Chapter 9 – Recursion

1 Writing recursive functions

In chapter 8 we learned how to add a prologue and epilogue to our functions. This makes it possible to write (and think about) each function separately. Because all functions follow the same calling convention, we can combine them together into a program which will always execute correctly. We picked the simplest convention possible: a called function preserves all its callers' registers.

However, we must adjust this convention to allow our functions to return something. Just like some registers are used to pass **input parameters** (aka arguments), a calling convention designates one or more registers to pass **output parameters** (aka return values) from the callee to the caller. Typically, a calling convention uses the same registers for both directions.

Exercise 1 Write a recursive `factorial` function which receives a positive integer N in R1 and returns $N!$ also in R1. Use the fact that $1! = 1$ and $N! = (N-1)! \times N$ for $N > 1$.

Reminder: Given a positive integer N, its factorial $N!$ is defined as the product of all the positive integers up to N. In other words, $N! = N \times (N-1) \times ... \times 2 \times 1$. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Exercise 2 Write a recursive `fibonacci` function which receives a positive integer N in R1 and returns the Nth Fibonacci number F_N also in R1. Use the fact that $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

2 Passing parameters on the stack

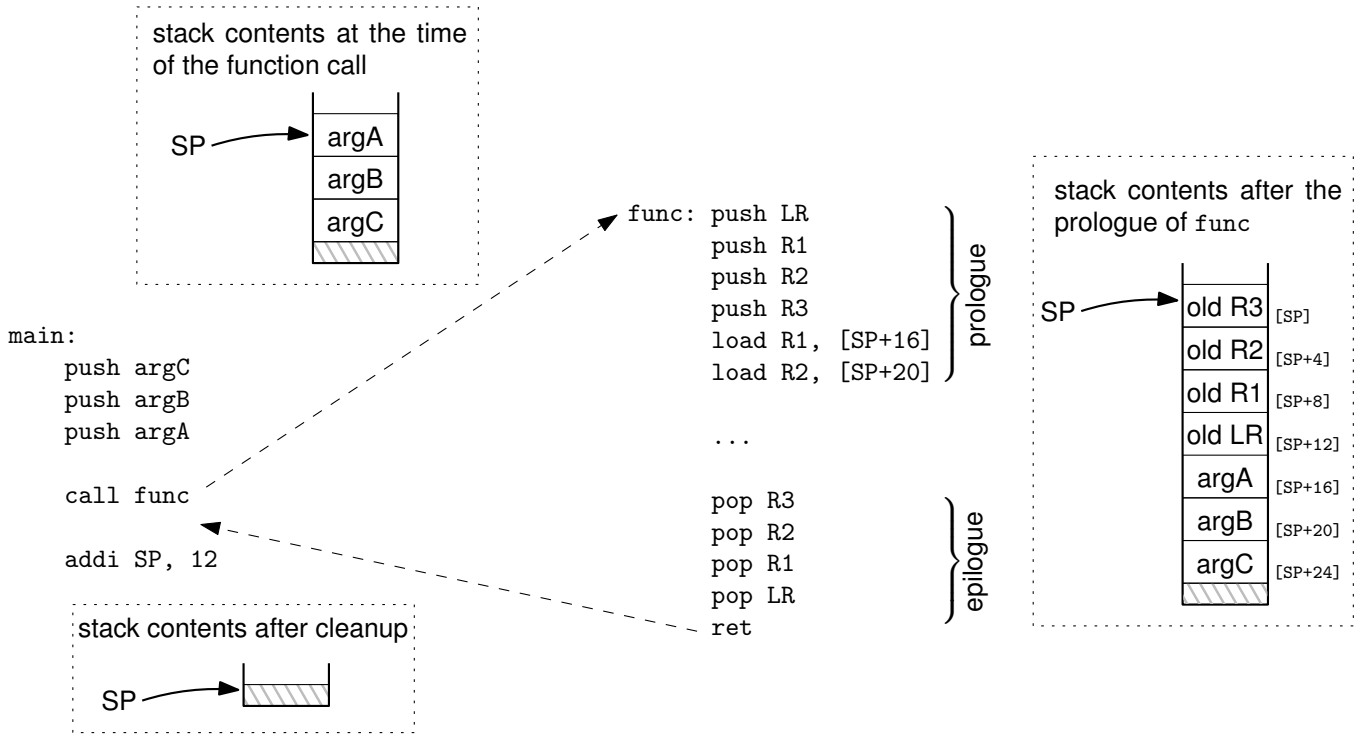
There are situations where it is not possible to assign each parameter of a function to a register. For instance, if there are more parameters than the CPU has registers. Or if the function is *variadic*, i.e. when the number of arguments is not known in advance, like `printf()` in C or in Java. In these cases, we have to use a different calling convention and pass parameters in memory instead.

As illustrated on the diagram below, the idea is to push each argument on the stack before actually calling (jumping to) the function. The prologue of the called function now has two roles:

- **save the context** of the caller on the stack
- **unwrap the arguments** using SP-relative indirect addressing.

These two operations are closely related: the stack offset of each argument depends on how many context registers were just saved.

After the function call has returned, it is the responsibility of the caller to clean up the arguments from the stack. Instead of popping multiple useless values, the usual practice is to simply move the stack pointer beyond the arguments without actually touching them.



Exercise 3 Write a new version of your `fibonacci` function which uses the stack for parameter passing.

Exercise 4 Write a non-recursive, variadic function named `sum` function which receives multiple values on the stack and adds them all up. A "first" argument (i.e. pushed after all the others, see code below) indicates how many values there are. To transfer the return value back to the caller you may use either R1 or the first argument slot in the stack.

```
main:
        ;; value arguments
        leti R1, 42
        push R1
        leti R1, 36
        push R1
        leti R1, 15
        push R1
        leti R1, -5
        push R1
        leti R1, 1
        push R1
        leti R1, 23
        push R1

        ;; number of values
        leti R1, 6
        push R1

        call sum

        addi SP, SP, 28 ;; cleanup 7 stack words
```