The Assembler---Practical Consideration

- The ***DCD***, ***DCW***, or ***DCB*** directives tell the assembler to
  - *reserve* one or more *32-bit*, *16-bit*, or *8-bit* of storage in memory, respectively
    - The memory location used is the next location in sequence,
    - *In case of DCD or DCW, the used location must be on the 32-bit word boundary, or16-bit word boundary, respectively;*
      - *if not, the assembler will insert byte(s) with value of zero to insure that the data location is on the appropriate boundary*
  - *load* whatever value(s) to the right of ***DCD***, ***DCW***, or ***DCB*** into these location(s).
  - *advance* the ***location counter*** by one or more *four*, *two*, or *one* bytes, respectively, so that the next instruction/data will be put in the next place in memory.
- The ***Location Counter*** is a ***variable inside the assembler*** to ***keep track of memory locations during assembling a program***, whereas the ***Program Counter*** is a ***register*** to ***keep track of the next instruction to be executed*** in a program ***at run time***.
- The ***ALIGN*** directive tells the assembler to *align* the current position to be on the next word boundary, i.e., to start at a multiple of 4 address location, ***(explicit alignment)***

The Assembler---Practical Consideration

```
        AREA Directives, CODE, READONLY
        ENTRY
        MOV  r6,#XX          ;load r6 with 5 (i.e., XX)
        LDR  r7,P1           ;load r7 with the contents at location P1
        ADD  r5,r6,r7        ;just a dummy instruction
        MOV  r0, #0x18       ;angel_SWIreason_ReportException
        LDR  r1, =0x20026    ;ADP_Stopped_ApplicationExit
        SVC  #0x123456       ;ARM software interrupt
XX      EQU  5               ;equate XX to 5
P1      & 0x12345678         ;store hex 32-bit value 0x1345678
P3      DCB 25               ;store the one byte value 25 in memory
YY      DCB 'A'              ;store byte whose ASCII character is A in memory
Tx2     DCW 12342            ;store the 16-bit value 12342 in memory
        ALIGN                ;ensure code is on a 32-bit word boundary
Strg1   DCB "Hello"
Strg2   = "X2", &0C, &0A
Z3      DCW  0xABCD
        END
```

The & sign here is a synonym for DCD

assembler directives are in RED

The = sign here is a synonym for DCB

The Assembler---Practical Consideration

```
P1      &  0x12345678
P3      DCB  25
YY      DCB  'A'
Tx2     DCW 12342
        ALIGN
Strg1   DCB  "Hello"
Strg2   =  "X2", &0C, &0A
Z3      DCW   0xABCD
```

FIGURE 3.17 Allocating data to memory

Disassembly

```
     4:        MOV    r6,#XX          ;load
0x00000000  E3A06005  MOV        R6,#0x00000005
     5:        LDR    r7,P1           ;load r7 with the
0x00000004  E59F700C  LDR        R7,[PC,#0x000C]
     6:        ADD    r5,r6,r7        ;just a dummy ins
0x00000008  E0865007  ADD        R5,R6,R7
     7:        MOV    r0, #0x18       ;angel_SWIreason_
0x0000000C  E3A00018  MOV        R0,#0x00000018
     8:        LDR    r1, =0x20026    ;ADP_Stopped_Appl
0x00000010  E59F1014  LDR        R1,[PC,#0x0014]
     9:        SVC    #0x123456       ;ARM semihosting
0x00000014  EF123456  SWI        0x00123456
0x00000018  12345678  EORNES     R5,R4,#0x07800000
0x0000001C  19413036  STMNEDB    R1,{R1-R2,R4-R5,R12-R13}^
0x00000020  48656C6C  STMMIDA    R5!,{R2-R3,R5-R6,R10-R11,R13-R14}^
0x00000024  6F58320C  SWIVS      0x0058320C
0x00000028  0A00ABCD  BEQ        0x0002AF64
0x0000002C  00020026  ANDEQ      R0,R2,R6,LSR #32
0x00000030  00000000  ANDEQ      R0,R0,R0
```

