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

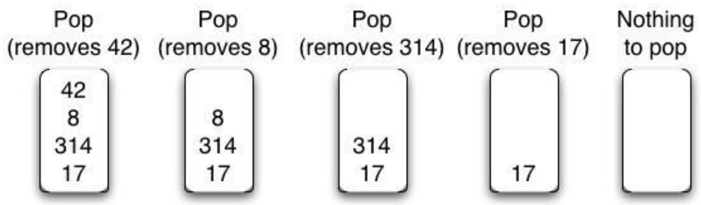
What is this “stack” you speak of?!

High Level:
A stack is a data structure that stores items in a LIFO manner. “LIFO” is an acronym that stands for **L**ast **I**n **F**irst **O**ut. What this mean is, 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 <u>reverse</u> *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:

- A stack structure consists of three members:
  1. stack_addr: A pointer to the beginning of the stack (say, R1)
  2. top: A pointer to the next available place to PUSH an item (say, R2)
  3. capacity: The number of items that the stack currently has room for (say, R3)

- To PUSH a value, X, onto a stack:
```
Verify that (capacity > 0) (else, print an overflow error message)
MEM[top] ← X (which store instruction should you use?)
top = top + 1
capacity = capacity – 1
```

- To POP off of a stack:
```
Verify that (top != stack_addr) (else, print an underflow error message)
top = top – 1
capacity = capacity + 1
```

Setup/Initialization:

- stack_addr should be set to wherever you want the stack to be stored in memory.
- top is initialized to the same value as stack_addr in the beginning.
- capacity should be initialized to the maximum size of the stack

Note that you don’t have to actually remove anything when you POP; all you have to do is decrement top and increment capacity. Any PUSH you do later will automatically 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 (R1): stack_addr: A pointer to the beginning of the stack
; Parameter (R2): top: A pointer to the next place to PUSH an item
; Parameter (R3): capacity: The number of additional items the stack can hold
; Postcondition: The subroutine has pushed (R0) onto the stack. If an overflow
;                occurred, the subroutine has printed an overflow error message
;                and terminated.
; Return Value: R2 ← updated top value
;               R3 ← updated capacity value
;-----------------------------------------------------------------------------------------------
```

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 (R1): stack_addr: A pointer to the beginning of the stack
; Parameter (R2): top: A pointer to the item to POP
; Parameter (R3): capacity: The # of additional items the stack can hold
; Postcondition: The subroutine has popped MEM[top] off of the stack.
;                If an underflow occurred, the subroutine has printed an
;                underflow error message and terminated.
; Return Value: R0 ← value popped off of the stack
;               R2 ← updated top value
;               R3 ← updated capacity value
;-----------------------------------------------------------------------------------------------
```

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 write, “12 + 7” and get 19 as a result. In RPN, to express the same thing, we write, “12 7 +” and get 19.

In this exercise, you will implement a Reverse Polish Notation Calculator that performs a single multiplication.

Subroutine (write me!)

```
;------------------------------------------------------------------------------------------
; Subroutine: SUB_RPN_MULTIPLY
; Parameter (R1): stack_addr
; Parameter (R2): top
; Parameter (R3): capacity
; 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: R2 ← updated top value
;               R3 ← updated capacity value
;------------------------------------------------------------------------------------------
```

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 (already written)
    - SUB_STACK_POP (already written)
    - SUB_RPN_MULTIPLY
    - SUB_MULTIPLY (already written)
    - SUB_PRINT_DECIMAL (already written)
- You already have a subroutine that performs numerical multiplication. Use it! ☺
- 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 four calls to subroutines you already have.