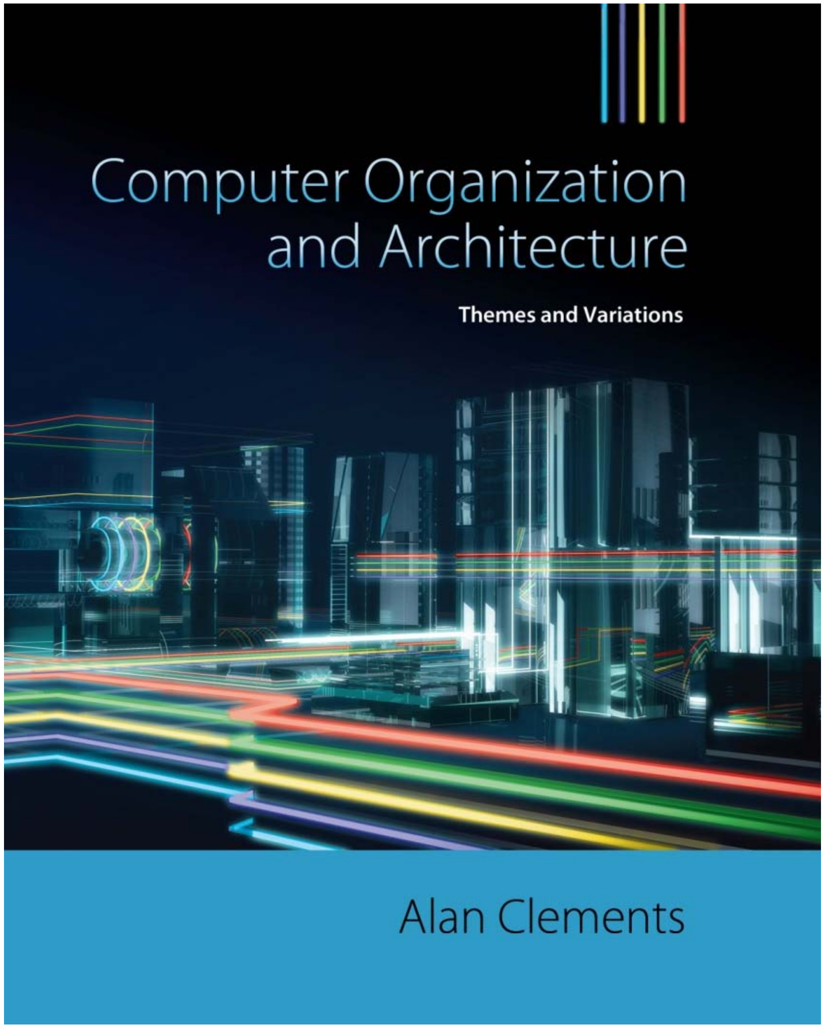
Part 0x5

CHAPTER 3

Architecture and Organization



These slides are being provided with permission from the copyright for in-class (CS2208B) use only. The slides must not be reproduced or provided to anyone outside of the class.

All download copies of the slides and/or lecture recordings are for personal use only. Students must destroy these copies within 30 days after receipt of final course evaluations.

Music: *“Corporate Success”* by *Scott Holmes*, used under *Attribution-NonCommercial License*

CENGAGE Learning

ARM's Data-Processing Instructions

Addition	**ADD, ADC,** ADDS, ADCS
Subtraction	**SUB, RSB,** SUBS, RSBS
Negation	**NEG,** NEGS
Move	**MOV, MVN,** MOVS, MVNS
Multiplication	**MUL, MLA,** MULS, MLAS
Bitwise logic	**AND, ORR, EOR, BIC,** ANDS, ORRS, EORS, BICS
Comparison	**CMP, CMN, TEQ, TST**
Shift	**LSL, LSR, ASR, ROR, RRX,** LSLS, LSRS, ASRS, RORS, RRXS

update the prosessor. Flags.

To learn more about any ARM assembly instruction, you can just Google the words ARM Keil + the operation-code.

