CHAPTER 7

Programming in Assembly
Part 3: Control Structures

Compare/Test Instructions

Compare Instructions	***Operation***	***{S}***	***<op>***	***Notes***
CMP R_n,<op>	R_n – <op>	n/a	imm. const. -or- reg{,<shift>}	Always updates: NZCV
CMN R_n,<op>	R_n + <op>	n/a		Always updates: NZCV
TST R_n,<op>	R_n & <op>	n/a		Always updates: NZC
TEQ R_n,<op>	R_n ^ <op>	n/a		Always updates: NZC

Branch Instructions

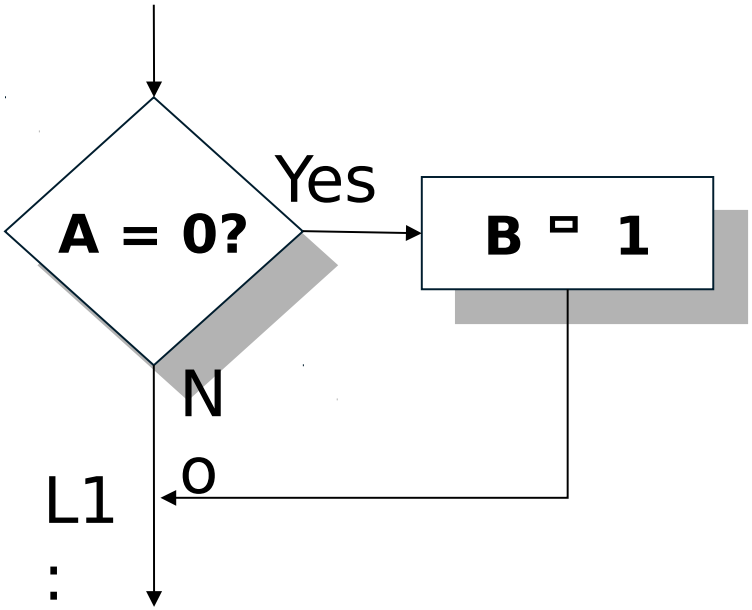
Branch Instructions	***Operation***	***{S}***	***Notes***
B{c} label	PC ≔ PC + imm	n/a	"c" is an *optional* condition code (see next slide)
BL label	PC ≔ PC + imm; LR ≔ rtn adr	n/a	Subroutine call
BX reg	PC ≔ reg	n/a	"BX LR" often used as function return
CBZ R_n,label	If R_n=0, PC ≔ PC + imm	n/a	Cannot append condition code to CBZ
CBNZ R_n,label	If R_n≠0, PC ≔ PC + imm	n/a	Cannot append condition code to CBNZ
IT$c_1c_2c_3$ cond	Each c_i is one of T, E, or *empty*	n/a	Controls 1-4 instructions in "IT block"

Any one of these may be appended to any instruction mnemonic when used inside an If-Then-Else (IT) block. (E.g., "IT NE followed by ADDNE" would add only if Z ≠ 0.)
Exceptions: CBZ, CBNZ, CMP, CMN, NEG, TST, or TEQ.

Condition Code	***Meaning***	***Requirements***
EQ	Equal	Z = 1
NE	Not equal	Z = 0
CS or HS	Carry set, or unsigned ≥ ("Higher or Same")	C = 1
CC or LO	Carry clear, or unsigned < ("Lower")	C = 0
MI	Minus/negative	N = 1
PL	Plus/positive or zero (non-negative)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned > ("Higher")	C = 1 && Z = 0
LS	Unsigned ≤ ("Lower or Same")	C = 0 || Z = 1
GE	Signed ≥ ("Greater than or Equal")	N = V
LT	Signed < ("Less Than")	N ≠ V
GT	Signed > ("Greater Than")	Z = 0 && N = V
LE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
AL	Always (unconditional)	only used with IT instruction

if-then statement

if (a == 0) b = 1 ;




```
        LDR     R0,A
        CMP     R0,#0
        BNE     L1
        LDR     R0,=1
        STR     R0,B
L1:...
```

