# CS061 – Lab 6

**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

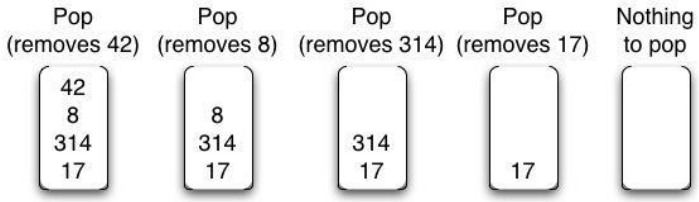
### 3. 1 What is this “stack” you speak of?!

**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.

### 3. 2 How to implement a stack in LC3:

The easiest way to implement a stack that checks for overflow and underflow is as follows:

**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 (we'll use R4) - actually, to *1 slot below the lowest available address*, in this case **BASE = xA000**
  2. MAX: The "highest" available address in the stack (R5), in this case **xA005**
  3. TOS: A pointer to the *current* **T**op **O**f the **S**tack (R6);
     in this case, for the empty stack, **TOS starts at BASE = xA000**

- To **PUSH** a value onto a stack *(we will push the value in R0)*:
  - Verify that TOS is less than MAX (if not, print Overflow message & quit)
  - Increment TOS
  - Write the desired value to the Top Of Stack: **`Mem[TOS] <- (R0)`**

- To **POP** a value off a stack *(we will "capture" it in R0)*:
  - Verify that TOS is greater than BASE (if not, print Underflow message & quit)
  - Copy the value at the Top Of Stack to the destination register: **`R0 <- Mem[TOS]`**
  - Decrement TOS

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

*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.*

### 3.3 Setup/Initialization of a 5-slot stack:

- Set R4 = BASE to xA000
- Set R5 = MAX to xA005
- Set R6 = TOS to BASE = xA000 *(i.e. stack starts out empty)*

**Exercise 1: 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.
Build an array of values *(numbers, ascii characters, whatever)* in local memory that you can use to push & pop.
Make sure those values are stored as expected in your stack (use the simulator in LC3-Tools to explore the state of the stack after each push & pop), and prints an overflow error message as necessary.

**Exercise 2: 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 and copied it to R0.
;                If the stack was already empty (TOS = BASE), the subroutine has printed
;                an underflow error message and terminated.
; Return Values: R0 ← value popped off the stack
;               R6 ← updated TOS
;-------------------------------------------------------------------------------------
```

Test Harness:

Add a call to the POP subroutine to your test harness from exercise 1. Make sure your TOS value updates

as expected, and that overflow/underflow error messages print appropriately.

**Exercise 3: Reverse Polish Calculator**

Reverse Polish Notation (RPN) is an alternative to the way we usually write arithmetic expressions. When we want to add two numbers together, we usually write, "1 + 7" and get 8 as a result. In RPN, to express the same thing, we write, "1 7 +" and get 8 - 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 operation, multiplication. (Don't worry about two digit outcomes, we'll only test 1 digit results during the demo)

Subroutine (write me!)

```
;-------------------------------------------------------------------------------------
; Subroutine: SUB_RPN_ADDITION
; 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,
;                added them together, and pushed the resulting value back
;                onto the stack.
; Return Value: R6 ← updated TOS address
;-------------------------------------------------------------------------------------
```

Your main program must do the following:

1. Prompt user for a single digit numeric character, and convert it to a number in R0 *(if you wish, you can invoke a helper s/r to do this)*; then invoke PUSH s/r to push R0 onto stack.
2. Repeat, to get the second operand & push it onto the stack.
3. Prompt for the operation symbol (in this case, a "*"). You can simply discard it once received, since multiplication is the only operation we are implementing.
4. Call the SUB_RPN_ADDITION subroutine to pop the top two values off the stack, add them, and push the result back onto the stack.
   - if you wish, you may use a helper s/r to do the actual multiplication - it only needs to handle the multiplication of two single digit numbers.
5. POP the result off the stack and print it out to the console in decimal format.
   - use a helper s/r to do the output: pass in the number in R0, and print it as numeric characters; it only has to be able to print 1. *(You can add x30 to convert the single digit integer to ASCII).*

**Hints:**

- You will need the following subroutines to get the job done:
  - SUB_STACK_PUSH (exercise 1)
  - SUB_STACK_POP (exercise 2)
  - SUB_RPN_ADDITION
  - SUB_ADDITION (optional)
  - SUB_GET_NUM (optional)
  - SUB_PRINT_DIGIT (convert integer to ASCII)

- You do not need to implement any input validation - we will test only with single-digit numbers.

- Use the subroutine protocol from lab 5, because you will have nested subroutine invocations - make sure you get the register backup & restore exactly right!!

### 3.4 Submission

Demo your lab exercises to your TA ***before you leave lab***.
If you are unable to complete all exercises in lab, show your TA how far you got, and request permission to complete it after lab.
Your TA will usually give you partial credit for what you have done, and allow you to complete & demo the rest later for full credit, so long as you have worked at it seriously.
When you're done, demo it to any of the TAs or instructors in office hours ***before your next lab***.
*Office hours are posted on Piazza, under the "Staff" tab.*

## 4 So what do I know now?

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