

# Part 5

## CHAPTER 3

### Architecture and Organization



1

These slides are provided with permission from the copyright for CS2208 use only. The slides must not be reproduced or provided to anyone outside the class.

All downloaded copies of the slides are for personal use only.

Students must destroy these copies within 30 days after receiving the course's final assessment.

## Pseudo instructions

- ❑ A *pseudo instruction* is an operation that the programmer can use when writing code.
  - The actual instruction ***does not*** have a ***direct*** machine language equivalent.
    - For example, you ***can't*** write `MOV r0,#0x12345678` to load register r0 with the 32-bit value 0x12345678 because the instruction is only 32 bits long in total.
    - Instead, you can use `LDR r0, = 0x12345678` *pseudo instruction*,  
 Yes, it is = not # **It is NOT** `MOV r0, = 0x12345678`
      - the assembler will generate suitable code to carry out the same action.
        - *store the constant*  $12345678_{16}$  in a so-called *literal pool* or *constant pool* somewhere in memory after the current AREA
        - *generates suitable code* to load the stored constant  $12345678_{16}$  to r0

*ADR R0, #0x12345678*

## Pseudo instructions

- ❑ Another *pseudo instruction* is **ADR r0, label**, which loads the 32-bit address of the line 'label' into register r0, using the appropriate code generated by the assembler.
- ❑ The following fragment demonstrates the use of the **ADR pseudo instruction**.

**ADR r1, MyArray** ;set up r1 to point to MyArray  
*add of MA* ; loads register r1 with the 32-bit address of MyArray

...

**LDR r3, r1** ;read an element using the pointer

**MyArray DCD 0x12345678** ;the address of this data will be loaded to r1

- ❑ The programmer does not have to know how the assembler generates suitable code to implement such *pseudo instructions*.
- ❑ All this is done automatically.
- ❑ This can be realized by utilizing the *program counter relative addressing*

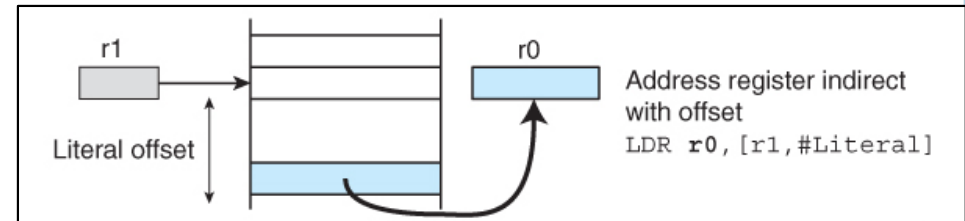
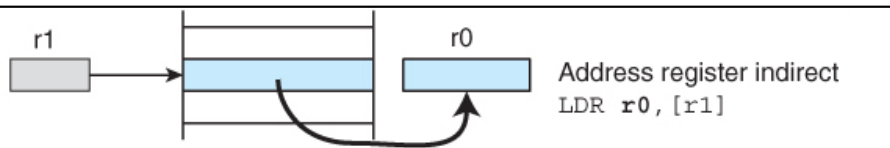
**But as a student, you need to know it!!**

# Program Counter Relative Addressing

- ❑ Register *indirect relative addressing allows* us to
  - *specify the location of an operand with respect to a register value.*

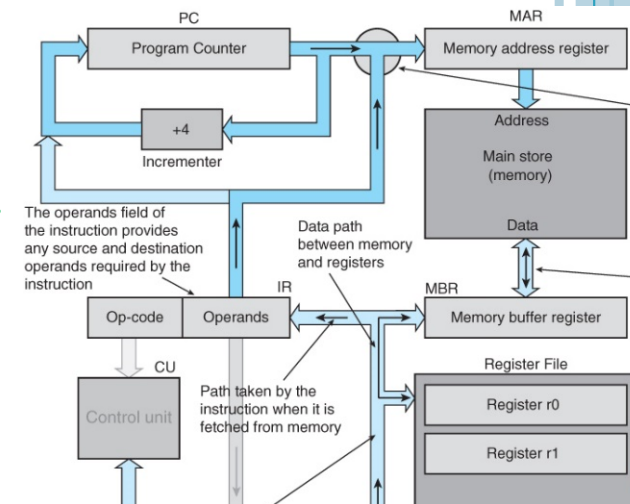
❑ LDR **r0**, [r1] specifies that the operand address is in r1

❑ LDR **r0**, [r1, #16] specifies that the operand is 16 bytes onward from r1.



- ❑ Suppose that we use *r15*, i.e., *the PC*, to generate an address by writing LDR **r0**, [PC, #16].