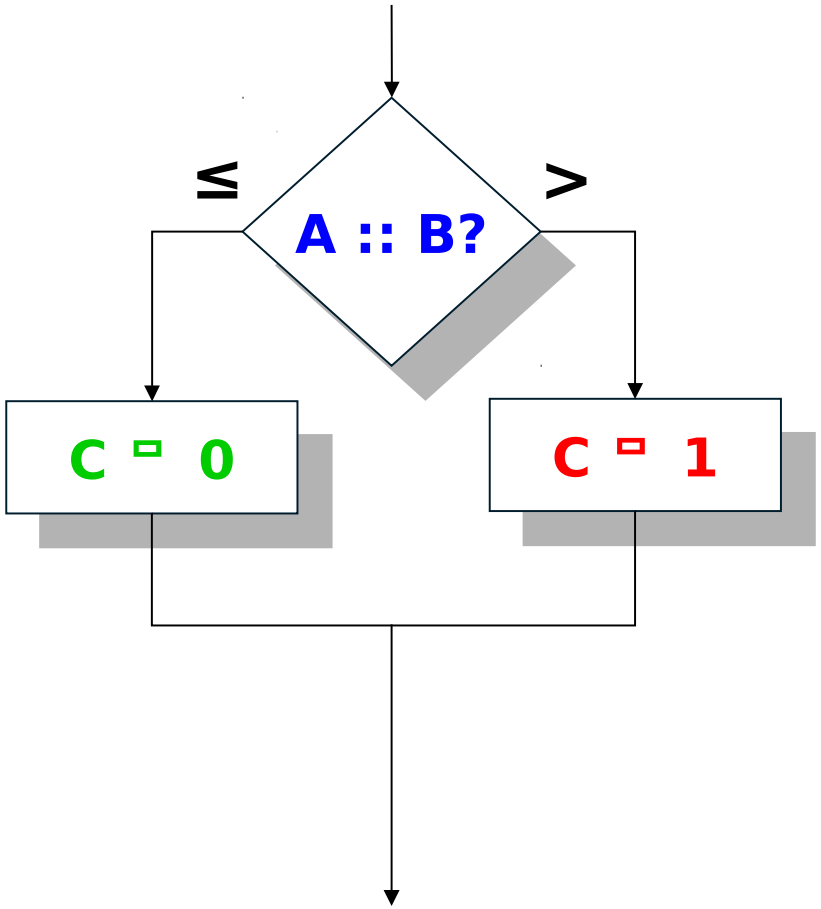
- or –

```
        LDR     R0,A
        CMP     R0,#0
        ITT     EQ
        LDREQ   R0,=1
        STREQ   R0,B
```

Signed vs. Unsigned

Compare	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
>=	BGE	BHS (same as BCS)
<	BLT	BLO (same as BCC)
<=	BLE	BLS

if-then-else statement




```
    LDR     R0,A
    LDR     R1,B
    CMP     R0,R1
    BLE     L1
    LDR     R0,=1
    B       L2
L1: LDR     R0,=0
L2:     STR     R0,C
    ...
```

- or –

```
    LDR     R0,A
    LDR     R1,B
    CMP     R0,R1
    ITE     GT
    LDRGT   R0,=1
    LDRLE   R0,=0
    STR     R0,C
```

Compound Conditionals

```
    if (lower_limit <= x && x <= upper_limit) y = x ;

    if (!(lower_limit <= x &
    y = x ;
L1:

    if (x < lower_limit || x
    y = x ;
L1:

    if (x < lower_limit) goto L1
    if (x > upper_limit) goto L1
    y = x ;
L1:
```

```
    LDR   R0,x
    LDR   R1,lower_limit
    CMP   R0,R1
    BLT   L1
    LDR   R1,upper_limit
    CMP   R0,R1
    BGT   L1
    STR   R0,y
L1:...
```

