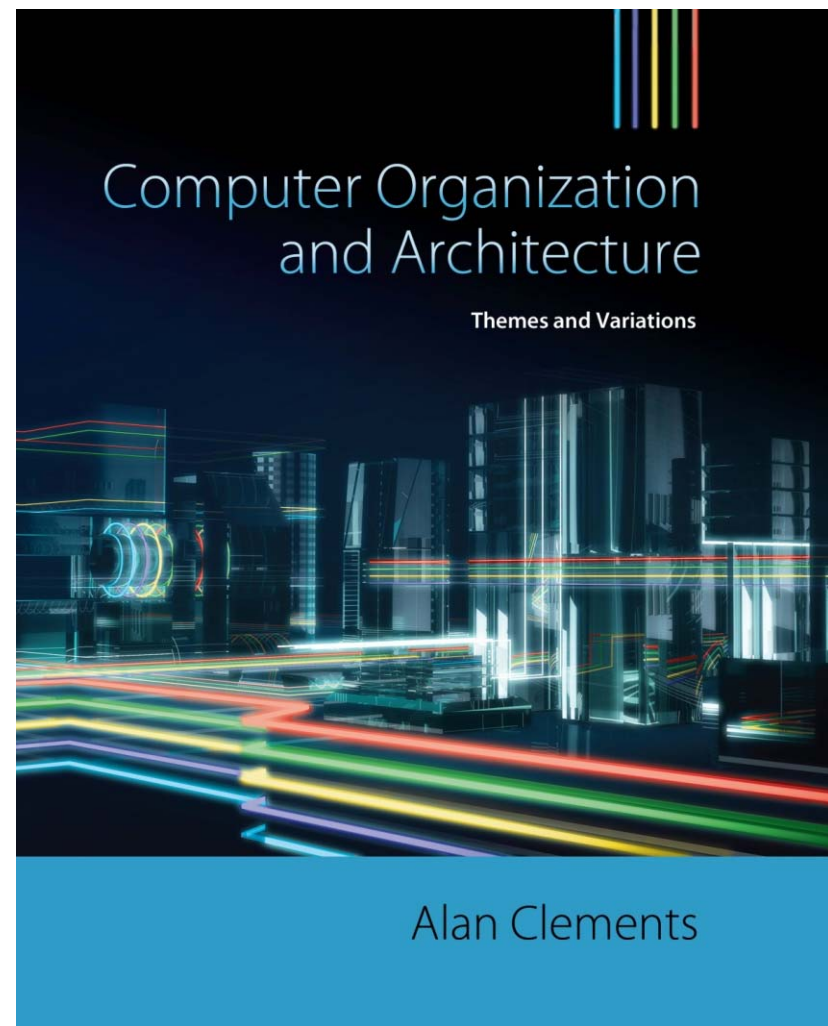


# Part 0x8

## CHAPTER 3

### Architecture and Organization

1



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.

# Addressing Modes

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

<u>Instruction</u>	<u>RTL form</u>	<u>Description</u>
ADD <b>r0</b> , r1, #Q	$[r0] \leftarrow [r1] + Q$	<p><b>Literal:</b></p> <p>Add the integer Q to the content of register r1 and store the result in r0</p>
LDR <b>r0</b> , Mem	$[r0] \leftarrow [Mem]$	<p><b>Direct</b> (i.e., <b>absolute</b>) :</p> <p>Load the content of memory-location Mem into register r0.</p> <p><i>This addressing mode is <b>not supported by ARM</b> but is <b>supported by all CISC processors</b></i></p>
LDR <b>r0</b> , [r1]	$[r0] \leftarrow [[r1]]$	<p><b>Register Indirect:</b></p> <p>Load r0 with the content of the memory-location pointed at by r1</p> <p><i>The memory-location is given by the contents of a register (that is why we call it <b>Register Indirect</b>)</i></p>

❑ This is also called:

- **Indexed**
- **Pointer-based**

- ❑ 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

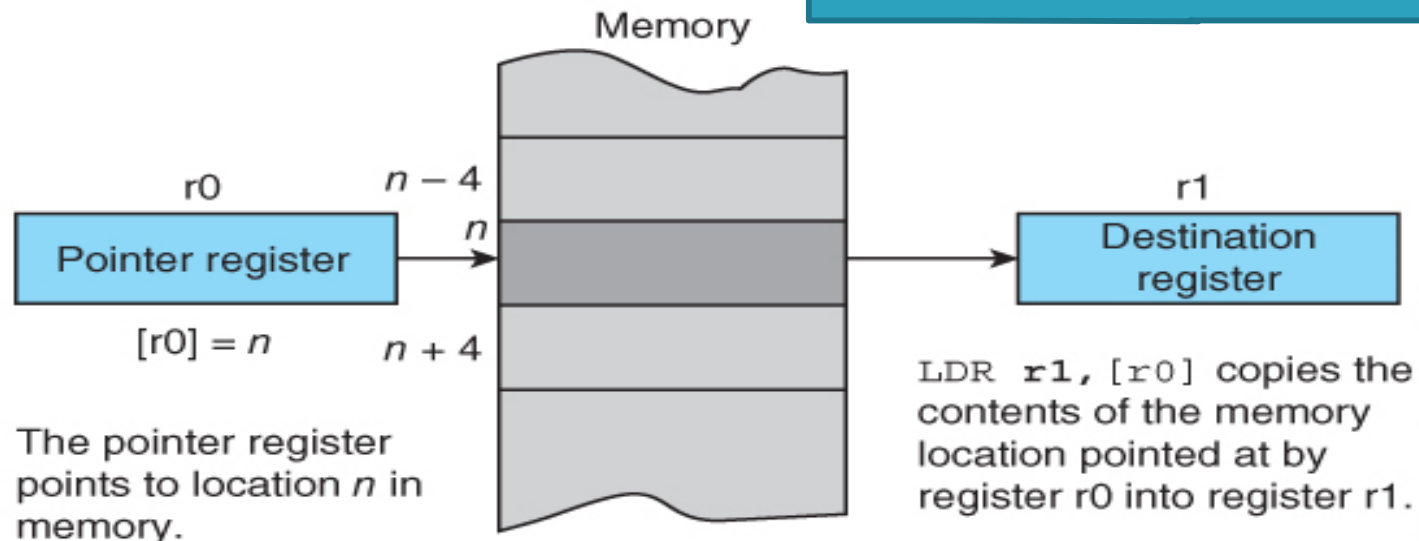
- In **ARM**, the register indirect addressing is **indicated** by means of **square brackets**; for example,

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

FIGURE 3.31

Register indirect addressing

If we look to the memory as an array, then “LDR **r1**, [r0]” means  $r1 = \text{memory}[r0]$ .



