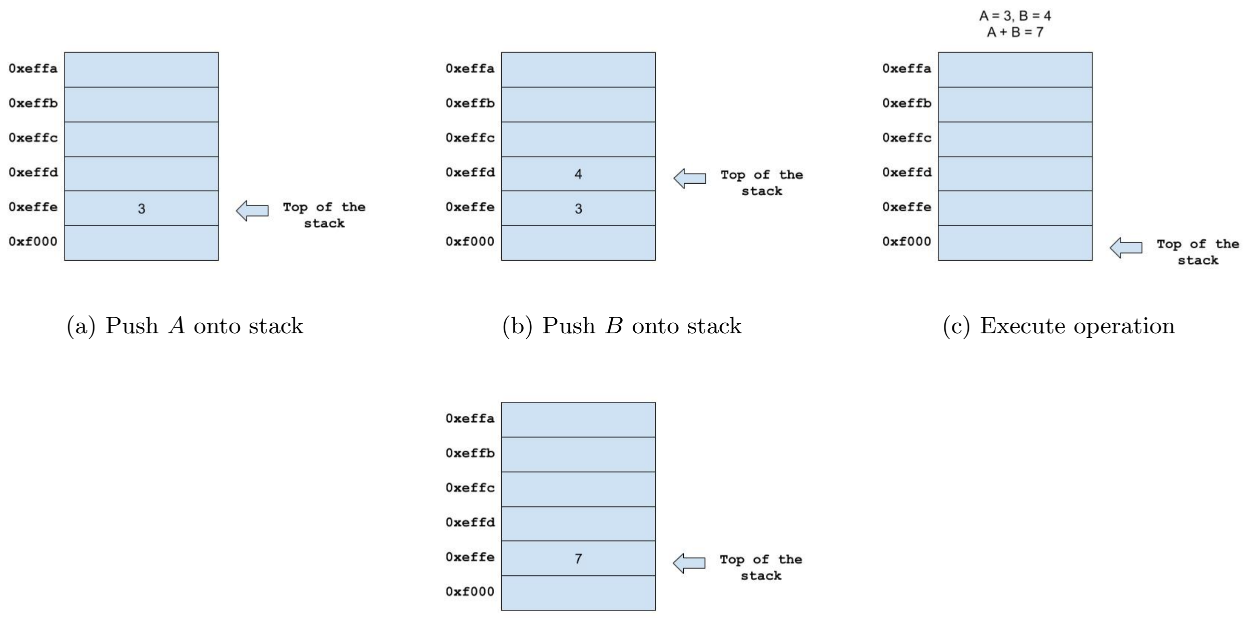
For $3\ 4\ +$, the first token is 3, which is a number. Since it's a number, we push it onto the stack (Figure 1a). Next, we find 4, which is also a number. So, we push 4 onto the the stack (Figure 1b). Then, we find $+$, which is an operator. $+$ is a binary operator, and takes in two operands. Therefore, we pop two values off the stack, carry out the addition on these two values, and then push the result back onto the stack. The two values in our case are 3 and 4. Performing the addition gives us 7 (Figure 1c). Pushing 7 results in Figure 1d. Since we don't have any more tokens to parse, we can stop. The final result 7 is on top of the stack.

Alright, so this is super cool and all, but does it solve the problem that we earlier had with infix expressions? You sure betcha! The postfix form of $5 * 8 - 20/10$ is $5\ 8\ *\ 20\ 10\ /\ -$ (Do not worry about how we got this postfix expression. Interestingly enough, you can build such an expression from the infix form using a stack). Try computing this postfix expression using the method described above. You will get 38 as your final answer, which is indeed the correct answer!

The postfix notation is also known as Reverse Polish Notation (RPN). An RPN calculator, therefore, is a stack based calculator that can compute postfix expressions.

Computerphile has a really neat video on RPN. Feel free to take a look if you need help understanding postfix notation a little more.

With background information out of the way, let's begin writing our RPN calculator!



(a) Push A onto stack (b) Push B onto stack (c) Execute operation

(d) Push $A + B$ onto stack

Figure 1: Computing (3 4 +) on the stack

3.2 Specification

3.2.1 Implementing the Operators

For this part, we will be implementing four operators:

1. Addition (+)
2. Subtraction (−)
3. Multiplication (∗)
4. Division (/)

You will write your code in `rpn.asm`.

For each subroutine in `rpn.asm`, you will find comments that document the expected API contract that you need to uphold.

