CS 2110 - Lab 10

Subroutines Review and Recursion

Monday, June 27, 2022

Lab Assignment: Subroutines Quiz

1. Go to Quizzes on Canvas
2. Select Lab 10, password: **recursionrecursion**
3. Get 100% to get attendance!
    a) Unlimited attempts
    b) Collaboration is **allowed**!
    c) Ask your TAs for help :)

Homework 5

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

Homework 6

- Released
- Covers assembly subroutines and calling convention
- **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!

Quiz 3

- **Next Wednesday, July 6th**
- Quiz opens at your assigned lab time (unless you have already established a different time with your professor)
- Full 75 minutes to take the quiz!
- As usual, you will join back to your lab section after the quiz.
- A topic list will be posted on Canvas

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
- The top of the stack is stored in the "stack pointer"
- In the LC-3, we use R6 as the stack pointer

Pushing to the Stack

What we want to do:

- Store 3 variables on the stack (R0 on top)
- R0 = 3 R1 = 230 R2 = -684

How do we do it?

Memory Address	Contents	Pointers
xEFF7	garbage	
xEFF8	garbage	
xEFF9	garbage	
xEFFA	garbage	
xEFFB	garbage	
xEFFC	garbage	
xEFFD	garbage	
xEFFE	garbage	
xEFFF	garbage	
xF000	Who knows	<-- R6, stack pointer

Pushing to the Stack

What we want to do:

- Store 3 variables on the stack (R0 on top)
- R0 = 3 R1 = 230 R2 = -684

ADD R6, R6, -3

STR R0, R6, 0

STR R1, R6, 1

STR R2, R6, 2

concise

OR

ADD R6, R6, -1

STR R2, R6, 0

ADD R6, R6, -1

STR R1, R6, 0

ADD R6, R6, -1

STR R0, R6, 0

thorough

Which one do you prefer??

Memory Address	Contents	Pointers
xEFF7	garbage	
xEFF8	garbage	
xEFF9	garbage	
xEFFA	garbage	
xEFFB	garbage	
xEFFC	garbage	
xEFFD	3	<-- R6, stack pointer
xEFFE	230	
xEFFF	-684	
xF000	Who knows	

Popping from the Stack

What we want to do (in order):

- Retrieve 1 value from the stack (NOT popping)
  - unknown3 → R0
- Pop 2 values from the top stack
  - (We don't care about the value of unknown2)
  - unknown1 → R1
  - unknown2 → nowhere

How do we do it?

Memory Address	Contents	Pointers
xEFF7	garbage	
xEFF8	garbage	
xEFF9	garbage	
xEFFA	garbage	
xEFFB	garbage	
xEFFC	garbage	
xEFFD	unknown1	<-- R6, stack pointer
xEFFE	unknown2	
xEFFF	unknown3	
xF000	Who knows	

Popping from the Stack

What we want to do (in order):

- unknown3 → R0
- unknown1 → R1
- unknown2 → nowhere

```
LDR R0, R6, 2
LDR R1, R6, 0
ADD R6, R6, 2
```

concise

OR

```
LDR R0, R6, 2
LDR R1, R6, 0
ADD R6, R6, 1
ADD R6, R6, 1
```

thorough

Which one do you prefer??

Memory Address	Contents	Pointers
xEFF7	garbage	
xEFF8	garbage	
xEFF9	garbage	
xEFFA	garbage	
xEFFB	garbage	
xEFFC	garbage	
xEFFD	unknown1	<-- R6, stack pointer
xEFFE	unknown2	
xEFFF	unknown3	
xF000	Who knows	

To push and pop values onto the stack, we use the ST and LD instructions. True/False?

Calling Convention

- “Subroutine” is another name for “function”; it involves one section of code (the caller) invoking another section of code (the callee)
- We need to figure out a way to communicate important information like parameters and return values between the caller and callee
- This is called a calling convention
- Example of a simple calling convention: “the caller will put the arguments in R0 and R1 and the callee will put the result in R2”
- However, we want a calling convention that is more general and doesn’t “clobber” (overwrite) our registers