Compound Conditionals

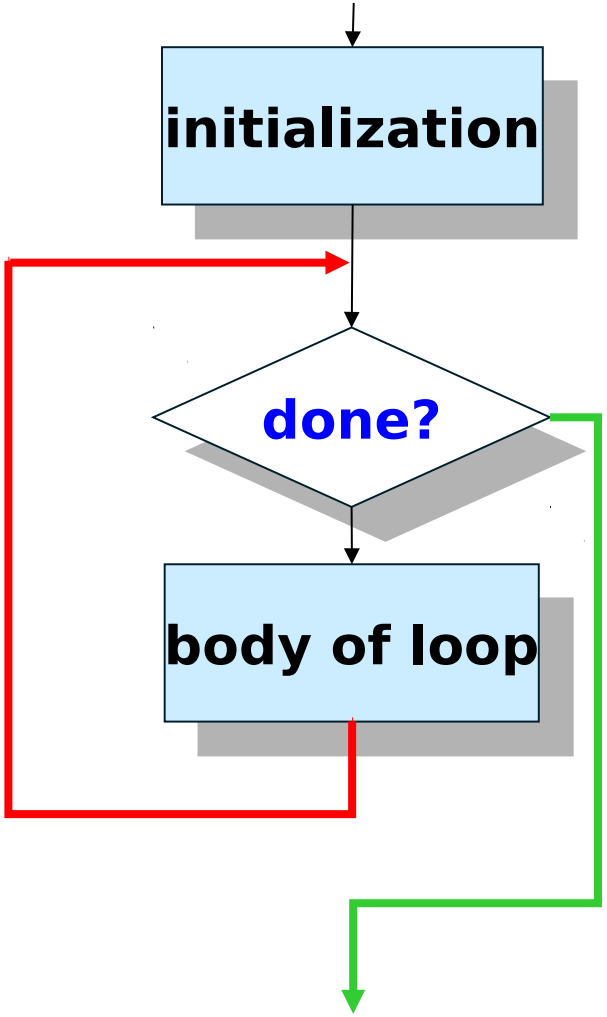
```
if (x < lower_limit || upper_limit < x) y = x ;
```

```
     if (x < lower_limit) goto
L1
     if (x > upper_limit) goto
L1
     goto L2 ;
L1:  y = x ;
L2:
```

```
     if (x < lower_limit) goto L1
     if (!(x > upper_limit)) goto L2
L1:  y = x ;
L2:
```

```
     LDR   R0,x
     LDR   R1,lower_limit
     CMP   R0,R1
     BLT   L1
     LDR   R1,upper_limit
     CMP   R0,R1
     BLE   L2
L1:  STR   R0,y
L2:  ...
```

Loops: Basic Structure




```
        ...
top:    CMP     ...
        B{cond} done
        ...
        ...
        B       top
done:
```

Loops: Predetermined #Iterations

```
for (n = 0; n < 100; n++)
    {

        n = 0,1,…,99

    }
```

```
        LDR   R0,=0
        STR   R0,n
top:    LDR   R0,n
        CMP   R0,#100
        BGE   done
        …
        LDR   R0,n
        ADD   R0,R0,#1
        STR   R0,n
        B   top
done:
```

Loops: Variable #Iterations

Ex: GCD(a,b)

```
while (a != b)
    {
    if (a > b) a = a – b ;
    else b = b – a ;
    }
```

```
        LDR     R0,a
        LDR     R1,b
top:    CMP     R0,R1
        BEQ     done
        ITE     GT
        SUBGT   R0,R0,R1
        SUBLE   R1,R1,R0
        B       top
done:
        ; R0 = R1 = GCD(a,b)
```

Miscellaneous Instructions

Bits / Bytes / Words	***Operation***	***{S}***	***Notes***
CLZ R_d,R_n	R_d ≔ CountZeroes(R_n)	n/a	# leading zeroes (0-32)
RBIT R_d,R_n	R_d ≔ RevBits(R_n)	n/a	Reverses bit order
REV R_d,R_n	R_d ≔ RevByteOrder(R_n)	n/a	Reverses byte order
REV16 R_d,R_n	R_d ≔ RevHalfWords(R_n)	n/a	Reverses half words
REVSH R_d,R_n	R_d ≔ RevLoHalf(R_n)	n/a	Reverses 2 LSbytes, sign extends
SXTB R_d,R_n	R_d ≔ SignedByte(R_n)	n/a	Sign extends, may pre-rotate R_n
SXTH R_d,R_n	R_d ≔ SignedHalf(R_n)	n/a	
UXTB R_d,R_n	R_d ≔ UnsignedByte(R_n)	n/a	Zero extends, may pre-rotate R_n
UXTH R_d,R_n	R_d ≔ UnsignedHalf(R_n)	n/a	

