

## **CS061 – Lab 7**

### **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  $n! = \prod_{i=1}^n i = 1 \times 2 \times 3 \dots \times n = 1 \times n$  or as the relation  $\text{Factorial}(n) = \text{Factorial}(n - 1) \times n$ . This relation is defined as a recursive relation since computing the  $n^{\text{th}}$  factorial relies on computing the  $(n - 1)^{\text{th}}$  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 lab7\_ex1.asm file into the LC-3 tools program. The FACT subroutine in this file is similar to that from figure 8.12 in the third edition of the textbook. The only difference is that the provided code implements an actual MUL subroutine rather than pretending that a MUL instruction exists in LC3.

Next, assemble, and run the code in the simulator.

Ex 1. Q1. What happens when you run this code?

(In order to recover from what happens, hit the pause button in the simulator).

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 x3100.

Next, put a breakpoint on address x3105. This should correspond to the instruction 'JSR FACT'. Now note the values in R0 and R1.

Ex 1. Q2. What are the values in registers R0 and R1?

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?

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 R0 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 x3102 which has the instruction 'BRz DONE'. 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 x3108, which corresponds to the DONE label.

R0	R1

Now continue pressing the 'Step in' button until you reach address x3109 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?

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 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 learned in Lab 5 to always back up and restore R7.

For this exercise, use the code provided in the file lab7\_ex2.asm. It has been modified so that FACT and MUL subroutines properly backup and restore the R7 register using memory at the end of the subroutine. Assemble and run the code.

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

(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 x3100.

This time, put a breakpoint on address x3106 . This should correspond to the instruction 'JSR FACT'.(the addition of backing up R7 moved this address here). Now note the values in R0 and R1.

Ex 2. Q2. What are the values in registers R0 and R1?

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?

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 R0 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 x3103 which has the instruction 'BRz DONE'. 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 x3109 which has the instruction 'DONE LD R1, Save1\_3100'. From here, use only the 'Step over' button. This way you don't have to go through the content of the MUL 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 address x310B which has the instruction 'RET'. 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?

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?

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 final 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. In Lab 5 you were told to use the stack to back up the registers without understanding what a stack is, and without understanding why we need to use a stack. The last lab, Lab 6, taught you more about programming stack in assembly. This exercise will now illustrate why you need to use a stack, namely, to support recursion.

For this exercise, copy your code from lab 7 exercise 2, above, into the file lab7\_ex3.asm. You'll then add 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 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 x3100
21 STACK_ADDR      .FILL xFE00
22 .END
```

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?

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

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

---

### 3.2 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 Canvas, in the "Office Hours" page.*