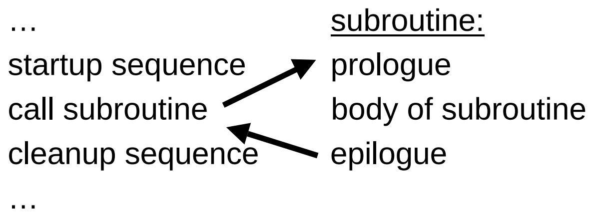
Subroutines

- A *subroutine* is a generic subprogram, including both functions and procedures
  - *Function* = returns a value
  - *Procedure* = does not return a value
- We can illustrate a subroutine call as follows:



  - Note that, in a high-level language such as C, the startup sequence, prologue, epilogue, and cleanup sequence are essentially non-existent

Simple Subroutine Calling

- So that a subroutine knows where to return, the return address is stored in a register, and used by the subroutine

```
        …
        MOVE R31,#ret    ; store return addr
        JUMP  sub1
ret:                     ; continue prog.
        …
sub1:   …                ; contents of sub1
        JUMP  @R31       ; return to caller
```

  - This *subroutine calling convention* uses R31 to hold the return address
  - The JUMP instructions above use either:
    - Direct addressing
    - Register indirect addressing
  - Some machines support a jump-to-subroutine instruction that combines the MOVE and JUMP above

```
    JSR      R31,sub1
```

Saving Return Address in Memory

- To allow hierarchical subroutines, the return address must be stored in memory

```
        …
        JSR     R31,sub1
        …                   ; continue prog.
sub1:   STORE s1ret,R31; save return addr
        …                   ; contents of sub1
        JSR     R31,sub2
        …
        LOAD  R31,s1ret  ; restore ret addr
        JUMP  @R31        ; return to caller
sub2:   STORE s2ret,R31; save return addr
        …                   ; contents of sub2
        LOAD  R31,s2ret  ; restore ret addr
        JUMP  @R31        ; return to caller
        .bss
s1ret:  .reserve    4
s2ret:  .reserve    4
```

  - Subroutine sub2 is a *leaf subroutine* — it does not call any other subroutine
    - Can it be improved on?

Saving Registers in Memory

- Since the subroutine doesn't know which registers the routine calling it is using, it must save (in memory) the contents of any registers it will use, and restore those values before returning

```
sub1:   STORE s1ret,R31; save return addr
        STORE s1r0,R0    ; save registers
        STORE s1r1,R1
        …
        LOAD  R0,s1r0     ; restore registers
        LOAD  R1,s1r1
        …
        JUMP  @R31        ; return to caller
        .bss
s1ret:  .reserve    4
s1r0:   .reserve    4
s1r1:   .reserve    4
        …
```

  - It might be useful to have subroutines available to save/restore all registers

Passing Parameters

- To pass parameters to a subroutine, or return a result, we must establish a *parameter passing convention*
- Example convention:
  - R2 = returned result
  - R3–R30 = parameters
  - R31 = return address
- Parameter passing by value / result
  - Caller passes *values* to subroutine in registers R3–R30
  - Subroutine can modify those values to pass *results* back to caller
  - Caller might then need to store returned values in memory
  - Good for passing simple variables only

Passing Parameters (cont.)

- Parameter passing by value
  - Same as passing by value / result, but subroutine shouldn't modify the values
  - Good for passing constants
- Parameter passing by reference
  - Caller passes *address of* (pointers to) *values* stored in memory to subroutine in registers R3–R30
  - Subroutine can use indirect addressing to access those values, and modify them as necessary to pass result back to caller
  - Caller no longer needs to store returned values in memory (they're already there)
  - Good for passing complex data structures (strings, arrays, structures)

Static Allocation of a Parameter Block

- One deficiency of the parameter-passing methods introduced so far:
  - You can't pass more parameters than you have registers available
- Solution:
  - Allocate a parameter block in memory (a collection of contiguous memory locations) to store all parameters
    - Either values, or pointers to values
  - Keep track of offset of each parameter from the beginning of the block
  - Save the return address and any return values in the parameter block as well

Dynamic Allocation of a Parameter Stack

- More deficiencies of the parameter-passing methods introduced so far
  - At any point in time, memory is allocated for storing registers for every subroutine, whether that space is currently needed or not
  - There is no provision for recursion
- Solution (skim details in Section 6.5):
  - Use a stack (in memory) to dynamically allocate space to store parameters passed to subroutines
  - At any point in time, only active subroutines are using space on the stack
  - Stack frame (part of stack corresponding to a particular subroutine call) contains:
    - Parameters being passed
    - Return value & return address
    - Space for local variables