Instruction Encoding

ARM Instruction: `CMPGT r3,r5`

Condition = 1100 (GT)

Op-Code = 1010 (i.e., CMP)

S = 1 (update flags)

$r_{destination}$ = 0000 (must be zeros)

$r_{source1}$ = 0011 (first *operand*)

# = 0 (second operand not a constant)

Operand 2

$r_{source2}$ = 0101

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	09	08	07	06	05	04	03	02	01	00
1	1	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

0xC1530005

In all *test-and-compare instructions, i.e., TST, TEQ, CMP, and CMN, the destination register field MUST BE encoded as 0000* and ***S = 1***

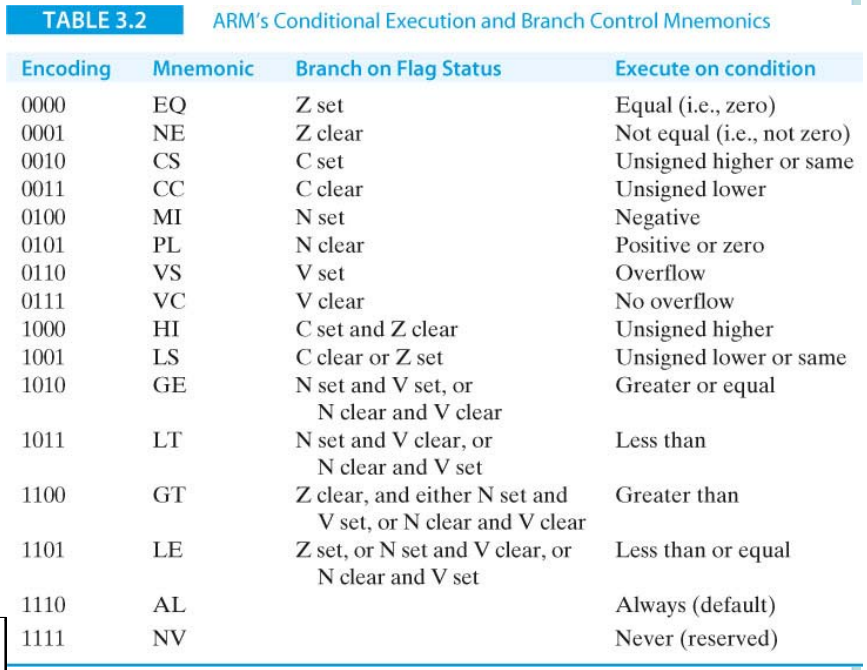
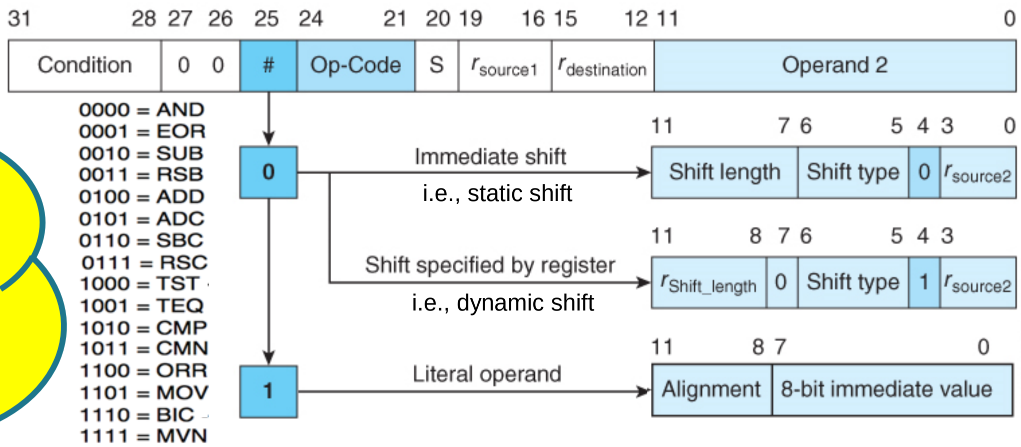


TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

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



Shift type

00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

Instruction Encoding

ARM Instruction: MOV **PC**, LR

Condition = 1110 (always – unconditional)

Op-Code = 1101 (i.e., MOV)

S = 0 (not MOVS)

$r_{destination}$ = 1111 (PC)

$r_{source1}$ = 0000 (must be zeros)

# = 0 (second operand not a constant)

Operand 2

$r_{source2}$ = 1110

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	09	08	07	06	05	04	03	02	01	00
1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0

0xE1A0F00E

In all moving instructions, i.e., MOV, and MVN, the source$_1$ register field MUST BE encoded as 0000

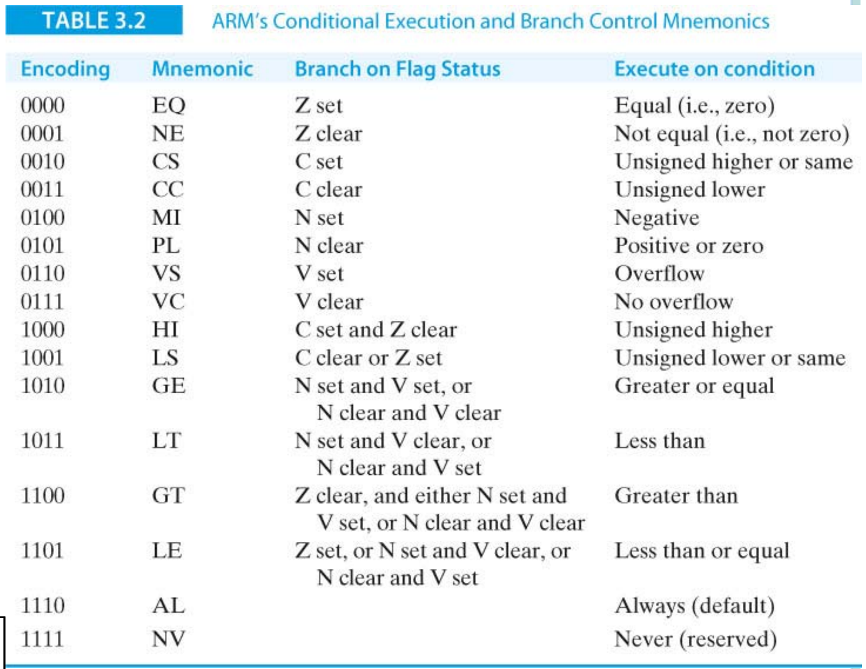


TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

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)

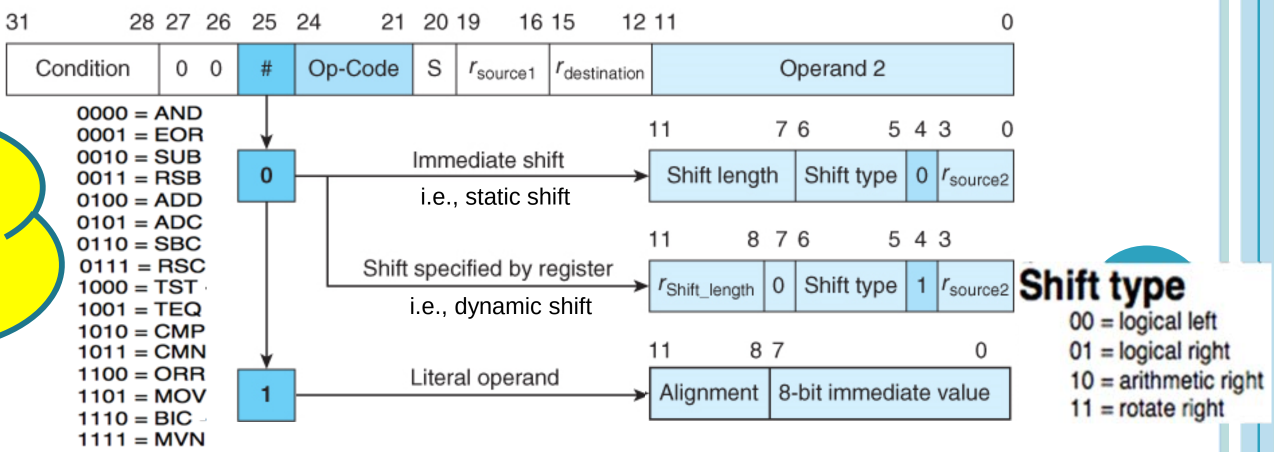


FIGURE 3.26 Encoding the ARM's data processing instructions

Instruction Decoding

Machine Language Instruction: **0xE1B01061**

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	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1

Bits number 26 27 = 00

Op-Code = 1101 (i.e., MOV)

Condition = 1110 (Always)

S = 1 (i.e., MOVS)

$r_{destination}$ = 0001 → r1

$r_{source1}$ = 0000 (must be zeros)

# = 0 (second operand not a constant)

Operand 2 (bit number 4 = 0)

