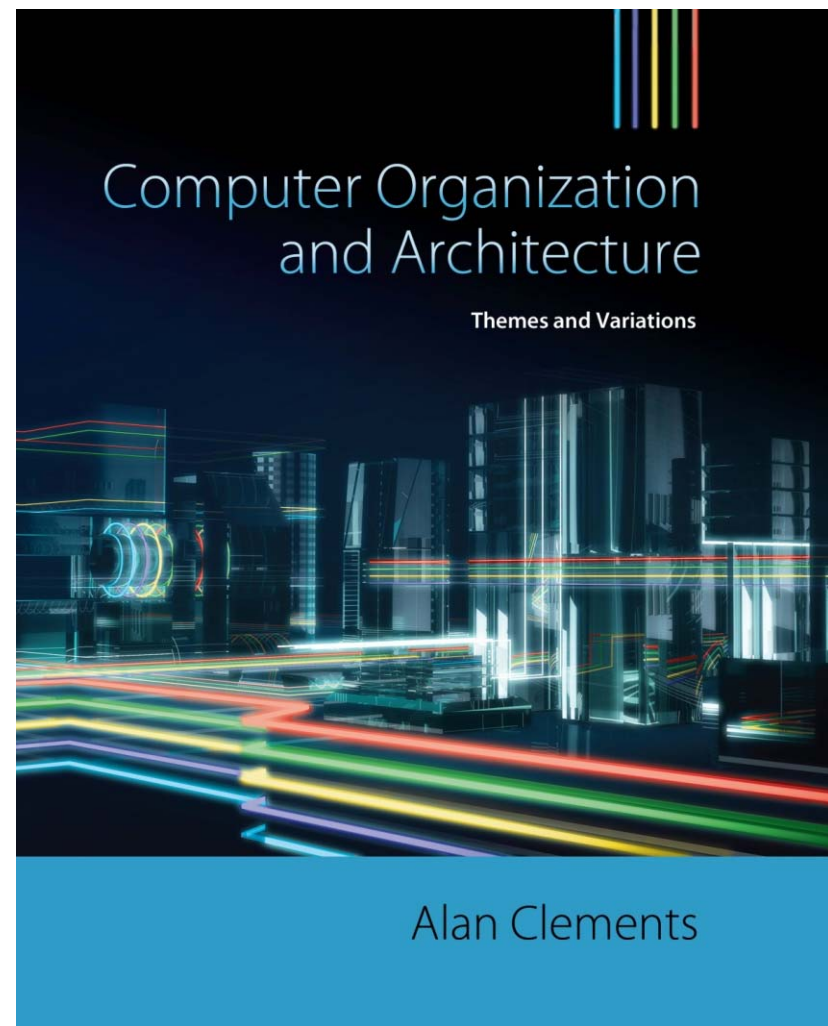


# Part 0x2

## 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.

# Sample ARM Assembly Instructions

*at least one  
white space here.*  
**LDR** **r0**, **address** **Load** the contents of the memory-location at **address** into register **r0**.  
*free to  
add white space  
or not.*

**We will talk about the format of **address** later on.**

**STR** **r0**, **address** **Store** the contents of register **r0** at the specified **address** in memory.

*r0 = r1 + r2*  
**ADD** **r0**, **r1**, **r2** **Add** the contents of register **r1** to the contents of register **r2** and store the result in register **r0**.  
*ADDS.*

*r0 = r1 - r2.*  
**SUB** **r0**, **r1**, **r2** **Subtract** the contents of register **r2** from the contents of register **r1** and store the result in register **r0**.

*to see the flags:  
SUBS*  
**BPL** **target** **If** the result of the previous operation was **plus** (**+ve** or **zero**) **then** branch to the instruction at address **target**.  
*N flag = 0*

**BEQ** **target** **If** the result of the previous operation was **zero**, **then** branch to the instruction at address **target**.  
*Z flag = 1*

**We will talk about the format of **target** later on.**

**B** **target** **Branch unconditionally** to the instruction stored at the memory address **target**.

**Note the number of operands in each instruction.**

## Example 1: Conditional Operation

Label  
to identify  
certain cases.

$r5 = r5 - 1.$

SUBS **r5,r5,#1** ;Subtract 1 from r5

BEQ **onZero** ;IF zero THEN go to the line labeled 'onZero'

ADD **r1,r2,r3** ;ELSE continue from here

$r1 = r2 + r3.$

$z=0$

notZero

$z=1$

**onZero** SUB **r1,r2,r3** ;Here's where we end up if we take the branch

### Explanation

#### SUBS r5,r5,#1

- ❑ subtracts 1 from the contents of register r5.
- ❑ After completing this operation the number remaining in r5 may, or may not, be zero.

#### BEQ onZero

- ❑ forces a branching (i.e., goto) to the line labeled '**onZero**' if the outcome of the last operation was zero.
- ❑ Otherwise, the next instruction in sequence after the BEQ is executed.

