

ARM Cortex-M4 Programming Model

Stacks and Subroutines

Textbook: Chapter 8.1 - Subroutine call/return

Chapter 8.2,8.3 – Stack operations

Chapter 8.4, 8.5 – Passing arguments to/from subroutines

“ARM Cortex-M Users Manual”, Chapter 3

CPU instruction types

- **Data movement operations**
 - memory-to-register and register-to-memory
 - includes different memory “addressing” options
 - “memory” includes peripheral function registers
 - register-to-register
 - constant-to-register (or to memory in some CPUs)
- **Arithmetic operations**
 - add/subtract/multiply/divide
 - multi-precision operations (more than 32 bits)
- **Logical operations**
 - and/or/exclusive-or/complement (between operand bits)
 - shift/rotate
 - bit test/set/reset
- **Flow control operations**
 - branch to a location (conditionally or unconditionally)
 - branch to a subroutine/function
 - return from a subroutine/function

Subroutine

- A **subroutine**, also called a **function** or a **procedure**:
 - single-entry, single-exit
 - return to caller after it exits
- When a subroutine is called, the **Link Register** (LR) holds the memory address of the next instruction to be executed in the calling program when the subroutine exits.

Subroutine calls

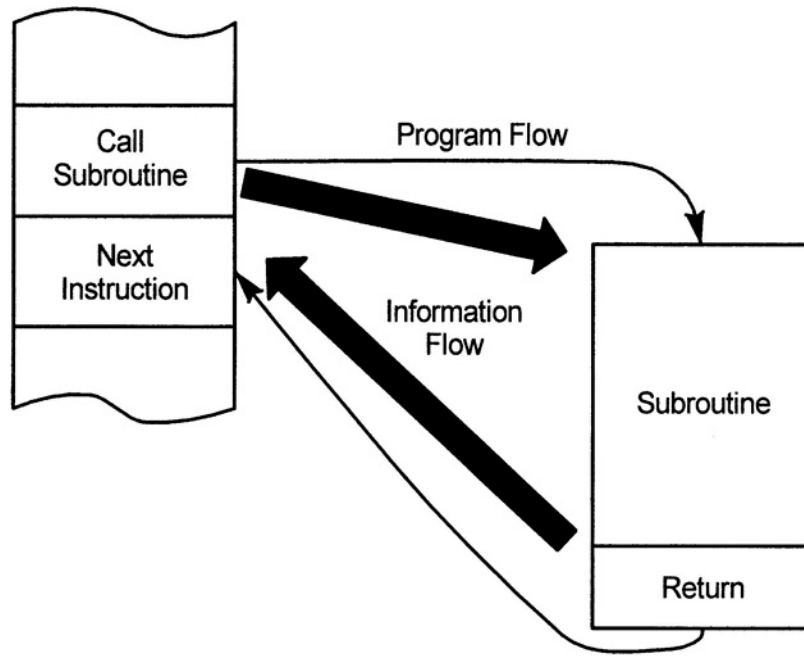


Figure 8-1 Information transfer between modules.

ARM subroutine linkage

- Branch and link instruction:

BL foo ; Copies current PC to r14.

- To return from subroutine:

BX r14 ; branch to address in r14

or:

MOV r15, r14 --**Not recommended for Cortex**

- May need subroutine to be “reentrant”
 - interrupt it, with interrupting routine calling the subroutine (2 instances of the subroutine)
 - support by creating a “stack” to save subroutine state

Call a Subroutine

| Caller Program | Subroutine/Callee |
|------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>MOV r4, #100 ... BL foo ... ADD r4, r4, #1 ; r4 = 101, not 11</pre> | <pre>foo PROC PUSH {r4} ... MOV r4, #10 ; foo changes r4 ... POP {r4} BX LR ENDP</pre> |

Saving/restoring multiple registers

- LDM/STM – load/store multiple registers
 - LDMIA – increment address after xfer
 - LDMIB – increment address before xfer
 - LDMDA – decrement address after xfer
 - LDMDB – decrement address before xfer
 - LDM/STM default to LDMIA/STMIA

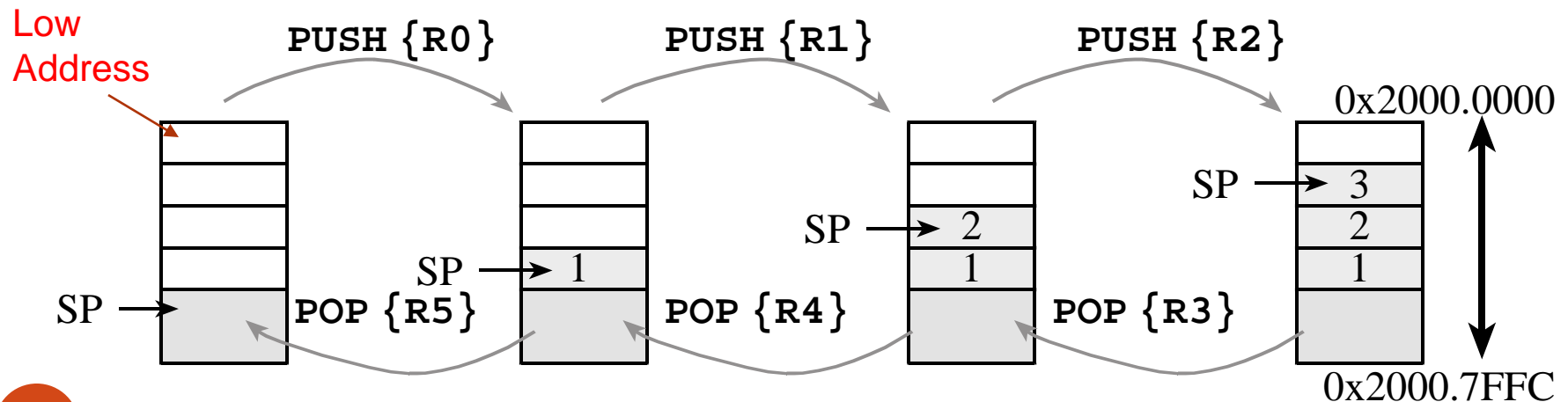
Examples:

`ldmia r13!, {r8-r12, r14}` ; r13 updated at end

`stmda r13, {r8-r12, r14}` ; r13 not updated at end

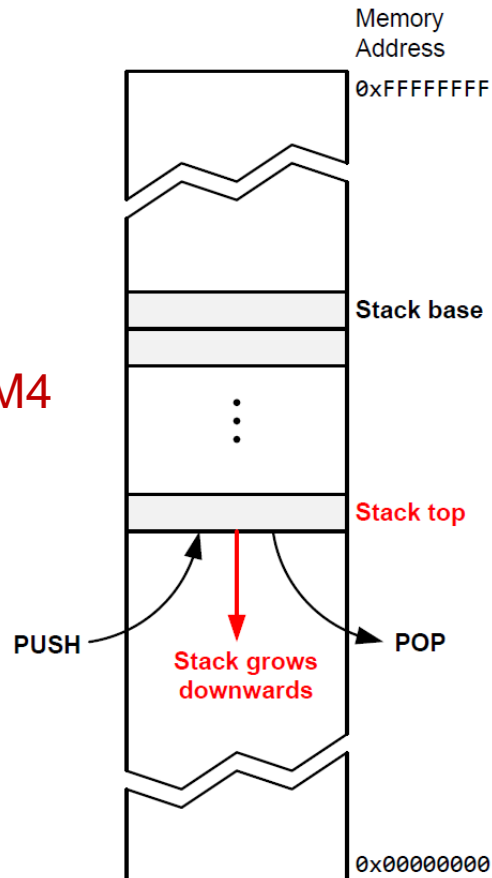
The Stack

- ❑ Stack is last-in-first-out (LIFO) storage
 - ❖ 32-bit data
- ❑ Stack pointer, SP or R13, *points to top element* of stack
 - ❖ SP *decremented* before data placed ("pushed") onto stack
 - ❖ SP *incremented* after data removed ("popped") from stack
- ❑ **PUSH** and **POP** instructions used to load and retrieve data
 - ❑ `PUSH {reglist} = STMDB sp!,{reglist}`
 - ❑ `POP {reglist} = LDMIA sp!,{reglist}`

