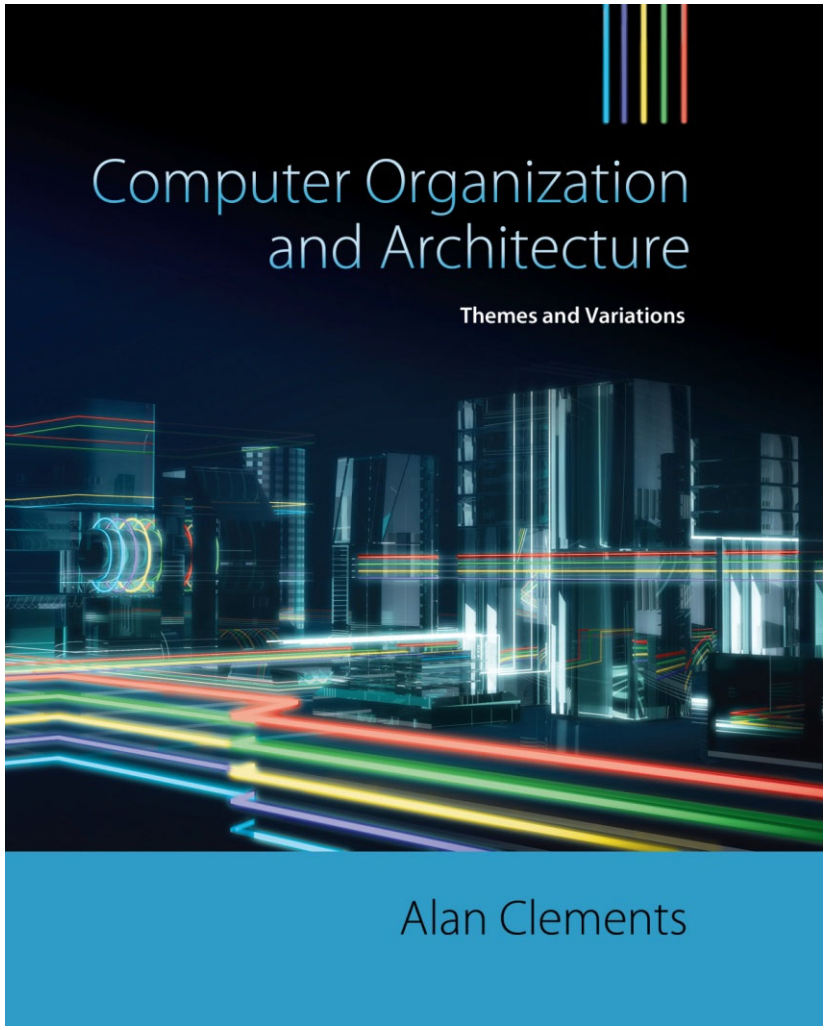
Part 8

Chapter 3

Architecture and Organization



CENGAGE Learning

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 memory-location is given by the contents of a register (that is why we call it **Register Indirect**)*

- **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.)

this is why this method is needed in ARM.

Register Indirect Addressing

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

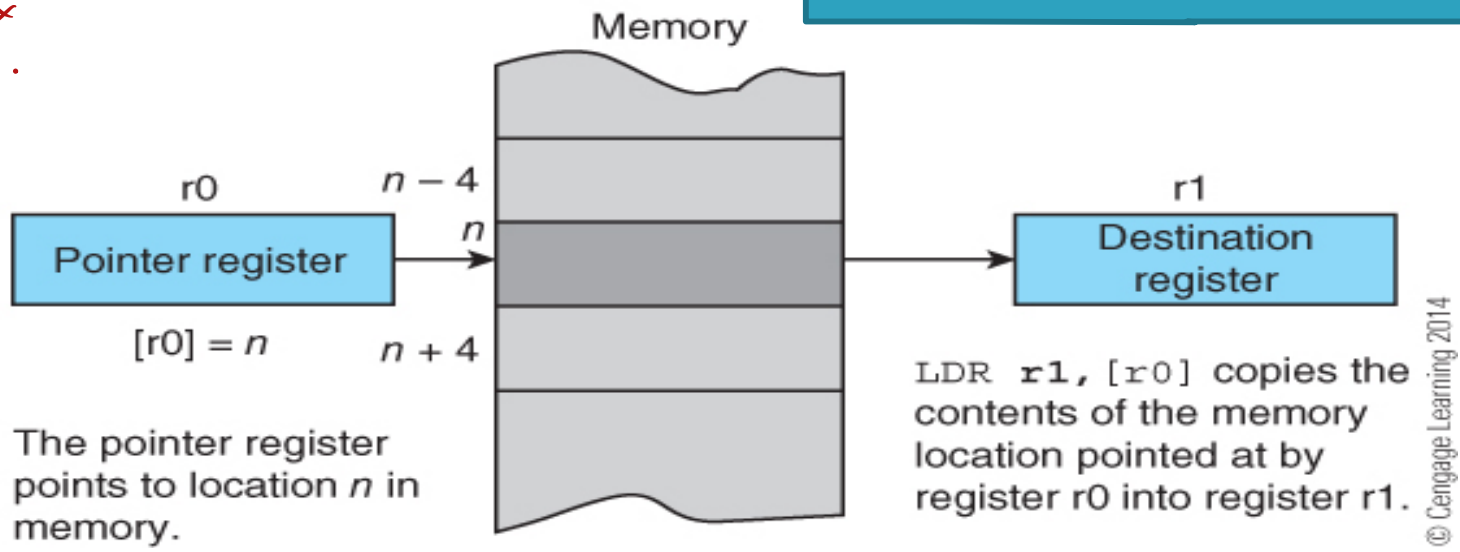
```
LDR r1,[r0]        ;[r1] ← [[r0]]
                   ;Load r1 with the content of
                   ;the memory-location pointed at by r0
```