© Cengage Learning 2014

## Register Indirect Addressing

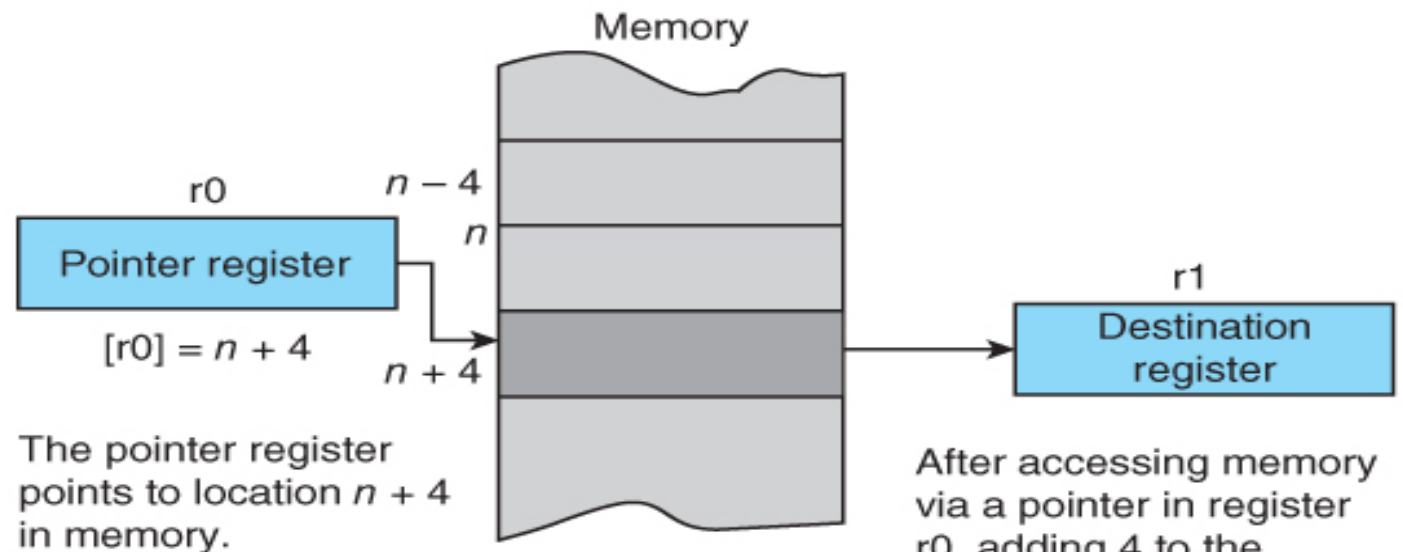
- Consider what happens if we next execute

```
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



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.

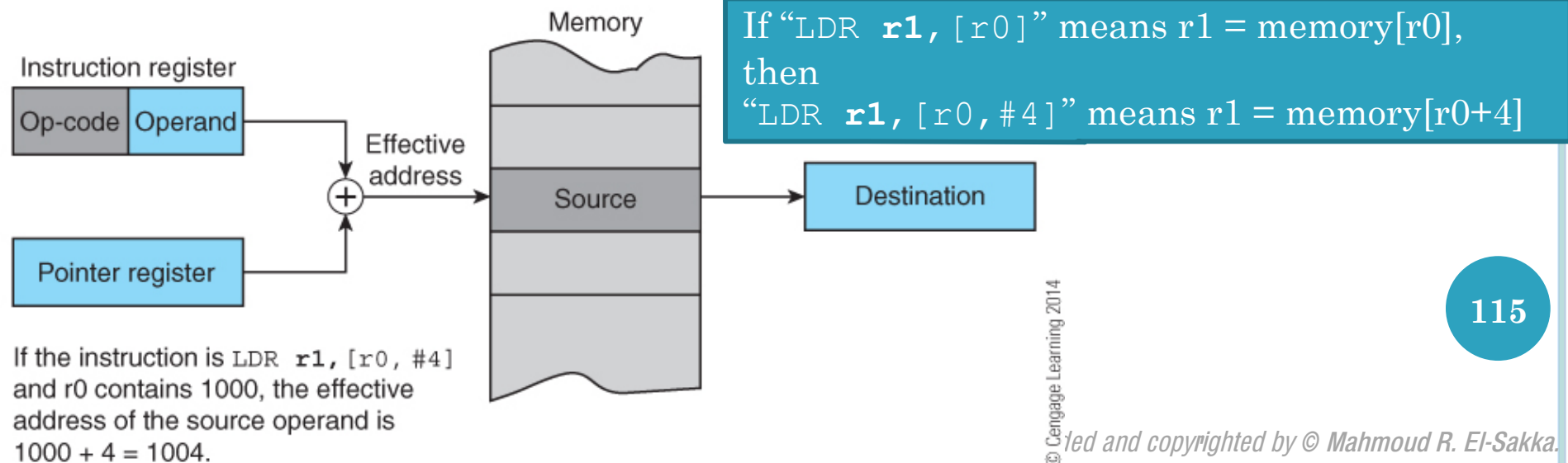
© Cengage Learning 2014

moud R. El-Sakka.

# 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.  
The literal offset must be preceded by “#” sign
- ❑ 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



# Register Indirect Addressing with an Offset

- ❑ The following fragment of code demonstrates the use of offsets to implement array access.
- ❑ *Note that: constants (defined by EQU here) **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
```

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

If “STR r4, [r0]” means memory[r0] = r4,  
then  
“STR r4, [r0, #4]” means memory[r0+4] = r4

```
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)
```



# 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.

The screenshot shows the Keil uVision4 IDE with the following components:

- Registers Window:**

Register	Value
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
- Disassembly Window:**

```

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