To be stored as ASCII values

3.18 Allocating data to memory—the memory map

```
000000000018    12          Word 0x12345678
000000000019    34
00000000001A    56
00000000001B    78
00000000001C    19          Byte 25
00000000001D    41          Byte 'A'
00000000001E    30          Half Word 12342
00000000001F    36
000000000020    H           String "Hello"
000000000021    e
000000000022    l
000000000023    l
000000000024    O
000000000025    X           String "X2"
000000000026    2
000000000027    0C          Byte 0x0C
000000000028    0A          Byte 0x0A
000000000029    00            Forced alignment
00000000002A    AB          Half Word 0xABAC
00000000002B    CD
```

This is X, not x

Pseudo instructions

- A *pseudo instruction* is an operation that the programmer can use when writing code.
  - The actual instruction ***does not*** have a ***direct*** machine language equivalent.
    - For example, you ***can't*** write MOV **r0**,#0x12345678 to load register r0 with the 32-bit value 0x12345678 because the instruction is only 32 bits long in total.
    - Instead, you can use LDR **r0**, **=** 0x12345678 *pseudo instruction,*
      Yes, it is = not #
      - the assembler will generate suitable code to carry out the same action.
        - *store the constant* 12345678_{16} in a so-called *literal pool* or *constant pool* somewhere in memory
        - *generates suitable code* to load the stored constant 12345678_{16} to r0

Pseudo instructions

- Another *pseudo instruction* is ADR **r0**, label, which loads the *32-bit address* of the line 'label' into register r0, using the appropriate code generated by the assembler
- The following fragment demonstrates the use of the ADR *pseudo instruction*.

This LDR instruction here is ***NOT*** a pseudo instruction

```
ADR   r1,MyArray  ;set up r1 to point to MyArray
                  ; loads register r1 with the 32-bit address of MyArray
...
LDR   r3,[r1]     ;read an element using the pointer

MyArray DCD  0x12345678 ;the address of this data will be loaded to r1
```

- The programmer does not have to know how the assembler generates suitable code to implement such *pseudo instructions*
- All this is done automatically.

But as a student, you need to know it!!

- This can be realized by utilizing the *program counter relative addressing*

Program Counter Relative Addressing

- Register *indirect relative addressing allows* us to
  - *specify the location of an operand with respect to a register value*.

- `LDR r0,[r1]` specifies that the operand address is in r1
- `LDR r0,[r1, #16]` specifies that the operand is 16 bytes onward from r1.

- Suppose that we use *r15*, i.e., *the PC*, to generate an address and write `LDR r0,[PC,#16]`.
  - The operand is 16 bytes onward from the PC
  - i.e, 8 + 16 = 24 bytes from the current instruction.
    - *The ARM's PC **in most of the cases** is 8 bytes from the current instruction to be executed, due to pipelining* (automatically fetches the next instruction before the current one has been executed).

- *Program counter relative addressing* allows you to generate the address of an operand with respect to the program accessing it.

- If the program and its data are relocated elsewhere in memory, the relative offset does not change.

ADR r4, P3 will use the ADD instruction and the PC value to load the address of P3 in R4.

Pseudo instructions

FIGURE 3.20 Code using pseudoinst...

What is the difference between LDR r4, P3 and ADR r4, P3

To answer all these questions, try to put each pair of instructions in an assembly program and analyze the disassembly result.

Note that there is a different between LDR r4, P3 and LDR r4, = P3

The 1st one will load the *VLAUE* of P3 in R4.

The 2nd one will store the *ADDRESS* of P3 at the literal pool and then load the value of this address in R4.

What will be the generated code if you replaced LDR r4, P3 by ADR r4, P3

What is the difference between LDR r4, = P3 and ADR r4, P3

Note that there is a different between LDR r4, = 0x1234 and LDR r4, = P3

The 1st one will store *0x1234* at the literal pool and then load the value of this address in R4.

The 2nd one will store the *ADDRESS* of P3 at the literal pool and then load the value of this address in R4.