ADR could only used for label loading (DCD, DCW, DCB).

LDR, R0, =x
ADR R0, x.

If we look to the memory as an array, then "`LDR r1,[r0]`" means r1 = memory[r0]

FIGURE 3.31 Register indirect addressing



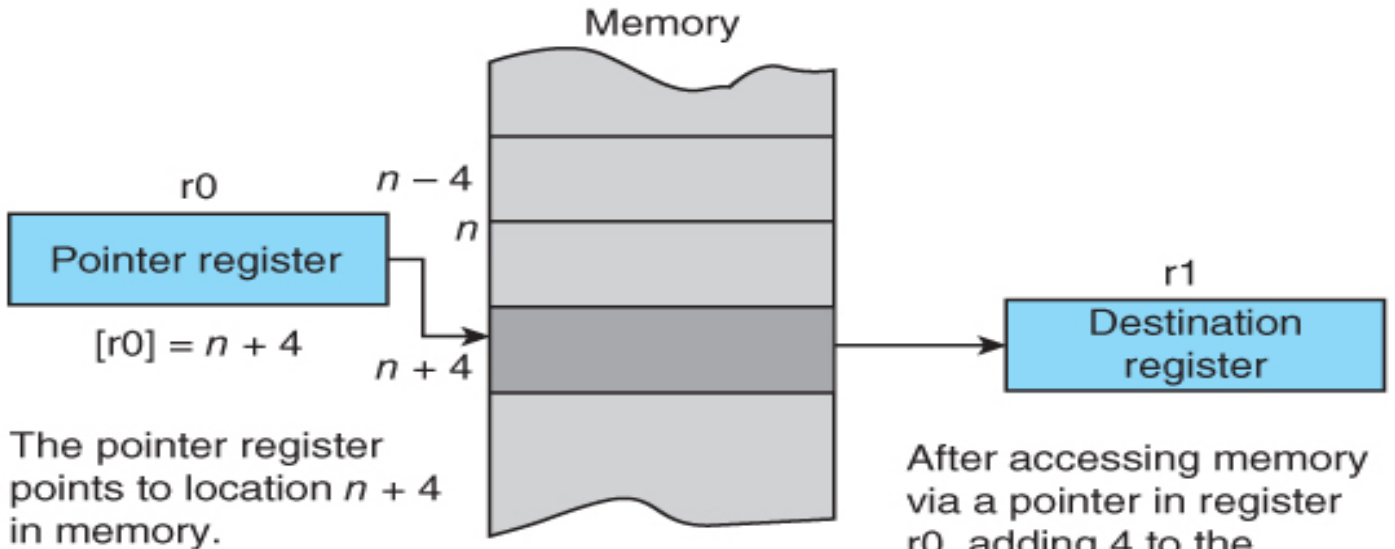
Register Indirect Addressing

- 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



Register Indirect Addressing with an Offset

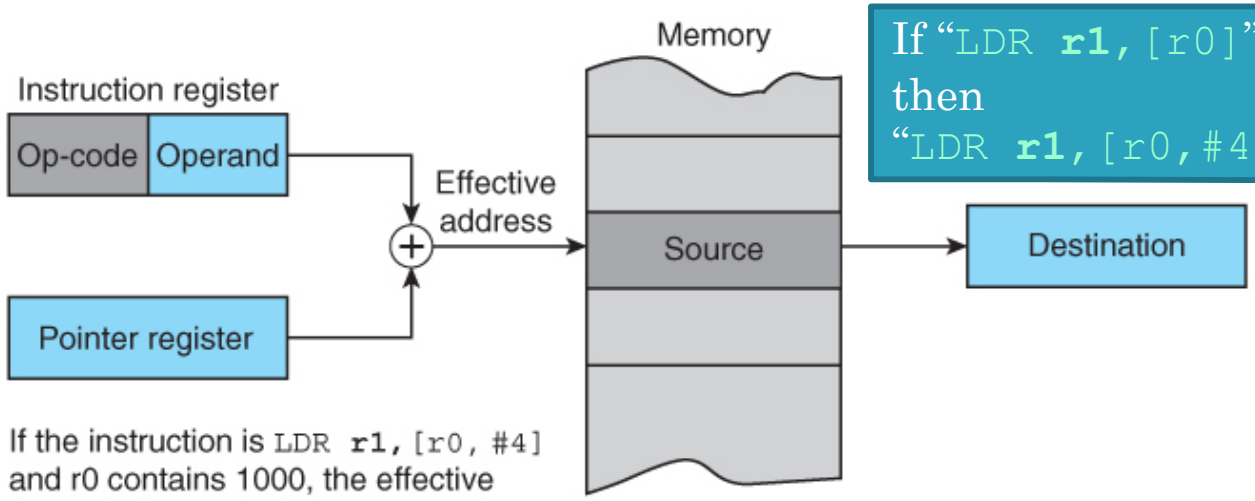
- **ARM** supports a memory-addressing mode where the *effective address* of an operand is **computed by adding** the *contents of a register* to a *literal offset* encoded into the load/store instruction.