## Example 2: Conditional Operation

$P \geq Q$   
is the same as  
 $P - Q \geq 0$

```
IF P ≥ Q THEN X = P + 5
ELSE X = P + 20
```

*define P, Q.*

```
LDR r0, P ;Load r0 with the contents in location P
```

```
LDR r1, Q ;Load r1 with the contents in location Q
```

```
SUBS r2, r0, r1 ;Subtract the contents of Q from P
```

```
BPL THEN ;IF P - Q ≥ 0 then execute the 'THEN' part
```

```
ADD r0, r0, #20 ;ELSE Add 20 to the contents of r0 to get P + 20
```

```
B EXIT ;Skip past 'THEN' part to 'EXIT'
```

```
THEN ADD r0, r0, #5 ;Add 5 to r0 to get P + 5
```

```
EXIT STR r0, X ;Store r0 in memory-location X
```

```
STOP
```

*P is 4 bytes and initialized by 12.*

```
P DCD 12
```

```
Q DCD 9
```

```
X DCD 0
```

*;These three lines reserve memory space for  
the three operands P, Q, X and initialize them.*

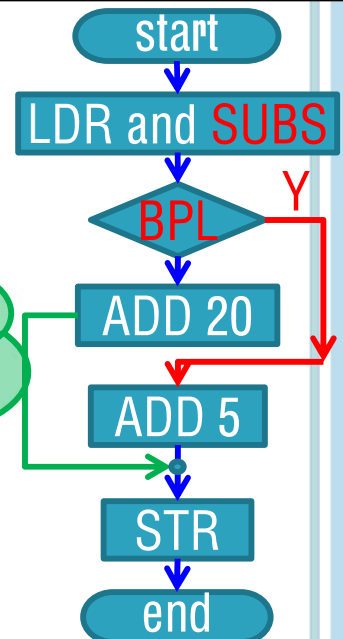
*;The memory-locations are 36, 40, and 44, respectively.*

*Define P, Q and X.*

**DCD means, Define Constant Data**

*reserve 4 bytes for each location.*

This is an unconditional branching that prevents the following instruction from being executed.



Here's where the test and conditional branching take place

## Example 2: Conditional Operation

IF  $P \geq Q$  THEN  $X = P + 5$   
ELSE  $X = P + 20$

*set flags.*

LDR  $r0, P$  ;[r0]  $\leftarrow [P]$

LDR  $r1, Q$  ;[r1]  $\leftarrow [Q]$

SUBS  $r2, r0, r1$  ;[r2]  $\leftarrow [r0] - [r1]$

BPL THEN ;IF  $[r2] \geq 0$  [PC]  $\leftarrow$  THEN

ADD  $r0, r0, \#20$  ;[r0]  $\leftarrow [r0] + 20$  → no brackets indicate it is a constant.

B EXIT ;[PC]  $\leftarrow$  EXIT

THEN ADD  $r0, r0, \#5$  ;[r0]  $\leftarrow [r0] + 5$

EXIT STR  $r0, X$  ;[X]  $\leftarrow [r0]$

STOP

P DCD 12 ;[P] = 12

Q DCD 9 ;[Q] = 9

X DCD 0 ;[X] = 0

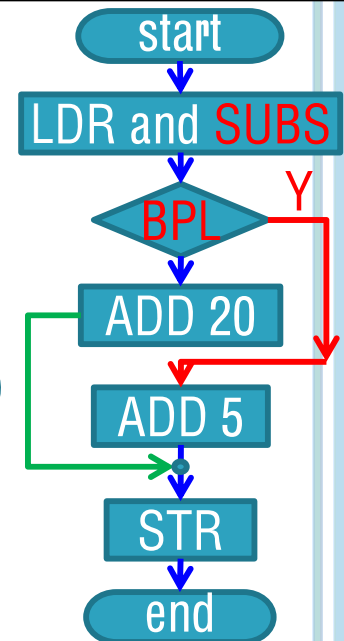
*copy the value of P, Q into registers*

*subtract registers and get flags*

*check flags.*

How are the locations P, Q, and X calculated?

Same example,  
but with  
RTL comments



## Example 2: Conditional Operation

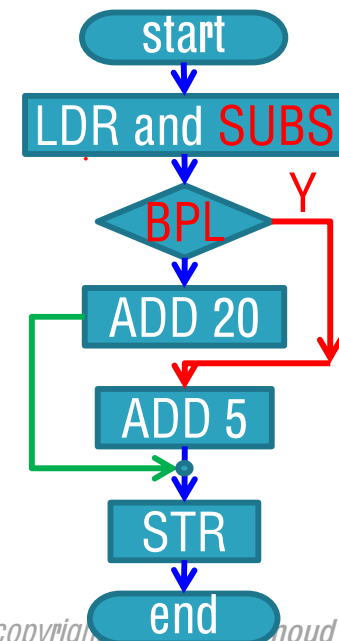
Case 1:  $P = 12$ ,  $Q = 9$ , and hence the conditional branching is *taken* (i.e., will branch to *THEN*)

0	LDR	r0,36	
4	LDR	r1,40	
8	SUBS	r2,r0,r1	
12	BPL	24	
16	ADD	r0,r0,#20	skipped.
20	B	28	
24	ADD	r0,r0,#5	Next instruction
28	STR	r0,44	
32	STOP		
36		12	P
40		9	Q
44			X

LDR	r0,P	;[r0] ← [P]
LDR	r1,Q	;[r1] ← [Q]
SUBS	r2,r0,r1	;[r2] ← [r0] - [r1]
BPL	THEN	;IF [r2] ≥ 0 [PC] ← THEN
ADD	r0,r0,#20	;[r0] ← [r0] + 20
B	EXIT	;[PC] ← EXIT
THEN	ADD	r0,r0,#5 ;[r0] ← [r0] + 5
EXIT	STR	r0,X ;[X] ← [r0]
	STOP	
P	DCD	12
Q	DCD	9
X	DCD	0

