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

- 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-order32 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`,
  - 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

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].
```

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

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

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

- The following code fragment 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 (Rm × Rd yields 64-bit product in two registers)
  - UMLAL Unsigned long multiply-accumulate
  - SMULL Signed long multiply
  - SMLAL Signed long multiply-accumulate

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 has to 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

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 →
                          11111111111111111111111100110101
```

- The NOT 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***)
  - the value of $\boldsymbol{x} \oplus \mathbf{(11...1111)_2}$ is = **NOT** $\boldsymbol{x}$.

```
MOV r1,#0xFFFFFFFF
EOR r2,r1,r0
```

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

- Example: 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:

```
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 to indicate hexadecimal***
  - ***no prefix to indicate decimal***

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

- Example: suppose we have an 8-bit string **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
```

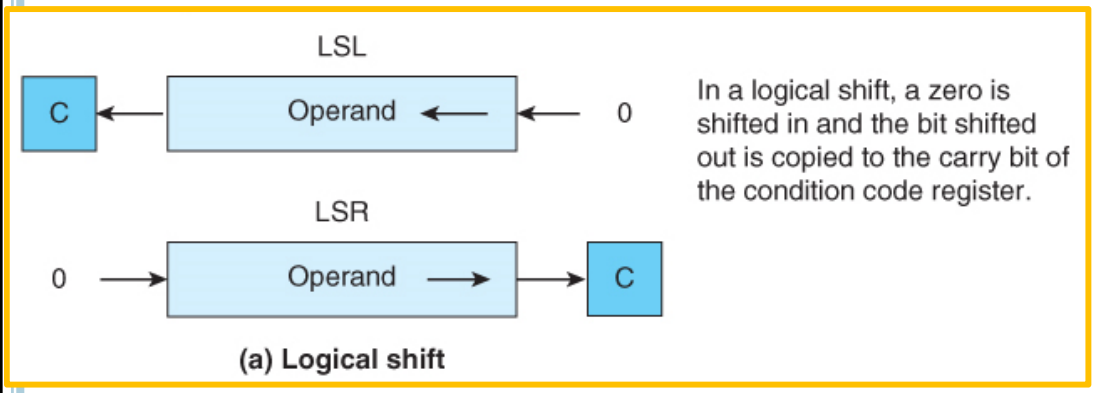
ARM's Data-Processing Instructions (Bitwise Logical Operations)

- ***ARM*** provides a *bit clear* instruction, **`BIC`**, that
  - ANDs its first operand with the *complement* of its second operand.

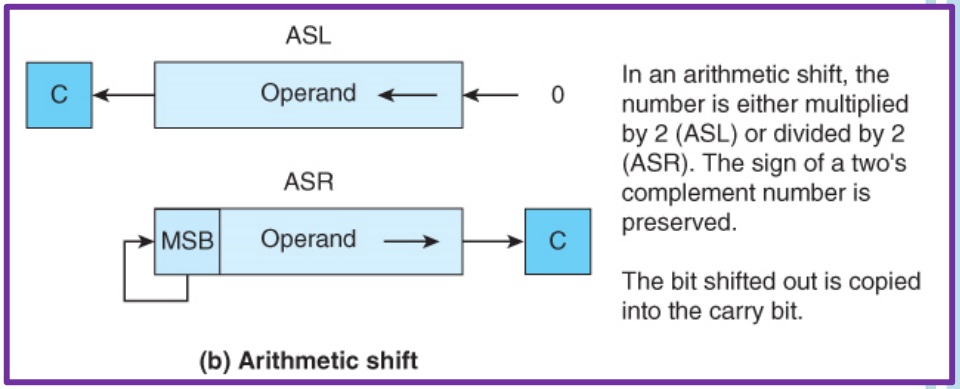
- Example: suppose we have `r1 = 10101010` and `r2 = 00001111`.
  - The instruction `BIC r0, r1, r2` yield `10100000`
  - Same thing can be done using `AND r0, r1, #0xFFFFFFF0`

ARM's Data-Processing Instructions (Shift Operations)

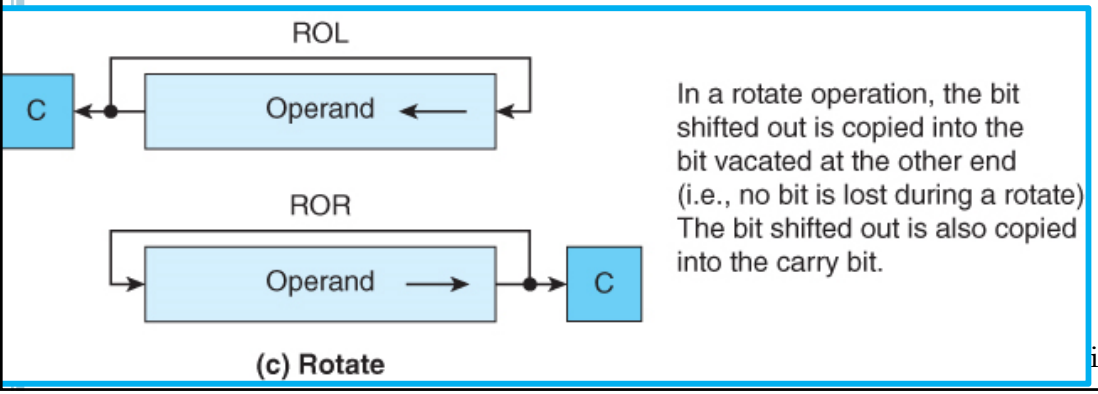
- *Shift* operations move bits one ***or more*** places *left* or *right*.
  - ***Logical shifts***
    - *insert a 0* in the vacated position.
  - ***Arithmetic shifts***
    - *replicate the sign-bit* during a right shift
  - ***Circular shifts***
    - *the bit shifted out of one end is shifted in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



