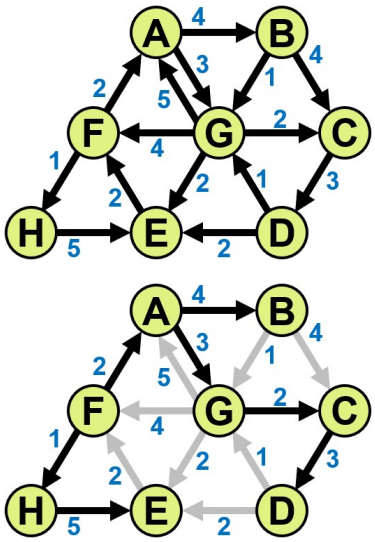
ECE220: Computer Systems & Programming **Spring 2024 ZJUI**
Machine Problem 1 **due: Saturday 27 January, 11:59:59 p.m.**

Printing Subroutines



Your task this week is to write three subroutines to support finding and printing the shortest paths in a graph using Dijkstra's single-source shortest path algorithm. Don't fret, though! The algorithm itself will be MP3. In this first assignment, you need only write a few subroutines that you can use later. The first subroutine, FIND_PARENT, performs a simple arithmetic computation. The second subroutine walks backwards along a shortest path, pushing graph node indices onto a stack. Looking at the example of a graph (upper right) and the shortest paths in that graph starting from node F (lower right), the second subroutine, when asked to push the path from F to C, pushes first C, then G, and A, then F, all on to a stack. Finally, the third subroutine prints a sequence of graph nodes from a stack to the display using a specific format. Using the same example, the third subroutine, given the values pushed on to a stack by the second subroutine, pops those values off and prints the string `F->A->G->C` followed by a line feed (ASCII character #10) to the display. Together, the three subroutines require about 100 lines of LC-3 assembly, including comments. In the next two MPs, you will make use of these subroutines to accept any graph, such as the example shown here, and produce a printed list of all shortest paths starting from any node in the graph, as shown to the right when starting with node F in the example graph.


```
F->A
F->A->B
F->A->G->C
F->A->G->C->D
F->H->E
F
F->A->G
F->H
```

The objective for this week is to give you some experience with decomposing tasks into algorithms, with writing LC-3 assembly code, with using stacks, and with formatting output. We will make an effort to use different words when referring to parts of different data structures, but please note that should you choose to look at any outside material, these sources are likely to use the same words with different meanings.

The Task

The first subroutine is FIND_PARENT. A positive 2's complement number is passed to your subroutine in R0. The number represents the index of an element in a heap, which Dijkstra's algorithm uses to decide which graph node to explore next. The parent index ***P*** of a heap element with index ***E*** is given by the expression $P = \lfloor (E-1)/2 \rfloor$, where the brackets indicate that the quotient is rounded down to the nearest integer. Your subroutine must compute the value of ***P*** and return it in R1, preserving all register values except for R1 and R7 (the return address).

The second subroutine is FILL_STACK. Graph nodes (the circles in the example graph) are represented by non-negative numbers—for example, node A is 0, node B is 1, and so on. The number representing the last node in a path is passed to your subroutine in R1 as a non-negative 2's complement value. A table containing each graph node's predecessor in the path is also provided. The table starts at memory address x5000 and is indexed by node number. An example is provided on the next page. The example corresponds to the shortest paths from node F (node number 5) in the graph shown above. A pointer to the top of a stack is also provided to your subroutine in R6.

Your subroutine must start by pushing the value -1 on to the stack to serve as a marker for the end of the path (see PRINT_PATH below). After doing so, your subroutine must push each node number in the path onto the stack. For example, if the value 2 (node C) were passed in R1, your subroutine must push 2 onto the stack, then find the previous node by reading address x5002 (x5000 + 2) to obtain the value 6 (node G). The subroutine must repeat this process, pushing 6 and reading 0 from address x5006, pushing 0 and reading 5 from address x5000, and finally pushing 5 and reading -1 from address x5005. The -1 value means that your subroutine has reached the first node in the path and is done. FILL_STACK must preserve the values of all registers other than R6 (the stack must now contain the graph nodes) and R7, the return address. You may assume that the table is complete and correct—in other words, that all node numbers encountered by a correctly written subroutine have predecessor nodes in the table, and that all paths are properly terminated (with a -1 predecessor).

Address	Contents	Meaning
x5000	x0005	F
x5001	x0000	A
x5002	x0006	G
x5003	x0002	C
x5004	x0007	H
x5005	xFFFF	(none)
x5006	x0000	A
x5007	x0005	F

The third subroutine is PRINT_PATH, which prints a path of node numbers stored on the stack to the display, popping the node numbers from the stack. Since node numbers are non-negative, a final value of -1 is also placed on the stack (by your FILL_STACK subroutine) to indicate that the path has ended. Your PRINT_PATH subroutine must also pop this sentinel value (the -1) from the stack. Nodes should be printed to the display using their ASCII node names. So node 0 should be printed as A, node 1 as B, and so forth. Between each pair of nodes in the path, your subroutine must print the characters "**->**". Finally, after printing the whole path, your subroutine must print a line feed (ASCII character #10) to the display. PRINT_PATH must preserve the values of all registers other than R6 (the path and sentinel must have been popped from the stack) and R7, the return address. You may assume that all node numbers are valid and that the path is properly terminated with a -1 value on the stack.