$r_{source2}$ = 0001 → r1

shift type = 11 → rotate right

shift length = 00000 → **00**

rotate right and

shift length =00 →RRX

MOVS **r1**,r1,RRX

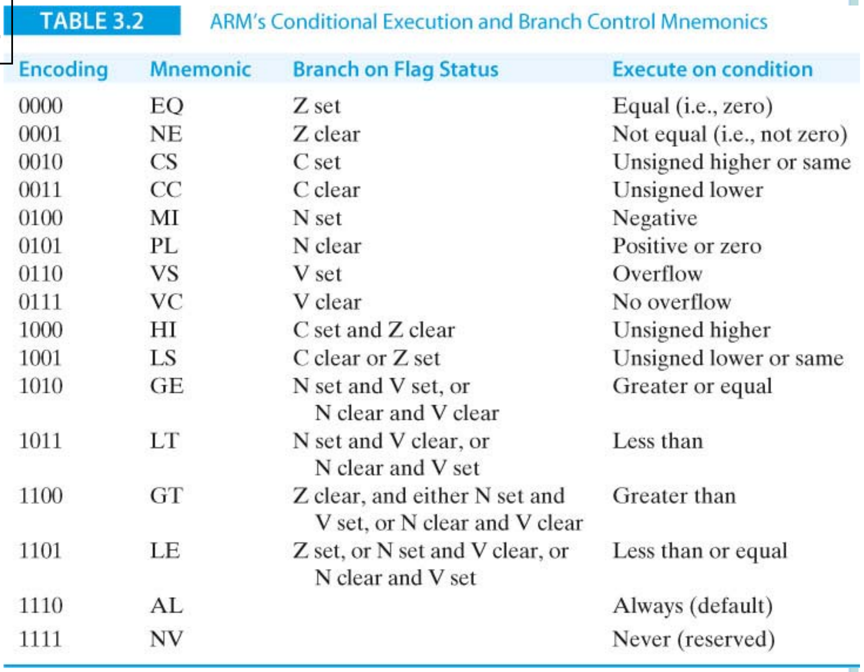
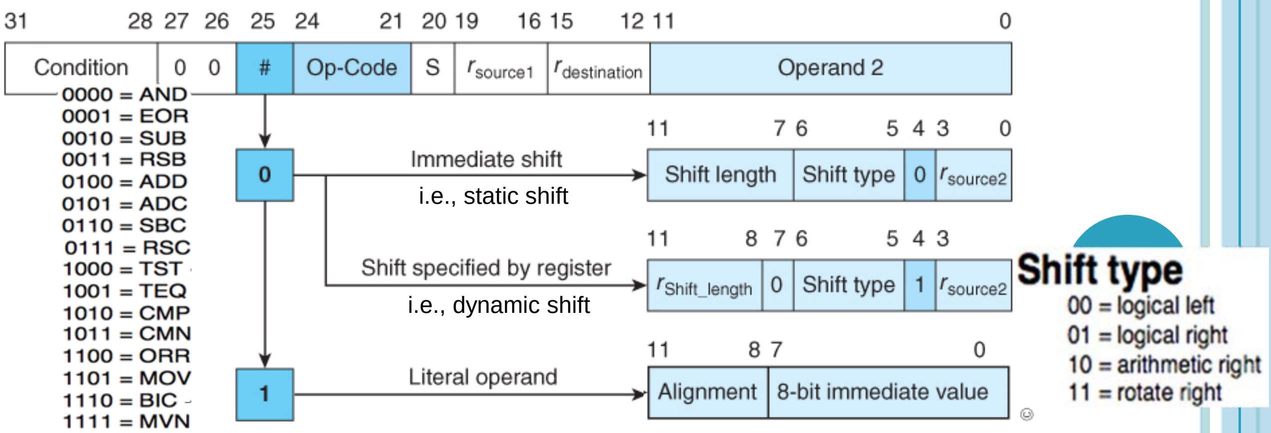


TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

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



Handling Literals

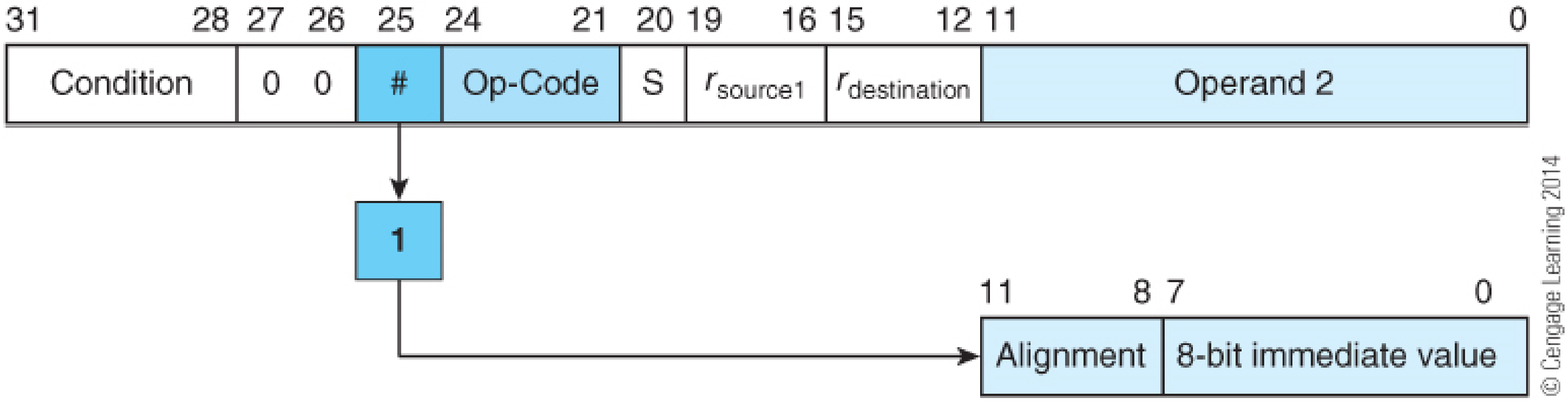
- In **ARM**, ***operand 2*** can is a literal.

```
ADD r0,r1,#7      ;adds 7 to r1 and puts the result in r0.
MOV r3,#25        ;moves 25 into r3.
```

- ***What is the range of such literals?***
  - ***Operand 2*** is a 12-bits field, i.e., it can encode **4096** different values
    - **ARM** encodes these12-bits as a value from 0 to 255 (i.e., 8-bits) to be rotated (aligned) according to the value of the other bits (i.e., 4-bits)

- Figure 3.28 illustrate the format of **ARM**'s instructions with a literal operand.

FIGURE 3.28 Diagram of ARM's literal operand encoding



Handling Literals

Replace the word “shift” in this slide by “rotate”. The book is wrong!!

Zero rotation

4 rotations left

10 rotations left

24 rotations left

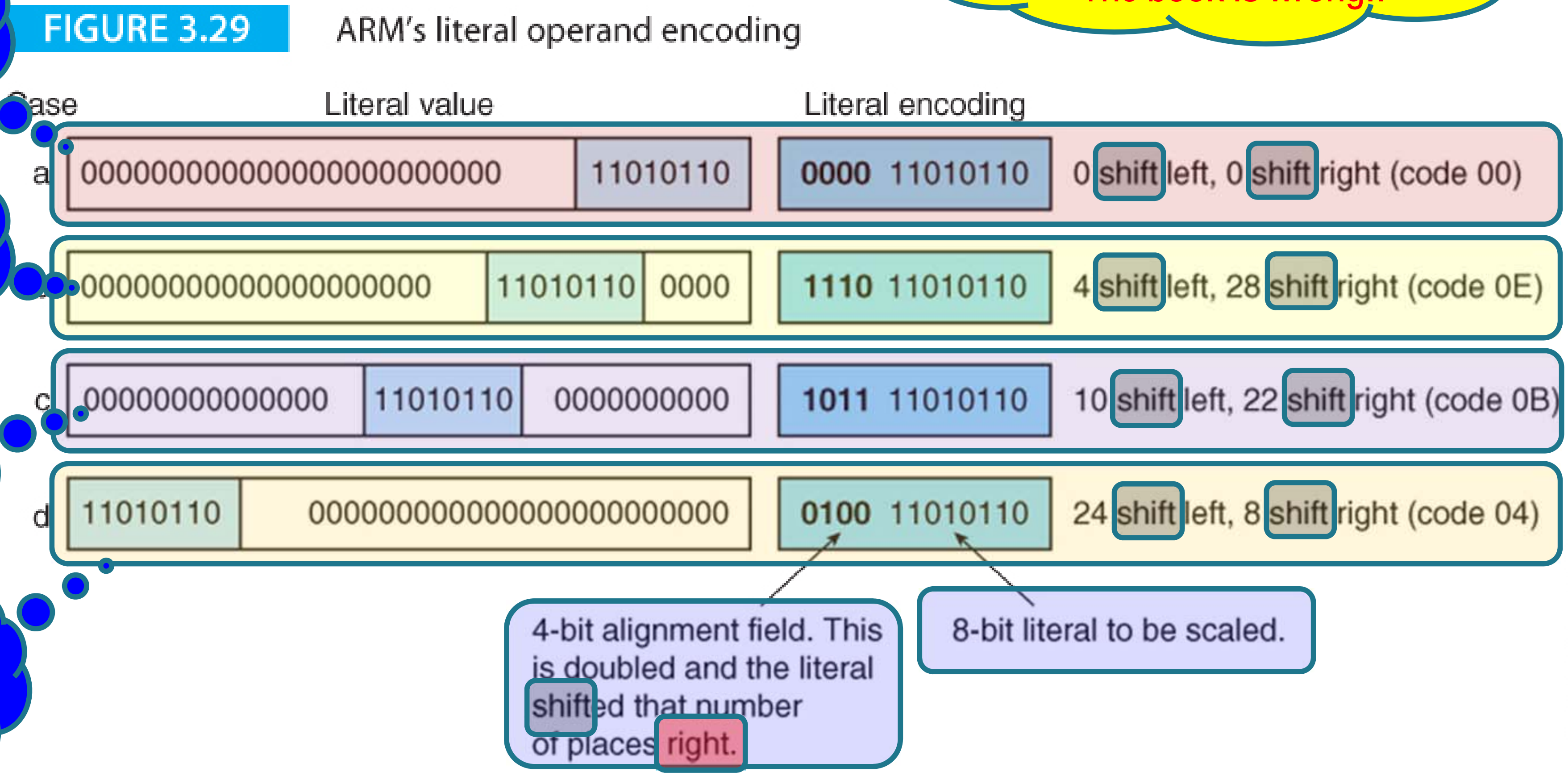


FIGURE 3.29 ARM’s literal operand encoding

Case	Literal value			Literal encoding	
a	000000000000000000000000	11010110		0000 11010110	0 shift left, 0 shift right (code 00)
	00000000000000000000	11010110	0000	1110 11010110	4 shift left, 28 shift right (code 0E)
c	00000000000000	11010110	0000000000	1011 11010110	10 shift left, 22 shift right (code 0B)
d	11010110	000000000000000000000000		0100 11010110	24 shift left, 8 shift right (code 04)

4-bit alignment field. This is doubled and the literal shifted that number of places right.