```
- Source Code Window (DaysOfWeek.asm):**

```

01      AREA DaysOfWeek, CODE, READONLY
02      Sun EQU 0          ;0 - offsets for days of the week
03      Mon EQU 4          ;4
04      Tue EQU 8          ;8
05      Wed EQU 0xC        ;12
06      Thu EQU 0x10       ;16
07      Fri EQU 0x14       ;20
08      Sat EQU 0x18       ;24
09      ENTER
10      ADR r0, Week      ;r0 points to array week
11      LDR r2,[r0,#Tue]  ;read the data for Tuesday into r2
12      LDR r3,[r0,#Wed]  ;read the dat for Wednesday into r3
13      ADD r4,r2,r3      ;add Tuesday and Wednesday
14      STR r4,[r0,#Mon]  ;put the result in Monday
15      NOP
16      NOP
17      AREA DaysOfWeek, DATA, READWRITE
18      Week DCD 0x11111111 ;data for day 1 (Sunday)
19      DCD 0x22222222 ;data for day 2 (Monday)
20      DCD 0x33333333 ;data for day 3 (Tuesday)
21      DCD 0x44444444 ;data for day 4 (Wednesday)
22      DCD 0x55555555 ;data for day 5 (Thursday)
23      DCD 0x66666666 ;data for day 6 (Friday)
24      DCD 0x77777777 ;data for day 7 (Saturday)
25      END

```

## Program counter Relative Addressing

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

- ❑ Consider the instruction

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

- The operand is specified as 100 bytes from the content of **r15**.
- This is *not* 100 bytes from the “LDR **r0**, [r15, #100]” instruction.
- Note that, the **PC** (**r15**) is incremented after fetching an instruction.
  - The **ARM**'s **PC** is actually 8 bytes after the current instruction, i.e., **r0** will be loaded with the value located 108 bytes away from the instruction.  
(*This is 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 + r1

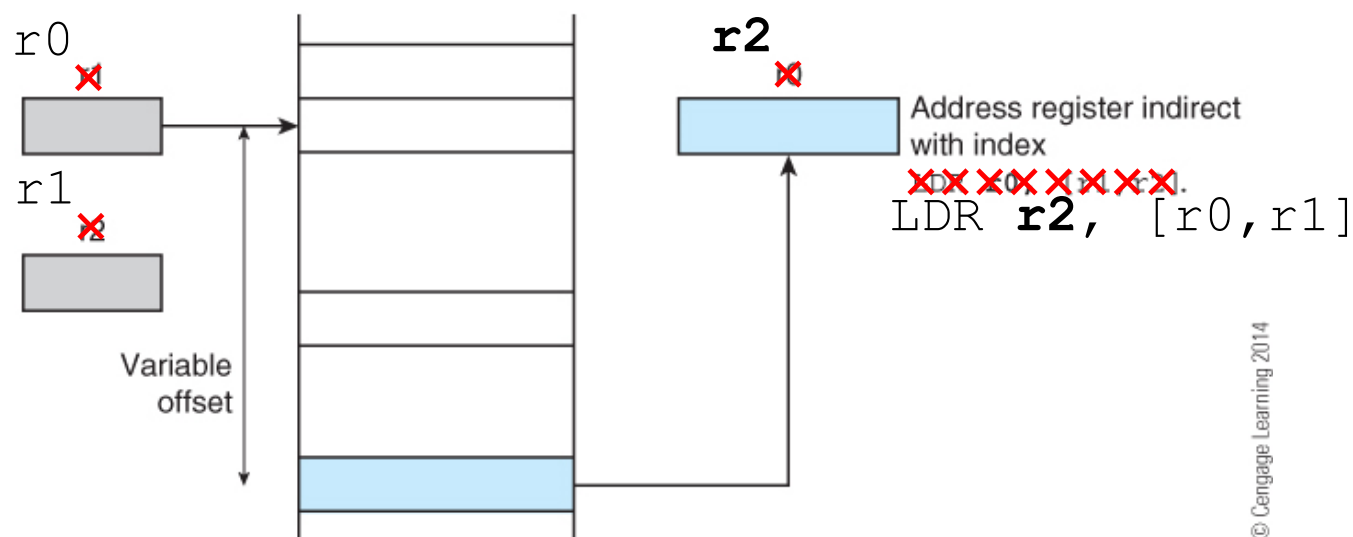
The above instruction and the figure in the book (page 188 - 189)  
are *not* compatible.

