CS061 - Lab 4

Intro to Subroutines

1 High Level Description

In today's lab you will be introduced to subroutines and use subroutines to mess with arrays and console output.

2 Our Targets for This Week

1. The art of writing something *once*
2. Filling up the plate – Exercise 1
3. What a character! Exercise 2
4. As far as the eye can see… – Exercises 3 - 4
5. I rock!

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              ; every .orig needs a matching .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, you would back up register values to fixed (e.g. arrays) or dynamic structures (e.g. stacks). **For this lab, we will ignore backing up and restoring registers**. As you go through this lab though, try to ask yourself "what benefit does backing up registers give us?".

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, <u>not</u> in "main" where the subroutine is invoked.

3. **Restore registers:**
   In this step, you would restore the register values that you backed up in step 1. As mentioned above, **you will skip this step for this lab**!

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****)*

<u>Reminder about Register Transfer Notation:</u>

- 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 *<u>value</u>* b to the *<u>location</u>* 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 *<u>value</u>* into memory at the *<u>address</u>* xA400.

<u>Invoking a subroutine</u>

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 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).
This is a single file, with single .END, containing both main and subroutine, each with its own .ORIG.

```
; =======================================================
; Main
;    Test harness for SUB_TWO_COMPLEMENT_3200 subroutine
; =======================================================
.ORIG x3000
; Main Program (Test Harness) Instructions:
LD R1, CONST ; R1 <- x29 ; Load R1 with the constant value to test
with.

; Load subroutine address in R5
LD R5, SUB_TWO_COMPLEMENT_3200 ; R5 <- x3200
; Call subroutine (i.e. jump to address in R5)
JSRR R5

; Print out completed message
LEA R0, COMPLETED_MSG
PUTS

HALT
; Local data
CONST                        .FILL x29
SUB_TWO_COMPLEMENT_3200      .FILL x3200
COMPLETED_MSG                .STRINGZ   "The 2's comp. Value in R1 is
now available in R2"
.END
```

```
;===================================================================
; Subroutine: SUB_TWO_COMPLEMENT_3200
; Parameter: (R1): Value whose two’s complement will be calculated.
; Postcondition: Calculates two’s complement of R1 and stores it into
R2.
; Return Value (R2): Two’s complement of value in R1. i.e. R2 <- -R1.
;===================================================================
.ORIG x3200
; (1) Backup registers (skipped)
; (2) Subroutine Algorithm
NOT R2, R1      ; R2 <- ~R1
ADD R2, R2, #1  ; R2 <- R2 + 1
; (3) Restore registers (skipped)
; (4) Return out of the sub-routine (i.e. go back to the main
program).
RET
; Sub-routine Data
; Use like local data, but can’t have the same names as any
; other labels in the program!
.END
```

3.1 Filling up the plate

Along with subroutines, you will be building on the skills you learned last week: arrays, so let's start with a variation on last week's array – this time storing the results of a calculation, rather than user input.

Exercise 1

Create a subroutine called “SUB_FILL_ARRAY” which programmatically fills up an array starting at an address in R1 with the *numbers* 0 through 9 *(note - NOT the characters '0' through '9'!)* – i.e. hard-code just the first value (0), then **calculate** the rest one at a time, storing them in the array as you go. *Remember the proper way of setting a register to 0!*

```
;-----------------------------------------------------------------------
; Subroutine: SUB_FILL_ARRAY
; Parameter (R1): The starting address of the array. This should be unchanged at the
end of the subroutine!
; Postcondition: The array has values from 0 through 9.
; Return Value (None)
;------------------------------------------------------------------------
```

For example, if R1 has the address of x4000, then the array should span from x4000 through x4009. After the subroutine finishes, the LC-3 memory should look like:

Memory Address	**Value**
x4000	0
x4001	1
x4002	2
x4003	3
x4004	4
x4005	5
x4006	6
x4007	7
x4008	8
x4009	9

Things to remember:

1. The sub-routine should be placed at x3200 and the name (in the subroutine header) should reflect where it was placed (e.g. "SUB_FILL_ARRAY_3200").
2. Only hard-code the zero value, but calculate the rest!
3. At the end of the subroutine, R1 should be unchanged (i.e. still have the starting address of the array).

Test Harness:

Now write a test harness *(i.e. the main 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 10 numbers - e.g. #10.
2. Call the subroutine!

As always, step through your program and **examine the values as they are stored in the array**.

3.2 What a character!

Exercise 2

You'll notice that Exercise 1 didn't ask you to output to console the array you built.
Why not?
Because as you know by now, numbers are not characters!

The end goal of this lab is to print out the array of values. In order to make that happen, the first thing you need to do is convert the numbers into characters!

First, copy your exercise 1 code into lab4_ex2.asm. Now create another subroutine that traverses the array and converts each element of the array into characters. Make sure to store the converted value back into the array!

```
;----------------------------------------------------------------------
; Subroutine: SUB_CONVERT_ARRAY
; Parameter (R1): The starting address of the array. This should be unchanged at the
end of the subroutine!
; Postcondition: Each element (number) in the array should be represented as a
character. E.g. 0 -> ‘0’
; Return Value (None)
;-----------------------------------------------------------------------
```

Things to remember:

1. The new subroutine should be placed at x3400 since there's already a subroutine at x3200. The name (in the subroutine header) should reflect where it was placed (e.g. “SUB_CONVERT_ARRAY_3400”).
2. During the array traversal, first load the element from the array, convert the value, and then store the value back into the array.
    a. Think about what type of load and store you'll need to do this!
    b. How do you convert a digit (0 - 9) to a character (‘0’ - ‘9’)? Hint: Look at the ASCII table!
3. At the end of the subroutine, R1 should be unchanged (i.e. still have the starting address of the array).