8-bit literal to be scaled.

© Cengage Learning 2014

The number of actual rotations is ALWAYS even (i.e., can not be odd at all).

Also called the 4 bits *alignment*

Handling Literals

You need to know how to *decode* and encode literals

Encoded literal	Scale value	#of rotations right =2 × Scale value	# of rotations left =32 - 2 × Scale value	Decoded literal
0000 mnop wxyz	0	0	$(32)_{10}$	0000 0000 0000 0000 0000 0000 mnop wxyz
1111 mnop wxyz	$(15)_{10}$	$(30)_{10}$	2	0000 0000 0000 0000 0000 00mn opwx yz00
1110 mnop wxyz	$(14)_{10}$	$(28)_{10}$	4	0000 0000 0000 0000 0000 mnop wxyz 0000
1101 mnop wxyz	$(13)_{10}$	$(26)_{10}$	6	0000 0000 0000 0000 00mn opwx yz00 0000
1100 mnop wxyz	$(12)_{10}$	$(24)_{10}$	8	0000 0000 0000 0000 mnop wxyz 0000 0000
1011 mnop wxyz	$(11)_{10}$	$(22)_{10}$	$(10)_{10}$	0000 0000 0000 00mn opwx yz00 0000 0000
1010 mnop wxyz	$(10)_{10}$	$(20)_{10}$	$(12)_{10}$	0000 0000 0000 mnop wxyz 0000 0000 0000
1001 mnop wxyz	9	$(18)_{10}$	$(14)_{10}$	0000 0000 00mn opwx yz00 0000 0000 0000
1000 mnop wxyz	8	$(16)_{10}$	$(16)_{10}$	0000 0000 mnop wxyz 0000 0000 0000 0000
0111 mnop wxyz	7	$(14)_{10}$	$(18)_{10}$	0000 00mn opwx yz00 0000 0000 0000 0000
0110 mnop wxyz	6	$(12)_{10}$	$(20)_{10}$	0000 mnop wxyz 0000 0000 0000 0000 0000
0101 mnop wxyz	5	$(10)_{10}$	$(22)_{10}$	00mn opwx yz00 0000 0000 0000 0000 0000
0100 mnop wxyz	4	8	$(24)_{10}$	mnop wxyz 0000 0000 0000 0000 0000 0000
0011 mnop wxyz	3	6	$(26)_{10}$	opwx yz00 0000 0000 0000 0000 0000 00mn
0010 mnop wxyz	2	4	$(28)_{10}$	wxyz 0000 0000 0000 0000 0000 0000 mnop
0001 mnop wxyz	1	2	$(30)_{10}$	yz00 0000 0000 0000 0000 0000 00mn opwx

Handling Literals

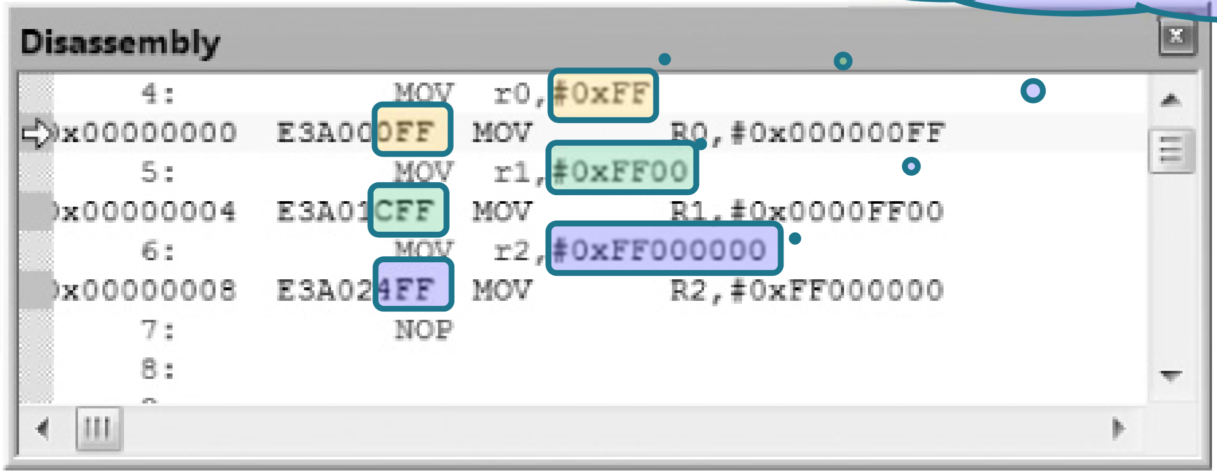
8-bit immediate value =0xFF
and the ZERO rotation left

8-bit immediate value =0xFF
and the 8 rotations left

8-bit immediate value =0xFF
and the 24 rotations left

FIGURE 3.30 Example of ARM literal encoding

If the value can not be represented that way, you will get an error message saying:
“cannot be represented by 0-255 and a rotation.”

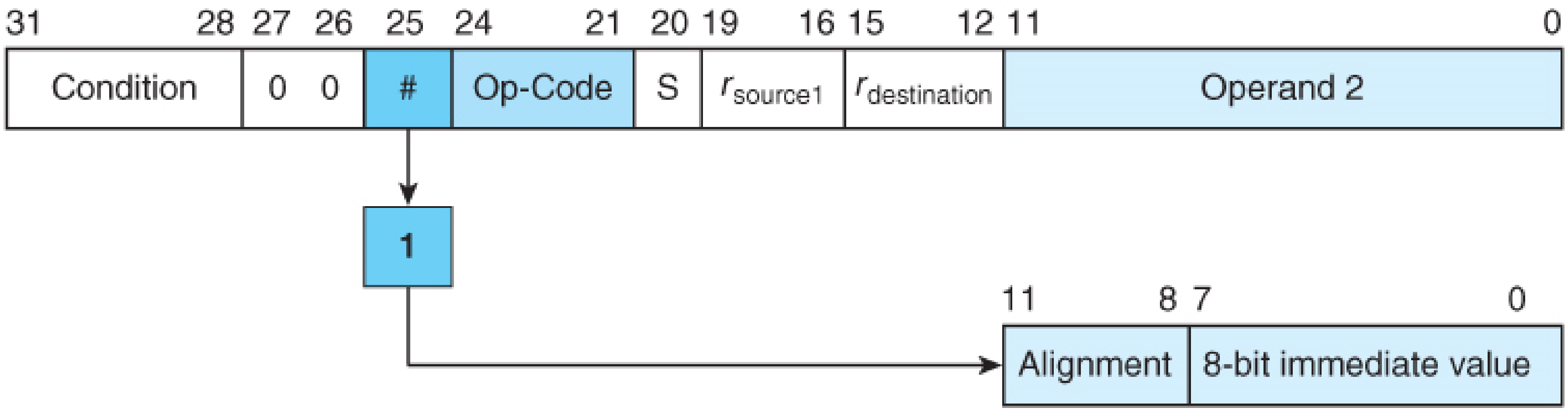


Disassembly

```
      4:                     MOV    r0,#0xFF
0x00000000   E3A000FF   MOV             R0,#0x000000FF
      5:                     MOV    r1,#0xFF00
0x00000004   E3A01CFF   MOV             R1,#0x0000FF00
      6:                     MOV    r2,#0xFF000000
0x00000008   E3A024FF   MOV             R2,#0xFF000000
      7:                     NOP
      8:
```

ARM® Software

FIGURE 3.28 Diagram of ARM’s literal operand encoding



© Cengage Learning 2014

Instruction Encoding

ARM Instruction: `ORRGTS` **`r1`**`,r2,#0xAA00`

Condition = 1100 (Greater than)

Op-Code = 1100 (i.e., ORR)

S = 1 (ORRGTS)

$r_{destination}$ = 0001 (destination *operand*)

$r_{source1}$ = 0010 (first *operand*)

# = 1 (second operand is a constant)

Operand 2 (*to be 0-255 and a rotation*)

8-bit immediate value = 0xAA

rotations left = 8

equivalent to 24 rotations right

Half of the rotations right = 12

i.e., 1100 in binary

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	0	0	0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1	0	1	0	1	0	1	0

