Chapter 6

MODULAR PROGRAM DESIGN

- Software decomposed to several less complex modules
  - → Modules can be designed & test independently
  - → Modules can reduce overall program size
  - → General Modules can be re-used in other projects

CHARACTERISTICS OF A GOOD SW MODULE

- Loose coupling → data within module is entirely independent of other modules (local variables)
- Strong modularity → should perform a single logically coherent task

SUBROUTINES

- Modules are implemented as subroutines
- called from various parts of program
- Caller : Main program → issue request to callee to execute subroutine
- Callee : subroutine
- On completion, subroutine returns control to the caller
  - → Exactly after subroutine was called
- Calling & returning from a Subroutine
  - → To go to subroutine (SUB1) : BL SUB1 (BRANCH WITH LINK)
  - → To return to caller program: BX LR (BRANCH AND EXCHANGE)

* Before / After BL, need to store where program should be getting back to

WHY BL AND BX AND NOT B?

- Main program branches to subroutine:
  - → Can be done with B
    - ⇒ Drawbacks: Does not preserve where to go back to; just takes destination address & goes there → no info on where it was before / next address.
  - → B overwrites value in PC
    - ⇒ old PC value in main program is LOST
  - → to use, add another branch at end of subroutine → ineffective approach
    - ⇒ may mean subroutine linked to caller
    - ⇒ need to know exact mem location during compilation
    - ⇒ if another program wants to use subroutine, need to create new one

BRANCH WITH LINK (BL)

⇒ make subroutine call

⇒ return address (BL inst. location +4) [aka points to next instruction] is stored in the link register (R14)

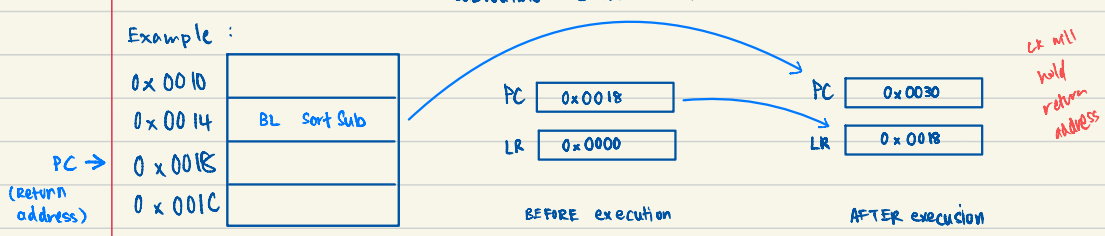
⇒ Link register: stores return address so that can go back to main program without any problems

SUBROUTINE CALL
BL Sort Sub

Execution Sequence:

1. Return address (BL inst. location +4) is stored in link register (R14)
2. Subroutine address stored to PC

Example:

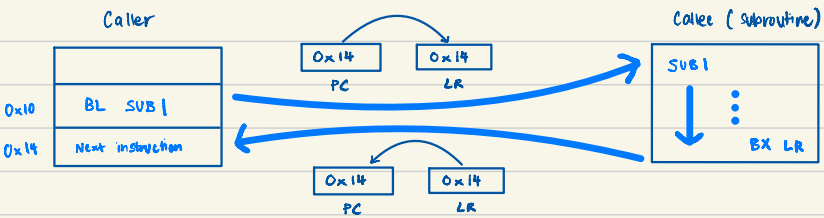


* Assume Sort Sub is in 0x0030

BRANCH AND EXCHANGE (BX)

⇒ BX LR returns from subroutine

⇒ LR contains return address ; instruction copies the value over to PC



NOTE: VISUAL DOES NOT support BX LR

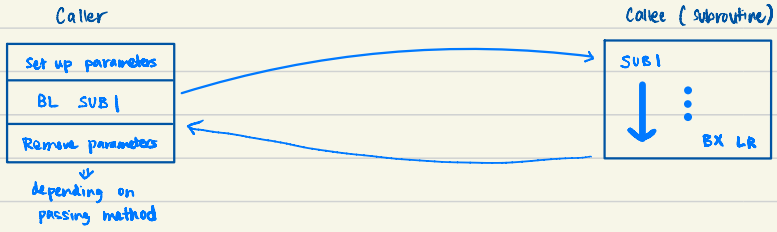
→ USE MOV PC, LR (only this used in exam)

