Chapter 8 – The Execution Stack

How can we get procedures to call other procedures ?

In SCAT, `CALL`ing and `RET`urning from functions is implemented with the Jump-and-Link instruction. When function A calls function B, `JAL` saves the **return address** into the **Link Register** and then jumps to B. This return address stays in LR during all of B's **activation**, until the "return `JAL`" transfers it back to PC.
But if function B calls another function C, the `JAL` from B to C will overwrite the contents of LR and the return address back to A will be lost. Unless we save it to memory.

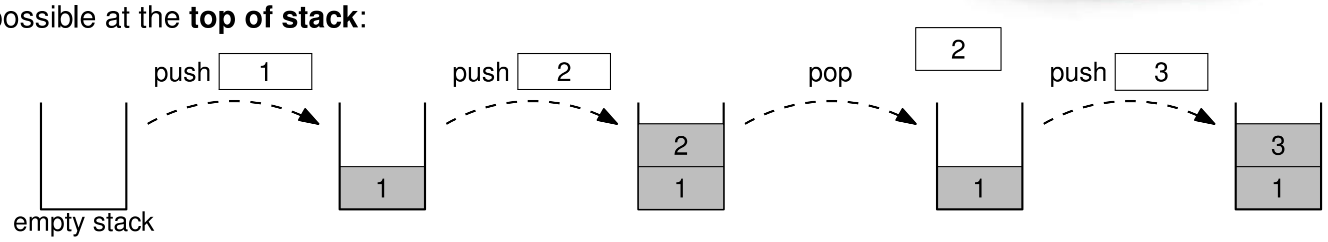
1 Introduction

Idea At runtime, function activations are always "nested" in a Last-In First-Out order. If in the past A called B and later B called C, then we know that C will return to B before B will return to A. Accordingly, we need a Last-In First-Out memory data structure in which to save and restore return addresses.

Definition In computer science, a data structure with such LIFO access is known as **a stack**, i.e. a collection of elements, offering just two operations:

- **push** which adds an element to the collection,
- **pop** which removes the most recently pushed element.

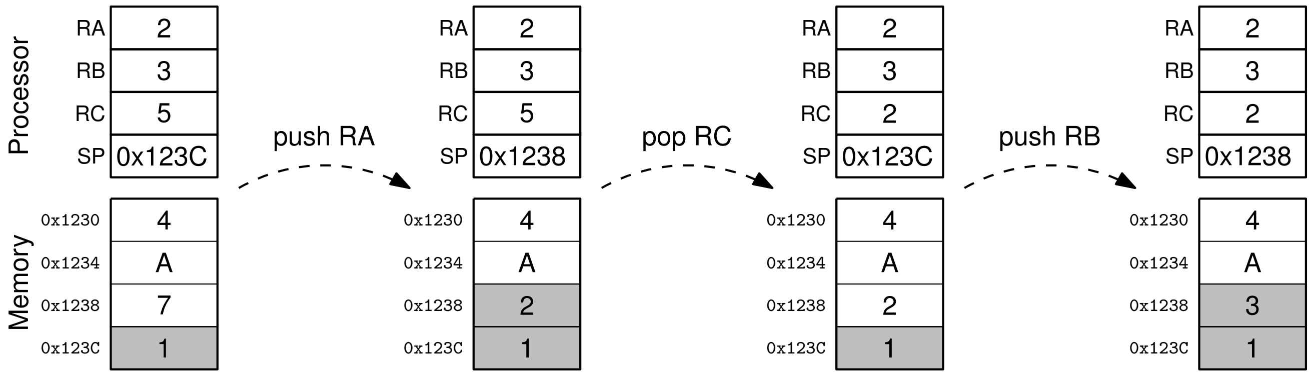
Similar to a stack of plates, adding or removing elements is only possible at the **top of stack**:



1.1 Implementation in processors

Because it is so simple to implement, basically all computer architectures support stack operations natively:

- Stack elements are stored in memory, just like in a partially-full array.
- One of the CPU registers is reserved to always contain the current address of the top of the stack. We call this register the **Stack Pointer** SP.
- Push and pop are implemented directly as processor instructions, as illustrated below.