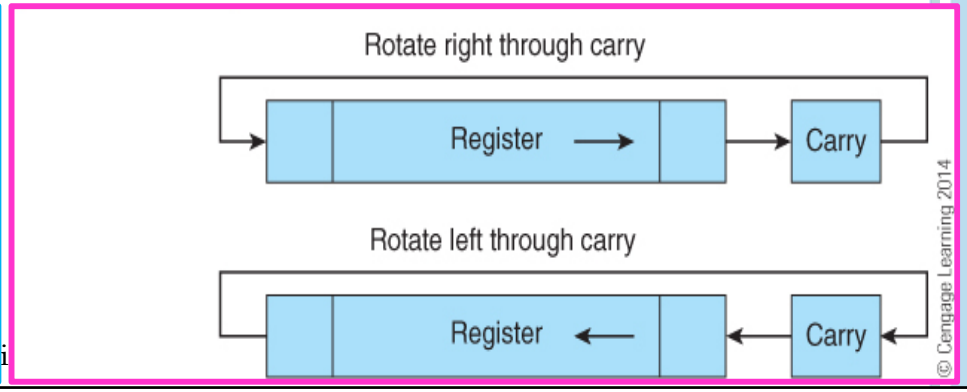
(a) Logical shift



(b) Arithmetic shift



(c) Rotate



ARM's Data-Processing Instructions (Shift Operations)

Examples of **logical shifts**

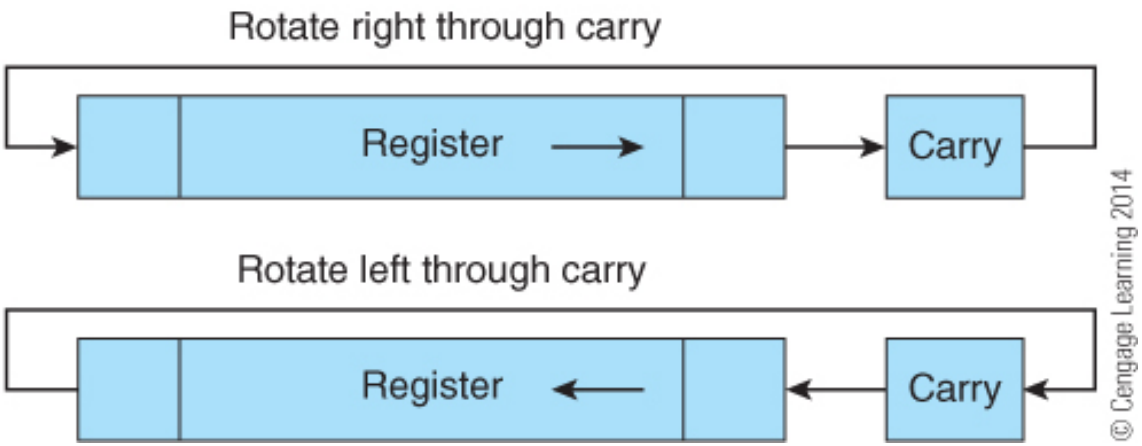
Source string	Direction	Number of shifts	Destination string
0110011111010111	Left	1	1100111110101110
0110011111010111	Left	2	1001111101011100
0110011111010111	Left	3	0011111010111000
0110011111010111	Right	1	0011001111101011
0110011111010111	Right	2	0001100111110101
0110011111010111	Right	3	0000110011111010

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 ***third 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*
  - This way, a shift operation can be performed as a stand alone operation.

  - 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`

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 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*
    - Except in some processors (including **ARM**)
      - the overflow flag will not be updated after a *logical shift*

    Note: the overflow flag should be updated after *arithmetic shift left*
  - For a 32-bit value,
    an ***n*-bit** *rotate shift left* is identical to a ***32 – n*** *rotate shift right*
    - Except the value of the carry bit will be shifted by one place.
  - 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

ARM's Flow Control Instructions (Unconditional Branch)

- **ARM**'s ***unconditional branch*** *instruction* has the form `B target`, where `target` denotes the *branch target address* which is *the address of the next instruction to be executed*.