You should change one of them.

If "LDR **r2**, [r0]" means  $r2 = \text{memory}[r0]$ ,  
then  
"LDR **r2**, [r0, r1]" means  $r2 = \text{memory}[r0+r1]$

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 multiplied by 4.
  - This allows you to use a scaled offset when dealing with arrays.

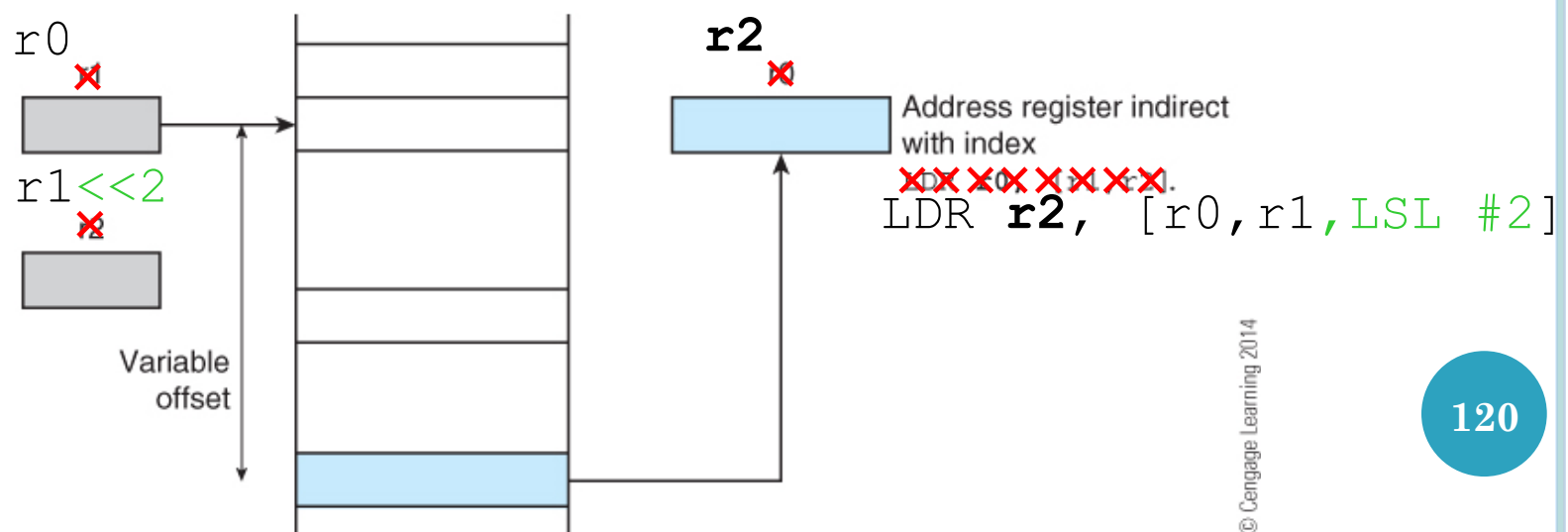
LDR **r2**, [r0, r1, LSL #2] ; [r2] ← [[r0] + [r1] × 4] Scale r1 by 4