- The operand is 16 bytes onward from the PC
- i.e., 8 + 16 = 24 bytes from the current instruction.
  - *The ARM's PC in most of the cases is 8 bytes from the current instruction to be executed, due to **pipelining** (automatically fetches the next instruction before the current one has been executed).*



- ❑ If the program and its data are relocated elsewhere in memory, the *relative offset* does not change.

Having **ADR r4, P3** at line 08 will utilize the ADD instruction and the PC value to load the address of P3 in R4. To be translated to: **ADD r4, PC, #0x4**

## Pseudo instructions

FIGURE 3.20

To understand all these questions, try to put each pair of instructions in an assembly program and analyze the disassembly result.

What is the difference between **LDR r4, P3** and **ADR r4, P3**?

The **LDR** will load the value of P3 in r4. The **ADR** will utilize ADD/SUB to load the address of P3 in r4.

What will be the generated code if you replaced **LDR r4, P3** by **ADR r4, P3**?

What is the difference between **ADR r4, P3** and **LDR r4, = P3**?

Note that there is a difference between **LDR r4, P3** and **LDR r4, = P3**

The 1<sup>st</sup> one will load the VALUE of P3 in R4.

The 2<sup>nd</sup> one will copy the ADDRESS of P3 at the literal pool and then load the value of this address in R4.

Note that there is a difference between **LDR r4, = 0x1234** and **LDR r4, = P3**

The 1<sup>st</sup> one will copy 0x1234 at the literal pool and then load the value of this address in R4.

The 2<sup>nd</sup> one will copy the ADDRESS of P3 at the literal pool and then load the value of this address in R4.

Disassembly window showing 6 instructions (indicated by a red arrow and text):

4:	LDR	r0,=0x12345678	
0x00000000	E59F0018	LDR	R0,[PC,#0x0018] 0x00 + 0x08 + 0x18 = 0x20
5:	ADR	r1,Table	
0x00000004	E28F1008	ADD	R1,PC,#0x00000008 0x04 + 0x08 + 0x08 = 0x14
6:	ADR	r2,Table1	
0x00000008	E28F2008	ADD	R2,PC,#0x00000008 0x08 + 0x08 + 0x08 = 0x18
7:	LDR	r3, = 0xAAAAAAAA	
0x0000000C	E59F3010	LDR	R3,[PC,#0x0010] 0x0C + 0x08 + 0x10 = 0x24
8:	LDR	r4,P3	
0x00000010	E59F4004	LDR	R4,[PC,#0x0004] 0x10 + 0x08 + 0x04 = 0x1C
0x00000014	ABCD DCBA	BLGE	0xFF377304
0x00000018	FFFFFF	(???)	
0x0000001C	22222222	EORCS	R2,R2,#0x20000002
0x00000020	12345678	EORNES	R5,R4,#0x07800000
0x00000024	AAAAAAA	BGE	0xFEAAAAAD4
0x00000028	00000000	ANDEQ	R0,R0,R0

Assembly window showing PseudoInst.asm:

01	AREA	ConstPool, CODE, READONLY
02	ENTRY	
03		
04	LDR	r0,=0x12345678
05	ADR	r1,Table
06	ADR	r2,Table1
07	LDR	r3, = 0xAAAAAAAA
08	LDR	r4,P3
09		
10	Table	DCD 0xABCD DCBA
11	Table1	DCD 0xFFFFFFFF
12	P3	DCD 0x22222222

In "**ADR r4, P3**", the distance between **P3** and the **ADR** instruction MUST be represented as 0-255 and a rotation!!, while in "**LDR r4, = P3**", the distance Must be < 4096 (i.e., < 4K).



Addition	ADD,    ADC, <b>ADDS,   ADCS</b>
Subtraction	SUB,    RSB, <b>SUBS,   RSBS</b>
Negation	NEG, <b>NEGS</b>
Move	MOV,    MVN, <b>MOVS,   MVNS</b>
Multiplication	MUL,    MLA, <b>MULS,   MLAS</b>
Bitwise logic	AND,    ORR, <b>ANDS,   ORRS</b>
Comparison	<b>CMP,   CMN,  </b>
Shift	LSL,    LSR, <b>LSLS,   LSRS</b>

- ✓ *Data-Processing* instructions
- ✓ *Branching* instructions
- ✓ *Loading/Storing a single-register* instructions
- ✓ *Loading/Storing multi-registers* instructions

For example, to learn more about “**ADD**” instruction, you need to Google(“**ARM Keil add**”).  
It is usually the first link.