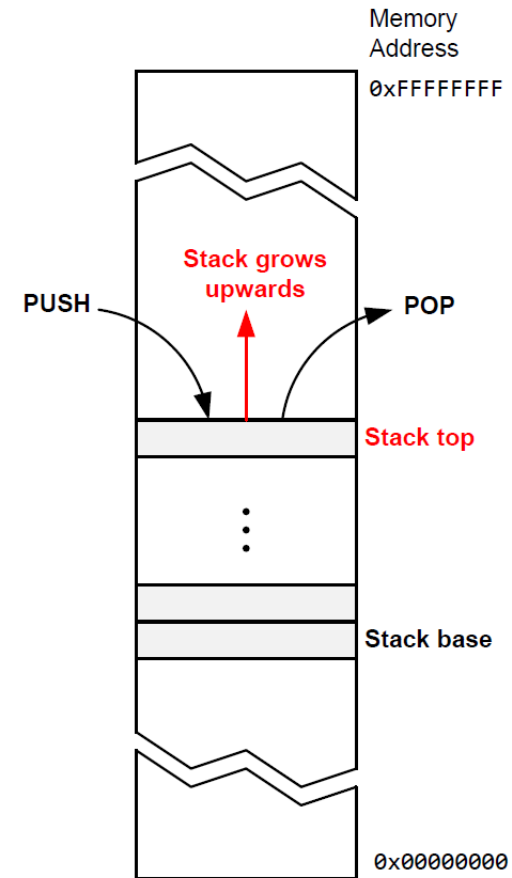


Stack Growth Convention: Ascending vs Descending

Used in
Cortex-M4



Descending stack: Stack grows towards low memory address

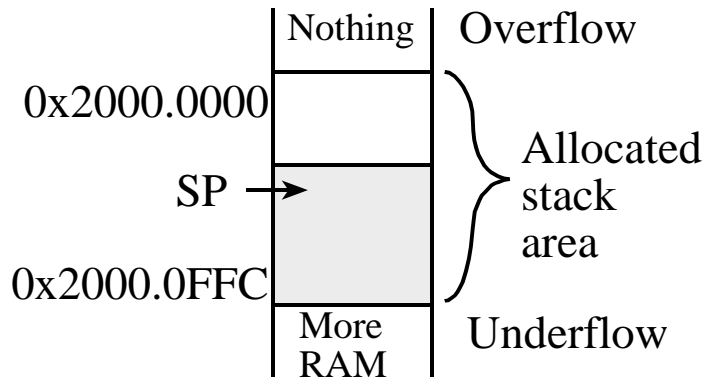


Ascending stack: Stack grows towards high memory address

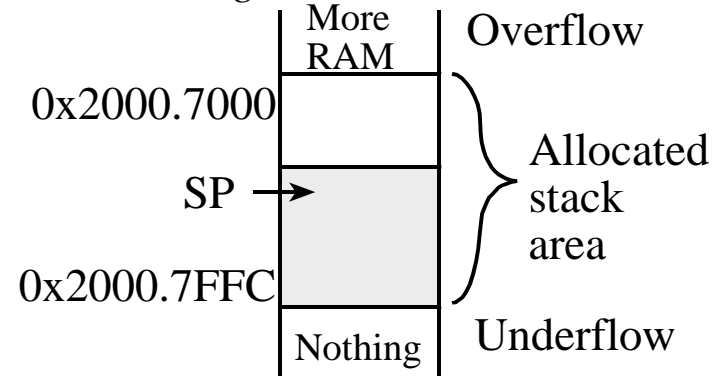
Stack Usage

❑ Stack memory allocation

Stack starting at the first RAM location



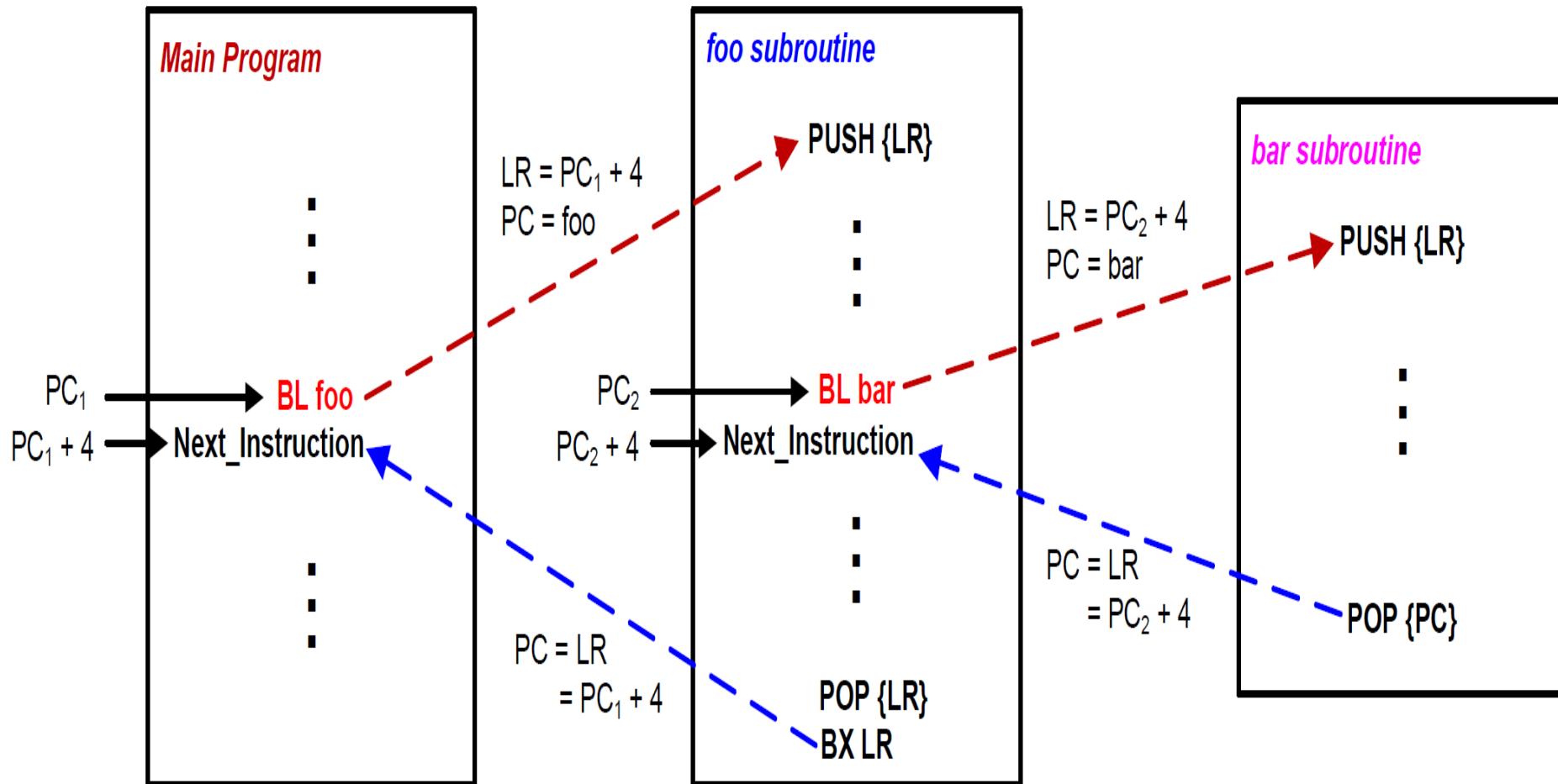
Stack ending at the last RAM location



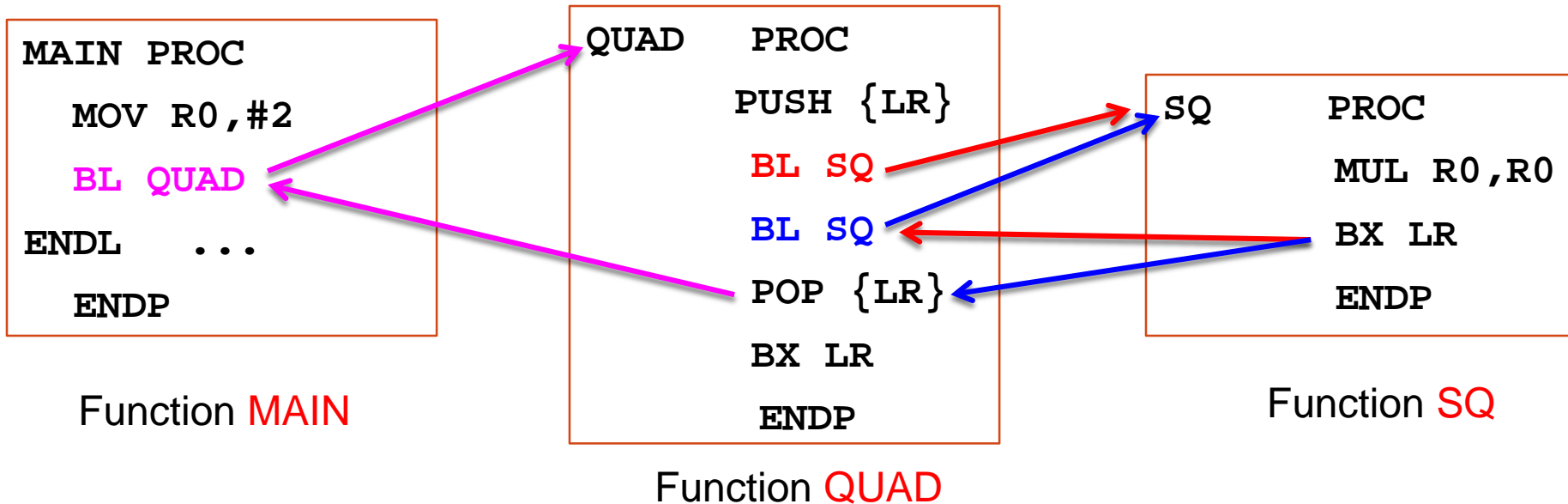
❑ Rules for stack use

- ❖ Stack should always be balanced, i.e. functions should have an equal number of pushes and pops
- ❖ Stack accesses (push or pop) should not be performed outside the allocated area
- ❖ Stack reads and writes should not be performed within the free area

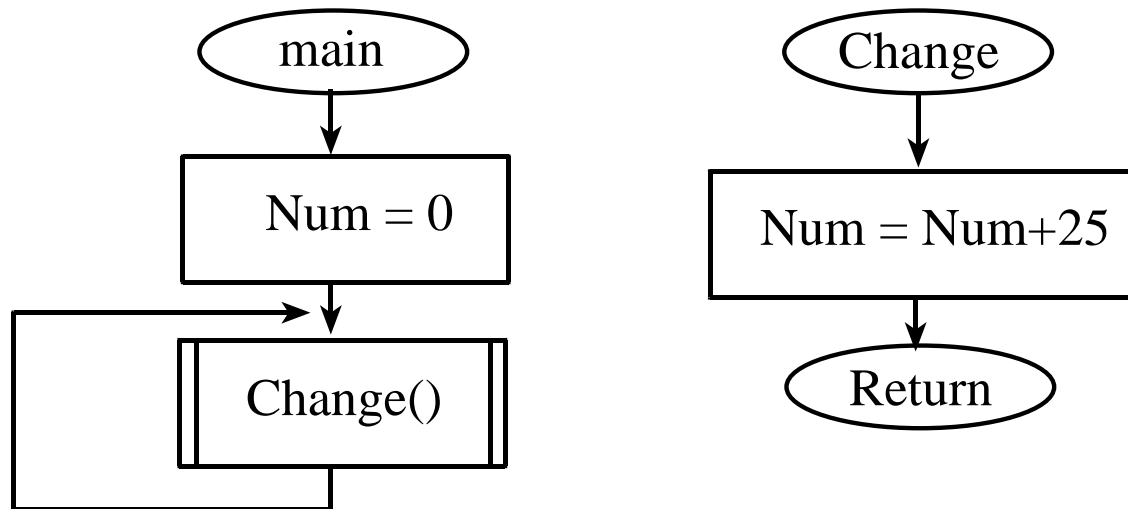
Stacks and Subroutines



Subroutine Calling Another Subroutine



Functions



| | | | |
|--------|-----|-----------|--------------------|
| Change | LDR | R1,=Num | ; 5) R1 = &Num |
| | LDR | R0,[R1] | ; 6) R0 = Num |
| | ADD | R0,R0,#25 | ; 7) R0 = Num+25 |
| | STR | R0,[R1] | ; 8) Num = Num+25 |
| | BX | LR | ; 9) return |
| main | LDR | R1,=Num | ; 1) R1 = &Num |
| | MOV | R0,#0 | ; 2) R0 = 0 |
| | STR | R0,[R1] | ; 3) Num = 0 |
| loop | BL | Change | ; 4) function call |
| | B | loop | ; 10) repeat |

```

unsigned long Num;
void Change(void){
    Num = Num+25;
}
void main(void){
    Num = 0;
    while(1){
        Change();
    }
}
  
```

Subroutines

```
;-----Rand100-----  