⇒ However, overwrites PC value (dangerous, but no choice in emulator)

⇒ Alternative: sub → MOV LR, [PC+4] → B sub (safer, but takes up performance [2 lines vs 1 line])

PARAMETER PASSING

- Calling programs need to pass parameters to influence a subroutine's execution.
- Parameters must be set up properly before the subroutine is called & appropriately removed after returning.
- Methods to pass parameters, via:
  - → registers
  - → memory block : allocate memory to location of variable; allocation known to caller, sends address to subroutine to perform operations
  - → system stack



PARAMETER PASSING USING REGISTERS

- Parameters placed into register before calling subroutine
- Number of parameters passed limited to available registers
  - → Useful when number of parameters is small
  - → Pros: efficient as parameters are already in register within subroutine & can be used immediately
  - → Cons: lacks generality due to the limited number of registers

Total No. of Registers: 16
Registers that CANNOT be used to store parameters:
PC, LR, SP
→ Left with 13 (R0 - R12)

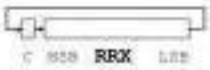
CALLING CONVENTION

- R0 - R3 (by value / reference)
  - → "free-to-use"
  - → Caller can pass argument values
  - → Subroutine can return values / modify values
- R4 - R11
  - → hold local variables ; Not for passing arguments
  - → Must be preserved in the subroutine
    - ⇒ Subroutine needs to store a copy first in stack before working on parameters (↳ backup)
- R12
  - → Scratchpad register ; does NOT need to be preserved
  - → Can be used as return register
  - → May be included under 'R0 - R3' (not really used for passing)

Program Counter (PC)
→ always points 8 bytes ahead of current instruction
⇒ point PC to base address of subroutine
⇒ Then it will go 8 bytes ahead after pointing at base address

Example: Bit Counting Subroutine

- Write a subroutine to:
  - Count the number of "1" bits in a word.
  - Return result in register **R0**.
  - **Design considerations:**
  - How do we transfer the word into the subroutine?
    - Put the word into a **register**, which can then be accessed within the subroutine (e.g. register **R1**).
  - How do we check if each individual bit is a "1" or a "0"?
    - Rotate **R1** right 32 times with the carry bit. After each rotate, test carry bit to check if (C=1). If yes, increment bit counter register **R0**.



Solution #1


```
Count1s  MOV     R0, #0          ;Clear R0
         MOV     R2, #32  ; Set counterR2 with 32
Loop     RRXS    R1,R1           ; Rotate right and extend R1
         ADC     R0,R0,#0        ; Add the carry to R0
         SUBS    R2,R2,#1        ; decrement counter by 1
         BNE     Loop            ; loop if not zero
         MOV     PC, LR          ; same as bx lr
```

Value passed in R1; return value in R0 — VisUAL does not support bx lr

Solution #2

```
                 ↗ can be used instead of MOV
Count1s  EOR     R0,R0,R0        ;Clear R0
         ADD     R2, R0, #32     ; Set counterR2 with 32
         ADD     R3, R0, #1      ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2        ; ?
         ADD     R0,R0,R4        ; Add the lsb of R0
         SUBS    R2,R2,#1        ; decrement counter by 1
         BNE     Loop            ; loop if not zero
         MOV     PC, LR          ; same as bx lr
```

Value passed in R1; return value in R0 — VisUAL does not support bx lr

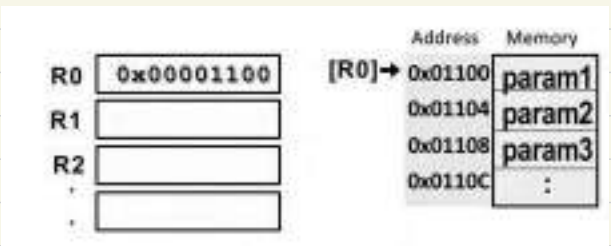
(R3 - mask of value 1)

AND operator using R3 (mask) and rotated value of R1 → store in R4;

R1, ROR R2: take value of R1 and rotate it by R2 values → creates a phantom register

PARAMETER PASSING USING MEMORY

- A region in memory is treated like a mail box and is used by both the calling program and subroutine.
  - → Parameters to be passed are gathered into a block at a predefined memory location
  - → Start address of memory block is passed to subroutine via an address register
  - → Using for passing large number of parameters