C:\K...

Help

Disassembly

```
     4:                 LDR   r0,=0x12345678
0x00000000  E59F0018  LDR       R0,[PC,#0x0018]
     5:                 ADR   r1,Table
0x00000004  E28F1008  ADD       R1,PC,#0x00000008
     6:                 ADR   r2,Table1
0x00000008  E28F2008  ADD       R2,PC,#0x00000008
     7:                 LDR   r3, = 0xAAAAAAAA
0x0000000C  E59F3010  LDR       R3,[PC,#0x0010]
     8:                 LDR   r4,P3
0x00000010  E59F4004  LDR       R4,[PC,#0x0004]
0x00000014  ABCDDCBA  BLGE      0xFF377304
0x00000018  FFFFFFFF  (???)
0x0000001C  22222222  EORCS     R2,R2,#0x20000002
0x00000020  12345678  EORNES    R5,R4,#0x07800000
0x00000024  AAAAAAAA  BGE       0xFEAAAAD4
0x00000028  00000000  ANDEQ     R0,R0,R0
```

0x00 + 0x08 + 0x18 = 0x20

0x04 + 0x08 + 0x08 = 0x14

0x08 + 0x08 + 0x08 = 0x18

0x0C + 0x08 + 0x10 = 0x24

0x10 + 0x08 + 0x04 = 0x1C

PseudoInst.asm

```
01          AREA ConstPool, CODE, READONLY
02          ENTRY
03
04          LDR   r0,=0x12345678
05          ADR   r1,Table
06          ADR   r2,Table1
07          LDR   r3, = 0xAAAAAAAA
08          LDR   r4,P3
09
10  Table   DCD   0xABCDDCBA
11  Table1  DCD   0xFFFFFFFF
12  P3      DCD   0x22222222
```

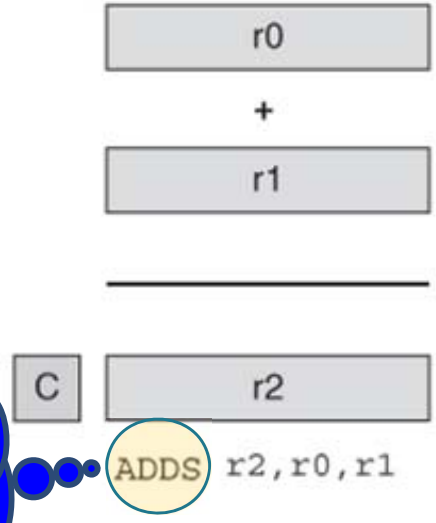
ARM's Data-Processing Instructions (Arithmetic Instructions)

Addition	ADD
Subtraction	SUB
Negation	NEG
Comparison	CMP
Multiplication	MUL
Bitwise logic operations	AND, OR, EOR
Shift operations	LSL, LSR, ASR, ROR, RRX

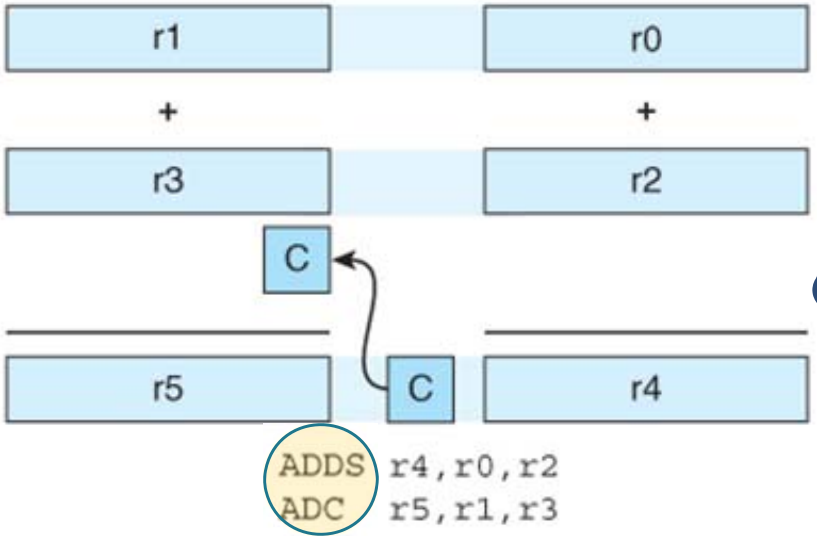
ARM's Data-Processing Instructions (Arithmetic Instructions: Addition)