If “LDR **r2**, [r0]” means  $r2 = \text{memory}[r0]$ ,  
then

“LDR **r2**, [r0, r1, LSL #2]” means  $r2 = \text{memory}[r0 + r1 \times 4]$

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;
}
```

Array `j` has 21 elements (indexed from 0 to 20 and we want to add 10 to each element of this array).

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

MOV	<b>r0</b> , #0		; Use r0 as the counter i
ADR	<b>r8</b> , j		; Initialize counter i to zero
			; Base register r8 points to
			; array j (pseudo instruction)
Loop LDR	<b>r1</b> , [r8, <b>r0</b> , <b>lsl</b> #2]		; <b>REPEAT</b> Get j[i]
ADD	<b>r1</b> , r1, #10		; Add 10 to j[i]
STR	r1, [r8, <b>r0</b> , <b>lsl</b> #2]		; Save j[i]
ADD	<b>r0</b> , r0, #1		; Increment loop counter i
CMP	r0, #21		; Compare loop counter with
			; <b>terminal value + 1</b>
BNE	Loop		; <b>UNTIL</b> 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 **memory[++r1]** and **memory [r1++]**, respectively, in C and similar to **memory[--r1]** and **memory [r1--]**, respectively, in C
- ❑ **ARM**'s auto-indexing modes are implemented by
  - adding an offset to the base register
- ❑ **ARM** implements two auto-indexing modes
  - Auto-indexing *pre*-indexed
  - Auto-indexing *post*-indexed

In ARM, The amount of the increment or decrement can be any value, not just an increment or decrement by 1, depending on the problem in hand.

- ❑ *Auto-indexing **pre-indexed*** addressing
  - increments the base register by an offset
  - accesses the operand at the location pointed to by the ***updated*** base register.
  - similar to `memory[++r1]` in C
- ❑ **ARM's** *auto-indexing **pre-indexed*** addressing mode is
  - *indicated by* appending the suffix **!** to the end of the address.

```
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

$[r0] \leftarrow [[r1] + 8]$	Access the memory 8 bytes beyond the base register r1
$[r1] \leftarrow [r1] + 8$	Update the pointer (base register) by adding the offset

123

# Auto-indexing Pre-indexed Addressing Mode

- Consider this example of adding two arrays (each array is 8 elements of 4 bytes each).

