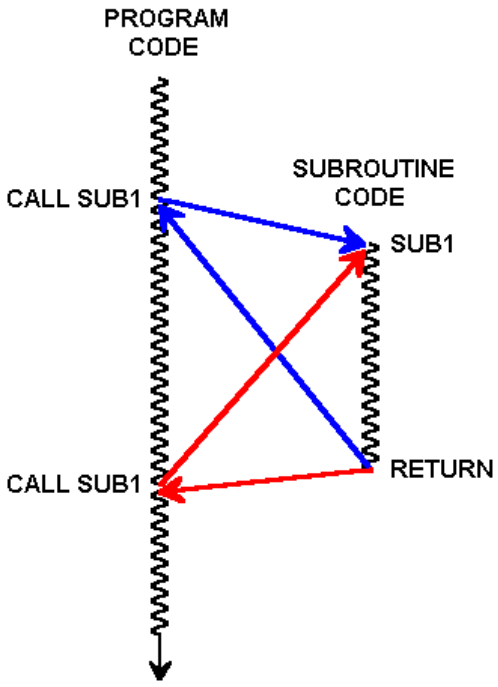
# Lecture #9 – February 18, 2004

## Introduction to Subroutines

In all computer systems there is a hardware mechanism to support ***subroutines***, which are roughly equivalent to procedures in a high-level language. A subroutine is a section of the program code with an entry point (an address) at the beginning of the section and a special return mechanism at the end. When a subroutine is called from some program location, the return mechanism must jump back to the instruction after the call. The return mechanism cannot be a simple branch or jump, because the return point changes depending on where the subroutine was called from. When a subroutine is called, the hardware must somehow store the current return address in a way that it can be retrieved at the end of the subroutine. In the diagram below, the subroutine code is shown off to the side of the main program, but in reality it is in the same memory space as the rest of the program, at some point later on in the stream of program bytes.




In older machines such as the DEC PDP-8 and the CDC-3300 and CDC-6000 series, the return address of a subroutine call was stored in the first word of the subroutine, and the first executable instruction of the subroutine was in the second word. When a subroutine is called, the hardware automatically stores the return address in the first word and jumps to the second. At the end of the subroutine (or at any point where the appropriate action is to exit the subroutine early) the instruction to exit is to ***jump indirect*** through the beginning word of the subroutine. A jump indirect instruction says to go to the address specified in the instruction and then use the *contents* of that word as the actual target address of the jump (i.e., "I don't know where to go, but I know who does").

This process is outlined in the following code fragment. The opcodes shown are not for any particular computer, but are "generic" in some sense. The `JMS` opcode means "jump to subroutine", `DW` means "define word" (i.e., reserve a word of memory), and `JMP I` means "jump indirect".

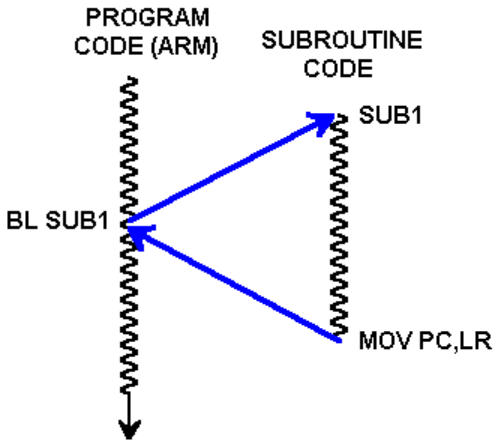
```
        …
        JMS SUB1        Subroutine call.
        …               ←   The address of this
                            instruction is stored
                            in the first word of the
                            subroutine.

---------------

SUB1 DW   0             Return address placeholder
     …                  ←   1ST actual instruction
     …
     JMP I SUB1         Return from subroutine
```

In this approach there is a unique memory location at the start of every subroutine to store the return address for that subroutine. Thus, it is very easy for one subroutine to call another, creating a ***nested call***. While it is simple, easy to implement in hardware, and fast to execute, this call-return mechanism normally does not support ***recursion*** (a subroutine that calls itself). To allow for recursion, an assembly language programmer must write code in every recursive subroutine to explicitly push the return address onto a stack upon entry and pop it upon exit.