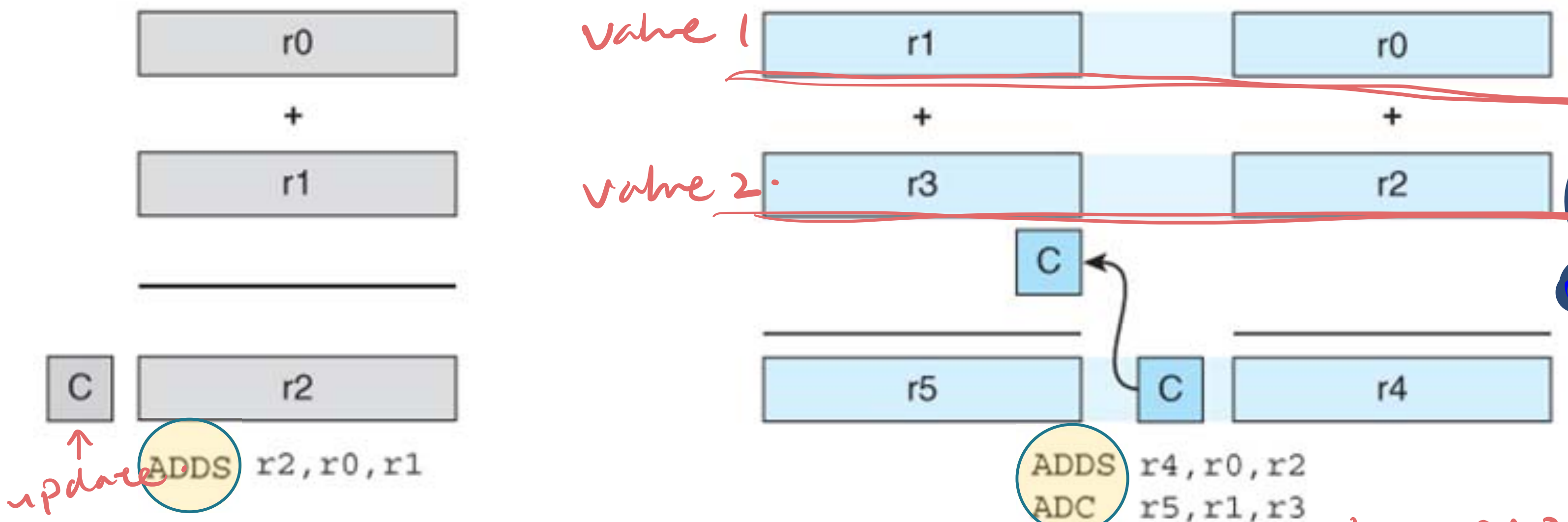
For example, to learn more about "ADD" instruction, you need to Google("ARM Keil add"). It is usually the first link.

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. deal with more than 32 bits value e.g. add up two 64 bits value.
  - o This allows extended precision arithmetic as Figure 3.21 demonstrates.

FIGURE 3.21 Single- and extended-precision addition



(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.

(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`.

© Cengage Learning 2014

Can be extended to integers of arbitrary length

break into parts.

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
    ```

In ARM, the 2nd operand must be a register

the 3rd can be a register or a constant.

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

- Note that

```
RSB r1,#5
```

means

```
RSB r1,r1,#5
```

r1 -= 5

When having 3 operands instruction, if the 1st and 2nd 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.

```
ADD r1, #5
```

means

```
ADD r1,r1,#5
```

r1 += 5.

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 r2 = -r2.

```
RSB r2,r2,#0
```

Can not be shortened to NEG r2

NEG has only 2 operants.

```
Or simple
RSB r2,#0
```

short hand.

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.

- 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.

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.

last.

64 bits. => last 32 bits

- In `MUL` instruction, *same register* ***can't*** be used to specify both the *destination* **Rd** and the *operand* Rm,
  - 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

all operants must be registers.

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

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*).

dot. e.g. array

- The inner product of two vectors **a** = [a_1, a_2, ..., a_n] and **b** = [b_1, b_2, ..., b_n] is defined as

$$s = \boldsymbol{a} \, . \, \boldsymbol{b} = \sum_{1}^{n} a_i \, b_i = a_1 \, b_1 + a_2 \, b_2 + \ \dots + a_n \, b_n$$

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   count the number in the array.
n        EQU  4              ;4 components in this example
         MOV  r4,#n          ;r4 is the loop counter
         MOV  r3,#0      add ;clear the inner product   accumulator.
         ADR  r5,Vector1     ;r5 points to vector 1
         ADR  r6,Vector2     ;r6 points to vector 2

Loop     LDR  r0,[r5],#4     ;REPEAT  take r5 as a pointer, read 4 bytes
                             ;  read a component of A   after r5, load it to r0.
                 auto increment
                             ;  and update the pointer
         LDR  r1,[r6],#4     ;  get the second element
these two can't be the same  ;  and update the pointer
                   register.
         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)
   if r4≠0
         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 (Rm × Rd yields 64-bit product in two registers)
UMLAL	Unsigned long multiply-accumulate
SMULL	Signed long multiply
SMLAL	Signed long multiply-accumulate

these will not be used in this course.