Test Harness:

Expand upon the test harness from exercise 1. Add a subroutine call (after your subroutine call to “SUB_FILL_ARRAY”) that calls your “SUB_CONVERT_ARRAY” subroutine.

After your program executes, make sure that the values in the array are now the character representations of the numbers you originally stored! For example, the first couple array elements should be (if R1 is x4000):

Memory Address	Value	Character
x4000	48	‘0’
x4001	49	‘1’
…	…	…

3.3 As far as the eye can see…

Exercise 3

Now that the array has characters, you can print them out! For this exercise, you’ll create another subroutine that traverses the array and prints out each character.

Go ahead and copy your exercise 2 code into lab4_ex3.asm . Create another subroutine with the following header:

```
;-------------------------------------------------------------------------
; Subroutine: SUB_PRINT_ARRAY
; Parameter (R1): The starting address of the array. This should be unchanged at the
end of the subroutine!
; Postcondition: Each element (character) in the array is printed out to the console.
; Return Value (None)
;--------------------------------------------------------------------------
```

Things to remember:

1. The new subroutine should be placed at x3600 since there’s already subroutines at x3200 and x3400. The name (in the subroutine header) should reflect where it was placed (e.g. “SUB_PRINT_ARRAY_3600”).
2. During the array traversal, first load the element from the array, then print out the character.
    a. What instruction prints out a single character?
3. At the end of the subroutine, R1 should be unchanged (i.e. still have the starting address of the array).

Test Harness:

Expand upon the test harness from exercise 2. Add a subroutine call (after your subroutine call to “SUB_CONVERT_ARRAY”) that calls your “SUB_PRINT_ARRAY” subroutine.

After you run your program, the characters should be printed out to the console.

Exercise 4

Yay it prints out correctly! The benefit of subroutines though is being able to reuse them. For example, say you wanted to print out text before and after you printed out the array. Without subroutines, you would have to write the same print logic again, but not with subroutines!

Copy your exercise 3 code into lab4_ex4.asm. This time, create another subroutine with the following header:

```
;----------------------------------------------------------------------
; Subroutine: SUB_PRETTY_PRINT_ARRAY
; Parameter (R1): The starting address of the array. This should be unchanged at the
end of the subroutine!
; Postcondition: Prints out "=====" (5 equal signs), prints out the array, and after
prints out "=====" again.
; Return Value (None)
;-----------------------------------------------------------------------
```

Things to remember:

1. The new subroutine should be placed at x3800 since there's already subroutines at x3200, x3400, and x3600. The name (in the subroutine header) should reflect where it was placed (e.g. "SUB_PRETTY_PRINT_ARRAY_3800").
2. **Do not repeat the print logic for the array, make a call to the subroutine you have already created for printing the array!**
   a. You will need to create a pointer to your "SUB_PRINT_ARRAY" subroutine in the *subroutine data*. Then call the subroutine (similar to how you do in the main program).
3. For printing the equal signs, the easiest thing to do would be to define a string in the subroutine data (i.e. with ".STRINGZ"). Then print out the string (what instruction does this?) before calling the SUB_PRINT_ARRAY subroutine and print it out again after.

Test Harness:

Expand upon the test harness from exercise 3. Add a subroutine call (after your subroutine call to "SUB_PRINT_ARRAY") that calls your "SUB_PRETTY_PRINT_ARRAY" subroutine.

Run the program and

…it doesn't stop?! Everything looks like it should work, what's going on?

There's a sneaky bug in the program such that the program gets stuck in an *infinite loop*. Your job is to step through the program and figure out why it happens.

As you do so, make sure your able to answer the following questions:

- What causes the program to never finish?

- How do subroutines get back to the main program when they finish?
    - How does JSR/JSRR work?
    - How does RET work?
- What instruction messes up the control flow of the program?
    - Why?
- How would backing up/restoring registers fix this?

3.4 Submission

Your TA will maintain a Google Sheet for questions and demos.
Demo your lab exercises to your TA ***before you leave the lab***.
If you are unable to complete all exercises in the lab, demo the rest during office hours *before* your next lab to get full credit. You can demo during the next lab for -3 points.
Office hours are posted on Canvas (under Syllabus -> Office Hours) and Discord (#office-hours).

4 So what do I know now?

- The basics of subroutines!
- How JSR/JSRR work.
- How RET works.
- The importance of backing up and restoring registers in subroutines.
- You should be able to do counter and sentinel controlled loops in your sleep :)
- Ditto for multi-way branches (the AL version of if statements)
- The difference between a *number* (an abstract concept) and its various *representations* in different bases (2, 10, 16 ...), and as ascii character(s) (specifically, decimal numeric digits)
- Output values from 0 to 9 as their corresponding ascii characters
- How to use the Console to watch for runtime errors and warnings.