CS061 – Lab 5

Building Subroutines

1 High Level Description

Today you will learn the art of building subroutines, and become invincible.

2 Our Objectives for This Week

1. If you have not done so already, complete Lab 4, exercise 4
   (Go back to your your lab 4 directory, and follow the instructions in your lab 4 specs)
   You must complete this step BEFORE the start of your lab 5 session)
2. Lab 4 review, & intro to subroutines
3. Roll your own subs – Exercises 1 - 3 (Palindrome checker)
4. Feel smug about it all -- Exercise 4

3. Subroutines: The Art of writing something *once*

Subroutines are a bit like functions in C++, they allow you to "package" your code for re-use, or for repeated use, or simply to make your code more organized & readable.
Here is the basic structure of every subroutine you will ever write in LC3:

```
;===================================================================
; 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              ; because for every .orig, there must be a .end
```

The header contains all the information needed when you or other programmers reuse the subroutine:

- **Subroutine name:**
  Give the subroutine a good name and append the subroutine's address to it to make it *unique*.
  For example, if the subroutine "SUB_PRINT_ARRAY" starts at x3600, you should name the subroutine "SUB_PRINT_ARRAY_3600" to make it completely unique.
- **Parameters:**
  Any parameters that you pass to the subroutine. This is a bit like passing params into a C++ function, except here you pass them in via specific *registers* rather than named *variables*.
- **Postcondition:**
  What the subroutine actually does so you won't have to guess later ...
- **Return Value:**
  The register(s) in which the subroutine returns its result (if any). Unlike C++ functions, you can return multiple values from a subroutine, one per register.
  We sometimes use this to return both a value and a "success/failure" flag, for instance.
  If you are *really* careful you can even return a value in the same register that was used to pass in a parameter *(but not until you really know what you're doing!!)*
- **.ORIG value:**
  Each subroutine that you write needs to be placed somewhere specific in memory (just like our "main" code which we always locate at x3000).
  A good convention to use is {x3200, x3400, x3600, …} for the {first, second, third, …} subroutine.

Once your header is done, you can write your subroutine. This is a 4-step process:

1. **Backing up registers:**
   In this step, use the stack (we'll talk more about this abstract data type later) to backup R7 and

any other registers that this subroutine changes *except for registers used for passing in parameters and/or for returning values*.

To back up a register using the stack, for each register use the following pattern:

```
add r6, r6, #-1
str rn, r6, #0  ; rn is the modified register r0 - r7
```

To restore a register using the stack, each register use the following pattern:

```
ldr rn, r6, #0  ; rn is the modified register r0 - r7
add r6, r6, #1
```

For multiple registers, you must always restore registers in the opposite order they were backed up. For example, the following code snippet backs up and restores registers R7 and R1.

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

; Subroutine code in between

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

Warning, *do not use R6 anywhere else in your code except to back up and restore registers at the beginning and end of your subroutines.*

You ***must always*** backup R7 because - as we will see below - R7 stores the return address *(the address of the instruction following the invocation)*.

Note that BIOS routines (TRAP instructions) are themselves subroutines, and therefore also use R7 to store their return address. So if you use, say, OUT inside your subroutine, and you haven't backed up your own subroutine's R7, the BIOS routine will overwrite R7, and you will be trapped forever at the bottom of a black hole, unable to return home!

2. **Write your subroutine code:**
   Write the code to have the subroutine do its thing.
   All prompts & error messages relating to the subroutine should be part of the subroutine - so, for instance, if a subroutine takes input from the user, the corresponding prompt must be in the subroutine data block, not in "main" where the subroutine is invoked.
3. **Restore registers:**
   In this step, use the stack to restore the registers that you backed up in step 1.
   Remember to *always* backup/restore R7.
   Remember to *never* backup/restore register(s) that contain your Return Value(s).
4. **Return:**
   Use the RET instruction to return to where you came from - a bit like return in C++
   *(RET is not a separate LC3 instruction - it is just an alias for **JMP R7**)*

Reminder about Register Transfer Notation:

- Rn = a register
- (Rn) = the contents of that register
- Mem[some address] = the contents of the specified memory address.
- a <-- b = transfer (i.e. copy) the *value* b to the *location* a.

- R5 <-- (R4) means "copy the contents of Register 4 to Register 5, overwriting any previous contents of Register 5" - e.g. `ADD R5, R4, #0`
- Mem[xA400] <-- Mem[(R3)]
  means “obtain the value stored in R3 and treat it as a memory address; obtain the *value stored at that address*; copy that *value* into memory at the *address* xA400.

Invoking a subroutine
This is a bit like a function call in C++ *(except you have to take care of all the details yourself!)*
JSR and JSRR *(two versions of the same instruction, differing only in their memory addressing modes)*

`JSR label` works just like BRnzp - i.e. it *unconditionally* transfers control to the instruction at label;
`JSRR R5` works just like JMP - i.e. it transfers control to the instruction located at the address stored in R5, known as the *base register (you can use any register for this, but we standardize on R5)*.

