Chapter 7 – Programming with subroutines

How to avoid copy-pasting similar pieces of code in many places ?

Idea It is possible to break a program down into several sub-programs and execute each sub-program when needed. This allows for code reuse, because a sub-program (also known as a **subroutine**, a **function**, a **procedure**, a **method**, or many other names) can be written once and then **invoked** (aka **called**) several times from various locations in the main program.
In terms of implementation, this mechanism is very similar to the branching instructions from chapter 4: the processor can *jump* to another location in the program just by altering its program counter. However, there is a major difference: after a sub-routine is finished, we want the CPU to **return** (i.e. jump back) to where it was before and continue from there.

There are various strategies for saving this **return address**: some architectures (like x86) save it to memory ; others (like ARM, or SCAT) save it to a CPU register.

1 Subroutines in SCAT

Our assembler offers two instructions to write subroutines: `CALL` and `RET`, illustrated below. `CALL label` saves the address of the next instruction into register R14, then jumps to `label`. Conversely, `RET` copies the contents of R14 back into PC. Because this register helps us *link* different procedures together, we will call it the **Link Register** LR.

```
main:
        leti r1, 0
        leti r2, 0
        call drawpixel

        leti r1, 0
        leti r2, 59
        call drawpixel

        leti r1, 79
        leti r2, 0
        call drawpixel

        leti r1, 79
        leti r2, 59
        call drawpixel

        bra +0

drawpixel:
        muli R1, R1, 4       ; horiz: 4 bytes per pixel
        muli R2, R2, 320     ; vertical: 320 bytes per line
        add  R3, R2, R1
        leti R4, 0xB0000000 ; VRAM base address
        add  R5, R4, R3
        leti R6, 0xFF00FF00 ; RGB hex triplet for magenta

        store [r5], r6

        ret
```

Exercise Retype the program above in a text file, assemble it and then execute it step-by-step in the simulator. Observe how the CPU saves the return address to LR at each function call, and how `RET` jumps back to the instruction immediately following the call site.

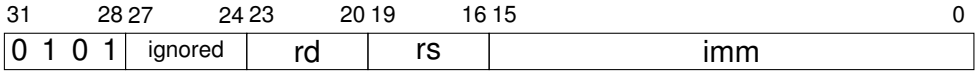
2 Implementation

In SCAT, both `CALL` and `RET` are pseudo-instructions. The processor only knows about a single machine instruction named Jump-and-Link, or `JAL`:

`asm`	Name	Description
`jal rd, rs, imm`	Jump-And-Link	`rd = PC+4 ; PC = rs + sxt(imm)`

When it encounters a function call, the assembler generates a Jump-And-Link with R14 (aka `LR`) as destination register and PC as a source register. The offset is computed as the distance between the current instruction (i.e. the invocation site) and the address of the function. Conversely, returning from a subroutine is simply achieved with `JAL R0, LR, +0`.

The binary format of Jump-And-Link is illustrated below:



3 Practice Exercises

Exercise Modify the `drawpixel` procedure from the previous page so that it receives three input parameters: X/Y coordinates in R1/R2, and the desired color in R3. Then modify the main program so that each corner of the screen gets painted with a different color.

Exercise Write a `maximum` function which receives two numbers in R1 and R2, finds the largest one of the two and returns it in R3. Write a main program which calls this function several times with different parameters.

Exercise Take your graphical program from chapter 6 (bubble sort visualization, or bouncing ball) and rewrite the code so that is uses procedures.