- The following fragment of code demonstrates how the unconditional branch is used.

```
..    do this      ;Some code
..    then that    ;Some other code
      B  Next      ;Now skip past next instructions
..                 ;…the code being skipped
..                 ;…the code being skipped
Next ..            ;Target address for the branch
```

- In a high-level language, the unconditional branch is called a *goto*, which is considered a poor programming style;
- *Yet, in assembly, the unconditional branch is unavoidable*,
  - Assembly is a low-level language which ***does not*** have constructs such as *if …then.. else*, *while*, *repeat*, *for*, …

ARM's Flow Control Instructions (Conditional Branch)

- Consider the following if statement,
  IF (X == Y)
  THEN Y = Y + 1
  ELSE Y = Y + 2

- A test is performed and one of two courses of action is carried out depending on the outcome.
- We can translate this as:

```
      CMP r1,r2     ;Compare r1 and r2,
                    ;where r1 contains y and r2 contains x
      BNE Plus2     ;if not equal then branch to the else part
      ADD r1,r1,#1  ;if equal fall through to here
                    ;and add one to y
      B   leave     ;now skip past the else part
Plus2 ADD r1,r1,#2  ;ELSE part add 2 to y
leave  …            ;continue from here
```

ARM's Flow Control Instructions (Conditional Branch)

- The ***conditional branch*** *instruction*
  - tests the flag bits (*condition codes*) in the *current program status register* (***CPSR***), then
  - takes the branch if the tested condition is true.

- **ARM** dedicates 4 bits in each instruction to encode *16 different conditions* in total
  - ***eight*** possible conditional branches based on the state of a ***single bit***, namely Zero bit (Z), Negative bit (N), Carry bit (C), and oVerflow bit (V):
    - ***four*** that *branch on* ***true*** and
    - ***four*** that *branch on* ***false***.
  - ***six*** compound conditional branches
  - ***one*** always branch (unconditional)
  - ***one*** never branch (reserved)

ARM's Flow Control Instructions (Conditional Branch)

TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

Encoding	Mnemonic	Branch on Flag Status	Execute on condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear, and either N set and V set, or N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

ARM's Flow Control Instructions (Conditional Branch)

- **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)
    - **TEQ** *does* ***not*** *update the* ***overflow flag*** *or the* ***carry flag***
  - **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 complement 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

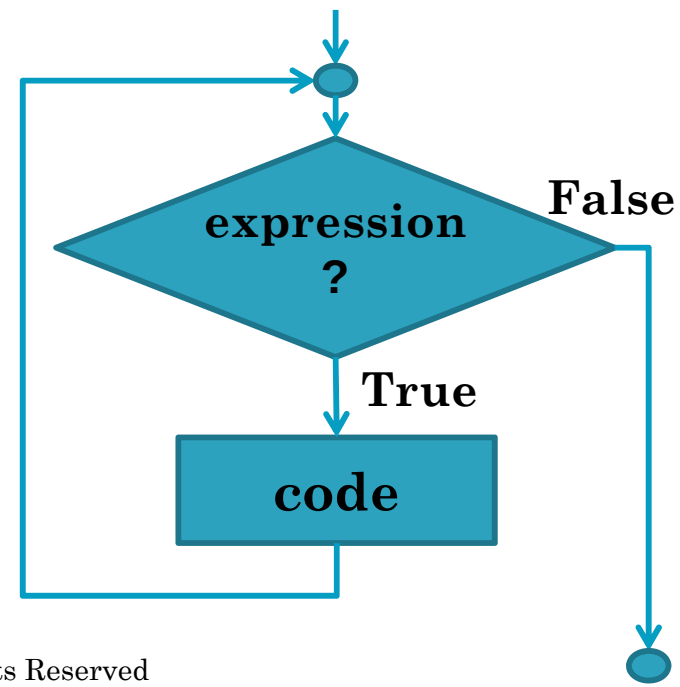
ARM's Flow Control Instructions (Branching and Loop Constructs)

- Nothing illustrates the concept of flow control better than the classic loop constructs that are at the core of so-called structured programming.
- The following demonstrate the structure of
  - The WHILE loop,
  - The UNTIL loop, and
  - The FOR loop

ARM's Flow Control Instructions (Branching and Loop Constructs)

The WHILE loop example

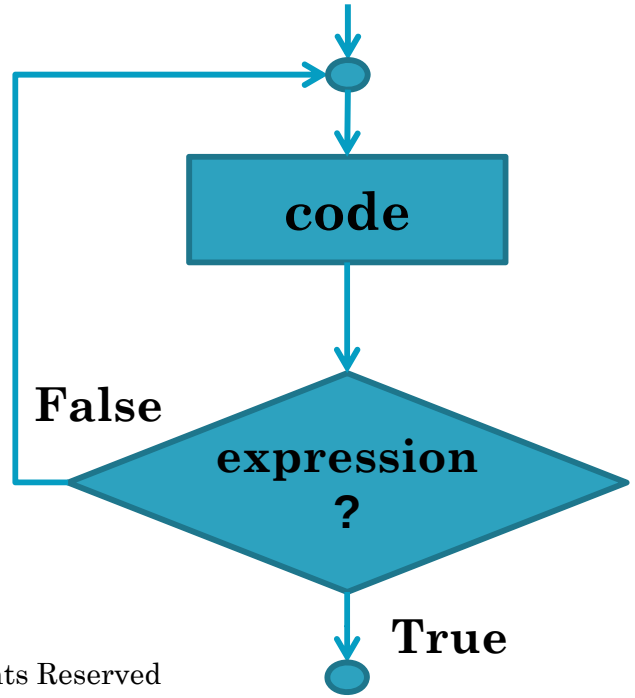
```
Loop        CMP   r0,#0       ;perform test at start of loop
            BEQ   WhileExit   ;exit on test true

            code  ...         ;body of the loop

            B     Loop        ;Repeat WHILE true
WhileExit   Post  loop ...    ;Exit
```



ARM's Flow Control Instructions (Branching and Loop Constructs)