Program counter

$PC_{start} = 0, PC_{end} = 4, r0 = 12$   
 $PC_{start} = 4, PC_{end} = 8, r1 = 9$   
 $PC_{start} = 8, PC_{end} = 12, r2 = 3$   
 $PC_{start} = 12, PC_{end} = 24,$   
 $PC_{start} = 24, PC_{end} = 28, r0 = 17$   
 $PC_{start} = 28, PC_{end} = 32, X = 17$





## Example 2: Conditional Operation

Case 2:  $P = 12$ ,  $Q = 14$ , and hence the conditional branching is *not taken* (i.e., will **NOT** branch to *THEN*)

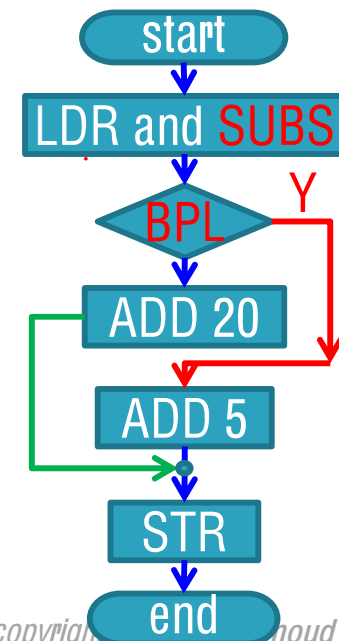
0	LDR	r0,36
4	LDR	r1,40
8	SUBS	r2,r0,r1
12	BPL	24
16	ADD	r0,r0,#20
20	B	28
24	ADD	r0,r0,#5
28	STR	r0,44
32	STOP	
36		12
40		14
44		

Next instruction

directly branch to 24.

$PC_{start} = 0, \quad PC_{end} = 4, \quad r0 = 12$   
 $PC_{start} = 4, \quad PC_{end} = 8, \quad r1 = 14$   
 $PC_{start} = 8, \quad PC_{end} = 12, \quad r2 = -2$   
 $PC_{start} = 12, \quad PC_{end} = 16,$   
 $PC_{start} = 16, \quad PC_{end} = 20, \quad r0 = 32$   
 $PC_{start} = 20, \quad PC_{end} = 28,$   
 $PC_{start} = 28, \quad PC_{end} = 32, \quad X = 32$

LDR	r0,P	;[r0] ← [P]
LDR	r1,Q	;[r1] ← [Q]
SUBS	r2,r0,r1	;[r2] ← [r0] - [r1]
BPL	THEN	;IF [r2] ≥ 0 [PC] ← THEN
ADD	r0,r0,#20	;[r0] ← [r0] + 20
B	EXIT	;[PC] ← EXIT
ADD	r0,r0,#5	;[r0] ← [r0] + 5
STR	r0,X	;[X] ← [r0]
STOP		
P	DCD	12
Q	DCD	14
X	DCD	0



## Example 3: Conditional Operation

□ Consider the code needed to calculate  $1 + 2 + 3 + 4 + \dots + 20$

LD R: copy the content of the memory to register

The **MOV** instruction copies the value of Operand2 (can be a lateral or a register) into the destination register.

```

MOV r0,#1      ;Put 1 in register r0 (the counter)
MOV r1,#0      ;Put 0 in register r1 (the sum)
Next ADD r1,r1,r0 ;REPEAT: Add current counter to sum
  ADD r0,r0,#1  ; Add 1 to the counter (i.e., increment counter)
  CMP r0,#21   ; Have we added all 20 numbers?
  BNE Next     ;UNTIL we have made 20 iterations
  STOP        ;If we have then stop

```

when  $r0 - 21 = 0$

**MOV** and **CMP** instructions need ONLY two operands.

**CMP** compares the value in a register with *Operand2*, i.e., subtracting *Operand2* from the register value.

It automatically updates the condition flags on the result, but does **not** place the result in any register.

The "**S**" is **not** needed in such instruction.



# General-Purpose Registers

❑ Computers might have

- *general-purpose registers*    *no restriction*
- *special-purpose* (dedicated) registers

❑ Registers usually have the same width as the fundamental word of a computer.

*r = 32 bits*

❑ The **ARM** processors have

- general-purpose registers, and
- two special purpose registers (have special hardware-defined functions)