TIP: 6 THINGS TO CHECK WHEN WRITING ARM CODE

1. Operand Size: Byte, Half-word, Word, or Double-word?
2. Loading Byte or Half-word: Signed or Unsigned?
3. Need to update flags (NZCV)? ␣ Append “S”
4. Memory access: Only LDR’s, STR’s (and their variants)
5. No variable names or arithmetic operators inside square brackets
6. Expressions as operands (without brackets): Address calculation evaluated during assembly

Function Call and Return

Function Call: “BL function”

- Loads program counter (pc) with entry point address of function.
- Saves return address in the link register.

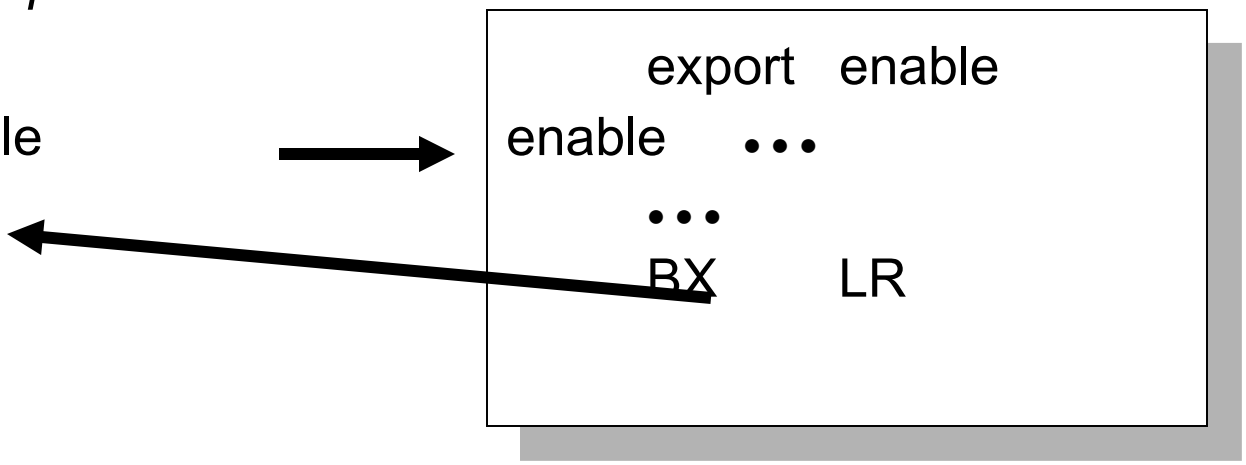
Function Return: “BX LR”