The UNTIL loop example

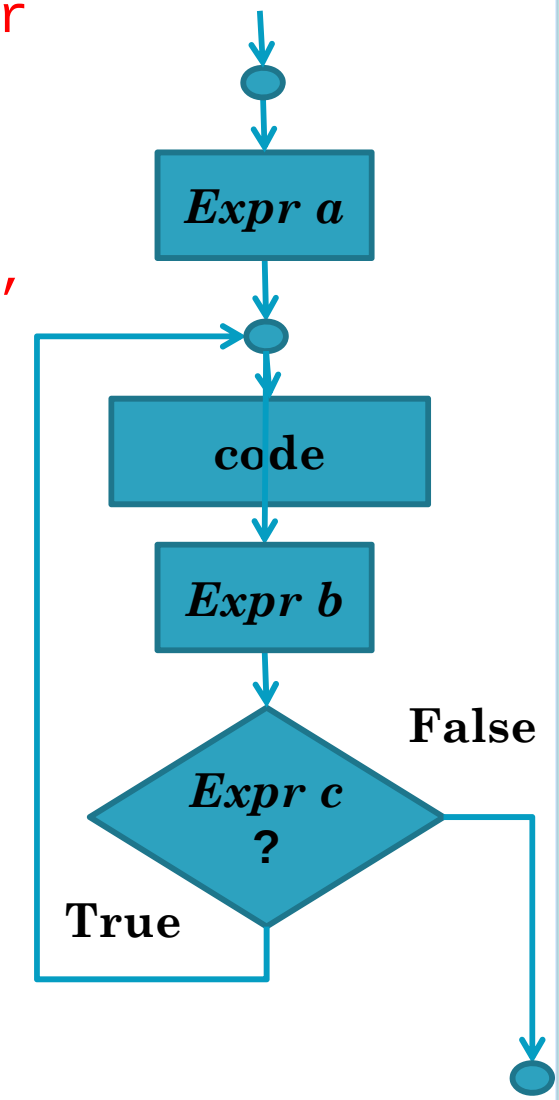
```
Loop        code  ...          ;body of the loop

            CMP   r0,#0        ;perform test at end of loop
            BNE   Loop         ;Repeat UNTIL true
WhileExit   Post  loop ...     ;Exit
```



ARM's Flow Control Instructions (Branching and Loop Constructs)

The FOR loop example

```
      MOV    r0,#10      ;set up the loop counter

Loop  code   ...         ;body of the loop

      SUBS   r0,r0,#1    ;decrement loop counter,
                         ;set flags
      BNE    Loop        ;continue until
                         ;count zero
      Post   loop ...    ;fall through on
                         ;zero count
```



This FOR loop is different than the C FOR loop

ARM's Flow Control Instructions (Branching and Loop Constructs)

The combination loop example

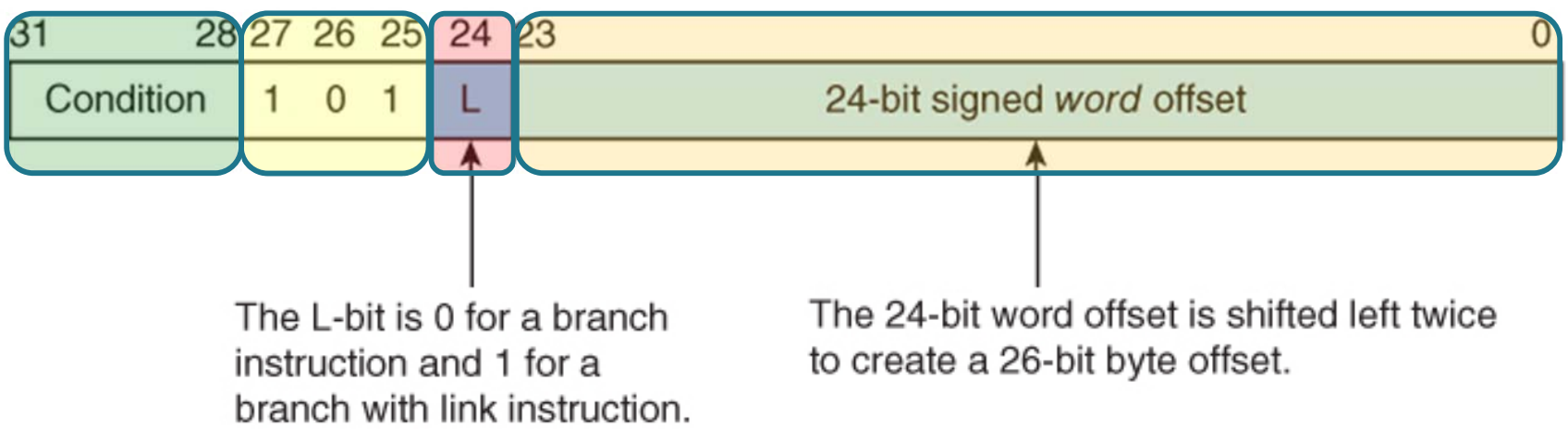
```
           MOV   r0,#10      ;set up the loop counter
LoopStart  CMP   r1,#0       ;perform test at start of loop
           BEQ   ComboExit   ;exit on test true

           code  ...         ;body of the loop

           CMP   r2,#0       ;perform test at end of loop
           BEQ   ComboExit   ;exit on test true

           SUBS  r0,r0,#1    ;decrement loop counter, set flags
           BNE   LoopStart   ;continue until count zero
ComboExit  Post  loop ...    ;Exit
```

Instruction Encoding
An Insight into the ARM's Architecture

- The branch instruction (Figure 3.41) has an 8-bit op-code with a 24-bit ***signed*** program counter relative ***offset*** (***word*** *address offset*).
- Converting the 24-bit offset to real address:
  - shift left twice the 24-bit offset to convert the word-offset address to a byte-offset address,
  - ***sign-extended*** to 32 bits,
  - added it to the current value of the program counter (***the result is in the range PC ± 32 MBytes***).