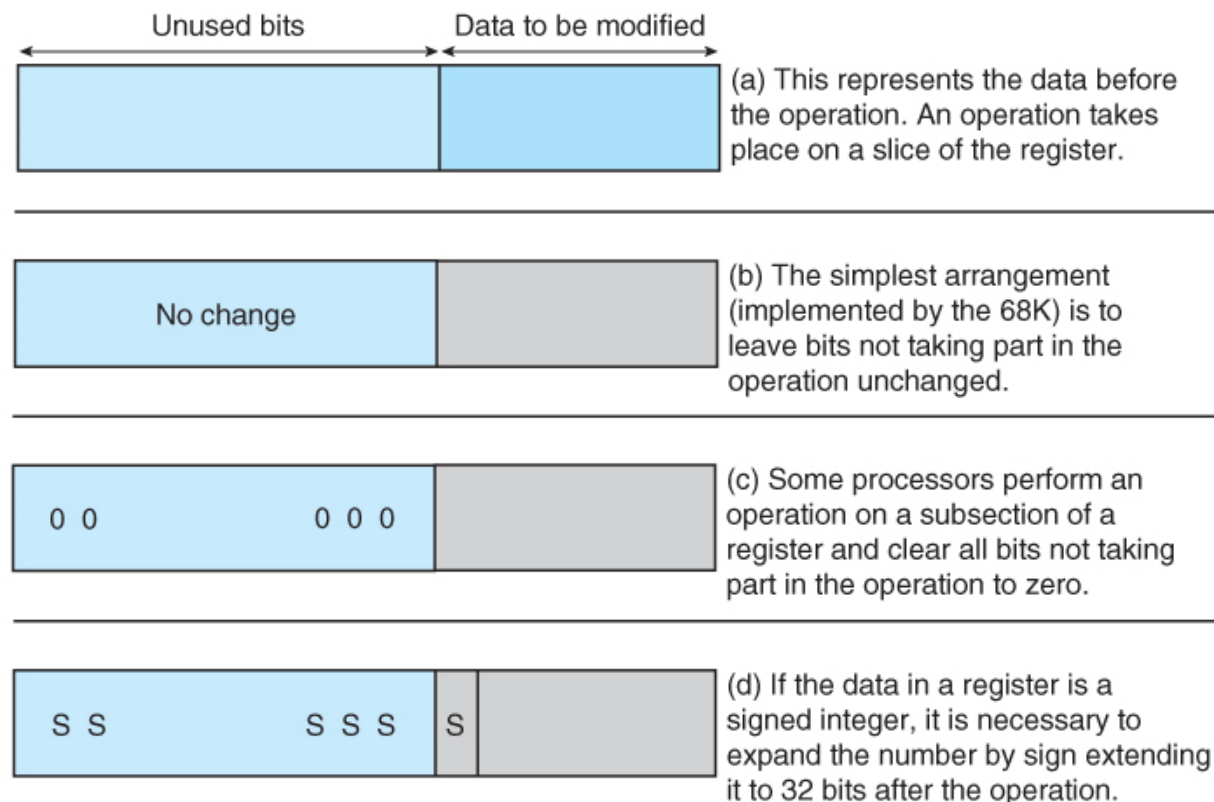
*① program counter => point to the next instruction to be executed*

*② linked register*

## Data Extension

- ❑ Sometimes registers hold data values smaller than their actual length
  - for example, a 16-bit (halfword in a 32-bit word register).
- ❑ What happens to the other bits? (*processor dependent*)
  - some leave the unused bits unchanged,
  - some set the unused bits to 0, and
  - some sign-extend the 16-bit halfword to 32-bits (*two's complement*)

**FIGURE 3.9** Operations on a subsection of a register



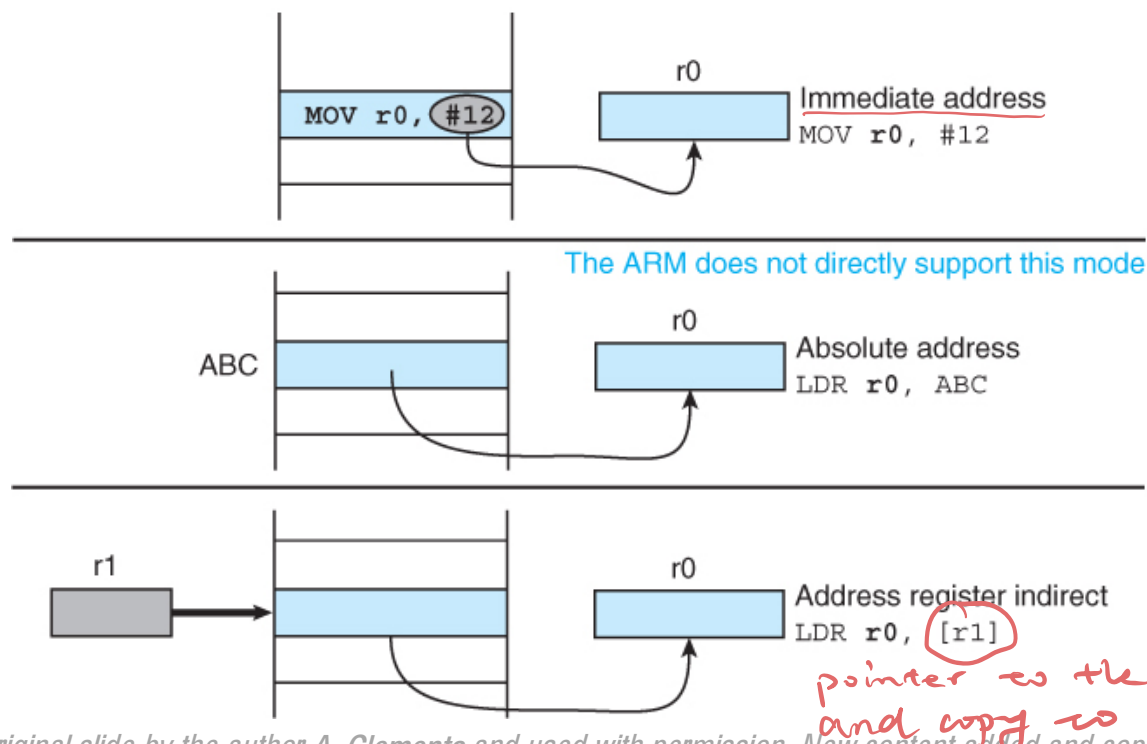
© Cengage Learning 2014

# Addressing Modes

- There are three fundamental addressing modes
  - *Literal or immediate*
    - the actual value is part of the instruction
  - *Direct or absolute*
    - the instruction provides the memory address of the operand
    - *The ARM architecture does not directly support this mode*
  - *Register indirect or pointer based or indexed*
    - a register contains the address of the operand

FIGURE 3.10

Progressive sequence of addressing modes



# Instruction Types

- ❑ **Memory-to-register**
  - The source operands are in memory and
  - the destination operand is in a register
- ❑ **Register-to-memory**
  - The source operands are in registers and
  - the destination operand is in memory
- ❑ **Register-to-register**
  - Both operands are in registers.