LC-3 Calling Convention

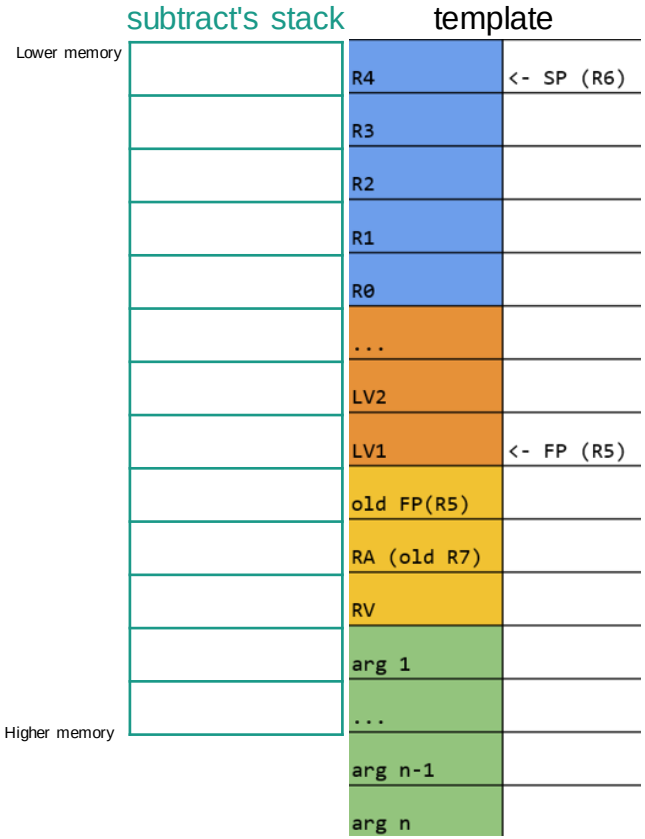
Who Saves	STACK	
Callee	Saved Regs	← R6
Callee	First local	← R5
Callee	oldFP/old R5	
Callee	RA (callee r7)	
Callee	RV (space)	R5+3
Caller	Arg 1	R5+4
Caller	Arg 2	R5+5

- We create a *stack frame* (or *activation record*) on the stack which holds the data we need to save
- We save the old register values so they don't get clobbered by our subroutine
- We use the frame pointer (R5) to have a fixed reference point to access arguments and the return value—how can we compute the address of the *n*th argument?
- We also save the return address to prevent it from being clobbered by future JSR calls

Why do we have a frame pointer?

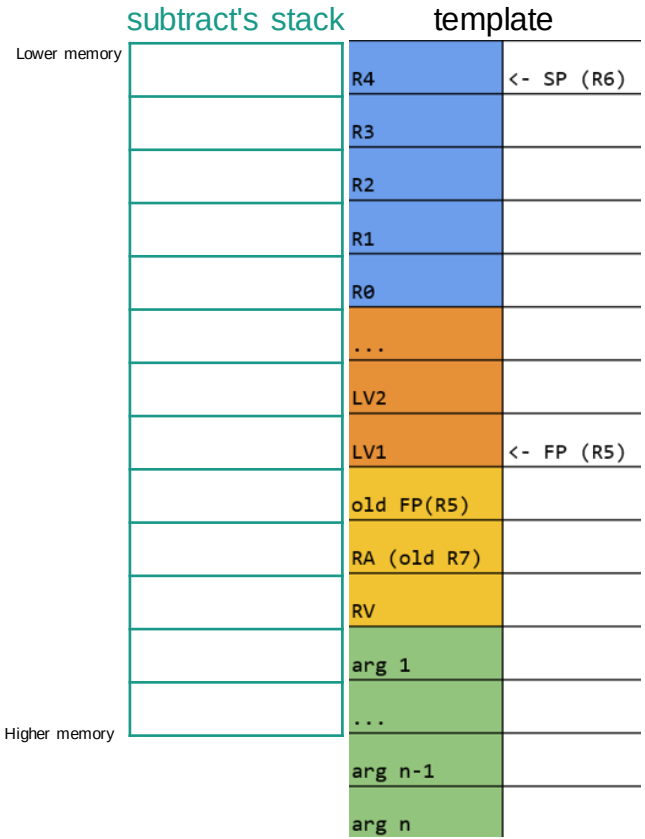
Caller – subroutine example (subtract)

;; MAIN HERE	
LD R0, A	Get A from Memory
LD R1, B	Get B from Memory
ADD R6, R6, -2	Make room for args
STR R0, R6, 0	Store args in reverse order
STR R1, R6, 1	A goes above B
JSR subtract	Call subtract
LDR R2, R6, 0	Get return value off of the stack
ADD R6, R6, 3	Deallocate space for RV and args
ST R2, RESULT	Save result in Memory
;; END MAIN	



