Lab Assignment 4—*What's Your Calling?*
Subroutines, Recursion, and the Stack

Overview:

The *Factorial* of *n* (denoted *n!*) is the result of multiplying integers 1,2,…*n* together. For example, the factorial of four is:

$$\text{Factorial}(4) = 4! = 1 \times 2 \times 3 \times 4 = 24$$

Another way to calculate the factorial of *n* is to multiply *n* by the factorial of *n-1*. Several examples of this are shown below. **Remember that Factorial(1) = 1 by definition**.

$$\text{Factorial}(4) = 4 \times \text{Factorial}(3)$$
$$\text{Factorial}(3) = 3 \times \text{Factorial}(2)$$
$$\text{Factorial}(2) = 2 \times \text{Factorial}(1)$$

The fact that a factorial can be represented as the product of a number and another factorial makes **recursion** an obvious software design choice when calculating factorials. In this lab you will create a **recursive subroutine** to perform Factorial calculations.

Lab Procedure:

Begin by accepting the Lab04 assignment. After the remote assignment repo has been created for you, open **Git Shell** and navigate to your `D:\EECE3624Labs` solution directory. Enter the following command to clone the remote `Lab04` assignment repo to your local storage device.

```
D:\EECE3624Labs> git clone <URL-to-remote-assignment-repo> Lab04
```

Add the new `Lab04` project to your `EECE3624Labs` solution using **Atmel Studio**, and remember to set `Lab04` as the new startup project and `Lab04.asm` as the entry file.

Open `Lab04.asm` for editing, and add the following **assembler definitions** to your code just below the main program header comments (i.e. just before the `.org 0x0000` statement).

```
.def      n = R16
.def result = R17
```

This allows the newly defined names `n` and `result` to be used in the program instead of `R16` and `R17`, respectively, making the code more readable and easier to follow. You should always keep readability in mind when creating programs that others (or future you!) will be using. It's amazing how quickly we forget the decisions we made when designing software, so get in the habit of using descriptive variable names and succinct but expressive comments.

Below the stack pointer initialization code, remove the "`; your code starts here...`" line and replace it with the following lines, typing them in ***exactly as shown below***:

```
        LDI   n, 4     ; load a value into n
        PUSH  n        ; push it on the stack
        CALL  factN    ; calculate the factorial of n
        POP   result   ; pop result off stack
here:   RJMP  here     ; loop forever

factN:
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ; Comments regarding the factN subroutine go here
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ; recursive factorial code begins here

        ; return from the factN subroutine
        ret
```

You will need to write the `factN` subroutine, and you cannot change the "main" (calling) part of the program in any way. As the code above shows, the `factN` subroutine is used to calculate $n!$ by first pushing the value of n onto the stack and then calling the subroutine. The subroutine calculates the factorial of n, placing the result on the stack in such a way that the caller can pop off the value as soon as the subroutine returns. In the code above, this happens when the main code executes the `POP result` instruction, placing the answer into `result` (i.e. `R17`). **Your subroutine must completely conform to this calling mechanism**.

Question 1: What is in the stack right before the CALL?

Question 2: What is in the stack right after the CALL (before the 1st line of the subroutine)?

Question 3: How will we access the parameter that was PUSHed before the CALL?

Question 4: Where will we place the return result so it can be POPped after the RET?

Remember, your subroutine can use registers R0-R31 for holding and manipulating data during a single recursive subroutine invocation, **but not for data used by subsequent invocations**. Also, you cannot use registers to pass data between the main program and the subroutine. Likewise, you cannot use global "scoreboard" SRAM locations either. In other words, **you may only the stack to pass n to the subrouting, to store any intermediate subroutine values, and to pass the final $n!$ result back to the main program**. Each recursive subroutine call must function without any knowledge of what has happened previously or what will happen next.

> **Important**: As you develop your solution, "execute" your algorithm by hand, creating pictures of the stack. This allows you to verify that your algorithm is working correctly and ensures the main code can properly use your `FactN` subroutine.

