CS 2110 Homework 6
Introduction to LC-3 Assembly

Austin Adams, Lauren Chen, Cem Gokmen, Henry Harris, Preston Olds, Clifford Panos

Fall 2018

Contents

1 Overview

The goal of this assembly assignment is to familiarize you with coding in the LC-3 assembly language. This will involve writing small programs, modifying memory, and converting from high-level code to assembly.

2 Instructions

2.1 Part 1: DeMorgan's Revenge (`or.asm`)

For this part, notice that the LC-3 assembly language directly accommodates `AND` and `NOT` operations with corresponding instructions but not an `OR` instruction.

Use your knowledge of DeMorgan's Laws to compute `A | B`.

Open `or.asm`, and notice there are three labels. The labels `A` and `B` point to two `.fill` values to be `OR`'d.

Store the result of `A | B` in memory at the third label `ANSWER`.

2.2 Part 2: Arrays – Summing Negative Numbers (`sum_negative.asm`)

For this part, write a program which traverses an array and sums all of the negative numbers (there will also be positive numbers in the array).

Open `sum_negative.asm`, and notice there are three labels. The labels `ARRAY_ADDR` and `ARRAY_LEN` point to the starting address and length of the array.

Before `HALT`ing the program, be sure to store the result at the third label `ANSWER`.

Note: The sum will not necessarily be negative. For example, if there is overflow, you might end up with a positive number.

2.3 Part 3: Arrays – Selection Sort (`select.asm`)

For this part, write a program which sorts an array in ascending order according to the selection sort algorithm. Consider the following pseudocode:

```
for i = 0 to ARRAY_LEN - 1:          # Iterate through every position in the array
  min_idx = i                        # Find the element that will move to position i

                                     # Everything before index i is sorted, so we are
                                     # looking for the minimum value after index i
  for j = i + 1 to ARRAY_LEN:
    if ARRAY_ADDR[j] < ARRAY_ADDR[min_idx]:
      min_idx = j
                                     # Swap the value that belongs here with the current
                                     # value; these might be the same value!
                                     # (and index i might be equal to index min_idx)

  swap(ARRAY_ADDR[i], ARRAY_ADDR[min_idx])
```

Open `select.asm`, and notice that there are two labels. The labels `ARRAY_ADDR` and `ARRAY_LEN` correspond to the starting address and length of the array, respectively.

Note: This sort should be in-place, so when your program finishes, the location in memory that contained your original array should now contain the sorted array.

2.4 Part 4: Strings – Comparing (strcmp.asm)

For this part, write a program which compares two null-terminated strings, or sequences of characters, according to the following specification:

- The function should store either `1`, `0`, or `-1` at the label `ANSWER` according to whether the first string is greater than, equal to, or less than the second string
- Determine greater than, equal to, or less than by comparing their ASCII values

Consider the following examples:

```
strcmp("dogcat", catdog")    ==>  1
strcmp("whether", "whither") ==> -1
strcmp("", "blank")          ==> -1
strcmp("same", "same")       ==>  0
```

Open `strcmp.asm`, and notice there are three labels. The labels `ADDR_STR_1` and `ADDR_STR_2` correspond to the starting addresses of the first and second strings to be compared.

The process is very simple: Iterate through both strings, comparing the character values at the same indices. When you find an inequality, or one of the two characters is a zero, stop the iteration and compute the result based on those two characters. If both characters are zero, the strings are equal. If one character is zero, that string comes first because it's shorter. Otherwise, compare the differing characters, and the string with the smaller character value comes first.

Store the result at the third label `ANSWER`.

Note: Strings are zero-terminated sequences of 16-bit words. Consider the string `"ASSEMBLY"` starting at address `x4000`. The ASCII value for character `'A'` (`x0041`) is stored at `x4000`, the ASCII for character `'S'` (`x0053`) is stored at `x4001`, etc. Finally, the value `0` is stored at address `x4008`. You, of course, do not need to memorize or interpret the ASCII table to determine which value is smaller.

2.5 Part 5: Linked List – Finding Extrema (max_min.asm)

For this part, write a program that traverses a singly-linked list of `node`s and finds both the maximum and minimum elements, where each `node` is comprised of a `next` address that points to the next node and an integer `data`. Consider the following pseudocode:

```
node = HEAD_ADDR

if node == NULL:               # For an empty list, set some invalid values:
  max = MIN_INT                #   - The smallest possible number for max
  min = MAX_INT                #   - The largest possible number for min

else:                          # For a non-empty list, initialize with the head node's data:
  max = mem[node + 1]          #   Remember: mem[node + 1] is the node's integer data
  min = mem[node + 1]

  node = mem[node]             #   Remember: mem[node] is the node's next address

  while node != 0:             # For all remaining nodes in the list:
    if mem[node + 1] > max:    #   Check for a new max
      max = mem[node + 1]

    if mem[node + 1] < min:    #   Check for a new min
      min = mem[node + 1]

    node = mem[node]           # How do we get the address of the next node?
```

Once again, each `node` of this list is comprised of two consecutive 16-bit values in memory: The address of the `next` node and the `data` being stored.

Consider an example list with a `HEAD_ADDR`, or starting `node`, at address `x4000`.

Address	Contents	Comments
x4000	x4004	Node 1: next
x4001	x0035	Node 1: data
x4002	x345A	...
x4003	x7441	...
x4004	x4008	Node 2: next
x4005	x0101	Node 2: data
x4006	x0000	Node 4: next
x4007	x9040	Node 4: data
x4008	x4006	Node 3: next
x4009	x7000	Node 3: data

Observe that `Node 1` points to `Node 2` (at address `x4004`). `Node 2` points to `Node 3` (at address `x4008`). `Node 3` points to `Node 4` (at address `x4006`). Finally, `Node 4` points to address `x0000` (i.e. `NULL`), signaling the end of the list!