0xC3921CAA

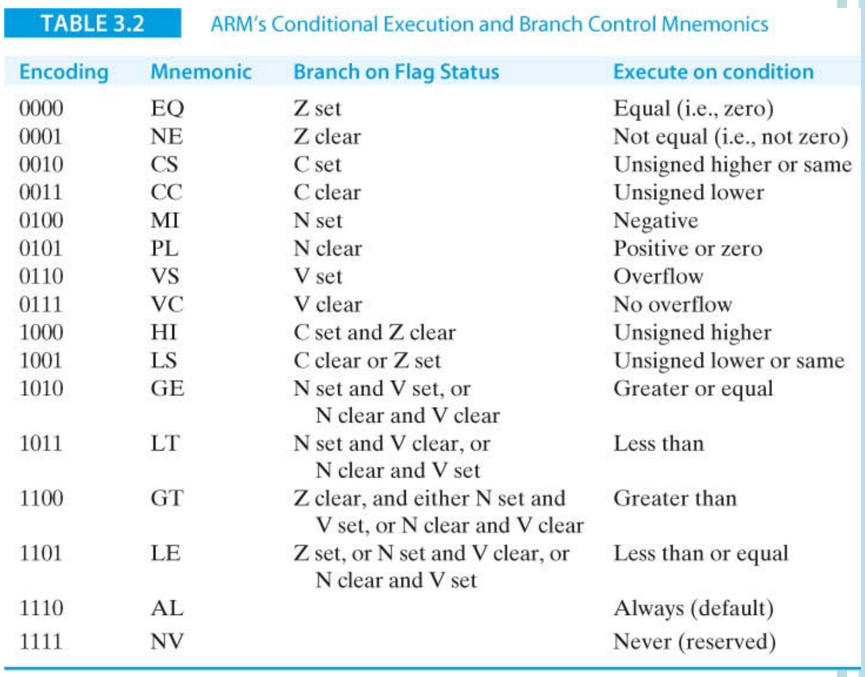
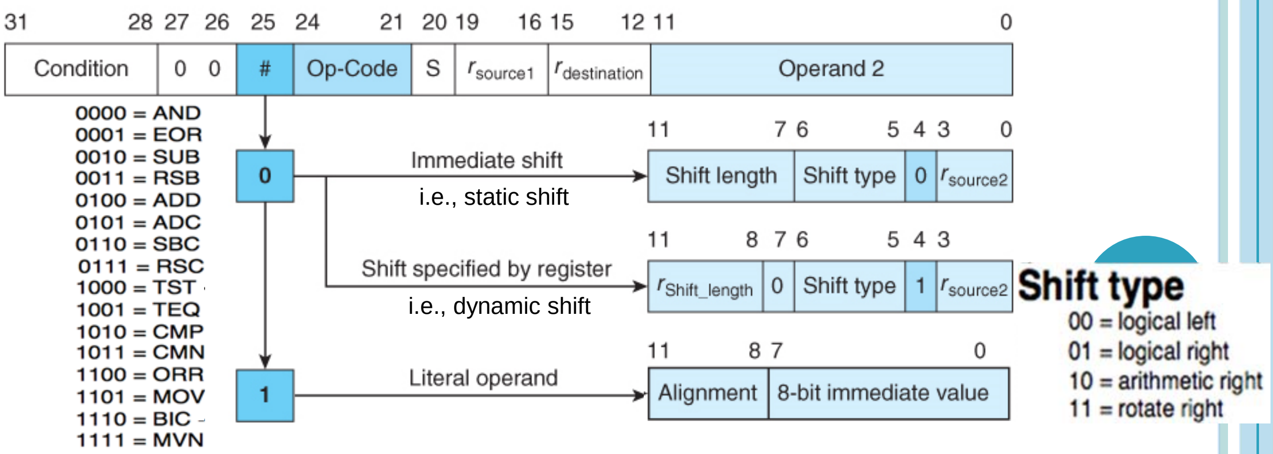


TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

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: **0x42742F55**

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	1	0	0	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	0	1	0	1	0	1	0	1

Bits number 26 27 = 00

Op-Code = 0011 (i.e., RSB)

Condition = 0100 (MI)

S = 1 (RSBMIS)

$r_{destination}$ = 0010 ➔ r2

$r_{source1}$ = 0100 ➔ r4

# = 1 (second operand is a constant)

Operand 2 (*to be 0-255 and a rotation*)

8-bit immediate value = 0x55

Alignment = 0xF, i.e.,

Half of the rotations right=0xF

rotations right =30

rotations left = 2

0x55 = 2_0101 0101

0x55 << 2

= 2_0101010100

= 0x154

RSBMIS **r2**,r4,#0x154

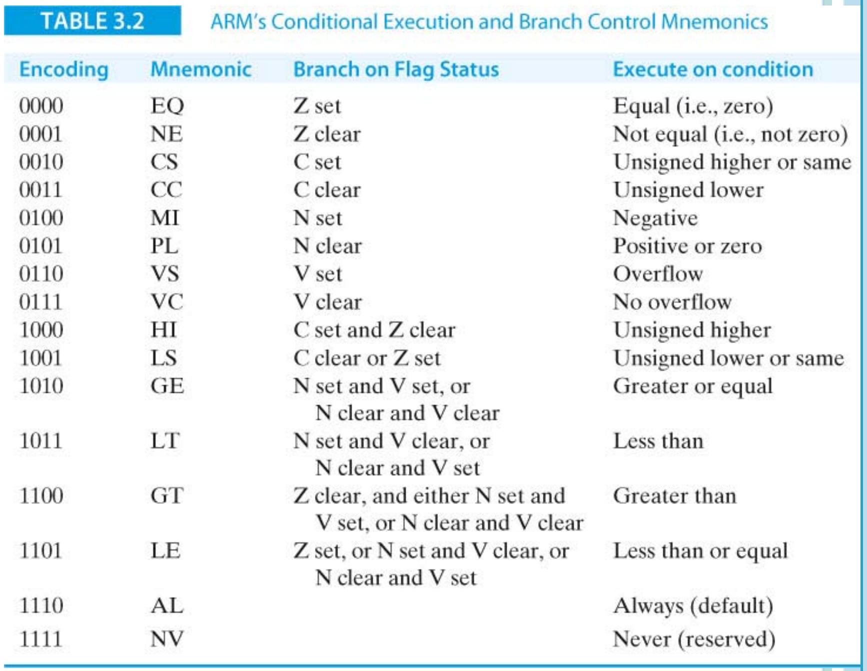
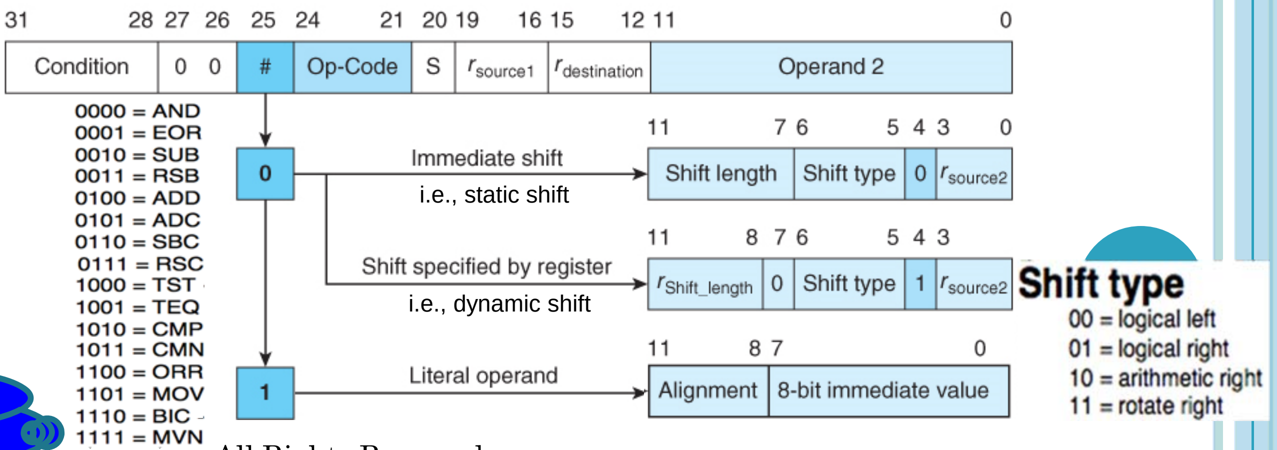


TABLE 3.2 ARM's Conditional Execution and Branch Control Mnemonics

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



As a practice, try to encode this instruction by yourself

Addressing Modes

Note that, the ARM assembly language and the RTL language have two different interpretations to the square brackets.

Instruction	*RTL form*	*Description*
`ADD` **`r0`**`,r1,#Q`	[r0] ← [r1] + Q	***Literal***: Add the integer Q to the content of register r1 and store the result in r0
`LDR` **`r0`**`,Mem`	[r0] ← [Mem]	***direct*** (i.e., ***absolute***) : Load the content of memory location Mem into register r0. *This addressing mode is* ***not supported by ARM*** *but is* ***supported by all CISC processors***
`LDR` **`r0`**`,[r1]`	[r0] ← [[r1]]	***Register Indirect***: Load r0 with the content of the memory location pointed at by r1

- The **ARM** lacks a simple memory *direct* (i.e., *absolute*) addressing mode (i.e., ***does not*** have an `LDR` **`r0`**`,address` instruction that implements direct addressing to load the contents of a memory location denoted by `address` into a register.)

Register Indirect Addressing

- In ***register indirect addressing***, the memory location of an operand is given by the contents of a register.

