CS 2110 - Lab 09

Subroutines & The Stack

Wednesday, June 22, 2022

Lab Assignment: Subroutines Quiz

1. Go to Quizzes on Canvas
2. Select Lab 09, password: **stacksonstacks**
3. Get 100% to get attendance!
    a) Unlimited attempts
    b) Collaboration is **allowed**!
    c) Ask your TAs for help :)

Homework 4 Demos!

- Demos are this week!
- Sign-ups are on Canvas calendar; **we can no longer guarantee you a slot if there isn't one that works for you**
- Each demo is about 10 minutes—please be on time!
  - Your demo time ends strictly at the end of your slot. If you arrive 6 minutes late, you will get 4 minutes to do your demo!
- The demo is worth 50% of your Homework 4 grade
- If you miss your demo or cancel within 24 hours, you will not receive the 50 demo points

Homework 5

- Covers basic assembly programming topics
- Released!
- **Due Monday, June 27th at 11:59 PM**
- Files available on Canvas
- Submit to Gradescope (unlimited submissions)

Homework 6

- Covers assembly subroutines and calling convention
- Will be released on Friday, June 26th
- **Due Thursday, July 7th at 11:59 PM**
- Files available on Canvas
- Submit to Gradescope (unlimited submissions)
- Please don't wait until the very last hours before the homework is due to ask for help!

Mid-Semester Grade Releases

- Drop deadline is July 2, 2022
- By Wednesday, June 29th, the following grades will be released:
  - Homework 1-5
  - Quiz 1-2
  - Timed Lab 1-2
  - All Lecture Attendance Quizzes up to Monday, June 27th
  - All Lab Attendance Quizzes up to Wednesday, June 22nd
- All excused absences will also be reflected in the Canvas gradebook
- If you have any discrepancies with grades/excusals thus far, please email Shawn Wahi *between Wednesday, June 29th and Friday, July 1*.

Assembly Subroutine

- Subroutine is another name for function
- But there is no “function” abstraction in LC-3 assembly!
- So, we need to create our own abstraction

What do we need to have the function abstraction as in the C program on the right?

- `multiply()` should return to the correct place
- We need to communicate function parameters and return values somehow
- The state of other variables like `d` should not change after the call

```c
#include <stdio.h>

int multiply(int op1, int op2) {
    int out = 0;
    for (int i = 0; i < op2; i++) {
        out += op1;
    }
    return out;
}

int main() {
    int a = 2;
    int b = 3;
    double d = 3.1415926;

    int c = multiply(a, b);
    printf("%d\n", a); // 2
    printf("%d\n", b); // 3
    printf("%d\n", c); // 6
    printf("%f\n", d); //3.1415926
}
```

Answer: The Stack

- The stack is a location in memory that is useful for storing temporary program data
  - Subroutine calls – parameters, return values
  - Local variables – what if our 8 registers aren't enough
- Grows "downwards" towards smaller memory addresses from a fixed starting location

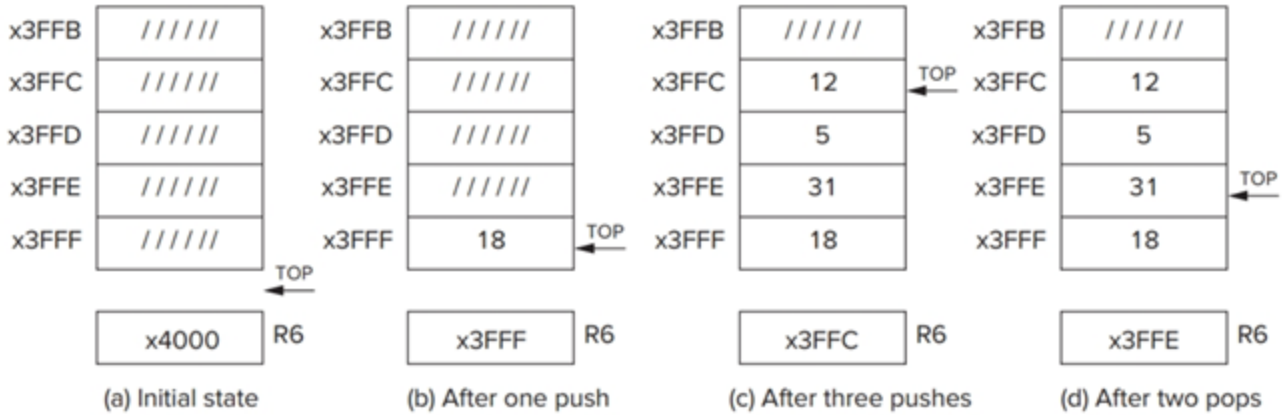
What is the stack?