Remarks

- A stack element is always the same the size as a CPU a register.
- Popping from the stack does not erase memory: the “pop RC” instruction leaves the “2” in place.
    - By convention, all memory contents “above” the top of stack is considered noise. In our example, values “4”,“A” and “7” have no meaning, they are just leftovers from previous computations.
- By definition, SP points to the topmost element of the stack. But when the stack is empty there is no such element. In general, there are two possible conventions to represent an empty stack:
    - keep an additional dummy value at the **stack origin**,
    - allow for SP to point one word beyond the stack origin.

  Both conventions are equivalent: when the stack is empty, it is a programming error to pop from it.

2 Stack instructions in SCAT

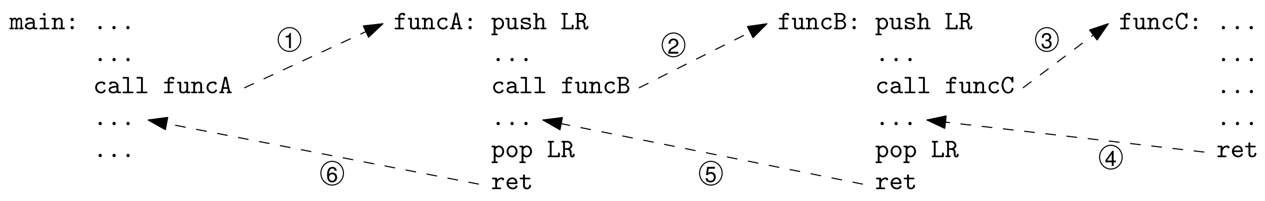
Our assembler uses R13 as SP and implements stack operations as pseudo-instructions:

- `push reg` is assembled into:
  ```
  subi r13, r13, 4
  store [r13], reg
  ```
- `pop reg` is assembled into:
  ```
  load reg, [r13]
  addi r13, r13, 4
  ```

Programs using these instructions should initialize SP to point somewhere in memory, far beyond the end of the executable, for example with `leti SP, 0x10000000`.

3 Adding a prologue and epilogue to functions

To implement nested functions calls, we can **save the return address on the stack** before overwriting LR. Then we restore it when needed, i.e. just before returning. This idea is illustrated below: left-to-right arrows represent function calls, and right-to-left arrows represent function returns. Observe that `funcA` and `funcB` must save/restore LR, but not `funcC`, because `funcC` never changes the value of LR.



```
main: ...                 funcA: push LR            funcB: push LR           funcC: ...
      ...                        ...                       ...                      ...
      call funcA                 call funcB                call funcC               ...
      ...                        ...                       ...                      ...
      ...                        pop LR                    pop LR                   ret
                                 ret                       ret
```

We can even take this idea one step further and save/restore all touched registers on function entry/exit. This way, the calling function (the **caller**) does not have to worry about its registers being clobbered by the function being called (the **callee**). We are effectively designing a **calling convention** where the caller’s registers are **preserved across function calls**.

In the general case, any function is going to be structured in three code sections, like illustrated below:

```
myfunc: push LR
        push R1
        push R2
        push R5

        ...
        ...
        ...
        ...

        pop R5
        pop R2
        pop R1
        pop LR
        ret
```

function prologue: save all registers that we are going to clobber

function body: unrestricted code (possibly calling other functions)

function epilogue: restore all saved registers then exit

4 Practice time

Exercise 1 Modify your `drawpixel` function from last session so that it preserves the caller's registers.

Exercise 2 Write a `drawrectangle` function which draws a rectangle by calling `drawpixel` multiple times. There are five parameters to `drawrectangle`: X/Y coordinates of two opposite corners, and a color argument.

Exercise 3 (optional & hard) Write a generic `drawline` function which draws a line between any two screen positions. The idea is to implement Bresenham's algorithm:

```
drawLine(x0, y0, x1, y1, color)
    dx = abs(x1 - x0)
    sx = x0 < x1 ? 1 : -1
    dy = -abs(y1 - y0)
    sy = y0 < y1 ? 1 : -1
    error = dx + dy

    x = x0
    y = y0
    while true
        drawPixel(x, y, color)
        if x == x1 && y == y1 break
        e2 = 2 * error
        if e2 >= dy
            if x == x1 break
            error = error + dy
            x = x + sx
        end if
        if e2 <= dx
            if y == y1 break
            error = error + dx
            y = y + sy
        end if
    end while
```