- A simple ADD (and ADDS) instruction adds two 32-bit values located in registers.
- ARM also has an ADC (add with carry), as well as ADCS, that adds two registers together with the carry bit.
  - This allows extended precision arithmetic as Figure 3.21 demonstrates.

FIGURE 3.21 Single- and extended-precision addition



(a) Single-precision addition. When r0 is added to r1, the result is loaded into r2, and the carry bit is loaded into the carry flag.



(b) Double-precision extended addition. When r0 is added to r2, any carry out is stored in the the carry bit. When r1 is added to r3, the carry bit is added to their sum. In other words, the carry out generated by `ADDS r4,r0,r2` becomes the carry in used by `ADC r5,r1,r3`.

It is ADDS, not just ADD

Can be extended to integers of arbitrary length

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

- Beside the *normal subtraction* (SUB), ARM also provides *reverse subtraction* (RSB)
  - `SUB r1,r2,r3  ;[r1]←[r2]-[r3]`
  - `RSB r1,r2,r3  ;[r1]←[r3]-[r2]`

- RSB is useful, as ARM treats its operands differently.
  - For example, to perform `[r1] ← 10 –[r2]`, you can **not** use
    `SUB r1, #10, r2  ; THIS IS WRONG`
    instead, you can use
    `RSB r1, r2, #10  ; CORRECT`

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

- Note that

```
RSB r1,#5
```

means

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

```
ADD r1, #5
```

means

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

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

- Negation is to subtract a number from 0 (*arithmetic complement, i.e., 2's complement*)
  - ARM does not have a negation instruction as such
  - Instead, ARM provides a *pseudo instruction* called `NEG`

    `NEG r1,r2`
  - The `RSB` instruction is utilized to implement `NEG`
    - To negate r2 (i.e., calculating 0 – [r2]) and store the result in r1,

      `NEG r1,r2`

      or

      `RSB r1,r2,#0`
    - To negate r2 (i.e., calculating 0 – [r2]) and store the result in r2,

      `NEG r2,r2`

      or

      `RSB r2,r2,#0`

      Or simple

      `RSB r2,#0`

The end effect as if multiplying the operand by -1

Can not be shortened to `NEG r2`

ARM's Data-Processing Instructions (Arithmetic Instructions: Move and Move NOT)

- ARM provides a MOV instruction that copies the value of an operand into the other operand
  - To copy the content of r1 to r0,

```
MOV r0, r1
```

- ARM also provides MVN (*move* ***not***) that takes the value of an operand, performs a bitwise ***logical NOT*** operation on the value (i.e., flipping each zero to one and each one to zero), and places the result into the other operand
  - To copy the ***logical complement*** of the content of r1 to r0,

```
MVN r0, r1
```

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

- Comparisons can be *implicit* or *explicit*
- Both implicit and explicit comparisons modify the contents of *the condition code register (CCR)*, a.k.a. *current program status register* (***CPSR***), which is later can be tested to determine whether execution continues in sequence or a branch is taken
  - Example of *implicit* comparison

```
SUBS r1,r1,#1
```

  - Example of *explicit* comparison

```
CMP r1,r2
```

This instruction will evaluate `r1 – r2` *without storing the result*, and set the *condition code register*

```
        CMP  r1,r2        ;is r1 = r2?
        BEQ  DoThis       ;if equal then goto DoThis
        ADD  r1,r1,#1     ;else add 1 to r1
        B    Next         ;jump past the then part
        .
DoThis  SUB r1,r1,#1      ;subtract 1 from r1
Next    ...               ;both forks end up here
```