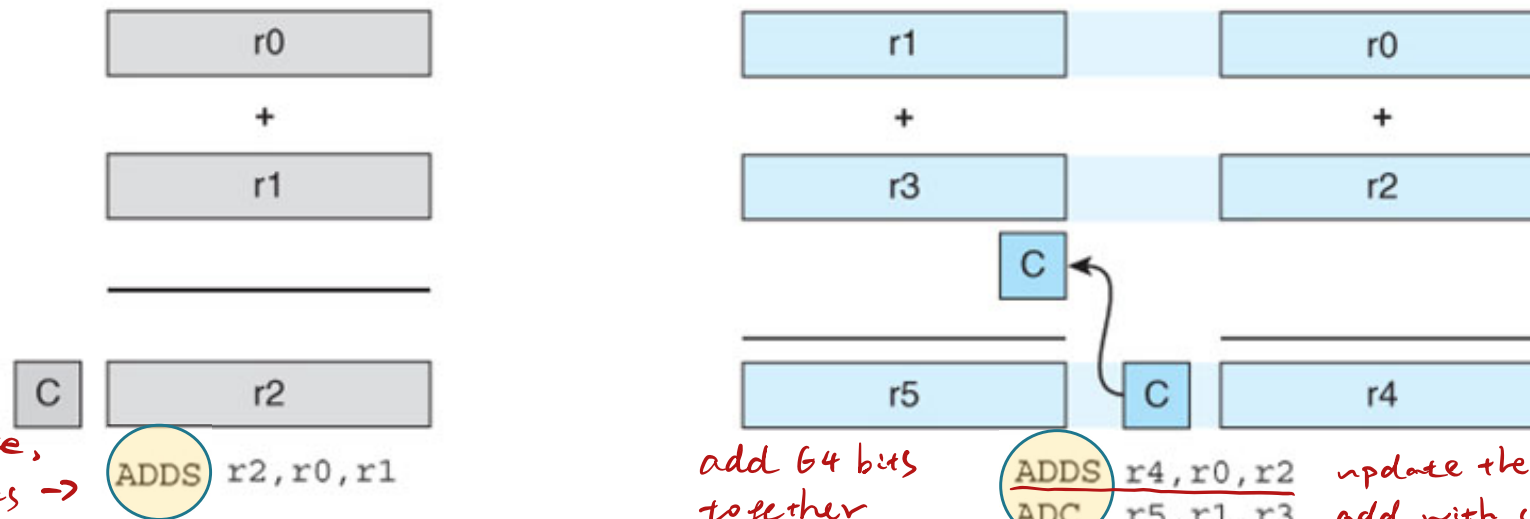
↳ instructions in holds are pseudo instructions.

# ARM's Data-Processing Instructions

## (Arithmetic Instructions: Addition)

- ❑ A simple ADD (and ADDS) instruction adds two 32-bit values.
- ❑ ARM also has an ADC (add with carry), as well as ADCS, that adds two 32-bit values together with the carry bit.
  - This allows extended precision arithmetic as Figure 3.21 demonstrates.

**FIGURE 3.21** Single- and extended-precision addition



It can be extended to integers of arbitrary lengths.

in this case, ADDS works → same result as ADD

(a) Single-precision addition. When `r0` is added to `r1`, the result is loaded into `r2`, and the carry bit is loaded into the carry flag.

add 64 bits together

(b) Double-precision extended addition. When `r0` is added to `r2`, any carry out is stored in the the carry bit. When `r1` is added to `r3`, the carry bit is added to their sum. In other words, the carry out generated by `ADDS r4, r0, r2` becomes the carry in used by `ADC r5, r1, r3`.

update the carry flag. add with carry flag.

ADC could add multiple bits.

## ARM's Data-Processing Instructions (Arithmetic Instructions: Subtraction)

- Beside the *normal subtraction* (SUB), ARM also provides *reverse subtraction* (RSB)

- SUB **r1**, r2, r3 ; [r1] ← [r2] - [r3]
- RSB **r1**, r2, r3 ; [r1] ← [r3] - [r2]

- RSB is useful, as ARM treats its operands differently.

- For example, to perform [r1] ← 10 - [r2], you can **not** use  
SUB **r1**, #10, r2 ; **THIS IS WRONG**  
instead, you can use  
RSB **r1**, r2, #10 ; **CORRECT**



# ARM's Data-Processing Instructions

## (Arithmetic Instructions: Subtraction)

□ Note that

RSB **r1**, #5

means

RSB **r1**, r1, #5

ADD **r1**, #5

means

ADD **r1**, r1, #5

When having a 3-operands instruction, if the 1<sup>st</sup> and 2<sup>nd</sup> operands are the same registers, it is allowed to short-hand the instruction by typing the register once. The assembler will take care of this short-hand and repeat the operand.