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 the literals in the data processing instructions.*

Review Slides 105 -- 112 in Chapter 3.

FIGURE 3.33 Register indirect addressing with an offset



If "`LDR r1,[r0]`" means r1 = memory[r0], then "`LDR r1,[r0,#4]`" means r1 = memory[r0+4]

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

       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

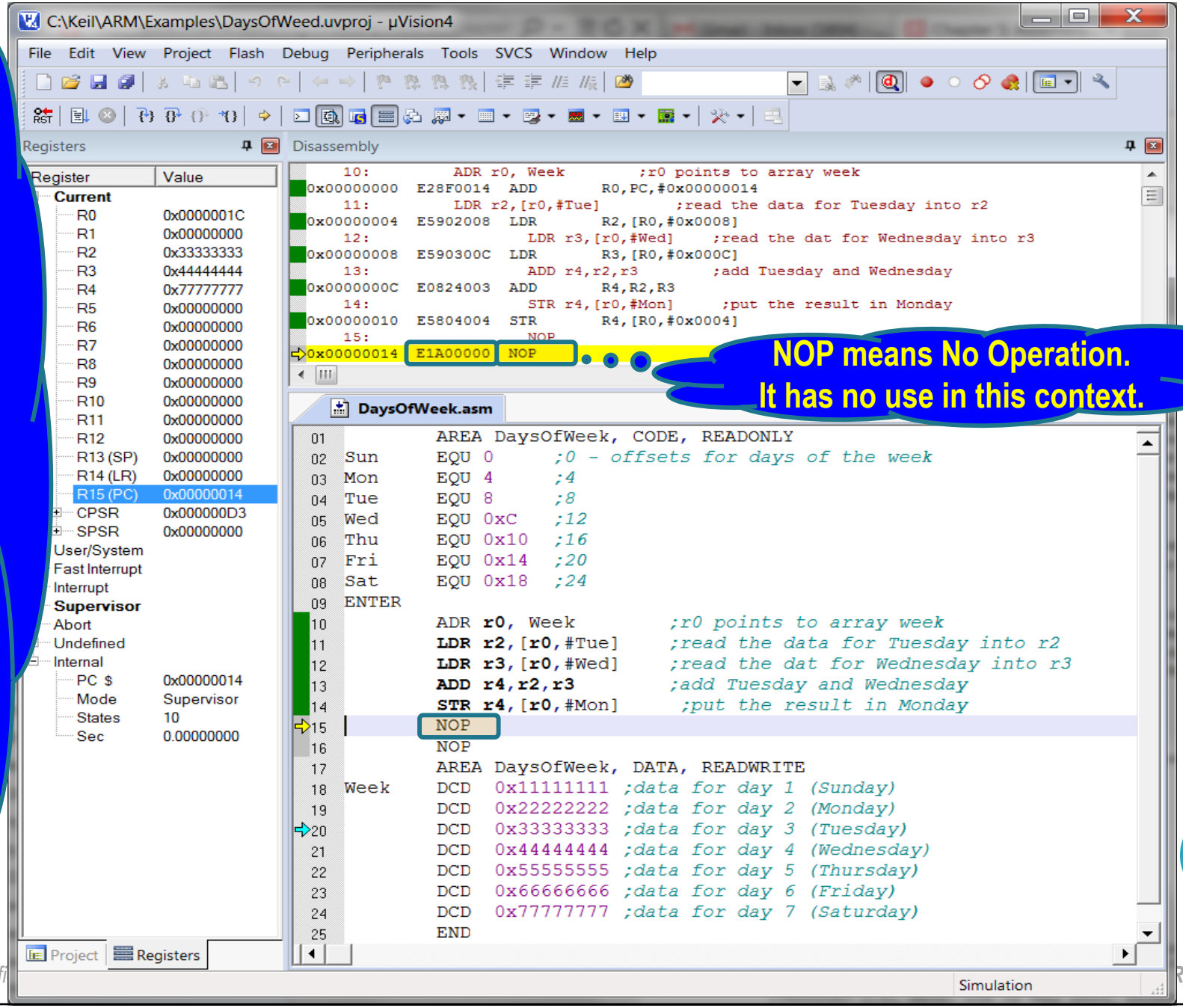
If "STR r4,[r0]" means memory[r0] = r4, then "STR r4,[r0,#4]" means memory[r0+4] = r4

