

Programming Lab 3: Minesweeper, Part 1

EE 306: Introduction to Computing

Professor: Dr. Al Cuevas

TAs: Apurv Narkhede, Nic Key, Ramya Raj, Jerry Yang

Due: 11/12/2018 at 12pm

NB: I know this is a long lab document. If you know what subroutines, TRAPs, and linked lists are, skim the Background section - but read section 2.2.2 The Contract.

1 Overview

This lab is intended to familiarize you with subroutines, linked lists, and output using TRAP (sub)routines. By the end of this lab, you should be able to:

1. write and debug about 200 lines of assembly code across four subroutines.
2. use LC3 pseudo-ops `.FILL`, `.BLKW`, and `.STRINGZ`.
3. implement subroutines using `JSR/JSRR` (subroutine calls) and `RET` (subroutine returns).
4. address issues that arise from nested subroutines in LC3.
5. output strings to the LC3 console using TRAP subroutines `PUTS` and `OUT`.
6. traverse and search a linked list in assembly.
7. iterate through all locations of a grid by using a nested loop in assembly.

Late submissions will not be accepted; grades will be published within one week of the due date. Regrade requests will only be considered if the autograder is incorrect (see regrade policy below).

Note: You may NOT show anyone other than the TAs or Dr. Cuevas your code.

Any violation of this rule constitutes academic dishonesty.

2 Background

This lab is the first part of a two-part lab that will culminate in the game Minesweeper. For this lab, you will be writing four subroutines that constitute the “back end” of the game: displaying the board, loading the bombs, calculating bomb locations, and calculating memory locations for a given board configuration. The subsections in this section describe the key ideas you will need to implement this lab.

2.1 About Minesweeper

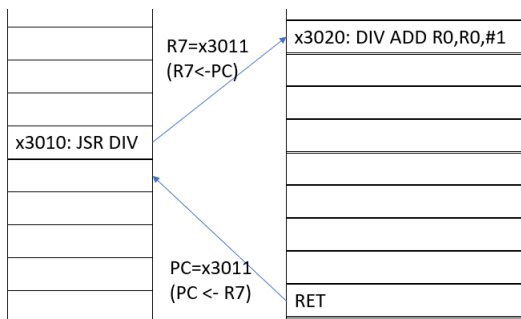
Minesweeper is a single-player game dating from the 1960s in which a player attempts to clear a board containing hidden bombs by clicking on squares that reveal the number of bombs next to it. The objective of the game is to determine the locations of all the bombs without finding a bomb.

The version of Minesweeper we will be creating will feature a 4x4 grid with an undetermined number of bombs and a score counter. You don't need to worry about the score counter for now.

2.2 Subroutines, JSR and RET

In Lab 2, you probably found yourself copying and pasting your multiply routine several times to compute several different things. This style of coding, as you may have discovered, is largely inefficient and difficult to debug - you probably wished you had a way to write the multiply only once, then use it wherever you needed it.

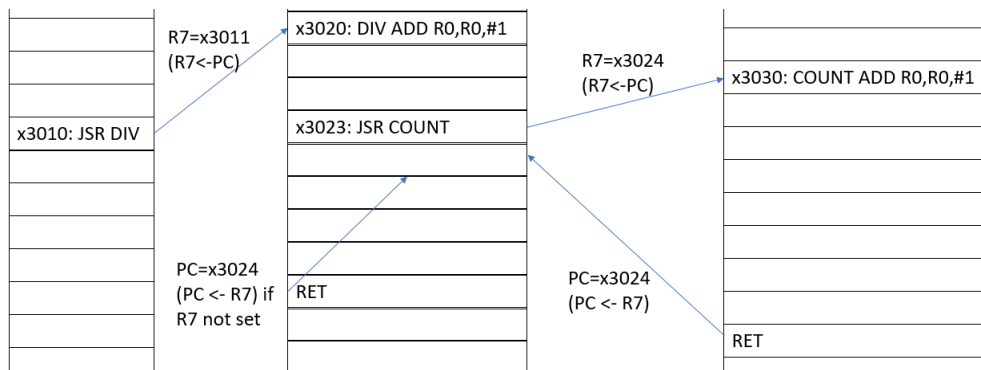
Subroutines are the answer to your wish. A subroutine is a piece of code that does one specific job in a larger program. With subroutines, you can divide a larger program into smaller parts, then write and debug them separately without knowing what the main code does.