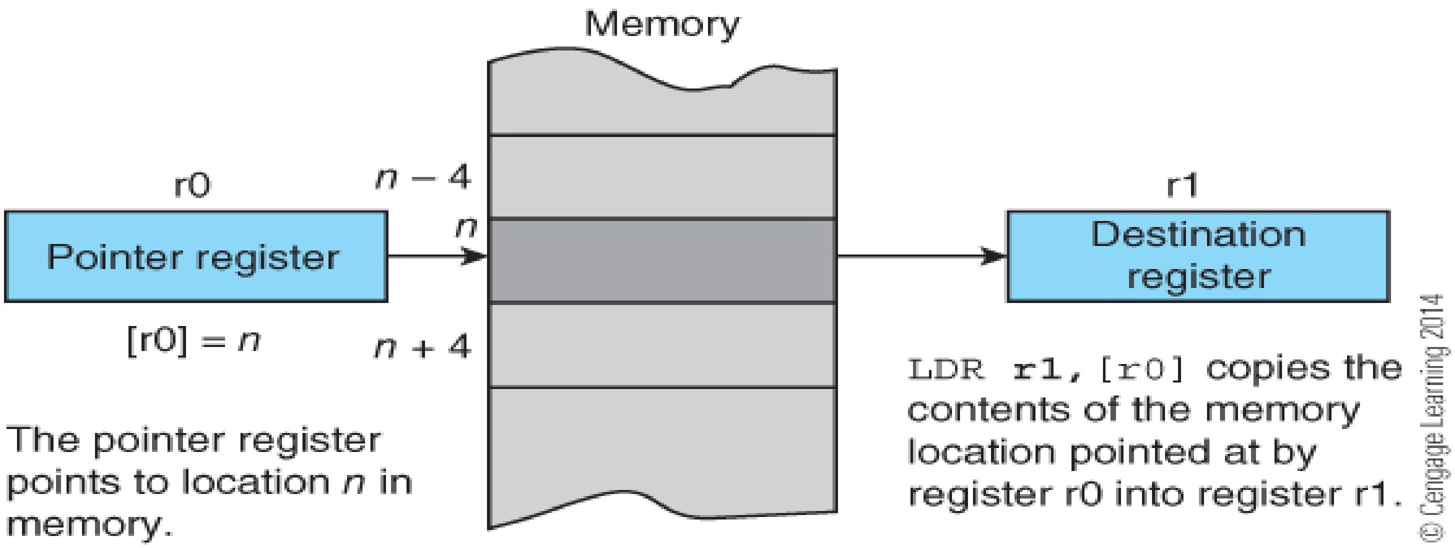
- All computers support some form of register indirect addressing.
- This is also called:
  - ***Indexed***
  - ***Pointer-based***

Register Indirect Addressing

- In ARM, the register indirect addressing is indicated by means of ***square brackets***; for example,

```
LDR r1,[r0]           ;[r0] ← [[r1]]
                      ;Load r1 with the content of
                      ;the memory location pointed at by r0
```

FIGURE 3.31 Register indirect addressing



The pointer register points to location *n* in memory.

LDR r1,[r0] copies the contents of the memory location pointed at by register r0 into register r1.

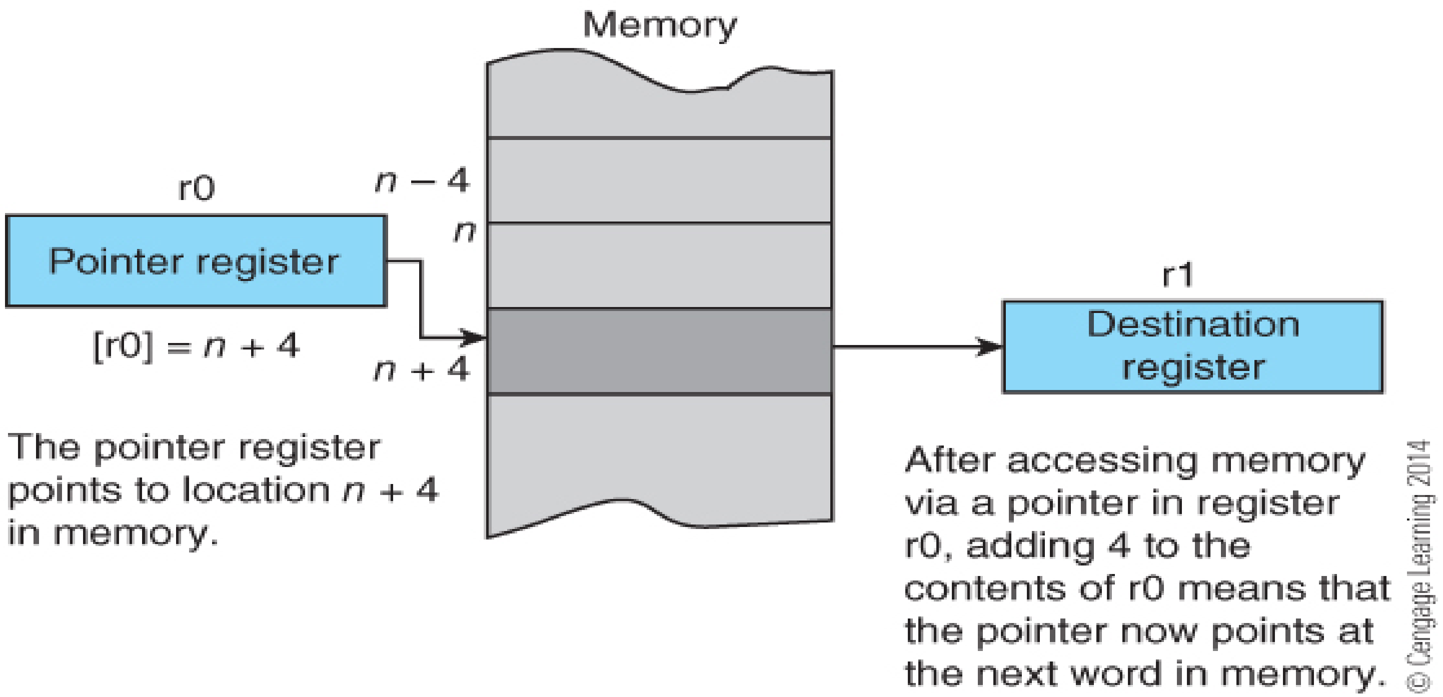
Register Indirect Addressing

- ❑ Consider what happens if we next execute

```
ADD r0,r0,#4 ;[r0] ← [r0] + 4
                   ;Add 4 to the contents of register r0
                   ;i.e., increment the pointer by one word
```

- ❑ Figure 3.32 demonstrates the effect of incrementing the pointer register. It now points to the next location in memory.
- ❑ This allows us to use the same instruction (`LDR r1,[r0]`) to access a sequence of memory locations; for example, a list, matrix, vector, array, or table.

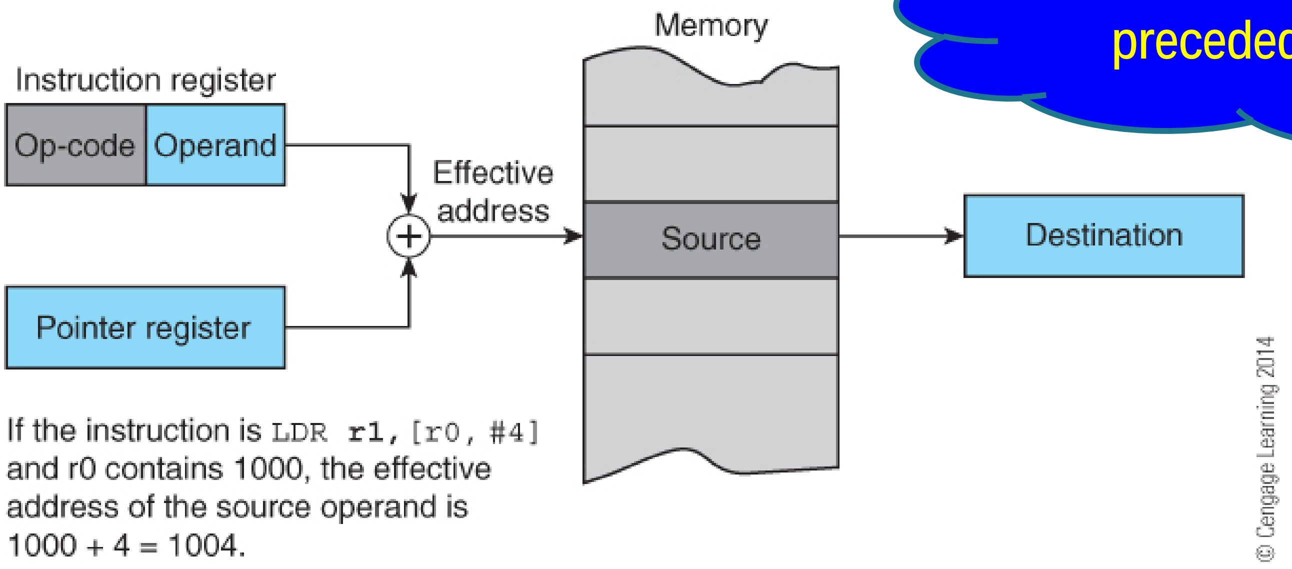
FIGURE 3.32 Effect of incrementing the pointer register


Memory
r0
Pointer register
[r0] = n + 4
n − 4
n
n + 4
r1
Destination register
The pointer register points to location n + 4 in memory.
After accessing memory via a pointer in register r0, adding 4 to the contents of r0 means that the pointer now points at the next word in memory.
© Cengage Learning 2014

Register Indirect Addressing with an Offset

- □ **ARM** supports a memory-addressing mode where the *effective address* of an operand is **computed by adding** the *contents of a register* to a *literal offset* encoded into the load/store instruction.
- □ This addressing mode is often called ***base plus displacement addressing***.
- □ Figure 3.33 illustrates the instruction `LDR` **`r0`**`, [r1,#4]`. The effective address is the sum of the content of the pointer register `r1` plus offset 4; that is, the operand is 4 bytes after the address specified by the pointer.
- □ In ***base plus displacement addressing*** mode,
  - ○ *the literal offset is a* ***true*** *12-bit literal (0—4095),* ***not*** *0—255 and a rotation as in the literals.*

FIGURE 3.33 Register indirect addressing with an offset



If the instruction is `LDR r1, [r0, #4]` and r0 contains 1000, the effective address of the source operand is 1000 + 4 = 1004.

