CS061 – Lab 06
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. Lab 05 review & intro to subroutines – Exercise 01
2. Roll your own subs – Exercises 02 - 03
3. Feel smug about it all -- Exercise 04

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

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" as a parameter): [description of parameter]
; Postcondition: [a short description of what the subroutine does]
; Return Value: [which register (if any) has a return value and what it means]

.orig x3200
;===============================================================================
;========================
; Subroutine Instructions
;========================
; (1) Backup R7 and all registers that this subroutine changes except for Return Values

; (2) Perform whatever algorithm(s) this subroutine is suppose to perform

; (3) Restore the registers that you backed up

; (4) RETurn to where you came from
;========================
; Subroutine Data
;========================
BACKUP_R0_3200   .BLKW #1 ; Make one of these for each register that the subroutine changes
BACKUP_R7_3200   .BLKW #1 ; EXCEPT for Return Value(s)
```

The header contains information you will need when reusing the subroutine later (which you will be doing ***a lot*** from now on!):

- **.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.
- **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 (similar to a function in C++, except in assembly 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)—again, so you don't have to try to guess later when you want to reuse the subroutine.

Once your header is out of the way, you can write your subroutine. This is a 4-step process:

1. **Backing up registers:**
   Have you run into the problem yet where you said, "awh heck… I ran out of registers to use!"? Well, without this step, you would encounter that problem a **lot** more.
   In this step, use ST to backup R7 and any registers that this subroutine changes *except for the return value(s)*.
   You ***must*** backup R7 because - as we will see below - R7 stores the address to return to after executing, and if you use any TRAPs inside your subroutine, they will use R7 for the same purpose themselves - and you will not be able to return!

2. **Write your subroutine code:**
   Write whatever code is necessary to make the subroutine do its thing. If your subroutine is called SUB_PRINT_ARRAY_3600, then the code would *probably* have something to do with printing an array of some kind, no?

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 on a line by itself to return to where you came from (it is actually just an alias for the instruction "JMP R7"). It does the same thing as "return" in C++ ☺.

Reminder about Register Transfer Notation:

- Rn = a register
- (Rn) = the contents of that register
- Mem[some value] = the contents of the memory address (some value)
- 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. 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.

The instruction you will need

JSR and JSRR *(two versions of the same instruction, differing only in their memory addressing modes)*

JSR works just like BRnzp - i.e. it unconditionally transfers control to the instruction at a label;
and JSRR works just like JMP - i.e. it transfers control to the instruction located at the address stored in a base register;
- ***with one very big difference:*** before transferring control, both JSR and JSRR store the address of the next instruction in R7. This means that at the end of the subroutine, we can get back to where we jumped from with RET, which is just an alias for JMP R7.

Example Code: Yay!!!

Following is an example that calls a really short subroutine that takes the 2's complement of R1:

L:\instruments\labs\lab05\subroutine_example_code\lab5_example1_code.asm Tuesday, February 05, 2013 12:26 PM

```
;---------------------------------------------------------------------------------------------------
; Main:
;   A test harness for the SUB_MAKE_NEGATIVE_3200 subroutine
;---------------------------------------------------------------------------------------------------

                    .ORIG x3000
; Instructions
                    LD R1, DEC_29                       ; R1 <-- #29

; Call the subroutine (takes two lines of code)
                    LD R5, SUB_MAKE_NEGATIVE_3200       ; R5 <-- x3200
                    JSRR R5                             ; jump to subroutine
                                                        ; (and R7 <-- PC)
                    LEA R0, DONE_MSG
                    PUTS                                ; print status message

                    HALT

; Local Data
DEC_29              .FILL #29
SUB_MAKE_NEGATIVE   .FILL x3200
DONE_MSG            .STRINGZ "All done.\n"

;---------------------------------------------------------------------------------------------------
; Subroutine: SUB_MAKE_NEGATIVE_3200
; Input (R1): Some value to take the 2's compliment of
; Postcondition: The subroutine has taken the 2's compliment of (R1) and
;                           left the result in R1.
; Return Value:  R1   <--   -(R1)
;---------------------------------------------------------------------------------------------------
                    .ORIG x3200
; Subroutine Instructions
; (1) Backup R7 & any registers the subroutine changes except Return Values
                    ST R2, BACKUP_R2_3200
                    ST R7, BACKUP_R7_3200

; (2) Subroutine's Algorithm
                    NOT R2, R1                      ; R2 <-- R1 with all the bits inverted
                    ADD R2, R2, #1                  ; R2 <-- R2 + 1
                    ADD R2, R1, #0                  ; R1 <-- R2

; (3) Restore any registers that were backed up
                    LD R2, BACKUP_R2_3200
                    LD R7, BACKUP_R7_3200

; (4) Return
                    RET

; Subroutine Data
                    BACKUP_R2_3200 .BLKW    #1
BACKUP_R7_3200      .BLKW #1

                    .END                            ; Note how there is only ONE .END per file
```

Next is a slightly longer program that has three different strings (each stored in its own array) and calls a subroutine that prints out the contents of an array – in other words, we call the same subroutine 3 times. The address of the string to print is passed to the subroutine in R1.

This time, the main code block is in a *different file* than the subroutine code. Each file has its own .ORIG and its own .END. You can load both files into the simpl simulator by typing the following on the command line: `simpl main.asm library.asm`

L:\instruments\labs\lab05\subroutine_example_code\lab5_example2a_code.asm Tuesday, February 05, 2013 1:02 PM