For multiplication and division, we highly recommend that you use the following pseudocode as a guide when writing your assembly. Recursive implementations of these operations exist, and you can definitely solve it that way for an added challenge, but we don't test for recursion, and so you are not required to do it.

Suggested Pseudocode:

```
Multiply:
    result = 0;
    counter = B;
    negate = 0;
    if (counter < 0) {
        counter = -1 * counter;
        negate = 1;
    }
    while (counter > 0) {
        result = result + A;
        counter = counter -1;
    }
    if (negate > 0) {
        result = -1 * result;
    }

Divide:
    result = 0;
    tempA = A;
    negate = 0;
    if (tempA < 0) {
        negate = negate - 1;
        tempA = -1 * tempA;
    }
    tempB = B;
    if (tempB < 0) {
        negate = negate + 1;
        tempB = -1 * tempB;
    }
    while (tempA >= tempB) {
        result = result + 1;
        tempA = tempA - tempB
    }
    if (negate != 0) {
        result = -1 * result;
    }
```

3.2.2 Bringing it all together

Now that we have all of our operators implemented, it is time to build the RPN calculator.

You will write your code for this part in `rpn.asm` as a subroutine under the label `rpn`. The comments above the subroutine document the expected API contract that you need to uphold.

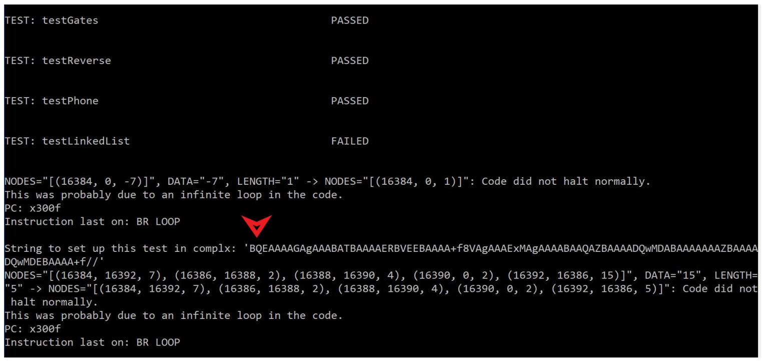
4 Debugging

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:

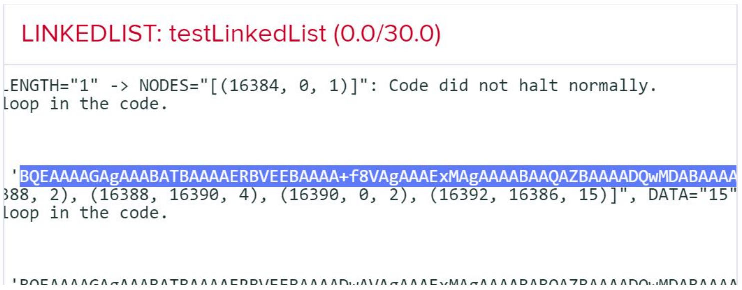
   - Mac/Linux Users:
     (a) Navigate to the directory your project is in. **In your terminal, not in your browser**
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) On **docker quickstart**, navigate to the directory your project is in
     (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
```

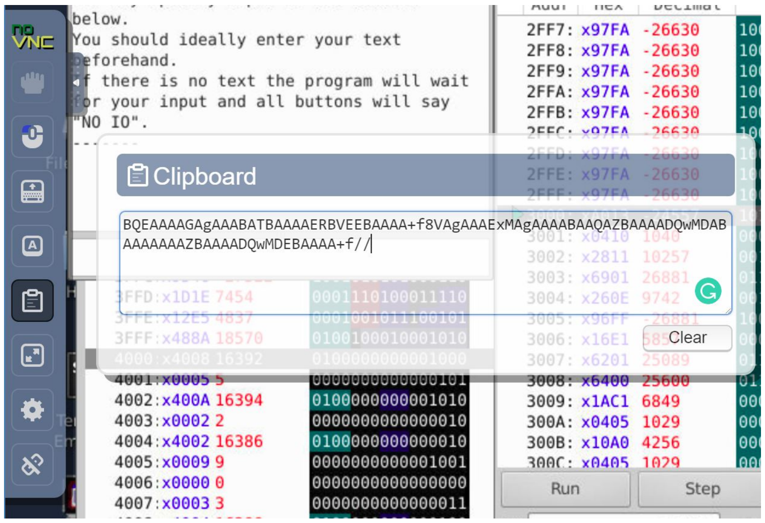
   Copy the string, starting with the leading 'B' and ending with the final backslace. Do not include the quotations.
   Side Note: If you do not have docker installed, you can still use the tester strings to debug your assembly code. In your Gradescope error output, you will see a tester string. When copying,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

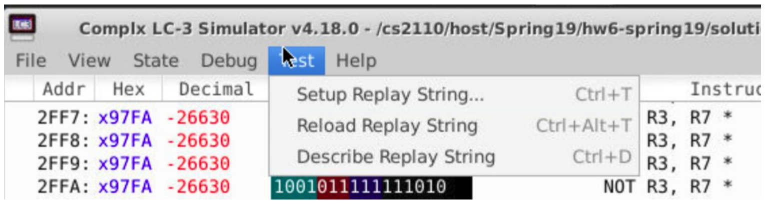