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.



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

Program counter Relative Addressing

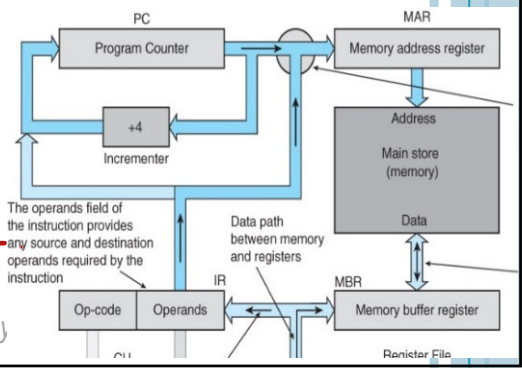
- *Any* **ARM** register can be used to implement register indirect addressing.
- If **r15** (i.e., *program counter*) is used as a *pointer register* to access an operand, the resulting address is called ***program counter relative addressing***.
  - The operand-location is
    - specified with respect 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]
```

PC's add = [curr instruction] + 8
r0 = [curr ins] + 8 + 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

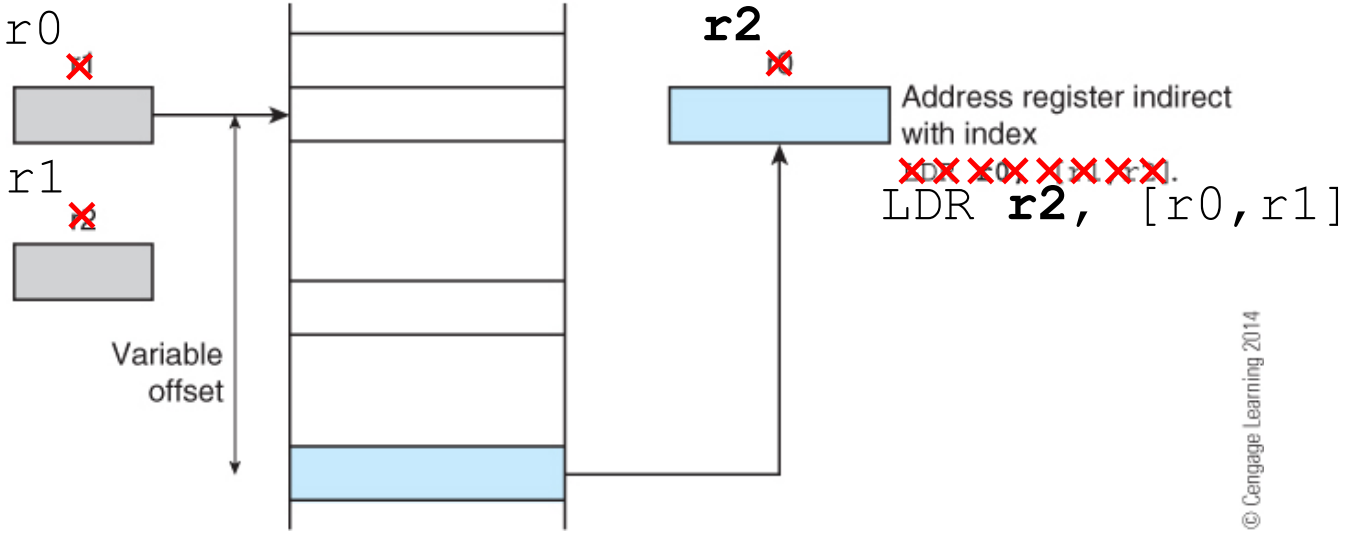
- You can specify the offset as a second register so that you can use a *dynamic offset* that can be modified at runtime (See Figure 3.35).