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

- ❑ ARM *does not implement a division* operation (at least in its basic models)
- ❑ If needed, the programmer must write a suitable division routine to implement division

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

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

AND each bits, to all 32 bits.

It is ORR, *not* OR.

1∧0   1∧1 1∧1
0∧0   1∧0 0∧1

```
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 ➔
                           11111111111111111111111100110101
```

Move not.
Flip each bits 1↔0

the rest of 32 bits.

- ❑ The MVN operation can also be performed by using an EOR with the second operand equal to FFFFFFFF_{16} (i.e., 32 1's ***in a register***)
  - o the value of $x \oplus (11\ldots1111)_2$ is = **NOT** x.

```
MOV r1,#0xFFFFFFFF
EOR r2,r1,r0         ;Same as MVN r2,r0
```

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 zzzbbbbb,

  where
  - x, y, and z 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: binary.

```
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
```

It is ORR, *not* OR.

- ***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 string **abcdefgh** and
- we wish to
  - clear bits b and d, -> make sure to be 0
  - set bits a, e, and f, and -> make sure to be 1
  - toggle (invert) bit h, EOR.

  i.e., generate the following output $10c011g\bar{h}$

- We can achieve this by:

```
                b  d.
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
```

clear => AND
set => ORR
invert => EOR.

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

- ❑ *ARM* provides a *bit clear* instruction, **BIC**, that
  - o 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`
  - ▪ Same thing can be done using `AND r0, r1, #0xFFFFFFF0`

complement of the mask.

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

- In ARM, Comparisons can be *implicit* or *explicit*
- Both implicit and explicit comparisons 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* comparison

    ```
    SUBS r1,r1,r2
    ```

  - Example of *explicit* comparison

    compare.
    ```
    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*

      no result is made
  - **TEQ** (test equivalent instruction)
    - Determines whether two operands are equivalent or not (similar to **EORS**, except that the result is discarded)
    - **TEQ** *does **not** update the **overflow flag** or the **carry flag***

      no meaning
  - **TST** (test instruction)
    - Compares two operands by **ANDing** them together and *update flags*
    - Usually used to *test individual bits;*
    - **TST** *does **not** update the **overflow flag** or the **carry flag***

```
TST r0, #2_00100000   ;AND r0 with 00100000 to test bit 5
BNE LowerCase         ;If bit 5 is 1, jump to lowercase
```

It is BNE, *not* BEQ.

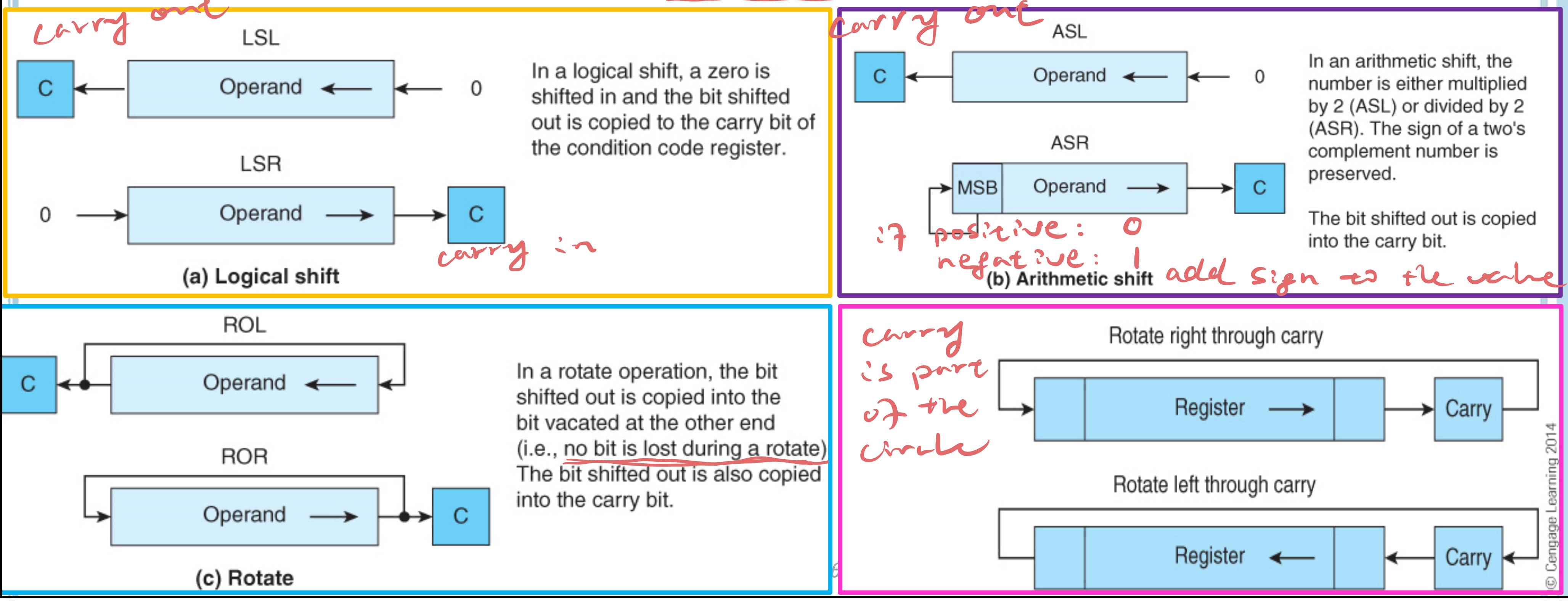
  - **CMN** (compare negative instruction).
    - 2's complements the second operand before performing the comparison

```
CMN r1, r2    ; evaluates [r1] – (-[r2])
              ; i.e., evaluate [r1] + [r2]
```