Select the following that are valid reasons for why we have assembly subroutines

- Provides a useful abstraction for breaking down a difficult problem into smaller steps
- Makes code execute faster
- Allows for easier organization of code
- Reduces amount of duplicate code
- Allows us to create recursive functions

The Stack

- Grows “down” from some location in memory toward smaller addresses
- Top of stack is held in **R6**, aka the stack pointer



Using the Stack

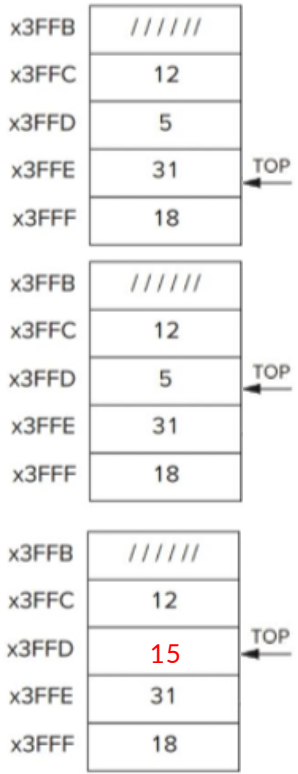
- Initialize R6 to some memory location
- PUSH data to top of the stack from some register RX

```
ADD R6, R6, #-1 ; move stack pointer
STR RX, R6, #0  ; store data
```

- POP data from top of stack into some register RX

```
LDR RX, R6, #0  ; restore data
ADD R6, R6, #1  ; move stack pointer
```

Example: pushing 15 (the value in R4) to the stack:



What values are on the stack in each step?

JSR[R] and RET instructions

- **JSR:** Saves the PC value into R7 and then sets PC to the target
  1. `R7 = PC*`
  2. `PC = PC* + PCOffset11`
     - PC-offset addressing (usually called with a label)
     - Note the order is important; we need to save the PC before changing it
     - "Jump to subroutine“
- **JSRR:** Same as `JSR`, but uses a register for the subroutine's address instead of a label/offset
  1. `R7 = PC*`
  2. `PC = SR`
     - "Jump to subroutine (register)"
- **RET:** A special case of `JMP`. Equivalent to `JMP R7`
  1. `PC = R7`
     - This is actually a “pseudo-instruction” since it assembles to the exact same bits as `JMP R7`
     - "Return“

What does RET do?

Calling Subroutines

- The LC-3 calling convention!
- Create a **stack frame** (or **activation record**) on the stack that holds important information like parameters, return address, return value, etc.
- Save old register values in our stack frame so they can be restored

TOP (LOWER MEMORY)			
	param	<-R6 to call next sub	
	R4		
	R3		
	R2		
	R1		
	R0		
	local	<-R5 (frame pointer)	
	old FP (R5)		
	RA (old R7)		
	RV		
	1st param		
xF000			
BOTTOM (HIGHER MEMORY)			

Frame Pointer

- The Frame Pointer (R5) holds the address of a fixed location in the stack frame/activation record
- Allows us to easily locate our local variables, arguments, return address, etc.
- In the LC-3 calling convention, R5 points to the first local variable saved in our current stack frame

Calling Convention: Important Registers

- R7 - current return address
- R6 - holds the stack pointer/top of the stack
- R5 - holds the current frame pointer

What is a calling convention?