```
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 = memory[r0], then "`LDR r2,[r0,r1]`" means r2 = memory[r0+r1]

FIGURE 3.35 Indexed addressing with a register offset



Register Indirect Addressing with Base and Index Registers + Scaling

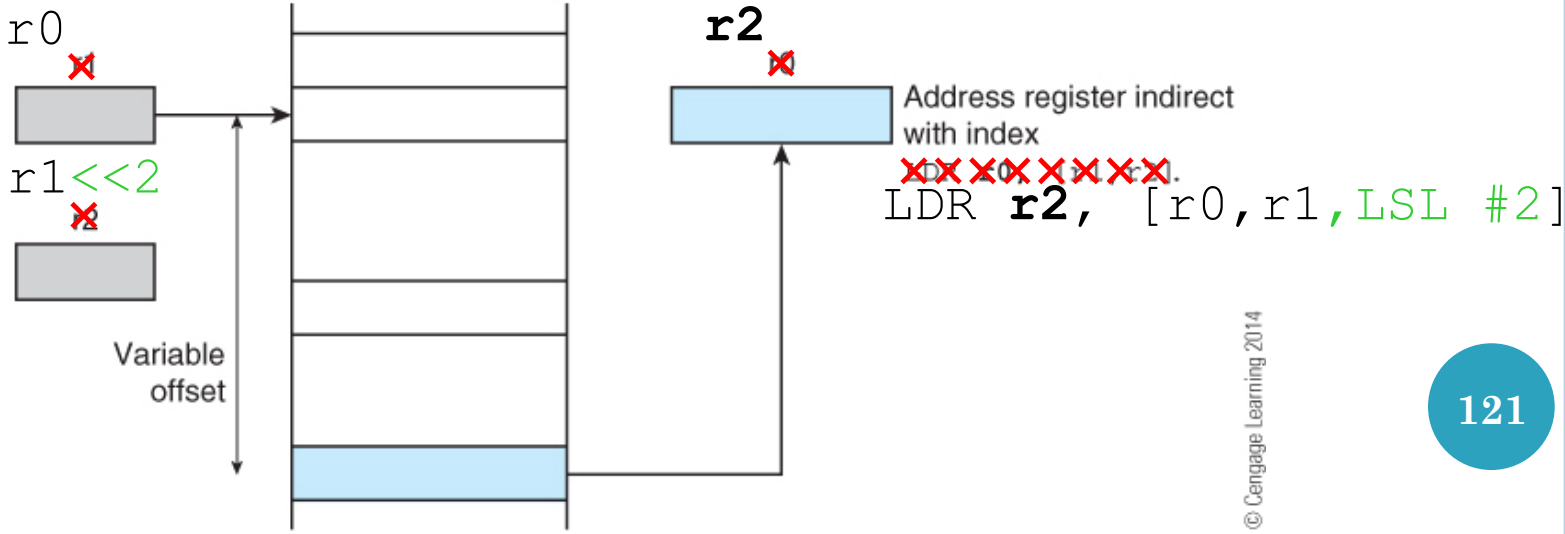
- In this example below, register r1 is multiplied by 4.
  - This allows you to use a scaled offset when dealing with arrays.

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

r2 = [r0 + r1 × 2²].

If "LDR r2,[r0]" means r2 = memory[r0],
then
"LDR r2,[r0,r1,LSL #2]" means r2 = memory[r0+r1×4]

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   r0,#0                ;Use r0 as the counter i
                                ;Initialize counter i to zero
     ADR   r8,j                 ;Base 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 **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
  - Updating the base register by adding an offset to it.

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.