```
below.
You should ideally enter your text
beforehand.
If there is no text the program will wait
for your input and all buttons will say
"NO IO".
```

```
BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDAB
AAAAAAAZBAAAADQwMDEBAAAA+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
```

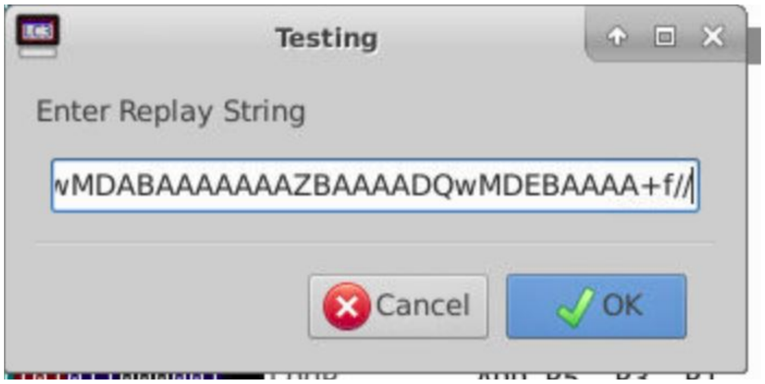
3. Next, go to the Test Tab and click Setup Replay String


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

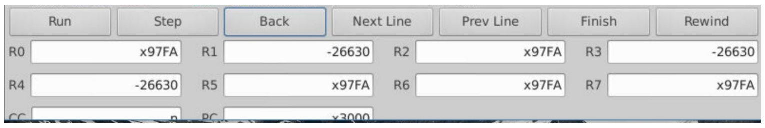
```
Addr   Hex    Decimal                                   Instruc
2FF7: x97FA  -26630                               R3, R7 *
2FF8: x97FA  -26630                               R3, R7 *
2FF9: x97FA  -26630                               R3, R7 *
2FFA: x97FA  -26630  1001011111111010         NOT R3, R7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

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



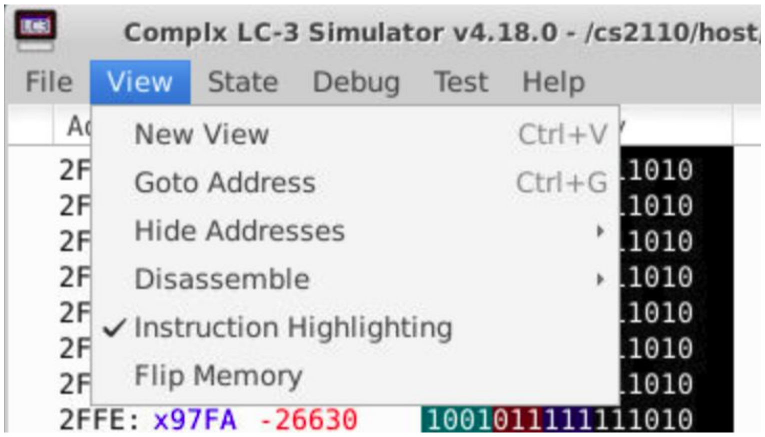
```
wMDABAAAAAAAZBAAAADQwMDEBAAAA+f//
```

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.