Do not forget the pipelining effect

FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions



Instruction Encoding

ARM Instruction: : **Loop B Loop**

Condition = 1110 (always – unconditional)

L = 0 (Not **BL**)

Byte-offset

Destination address

```
Difference = Loop address
           - (Current address
             + pipelining effect)
           = 0 - 8= -2_0000 1000
```

TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

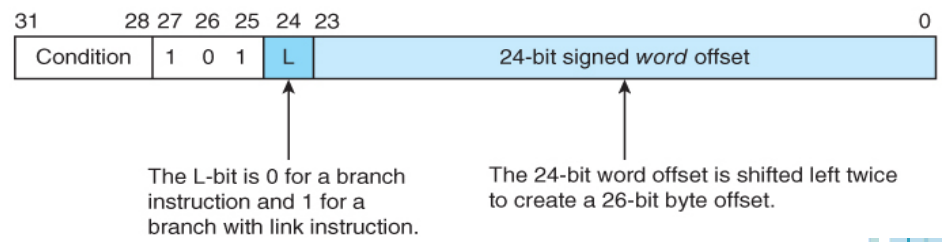
Encoding	Mnemonic	Branch on Flag Status	Execute on condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear, and either N set and V set, or N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

```
Difference >> 2 = -(2_0000 1000 >> 2)=-2_0000 0010
                =  2_1111 1110
                =  2_1111 1111 1111 1111 1111 1110
```

Word-offset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions



0xEAFFFFFE

Instruction Decoding

Machine Language Instruction: **0x1AFFFFFD**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1

Bits number 25 26 27 = 101

L = 0 (Not **BL**)

Condition = 0001 (BNE)

Destination address

Word offset= 0xFFFFFD = -3

Byte offset = -3 × 4 = -12

= **Loop** address

- (Current address

+ pipelining effect)

= **Loop** address

- Current address - 8

Hence, (**Loop** address - Current address) = -12 + 8 = -4

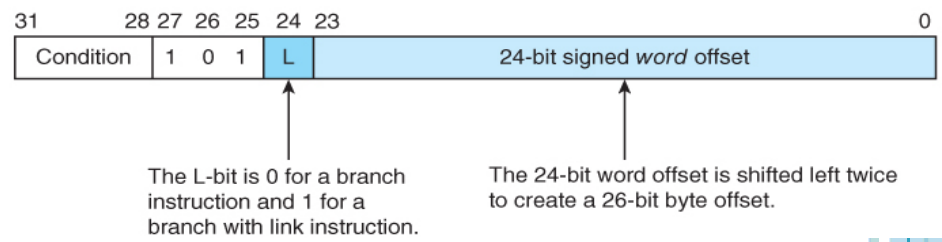
LOOP ....

BNE LOOP

TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

Encoding	Mnemonic	Branch on Flag Status	Execute on condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear, and either N set and V set, or N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