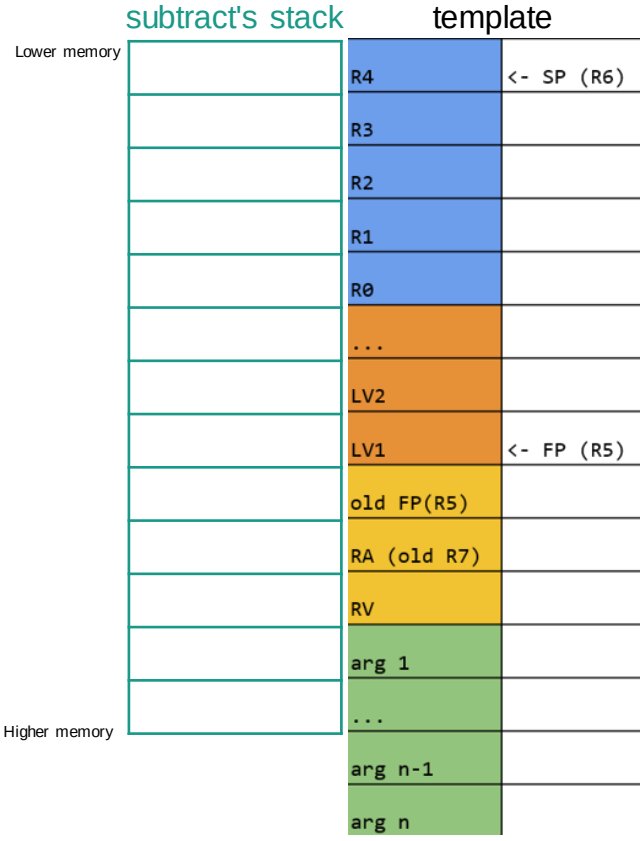
Callee (subtract function)

;; Stack setup	
ADD R6, R6, -4	Make room for RV, RA, old FP, and 1 local
STR R7, R6, 2	Save RA in the space we made for it
STR R5, R6, 1	Save R5 in the space for the old FP
ADD R5, R6, 0	Set R5 to the FP of this Activation Record
ADD R6, R6, -5	Make room for saving Registers
STR R0, R6, 4	Save R0
STR R1, R6, 3	Save R1
STR R2, R6, 2	Save R2
STR R3, R6, 1	Save R3
STR R4, R6, 0	Save R4



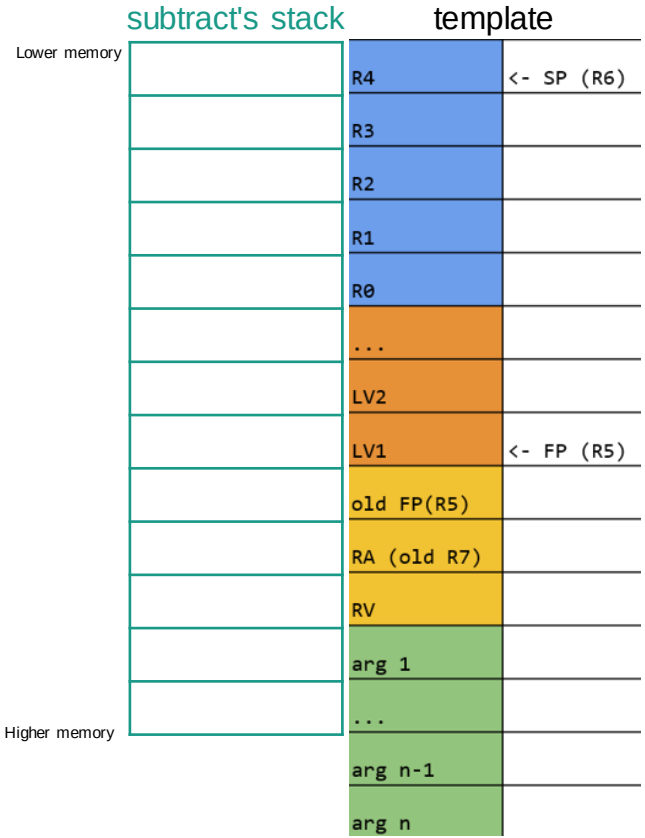
Callee (subtract function)

;; EXTRACT ARGS	
LDR R0, R5, 4	Read A off stack and into R0
LDR R1, R5, 5	Read B off stack and into R1
;; SUBTRACT	
NOT R1, R1	2's complement negation
ADD R1, R1, 1	R1 <= -B
ADD R2, R0, R1	Perform subtraction, R2 <= A - B
;; STORE RV	
STR R2, R5, 3	Put the result in the space allocated for RV



Callee (subtract function)

;; TEARDOWN	
LDR R0, R6, 4	Restore R0
LDR R1, R6, 3	Restore R1
LDR R2, R6, 2	Restore R2
LDR R3, R6, 1	Restore R3
LDR R4, R6, 0	Restore R4
ADD R6, R5, 0	Reset stack pointer to frame pointer
LDR R5, R6, 1	Restore old FP into R5
LDR R7, R6, 2	Restore RA into R7
ADD R6, R6, 3	Deallocate space for local, oldFP, and RA
RET	Return to main



What values does the caller push onto the stack before calling a function?