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

- 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 adding two arrays (each array is 8 elements of 4 bytes 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
```

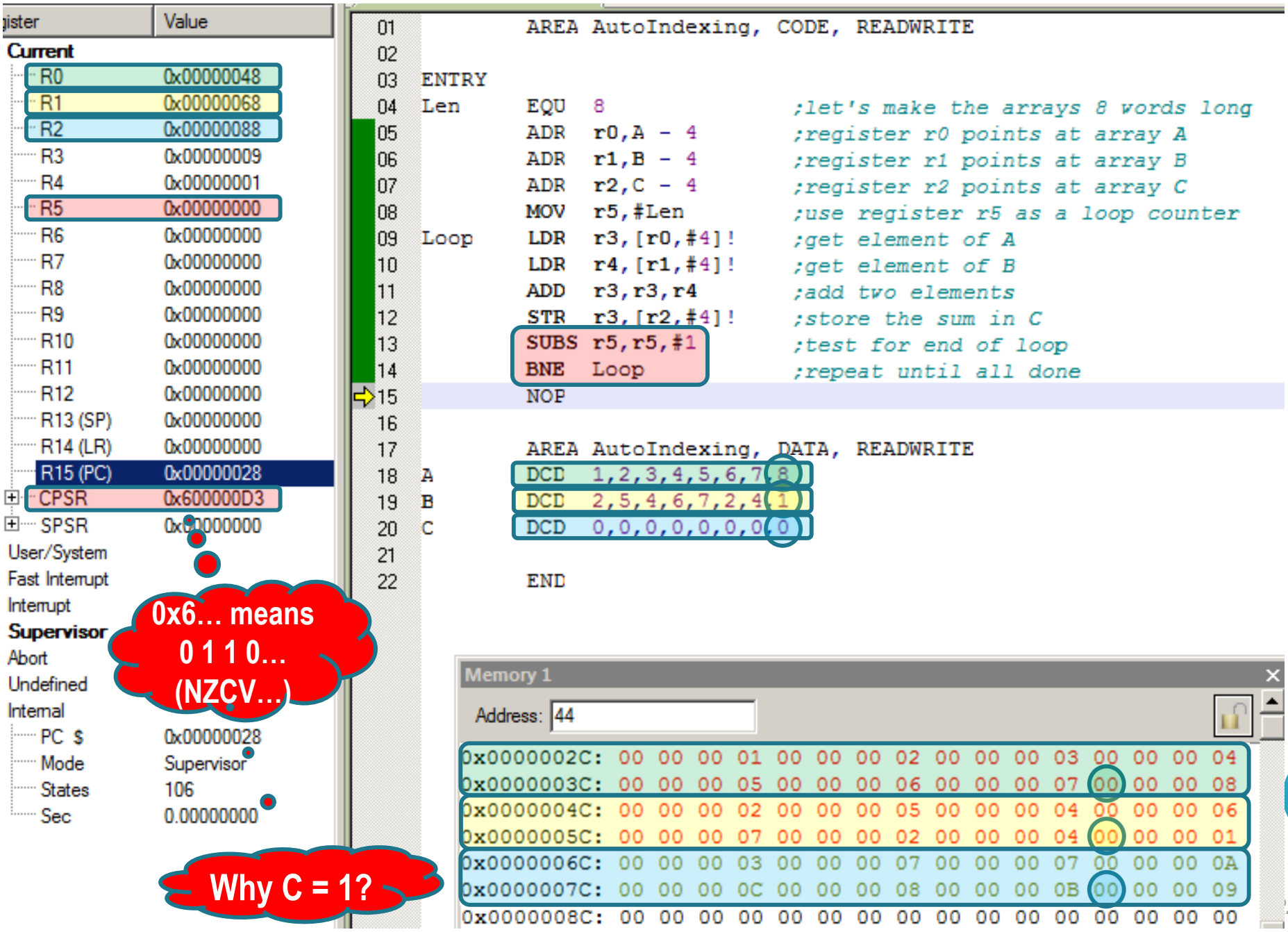
it would increment before using point, and it is why -4 is needed above

and it will automatically update the value.

C[i] = a[i] + b[i].

counter.

Auto-indexing Pre-indexed Addressing Mode



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
```

0xE4910008

post index.