Our autograder will, in general, store nodes near `x4000`, so please avoid writing any instructions or storing any data near there.

Note: **The grader will only test lists with nonnegative integer data! (i.e.** `0x0000` $\cdots$ `0x7FFF`**)**

Store `max` at `ANSWER_MAX` and `min` at `ANSWER_MIN`.

3 Deliverables

Please upload the following files to the assignment on Gradescope:

1. `or.asm`
2. `sum_negative.asm`
3. `select.asm`
4. `strcmp.asm`
5. `max_min.asm`

Be sure to check your score to see if you submitted the right files, as well as your email frequently until the due date of the assignment for any potential updates.

4 LC-3 Assembly Programming Requirements

4.1 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 the code in this file 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. Try to write high-level pseudo-code instead!

   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. Do not add any comments beginning with @plugin or change any comments of this kind.

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

5 Hints

5.1 Pseudo-Ops

Pseudo-Ops are directions for the assembler that aren't actually instructions in the ISA.

1. `.orig`: Tells the assembler to put this block of code at the desired address.

   Example: `.orig x3000` will put the block of code at address `x3000`.

2. `.stringz "something"`: Will put a string of characters in memory followed by `NULL` (which is a single ASCII character with the value `0`).

   Example: `.stringz "sanjay"` will put the ASCII code for the letter `'s'` in the first memory location, the code for `'a'` in the second memory location, and so on until putting `'y'` in the next-to-last position and `NULL` (which again, has the ASCII code `0`) in the last memory location as the terminator.

3. `.blkw n`: A pseudo-op that will allocate the next `n` locations of memory for a specified label.

4. `.fill value`: A pseudo-op that will put the `value` in that memory location.

5. `.end`: A psuedo-op that denotes the end of an address block. Matches with an `.orig`.

5.2 Traps

Traps are subroutines built into the LC-3 to help simplify instructions. Some helpful TRAPS include:

1. `HALT`: `HALT` is an alias for a TRAP that will stop the LC-3 from running

2. `OUT`: `OUT` is an alias for a TRAP that will take whatever character is in `R0` and print it to the console

3. `PUTS`: `PUTS` is an alias for a TRAP that will print a string of characters with the starting address saved in `R0` until it reaches a `NULL` (`0`) character
4. `GETC`: `GETC` is another TRAP alias that will take in a character input and store it in `R0`

   Being an alias for a TRAP instruction means that the assembler will convert them to TRAP instructions at assembly time. For example, if you write HALT in your code, the assembler will convert it to the instruction, TRAP x25 for you.

5.3 Example

The following small example will demonstrate the use of these Traps and pseudo-ops:

```
.orig x3000                ; Where the code will begin
LEA R0, HW                 ; Loads the address of the string into R0
PUTS                       ; Print the string whose address is in R0
HALT                       ; Stops the program from executing
HW .stringz "Hello. \n"    ; Stores the word 'Hello' in memory with 'H' be located at
                           ; address x3003, 'e' will be located at address x3004, etc
.end                       ; Denotes the end of the address block
```

6 Rules and Regulations

6.1 General Rules

1. Starting with the assembly homeworks, any code you write must be meaningfully commented. You should comment your code in terms of the algorithm you are implementing; we all know what each line of code does.
2. 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.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.
2. 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. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.
3. Do not submit compiled files, that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.
4. 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.

6.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; 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 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

6.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

6.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own. Consider instead using a screen-sharing collaboration app, such as `http://webex.gatech.edu/`, to help someone with debugging if you're not in the same room.

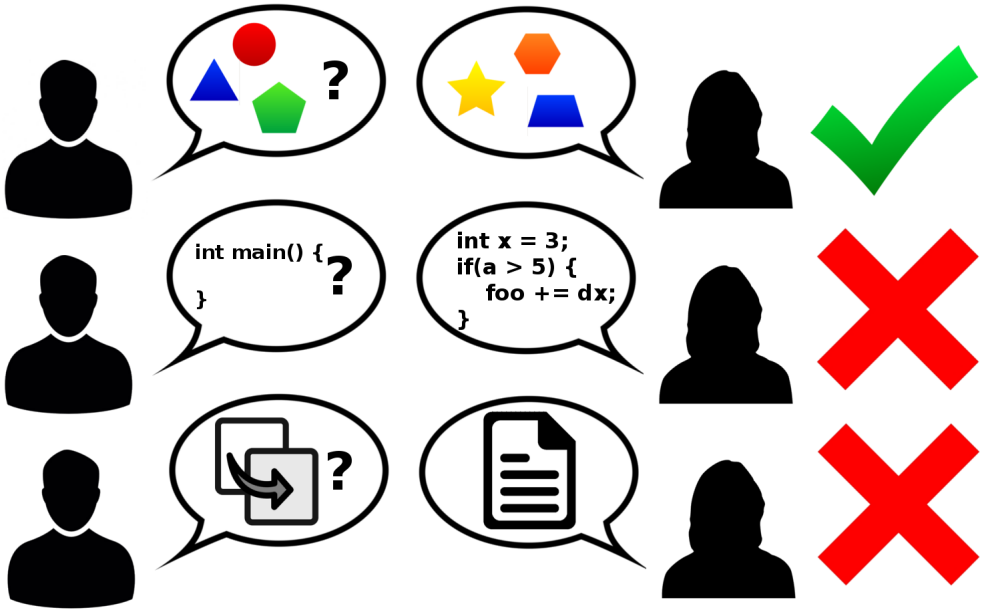

Figure 1: Collaboration rules, explained colorfully