CS061 – Lab 09

Stacks

1 High Level Description

The purpose of this lab is to investigate the innermost intricacies of the stack data structure by implementing one and then using it in a “simple” application.

2 Our Objectives for This Week

1. Understand what a stack is
2. Understand how to PUSH onto and POP from a stack
3. Exercise 1 ~ Implement PUSH
4. Exercise 2 ~ Implement POP
5. Exercise 3 ~ Use Ex1 and Ex2 to build a Reverse Polish Notation Calculator

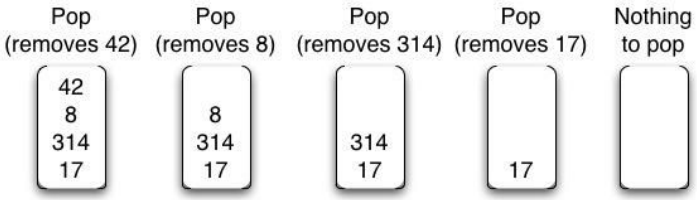
3. 1 What is this “stack” you speak of?!

High Level:
A stack is a data structure that stores items in a LIFO order - "**L**ast **I**n **F**irst **O**ut".
In other words, the last item you put onto your stack is the first item you can take out. Unlike an array, you cannot take items out of a stack in any order you like, nor can you put items into a stack in any order you like.

LIFO Analogy:
Imagine that you had a Pez dispenser. Each candy is inserted into the Pez dispenser and falls to the bottom. Whenever you want to extract a piece of candy from the dispenser, you can only take the one on top (that last one that you put in). You cannot get the first piece of candy back without taking the second piece of candy out of the dispenser first. Thus, the **last** piece of candy that you put **in**to the dispenser must be the **first** piece of candy you take **out**.

Visual Stack Example:
Below is a numerical example. Let’s say you are given the numbers {17, 314, 8, 42}. You can PUSH these onto a stack in the manner depicted below.

Now that your stack has a bunch of items in it, you can take them out by calling POP on the stack—remember, they have to come out in LIFO order. Note that the order of items popped is always the reverse *of the PUSH order*. Popping is visually depicted below.



Notice how the PUSH order was {17, 314, 8, 42} but the POP order was {42, 8, 314, 17}.

Important Stack Lexicon:
Definition: **Overflow**
To overflow a stack is to try to PUSH an element onto it when it is full.
Definition: **Underflow**
To underflow a stack is to try to POP an element off of it when it is empty.

How to implement a stack in LC3:

The easiest way to implement a stack that checks for overflow and underflow is to use the following specifications:

Specs for a stack located from xA001 to xA005 *(i.e. a total of 5 "slots")*

- A stack structure consists of three members:
  1. BASE: A pointer to the bottom of the stack (say, R4) - actually, to *1 slot below the lowest available address*, in this case **BASE = xA000**
  2. MAX: The "highest" available address in the stack (say, R5), in this case **xA005**
  3. TOS: A pointer to the *current* top of the stack (say, R6); in this case, for the empty stack, **TOS starts at BASE = xA000**

- To PUSH a value, X, onto a stack:
  - Verify that TOS is less than MAX (if not, print Overflow message & quit)
  - Increment TOS
  - Write X to Mem[TOS]

- To POP off of a stack:
  - Verify that TOS is greater than BASE (if not, print Underflow msg & quit)
  - Read Mem[TOS] to a register
  - Decrement TOS

Note that when pushing, we *first increment*, then write; while when popping, we first read, *then decrement*.

Setup/Initialization of a 5-slot stack:

- Set BASE to xA000
- Set MAX to xA005
- Set TOS = BASE = xA000 *(i.e. stack starts out empty)*

Note that you don't have to actually "remove" anything when you POP; all you have to do is decrement TOS after reading. Any PUSH you do later will simply overwrite whatever was there previously.