One more *very important point*. Your subroutine must not "leak" memory. **Memory leaks** occur when programs or subroutines change the stack pointer in such a way that *a portion of stack memory can never be accessed again*. For example, imagine what would happen if a subroutine

copied its own return address from its original location in stack memory to an unused location nearer the top of the stack (i.e. a lower SRAM address), and also changed the stack pointer value to point to the location one address above the newly copied return address. Once the subroutine executed its `ret` instruction, control would return to the caller as expected, **but the stack would forever be "shortened" by this change**. As mentioned in the box above, a great way to prove this to yourself is to simulate (on paper) a program making these types of changes to stack memory and the stack pointer and observe what happens to stack memory. As always, if things are still unclear ask your instructor for help. Memory leaks are another concept you need to understand and avoid. *The best defense against memory leaks is a good offense!*

The last page of this handout shows a pseudocode algorithm for recursively calculating factorials. This is a good *starting point* for your program, but ***it does not show how to use the stack for storing intermediate recursion results***—you'll have to figure that out by yourself! And again, **you aren't allowed to hold intermediate values needed by future invocations of the recursive subroutine in registers or global memory**. Make absolutely sure you understand what this means. It's very **important**! (Translation: Do it wrong and you'll lose **lots** of points!)

One more thing. For this lab, **you do not need to preserve the values of any registers your subroutine uses**. That will shorten your code (see Bonus Opportunity below!)

Once your program is working, determine **the largest value of *n*** that can be calculated correctly and **use that value in your code**. In your report include **screen shots** of the bottom 16 locations of stack memory for each of the following cases:

1. **Just *before* the** `CALL factN` **instruction**. Indicate where *n* is on the stack.
2. **When the recursive "base case" has been reached** (i.e. when `factN` is called to find the factorial of 1). This is the point at which the subroutine has called itself as many times as it's going to, and it will now begin to "unroll" the stack, returning values to previous instances of itself.
   Upshot: I want to see the stack **just before it begins to unroll**.
3. **Just *after* the** `CALL factN` **instruction finishes** (i.e. just before `POP result` executes). Use circles, arrows, etc. to indicate where the result is on the stack.

Report:

This is a **one week lab**. In your report, discuss the **design** of your subroutine and include a **pseudocode algorithm** of your `factN` subroutine and as well as a printout of your assembly program. Be sure your subroutine header includes comments that describe **what the subroutine does and how it can be used by a caller.** (Assume someone will need to use your subroutine without ever contacting you. All the important points about using your subroutine must be explained in the subroutine's header and code comments!)

Submit your lab report (**PDF** version) via Canvas and your code via the GitHub project repo. **Both must be submitted on or before the beginning of lab on Feb 19, 2021**.

Bonus Opportunity: My `factN` subroutine consists of **15 instructions (**don't count instructions in the main program because your main code will be exactly the same as mine). Beat my instruction count and you'll earn some bonus points. To be eligible, you must **explicitly mention the number of instructions in your lab report**. And remember—have fun!

A Recursive Algorithm for Calculating Factorials

re·cur'·sion
noun; Mathematics, Computer Science
1. See "recursion."

The following algorithm uses recursion to calculate factorials, taking advantage of the factorial property fact(n) = n × fact(n-1), for n greater than 1.

```
fact(n)
{
   if n==1
      result = 1
   else
      temp1 = n
      temp2 = fact(n-1)
      result = temp1 × temp2
   endif
   return result
}
```

Study this algorithm to prove to yourself that it works. But remember: **This algorithm does not indicate *how* values are passed to and from recursive calls to `fact(n)` as it repeatedly calls itself**.

Important consideration: Each time `fact(n)` is called (either from the main program the first time, or recursively from previous invocations of the subroutine), each new subroutine call **needs its own set of variables**! A particular `fact()` instance cannot be allowed to use the same variables as the previous/calling instance (why not?). **In other words, global variables are out!** You must design your data structures to ensure each invocation of `fact()` gets its own private data variables—so **you'll need to use the stack**!