## ARM's Data-Processing Instructions (Arithmetic Instructions: Negation)

□ Negation is to subtract a number from 0

(*arithmetic complement, i.e., 2's complement*)

The end effect as if  
multiplying the  
operand by -1

○ ARM does **not** have a negation instruction as such

○ Instead, ARM provides a pseudo instruction called NEG

NEG **r1**, r2

NEG instruction has only two operands.

○ The RSB instruction is utilized to implement NEG

▪ To negate r2 (i.e., calculating  $0 - [r2]$ ) and store the result in r1,

NEG **r1**, r2

or

RSB **r1**, r2, #0

▪ To negate r2 (i.e., calculating  $0 - [r2]$ ) and store the result in r2,

NEG **r2**, r2

or

RSB **r2**, r2, #0

Or simple

RSB **r2**, #0

Can not be shortened to NEG r2

## ARM's Data-Processing Instructions (Arithmetic Instructions: Move and Move NOT)

- ❑ ARM provides a MOV instruction that copies the value of the second operand into the first operand

- To copy the content of r1 to r0,

MOV **r0**, r1 . . .

MOV instruction has only two operands.

*the value in r1 unchanged.*

- ❑ ARM also provides MVN (*move not*) that performs a **bitwise logical complement operation (i.e., logical NOT)** on the value of the second operand (i.e., *flipping each zero to one and each one to zero*), and places the result into the first operand

- To copy the **logical complement** of the content of r1 to r0,

MVN **r0**, r1 . . .

MVN instruction has only two operands.

### MOVS and MVNS

- ✓ **Update the N, Z and C flags according to the result**
- ✓ **Do NOT affect the V flag**

# ARM's Data-Processing Instructions

## (Arithmetic Instructions: Multiplication)

**MUL can not be shortened to two operands**

- ❑ The *multiply* instruction, MUL Rd, Rm, Rs
  - Takes two 32-bit signed integer values from registers Rm and Rs
  - Forms their 64-bit product
  - Stores in 32-bit register Rd *the lower-order 32 bits of the 64-bit product.*

```
MOV    r0, #121      ;load r0 with 121
MOV    r1, #96        ;load r1 with 96
MUL    r2, r0, r1     ;r2 = r0 x r1
```

- ❑ A 32-bit by 32-bit **multiplication** is *truncated* to the **lower-order 32 bits**.
- ❑ In MUL instruction, *same register can't* be used to specify both the *destination* Rd and the *operand* Rm, *Since it is implemented by an accumulator*
  - because ARM's implementation uses **Rd as a temporary register** during multiplication. This is a feature of the ARM processor.
- ❑ ARM *does not* allow multiply by a constant

### MULS

- ✓ *Updates the N and Z flags according to the result*
- ✓ *Corrupts the C and V flags*

## ARM's Data-Processing Instructions (Arithmetic Instructions: Multiplication)

- ❑ ARM has a *multiply and accumulate* instruction, MLA, that
  - performs a multiplication and adds the product to a running total.

- ❑ MLA instruction has a four-operand form:

MLA **Rd**, Rm, Rs, Rn ; [Rd] = [Rm] × [Rs] + [Rn] .

**MLA can not be shortened to three operands**

- ❑ As in the normal MUL instruction,
  - A 32-bit by 32-bit **multiplication** is *truncated* to the **lower-order 32 bits**.
  - *same register can't* be used to specify both the *destination* **Rd** and the *operand* Rm
- ❑ ARM *does not* allow *multiply by a constant*

### MLAS

- ✓ *Updates the N and Z flags according to the result*
- ✓ *Corrupts the C and V flags*

## ARM's Data-Processing Instructions (Arithmetic Instructions: Multiplication)

- ❑ ARM's *multiply and accumulate* supports the calculation of an *inner product* (a.k.a. *dot product*).
- ❑ The inner product of two vectors  $\mathbf{a} = [a_1, a_2, \dots, a_n]$  and  $\mathbf{b} = [b_1, b_2, \dots, b_n]$  is defined as

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$



# ARM's Data-Processing Instructions

## (Arithmetic Instructions: Multiplication)

- ❑ The following program shows how the multiply and accumulate instruction is used to form the **inner product** between n-component vectors, Vector1 and Vector2

```

AREA MultiplyAndAccumulateExample, CODE, READONLY
ENTRY
n    EQU    4                ;4 components in this example
MOV    r4, #n                ;r4 is the loop counter
MOV    r3, #0                ;clear the inner product
ADR    r5, Vector1           ;r5 points to vector 1
ADR    r6, Vector2           ;r6 points to vector 2

Loop LDR    r0, [r5], #4      ;REPEAT
                                ; read a component of A
                                ; and update the pointer
    LDR    r1, [r6], #4      ; get the second element
                                ; and update the pointer
    MLA    r3, r0, r1, r3    ; add new product term to the total
                                ; (r3 = r3 + r0·r1)
    SUBS   r4, r4, #1        ; decrement the loop counter
                                ; (and remember to set the CCR)
    BNE    Loop              ;UNTIL all done

Vector1 DCD    1, 2, 3, 4
Vector2 DCD    2, 3, 4, 5
END