Example : Lower to Upper Case Subroutine

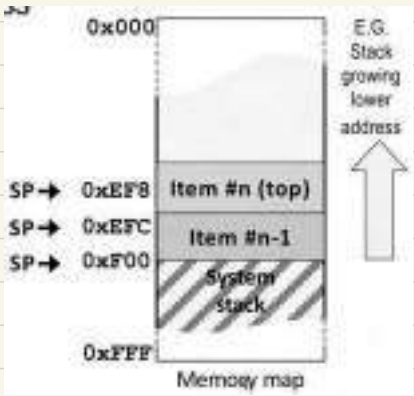
- How to convert an ASCII character from lower to upper case?
  - → Check char value is between `a' and `z'
  - → If so, subtract value by 32.
- Example assumes each character in 32 bits ← need to optimise memory usage
  - → But each character requires 8 bits only
  - → Use the {B} option in Access (LDRB, STRB) to access each Byte separately (gets extended by 0s)
  - → STMFD SP!, {r0, r1} ; save registers used within subroutines
  - → LDMFD SP!, {r0, r1} ; restore saved registers before returning
    - ⇒ produces transparent subroutine that does not affect the proper operation of calling program

PASSING PARAMETERS: REGISTERS VS MEMORY

- Registers : simplest & fastest; but no. of parameters that can be passed limited by the available registers
- Memory blocks : supports a large number of parameters or datatypes like arrays

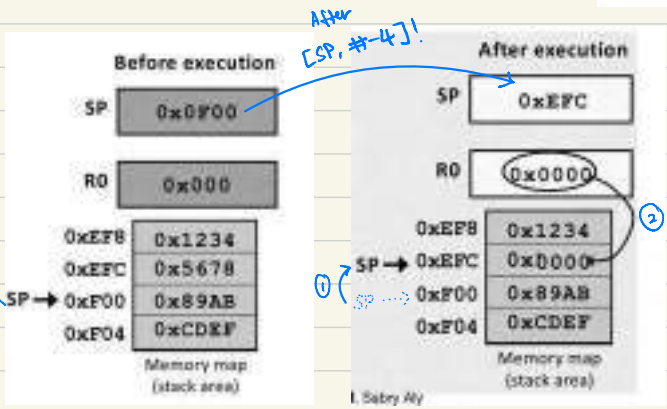
SYSTEM STACK

- first-in, last out (FILO) linear data structure that is maintained in the memory's data area.
- maintained by SP (R13)
- can grow towards lower or higher memory address (LOWER preffered)
- SP points to top item on system stack
  → points to valid data, not free space
- Stack operations: push, pop, access



PUSH DATA TO STACK

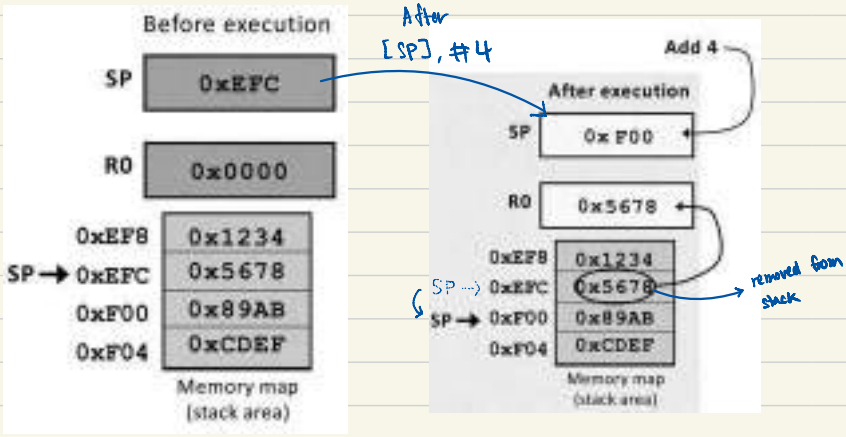
- Use STR to write to stack
  → increase SP before storing
  → STR R0, [SP, #-4]! (pre-indexing)



- Any value to be stored in stack should be placed into a register first
  → constants / numbers CANNOT be pushed into stack
  ⇒ get constant → push into register → push register into stack

POP DATA OFF THE STACK

- LDR used to pop from stack
  → need to update pointer after read
  → LDR R0, [SP], #4 (post-indexing)