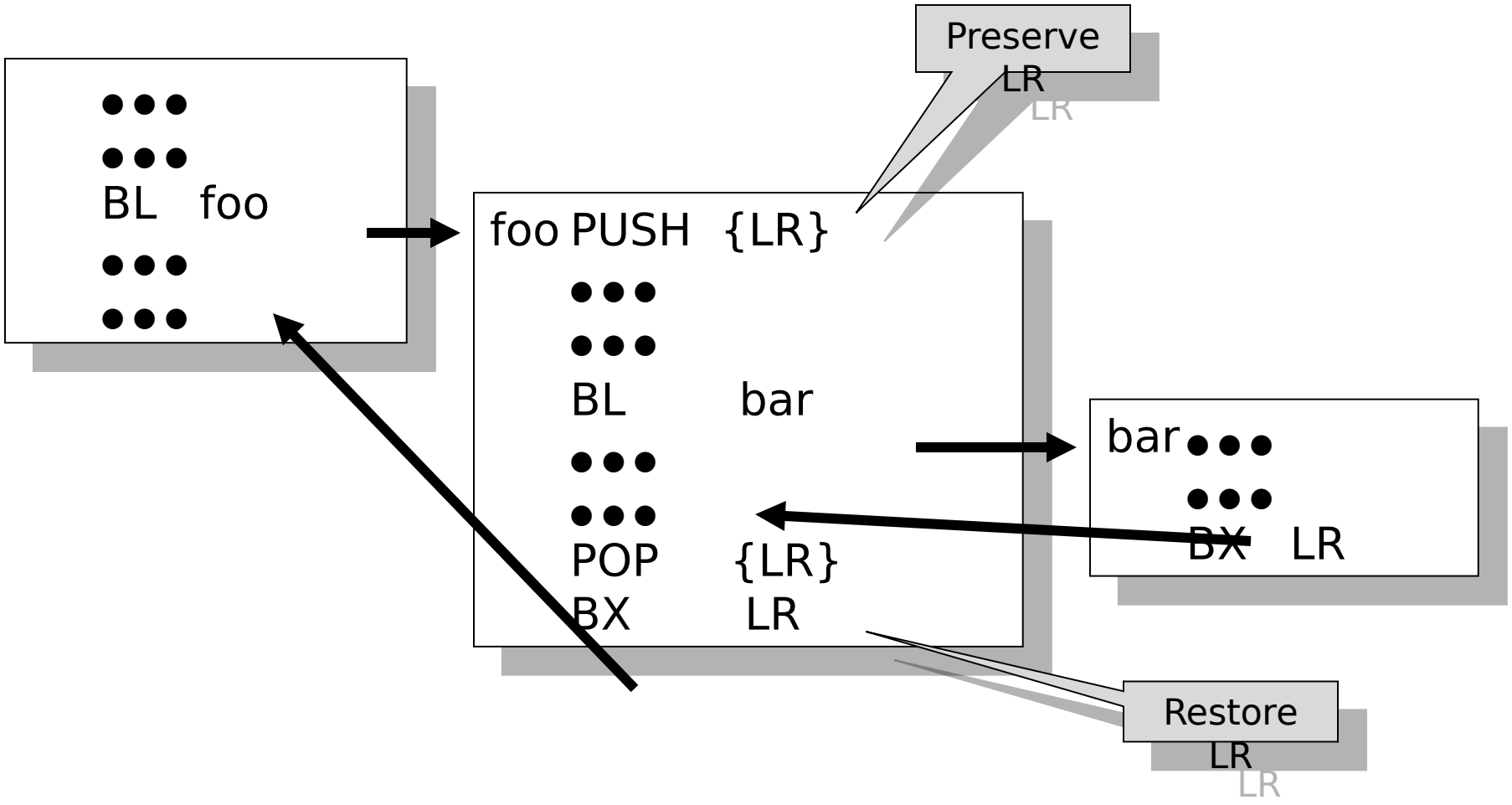
- copies link register back into program counter.

```
void enable(void) ;
• • •
enable() ;
• • •
```

Compiler

```
• • •
BL  enable
• • •
```




Preserve
LR
BL foo
foo PUSH {LR}
BL bar
POP {LR}
BX LR
bar
BX LR
Restore
LR

ARM Procedure Call Standard

Register Number	APCS Name	APCS Role
0	a1	argument 1 / integer result / scratch register
1	a2	argument 2 / scratch register
2	a3	argument 3 / scratch register
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4	register variable
8	v5	register variable
9	sb/v6	static base / register variable
10	sl/v7	stack limit / stack chunk handle / reg. variable
11	fp	frame pointer
12	ip	scratch register / new-sb in inter-link-unit calls
13	sp	lower end of current stack frame
14	lr	link address / scratch register
15	pc	program counter
f0	0	FP argument 1 / FP result / FP scratch register
f1	1	FP argument 2 / FP scratch register
f2	2	FP argument 3 / FP scratch register
f3	3	FP argument 4 / FP scratch register

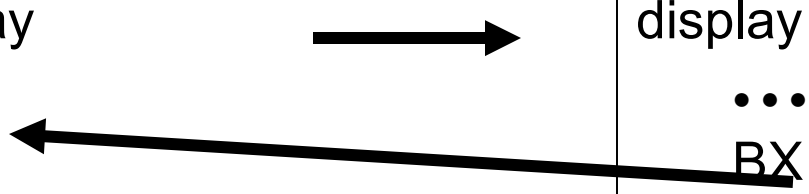
```
void display(uint8_t [ ], int32_t) ;
• • •
display(buffer, 5) ;
• • •
```