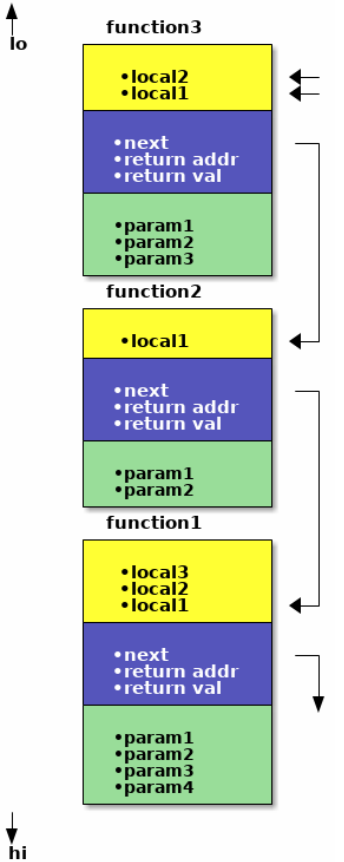
Inside the callee function, we can always load the first argument into R0 using LDR R0, R5, __

Recursion!

- How can we implement subroutines calling themselves in the LC-3?
- Recursion in assembly is the same as other function calls!
- The recursive function will have to act as both the caller and the callee

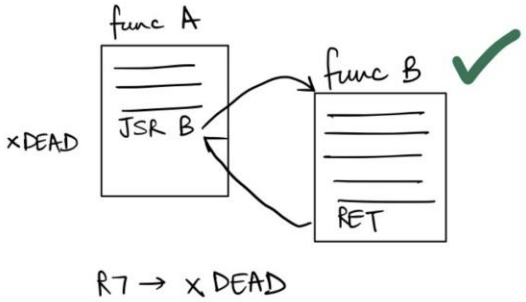
What if a subroutine calls a subroutine?

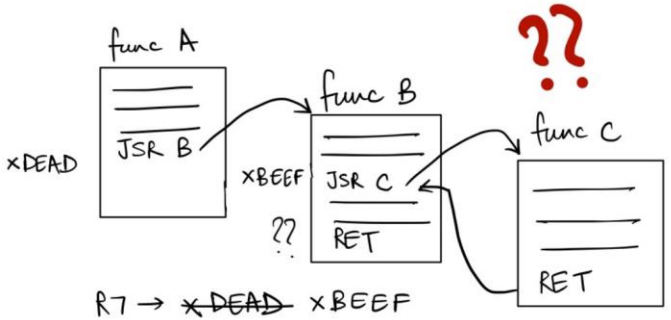
- Just push another stack frame above it
- This works normally, and R6 will always point the current stack frame
  - A stack is a “last-in, first out” data structure
  - The most recently pushed stack frame will be that corresponding to the current function
  - Therefore, our current stack frame will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





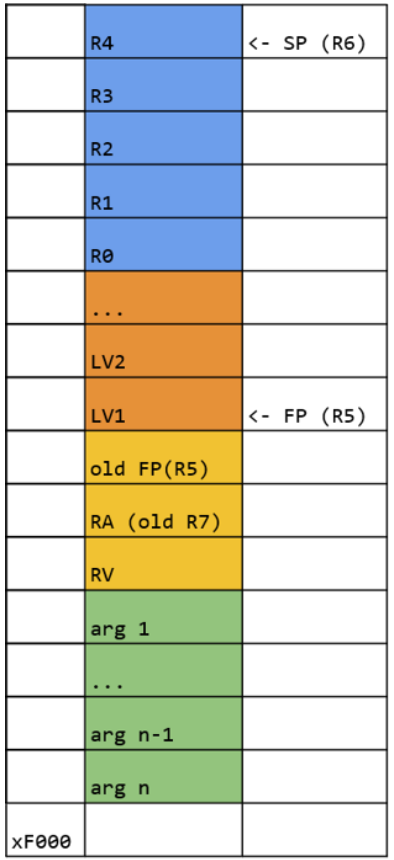
- This is why we save the return address (from R7) in our stack frame
- We simply have to restore R7 by popping the RA off the stack before calling RET

Debugging

Debug **pow2.asm**

Recursively compute a power of 2 given an input integer n.

```
int pow2(int n){
    if (n == 0) {
        return 1;
    }
    return 2 * pow2(n-1);
}
```



The _____ is stored in the trap vector table.

Review: TRAPs

- Subroutines built into LC-3 to simplify instructions
- Look like normal instructions, but are aliases for TRAP calls
  - "`HALT`" is exactly the same as "`TRAP x25`"
- Each has a corresponding 8-bit Trap Vector
  - The LC-3 will look up the location of the subroutine in the Trap Vector Table
  - The trap vector table is stored from address x0000 to x00FF

`HALT` (x25): stops running the program

`OUT` (x21): takes character (in ASCII) in `R0` and prints it on console

`PUTS` (x22): given mem address in `R0`, print characters until NULL terminating character (`'\0'`)

`GETC` (x20): takes character input from console and stores it in `R0`