FIGURE 3.41 Encoding ARM's branch and branch-with-link instructions



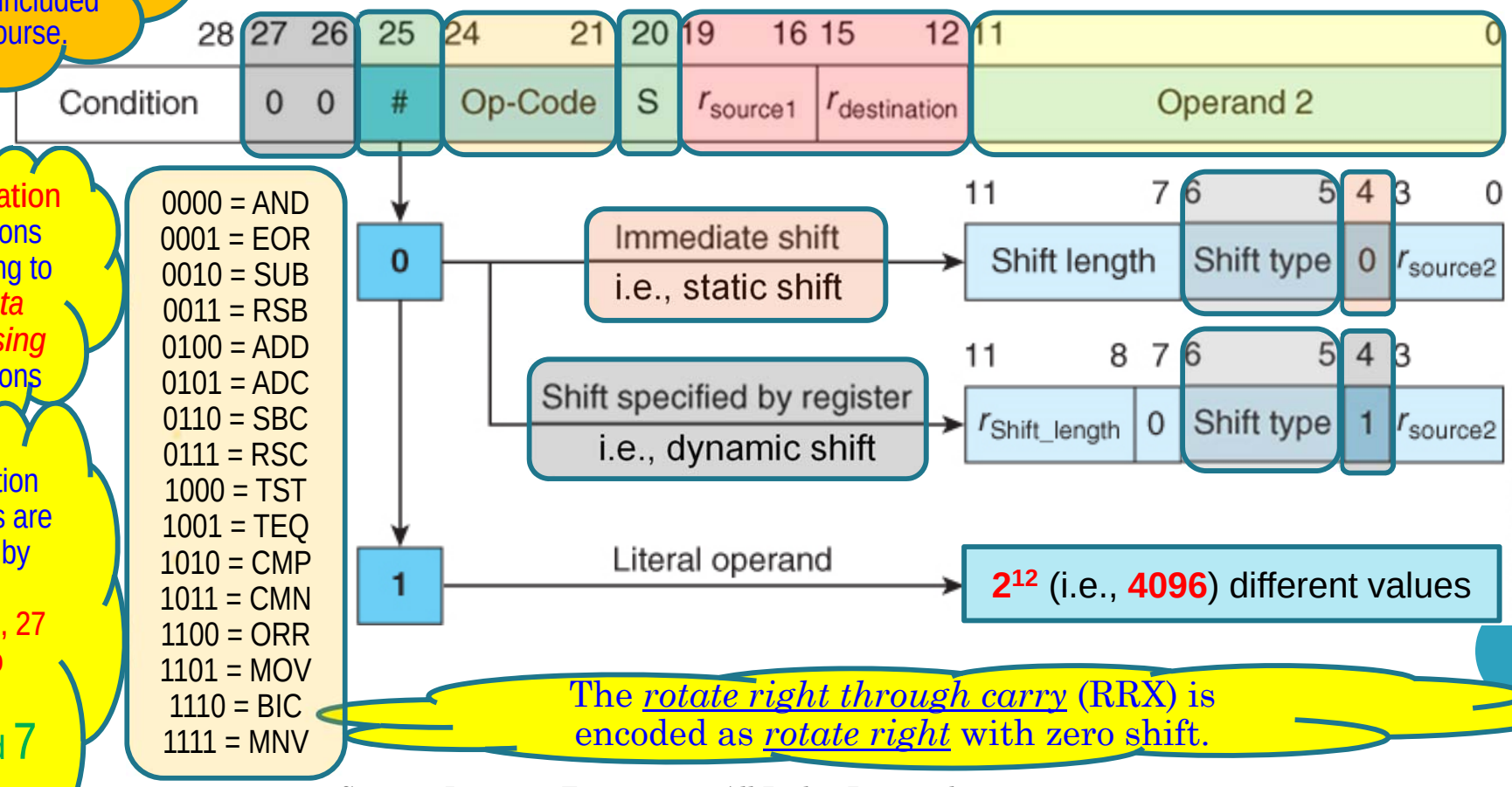
Instruction Encoding
An Insight into the ARM's Architecture

- Figure 3.26 illustrates the structure of the ARM's ***data processing*** instructions and demonstrates how bit 25 is used to control the nature of the second source operand.

Shift type
00 = logical left
01 = logical right
10 = Arithmetic right
11 = rotate right

To reduce the course load, Multiplication encoding/decoding will not be included in this course.

3.26 Encoding the ARM's data processing instructions



Multiplication instructions are belong to the *data processing* instructions

Multiplication instructions are encoded by setting bits 25, 26, 27 to zero and bits 4 and 7 to one.

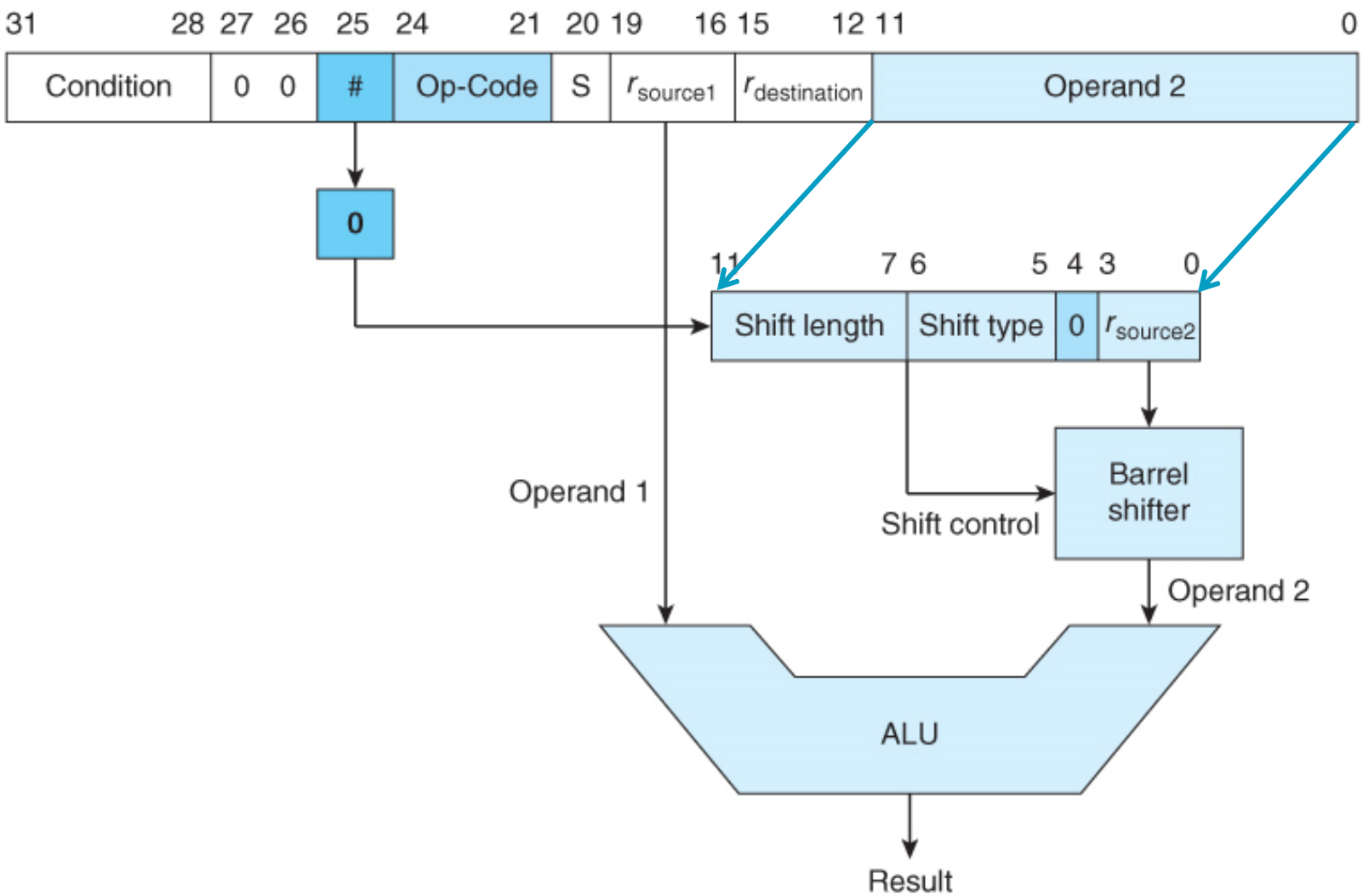
0000 = AND
0001 = EOR
0010 = SUB
0011 = RSB
0100 = ADD
0101 = ADC
0110 = SBC
0111 = RSC
1000 = TST
1001 = TEQ
1010 = CMP
1011 = CMN
1100 = ORR
1101 = MOV
1110 = BIC
1111 = MNV

