| 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 12:18 | 103 | 1:17 | 1:36 | 1:51 | 1:56 |

# Modular Programming

## characteristics of a good module

- loose coupling : data within module is independent of other modules (local variables)
- strong modularity : should perform a single logically coherent task

## subroutine

- module: C :: subroutine : assembly
- caller : main prog that calls the subroutine ↑ callee

how to transfer control and return?

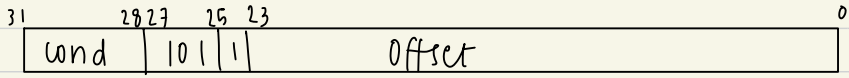
- BL : branch with link
- BX branch and exchange

why not B ?

BL : preserves the address of where we have to return
B does not.
B overwrites value in PC, old value is lost. having another B at the end of the subroutine requires knowledge of the address to return to.
↑ plus that subroutine gets hardwired to the caller at that point. ∴ can't be called again w/o returning there

- BL

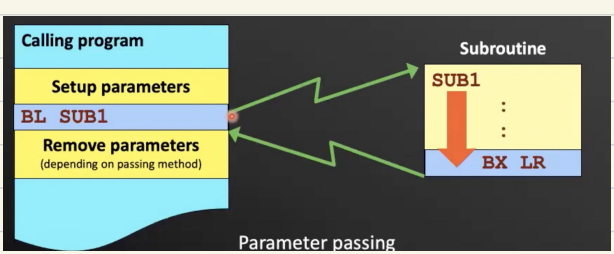
a) return address (PC+4), is stored in link register (R14) ← must go to next instruction




b) subroutine address is stored in PC

- BX LR = MOV PC, LR
takes what is in LR, stores it in PC

Parameter Passing
- 1. reg
- 2. memory block (add. is known by caller)
- 3. system stack




← steps

## 1. Register ← no of parameters is small

- parameters are put in reg before calling
- number of reg are less
  ↓
  only R0 - R12 can be used

| R0 - R3 ← main | R4 - R11 ← local |
|---|---|
| passes values<br>accepts returns<br>values can be modified | · holds local variable<br>· must be preserved in subroutine<br>· untouched values |

R12 : scratchpad register, not preserved, can be used for returns

- disadvantages

→ lacks generality due to limited number of reg

- advantage :

→ efficient, can be read immediately

prog. example: count number of 1's

- return to R0
- parameter in R1

check each bit ⇒ rotate, store in Carry
32 times

name →

```
Count1s  MOV   R0, #0        1 counter
         MOV   R2, #32       loop counter
Loop     RRXS  R1, R1        rotate right + extend.
         ADC   R0, R0, #0    R0 = R0 + C
         SUBS  R2, R2, #1    decrement counter
         BNE   Loop          branch if not equal zero
         MOV   PC, LR
```

affects carry

return

∴ R1 maybe ruined.

↓ improve

```
Count1s  EOR   R0, R0, R0          clears R0
         ADD   R2, R0, #32         counter
         ADD   R3, R0, #1          R3 = R0 (=0) + 1 = 1
Loop     AND   R4, R3, R1, ROR R2  rotates R1 by R2, ands w/ R3, stores in R4
         ADD   R0, R0, R4
         SUBS  R2, R2, #1
         BNE   Loop
         MOV   PC, LR
```

mask bit

R4 has LSB

## 2. Memory

- region in memory is treated like a mailbox and is used by calling program and sub routine
- all parameters to be passed are gathered into a block at a pre-defined memory location
- start address of that block is passed to the subroutine via an address register

program example: lower to upper string

terminated by null, convert by subtracting 32 from ascii

Calling prog:

```
        :
        MOV  R0, #0x100     <- add.
        BL   Lo2Up
```

if each char takes only a byte

| Address | Memory |
|---|---|
| 0x100 | "a" |
| 0x104 | "p" |
| 0x108 | "p" |
| 0x10C | "l" |
| 0x110 | "e" |
| 0x114 | 0x000 |
| 0x118 | : |