; Return R0=a random number between  
; 1 and 100. Call Random and then divide  
; the generated number by 100  
; return the remainder+1
```

Rand100

```
PUSH {LR}    ; SAVE Link  
BL    Random  
  
;R0 is a 32-bit random number  
LDR   R1,=100  
  
BL    Divide  
ADD   R0,R3,#1  
POP   {LR}   ;Restore Link back  
BX    LR
```

POP {PC}

```
;-----Divide-----  
; find the unsigned quotient and remainder  
; Inputs:  dividend in R0  
;          divisor in R1  
; Outputs: quotient in R2  
;          remainder in R3  
;dividend = divisor*quotient + remainder
```

Divide

```
UDIV R2,R0,R1    ;R2=R0/R1,R2 is quotient  
MUL  R3,R2,R1    ;R3=(R0/R1)*R1  
SUB  R3,R0,R3    ;R3=R0%R1,  
                        ;R3 is remainder of R0/R1  
  
BX   LR          ;return
```

ALIGN

END

One function calls another,
so LR must be saved

Reset, Subroutines and Stack

- A **Reset** occurs immediately after power is applied and when the reset signal is asserted (Reset button pressed)
- The Stack Pointer, SP (R13) is initialized at **Reset** to the 32-bit value at location 0 within the ROM
- The Program Counter, PC (R15) is initialized at **Reset** to the 32-bit value at location 4 within the ROM (Reset Vector)
- The Link Register (R14) is initialized at Reset to 0xFFFFFFFF
- Thumb bit is set at **Reset** (Cortex-M4)
- Processor automatically saves return address in LR when a subroutine call is invoked.
- User can push and pull multiple registers on or from the **Stack** at subroutine entry and before subroutine return.