However, there is one very big difference between JSR/JSRR and BR/JMP:
Before JSR/JSRR transfers control, the address of the next instruction is stored in **R7**.
At the end of the subroutine, RET (= JMP R7) transfers control back to "main"
JSRR requires that we first load the subroutine address in a BaseRegister, via a pointer *(the same techniques we used to set up arrays in lab 4: LD into R5 from local data SUB_PTR)*.

Example Code: Yay!!!
Here is an example that calls a really short subroutine that takes the 2's complement of (R1).
This is a single file, with single .END, containing both main and subroutine, each with its own .ORIG

```
;===============================================================================
; Main:
;      A test harness for the SUB_TWO_COMPLEMENT_3200 subroutine.
;===============================================================================
    .orig x3000
; Instructions:
    ld r1, const             ; get number to test subroutine with

; Initialize the stack. Don't worry about what that means for now
    ld r6, top_stack_addr

; Call the subroutine (get its address, then jump to subroutine):
    ld r5, sub_twos_comp_ptr
    jsrr r5

    lea r0, completed_msg    ; tell the user the job is done
    puts

    halt

; Local data (Main):
const               .fill #29
sub_twos_comp_ptr   .fill x3200
completed_msg       .stringz "The 2's comp. of the value in R1 is now available in R2\n"
top_stack_addr      .fill xFE00

.end

;===============================================================================
; subroutine: SUB_TWO_COMPLEMENT_3200
; Input (R1): the value whos twos complement will be calculated. This value is
;             not modified by the program.
; Postcondition: the subroutine has calculated the twos complement of the value
;                in R1, and stored it in R2.
; Return value (R2): the twos complement of the value in R1. i.e. R2 <= -R1
;===============================================================================
    .orig x3200
; subroutine instructions:

; (1) backup the important and modified registers
    add r6, r6, #-1
    str r7, r6, #0  ; r7 is always backed up
    add r6, r6, #-1
    str r3, r6, #0  ; r3 is modified by the subroutine
; We didn't really need to back up R3 as a temporary data store - we are just
; using it to illustrate good register backup practices.
; Also, the subroutine does not do any i/o, so we don't really need to back up
; R7 either - but we ALWAYS back up R7 anyway, just in case!

; (2) subroutine algorithm:
    not r3, r1
    add r3, r3, #1  ; r3 <= -(r1)
    add r2, r3, #0  ; r2 <= (r3) copy r3 to r2

; (3) restore backed up registers
    ldr r3, r6, #0 ; restore r3
    add r6, r6, #1
    ldr r7, r6, #0 ; restore r7
    add r6, r6, #1

; (4) Return:
    ret

;===============================================================================
;                  END SUBROUTINE
;===============================================================================

    .end         ; remember, for every .orig, there must be a matching .end
```

3.1 Exercises

What is a Palindrome?

In case you didn't already know, a palindrome is a word or phrase that is spelled the same forwards as backwards. Such words include:

- "racecar"
- "madam"
- "deified"
- "tacocat"

Phrases can be palindromes too (see Exercise 3)! For example, the following are all palindromes *(with the assumption that anything except alphabetic characters are ignored)*

- "live not on evil"
- "So many dynamos"
- "Are we not drawn onward, we few, drawn onward to new era"

Exercise 1

Write the following subroutine, which captures a user-entered string at run-time, storing it as a null-terminated character array *(just like the .STRINGZ pseudo-op, with the big difference that that .STRINGZ requires a hard-coded character array at compile-time)*.

```
;-------------------------------------------------------------------------
; Subroutine: SUB_GET_STRING
; Parameter (R1): The starting address of the character array
; Postcondition: The subroutine has prompted the user to input a string,
;      terminated by the [ENTER] key (the "sentinel"), and has stored
;      the received characters in an array of characters starting at (R1).
;      the array is NULL-terminated; the sentinel character is NOT stored.
; Return Value (R5): The number of non-sentinel chars read from the user.
;      R1 contains the starting address of the array unchanged.
;-------------------------------------------------------------------------
```

This subroutine should prompt the user to enter in a string of text, ending with the [ENTER] key. The string of text will be stored starting at whatever address is specified by (R1) and will be NULL-terminated (i.e. the subroutine will store zero (x0000 = #0 ='\0') as a sentinel at the end of the array).

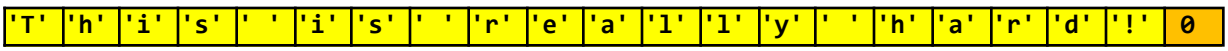
The *entry* sentinel value itself (i.e. the newline) must not be stored in the array!

The subroutine returns in R5 the number of non-sentinel characters entered.

REMEMBER: no "ghost writing"! Echo each character as it is received from the user!