**CISC means COMPLEX Instruction Set Computer**

- ❑ **CISC processors** like the Intel IA32 family and Motorola/Freescale 68K family *allow* memory-to-register and register-to memory *data-processing* operations.

**RISC means REDUCED Instruction Set Computer**

- ❑ **RISC processors** like the ARM and MIPS *allow* only register-to-register *data-processing* operations.
- ❑ **RISC processors** *have* a special LOAD and a special STORE instructions (pseudo instructions) *to transfer data between memory and a register using Register indirect addressing mode.*

# Operands and Instructions

- ❑ **CISC** processors *typically* have
  - **Two-address** instructions where
    - **one** address is **memory** and the **other** is a **register**.
- ❑ **RISC** processors *typically* have a
  - **three-address** data processing instruction where
    - **the three** operand addresses **are registers**.
  - They *also have* **two** special **two-address** instructions,
    - **LOAD** and **STORE**.

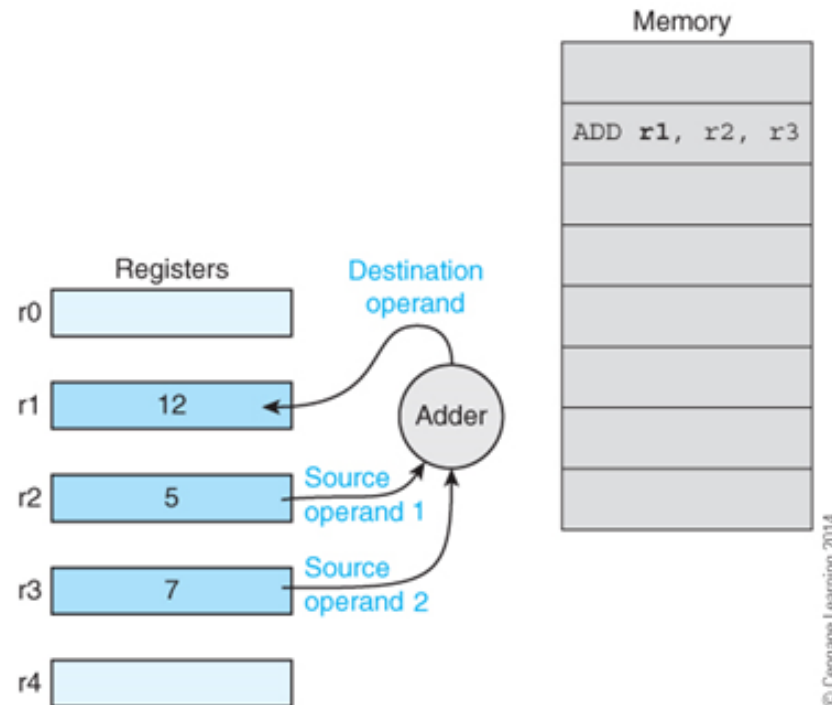
## Three Address Machines

- ❑ Processors *do not* implement *three memory address* instructions.
- ❑ A typical **RISC** processor *allows three register addresses* in an instruction
  - For example:

ADD r1,r2,r3      ;Add r2 to r3 and put the result in r1

FIGURE 3.11

The three address instruction





## What is ARM architecture?

- ❑ The **ARM** architecture is the intellectual property of **ARM Holdings**, based in Cambridge, England.
- ❑ The company was *founded in 1990* as **Advanced RISC Machines (ARM)** by
  - Acorn Computers,
  - Apple Computers, and
  - VLSI Technology.
- ❑ The *1<sup>st</sup>-generation* of **ARM** was *8-bit* microprocessors.
- ❑ The *2<sup>nd</sup>-generation* of **ARM** was *32-bit* microprocessors.
  - In **ARM** terminology, **16 bits is a half-word**, and **32 bits is a word**.
- ❑ *There has been remarkably little change in instruction set architecture between today's high performance machines and 1<sup>st</sup>-generation microprocessors.*
- ❑ Unlike other microprocessor manufactures, e.g., Intel, AMD, and Freescale, **ARM** does **NOT build chips**, but **licenses to semiconductor companies** its core processors for use in *systems on chips* and *microcontrollers*.
- ❑ **ARM** successfully targeted the world of mobile devices, e.g., netbooks, tablets, and cell phones.
- ❑ **ARM** is a machine with **register-to-register architecture**, as well as **load/store instructions** that move data between memory and registers.
- ❑ **ARM operand values** are *32-bit wide*, except for several multiplication instructions that generate a 64-bit product stored in two 32-bit registers.

