# CS061 – Lab 5

**Advanced subroutines!**

## 1 High Level Description

The purpose of this lab is to teach you some advanced subroutine techniques that allow you to write recursive subroutines (a subroutine that calls itself directly or indirectly)!

## 2 Our Objectives for This Week

1. Exercise 1 ~ Demonstrate the shortcomings of subroutine protocols from the book
2. Exercise 2 ~ Improve upon the book's subroutine protocols and show the shortcomings even with this improvement
3. Exercise 3 ~ Refine the subroutine protocols by using stacks to allow for recursion

## 3.1 Exercises

Exercise 1

Recall that the definition of Factorial is $n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \ldots \times n - 1 \times n$ or as the relation Factorial($n$) = Factorial($n - 1$) $\times$ $n$. This relation is defined as a recursive relation since computing the $n^{th}$ factorial relies on computing the $(n - 1)^{th}$ factorial.

We can implement a recursive relation using a recursive subroutine. A recursive subroutine is any subroutine that calls itself either directly or indirectly. A directly recursive subroutine makes a direct call to itself within the subroutine. An indirectly recursive subroutine makes no direct call to itself, but rather calls another subroutine that calls itself. Either way, we need to write our subroutines carefully so that they support recursion, even if we did not deliberately write it to be recursive (in other words, indirectly recursive).

For this exercise, load the provided code from the lab5_ex1.asm file into the LC-3 tools program, assemble, and run the code in the simulator. What happens? (In order to recover from what happens, hit the pause button in the simulator)

*Ex 1. Q1. What happens when you run this code?*
*The code loops infinitely*

Now reset the simulator so you can run the program again. This time, you will step through the code to discover why the program behaved the way it did. Remember, our eventual goal is to implement recursion and this program fails when returning from the original subroutine called after it stops recursing.

Put a breakpoint on address x3003. This should correspond to the instruction 'JSRR R5' which calls the first subroutine. Press play and wait for the program to stop at that breakpoint. Next, press the 'Step in' icon ( ↳ ). That's the icon that looks like a right angle with an arrow pointing to the right. This action will step into the subroutine that is at the address in register R5, which should be x3200.

Next, put a breakpoint on address x3205. This should correspond to the instruction 'JSR FACT_SUB_START'. Now note the values in R0 and R1.

*Ex 1. Q2. What are the values in registers R0 and R1?*
*The values in r0 = 0 and r1 = 5*

Press the play button one more time.

*Ex 1. Q3. What are the values in registers R0 and R1 after pressing play, but before pressing any other buttons?*

*The program at first, stores 5 into r1 before the recursive call but now it has values for r1 = 1 and r0 = 2 while recursively going back and forth in the 3200 subroutine call.*

Then from there use the 'Step in' button. If you use the 'Step over' button you won't be able to observe the recursion.

The register R1 will count down from 5 to 0 upon each invocation of the recursion. Once R0 has the value 0, it will stop recursing and finish executing the last call to the FACT subroutine. It stops because of the branch instruction in address x3204 which has the instruction 'BRz BASE_3200'. The condition code is finally zero upon the last decrement of R1 in the previous address.

Ex 1. Q4 Write the values of registers R0 and R1 for each iteration of the recursion until the recursion stops. You'll know the recursion stops when the program is on the instruction in address x320B, which corresponds to the BASE_3200 label.

| **R0** | **R1** |
|---|---|
| 0 | 5 |
| 0 | 4 |
| 0 | 3 |
| 0 | 2 |
| 0 | 1 |
| 0 | 0 |

Now continue pressing the 'Step in' button until you reach address x3210 which has the instruction 'RET'. The RET instruction is a pseudo-instruction for 'JMP R7'. That means that it jumps to the instruction at the address in R7.

*EX 1. Q5. What is the value of R7 before you execute the RET instruction?*

*The value of r7 is 12806*

Now press ‘Step in’ and you should go back to the address from your answer above and execution will continue executing from there.

*EX 1. Q6. What is the value of R7 every time you get to return after the first time? (Don’t worry if you miss it the first time, it will be the same forever)*

*The value is always 12806 which is where the add r2, r0, #0 instruction is.*

The program is now stuck in an infinite loop of returning to the address from Ex 1. Q 6. Above.

Discuss with the TA why that is and any possible solution to this problem. (Hint: Look at the next exercise)

---

Exercise 2

The shortcoming of the code in the previous exercise is that the R7 register was not backed up and restored in the subroutines like the other registers used. We now know to always back up and restore R7.

For this exercise, copy your code from Lab 5 Exercise 1 above into the file lab5_ex2.asm. Then modify both the FACT and MUL subroutines to properly backup and restore the R7 register (using the same backup and restore technique from exercise 1).

*Ex 2. Q 1. What happens this time when you run the code?*

*The code still runs infinitely but the values of each register changes*

(You’ll need to press the pause button in the simulator again to recover from what happens)

Again, put a breakpoint on address x3003. This should correspond to the instruction ‘JSRR R5’ which calls the first subroutine. Press play and wait for the program to stop at that breakpoint. Next press the ‘Step in’ icon. That’s the icon that looks like a right angle with an arrow pointing to the right. This action will step into the subroutine that is at the address in register R5, which should be x3200.

This time, put a breakpoint on the line corresponding to 'JSR FACT_SUB_START' (it will have moved after adding lines to backup/restore R7). Now note the values in R0 and R1.