The literal offset must be preceded by "#" sign

Register Indirect Addressing with an Offset

- The following fragment of code demonstrates the use of offsets to implement array access.
- *Because the offset is a constant, it cannot be changed at runtime.*

```
Sun    EQU  0          ;offsets for days of the week
Mon    EQU  4
Tue    EQU  8
Wed    EQU 12
Thu    EQU 16
Fri    EQU 20
Sat    EQU 24

       ADR r0, Week       ;r0 points to array Week
       LDR r2,[r0,#Tue]   ;Load the data for Tuesday into r2
       LDR r3,[r0,#Wed]   ;Load the data for Wednesday day into r2
       ADD r4,r2,r3       ;Add Tuesday  and Wednesday
       STR r4,[r0,#Mon]   ;Store the result in Monday

Week   DCD 0x11111111  ;data for day 1 (Sunday)
       DCD 0x22222222  ;data for day 2 (Monday)
       DCD 0x33333333  ;data for day 3 (Tuesday)
       DCD 0x44444444  ;data for day 4 (Wednesday)
       DCD 0x55555555  ;data for day 5 (Thursday)
       DCD 0x66666666  ;data for day 6 (Friday)
       DCD 0x77777777  ;data for day 7 (Saturday)
```

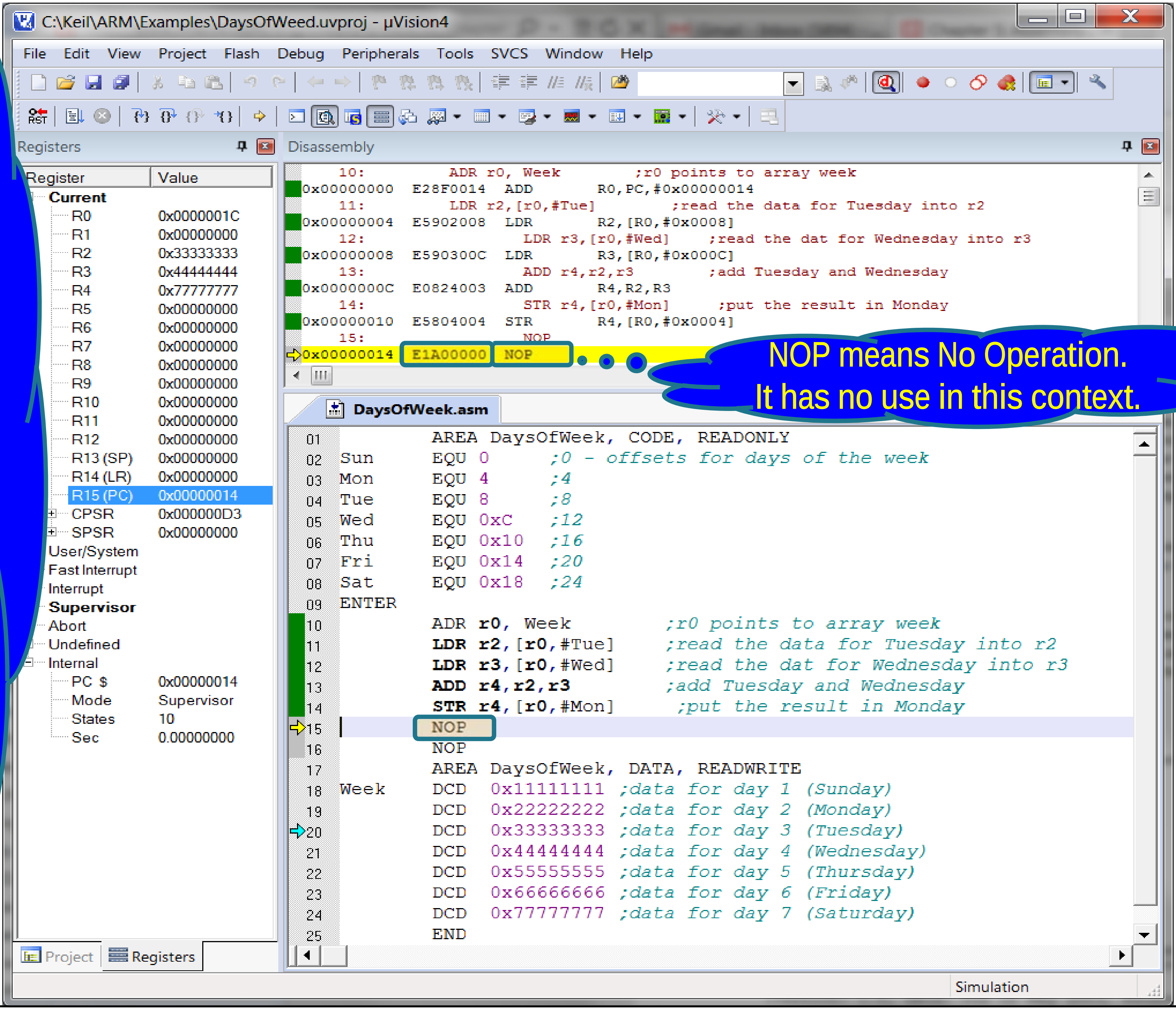
To store the address of Week
in r0, you may also use
LDR r0,=Week

Register Indirect Addressing with an Offset

NOP is a pseudo instruction.

The machine language code for it is “E1A00000”.

Decode this machine language code to know the actual instruction to be executed.


C:\Keil\ARM\Examples\DaysOfWeed.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Disassembly
DaysOfWeek.asm
Project
Registers
Simulation

Register	Value
Current	
R0	0x0000001C
R1	0x00000000
R2	0x33333333
R3	0x44444444
R4	0x77777777
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000014
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x00000014
Mode	Supervisor
States	10
Sec	0.00000000

```
    10:          ADR r0, Week         ;r0 points to array week
0x00000000  E28F0014  ADD       R0,PC,#0x00000014
    11:          LDR r2,[r0,#Tue]         ;read the data for Tuesday into r2
0x00000004  E5902008  LDR       R2,[R0,#0x0008]
    12:                  LDR r3,[r0,#Wed]     ;read the dat for Wednesday into r3
0x00000008  E590300C  LDR       R3,[R0,#0x000C]
    13:                  ADD r4,r2,r3         ;add Tuesday and Wednesday
0x0000000C  E0824003  ADD       R4,R2,R3
    14:                  STR r4,[r0,#Mon]      ;put the result in Monday
0x00000010  E5804004  STR       R4,[R0,#0x0004]
    15:                  NOP
0x00000014  E1A00000  NOP
```

NOP means No Operation.
It has no use in this context.

```
          AREA  DaysOfWeek, CODE, READONLY
Sun       EQU  0      ;0 - offsets for days of the week
Mon       EQU  4      ;4
Tue       EQU  8      ;8
Wed       EQU  0xC    ;12
Thu       EQU  0x10   ;16
Fri       EQU  0x14   ;20
Sat       EQU  0x18   ;24
ENTER
          ADR r0, Week           ;r0 points to array week
          LDR r2,[r0,#Tue]       ;read the data for Tuesday into r2
          LDR r3,[r0,#Wed]       ;read the dat for Wednesday into r3
          ADD r4,r2,r3           ;add Tuesday and Wednesday
          STR r4,[r0,#Mon]        ;put the result in Monday
          NOP
          NOP
          AREA  DaysOfWeek, DATA, READWRITE
Week      DCD  0x11111111 ;data for day 1 (Sunday)
          DCD  0x22222222 ;data for day 2 (Monday)
          DCD  0x33333333 ;data for day 3 (Tuesday)
          DCD  0x44444444 ;data for day 4 (Wednesday)
          DCD  0x55555555 ;data for day 5 (Thursday)
          DCD  0x66666666 ;data for day 6 (Friday)
          DCD  0x77777777 ;data for day 7 (Saturday)
          END
```

Register Indirect Addressing with an Offset

- ❑ Any **ARM** register can be used to implement register indirect addressing.
- ❑ However, **r15** is not just a register; it is the *program counter*.
- ❑ If **r15** is used as a *pointer register* to access an operand, the resulting address is called ***program counter relative addressing***.
  - o The operand location is
    - ▪ specified with respect to the current code location.
  - o Moving the code and its associated data to a different location in memory will not need any recalculation for operand addresses.