```

## ARM's Data-Processing Instructions (Arithmetic Instructions: Multiplication)

- ❑ In addition to the 32-bit MUL and MLA, ARM includes several forms of multiplication instruction, including
- UMLL     Unsigned long multiply  
              ( $R_m \times R_d$  yields 64-bit product in two registers)
  - UMLAL    Unsigned long multiply-accumulate
  - SMULL    Signed long multiply
  - SMLAL    Signed long multiply-accumulate



**We will NOT  
cover these  
instructions.**

## ARM's Data-Processing Instructions (Arithmetic Instructions: Division)

- ❑ ARM *does not implement a division* operation
- ❑ If needed, the programmer must write a suitable division routine to implement division

# ARM's Data-Processing Instructions (Bitwise Logical Operations)

The rest of the bits  
in these instructions  
are zeros.

- ❑ Logical operations are known as *bitwise operations* because they are applied to the individual bits of a register

It is  
**ORR**,  
*not*  
**OR**.

AND **r2**, r1, r0 → Example: 11001010 . 00001111 → 00001010

ORR **r2**, r1, r0 → Example: 11001010 + 00001111 → 11001111

EOR **r2**, r1, r0 → Example: 11001010 ⊕ 00001111 → 11000101

MVN **r2**, r0 → Example: 11001010 →  
1111111111111111111111111111111100110101

- ❑ The **MVN** operation can also be performed by using an **EOR** with the second operand equal to FFFFFFFF<sub>16</sub> (i.e., 32 1's *in a register*)
- the value of  $x \oplus (11...1111)_2$  is = **NOT**  $x$ .

MOV r1, #0xFFFFFFFF

EOR **r2**, r1, r0 ; Same as MVN **r2**, r0

## ANDS, ORRS, EORS, and MVNS

- ✓ Update the N, Z and C flags according to the result
- ✓ Do NOT affect the V flag

## ARM's Data-Processing Instructions (Bitwise Logical Operations)

❑ **Example 1:** suppose that


- register r0 contains the 8 bits **bbbbbbxx**,
- register r1 contains the 8 bits **bbbyyybb** and
- register r2 contains the 8 bits **zzzbbbb**,

where

- **xx**, **yyy**, and **zzz** represent the bits of desired fields and
- the **b**'s are unwanted bits.

❑ We wish to pack these bits to get the final value **zzzyyyxx** stored in r0.

❑ We can achieve this by:



```

AND    r0, r0, #2_11           ;Mask r0 to two bits xx
AND    r1, r1, #2_11100        ;Mask r1 to three bits yyy
AND    r2, r2, #2_11100000     ;Mask r2 to three bits zzz
ORR    r0, r0, r1              ;Merge r1 and r0 to get 000yyyxx
ORR    r0, r0, r2              ;Merge r2 and r0 to get zzzyyyxx
  
```

❑ *The Keil assembler uses a prefix*

- **2\_** to indicate binary
- **8\_** to indicate octal
- **0x** or **&** to indicate hexadecimal
- **no prefix** to indicate decimal

## ARM's Data-Processing Instructions (Bitwise Logical Operations)

- ❑ **Example 2:** suppose we have an 8-bit value **abcdefgh** and
- ❑ we wish to
  - clear bits **b** and **d**,
  - set bits **a**, **e**, and **f**, and
  - toggle (invert) bit **h**,
 i.e., generate the following output **10c011g $\bar{h}$**

- ❑ We can achieve this by:

```

AND  r0, r0, #2_10101111 ;Clear bits b and d to get  a0c0efgh
ORR  r0, r0, #2_10001100 ;Set bits a, e, and f to get  10c011gh
EOR  r2, r2, #2_1         ;Toggle bit h to get the result

```



## ARM's Data-Processing Instructions (Bitwise Logical Operations)

- ❑ **ARM** provides a *bit clear* instruction, **BIC**, that
  - ANDs its first operand with the complement of its second operand.
- ❑ **Example:** suppose we have  $r1 = 10101010$  and  $r2 = 00001111$ .
  - The instruction **BIC r0, r1, r2** yield  $10100000$

### BICS

- ✓ *Updates the N, Z and C flags according to the result*
- ✓ *Does NOT affect the V flag*