- 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
```

1 1 1 0 | 0 1 | 0 0 | 1 | 0 0 1 | 0 0 0 1 | 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0 0 0 0

AL- | L/S | post | imfd | LDR | base | desti | #8

immediate | no writeback | increment

LDR r0, [r1], #8.

Auto-indexing Post-indexed Addressing Mode

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

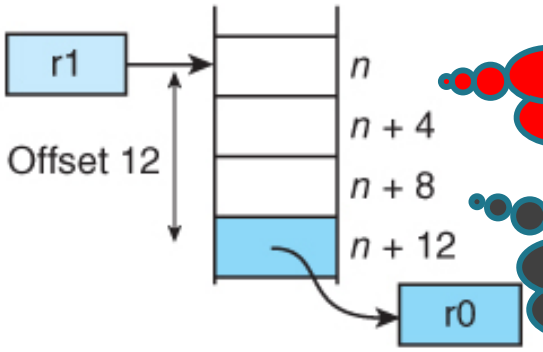
```
Len  EQU  8              ;let's make the arrays 8 words long
     ADR  r0,A          ;register r0 points at array A
     ADR  r1,B          ;register r1 points at array B
     ADR  r2,C          ;register r2 points at array C
     MOV  r5,#Len       ;use register r5 as a loop counter

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
```

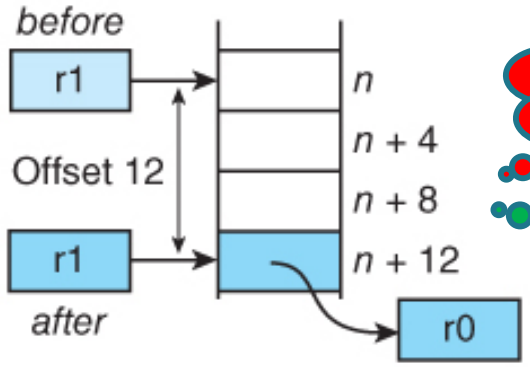
Since the pointer is incremented after used, so no need of -4 this time.

Register Indirect Addressing with Offset



Adjust the pointer then use the adjusted pointer.

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



Adjust the pointer then use the adjusted pointer.

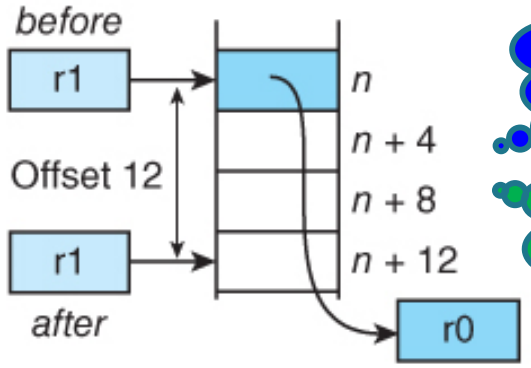
Write-back (update) the adjusted pointer.

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

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

the pointer would never updated and the program would reapeatly load the first elem.