Registers to pass parameters

| <u>High level program</u> | <u>Subroutine</u> |
|-------------------------------------|------------------------------------------|
| 1) Sets Registers to contain inputs | |
| 2) Calls subroutine | |
| | 3) Sees the inputs in registers |
| | 4) Performs the action of the subroutine |
| | 5) Places the outputs in registers |
| 6) Registers contain outputs | |

Stack to pass parameters

| <u>High level program</u> | <u>Subroutine</u> |
|--------------------------------|------------------------------------------|
| 1) Pushes inputs on the Stack | |
| 2) Calls subroutine | |
| | 3) Sees the inputs on stack (pops) |
| | 4) Performs the action of the subroutine |
| | 5) Pushes outputs on the stack |
| 6) Stack contain outputs (pop) | |
| 7) Balance stack | |

ARM Architecture Procedure Call Standard (AAPCS)

- Application Binary Interface (ABI) standard for ARM
 - Allows assembly subroutine to be callable from C or callable from someone else's software
- Parameters passed using registers and stack
 - Use registers R0, R1, R2, and R3 to pass the first four input parameters (in order) into any function, C or assembly.
 - Pass additional parameters via the stack
 - Place the return parameter in Register R0.
- Functions can freely modify registers R0–R3 and R12.
- If a function uses R4–R11, push current register values onto the stack, use the registers, and then pop the old values off the stack before returning.

ARM Procedure Call Standard

| Register | Usage | Subroutine Preserved | Notes |
|----------------|-------------------------------|----------------------|--------------------------------------------------------------------------------------|
| r0 (a1) | Argument 1 and return value | No | If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it. |
| r1 (a2) | Argument 2 | No | |
| r2 (a3) | Argument 3 | No | If the return has 128 bits, r0-r3 hold it. |
| r3 (a4) | Argument 4 | No | If more than 4 arguments, use the stack |
| r4 (v1) | General-purpose V1 | Yes | Variable register 1 holds a local variable. |
| r5 (v2) | General-purpose V2 | Yes | Variable register 2 holds a local variable. |
| r6 (v3) | General-purpose V3 | Yes | Variable register 3 holds a local variable. |
| r7 (v4) | General-purpose V4 | Yes | Variable register 4 holds a local variable. |
| r8 (v5) | General-purpose V5 | YES | Variable register 5 holds a local variable. |
| r9 (v6) | Platform specific/V6 | No | Usage is platform-dependent. |
| r10 (v7) | General-purpose V7 | Yes | Variable register 7 holds a local variable. |
| r11 (v8) | General-purpose V8 | Yes | Variable register 8 holds a local variable. |
| r12 (IP) | Intra-procedure-call register | No | It holds intermediate values between a procedure and the sub-procedure it calls. |
| r13 (SP) | Stack pointer | Yes | SP has to be the same after a subroutine has completed. |
| r14 (LR) | Link register | No | LR does not have to contain the same value after a subroutine has completed. |
| r15 (PC) | Program counter | N/A | Do not directly change PC |

Example: $R2 = R0 * R0 + R1 * R1$

```
MOV R0, #3
MOV R1, #4
BL SSQ
MOV R2, R0
B ENDL

...
SSQ MUL R2, R0, R0
    MUL R3, R1, R1
    ADD R2, R2, R3
    MOV R0, R2
    BX LR
...
```

R1: second argument

R0: first argument

```
int SSQ(int x, int y){
    int z;
    z = x*x + y * y;
    return z;
}
```

R0: Return Value

Parameter-Passing: Registers

Caller

--call a subroutine that

uses registers for parameter passing

MOV R0,#7

MOV R1,#3

BL Exp

;; R2 becomes $7^3 = 343$ (0x157)

Question:

Is this AAPCS-compliant?

Callee

-----Exp-----

; Input: R0 and R1 have inputs XX and YY (non-negative)

; Output: R2 has the result XX raised to YY

; Destroys input R1

Exp

ADDS r0,#0 *;check if XX is zero*

BEQ Zero *;skip algorithm if XX=0*

ADDS r1,#0 *;check if YY is zero*

BEQ One *;skip algorithm if YY=0*

MOV r2,#1 *;Initial product is 1*

More MUL r2,r0 *;multiply product with XX*

ADDS r1,#-1 *;Decrement YY*

BNE More

B Retn *;Done, so return*

Zero MOV r2,#0 *;XX is 0 so result is 0*

B Retn

One MOV r2,#1 *;YY is 0 so result is 1*

Retn BX LR

Parameter-Passing: Stack

Caller

;----- call a subroutine that

; uses stack for parameter passing

MOV R0,#12

MOV R1,#5

MOV R2,#22

MOV R3,#7

MOV R4,#18

PUSH {R0-R4}

; Stack has 12,5,22,7 and 18 (with 12 on top)