Example:

If the user enters: "This is really hard!", followed by [ENTER], then the array will look like this:



Where the initial 'T' is stored in the address specified by R1 (this value will not be altered by the subroutine); R5 will hold the value x14 = #20 *(count the characters to confirm!)*

Test Harness:

Now write a test harness *(i.e. a program that tests your subroutine to make sure it works)* that does the following:

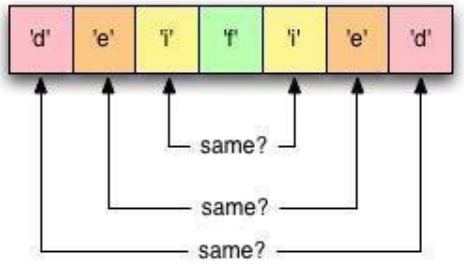
1. R1 <- The address at which to store the array.
   Hard code this address, and reserve space there using .BLKW
   Make sure you have enough free memory starting from this address to store the maximum number of characters likely to be entered - e.g. #100.
2. Calls the subroutine
3. Calls the BIOS routine PUTS (aka Trap x22) to print the string.
   *Remember that PUTS needs the starting address of the string in **R0**, not R1!*

Exercise 2

Now add a subroutine that decides whether the newly entered string is a palindrome or not.

```
;-------------------------------------------------------------------------
; Subroutine: SUB_IS_PALINDROME
; Parameter (R1): The starting address of a null-terminated string
; Parameter (R5): The number of characters in the array.
; Postcondition: The subroutine has determined whether the string at (R1)
;             is a palindrome or not, and returns a flag to that effect.
; Return Value: R4 {1 if the string is a palindrome, 0 otherwise}
;-------------------------------------------------------------------------
```

The algorithm:



Hints:

- You already know the starting address of the array.
- You already know how many characters are in the array.
- Thus, you can calculate the address of the last character of the array
- If the array has n characters, compare
    1. array[0] with array[n-1]
    2. array[1] with array[n-2]
    3. array[2] with array[n-3]
    4. …
- At what point can you decide that the string ***IS*** a palindrome?
  At what point can you decide that the string is ***NOT*** a palindrome?
  Hint: in NEITHER case is the answer "after n comparisons"

Test Harness: Add the following steps to your original test harness:

1. After capturing the string, call the palindrome-checking subroutine

2. Use the return value of the subroutine to report to the user whether the string was a palindrome or not - in other words, report the result in the *test harness*, NOT in the subroutine itself. The subroutine just sets a flag *(this is a standard technique)*.
   Your "main" should print out either
   `The string "<print the input string here>" IS a palindrome`
   or
   `The string "<print the input string here>" IS NOT a palindrome`

Exercise 3:

The subroutine from Exercise 2 would not recognize a phrase such as *"Madam, I'm Adam"* as a palindrome. It would be fairly simple to rework our subroutine to ignore whitespace and punctuation, but for now we will just handle character case, so that your subroutine could at least recognize as a palindrome the string *"MadamImAdam"*. We will do this by converting the entire phrase to the same case before doing the actual palindrome check.

Write the following subroutine:

```
;-------------------------------------------------------------------------
; Subroutine: SUB_TO_UPPER
; Parameter (R1): Starting address of a null-terminated string
; Postcondition: The subroutine has converted the string to upper-case
;     in-place i.e. the upper-case string has replaced the original string
; No return value, no output, but R1 still contains the array address, unchanged
;-------------------------------------------------------------------------
```

Hints:

- Check the ASCII table to see how uppercase and lowercase letters differ in binary/hex
- Use bit-masking, not arithmetic: the conversion of a letter to uppercase can be done with a total of two lines of LC3 code.

Test Harness:

Instead of writing a separate test harness for this subroutine, you can just add a call to it **inside your is_palindrome subroutine from exercise 2**, and test it with a palindrome like "Racecar".

*This is one of the few cases where you will create nested subroutines: it is safe to do so here because **a)** the nested routine has no output and no return value; and **b)** you properly back up & restore only the necessary registers in all subroutines, don't you ? :)*

Exercise 4: Just read this

From now on, ***all*** of your programs will consist of a simple test harness invoking one or more subroutines in which you will implement the assigned task – so make sure you have completely mastered the art of dividing your program up into these “self-contained” modules.

Each subroutine should perform a single task, and have CLEAR and EXPLICIT comments describing what it does, what input is required in which registers, and what will be returned in which register(s).

A subroutine may invoke another subroutine - but only if you are *very, very careful!!*
We will let you know when we want you to do this.

Make sure you understand and always employ basic “register hygiene”:

- backup and restore ***ONLY*** registers that are modified by the subroutine for internal purposes
- *always* backup and restore R7
- *never* backup and restore a register used to return a value
- it is pointless to backup and restore registers that are used to pass in parameters

3.3 Submission

Demo your lab exercises to your TA ***before you leave the lab***.

If you are unable to complete all exercises in the lab, show your TA how far you got, and request permission to complete it after the 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 for the full 3 hours.
When you're done, demo it to any of the TAs in office hours ***before your next lab***.
Office hours are posted on Piazza, under the "Staff" tab.

4 So what do I know now?

... Pretty much everything you need to write real Assembly Language programs :)