When using a subroutine, you need to make sure of two things: 1. that you know where you are going, and 2. how you will get back to your main program and pick up where you left off. In LC3, JSR and JSRR go to or “call” a subroutine. When a subroutine is called, R7 is overwritten with the address of the next line in your main program as a placeholder for when you come back. R7 is termed the “link register,” as it links you back to your main program once your subroutine finishes. RET takes you back to, or “returns to,” your main program by loading R7 into the PC.

2.2.1 Nested Subroutines

If you call a subroutine inside another subroutine, you will destroy the value in R7. When you attempt to return to the main program, you will be redirected to somewhere you don't want in memory (see figure below).



To remedy this, you will need to save the original value of R7 in memory before calling the subroutine, then restore R7 once you return from that subroutine.

2.2.2 LC3 Calling Conventions (a.k.a the Contract) *****IMPORTANT*****

Oftentimes in industry, you will be writing code as part of a larger project, and you will never see what the larger project is. To maintain consistency in coding standards between people's code (and for the autograder), we have "calling conventions" that everyone understands and follows to ensure everyone's code works when we put it all together.

The LC3 calling conventions (the Contract) are as follows:

1. You can only destroy input registers. All other registers must be preserved i.e. the same before and after a subroutine is called.
2. Inputs and outputs of subroutines are stored in R0-R3. The first input/output is stored in R0, the second in R1, etc.
3. If you have more than 4 inputs or outputs, they go on the stack (don't worry about this for now).

If you don't follow the Contract, there is no guarantee that your code will work, and the autograder will likely count your subroutine incorrect.

2.3 TRAPs and Console Output

In LC3, TRAP instructions are special subroutines used to perform input/output operations. For now, you only need to worry about displaying output on the console using the OUT and PUTS instructions.

2.3.1 Printing a Character: OUT

The OUT trap instruction reads R0 as an ASCII character and prints it on the console. ASCII characters are hex codes for the alphabet, numbers, and special characters, and can be found via Google. To use OUT, you need to:

1. Load the character you want to print into R0.
2. Type "OUT" in your code as an instruction without operands.

For example, if you set R0=x30 and call OUT, a '0' will be printed to the console.

Fun exercise: Print the alphabet using less than 10 lines. You can do both uppercase and lowercase.

2.3.2 Printing a String: PUTS

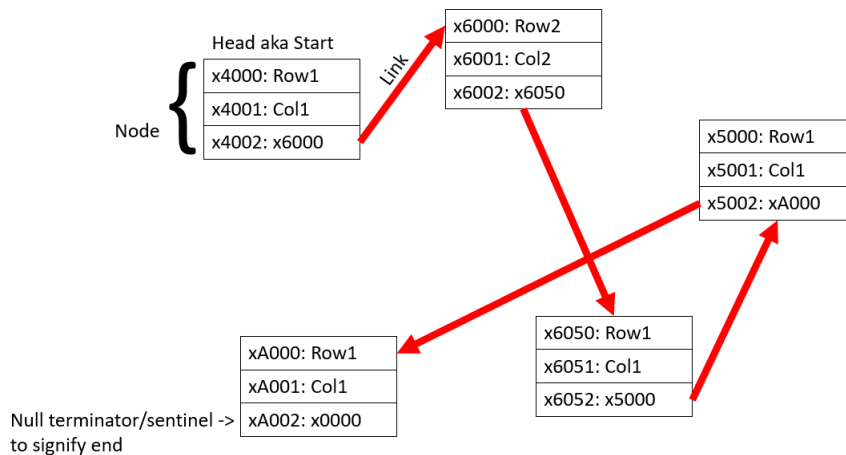
As you know, a string is stored in LC3 as an array of characters followed by a null terminator. PUTS uses a base+offset addressing mode to print your string to the console. To use PUTS, you need to:

1. Load the address of the first character of the string into R0.
2. Type “PUTS” in your code as an instruction without operands.