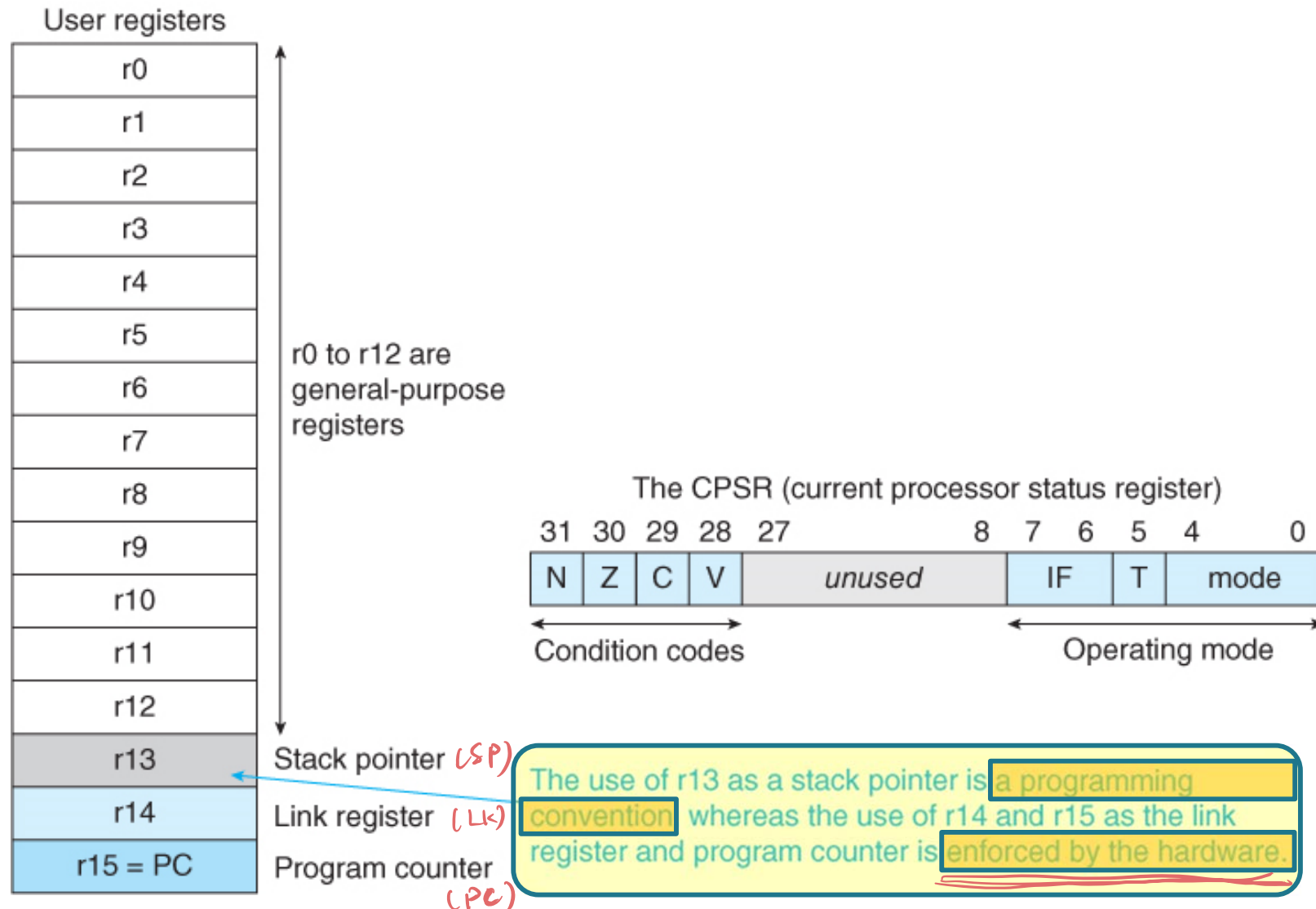
## ARM Register Set

- ❑ The **ARM** processor has
  - 16 **32-bit** registers (r0, r1, r2, ..., r12, r13, r14, r15)
    - r0, r1, r2, ..., r12 are general-purpose registers
    - r15 is the program counter
    - r14 is the link register—holds subroutine return addresses
    - r13 is *reserved* for use by the programmer as the stack pointer  
*other register can be stack pointer as well*
- ❑ Sixteen registers require a 4-bit address. . . **Review Slide 3 in Chapter 2.**
  - saves three bits per instruction (1 bit per operand) over **RISC** processors with 32-register architectures (5-bit address).  
**Condition Code Register**
- ❑ The **ARM**'s *current program status register* (**CPSR**) contains
  - **Condition codes** (bits number 31, 30, 29, and 28)  
**N** (negative), **Z** (zero), **C** (carry) and **V** (overflow) flag bits
  - **Operating mode** (bits number 0–7)  
May talk about them later
- ❑ **ARM** processors have a rich instruction set

# ARM Register Set

FIGURE 3.12

ARM register set



# Typical ARM Instructions

TABLE 3.1

ARM Data Processing, Data Transfer, and Compare Instructions

Instruction	ARM Mnemonic	Definition
Addition	ADD <b>r0</b> , r1, r2	$[r0] \leftarrow [r1] + [r2]$
Subtraction	SUB <b>r0</b> , r1, r2	$[r0] \leftarrow [r1] - [r2]$
AND	AND <b>r0</b> , r1, r2	$[r0] \leftarrow [r1] \cdot [r2]$
OR	ORR <b>r0</b> , r1, r2	$[r0] \leftarrow [r1] + [r2]$
Exclusive OR	EOR <b>r0</b> , r1, r2	$[r0] \leftarrow [r1] \oplus [r2]$
Multiply	MUL <b>r0</b> , r1, r2	$[r0] \leftarrow [r1] \times [r2]$
Register-to-register move	MOV <b>r0</b> , r1	$[r0] \leftarrow [r1]$
Compare	CMP r1, r2	$[r1] - [r2]$
Branch on zero to label	BEQ label	$[PC] \leftarrow \text{label}$ (jump to label)

© Cengage Learning 2014

# ARM Assembly Language

- ARM instructions are written in the form

{Label} Op-code operand1, operand2, operand3 {comment}

- Consider the following example of a loop.

```

MOV    r1,#0      ;initialize the total
MOV    r2,#10     ;initialize the value to be added
MOV    r7,#20     ;initialize the number of iterations
Test_5 ADD    r1,r1,r2 ;increment total by the value
      SUBS    r7,#1      ;decrement loop counter
                        ;same as SUBS r7, r7, #1
      BNE    Test_5     ;IF not zero THEN goto Test_5
  
```

What are the values of r1, r2, and r7 after executing this loop?