⇩ *Compiler*

```
• • •
ADR     R0,buffer
LDR     R1,=5
BL  display
• • •
```

Registers R0-R3 used for first 4 parameters; any more have to be pushed on stack

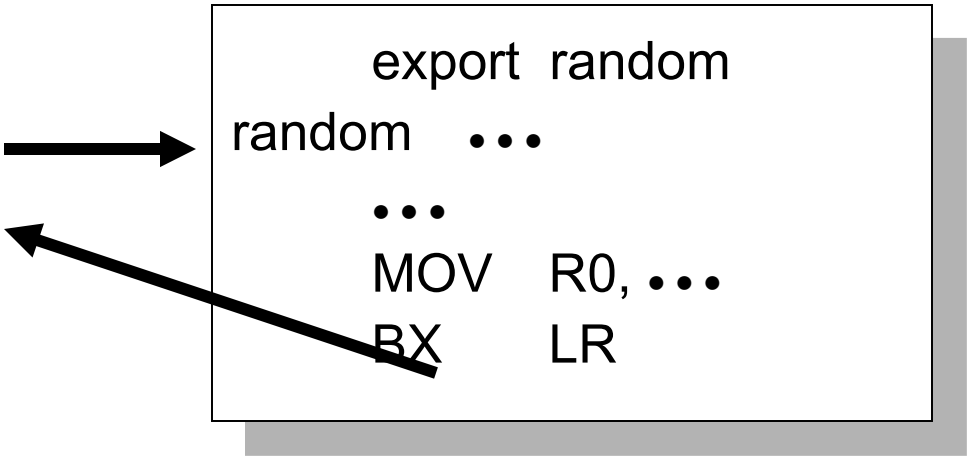
```
      export   display
display • • •
      • • •
      BX  LR
```



```
int32_t  random(void) ;
• • •
numb = random() ;
• • •
```

Compiler

```
• • •
BL  random
STR     R0,numb
 • • •
```



```
int64_t multiply(int32_t, int32_t) ;
    • • •
    product = multiply(a, b) ;
    • • •
```

Compile r

```
    • • •
    LDR     R0,a
    LDR     R1,b
    BL  multiply
    STRD   R0,R1,product
    • • •
```

```
        export  multiply
multiply   SMULL  R0,R1,R0,R1
           BX        LR
```

Function Parameters

- Parameters are copied into registers R0-R3 before the BL instruction that calls the function.
- Parameters are assigned to registers from left-to-right
- 8, 16 and 32-bit parameters each get 1 register
- 64-bit parameters each get a register pair (R0.R1, R1.R2, or R2.R3)

Function Return Value

- Functions return a single (scalar) result
- 8, 16 and 32-bit results must be returned in R0
- 64-bit results must be returned in R0 and R1:

  R0=least-significant half
  R1=most-significant half

Preserving Register Content

Registers R0-R3

- Functions do NOT need to preserve their content

Registers R4-R8

- Functions MUST preserve original content

  (PUSH on entry, POP before return)

Calling a Function

- Copy parameters into R0-R3, then call the function using a BL instruction.
- The return value (if any) will be in register R0 (and maybe R1) upon return
- Assume the function HAS modified R0-R3 and all the flags (N, C, V, Z, etc.)
- Assume the function has NOT modified R4-R8.

Writing a Function

Simple case: Does NOT call another function

- NEVER reference parameters using an identifier .
- Parameter values are already in registers R0-R3.
- Try to use only registers R0-R3.
- If your function uses any register from R4-R8, then you must PUSH those you use on entry and POP them just before the return.

Register Usage

Simple case: Does NOT call another function

If writing a routine in assembly that does NOT call another routine, then …

1. If you need to use R4-R8, preserve (PUSH) them on entry, restore (POP) them on return.
2. No need to PUSH & POP the link register (LR).
3. Try to use only R0-R3 for temporaries.

```
Function:
    … ▫ use only R0-R3
    BX LR
```

-or-

```
Function:
    PUSH {R4,R5,R6,R7,R8}
    … ▫ may use R0-R8
    POP  {R4,R5,R6,R7,R8}
    BX LR
```

Writing a Function

General case: Your function calls another function

- PUSH register LR on entry and POP it just before return.