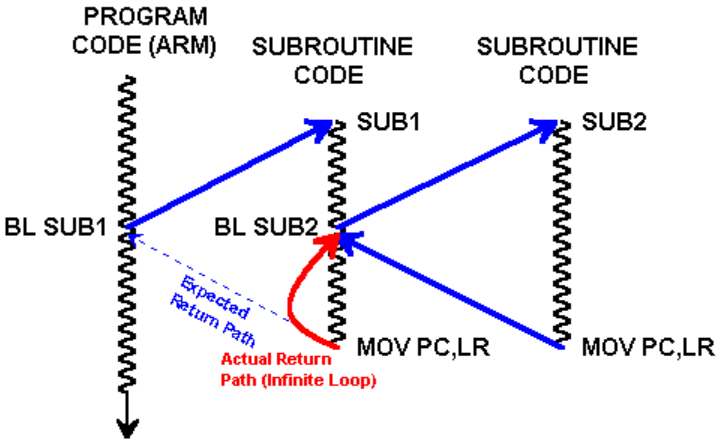
In the 6502 there is a hardware stack, fixed in memory at page 1, and on the 8088 the `SS` segment register is set by each program to point to the memory block containing the stack for that program. In both cases there are instructions that automatically push the return address on the stack during a subroutine call and pop it off during a subroutine return. The 6502 instructions are `JSR` (jump to subroutine) and `RTS` (return from subroutine), while on the 8088 the corresponding instructions are `CALL` and `RET`. By using the stack to hold return addresses, recursive subroutine calls are as easy to implement as are nested subroutine calls.

The 8088 has an extra complexity in that there are *two* sets of `CALL` and `RET` instructions; one set for "near" calls to addresses within the same code segment as the call and one set for "far" calls to an address in another segment. Near calls push only the address offset (two bytes) of the current code segment onto the stack, while far calls push both the address offset and the address segment (four bytes total) onto the stack. It is critical that the assembly language programmer insure the type of return instruction matches the type of call; pushing two bytes and popping four or pushing four and popping two both cause subroutines to return to the wrong address in the wrong segment (as well as leaving the stack in an unacceptable state).

On the ARM the subroutine call-return mechanism falls somewhere between the two methods listed earlier. While the ARM instruction set *does* support hardware stack operations, calling a subroutine *does not* automatically push the return address on the stack, and subroutine calls *cannot* be nested without the programmer paying special attention to the return address. Instead, the call instruction, called `BL` for ***branch and link***, automatically stores the return address of the subroutine into register `R14`. Register `R14` is also known as the ***link register***, and assemblers use the synonym `LR` for `R14`. For simple subroutines, returning to the calling program requires only that the programmer move the contents of `R14` into `R15`, or `MOV PC,LR`, as shown below:



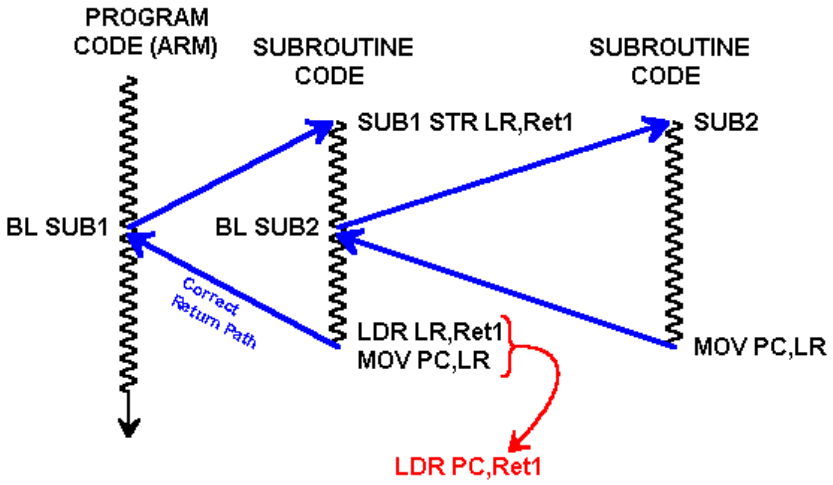

A naïve programmer might attempt to nest subroutine calls on the ARM without thinking about what happens to the return address. While calling `SUB1` from the main program puts the return address to `SUB1` in `LR`, calling `SUB2` from within `SUB1` replaces `LR` with the return address to `SUB2`. The return address to `SUB1` is lost. Subroutine `SUB2` works correctly, but subroutine `SUB1` returns back into itself, creating an infinite loop. This is shown below:




While we will examine use of the stack in a future lecture, the simple approach to nesting subroutines on the ARM is to reserve one word of memory for each subroutine and use that location as temporary storage for the return address. In some sense this is

simulating the mechanism of the PDP-8 and CDC-3300, but where the programmer is responsible for saving the return address to and restoring the return address from that memory location.

In the following diagram, subroutine `SUB1` has an associated memory location called `Ret1` (defined by the programmer). The first task of the subroutine is to store the contents of `LR` into `Ret1`, and the last task before the subroutine returns is to load `LR` from `Ret1`. Notice that the pair of instructions that end the subroutine can be combined into one: `LDR LR,Ret1` followed by `MOV PC,LR` can be combined into the single instruction `LDR PC,Ret1`.




Subroutine `SUB2` need not save and restore its return address since it does not call any other routines and thus never damages the contents of the `LR` register.