CS061: Machine Organization & Assembly Language Lab 5

Agenda

1. Presentation:
    a. Subroutine Review
    b. Backing Up / Restoring Registers
    c. Palindromes
    d. Lab Descriptions

2. Work Time / Questions / Demos

Sub-routine'd

- Sub-routines are similar to functions in C++ (and other programming languages).
  - Isolate code for re-use.
  - Takes in parameters.
  - Returns values.

- Similar execution flow:
  - Main program will make a call to a sub-routine (via JSRR)
  - Subroutine code does its magic.
  - Subroutine executes RET to go back to the main program!

Control Flow Review

```
.ORIG x3000

x3000   LD R5, SUB_MULT_PTR
x3001   JSRR R5

x3002   ADD R1, R1, #5

x3003   HALT
x3004   SUB_MULT_PTR   .FILL   x3200
.END
```

JSRR Rx: Call a subroutine!

1. Store address of next instruction into R7.
2. Jump to address specified in Rx.

R7 = x3002

```
.ORIG x3200
; Subroutine Code
; ...
RET
.END
```

RET: Go back to address in R7.

- Alias for JMP R7.

Sub-routine Data Review

- Like local data, but for a sub-routine!
- **Goes after RET (but before .END)!**
- Cannot be accessed from the main code!

```
.ORIG x3200

RET
; Sub-routine Data
HEX_40_3200  .FILL  x40
.END
```

- Label must have a unique name
  - Cannot match any other label name in the file!

Sub-routine Template

```
;=================================================================
; Subroutine: SUB_intelligent_name_goes_here_3200
; Parameter: (Register you are "passing in"): [description of parameter]
; Postcondition: [a short description of what the subroutine accomplishes]
; Return Value: [which register (if any) has a return value and what it means]
;=================================================================

.orig x3200       ; use the starting address as part of the sub name

;========================
; Subroutine Instructions
;========================

; (1) Backup R7 and any registers that this subroutine changes, except for Return Values
; (2) Whatever algorithm this subroutine is intended to perform - only ONE task per sub!!
; (3) Restore the registers that you backed up
; (4) RET - return to the instruction following the subroutine invocation

.end              ; every .orig needs a matching .end
```

📞 Calling Convention 📞

- Imagine we could only see what a subroutine does via its header.

```
.ORIG x3000

LD R1, ARRAY_PTR

LD R5, SUB_FILL_PTR
JSRR R5

ADD R1, R1, #5

HALT
SUB_FILL_PTR   .FILL   x3200
ARRAY_PTR .FILL x4000
.END
```

```
;-----------------------------------------------------
; Subroutine: SUB_FILL_ARRAY_3200
; Parameter (R1): The starting address of the array.
; Postcondition: The array has values from 0 through 9.
; Return Value (None)
;-----------------------------------------------------
```

- Intention: R1 should be x4005.
  - How do we know subroutine didn't change R1 though?

Callee Saved

- Let's define a contract:

✅ **Contract:** ✅

★ A subroutine is allowed to modify registers within the subroutine code.

★ BUT all registers it modifies must be set to their original values before the subroutine *exits*.

★ *EXCEPTION*: Return value registers can stay modified.

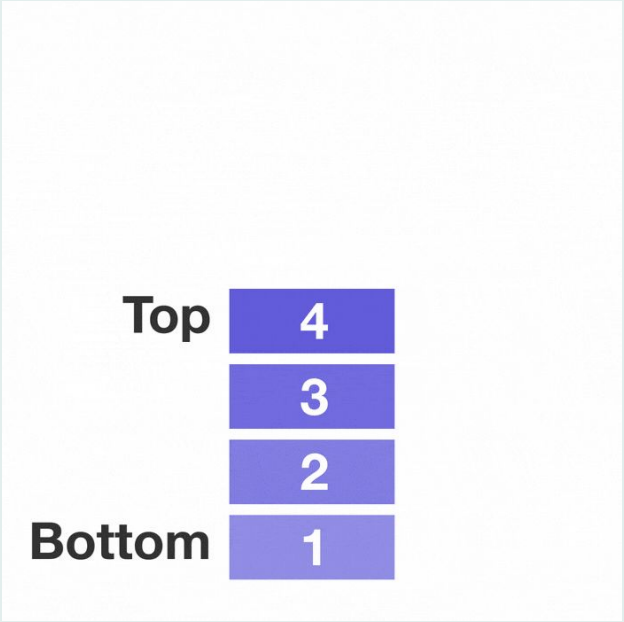
⚠️ Back it up! ⚠️

- How to follow the contract?
- Back up registers at the start of the subroutine!
- Backing up registers means storing the register values in memory!
  - Backup registers that are ***used*** in your sub-routine logic.
  - Always *always always* backup R7!
  - Don't backup registers that are being used as return values!
- How to back up? Using a *STACK*

Stack Crash Course

- Stacks are a LIFO (Last-in First-Out) data structure!
- Can only touch the top of the stack!
- Two Operations:
  - Push elements onto the top of the stack.
  - Pop elements from top of the stack.