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



# Auto-indexing Pre-indexed Addressing Mode

Register	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

```

01      AREA AutoIndexing, CODE, READWRITE
02
03  ENTRY
04  Len    EQU    8                ;let's make the arrays 8 words long
05      ADR    r0,A - 4            ;register r0 points at array A
06      ADR    r1,B - 4            ;register r1 points at array B
07      ADR    r2,C - 4            ;register r2 points at array C
08      MOV    r5,#Len             ;use register r5 as a loop counter
09  Loop   LDR    r3,[r0,#4]!       ;get element of A
10         LDR    r4,[r1,#4]!       ;get element of B
11         ADD    r3,r3,r4          ;add two elements
12         STR    r3,[r2,#4]!       ;store the sum in C
13         SUBS   r5,r5,#1          ;test for end of loop
14         BNE    Loop             ;repeat until all done
15      NOP
16
17      AREA AutoIndexing, DATA, READWRITE
18  A      DCD    1,2,3,4,5,6,7,8
19  B      DCD    2,5,4,6,7,2,4,1
20  C      DCD    0,0,0,0,0,0,0,0
21
22      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
  - first accesses the operand at the location pointed to by the base register,
  - *then* increments the base register.
  - similar to `memory[r1++]` in in C
- ❑ ARM's *auto-indexing post-indexed* is *denoted by placing the offset outside the square*.

The amount of the increment can be any value, depending on the problem in hand.

- ❑ Example:

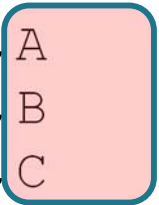
```
LDR    r0, [r1], #8 ; load r0 with the word pointed at by r1
                     ; now do the post-indexing by adding 8 to r1
```

- ❑ The RTL definition of this instruction is:

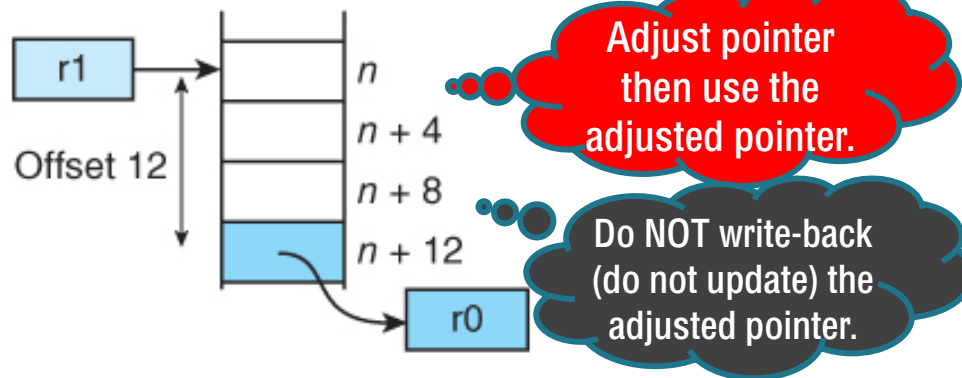
$[r0] \leftarrow [[r1]]$	Access the memory address in base register r1
$[r1] \leftarrow [r1] + 8$	Update pointer (base register) by adding offset

# Auto-indexing Post-indexed Addressing Mode

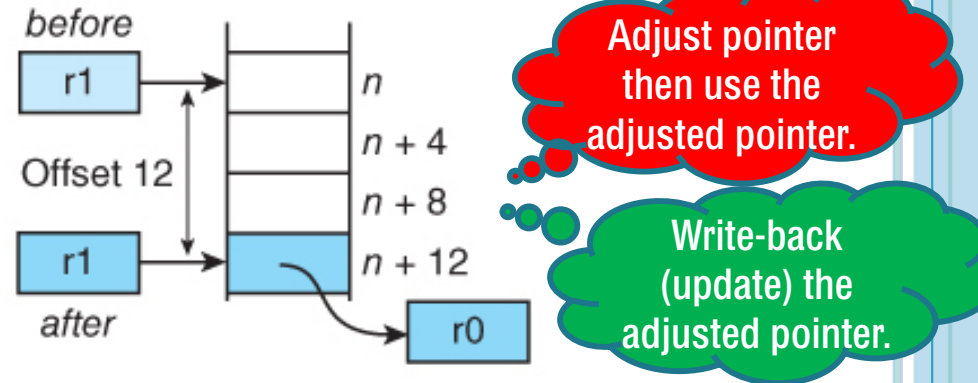
- Consider this example of the addition of two arrays (8 elements each, 4 bytes each).

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

# Register Indirect Addressing with Offset

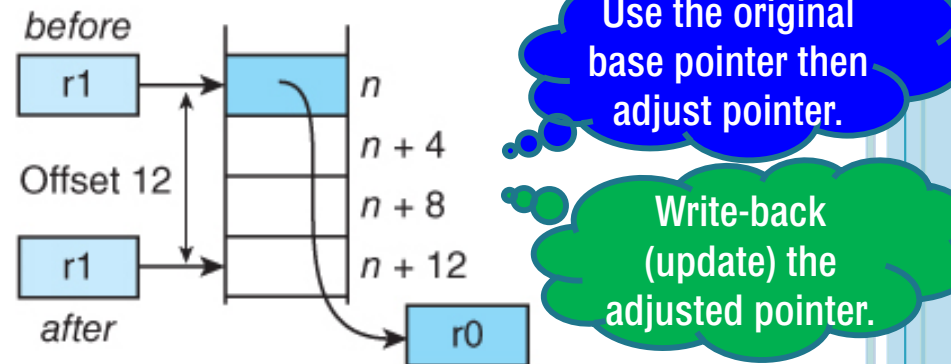


- (a) `LDR r0, [r1, #12]`  
 Offset added to base register to generate effective address. Operand accessed at effective address. Base register remains unchanged.



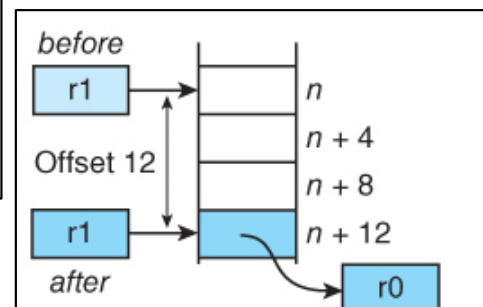
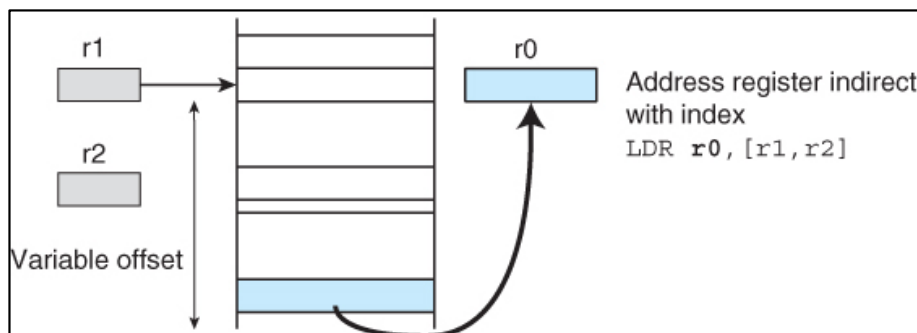
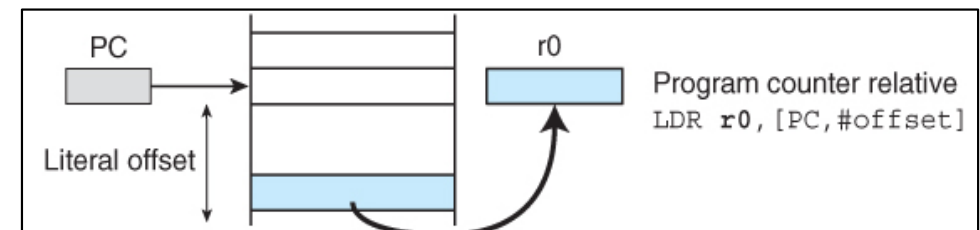
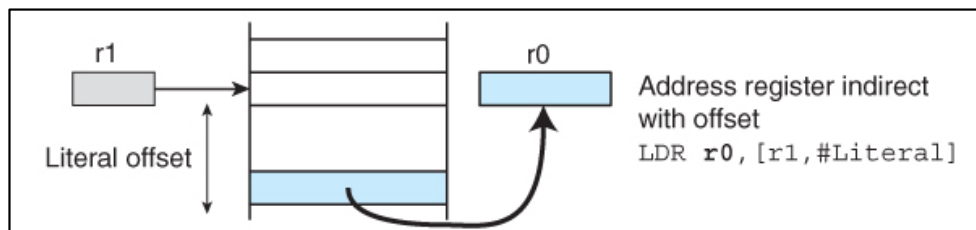
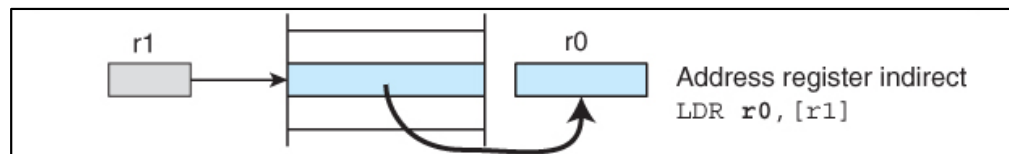
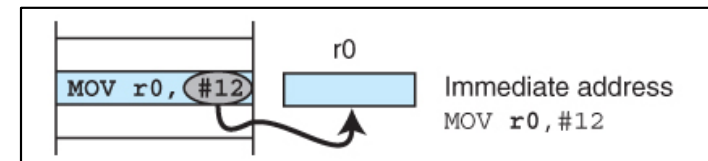
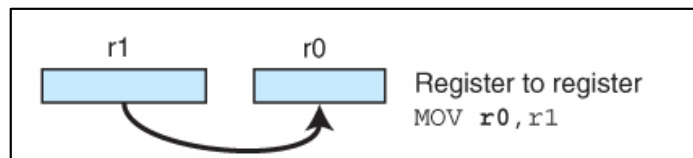
- (b) `LDR r0, [r1, #12]!`  
 Offset added to base register to generate effective address. Operand accessed at effective address. Base register updated after access.

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

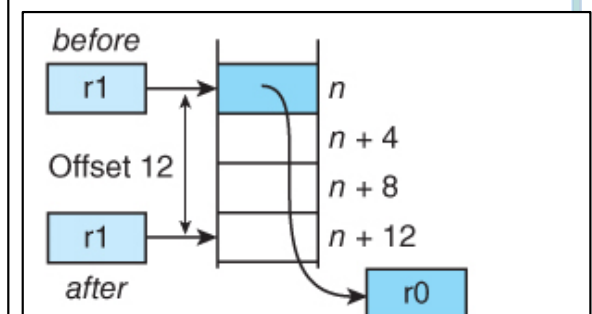


- (c) `LDR r0, [r1], #12`  
 Effective address specified by base register. Operand accessed at effective address. Offset added to base register after the access.

# Summary of ARM Addressing Modes



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



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

The offset in these two cases can also be an index register, for example, "r2" instead of "#12", or can be an index register with shift, for example, "r2, LSL#3" instead of "#12".