- The Label field is a user-defined label (case-sensitive single word without space) that can be used by other instructions to refer to the address of that line.
- Any text following a semicolon is regarded as a **comment** field which is ignored by the assembler.

## ARM Assembly Language

- ❑ Suppose we wish to generate the sum of the cubes of numbers from 1 to 10. We can use the *multiply and accumulate* instruction;

```
MOV    r0,#0           ;clear total in r0
MOV    r1,#10          ;FOR i = 10 to 1 (count down)
Next MUL r2,r1,r1       ; square the number (i × i)
      r2 = r12
      MLA r0,r2,r1,r0   ; cube the number and add it to total
SUBS   r1,r1,#1        ; decrement counter (set condition flags)
BNE    Next            ;END FOR (branch back on count not zero)
```

- ❑ This fragment of assembly language is *syntactically* correct.
  - **But it is not yet a program that we can run.**
- ❑ We must specify the environment to make it a standalone program.
- ❑ There are two types of statement:
  - *executable instructions* that are executed by the computer and
  - *assembler directives* that tell the assembler something about the environment.



# Structure of an ARM Program

AREA Cubes, CODE, READONLY  
ENTRY

Next    MOV   r0,#0            ;clear total in r0  
         MOV   r1,#10        ;FOR i = 10 to 1  
         MUL   r2,r1,r1       ; square number  
         MLA   r0,r2,r1,r0    ; cube number and add to total  
         SUBS   r1,r1,#1      ; decrement loop count  
         BNE   Next          ;END FOR

END

assembler  
directive

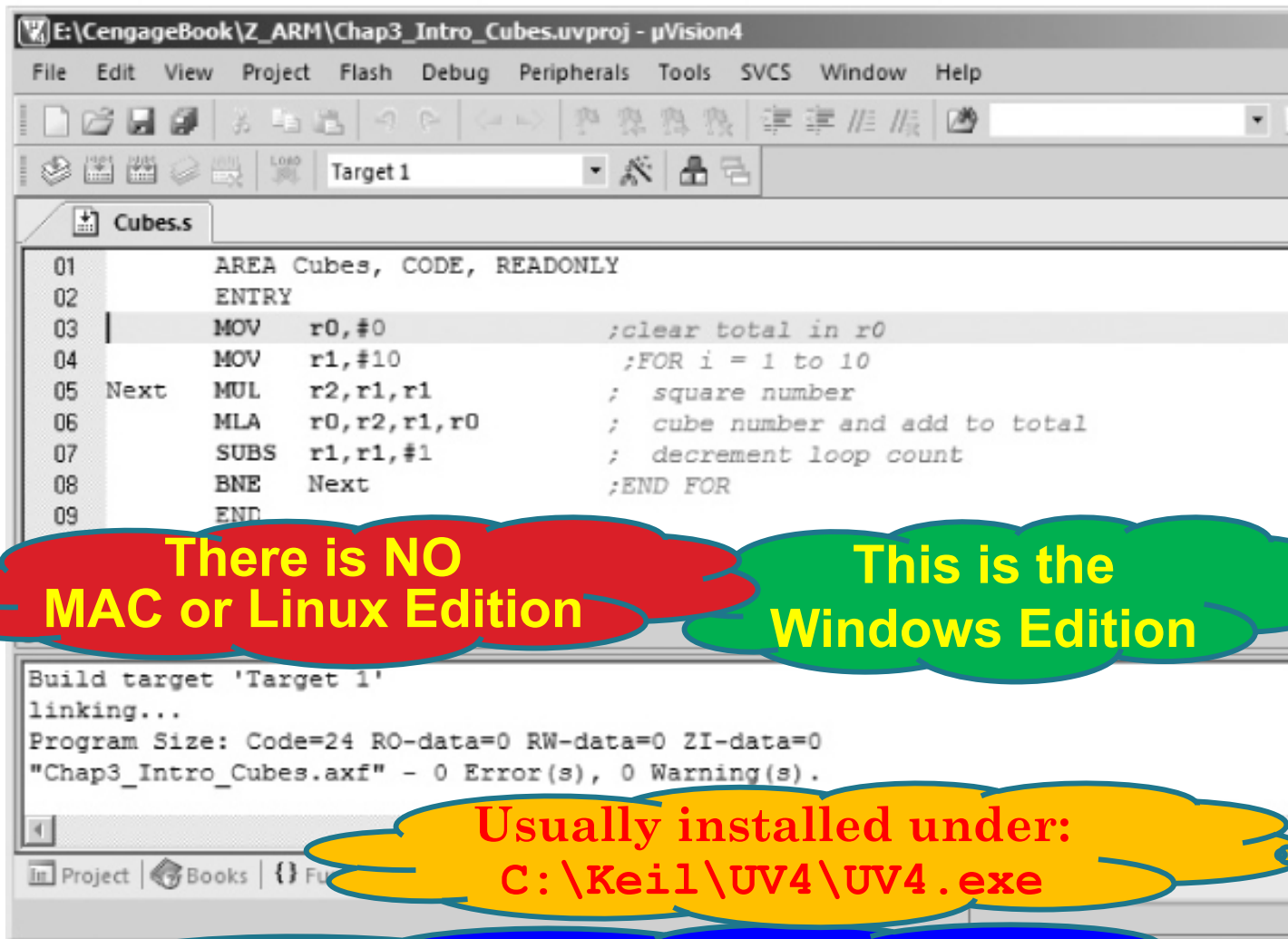
Assembly  
code

assembler  
directive

# Snapshot of the Display of an ARM Development System

**FIGURE 3.13**

Assembling an assembly language program using Kiel's ARM IDE



There is NO  
MAC or Linux Edition

This is the  
Windows Edition

Usually installed under:  
C:\Keil\UV4\UV4.exe

This is MicorVision 4, not 5.