Exercise 01: Stack PUSH

1. Write the following subroutine:

```
;-------------------------------------------------------------------------------------
; Subroutine: SUB_STACK_PUSH
; Parameter (R0): The value to push onto the stack
; Parameter (R4): BASE: A pointer to the base (one less than the lowest available
;                       address) of the stack
; Parameter (R5): MAX: The "highest" available address in the stack
; Parameter (R6): TOS (Top of Stack): A pointer to the current top of the stack
; Postcondition: The subroutine has pushed (R0) onto the stack (i.e to address TOS+1).
;                If the stack was already full (TOS = MAX), the subroutine has printed an
;                overflow error message and terminated.
; Return Value: R6 ← updated TOS
;-------------------------------------------------------------------------------------
```

Test Harness:

To ensure that your subroutine works, write a short test harness. Make sure your stack stores the values as expected in memory and prints an overflow error message as necessary.

Exercise 02: Stack POP

Write the following subroutine:

```
;-------------------------------------------------------------------------------------
; Subroutine: SUB_STACK_POP
; Parameter (R4): BASE: A pointer to the base (one less than the lowest available
;                       address) of the stack
; Parameter (R5): MAX: The "highest" available address in the stack
; Parameter (R6): TOS (Top of Stack): A pointer to the current top of the stack
; Postcondition: The subroutine has popped MEM[TOS] off of the stack.
;                If the stack was already empty (TOS = BASE), the subroutine has printed
;                an underflow error message and terminated.
; Return Value: R0 ← value popped off of the stack
;               R6 ← updated TOS
;-------------------------------------------------------------------------------------
```

Test Harness:

To make sure your subroutine works, write a short test harness. Make sure your top and capacity values update as expected and that overflow/underflow error messages print appropriately.

Exercise 03: Reverse Polish Calculator

Reverse Polish Notation (RPN) is an alternative to the way we write mathematical expressions. When we want to add two numbers together, we usually write, “12 + 7” and get 19 as a result. In RPN, to express the same thing, we write, “12 7 +” and get 19 - i.e. we first specify the two operands, then the operation to be performed on them. In this exercise, you will implement a Reverse Polish Notation Calculator that performs a single multiplication.

Subroutine (write me!)

```
;-------------------------------------------------------------------------------------
; Subroutine: SUB_RPN_MULTIPLY
; Parameter (R4): BASE: A pointer to the base (one less than the lowest available
;                       address) of the stack
; Parameter (R5): MAX: The "highest" available address in the stack
; Parameter (R6): TOS (Top of Stack): A pointer to the current top of the stack
; Postcondition: The subroutine has popped off the top two values of the stack,
;                multiplied them together, and pushed the resulting value back
;                onto the stack.
; Return Value: R6 ← updated TOS address
;-------------------------------------------------------------------------------------
```

Your program must do the following:

1. Get a single-digit number from the user and push it onto the stack.
2. Get another single-digit number from the user and push it onto the stack.
3. Get the operation symbol (in this case, a “*”) and disregard it since we are only implementing multiplication.
4. Call the SUB_RPN_MULTIPLY subroutine to pop the top two values off the stack, multiply them, and push the result back onto the stack.
5. POP the last value off of the stack and print it out to the console in decimal format

Hints:

- You will need to use the following subroutines to get the job done:
  - SUB_STACK_PUSH (exercise 01)
  - SUB_STACK_POP (exercise 02)
  - SUB_RPN_MULTIPLY
  - SUB_MULTIPLY (go check lab 2!)
  - SUB_PRINT_DECIMAL (already written in previous assignments/labs)
- You will need to create a subroutine that performs numerical multiplication.
- You already have a subroutine that prints a value in decimal notation. Use it! ☺
- Don’t panic! Most of the work is already done at this point.
  - SUB_RPN_MULTIPLY consists almost entirely of making calls to subroutines you already have.

3.5 Submission

Add, commit, and push your lab09_ex1.asm through lab09_ex3.asm files to your lab 9 GitHub repo.

4 So what do I know now?

... More than you ever really wanted to know about Assembly Language :)