The *rotate right through carry* (RRX) is encoded as *rotate right* with zero shift.

Implementing a Shift Operation on the ARM

- Figure 3.25 illustrates the structure of instructions with shifted operands and shows how the various fields control the shifter and the ALU.

FIGURE 3.25 ARM's barrel shifter



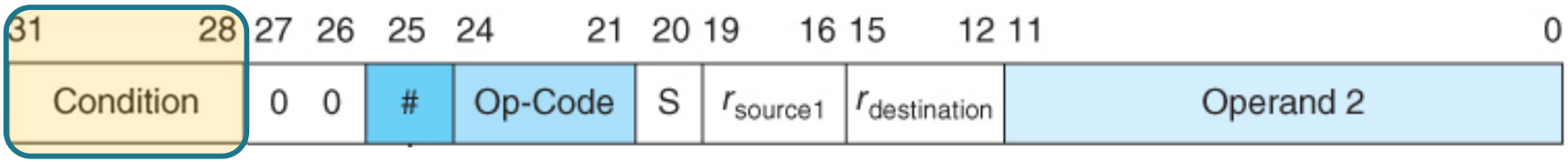
ARM's Flow Control Instructions (Conditional Execution)

- One of **ARM**'s most unusual features is that each instruction is *conditionally executed*
  - associating an instruction with a logical condition.
    - If the stated condition is true, the instruction is executed.
    - Otherwise it is bypassed (*squashed*).
- Assembly language programmers indicate the conditional execution mode by appending the appropriate condition to a mnemonic (***mnemonic*** *is a text abbreviation for an operation code*).
- for example,

```
ADDEQ  r1,r2,r3  ;IF Z = 1 THEN [r1] <- [r2] + [r3]
```

specifies that the addition is performed only if the Z-bit is set because a previous result was zero.

FIGURE 3.26 Encoding the ARM's data processing instructions



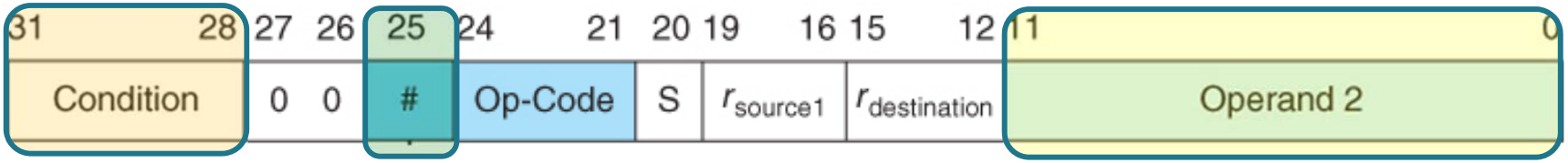
ARM's Flow Control Instructions (Conditional Execution)

- There is nothing to stop you combining *conditional execution* and *shifting* because the branch and shift fields of an instruction are independent.

- You can write

```
ADDCC r1,r2,r3,LSL r4   ;IF C=0 THEN [r1]<-[r2] + [r3]×2^[r4]
```

FIGURE 3.26 Encoding the ARM's data processing instructions



ARM's Flow Control Instructions (Conditional Execution)

- ARM's conditional execution mode makes it easy to implement conditional operations in a high-level language.