## Project

New µVision Project

Enter file name

Save

Select device for Target

ARM

ARM7 (Big Endian)

Ok

## File

New

Enter assembly program  
(i.e., code  
and  
assembler directives)

## File

Save

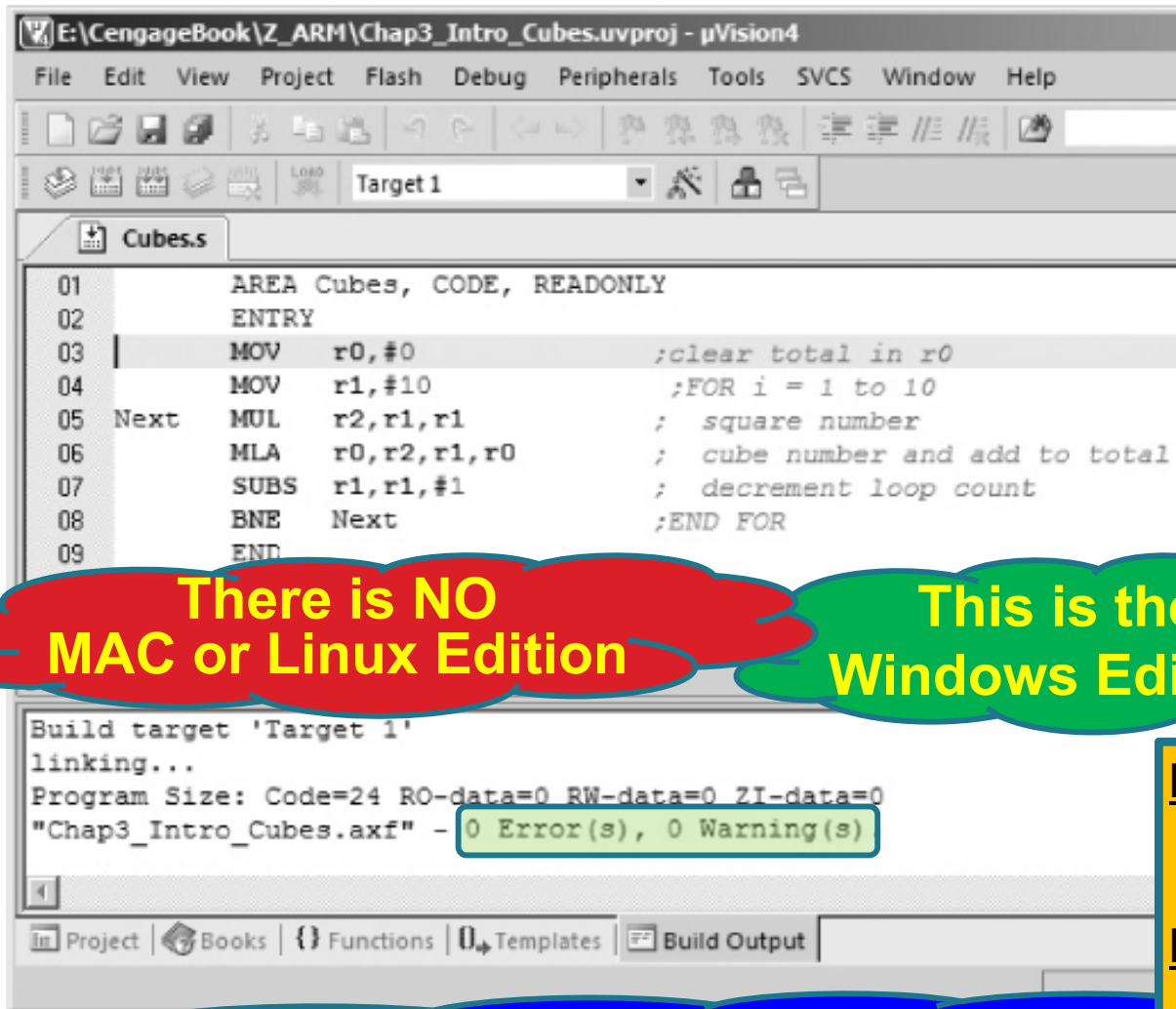
Enter file name  
(to simplify things, use  
.s as an extension

Save

# Snapshot of the Display of an ARM Development System

FIGURE 3.13

Assembling an assembly language program



There is NO  
MAC or Linux Edition

This is the  
Windows Edition

This is MicorVision 4, not 5.

## Project

Manage

Components, Environment, books

Add file

Enter file name

Add

Close

Ok

## Project

Build Target , or simply press F7

If you have **errors** or **warnings**,  
you **have to fix them** before continue.

## Debug

Start/Stop Debug Session

Ok

## Debug

Step, or simply press F11