https://clipartix.com/stack-of-books-clipart-18-image-60998/



GIF Source

Back onto a Stack

- Top of stack address will always be in R6 (e.g. xFE00)
- Subtract top of stack by 1!
- Store register value at current top of stack address!

```
ADD R6, R6, #-1
STR Rn, R6, #0   ; rn is the modified register r0 - r7
```

```
ADD R6, R6, #-1
STR R7, R6, #0
```

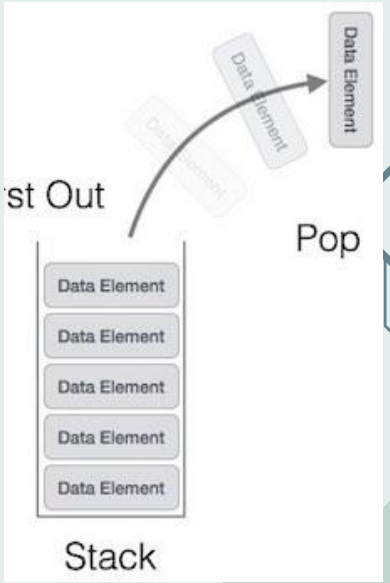
```
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R1, R6, #0
```

Restoring Registers

- After your sub-routine logic!
- **Before the RET statement!**
- Pop off values in the stack in the *reverse* order you pushed them on!

```
LDR R1, R6, #0 ; rn is the modified register r0 - r7
ADD R6, R6, #1
```

```
LDR R1, R6, #0
ADD R6, R6, #1
LDR R7, R6, #0
ADD R6, R6, #1
```



https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Routine so Far

- Stack pointer decrements first!
- Always store R7!

- Stack pointer increments!
- Reverse order!

```
.orig x3200

; Backup Registers
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R1, R6, #0

; Sub-routine logic

LDR R1, R6, #0
ADD R6, R6, #1
LDR R7, R6, #0
ADD R6, R6, #1

RET

; Sub-routine Data

.end
```

Palindromes

- Palindromes are “words, phrases, or sequences that reads the same backward as forward”!
  - Ignore spaces between words!

- Examples:
  - madam
  - taco cat
  - nurses run

https://uselessetymology.com/2019/10/01/on-the-etymology-of-palindromes-and-other-wordplay/

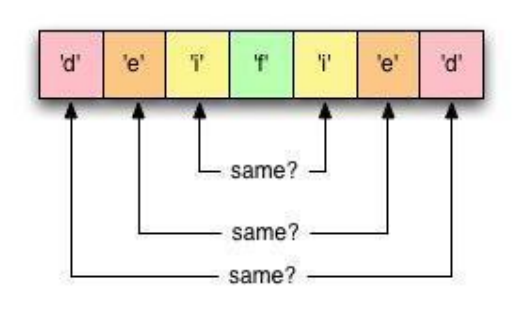
Exercise 1

- Create a sub-routine GET_STRING that takes in user input until the user enters the [ENTER] key!
  - Characters will be stored at an array specified by an address in R1.
  - This array should be null-terminated (have 0 at the end of it)!
  - Array **can't** store sentinel character (can't store the [ENTER] key the user types).
  - Sub-routine returns number of characters user entered in R5.

- Create a test harness (in your main code) that tests the sub-routine.
  - Hard-code an array address at R1.
  - Call the sub-routine.
  - Call PUTS on the array (remember array address must be in R0 to use PUTS)!

- Similar to your Lab 2 Exercise 4, but now in a sub-routine!

Exercise 2



- Create a subroutine “SUB_IS_PALINDROME”:
  - Parameter: Takes in address of a string in R1.
  - Parameter: Number of characters in the string in R5.
  - Determine if string in R1 is a palindrome!
  - Return Value: Set R4 to 1 if it is a palindrome, or to 0 if it is not!

- Create a test harness to test your sub-routine.
  - If the string is a palindrome, test harness should print out “The string <string> is a palindrome!”
  - If the string isn’t a palindrome, test harness should print out “The string <string> is not a palindrome!”

- Tips:
  - Compute address of last character in string.
  - In a loop, compare the first and last characters, then move the addresses up/down by one.
  - Check for when a string is not a palindrome and fast exit the loop!

Exercise 3

- Goal: Make your palindrome subroutine case insensitive!

- Create a new sub-routine: “SUB_TO_UPPER” that converts a string to uppercase!
  - Parameter: Starting address of the string in R1.
  - Convert all characters in the string to uppercase (in place).
  - Use bit-masking (see the ASCII table to figure out how to do this) for the conversion!

- **Call the “SUB_TO_UPPER” subroutine inside** of your “SUB_IS_PALINDROME” sub-routine!
  - Should be able to handle strings like “MadamImAdam” now!

Demo Info

- **Lab Grade Breakdown:**
  - 3 points for attendance.
  - 7 points for demoing (+1 bonus point demo'd before/during Friday).
  - 3 point penalty if lab is demo'd during the next lab session.

- **Tips before you demo:**
  - ***Understand your code!*** (Know what each line does & the input/output)
  - ***Test your code!*** (Check for correct output and that there are no errors)