```
;-------------------------------------------------------------------------------------------------
; File: main.asm
; Main:
;   A test harness for the SUB_PRINT_ARRAY_3400 subroutine
;-------------------------------------------------------------------------------------------------
                .ORIG x3000
; Instructions
                LEA R0, MSG_01
                PUTS                            ; print description of what we are doing
                LD R0, NEWLINE
                OUT                             ; print a newline

                LEA R1, ARRAY_1                 ; R1 <-- addr(ARRAY_1) (subroutine param)
                JSR SUB_PRINT_ARRAY_3400        ; jump to subroutine SUB_PRINT_ARRAY_3400
                                                ; - possible ONLY if sub is within +/- 1k
                LD R0, NEWLINE
                OUT                             ; print newline

                LEA R1, ARRAY_2
                JSR SUB_PRINT_ARRAY_3400        ; print the contents of ARRAY_2
                LD R0, NEWLINE
                OUT                             ; print a newline

                LEA R1, ARRAY_3
                JSR SUB_PRINT_ARRAY_3400        ; print the contents of ARRAY_3
                LD R0, NEWLINE
                OUT                             ; print a newline

                LEA R0, MSG_02
                PUTS                            ; print "Done."
                HALT

; Local Data
NEWLINE         .FILL #10                       ; newline char
MSG_01          .STRINGZ "Contents of ARRAY:"
MSG_02          .STRINGZ "Done."
ARRAY_1         .STRINGZ "Thank you very much!"
ARRAY_2         .FILL 'S'
                .FILL 'a'
                .FILL 'l'
                .FILL 'a'
                .FILL 'm'
                .FILL 'a'
                .FILL 't'
                .FILL ' '
                .FILL 'p'
                .FILL 'o'
                .FILL '!'
                .FILL #0                        ; marks end of string
ARRAY_3         .STRINGZ "Danke schon!"

                .END
```

```
;-------------------------------------------------------------------------------------------------
; File: library.asm
;
; Subroutine: PRINT_ARRAY
; Parameter (R1): The addr of the beginning of an array of characters
;                 that is terminated by the null character, #0.
; Postcondition: The subroutine has printed to console all characters
;                starting at R1 and continuing until a 0 is reached.
; Return Value: None
;-------------------------------------------------------------------------------------------------
                        .ORIG x3400
; Subroutine Instructions
; (1) Backup R7 & any registers the subroutine changes except Return Values
SUB_PRINT_ARRAY_3400    ST R0, BACKUP_R0_3400       ; backup R0
                        ST R1, BACKUP_R1_3400       ; backup R1
                        ST R7, BACKUP_R7_3400       ; backup R7 (why?)

; (2) Subroutine's Algorithm
WHILE_LOOP_3400         LDR R0, R1, #0              ; R0 <-- Mem[(R1) + 0]
                        BRz END_WHILE_LOOP_3400     ; if ((R0) == 0) { break out of loop }
                        else...
                        OUT                         ; print (R0)
                        ADD R1, R1, #1              ; R1 <-- (R1) + 1
                        BR WHILE_LOOP_3400          ; goto WHILE_LOOP_3400 no matter what
END_WHILE_LOOP_3400

; (3) Restore registers
                        LD R0, BACKUP_R0_3400       ; restore R0
                        LD R1, BACKUP_R1_3400       ; restore R1
                        LD R7, BACKUP_R7_3400       ; restore R7

; (4) Return
                        RET                         ; PC <-- (R7) i.e. jump to the address (R7)

; Subroutine Data
BACKUP_R0_3400          .BLKW #1
BACKUP_R1_3400          .BLKW #1
BACKUP_R7_3400          .BLKW #1

                        .END
```

Note that if you use any other LC-3 emulator than simpl, you will probably have to use the technique of this second example, with a separate .asm file for each routine/subroutine, since the emulator may not be able to handle multiple .orig pseudo-ops in a single file as simpl does.

3.2 Exercises

Exercise 01

Recall that exercise 04 from last week created an array of the first 10 powers of 2 {2^0, 2^1, 2^2, ..., 2^9}, and then printed out their respective 16-bit binary representations (e.g. b0000 0000 0000 0001) one per line.
You did this by pasting the print code directly inside the loop.

Rework this exercise by converting the code that prints out R2 in 16-bit binary into a proper subroutine (as above, including proper headers), and simply invoking it from inside the loop.

Exercise 02

Write the *inverse* of a binary printing subroutine. That is, write a binary reading subroutine:

First, 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. The result of (1) is transformed into a single 16-bit value, which is stored in R2.
3. Call the subroutine from Exercise 01 to print the value of R2 back out to the console to check your work.

Algorithm:

```
total       <--   0
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 );
```

Exercise 03

Enhance exercise 2 so that it now performs some *input validation*. Thus,

- If the first character entered is not 'b', the program should output an error message and go back to the beginning
- After that, if a SPACE is ever entered, the program should only echo it and continue (i.e. spaces are ignored in the conversion algorithm).
- If *any* character other than '1', '0' or SPACE is entered after the initial 'b', 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.

Exercise 04: 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 (and in which registers), and what will be returned (in which register).

Make sure you understand and employ basic "register hygiene" – i.e. backup and restore ***ONLY*** those registers that are modified by the subroutine for internal purposes only (and remember R7!)

3.3 Submission

Add, commit, and push your lab06_ex1.asm through lab06_ex3.asm files to your lab 6 GitHub repo.

4 So what do I know now?

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