## ARM's Data-Processing Instructions (Arithmetic Instructions: Comparison)

- ❑ In ARM, updating the condition flags can be *implicit* or *explicit*.
- ❑ Both implicit and explicit flags' updates modify the contents of *the condition code register (CCR)*, a.k.a. *current program status register (CPSR)*, which is later can be tested to determine whether execution continues in sequence, or a branch is taken

- Example of *implicit* updates for the condition flags

SUBS **r1**, r1, r2

- Example of *explicit* updates for the condition flags

CMP r1, r2

This instruction will evaluate  $r1 - r2$  *without storing the result*, and set the *condition code register*

```

CMP    r1, r2          ;is r1 = r2?
BEQ    DoThis          ;if equal then goto DoThis
ADD    r1, r1, #1      ;else add 1 to r1
B      Next            ;jump past the then part
.
DoThis SUB r1, r1, #1    ;subtract 1 from r1
Next   ...             ;both forks end up here

```

## ARM's Data-Processing Instructions (Arithmetic Instructions: Comparison)

- **ARM** has *four* instructions in its *test-and-compare* group which *explicitly update the condition code flags* (i.e., *no need to append an S to any of them*)
- **CMP** (compare instruction)
    - *Subtracts* the second operand from the first and *update all flags*
  - **TEQ** (test equivalent instruction)
    - Determines whether two operands are equivalent or not (*similar to EORS, except that the result is discarded*)
  - **TST** (test instruction)
    - Compares two operands by **ANDing** them together and *update flags*
    - Usually used to *test individual bits*;
    - `TST r0, #2_00100000 ; AND r0 with 00100000 to test bit 5`  
*••• BNE LowerCase ; If bit 5 is 1, jump to lowercase*
  - **CMN** (compare negative instruction).
    - 2's complements the second operand before performing the comparison  
`CMN r1, r2 ; evaluates [r1] - (-[r2])`  
*••• ; i.e., evaluate [r1] + [r2]*

It is  
**BNE**,  
*not*  
**BEQ**.

It is **CMN**, *not* **CPN**. Correct the book, page 178

### TEQ and TST

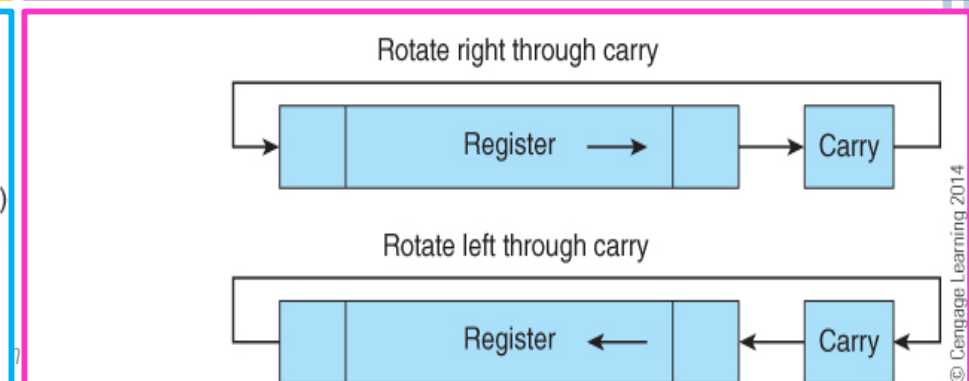
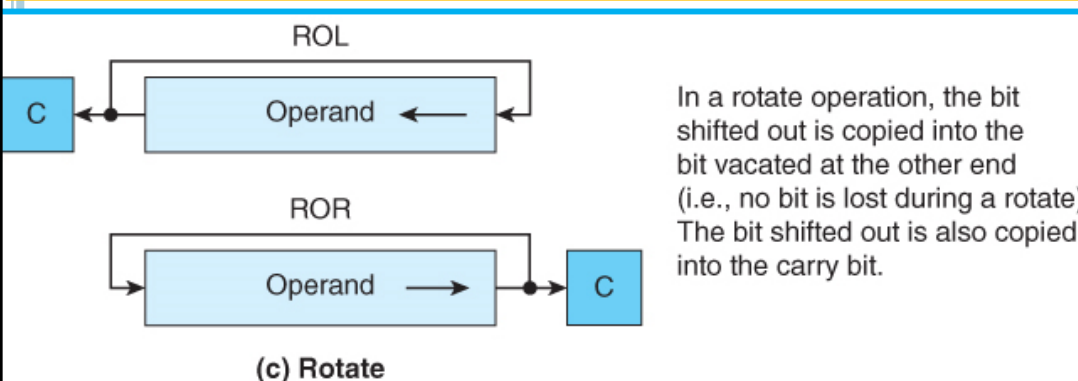
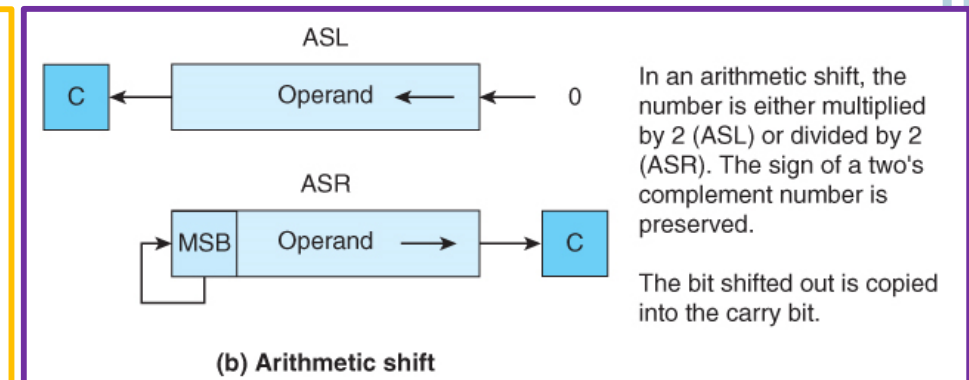
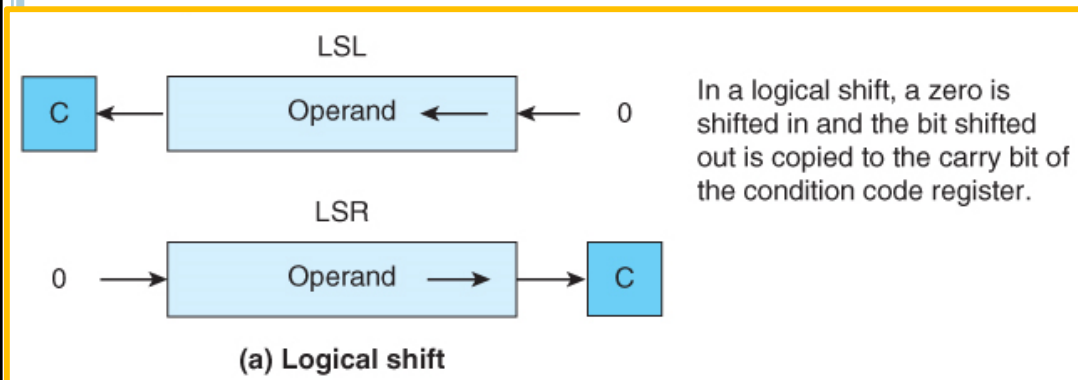
- ✓ *Update the N, Z and C flags according to the result*
- ✓ *Do NOT affect the V flag*