To see an example of this, please reference Lab 0, where I provided you a simple “Hello World” program to edit.

2.4 Linked Lists

Linked lists are one way data can be stored in memory. Instead of storing data sequentially as in an array, a linked list stores a piece of data and the address of the next piece of data. The figure below shows how data is stored in a linked list.



Each piece of data+address is called a node. The head of the list is the first piece of data of the first node. The end of the list is signified by a null terminator or sentinel - generally x0000 in the address location. When you access the data in the linked list, you will need to move from one node to another by using the addresses stored at each node - this is called “traversing” the list. To get from one node to another, you will need to:

1. Designate a register to store the address of the current node.
2. Calculate the location of the next node’s address as an offset from the top of the node.
3. Load the address of the next node into a register, and check if it is x0000.
4. Update the register storing your place in the list.

3 Lab Specifications

You will implement four key subroutines that will be useful in Lab 4. Two of the subroutines are called by my code, and two are called by your code; we will test them though.

There are two things you must NOT do:

1. You are not allowed to modify the starter code.
2. You may not use any memory location less than x3000 unless you execute a TRAP.
Doing so may mess with the autograder and ultimately give unpredictable results.

3.1 Global Inputs

There are two global inputs for this lab: a linked list storing the locations of the bombs, and the board.

3.1.1 Linked List: Bombs

The list of bomb locations will be provided to you in a linked list. Each node contains three fields, in order:

1. the row number of the bomb,
2. the column number of the bomb, and
3. the memory address of the next node.

An example input file (*Board.asm*) is provided in the starter code. *Board.asm* contains all the records in a single file linked using labels instead of addresses, instead of scattering them all over memory. This is done for your convenience; your code must work even if the linked list records are scattered across memory.

Notes:

- The address of the first record in the list will be stored in x6000. (The first record is not at x6000.)
- There may be zero or more bombs in the list. If there are zero bombs, the address of the first record will be set to x0000.
- There is no defined order for the list.

3.1.2 Grid: Board

The board is a 4x4 grid of spaces defined initially as shown. It is a set of .STRINGZ declarations located at the label GRID, as shown below.

You are not allowed to modify the grid in the starter code.

```
*****
; This is the data structure for the BOARD grid
*****
GRID      .STRINGZ "+--+--++"
ROW0      .STRINGZ "| | | |"
          .STRINGZ "+--+--++"
ROW1      .STRINGZ "| | | |"
          .STRINGZ "+--+--++"
ROW2      .STRINGZ "| | | |"
          .STRINGZ "+--+--++"
ROW3      .STRINGZ "| | | |"
          .STRINGZ "+--+--++"
*****
```

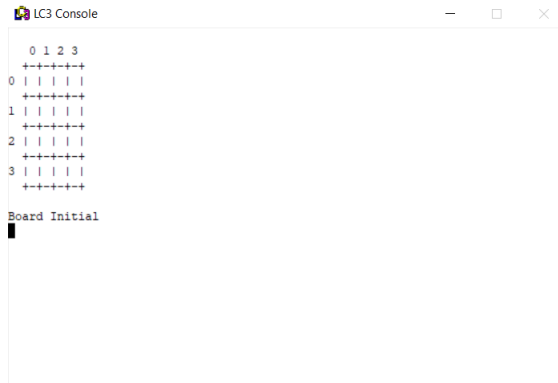
3.1.3 Starter Code

The main program (the game engine) is provided to you along with the declaration of the board and several variables. You are not allowed to modify the starter code.

3.2 Subroutine 1: DISPLAY_BOARD

- Inputs: None
- Outputs: None
- Purpose: Displays the board in the console.