As you develop your subroutines, you may want to include code to test them at the start of the assembly file. Since the LC-3 will begin execution at the start of the file by default, you can then easily test your subroutines with the LC-3 tools.

Specifics

- Your code must be written in LC-3 assembly language and must be contained in a single file called **`mp1.asm`**. We **will not grade** files with any other name.
- Do not make any assumptions unless they are stated explicitly to you in this document.
- Your subroutine for computing a heap element's parent's index must be called FIND_PARENT.
    - You may assume that R0 holds a positive 2's complement value when your subroutine is called.
    - Your subroutine must compute the value $P = \lfloor (E-1)/2 \rfloor$, where E is the value passed in R0, and return the value P in R1.
    - Your subroutine must preserve all register values other than R1 and R7.
- Your subroutine for pushing a path to a stack must be called FILL_STACK.
    - You may assume that R1 holds a non-negative 2's complement value in the range 0 to 10,000 inclusive.
    - You may assume that the stack to which R6 points has enough room to hold both the sentinel and all nodes in the path.
    - You may assume that the table provided starting at memory location x5000 holds valid node numbers and that the path pushed by a correctly written subroutine terminates with a -1 predecessor value.
    - Your subroutine must push a -1 value on to the stack, then push each node in the path starting from the last (provided in R1) and ending with the first (with predecessor -1).
    - Your subroutine must change R6 appropriately to reflect the new contents on the stack.
    - Your subroutine must preserve all register values other than R6 and R7.
- Your subroutine for printing a path from a stack must be called PRINT_PATH.
    - You may assume that all values from the top of the stack (provided in R6) down to the sentinel value of -1 are in the range 0 to 25 inclusive.
    - You may use either TRAP instructions or direct I/O register accesses to write output to the display, as you prefer.
    - Your subroutine must remove the entire path from the stack, including the -1 sentinel.
    - Your subroutine must print a single ASCII character corresponding to the node number plus 'A' (ASCII character #65) to the display for each node number popped from the stack.
    - Between adjacent node letters, your subroutine must print the two-character sequence "`->`". **You must NOT print the sequence after the last node in the path.**
    - After printing the last node's letter, your subroutine must print a single line feed character (ASCII character #10) to the display.
- Your code must be well-commented, must include a header explaining each subroutine, and must include a table describing how registers are used within each subroutine. Follow the style of examples provided to you in class and in the textbook.
- You may leave any code that you have used for testing at the start of your file, provided that it does not in any way interfere with your subroutines' functionality.
- None of your subroutines may access memory (neither to read nor to write) outside of your code and the regions provided for your subroutines' use (nothing for FIND_PARENT, the stack and the table for FILL_STACK, and the stack for PRINT_PATH).
- None of your subroutines may produce any output to the display other than what is specified in this document. Also, none of your subroutines may read input from the keyboard.
- All subroutines must operate correctly even when called many times without reloading your code.

Testing

You are responsible for testing your code to make sure that it executes correctly and follows the specification exactly.

To help you get started, we have provided a block of code (in **`test1.asm`**) that shows you how to check an example execution of FIND_PARENT. **Read the instructions** in the file on how to use it with your code. We have also provided an example table for FILL_STACK (the example earlier in this document) with eight nodes, along with an example (**`test2.asm`**, **`script2`**, and **`out2`**) of how to use it. Finally, we have given you a sample test file for PRINT_PATH (**`test3.asm`**).

Note that these files are meant as examples. You should create more tests to ensure that your code works correctly.

You will also find it useful to step through your code in the simulator and inspect the state after each step to make sure that your code does exactly what you intended.

Grading Rubric

Functionality (65%)

- FIND_PARENT
  - 10% - produces correct output in R1 for any positive number provided in R0
  - 5% - preserves all registers except R1 and R7 when called
- FILL_STACK
  - 5% - pushes sentinel to the stack first
  - 10% - pushes correct sequence and number of values to the stack
  - 5% - preserves all registers except R6 and R7 when called
- PRINT_PATH
  - 15% - prints the correct output string
  - 10% - pops the correct number of values from the stack, including sentinel
  - 5% - preserves all registers except R6 and R7 when called

Style (10%)

- 10% - PRINT_PATH holds 'A' (ASCII character #48) in a register to simplify printing.

Comments, Clarity, and Write-up (25%)

- 5% - each subroutine has a paragraph explaining the subroutine's purpose and interface (these are given to you; you just need to document your work)
- 10% - each subroutine includes a table explaining how each register is used
- 10% - code is clear and well-commented

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if your code does not assemble, you will receive no functionality points. If you name a subroutine incorrectly, you will receive no points for it, whether it works or not. Note also that the remaining LC-3 MPs (two of them) will build on these subroutines, so you may have difficulty testing those MPs if your code does not work properly for this MP.

Finally, we will deduct **10 points** if the LC-3 VSCode extension developed by former ZJUI student Qi Li (look in the marketplace—you'll see his name) gives **even a single warning** on your assembly code. Fix them. (There is one exception in this assignment—if you do not include test code, the plugin will warn you about your first subroutine executing RET—that is ok.)