- Consider the following fragment of C code.
  if (P == Q) X = P – Y ;

- If we assume that r1 contains P,
  r2 contains Q,
  r3 contains X, and
  r4 contains Y, then we can write

```
CMP     r1,r2          ;compare P == Q
SUBEQ   r3,r1,r4       ;if (P == Q) then r3 = r1 - r4
```

- Notice how simple this operation is implemented without using a branch instruction
  - In this case the subtraction is squashed if the comparison is false

ARM's Flow Control Instructions (Conditional Execution)

- Now consider a more complicated example of a C construct with a compound predicate:
  if ((a == b) && (c == d)) e++;

- We can write

```
CMP     r0,r1     ;compare a == b
CMPEQ   r2,r3     ;if a == b then test c == d
ADDEQ   r4,r4,#1  ;if a == b AND c == d THEN increment e
```

- The first line, `CMP r0,r1`, compares a and b.

- The next line, `CMPEQ r2,r3`, executes a conditional comparison only if the result of the first line was true (i.e., a == b). (***short circuit***)

- The third line, `ADDEQ r4,r4,#1`, is executed only if the previous line was true (i.e., c == d) to implement the e++.

ARM's Flow Control Instructions (Conditional Execution)

- You can also handle some testing with multiple conditions.
- Consider:
  if (a == b) e = e + 4;
  if (a < b) e = e + 7;
  if (a > b) e = e + 12;

We can use conditional execution to implement this as

```
CMP    r0,r1                    ;compare a == b
ADDEQ r4,r4,#4   ;if a == b then e = e +  4
ADDLT r4,r4,#7   ;if a < b  then e = e +  7
ADDGT r4,r4,#12 ;if a > b  then e = e + 12
```

- Without the conditional execution, we can implement it as follow:

```
          CMP r0,r1          ;compare a == b
          BNE NotEqual
Equal     ADD r4,r4,#4       ;if a == b then e = e + 4
          B   AfterAll
NotEqual  BLT LessThan
          ADD r4,r4,#12      ;if a > b  then e = e + 12
          B   AfterAll
LessThan  ADD r4,r4,#7       ;if a < b  then e = e + 7
AfterAll  …
```

Instruction Encoding

ARM Instruction: `ADD r0,r1,r2`

- `Condition = 1110` (always – unconditional)
- `Op-Code = 0100` (i.e., ADD)
- `S = 0` (not ADDS)
- $r_{destination}$ = `0000` (destination *operand*)
- $r_{source1}$ = `0001` (first *operand*)
- `# = 0` (second operand not a constant)
- `Operand 2`
  - $r_{source2}$ = `0010`
  - `shift type = 00` (logical left)
  - `shift length = 00000`

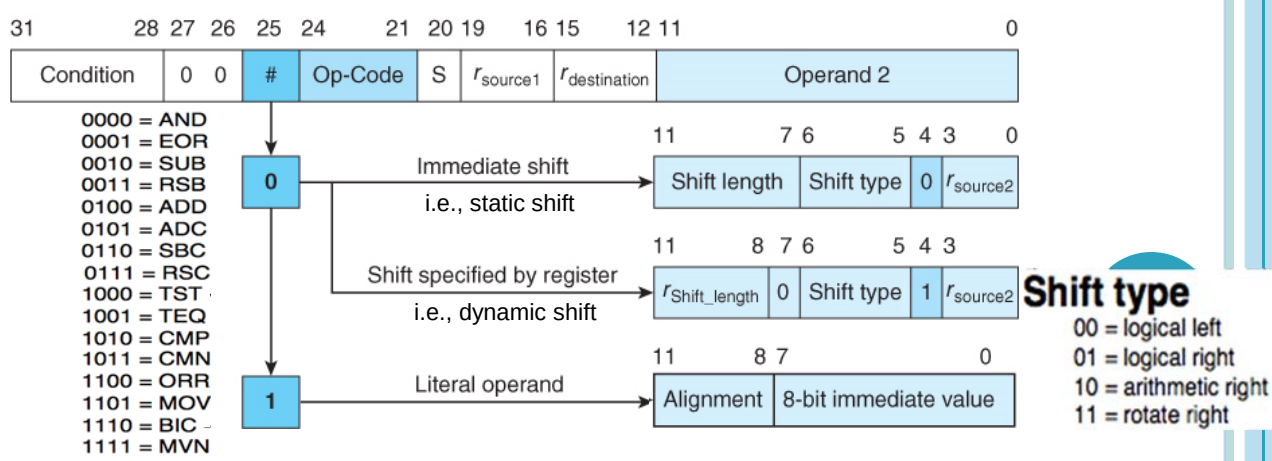
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

0xE0810002

TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

Encoding	Mnemonic	Branch on Flag Status	Execute on condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear, and either N set and V set, or N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

FIGURE 3.26 Encoding the ARM's data processing instructions



Instruction Decoding

Machine Language Instruction: **0x004A0182**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0

Bits number 26 27 = 00

- Op-Code = 0010 (i.e., SUB)
- Condition = 0000 (EQ)
- S = 0 (not SUBEQS)
- $r_{destination}$ = 0000 ➔ r0
- $r_{source1}$ = 1010 ➔ r10
- # = 0 (second operand not a constant)
- Operand 2 (bit number 4 = 0)
  - $r_{source2}$ = 0010 ➔ r2
  - shift type = 00 ➔ logical left
  - shift length = 00011 ➔ 3

TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

Encoding	Mnemonic	Branch on Flag Status	Execute on condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear, and either N set and V set, or N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

SUBEQ **r0**,r10,r2,LSL#3

FIGURE 3.26 Encoding the ARM's data processing instructions