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



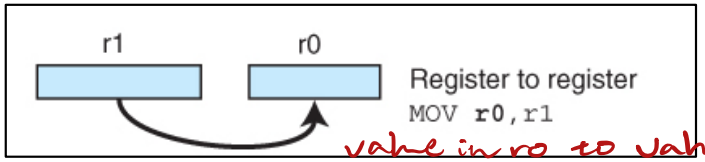
Use the original base pointer then adjust the pointer.

Write-back (update) the adjusted pointer.

LDR r0, [r1, #12]: post index, not write-back
LDR r0, [r1, #12]!: pre index, write back
LDR r0, [r1], #12: post index, write back.

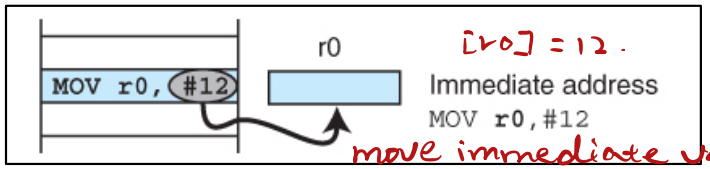
(c) `LDR r0,[r1], #12`
Effective address specified by base register. Operand accessed at effective address. Offset added to base register after the access.

Summary of ARM Addressing Modes



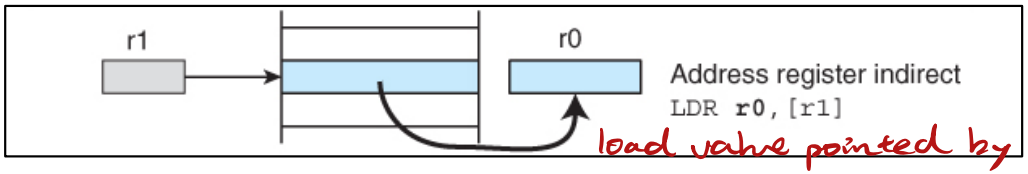
value in r0 to value in r1.

[r0] = [r1].



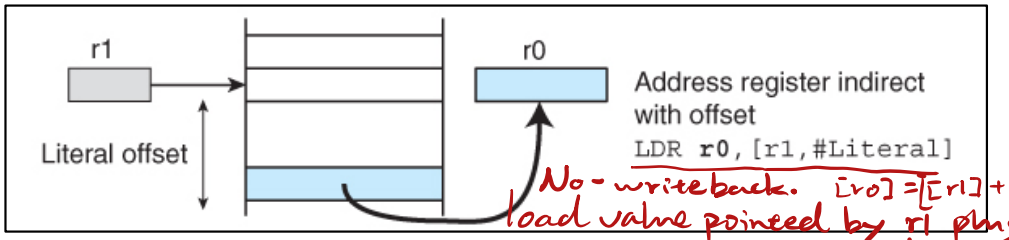
[r0] = 12.

move immediate value to r0.



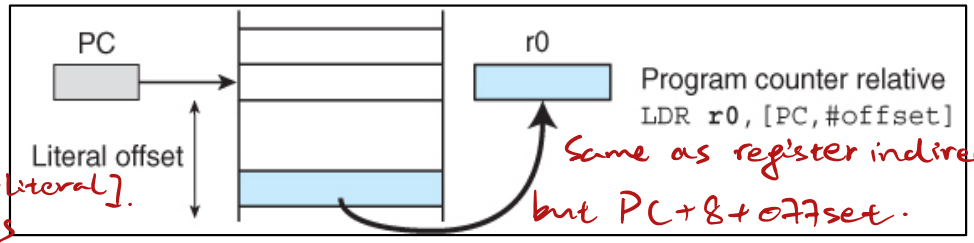
r1 stores an address.

load value pointed by r1 to r0 [r0] = [[r1]].



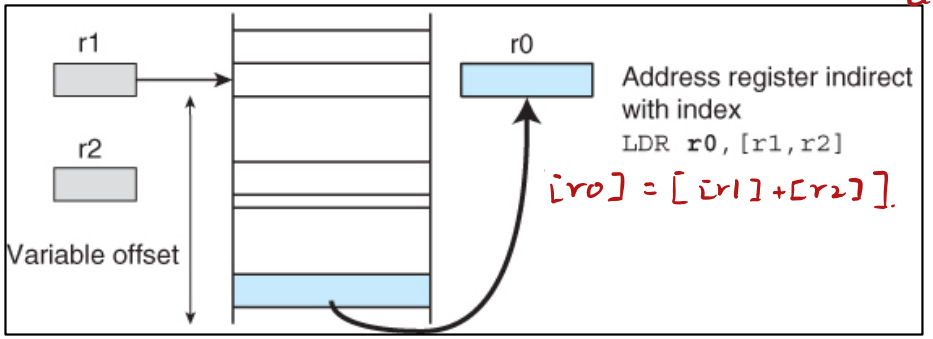
No - writeback. [r0] = [r1] + literal].

load value pointed by r1 plus Literal



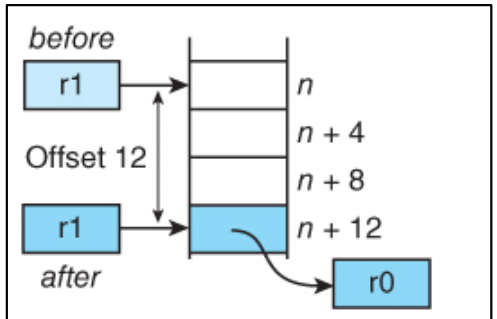
Same as register indirect but PC + 8 + offset.

[r0] = [[PC] + 8 + offset].

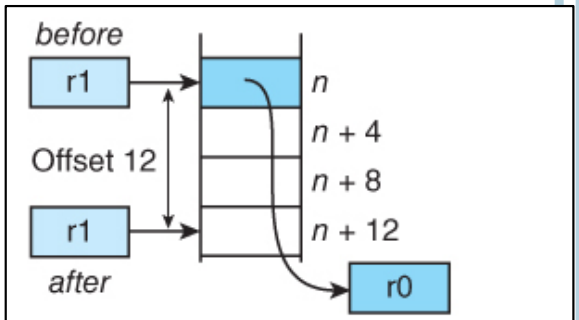


[r0] = [[r1] + [r2]].

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



(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.

[r1] = [r1] + 12
[r0] = [[r1]]

[r0] = [[r1]]
[r1] = [r1] + 12.