The board GRID is located in memory at the label GRID and is a set of .STRINGZ declarations provided to you in the starter code. Your job is to print the grid, along with row numbers and column numbers, as shown below. To go to the next line in the console, print the ASCII character “\n” to the console.



```
0 1 2 3
+++++
0 | | | |
+++++
1 | | | |
+++++
2 | | | |
+++++
3 | | | |
+++++
Board Initial
```

Note that the grid does not contain the row and column numbers, and you cannot modify the grid. The row and column numbers are not included in the GRID, so you will need to create other strings as needed. The row of column numbers has a space at the end of the line i.e. is printed as “ 0 1 2 3 ” instead of “ 0 1 2 3”.

3.3 Subroutine 2: LOAD_BOARD

- Inputs: R0 - Address of the head of the linked list of bombs, x0000 if linked list has zero nodes.
- Outputs: None
- Purpose: Loads the contents of the board by inserting bombs (*) or ASCII numbers into the grid

This subroutine reads the input linked list and populates the contents of the board by inserting bombs (*) or ASCII numbers into the appropriate locations on the grid (see section 3.1 on global inputs). The routine undergoes two steps:

1. Stores the bombs (*) into the grid at the memory location given by GRID_ADDRESS.
2. For all non-bomb spaces on the grid, it calculates the number of bombs near the space, converts it into an ASCII number, and stores it in the grid. See the COUNT_BOMBS subroutine for the definition of “near.”
3. If there are no bombs near the space, it does not enter anything in the space.

For both steps, you will need to call `GRID_ADDRESS`, and for the second step, you will need to call `COUNT_BOMBS`. Both of these subroutines are described below. After calling `LOAD_BOMBS`, a call to `DISPLAY_BOARD` should generate console output that looks like this:

```

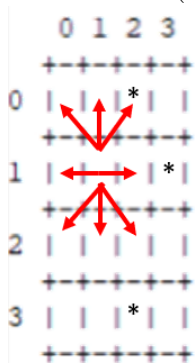
  0 1 2 3
  +--+--+
0 |1|3|*|2|
  +--+--+
1 |2|*|*|2|
  +--+--+
2 |3|*|4|1|
  +--+--+
3 |2|*|2| |
  +--+--+

```

3.4 Subroutine 3: COUNT_BOMBS

- Inputs: R0 - Address of the head of a linked list of bombs; R1 - Row number [0,3]; R2 - Column number [0,3]
- Outputs: R0 - Number of bombs, returns -1 if the location contains a bomb
- Purpose: Counts the number of bombs near a given location.
- Note: You may assume that the location indicated by the inputs is valid i.e. inside the board.

To implement this subroutine, you will need to traverse the linked list and determine if each bomb is near the desired location. “Near” is defined as within one space of the desired space either horizontally, vertically, or diagonally. In the figure below, `COUNT_BOMBS` will return 1 for the coordinates (1,1) since there is a bomb at (0,2).



```

  0 1 2 3
  +--+--+
0 | | |*| |
  +--+--+
1 | |*|*| |
  +--+--+
2 | | | | |
  +--+--+
3 | | |*| |
  +--+--+

```

Note that this subroutine does not require a call to `GRID_ADDRESS`.

3.5 Subroutine 4: GRID_ADDRESS

- Inputs: R1 - Row number [0,3]; R2 - Column number [0,3]
- Outputs: R0 - Address of corresponding GRID space in memory
- Purpose: Translates the (row,column) logical GRID coordinates of a location to the physical address in the GRID memory.

For this subroutine, you will have to do some math to figure out how each space corresponds to a location in memory. For example, (0,0) on the grid corresponds to x3035 in memory, and (3,3) corresponds to x3077.