- ❑ Consider the instruction

```
LDR r0,[r15,#100]
```

  - o The operand is specified as 100 bytes from the content of **r15**.
  - o This is ***not*** 100 bytes from the "`LDR r0,[r15,#100]`" instruction.
  - o Note that, the **PC** (**r15**) is incremented after fetching an instruction.
    - ▪ The **ARM**'s **PC** is actually 8 bytes after the current instruction (*due to the use of the pipelining mechanism that overlaps operations*)

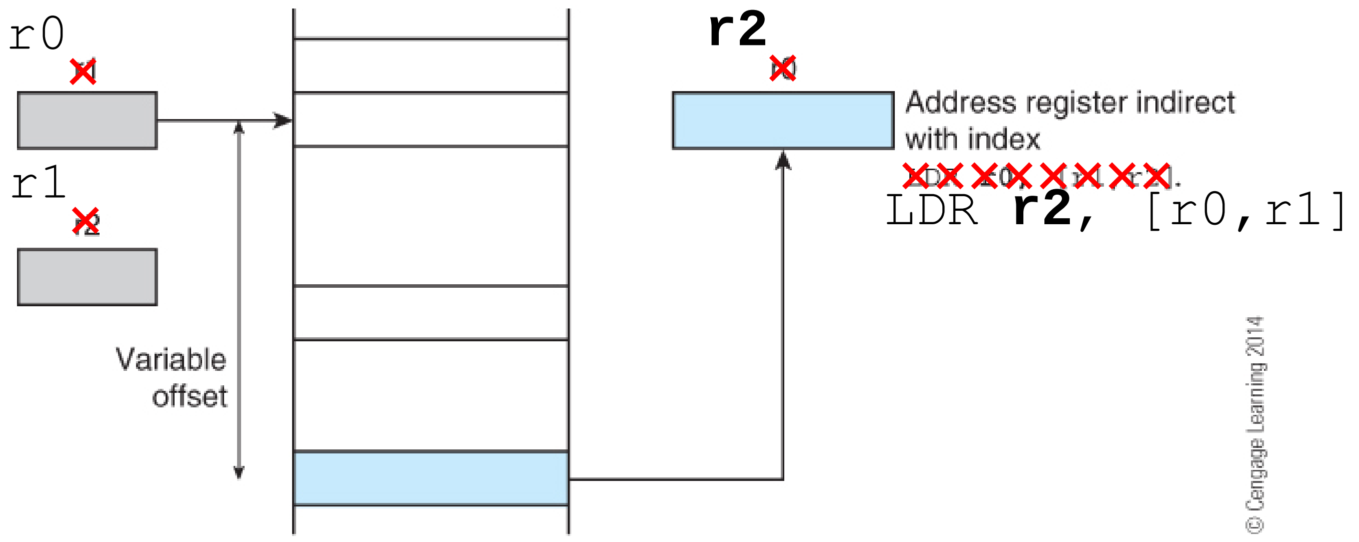
Register Indirect Addressing with Base and Index Registers

- You can specify the offset as a second register so that you can use a *dynamic offset* that can be modified at runtime (See Figure 3.35).

```
LDR r2,[r0,r1]              ;[r2] ← [[r0] + [r1]] load r2 with
                            ;the location pointed at by r0 plus r1
```

The above instruction and the figure in the book (page 188 - 189) are *not* compatible.
You should change one of them.

FIGURE 3.35 Indexed addressing with a register offset

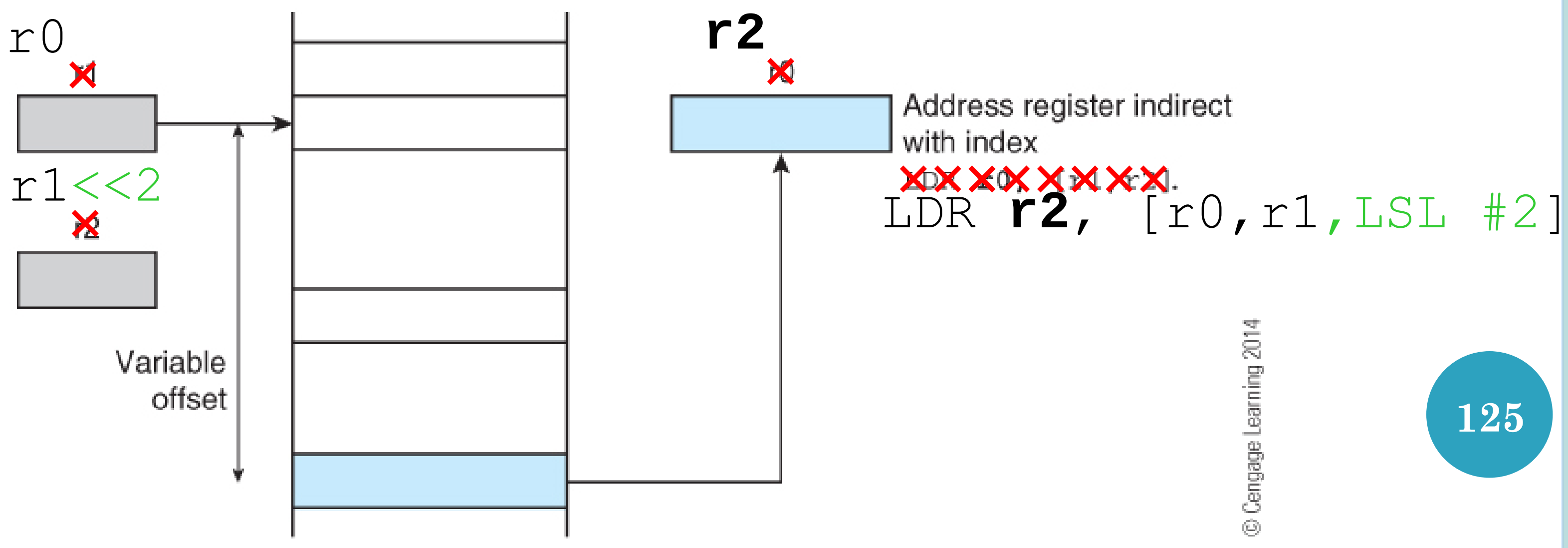


Register Indirect Addressing with Base and Index Registers + Scaling

- ❑ In this example below, register r1 is multipled by 4.
  - o This allows you to use a scaled offset when dealing with arrays.

```
LDR r2,[r0,r1,LSL #2] ;[r2] ← [[r0] + 4 × [r1]] Scale r1 by 4
```

FIGURE 3.35 Indexed addressing with a register offset



Register Indirect Addressing with Base and Index Registers + Scaling

- Example: Consider the following fragment of C code, where `j` is a `long int` array:

```
for(i = 0; i < 21; i++)
{
      j[i] = j[i] + 10;
}
```

- This C code can be translated into ***ARM*** assembly language as follow

```
     MOV    r0,#0              ;Use r0 as the counter i
                               ;Initialize counter i to zero
     ADR    r8,j               ;Index register r8 points to
                               ;array j (pseudo instruction)
Loop LDR    r1,[r8,r0,lsl #2]  ;REPEAT Get j[i]
     ADD    r1,r1,#10          ;  Add 10 to j[i]
     STR    r1,[r8,r0,lsl #2]  ;  Save j[i]
     ADD    r0,r0,#1           ;  Increment loop counter i
     CMP    r0,#21             ;  Compare loop counter with
                               ;  terminal value + 1
     BNE    Loop               ;UNTIL i = 21
```

Not correct in the book page 186

Not correct in the book page 186

Auto-indexing Addressing Mode

- Elements in an array, or similar data structure, are frequently accessed sequentially.
  - To facilitate such action, ***Auto-indexing addressing*** modes have been implemented.
    - In ***Auto-indexing addressing*** modes, the ***pointer is automatically adjusted*** to point at the next element ***before*** or ***after*** it is used, i.e., similar to `*++p` and `*p++`, respectively, in C
- **ARM**'s auto-indexing modes are implemented by
  - adding an offset to the base (i.e., pointer register).
- **ARM** implements two auto-indexing modes
  - Auto-indexing ***pre***-indexed
  - Auto-indexing ***post***-indexed

Auto-indexing Pre-indexed Addressing Mode

- ❑ *Auto-indexing **pre**-indexed* addressing
  - o increments the base register by an offset
  - o accesses the operand at the location pointed to by the ***updated*** base register.
  - o similar to `*++p` in C
- ❑ **ARM**'s *auto-indexing **pre**-indexed* addressing mode is
  - o ***indicated by*** appending the suffix **!** to the end of the address.
- ❑ Consider the following ARM instruction:

```
LDR  r0,[r1,#8]!   ;load r0 with the word pointed at by
                   ;register r1 plus 8 and update the
                   ;pointer by adding 8 to r1
```

- ❑ The RTL definition of this instruction is given by

`[r0] ← [[r1] + 8]` Access the memory 8 bytes beyond the base register r1

`[r1] ←  [r1] + 8` Update the pointer (base register) by adding the offset

- ❑ This *auto-indexing **pre**-indexed mode* does not cost additional execution time, because it is performed in parallel with memory access.