**CMP and CMN** update **ALL** flags  
*according to the result*

# ARM's Data-Processing Instructions

## (Shift Operations)

- ❑ *Shift* operations **move bits** one *or more* places *left* or *right*.
  - *Logical shifts*
    - *insert a 0* in the vacated position.
  - *Arithmetic shifts*
    - *replicate the sign-bit* during a right shift
  - *Circular shifts*
    - *the bit shifted-out of one end is shifted-in the other end*  
i.e., the register is treated as a ring
  - *Circular shifts through carry*
    - *included the carry bit in the shift path*



# ARM's Data-Processing Instructions (Shift Operations)

Examples of **logical shifts** on a *16-bit value*

Source string	Direction	Number of shifts	Destination string
<b>0</b> 110011111010111	Left	1	110011111010111 <b>0</b>
<b>01</b> 10011111010111	Left	2	10011111010111 <b>00</b>
<b>011</b> 0011111010111	Left	3	0011111010111 <b>000</b>
011001111101011 <b>1</b>	Right	1	<b>0</b> 011001111101011
01100111110101 <b>11</b>	Right	2	<b>00</b> 01100111110101
0110011111010 <b>111</b>	Right	3	<b>000</b> 0110011111010

These examples are hypothetical, as ARM registers are 32 bits, not 16 bits.