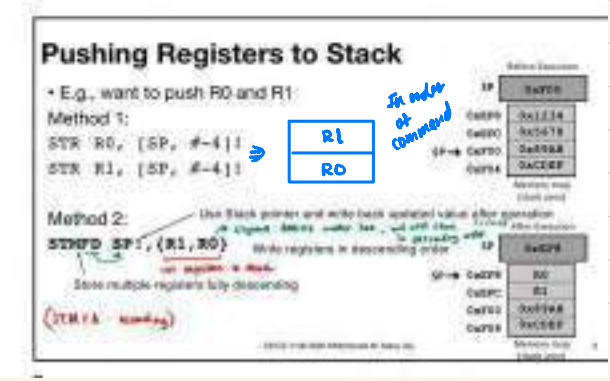


NOTE: Popping does NOT erased data from stack
→ Simply does not exist in data structure of stack
→ items considered "in the stack" it found below SP

- Data still resides in memory and can be read
  → as long as not overwritten by another value
  → To retrieve back: LDR R0, [SP, #-4] (SP not changed)

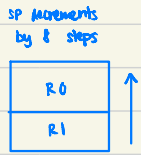
it want to change SP add "!"

- To erase value in memory
  → populate with 0s

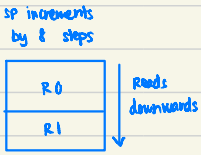


PUSHING AND POPPING TO STACK (METHOD 2)

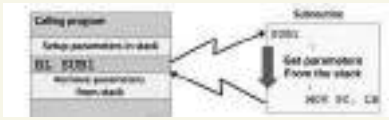
- Pushing : STMFD SP!, {list of registers}
  - → STMFD : Store multiple registers fully descending
  - → SP! : use SP & write back updated value after operation
  - → {list of registers}: sequence of registers do not matter ; registers still stored in descending order



- Popping : LDMFD SP!, {list of registers}
  - → assumes that registers stored in descending order, hence will read from ascending way



- Between STMFD & LDMFD, ensure code returns SP back to where STMFD was
  - → same set of registers to both instructions.
- If 2 STMFD instructions were used, use 2 LDMFD instructions to be 100% sure that the registers have the correct values loaded to it.

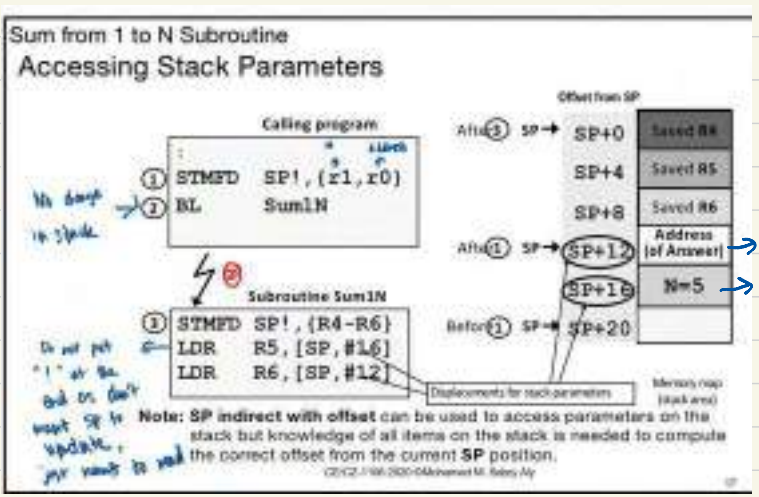


1 word = 4 bytes

PARAMETER PASSING USING STACK

- Parameters are pushed onto stack before calling subroutine and retrieved from stack within subroutine.
  - → Most general method of parameter passing – no registers needed, supports recursive programming
  - → Large number of parameters can be passed as long as no stack overflow.
- Parameters pushed to the stack must be removed by calling program immediately after returning from subroutine (After BL instruction)
  - → because might loop ⇒ repeated pushing of parameters to the stack will lead to a stack overflow
  - → removal of parameters do not happen in subroutine
    - ⇒ has to happen in caller program
    - ⇒ Possible code to do this:
      - ADD SP, SP, #8 } add 8 to pop two parameters
      - LDMFD SP!, {R1, R0} } from stack
- Use R4 – R11 in subroutine (stack) → return original value of SP before parameters were pushed to stack