- If parameters are needed AFTER calling the other function, then:

  1. PUSH registers R4-R7 as needed (see next step) on entry.
  2. Copy parameters from R0-R3 into R4-R7 using MOV instructions.
  3. Reference parameters using R4-R7 instead of R0-R3.
  4. Restore registers R4-R7 just before return

Register Assignment

General case: Your function calls another function

If writing a routine in assembly that calls another routine, then …

1. Preserve (PUSH) LR and R4-R8 on entry, restore (POP) on return.

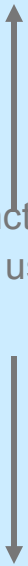
2. Use R4-R8 for temps and to hold copies of input parameters.

```
PUSH    {LR, R4,R5,…}
MOV     R4,R0 ; copy parameters
MOV     R5,R1
  …
  …
  …
POP     {LR, R4,R5,…}
BX  LR
```

Use R0-R3 to pass parameters and to compute expressions

Temporaries in Registers

```
func1   …
        …
```



no function calls;
OK to use R0 – R3


```
        …
        BX   LR
```

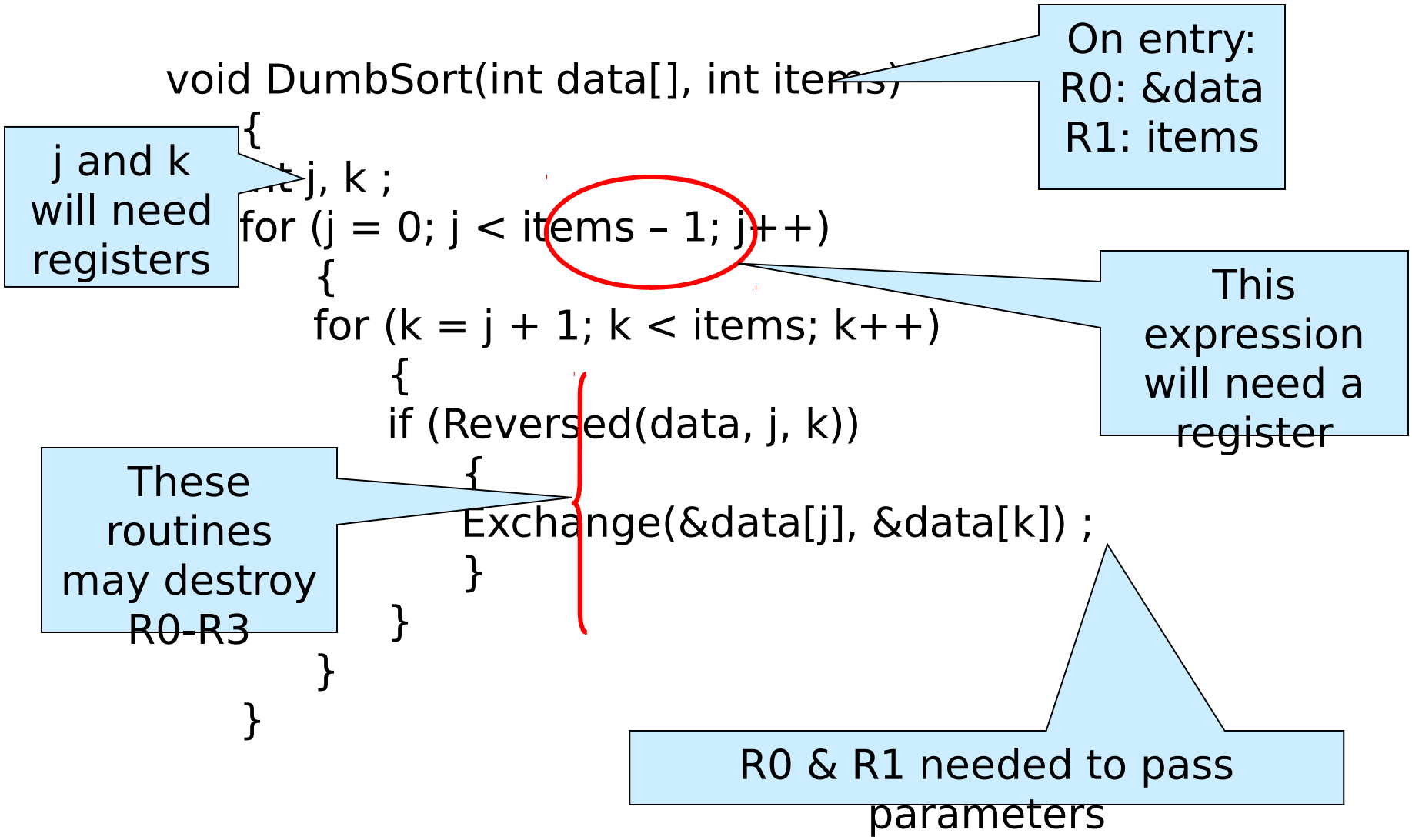
```
func2   PUSH    {R4,..,R8}
        …
        …

; If any of the registers R4-R8
; are modified by this function,
; those registers must be
; preserved on entry and restored
; just before the return.

        …
        …
        POP {R4,..,R8}      BX
        LR
```

```
func3   PUSH    {LR,..}
        …
; Since functions are not required
; to preserve R0-R3, then if used
; here, you must preserve/restore
; their values wherever this
function
; calls other functions.
        …
        PUSH    {R0,..,R3}
        BL  func4
        POP {R0,..,R3}
        …
        POP {LR,…}
        BX  LR
```