Auto-indexing Pre-indexed Addressing Mode

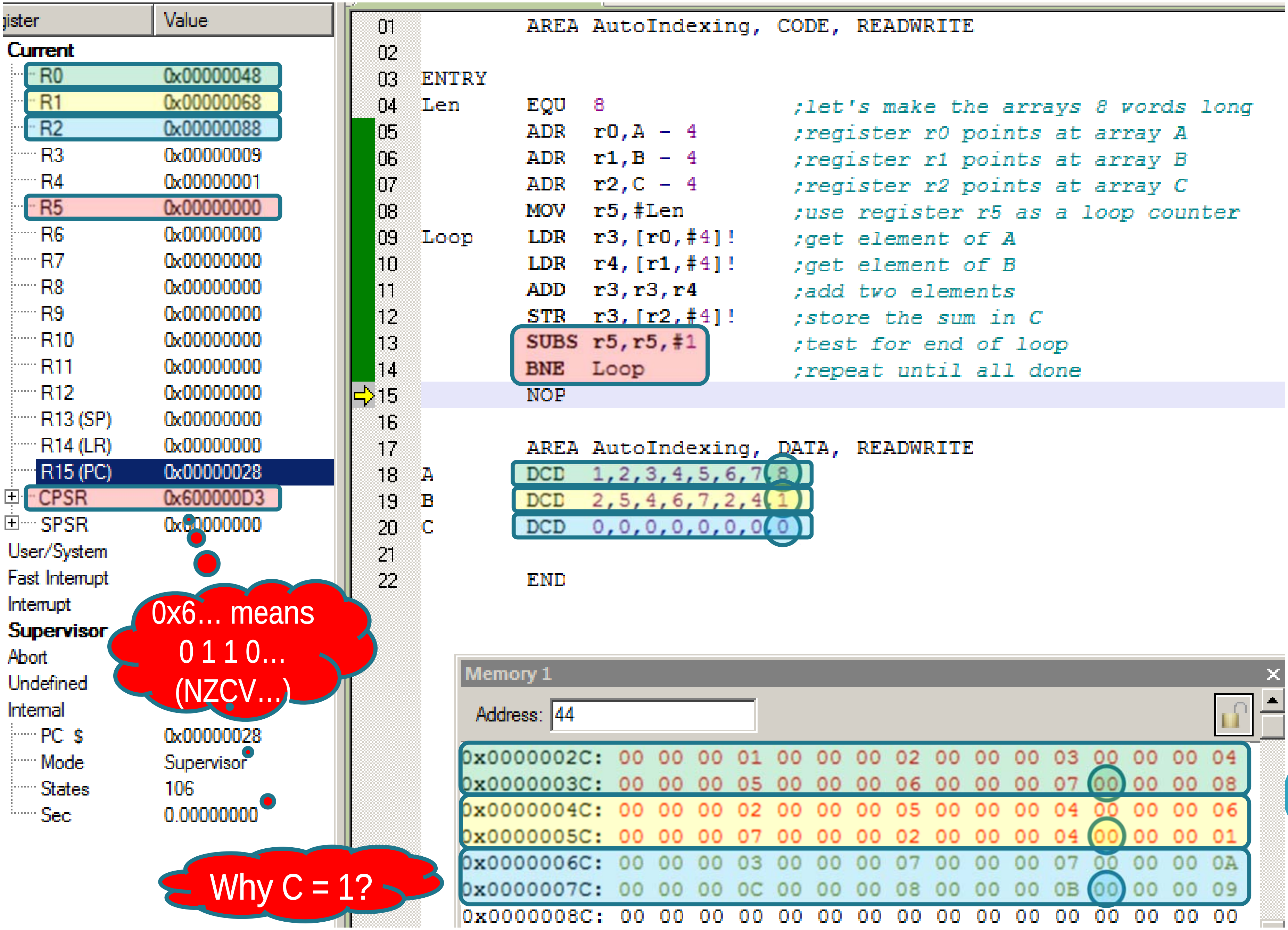
- Consider this example of the addition of two arrays (8 elements each).

```
Len  EQU  8               ;let's make the arrays 8 words long
     ADR  r0,A - 4        ;register r0 points at 4 bytes prior
                          ;to the beginning of array A
     ADR  r1,B - 4        ;register r1 points at 4 bytes prior
                          ;to the beginning of array B
     ADR  r2,C - 4        ;register r2 points at 4 bytes prior
                          ;to the beginning of array C
     MOV  r5,#Len         ;use register r5 as a loop counter

Loop LDR  r3,[r0,#4]!     ;get element of A
     LDR  r4,[r1,#4]!     ;get element of B
     ADD  r3,r3,r4        ;add two elements
     STR  r3,[r2,#4]!     ;store the sum in C

     SUBS r5,r5,#1        ;test for end of loop
     BNE  Loop            ;repeat until all done
```

Auto-indexing Pre-indexed Addressing Mode



gister	Value
Current	
R0	0x00000048
R1	0x00000068
R2	0x00000088
R3	0x00000009
R4	0x00000001
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000028
CPSR	0x600000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x00000028
Mode	Supervisor
States	106
Sec	0.00000000

```
            AREA AutoIndexing, CODE, READWRITE

ENTRY
Len         EQU   8                       ;let's make the arrays 8 words long
            ADR   r0,A - 4                ;register r0 points at array A
            ADR   r1,B - 4                ;register r1 points at array B
            ADR   r2,C - 4                ;register r2 points at array C
            MOV   r5,#Len                 ;use register r5 as a loop counter
Loop        LDR   r3,[r0,#4]!             ;get element of A
            LDR   r4,[r1,#4]!             ;get element of B
            ADD   r3,r3,r4                ;add two elements
            STR   r3,[r2,#4]!             ;store the sum in C
            SUBS  r5,r5,#1                ;test for end of loop
            BNE   Loop                    ;repeat until all done
            NOP

            AREA AutoIndexing, DATA, READWRITE
A           DCD   1,2,3,4,5,6,7,8
B           DCD   2,5,4,6,7,2,4,1
C           DCD   0,0,0,0,0,0,0,0

            END
```

Memory 1

Address: 44

```
0x0000002C: 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04
0x0000003C: 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00 08
0x0000004C: 00 00 00 02 00 00 00 05 00 00 00 04 00 00 00 06
0x0000005C: 00 00 00 07 00 00 00 02 00 00 00 04 00 00 00 01
0x0000006C: 00 00 00 03 00 00 00 07 00 00 00 07 00 00 00 0A
0x0000007C: 00 00 00 0C 00 00 00 08 00 00 00 0B 00 00 00 09
0x0000008C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

0x6… means 0 1 1 0… (NZCV…)

Why C = 1?

Auto-indexing Post-indexed Addressing Mode

- ❑ Auto-indexing ***post****-indexed* addressing
  - o first accesses the operand at the location pointed to by the base register,
  - o ***then*** increments the base register.
  - o similar to `*p++` in C

- ❑ ARM's *auto-indexing* ***post****-indexed* is ***denoted by placing the offset outside the square***.

- ❑ Example:

```
 LDR   r0,[r1],#8 ;load r0 with the word pointed at by r1
                  ;now do the post-indexing by adding 8 to r1
```

- ❑ The RTL definition of this instruction is:

```
[r0] ← [[r1]]      Access the memory address in base register r1
[r1] ←  [r1] + 8   Update pointer (base register) by adding offset
```

Auto-indexing Post-indexed Addressing Mode

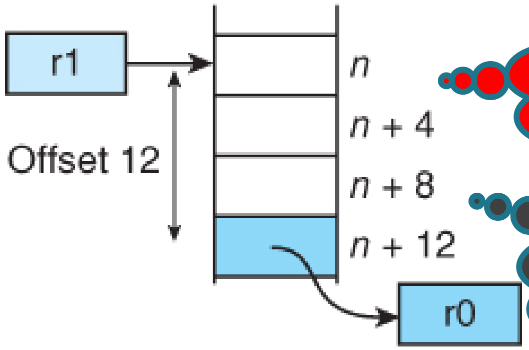
- ❑ Consider this example of the addition of two arrays (8 elements each).

```
Len  EQU   8               ;let's make the arrays 8 words long
     ADR   r0,A            ;register r0 points at array A
     ADR   r1,B            ;register r1 points at array B
     ADR   r2,C            ;register r2 points at array C
     MOV   r5,#Len         ;use register r5 as a loop counter

Loop LDR   r3,[r0],#4      ;get element of A
     LDR   r4,[r1],#4      ;get element of B
     ADD   r3,r3,r4        ;add two elements
     STR   r3,[r2],#4      ;store the sum in C

     SUBS  r5,r5,#1        ;test for end of loop
     BNE   Loop            ;repeat until all done
```

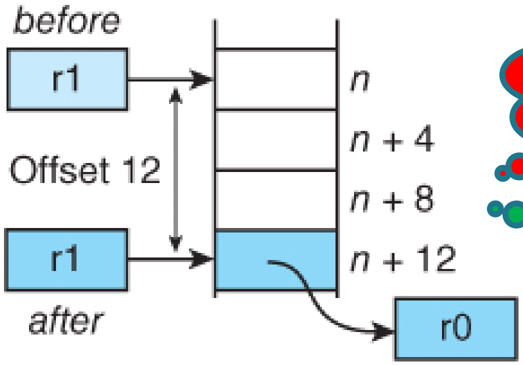
Register Indirect Addressing with Offset



Adjust pointer then use the adjusted pointer.

Do NOT write-back (do not update) the adjusted pointer.

(a) `LDR r0,[r1,#12]`
Offset added to base register to generate effective address. Operand accessed at effective address. Base register remains unchanged.

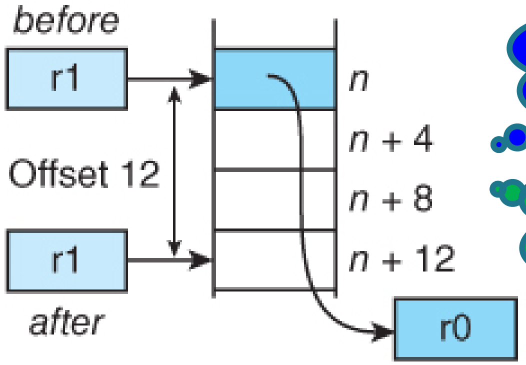


Adjust pointer then use the adjusted pointer.

Write-back (update) the adjusted pointer.

(b) `LDR r0,[r1,#12]!`
Offset added to base register to generate effective address. Operand accessed at effective address. Base register updated after access.

Why do not we have “Use the original base pointer then adjust pointer” with “Do NOT write-back (do not update) the adjusted pointer”?



Use the original base pointer then adjust pointer.

Write-back (update) the adjusted pointer.

(c) `LDR r0,[r1], #12`
Effective address specified by base register. Operand accessed at effective address. Offset added to base register after the access.