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 should have completed this step BEFORE the start of your lab 5 session)
2. Lab 4 review, & intro to subroutines – Exercise 1
3. Roll your own subs – Exercises 2 - 3
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"): [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

;========================
; Subroutine Data
;========================
BACKUP_R0_3200   .BLKW #1 ; Do this for R7 & each register that the subroutine changes
BACKUP_R7_3200   .BLKW #1 ; ... EXCEPT for Return Value(s)
```

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 ST to backup R7 and any other registers that this subroutine changes *except for registers used for passing in parameters and/or for returning values*.
   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 not be able to get back to main!

2. **Write your subroutine code:**
   Write whatever code is necessary to make 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 LD 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 R6 works just like JMP - i.e. it transfers control to the instruction located at the address stored in R6, known as the *base register (you can use any register as the BaseRegister, but we standardize on R6)*.

However, there is one very big difference between JSR/JSRR and BR/JMP:
Before transferring control with JSR or JSRR, the very first thing that happens is that the address of the next instruction - i.e. the return point - is stored in **R7**.
This means that at the end of the subroutine, we can get back to where we jumped from, with RET

In this class, we will be using the **JSRR** mode almost exclusively, as it allows us to invoke subroutines anywhere in memory; we just have to first get the subroutine address in a BaseRegister via a pointer *(the same techniques we used to set up arrays in lab 4: LD into R6 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).
Note that this is a single file, ending with single .END, containing both main and subroutine, each of which has its own .ORIG

```
;=================================================================
; Main:
;      A test harness for SUB_TWOS_COMPLEMENT_3200 subroutine
;=================================================================
                    .orig x3000
; Instructions:
                    ld r1, const            ; get number to test subroutine with

; Call the subroutine (get its address, then Jump to  Subroutine):
                    ld r6, sub_twos_comp_ptr
                    jsrr r6

                    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             ; address of subroutine
completed_msg       .stringz "The 2's comp. of the value in R1 is now available in R2\n"

;=================================================================
; subroutine: : SUB_TWOS_COMPLEMENT_3200
; Input (R1): the value whose two's 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 two's complement of the value in R1
;                      i.e. R2 <-- -(R1)
;=================================================================
                    .orig x3200
; subroutine instructions:

; (1) backup affected registers:
                    st r3, backup_r3_3200   ; r3 is modified by the subroutine.
                    st r7, backup_r7_3200
; We didn't really need to use 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
                    ld r3, backup_r3_3200
                    ld r7, backup_r7_3200

; (4) Return:
                    ret

; Local data for subroutine SUB_TWOS_COMPLEMENT_3200:
backup_r3_3200      .blkw #1
backup_r7_3200      .blkw #1

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

                    .end                    ; NOTE: just ONE .end for the whole file
```

3.2 Exercises
NOTE: No "ghost writing"! That gets an instant 0!

Exercise 1

Recall that exercise 4 from last week created an array of the first 10 powers of 2 {2^0, 2^1, 2^2, ..., 2^9}, and then attempted to print these values to console as though they were ascii characters - with interesting results!
But then in programming assignment 3, you figured out how to take the 16-bit value in a register and print to console the corresponding 16 ascii 1's and 0's.

Your first lab 5 exercise is to take your assn 3 code and turn it into a subroutine:

Copy your lab4 ex. 4 code into lab5_ex1.asm
Now take your assn 3 code and modify it as needed to transform it into a subroutine.
Add it into your lab5_ex1.asm file, with its own .orig
Be sure to adhere to the structure described above.
Finally, inside the display loop in the original lab4_ex4 code, replace OUT with an invocation of your new subroutine, passing in each power of 2 in turn.
The end result will be 10 lines of output *(because the subroutine will be invoked 10 times, once per iteration)*, representing the first 10 powers of 2 as 16-bit binary numbers (displayed, of course, as ascii 1's and 0's).

Exercise 2

Write the *converse* of a binary printing subroutine - that is, write a binary reading subroutine:

First, the subroutine will prompt the user to enter a 16-bit 2's complement binary number: the user will enter 'b' followed by exactly sixteen 1's and 0's: e.g. "`b0001001000110100`" *(no spaces on input)*

Your subroutine should do the following:

1. The user enters a binary number as a sequence of 17 ascii characters: `b0010010001101000`
2. This input is transformed into a single 16-bit value, which is stored in R2.

Your "main" can now invoke the subroutine you built in exercise 1 to print the value of R2 back out to the console to check your work.

"Binary read" Algorithm:

```
R2          <-- 0 (set to 0 - DO NOT load a hard-coded zero from memory!!)
counter     <-- 16
R0          <--  get input from user (the initial 'b') and do nothing with it
do
{
        ; that's your job :)
        ; HINT: the 4-bit binary number b1011 is (b101 * #2 + 1)
} while ( counter > 0 );
```

Once again, you might want to consult the AL control structure tutorials

Exercise 3

Enhance exercise 2 so that it now performs some *input validation*:

- If the first character entered is not 'b', the program should output an error message and start over.
- After that *(i.e. inside the loop)*, if a SPACE is ever entered, the program should simply echo it and continue - i.e. spaces are accepted but ignored in the conversion algorithm.
- If *any* character other than '1', '0' or SPACE is entered inside the loop, the program should output an error message and ask for a valid character - i.e. it should keep everything received so far, and keep looping until it gets a valid '0', '1' or space, and then continue.

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

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 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 :)