Register Assignment



```
int Reversed(int data[], int index1, int index2)
    {
    return data[index1] > data[index2] ;
    }
```

```
    EXPORT    Reversed

; R0 = &data
; R1 = index1
; R2 = index2

Reversed    LDR       R1,[R0,R1,LSL #2]    ; R1 = data[index1]
            LDR       r2,[R0,R2,LSL #2]    ; R2 = data[index2]
            CMP       R1,R2
            ITE       GT
            LDRGT     R0,=1
            LDRLE     R0,=0
            BX        LR
```

```
void Exchange(int *pItem1, int *pItem2)
    {
    int temp1 = *pItem1 ;
    int temp2 = *pItem2 ;
    *pItem1 = temp2 ;
    *pItem2 = temp1 ;
    }
```

```
        EXPORT    Exchange

    ; R0 = pItem1
    ; R1 = pItem2

Exchange  LDR       R2,[R0]       ; R2 = temp1
        LDR       R3,[R1]       ; R3 = temp2
        STR       R3,[R0]
        STR       R2,[R1]
        BX      LR
```

```
        EXPORT    DumbSort

    ; Parameters:  R0 = &data, R1 = items
    ; Temporaries: R4 = &data, R5 = items, R6 = j, R7 = k

DumbSort PUSH       {R4,R5,R6,R7,LR}
        MOV         R4,R0       ; use R4 for &data
        MOV         R5,R1       ; use R5 for items
        LDR         R6,=0       ; j = 0 ;
OuterTop:  SUB          R0,R5,#1  ; R0 = items – 1
        CMP         R6,R0       ; j < items – 1 ?
        BGE         OuterDone

    ; inner loop goes here ...

        ADD         R6,R6,#1  ; j++
        B       OuterTop
OuterDone:
        POP         {R4,R5,R6,R7,LR}
        BX      LR
```

```
; Inner Loop …

        ADD    R7,R6,#1       ; k = j + 1
InnerTop:  CMP    R7,R5           ; k < items?
        BGE    InnerDone
        MOV    R0,R4           ; R0 = &data
        MOV    R1,R6           ; R1 = j
        MOV    R2,R7           ; R2 = k
        BL  Reversed       ; Reversed?
        CMP    R0,#1
        BNE    NoExchange
        ADD    R0,R4,R6,LSL #2  ; R0 = &data[j]
        ADD    R1,R4,R7,LSL #2  ; R1 = &data[k]
        BL  Exchange       ; Exchange!
NoExchange:
        ADD    R7,R7,#1       ; k++
        B   InnerTop
InnerDone:
```