4 Testing Your Program

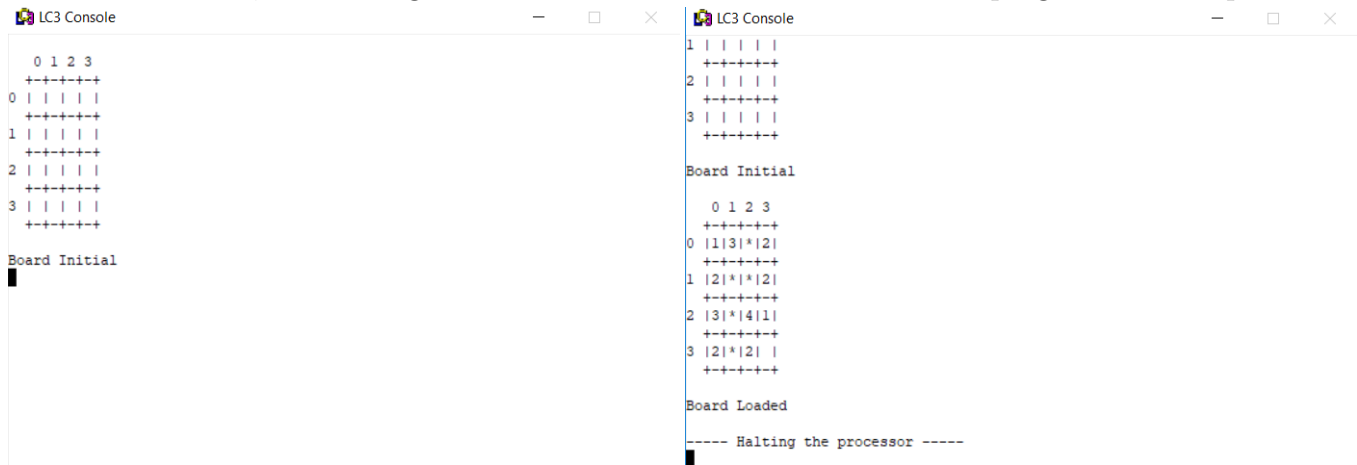
4.1 Loading Your Program into the Simulator

Here are the steps to load the board and the program into the simulator:

1. Load *Board.asm* into the simulator as you would a program. The PC will set to x6000; that's fine.
2. Load *Lab3.asm* into the simulator as you would a program. The PC should reset to x3000, and you can run/step through/debug your program like normal.

4.2 Expected Output

Here are screenshots of a sample run of a working solution. The left screenshot shows the initial state of the board, and the right screenshot shows the console after the program has completed.



When we test your program, we will test each subroutine individually, as well as your entire code altogether. We will also generate different linked lists for each test case; some test cases will have the same linked list, while others will use different linked lists. As such, you should write your own linked lists to test your code with.

Remember to check if you followed the Contract (see section 2.2.2)!

5 Hints, Tips, and Tricks

1. The document gives some hints for each subroutine in its corresponding section.
2. Debug each subroutine separately.
3. You don't have to finish one subroutine to start on another. In fact, if you do this, it may be harder to debug.
4. Your program will get too long to where you won't be able to access the board in memory anymore. One solution is to store the address of the board at intermediate points in your code.

6 Submission

1. Make sure you fill out the starter file header with your name, EID and recitation section time. Do NOT delete any information in the starter file, including comments.
2. Check your syntax to ensure it will work with the autograder. (See autograder syntax document on Canvas.)
3. Rename the Lab3.asm file “EIDLab3.asm”, replacing “EID” with your EID.
4. Submit the Lab3.asm file (and ONLY the Lab3.asm file) to the Lab 3 Canvas assignment. Do NOT include any other files.
5. If you do not follow these submission instructions exactly, the autograder will spit out a 0!

7 Regrade Policy

We will only award points back if the **autograder** graded your code incorrectly, i.e. your code generated the correct output, but the autograder said you failed the test case. Unfortunately, we will NOT award points back if you fail all the test cases because of a single mistyped character.

If you wish to submit a regrade request, complete the following steps:

1. Review the autograder output, uploaded as a submission comment on your lab submission.
2. Visit Dr. Cuevas or a TA (preferably Jerry) in office hours to confirm that the autograder output is incorrect **AND** that your code produces the correct output.
3. Email Jerry at jerryyang747@utexas.edu with the following:
 - Subject line: “EE 306: Lab 3 Regrade Request [EID]”
 - Screenshot of the autograder output
 - A description of what went wrong
 - Who you confirmed the error with
 - Your code, attached as an .asm file

After grades are released on Canvas, you have **7 days** to submit a regrade request.