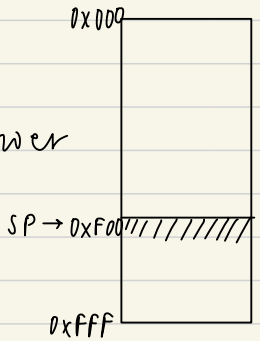
```
Lo2Up   STMFD   SP!, {r0, r1}     save reg used in subroutine.
Loop    LDRB    R1, [R0], #1      shift + read pointer
        CMP     R1, #0            checking if we reached
        BEQ     Done              null, if so, leaves
        CMP     R1, #0x061        } 'z' < char < 'a', go to next
        BLT     Loop              } char
        CMP     R1, #0x7A         }
        BGT     Loop              }
        SUB     R1, R1, #32       <- convert
        STRB    R1, [R0, #-1]#-1] <- store back to memory
        B       Loop
Done    LDMFD   SP! {r0, r1}      <- restore saved registers
        MOV     PC, LR            <- return from subroutine
```

| 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|
| 2:20 | 2:20 | 2:56 | 2:57 | 3:04 | 3:07 |

supports recursion →

## 3. Stack / System Stack

- Stack is FILO/LIFO data structure maintained in the memory data area
- Stack pointer = R13
- Stack can grow towards higher or lower memory address
- SP points to valid data
  ∴ to add, move and add.




- push data.
  `STR R0, [SP, #-4]!`
  ← move to lower memory add.
  pre-indexing..
- pop data.
  `LDR R0, [SP], #4`
  data is not erased from the memory
  post-index, move it down to higher memory location
- pushing multiple
  ∵ stack is descending, in memory R0 then R1
  R1 goes first, then R0
  `STMFD SP!, {R1, R0}`
  Store multiple fully descending
  ↑ update stack pointer
- pop multiple
  `LDMFD SP!, {R1, R0}`

Parameter passing using stack.

calling prog:

Set up parameters
BL sub1
Remove parameters

SUB 1
get parameters
MOV PC, LR

important.

10MHz → $\frac{1}{10 \times 10^6} = 0.1\ \mu s$

this was 45
4.5 μs

3.4

transparent subroutine doesn't affect CPU resources

program example : sum from 1 to N

should be updated to memory variable Answer → 0x100

value passed

pass by value
pass by ref.

```
MOV    R1, #5          store N
MOV    R0, #0x100      add of answere
STMFD  SP!, {r1, r0}   add values to stack to preserve
BL     Sum1N
ADD    SP, SP, #8      add 8 to pop items. they aren't
LDMFD  SP!, {r1, r0}   erased though.
```

either work

sum address pointer

```
Sum1N  ① STMFD  SP!, {R4, R5, R6}   save reg to stack
         LDR    R5, [SP, #16]       N R1 to R5
         LDR    R6, [SP, #12]       dest ans to R6
         MOV    R4, #0              or FOR R4, R4, R4
```

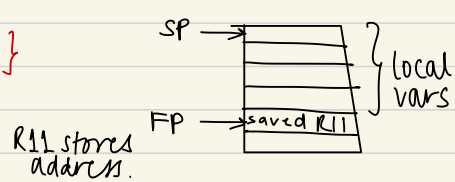
R4-R6

creates stack frame

```
Loop     ADD    R4, R4, R5
         SUBS   R5, R5, #1
         BNE    Loop                if Z=0 loop again
         STR    R4, [R6]
```

stack frame has the local variables

```
         ② LDMFD  SP!, {R4, R5, R6}
           MOV    PC, LR
```




R11 stores address.

Can have a frame pointer (FP)

↑ generally R11, points to beginning of frame.

3:10

| 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|
| 3:19 | 3:26 | 3:35 | 3:45 | |

3:50

## nested subroutine

- When branching multiple times, how to ensure it will finally return to the right place

∴ store LR somewhere safe ⟹ system stack

prog example : dot product.

indexing will change if LR here ← SP+16

R4-R7, LR

```
DotProb  STMFD   SP!, {R4-R7}      store oldies
         LDR     R4, [SP, #28] 32  loc X
         LDR     R5, [SP, #24] 28  loc Y
         LDR     R6, [SP, #20] 24  array length

         MOV     R7, #0
Loop1    LDR     R0, [R4], #4      get X[i]
         LDR     R1, [R5], #4      get y[i]
xpush    STR     LR, [SP, #-4]!
         BL      Mult              store result.
xpop.    LDR     LR, [SP], #4

         ADD     R7, R7, R12
         SUBS    R6, R6, #1
         BNE     Loop1

         LDR     R4, [SP, #16] 20  read dest address
         STR     R7, [R4]          place sum in

         LDMFD   SP!, {R4-R7}, PC  restore oldies
         MOV     PC, LR
```

SP is restored so none of the indexes change.

pop here directly

# recursion

```
Recur   STMFD   SP!, {...., LR}

        BL      Recur
        :

Done    LDMFD   SP!, {...., LR}
        BX      LR
```

stopping condition; return address stored in the main fr.
?
o