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

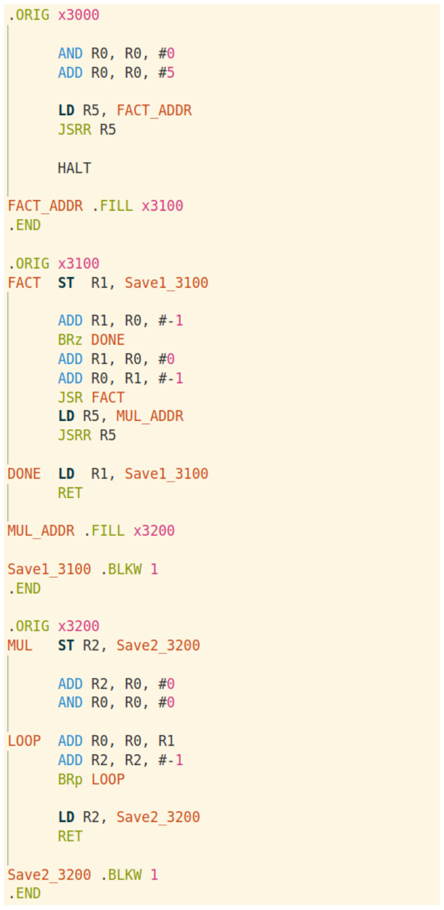
3.1 Exercises

Exercise 1

Recall that the definition of Factorial is $n! \;=\; \prod_{n}^{i=1} i \;=\; 1 \;\times\; 2 \;\times\; 3 \ldots \times\; n \;-\; 1 \;\times\; n$ 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 $(n \;-\; 1)^{\text{th}}$ 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).

Chapter 8 of the textbook illustrates how to write a recursive subroutine by first showing the shortcomings of the original subroutine protocol described in chapter 6. It starts with code similar to the following:



```
.ORIG x3000

      AND R0, R0, #0
      ADD R0, R0, #5

      LD R5, FACT_ADDR
      JSRR R5

      HALT

FACT_ADDR .FILL x3100
.END

.ORIG x3100
FACT  ST  R1, Save1_3100

      ADD R1, R0, #-1
      BRz DONE
      ADD R1, R0, #0
      ADD R0, R1, #-1
      JSR FACT
      LD R5, MUL_ADDR
      JSRR R5

DONE  LD  R1, Save1_3100
      RET

MUL_ADDR .FILL x3200

Save1_3100 .BLKW 1
.END

.ORIG x3200
MUL   ST R2, Save2_3200

      ADD R2, R0, #0
      AND R0, R0, #0

LOOP  ADD R0, R0, R1
      ADD R2, R2, #-1
      BRp LOOP

      LD R2, Save2_3200
      RET

Save2_3200 .BLKW 1
.END
```

The only difference between the code above and the code from the book is that the non-existent MUL LC3 assembly instruction in the book is replaced with an actual implementation of this operation in a subroutine. However, the fact that the subroutine MUL is called from the subroutine FACT demonstrates perfectly the main shortcoming of the book's original subroutine protocol. It does not allow one subroutine to call another subroutine. The goal of this exercise is to demonstrate this deficiency so that we can explore a fix in the next exercise.

For this exercise, type the code above into the file called lab7_ex1.asm and assemble and run the code in the simulator. What happens? (In order to recover from what happens, hit the pause button in the simulator)

Next, step through the code to discover what happens when one subroutine calls another subroutine, even without any recursion. Remember to use Step-in in the simulator whenever you get to a JSR or JSRR function. If you don't, you'll jump over the subroutine and miss what went wrong. You'll see what went wrong when you try to return from the first call to the MUL subroutine from the FACT subroutine.

When you demonstrate this lab be prepared to explain what went wrong.

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, copy your code from lab 7 exercise 1 above into the file lab7_ex2.asm. Then modify both the FACT and MUL subroutines to properly backup and restore the R7 register as we learned in lab 5. Once you've made this change, assemble and run the code. What happens this time? (You'll need to press the pause button in the simulator again to recover from what happens)

Again, you'll need to step through the code to discover what went wrong. This time this issue happens when returning from the first recursive subroutine call in FACT. Step through the code until you get where the FACT attempts to return from the first recursive call (where FACT calls FACT).

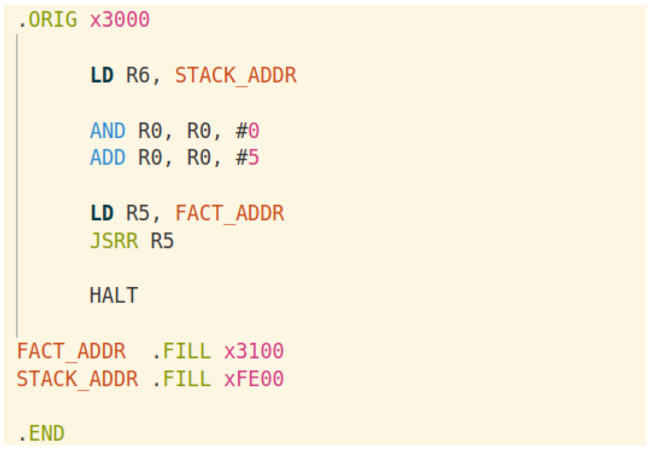
Once more, you'll be expected to explain what happened that caused the program to not work correctly.

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. You learned about how to implement and use stacks in the previous lab. Now you'll use that knowledge to solve a real problem, allowing for recursive subroutines.

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:



```
.ORIG x3000

      LD R6, STACK_ADDR

      AND R0, R0, #0
      ADD R0, R0, #5

      LD R5, FACT_ADDR
      JSRR R5

      HALT

FACT_ADDR  .FILL x3100
STACK_ADDR .FILL xFE00

.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:

```
ST R7, Save7_3100
ST R4, Save4_3100

; Subroutine code in between

LD R4, Save4_3100
LD R7, Save7_3100
```

Which backs up and restores R7 and R4 respectively, becomes:

```
ADD R6, R6, #-1
STR R7, R6, #0
ADD R6, R6, #-1
STR R4, R6, #0

; Subroutine code in between

LDR R4, R6, #0
ADD R6, R6, #1
LDR R7, R6, #0
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 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.

3.2 Submission

Demo your lab exercises to your TA ***before you leave lab***.

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 ***<u>before</u> your next lab***.
Office hours are posted on Canvas, in the "Office Hours" page.