*Ex 2. Q2. What are the values in registers R0 and R1?*
*As of now, they're 0 and 5 respectively*

Press the play button one more time.

*Ex 2. Q3. What are the values in registers R0 and R1 after pressing play, but before pressing any other buttons?*
*Now they're 0 and 4 respectively*

Then from there use the 'Step in' button. If you use the 'Step over' button you won't be able to observe the recursion.

Just as in exercise 1, the register R1 will count down from 5 to 0 upon each invocation of the recursion. Once R1 has the value 0, it will stop recursing and finish executing the last call to the FACT subroutine. It stops because of the branch instruction in address x3205 which has the instruction 'BRz BASE_3200. The condition code is finally zero upon the last decrement of R1 in the previous address.

Now continue pressing the 'Step in' button until you reach address x320E which has the instruction 'LD R1, BACKUP_R1_3200'. From here, use only the 'Step over' button. This way you don't have to go through the content of the MULT subroutine. This time, unlike last time, the return from this subroutine works. That's what backing up R7 fixed from exercise 1.

Continue pressing 'Step over' until you reach the RET instruction. The program is about to return from the last recursive call.

*EX 2. Q4. What is the value of R7 before you execute the RET instruction?*
*The value is currently 12807*

Now press 'Step over' and you should go back to the address from your answer above and execution will continue executing from there.

Continue pressing 'Step over' until the value in R0 is 1024.

*EX 2. Q5. Will this program ever finish, or does it just keep going?*

*The program will just keep going*

Now the program is stuck in a loop trying to return from the recursive subroutine. However, each time we return we keep overwriting R7 with the value where the recursive subroutine was called in the subroutine itself. It is never able to go back to where it was originally called in the main part of the program.

Once more, you'll be expected to explain what happened that caused the program to not work correctly.

---

Exercise 3

For this exercise we'll fix the shortcomings from the last exercise by using a stack to backup and restore R7 and any other register we modify instead of a single memory location below the subroutine as in the previous exercises. This exercise will now illustrate why you need to backup/restore registers using a stack, namely, to support recursion.

For this exercise, copy your code from lab 5 exercise 2, above, into the file lab5_ex3.asm. You'll then modify the code to back up and restore the registers on the stack instead of memory locations at the end of each subroutine. But first, you'll need to set up the stack to be used throughout the program.

On the LC3 processor, we usually use R6 to store the top of the stack. This stack starts at memory location xFE00. You'll need to modify the code from the previous exercise starting at address x3000. Change that code to look like the following:

```
1  ;=========================
2  ; Main Program
3  ;=========================
4  .ORIG x3000
5        ; Load stack
6        LD R6, STACK_ADDR
7
8        ; Set R1 to 5
9        AND R1, R1, #0
10       ADD R1, R1, #5
```

```
11
12        ; Call the factorial subroutine
13        LD R5, FACT_SUB_ADDR
14        JSRR R5
15
16        HALT
17  ;=========================
18  ; Local Data
19  ;=========================
20  FACT_SUB_ADDR    .FILL x3200
21  STACK_ADDR       .FILL xFE00
22  .END
```

**How to back up registers with a stack:**
Use the stack 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 uses the following pattern:

```
LDR R1, 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
```

Load R6 with xFE00 (register stack address) before any subroutines run. Use R6 exclusively for this purpose.

Register backup stack address: **xFE00**

**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:** The 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!*

All this additional code does is add a new label, STACK_ADDR, and load its value, xFE00 into R6. This initializes the stack, readying it for future calls to subroutines.

Next, change the beginning and end of each subroutine to use the stack instead of labels at the end of the subroutine. For example:

```
31  ST R7, Save7_3100
32  ST R1, Save1_3100
33
...  ; Subroutine code in between
40
41  LD R1, Save1_3100
42  LD R7, Save7_3100
```

Which backs up and restores R7 and R1 respectively, becomes:

```
31  ADD R6, R6, #-1
32  STR R7, R6, #0
33  ADD R6, R6, #-1
34  STR R1, R6, #0
35
...  ; Subroutine code in between
42
43  LDR R1, R6, #0
44  ADD R6, R6, #1
45  LDR R7, R6, #0
46  ADD R6, R6, #1
```

Which also backs up the same registers using the stack. Be sure to make this change for both the FACT and MUL subroutines. Also, pay close attention to the order in which you backup and

restore the registers. When the registers are restored at the end of the subroutine, they should be restored in the **reverse order** in which they were backed up at the beginning of the subroutine. Also, be sure to do the subtractions and additions in the correct place. A common mistake is to do the addition when you're restoring the values to the registers before the LDR. It should be after.

Once you're done, assemble the code, put a breakpoint on the HALT instruction and run the program. Observe what happens. Did it work? (it should) See if you can figure out where the result of calling the FACT routine is stored.

*Ex 3. Q1. Which register holds the result of calling the FACT subroutine?*

*R0*

*Ex 3. Q2. What value is stored in this register?*

*120*

*Ex 3. Q3. Is this the correct value for 5!?*

*Yes*

Exercise 4

For this final exercise we'll want you to look back at lab 4 Ex. 4, and simply think about how you would fix the nested subroutine call given your current understanding of backing up/restoring registers. There is no code necessary for this exercise, but be sure you fully understand the cause of the error, and the fix as well.

---

### 3.2 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 with a 3 point deduction.

*Office hours are posted on Canvas (under Syllabus -> Office Hours) and Discord (#office-hours).*