Subroutines

In assembler programming there only one difference between a subroutine and any other named block of code:

→ a subroutine end with a RET (return opcode)

The RET opcode causes an immediate jump to whatever 16 bit address is on the top of the stack (the top two bytes):

→ usually this is the return address pushed onto the stack by the ACALL/LCALL subroutine call.

IMPORTANT

The stack pointer must be the same position at the end of a subroutine call as at the beginning

→ still pointing at the return address

You can use the stack freely within a subroutine BUT the number of PUSH/POPs must balance

This the ONLY difference.

Note

There:

• is no mention of parameters, local variables or return values from functions,
• are no parameter passing instructions

Parameters & Return Values

• parameters are values made available by a calling routine for a subroutine to use
• return values are results from a subroutine

ANY CONVENTION that is adhered to by both calling code and called subroutine is workable:

Fixed memory locations could be used for parameters and return values:

negP1 equ 55 ; Negate Parameters
negR1 equ 56 ; Negate Return value

main lcall getvalue
    mov negP1,A
    lcall negate
    mov A,negR1
    ...

negate mov A,negP1 ; get parameter
    com A ; negate
    inc A
    mov negR1,A ; Store rtn value
    ret

This works but is both long, error-prone, and makes it difficult to use memory efficiently:

→ Can negP1 and negR1 be used safely by other routines?

Parameter Passing Conventions

Where possible, parameters and return values are usually passed in either registers, or on the stack.

Pass in Registers if:

• number of parameters is small
• size of parameter will fit

Use specific registers as required by the architecture

• use A, B & R2-R7 for numeric values
• use R0 & R1 for pointer values (addresses)

This is simply to clarify the intent, it's perfectly possible to pass pointer (address) values in A & B

; Clear Memory
; Start address in A, Stop address in B
clrMem: mov R0,A
    move adr to ptr reg
    mov @R0,#0       clear that byte
    cjne A,B,nxt      done last byte?
    sjmp negate      yes, exit
    nxt: inc A       no, point at next byte
    sjmp clrMem      and go around again
    done: ret

;============= Sample call to clrMem ===============
main: mov A,#$20   ; Put start adr in A
    mov B,#$30   ; and End adr in B
    lcall clrMem ; Go clear these bytes

Return Address Count Example

When a subroutine call starts executing, the address to return to will be on the top of the stack.

Version 1

; Absolute value function
ABS:  jnb Acc.7,xit ; If +ve, exit
      com A ; if -ve, negate
      inc A
    xit: ret

Negate com A ; if -ve, negate
      inc A
    xit2: ret

Version 2 - not good style but correct:

; Absolute value function
ABS:  jnb Acc.7,xit ; If +ve, exit
      sjmp negate
    xit: ret

; Negate Function, negate the value in Acc
Negate com A ; if -ve, negate
      inc A
    xit2: ret

When ABS is called only one RET is encountered.
Passing Parameters on the Stack

Passing parameters on the stack is often used but requires extra addressing modes to be able to be used efficiently:

**Advantages**
- Memory is reused automatically
- Recursion is automatically available
- Each invocation has its own set of parameters

**Disadvantages**
- Push/pop can take extra time
- Requires additional addressing modes to work around the return address
- *Exercise* – try passing parameters to clrMem on the stack

Who Pushes and Pops what?

Again, it's just a convention, but whoever pushes a value on the stack, should remove it:
- The caller pushes, parameters
  - The caller (on subroutine returns) remove them

Local Variables

- These can again be,
- In registers
- In fixed memory locations
- In the stack

The choice is up to the programmer

---

### Using the Stack for Local Variables

#### Recursive Evaluation of Factorial

Factorial 5 = 5 x 4 x 3 x 2 x 1 = 5 x Factorial(4)

```assembly
org 0
main:  mov a,#4       ; put parameter in A
        lcall fact     ; find Fact(a)
        jmp end
end:
```

#### Parameter

- *Parameter is in Acc*
- Return value will be in Acc
- While FACT(n-1) is evaluated, save n on the stack

```assembly
fact:  cjne a,#1,eval ; if a = 1
        sjmp done      ; yes, return 1
eval:  push a               ; else, save a
        dec a               ; a := a - 1
        lcall fact         ; acc = fact(a - 1)
        pop b               ; retrieve a
        mul ab              ; acc = a x fact(a - 1)
        done ret          ; return
```

Notice (on the next slide) that while FACT is being evaluated, stack holds all the earlier values of a, until a becomes 1, then the recursion unwinds.

---

### Memory while Evaluating FACT(5)

The snapshot is just after FACT has been called five times and the final call (which hasn't yet returned) is to FACT(1):

It's easier to see what's happening if we highlight the values on the stack:

---

### clrMem - Passing Parameters on the Stack

```assembly
main:   mov A,#$20 ; Put start addr in A
        push A
        mov A,#$30 ; and End addr in B
        push A
        lcall clrMem ; Go clear these bytes
        dec sp ; cut off parameters
        dec sp
```

; Clear Memory

- Parameters are on the stack
- Start address (1 byte) was pushed first
- Stop address (1 byte) was pushed second
- When subroutine is started, stack will look like:

  **Stack Contents Offset from SP**
  - SP → High byte of return address
  - Low byte of return address
  - Stop Address parameter
  - Start Address parameter

```assembly
clrMem:  mov A,sp    ; Get Stack pointer
         clr c      ; Subtract 2 to get
         subb a,#2 ; address of
         mov r0,A ; stop parameter in R0
         mov B,@r0 ; get stop addr into B
         dec r0 ; point 1st parameter
         mov A,@r0 ; get start addr to A
         lup:    mov r0,A ; get next addr to r0
                  mov @R0,#0 ; clear that byte
                  cjne A,B,nxt ; done last byte?
                  sjmp done ; yes, exit
                  nxt:     inc A ; no, point to next byte
                            sjmp lup ; and go around
                  again
                  done:    ret
```

---

159.253 - Subroutines 5
Copyright 2005 Giovanni Moretti