BL Max5

; Call Max5 to find the maximum of the five numbers

POP {R5}

;; R5 has the max element (22)

Callee

;-----Max5-----

; Input: 5 signed numbers pushed on the stack

; Output: put only the maximum number on the stack

; Comments: The input numbers are removed from stack

numM RN 1 ; current number

max RN 2 ; maximum so far

count RN 0 ; how many elements

Max5

POP {max} ; get top element (top of stack) into max

MOV count,#4 ; 4 more to go

Again **POP {numM}** ; get next element

CMP numM,max

BLT Next

MOV max, numM ; new numM is the max

Next ADDS count,#-1 ; one more checked

BNE Again

PUSH {max} ; found max so push it on stack

BX LR

Parameter-Passing: Stack & Regs

Caller

*;-----call a subroutine that uses both
;stack and registers for parameter passing*

MOV R0,#6 ; R0 elem count

MOV R1,#-14

MOV R2,#5

MOV R3,#32

MOV R4,#-7

MOV R5,#0

MOV R6,#-5

PUSH {R4-R6} ; rest on stack

; R0 has element count

; R1-R3 have first 3 elements;

; remaining parameters on Stack

BL MinMax

;; R0 has -14 and R1 has 32 upon return

Callee

;-----MinMax-----

; Input: N numbers reg+stack; N passed in R0

; Output: Return in R0 the min and R1 the max

; Comments: The input numbers are removed from stack

MinMax

PUSH {R1-R3} ; put all elements on stack

CMP r0,#0 ; if N is zero nothing to do

BEQ DoneMM

POP {r2} ; pop top and set it

MOV r1,r2 ; as the current min and max

loop ADDS r0,#-1 ; decrement and check

BEQ DoneMM

POP {r3}

CMP r3,r1

BLT Chkmin

MOV r1,r3 ; new num is the max

Chkmin CMP f3,r2

BGT NextMM

MOV r2,r3 ; new num is the min

NextMM B loop

DoneMM MOV R0,min ; R0 has min

BX LR

Abstraction - Device Driver

Abstraction allows us to modularize our code and give us the option to expose what we want users to see and hide what we don't want them to see.

A **Device Driver** is a good example where abstraction is used to expose *public* routines that we want users of the driver to call and use *private* routines to hide driver internals from the user (more on *private* routines later)

LED Driver (PE0)

LED_Init
LED_Off
LED_On
LED_Toggle

*A user simply has to know
what a routine expects and
what it returns in order to
call it (**calling convention**).
Internals do not matter to caller*

Port E LED Abstraction

```
RCC      EQU 0x40023800      ;RCC base address (Reset and Clock Control)
AHB1ENR  EQU 0x30            ;offset of RCC->AHB1ENR (clock enable register)
GPIOE    EQU 0x40021000      ;GPIOE base address
MODER    EQU 0x00            ;offset of GPIOE->MODER (mode register)
ODR       EQU 0x14           ;offset of GPIOE->ODR (output data register)

; Initialize port pin PE0, which drives the LED
; Enable GPIOE clock and configure PE0 as an output pin
LED_Init
    ; enable clock to GPIOE
    LDR R1, =RCC              ; R1 -> RCC (Reset & Clock Control Regs)
    LDR R0, [R1,#AHB1ENR]     ; previous value of clock enable reg
    ORR R0, #0x00000010       ; activate clock for Port E (GPIOE)
    STR R0, [R1,#AHB1ENR]     ; update RCC clock enable register
    ; configure PE0 as an output pin
    LDR R1, =GPIOE           ; R1 -> GPIOE registers
    LDR R0, [R1, #MODER]      ; previous value of GPIOE Mode Reg
    BIC R0, #0x03             ; clear PE0 mode bits
    ORR R0, #0x01             ; set PE0 mode as output (mode 01)
    STR R0, [R1, #MODER]      ; update GPIOE mode register
    BX  LR
```

Port E LED Abstraction

`GPIOE_ODR EQU 0x40021014`

`;GPIOE output data reg. address`

LED_Off ;turn off LED connected to PE0

```
LDR  R1, =GPIOE_ODR      ; R1 is address of PE output reg
LDRH R0, [R1]             ; read current PE output bits
BIC  R0, #0x0001          ; affect only PE0 (PE0 = 0)
STRH R0, [R1]             ; write back to PE output reg
BX   LR
```

LED_On ;turn on LED connected to PE0

```
LDR  R1, =GPIOE_ODR      ; R1 is address of PE output reg
LDRH R0, [R1]             ; read current PE output bits
ORR  R0, #0x0001          ; affect only PE0 (PE0 = 1)
STRH R0, [R1]             ; write back to PE output reg
BX   LR
```