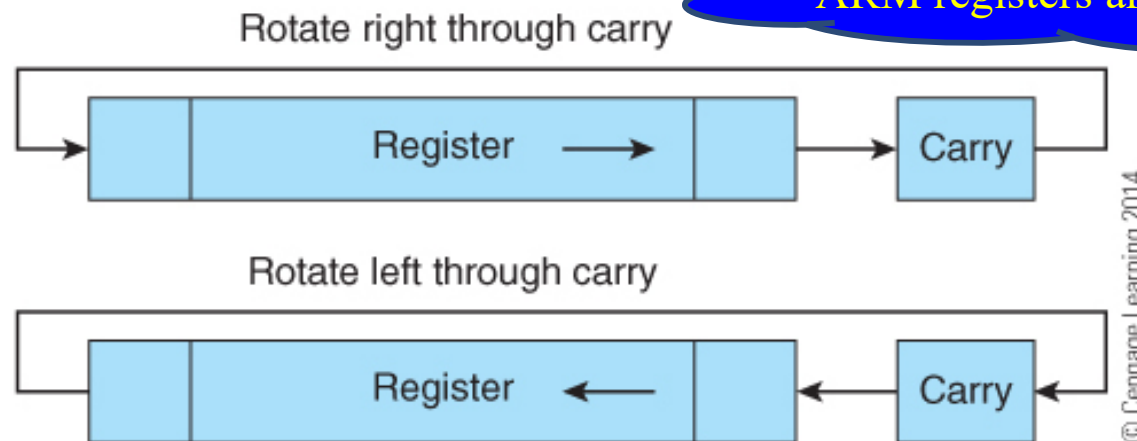
## ARM's Data-Processing Instructions (Shift Operations)

- ❑ The *rotate through carry* instruction (sometimes called *extended shift*) included the carry bit in the shift path.
  - The carry bit is shifted into the bit of the word vacated, and
  - the bit of the word shifted out is shifted into the carry.
- ❑ If the **carry** = 1 and the *eight-bit word* to be shifted is 01101110, a *rotate left through carry* would give 11011101 and

**carry** = 0

**FIGURE 3.24**

The rotate through carry



This example is hypothetical, as ARM registers are 32 bits, not 8 bits.



# Implementing a Shift Operation on the ARM

- ❑ **ARM** *has no explicit shift operations!!*.
- ❑ **ARM** *combines shifting with other data processing operations*, where
  - the **second operand** in the arithmetic operation (i.e., the **LAST parameter in the assembly arithmetic instruction**) is allowed to be shifted **before** it is used.
  - For example,  
ADD **r0**, r1, r2, LSL #1 ;  $[r0] \leftarrow [r1] + [r2] \times 2$ 
    - logically shift left the contents of r2,
    - add the result to the contents of r1, and
    - put the results in r0
- ❑ **ARM** *also combines shifting with MOV and MVN instructions*
  - For example,  
MOV **r3**, r3, LSL #1 ;  $[r3] \leftarrow [r3] \times 2$
  - **ARM** provides **pseudo** shift instructions, which are translated to **MOV** instructions.
  - For example,  
LSL **r3**, r3, #1 ; will be converted to:  
MOV **r3**, r3, LSL #1  
or simply  
LSL **r3**, #1

# Implementing a Shift Operation on the ARM

- ❑ **ARM** support both *static* and *dynamic* shifts (except *rotate through carry* instruction which allows *only one single shift* per instruction)
  - In *static shift*, the number of shift places
    - is determined *when the code is written*
    - can only have the following values, inclusive:
      - **LSL**: allowable values are from **#0** to **#31** (*32 different values*)
      - **LSR**: allowable values are from **#1** to **#32** (*32 different values*)
      - **ASR**: allowable values are from **#1** to **#32** (*32 different values*)
      - **ROR**: allowable values are from **#1** to **#31** (*31 different values*)
  - The remaining value is used to encode RRX*
  - In *dynamic shift*, the number of shift places
    - is determined *when the code is executed, i.e., at run time*
- ❑ You can perform *dynamic shifts* as follow

```
MOV r4, r3, LSL r2      ; [r4] ← [r3] × 2^r2
or
LSL r4, r3, r2          ; [r4] ← [r3] × 2^r2
```

This instruction

- shifts the contents of r3 left by the value in r2 and
- puts the result in r4.

○ *If the value in r2 is ≥ 32, zero will be stored in r4*

# Implementing a Shift Operation on the ARM

- ❑ **ARM** implements only the following five shifts

LSL   logical shift left

LSR   logical shift right

ASR   arithmetic shift right

ROR   rotate right

RRX   rotate right through carry                      (one shift)

- ❑ *Other shift operations must be synthesized by the programmer.*

# Implementing a Shift Operation on the ARM

## ❑ *Other shift operations must be synthesized by the programmer.*

- An *arithmetic shift left* is effectively the same as a *logical shift left*
- For a 32-bit value,  
an *n-bit rotate shift left* is identical to a *32 – n rotate shift right*
- *Rotate left through carry* can be implemented by means of  
ADCS `r0, r0, r0 ; add r0 to r0 with carry and set the flags`
  - The instruction means  $r0 + r0 + C$ , i.e.,  $2 \times r0 + C$ , i.e.,
    - shifting left the content of r0
    - store the value of C in the vacant bit to the left, and
    - storing the shifted-out bit in the carry flag