Example: Sum from 1 to N Subroutine



TRANSPARENT SUBROUTINES

- does not affect any CPU resources used by the program calling it.
  - ⇒ To achieve this, all local registers used by the subroutine (R4-R11) must be saved on the stack on entry and restored from stack before returning.

```
SUB1 STMFD SP!,{R4-R7}     ; save R4 to R7 to stack
     :
     :                     ; registers R4 to R7 are
     :                     ; used in subroutine
     LDMFD SP!,{R4-R7}     ; restore R4 to R7 from stack
     MOV PC,LR             ; return to calling program
```

PASSING BY VALUE

- value of data (or variable) passed to subroutine (use of integer)
  - → E.g. MOV R1, #5 ; LDR R1, [R0] (no such thing as MOV R1, [R0])

PASSING BY REFERENCE

- address of variable passed to subroutine (use of pointer)
  - → E.g. MOV R0, #0x100
  - → used when parameter passed is to be modified by subroutine
  - → used for large quantity of data (e.g. array) have to be passed between subroutine and calling program.
  - → Subroutine can directly access memory variable within calling program

LOCAL VARIABLES

- Subroutines often use local variables whose scope and life span exist only during the execution of the subroutine.

→ Memory storage for these variables is created on entry into the subroutine and released on exit from subroutine

→ System Stack : ideal place to create memory space for temporary variables

→ Stack Frame : temporary memory space

STACK FRAME

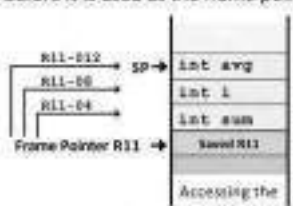
- Memory space within the stack frame can be accessed using a frame pointer (FP) or stack pointer (SP)
- Frame pointer in ARM can be any register (usually R11)

USING THE FRAME POINTER

Accessing Stack Frame Variables

Using the Frame Pointer

- Consider the use of a frame pointer register **R11**.
  - Original contents of **R11** is saved on the stack before it is used as the frame pointer.
  - Frame pointer (**R11**) now points to the saved **R11** and a stack frame is created by adding frame size **4N** to **SP**.
  - **R11** is the frame reference and an appropriate negative displacement from **R11** can be used to access any stack frame variable.
  - When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.



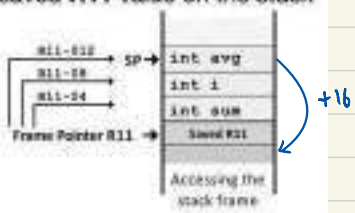
→ 1 function : 1 frame pointer

→ each word : 4 bytes

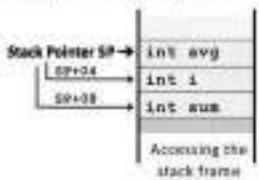
- When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.

For N=3

```
ADD SP, SP, #16
LDR R11,[R11]
```



- A more efficient approach is to use the stack pointer (**SP**) itself.
  - A stack frame is created by adding frame size **4N** to **SP**.
  - **SP** is used as a reference to access all local variables.
  - Appropriate **positive** displacements from **SP** is used to access any of the stack frame variables.
  - **Pro** - This method is more efficient because there is no need to setup a frame pointer. → less resources
  - **Con** – More restrictive as system stack cannot be used within subroutine without changing the reference **SP**.



- Using stack to pass parameters is the most favored approach
  → A combination of methods can be used to implement functions, e.g. parameters passed in via stack and a single result value passed out via register (e.g. R0)
  → Stack-based parameter passing supports recursion.

MULTIPLE SUBROUTINES

- Caller program can call multiple subroutines
- Support of sequential subroutine execution

* Do all applications have just 1-level functional call? NO
→ Subroutine can call another subroutine

SUPPORT FOR NESTED SUBROUTINE

LR: where return address is stored

- LR needs to be saved somewhere safe → System stack
- When to save it
→ Just before BL instruction (no advantages)
→ At the beginning of each subroutine (safest approach)

RECURSIVE SUBROUTINE

- recursive routine calls itself within its own body

Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)....2×1**)

- very efficient implementation of subroutines
→ Ensure : 1) stopping condition (stop function from calling itself)
2) return address stored