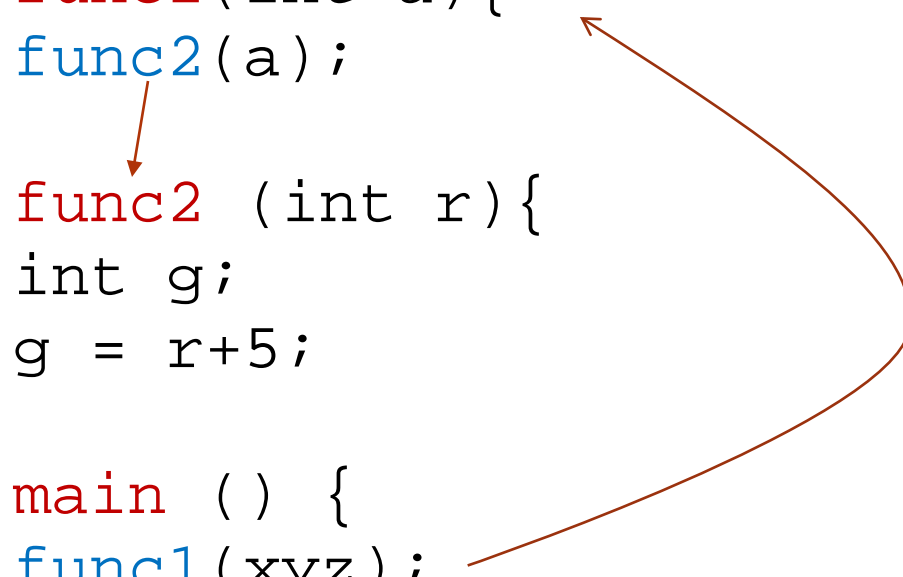
LED_Toggle ;toggle LED connected to PE0

```
LDR  R1, =GPIOE_ODR      ; R1 is address of PE output reg
LDRH R0, [R1]             ; read current PE output bits
EOR  R0, #0x0001          ; affect only PE0 (toggle PE0)
STRH R0, [R1]             ; write back to PE output reg
BX   LR
```

Nested subroutine calls

- Nested function calls in C:

```
void func1(int a) {  
    func2(a);  
}  
void func2 (int r) {  
    int g;  
    g = r+5;  
}  
void main () {  
    func1(xyz);  
}
```



Nested subroutine calls (1)

- Nesting/recursion requires a “coding convention” to save/pass parameters:

```
AREA Code1, CODE  
  
Main    LDR r13, =StackEnd    ;r13 points to last element on stack  
        MOV r1, #5           ;pass value 5 to func1  
        STR r1, [r13, #-4]!   ; push argument onto stack  
        BL  func1             ; call func1()  
  
here    B      here
```

Nested subroutine calls (2)

; Function func1(a)

| | | |
|-------|-----------------------|-----------------------------------------|
| Func1 | LDR r0,[r13] | ; load arg into r0 from stack |
| | ; call func2(a) | |
| | PUSH {r14} | ; store func1's return address |
| | PUSH (r0) | ; store arg to f2 on stack |
| | BL func2 | ; branch and link to f2 |
| | ; return from func1() | |
| | ADD r13,#4 | ; "pop" func2's arg off stack |
| | POP {r15} | ; restore saved "link" to PC and return |

Nested subroutine calls (3)

; Function func2(r)

```
Func2  POP    {r0}           ;retrieve arg from stack
        ADD    r1, r0, #5     ;g = r + 5
        BX     r14           ;preferred return instruction
```

; Stack area

```
        AREA   Data1,DATA
Stack   SPACE  20             ;allocate stack space
StackEnd
        END
```