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

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*



In a logical shift, a zero is shifted in and the bit shifted out is copied to the carry bit of the condition code register.

In an arithmetic shift, the number is either multiplied by 2 (ASL) or divided by 2 (ASR). The sign of a two's complement number is preserved.

The bit shifted out is copied into the carry bit.

In a rotate operation, the bit shifted out is copied into the bit vacated at the other end (i.e., no bit is lost during a rotate). The bit shifted out is also copied into the carry bit.

ARM’s Data-Processing Instructions (Shift Operations)

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

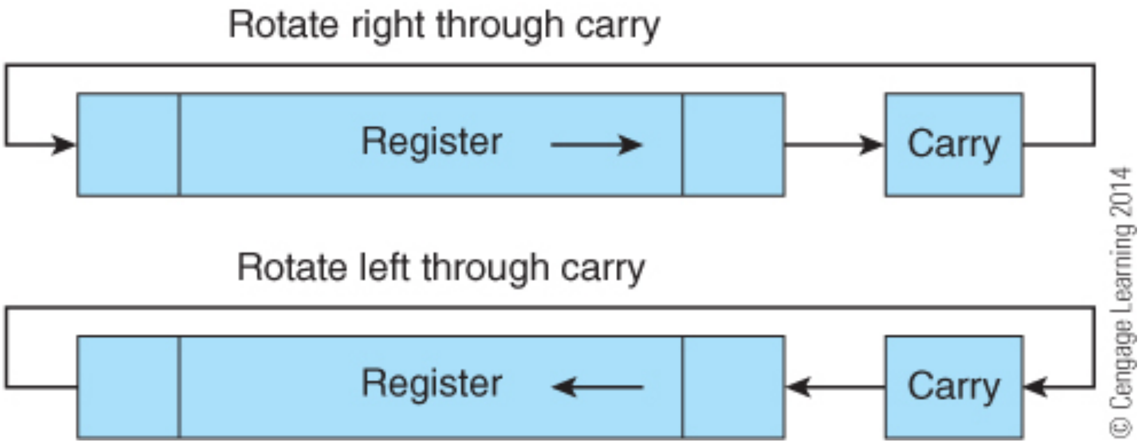
Source string	Direction	Number of shifts	Destination string
0110011111010111	Left	1	110011111010111**0**
0110011111010111	Left	2	10011111010111**00**
0110011111010111	Left	3	0011111010111**000**
011001111101011**1**	Right	1	**0**011001111101011
01100111110101**11**	Right	2	**00**01100111110101
0110011111010**111**	Right	3	**000**0110011111010

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



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] ← [r1] + [r2] × 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 moving operations*
  - For example,

    ```
    MOV r3,r3,LSL #1            ; [r3] ← [r3] × 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: 0 : no change of the value.
      - logical **LSL**: allowable values are from ***#0*** to ***#31*** ***(32 different values)***
      - **LSR**: allowable values are from ***#1*** to ***#32*** ***(32 different values)***
      - arithmetic **ASR**: allowable values are from ***#1*** to ***#32*** ***(32 different values)***
      - rotate **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 have to be synthesized by the programmer.***

Implementing a Shift Operation on the ARM

- ***Other shift operations have to 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 × 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