What do the caller and callee do?

Building up the stack

Who Saves	STACK
Callee	Saved Regs
Callee	First local
Callee	oldFP/old R5
Callee	RA (callee r7)
Callee	RV (space)
Caller	Arg 1
Caller	Arg 2

R6

R5

Order of operations:

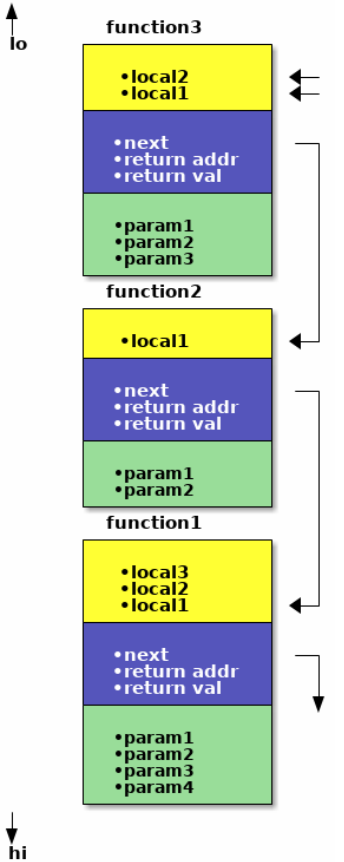
1. Caller pushes args in reverse order (argument 1 on top)
2. Caller uses JSR/JSRR to call subroutine
3. Callee allocates space for/saves: RV, RA, old FP (R5), at least one local variable
4. Callee allocates space for saving registers if need be
5. Reverse order at end of routine

Select all the following that we store on the stack in our calling convention?

- Parameters to a subroutine
- Registers R0-R4
- Return value
- At least 1 local variable
- Return address in R7
- Old frame pointer R5

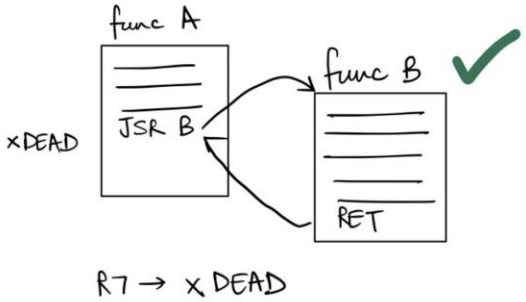
What if a subroutine calls a subroutine?

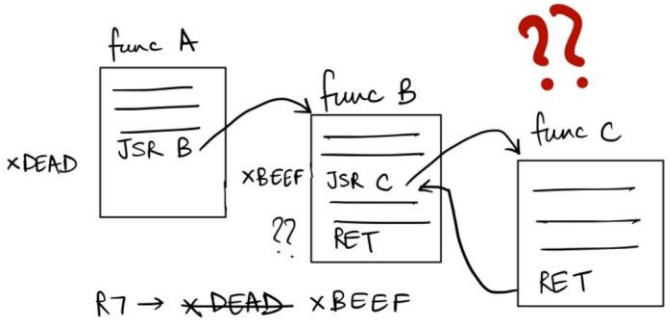
- Just push another stack frame above it
- This works normally, and R6 will always point the current stack frame
  - A stack is a “last-in, first out” data structure
  - The most recently pushed stack frame will be that corresponding to the current function
  - Therefore, our current stack frame will always be at the top of the stack!



What if a subroutine calls a subroutine?

- However, there's a problem: R7 can get "clobbered" by the second call to JSR/JSRR, and then we won't know where to return to!





- This is why we save the return address (from R7) in our stack frame
- We simply have to restore R7 by popping the RA off the stack before calling RET

Live Coding Example

- We will write our first subroutine together!
  - Subtraction Subroutine
- The sample file is posted on Canvas if you want to follow along.
  - Files>Lab Source Code > stack-subtract.zip
- We will post the final product on Canvas as well!