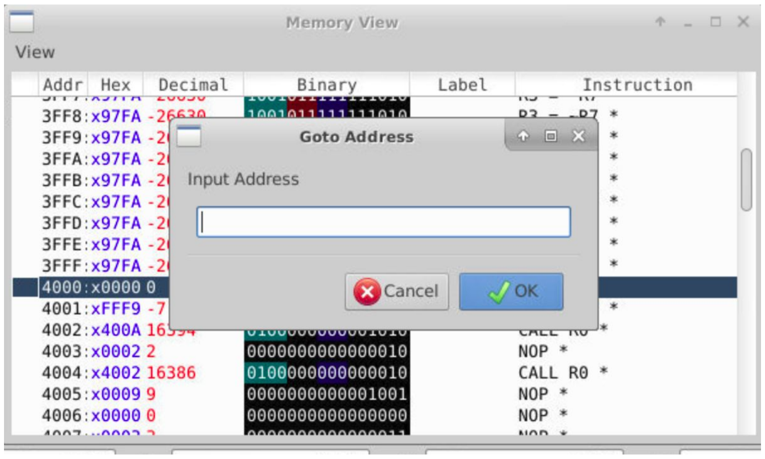


R0	x97FA	R1	-26630	R2	x97FA	R3	-26630
R4	-26630	R5	x97FA	R6	x97FA	R7	x97FA
CC	n	PC	x3000				

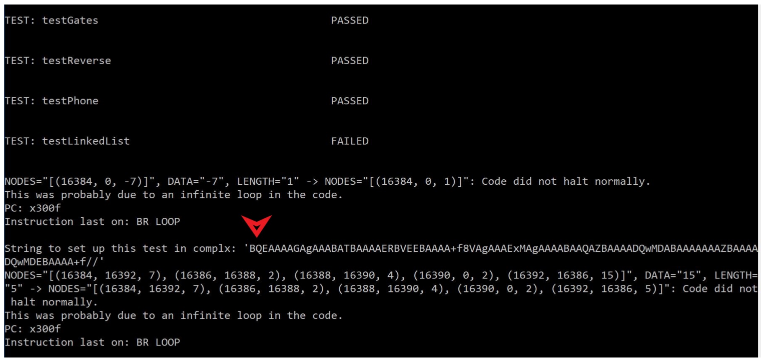
6. If you are interested in looking how your code changes different portions of memory, click the view tab and indicate 'New View'


Complx LC-3 Simulator v4.18.0 - /cs2110/host
File
View
State
Debug
Test
Help
New View Ctrl+V
Goto Address Ctrl+G
Hide Addresses
Disassemble
✓ Instruction Highlighting
Flip Memory
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
Goto Address
Input Address
Cancel
OK
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
```

5 Deliverables

Turn in the files comp.asm, mod.asm, toupper.asm, sum.asm, and rpn.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.

6 LC-3 Assembly Programming Requirements

6.1 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.
6. Use the LC-3 calling convention. This means that all local variables, frame pointer, etc... must be pushed onto the stack. Our autograder will be checking for correct stack setup.
7. Start the stack at xF000. **The stack pointer always points to the last used stack location.** This means you will allocate space **first**, then store onto the stack pointer.
8. Do NOT execute any data as if it were an instruction (meaning you should put .fills after **HALT** or RET).
9. Do not add any comments beginning with @plugin or change any comments of this kind.
10. **Test your assembly.** Don't just assume it works and turn it in.

7 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.

8 Rules and Regulations

8.1 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.

8.2 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.

8.3 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.

8.4 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.

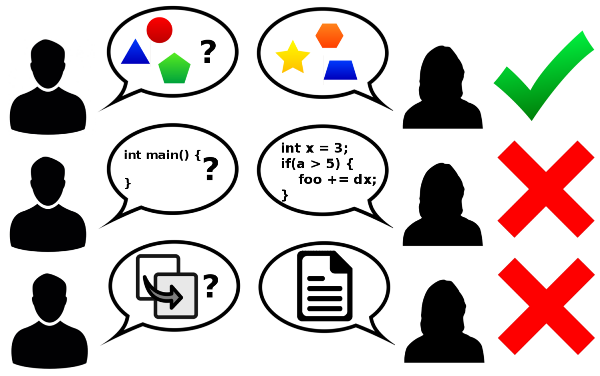
6. You must list any student with whom you have collaborated in your submission. Failure to list collaborators is a punishable offense.

7. TAs and Instructor determine whether the project is plagiarized. Trust us, it is really easy to determine this....

8.5 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 should not 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. Consider instead using a screen-sharing collaboration app, such as Bluejeans, to help someone with debugging if you're not in the same room.

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



```
int main() {
}
```

```
int x = 3;
if(a > 5) {
    foo += dx;
}
```