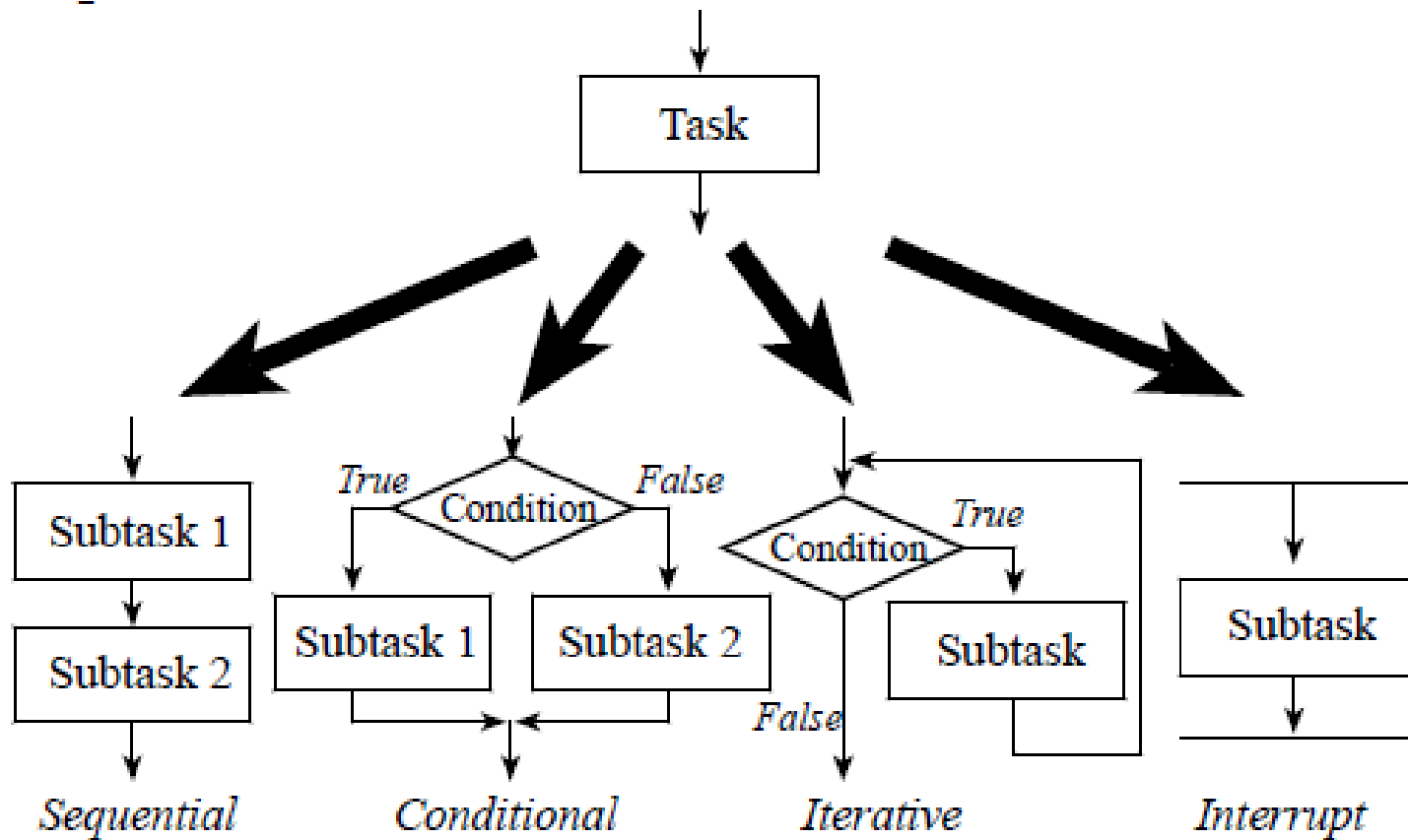
System Design

- Partition the problem into manageable parts
 - Successive Refinement
 - Stepwise Refinement
 - Systematic Decomposition
-
- Start with a task and decompose it into a set of simpler subtasks
 - Subtasks are decomposed into even simpler sub-subtasks
 - Each subtask is simpler than the task itself
 - Ultimately, subtask is so simple, it can be converted to software
 - Test the subtask before combining with other subtasks
 - Make design decisions
 - *document* decisions and subtask requirements

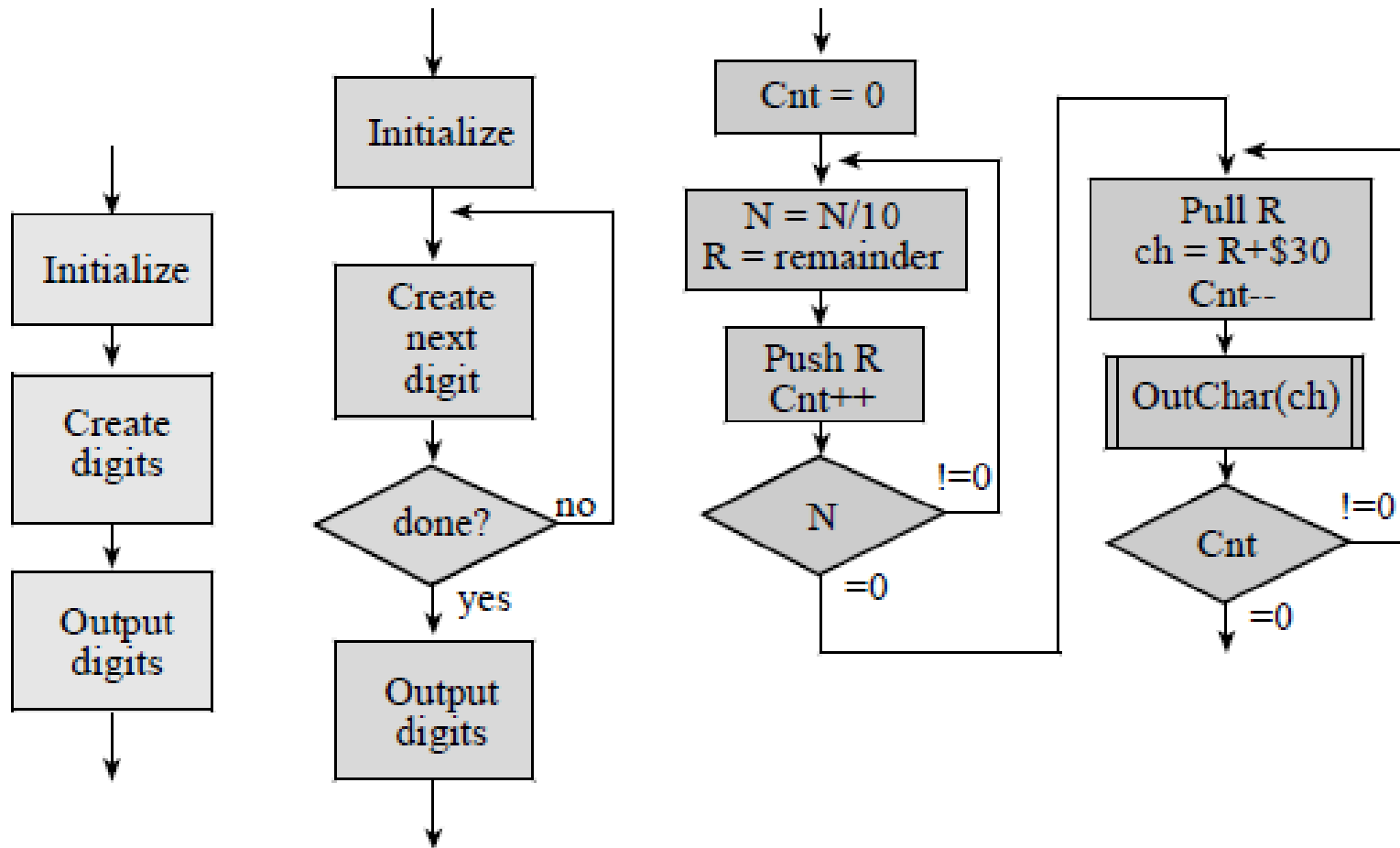
System Design

- Four structured program building blocks:
 - “do A then do B” → sequential
 - “do A and B in either order” → sequential (parallel)
 - “if A, then do B” → conditional
 - “for each A, do B” → iterative
 - “do A until B” → iterative
 - “repeat A over & over forever” → iterative (condition always true)
 - “on external event do B” → interrupt
 - “every t msec do B” → interrupt

Successive Refinement



Successive Refinement



Successive refinement example for iterative approach