Part 2

CHAPTER 3

Architecture and Organization
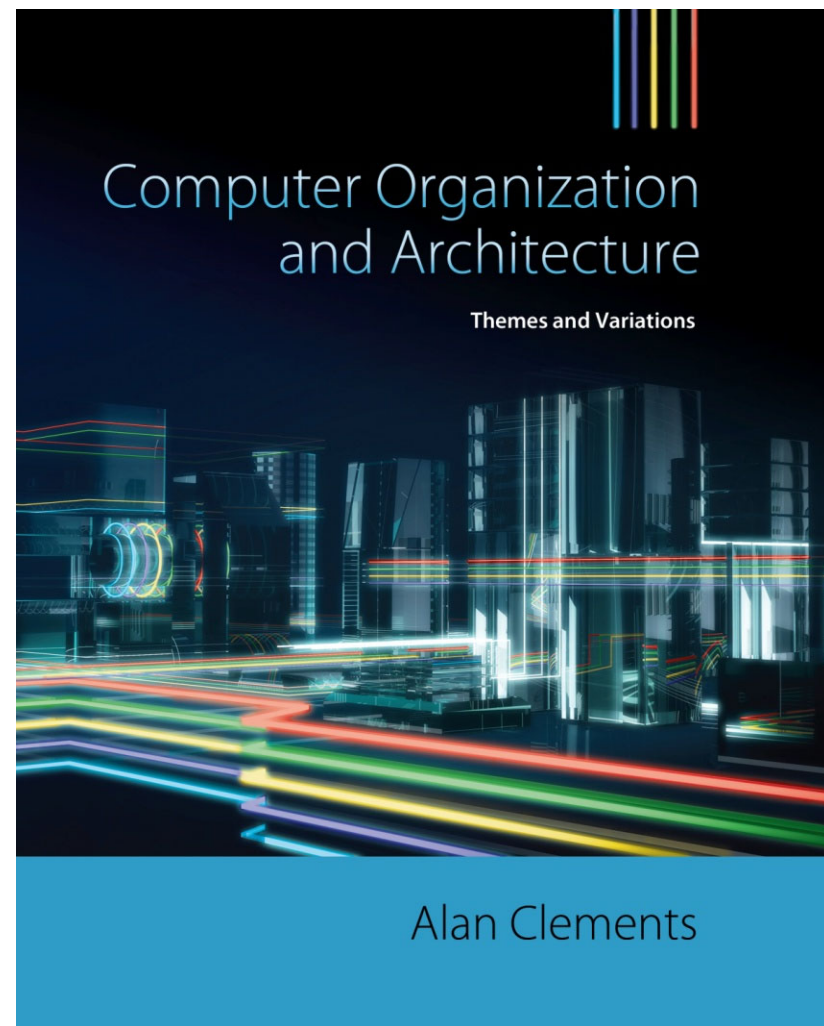
Computer Organization and Architecture
Themes and Variations

Alan Clements

1

CENGAGE Learning™

Sample ARM Assembly Instructions

instruction.

LDR r0,address *Load* the contents of the memory-location at address into register **r0**.

> 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.

won't change flags.

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

ADDS to request flags

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

SUBS to request flags

BPL target *If* the result of the previous operation was *plus* (*+ve* or *zero*) *then* branch to the instruction at address target.

Branch Plus.
→ check flag

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

→ check flag.

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

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

22

> Note the number of operands in each instruction.

Example 1: Conditional Operation

make subtraction and set flags.

```
        SUBS  r5,r5,#1   ;Subtract 1 from r5
        BEQ   onZero     ;IF zero THEN go to the line labeled 'onZero'
notZero ADD   r1,r2,r3   ;ELSE continue from here
  .
  .
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.

23

Example 2: Conditional Operation

P ≥ Q is the same as P − Q ≥ 0

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

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

```
        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       DCD    12          ;These three lines reserve memory space for
Q       DCD    9           ;the three operands P, Q, X and initialize them.
X       DCD    0           ;The memory-locations are 36, 40, and 44, respectively.
```
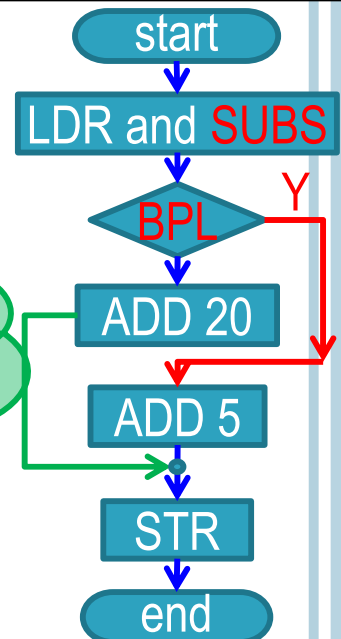
Here's where the test and conditional branching take place

DCD means, **Define Constant Data**

start

LDR and SUBS

BPL   Y

ADD 20

ADD 5

STR

end

24

Example 2: Conditional Operation

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

Same example, but with RTL comments

```
        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          ;[P] = 12
Q       DCD     9          ;[Q] =  9
X       DCD     0          ;[X] =  0
```
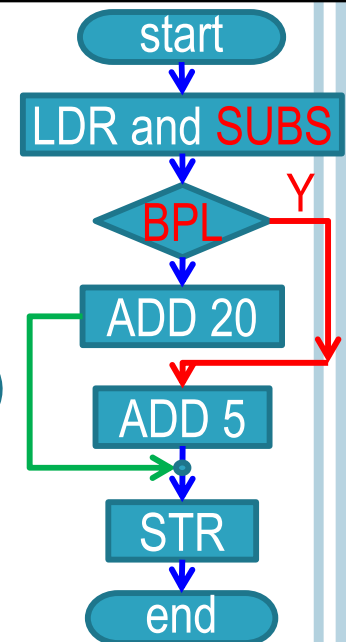
How are the locations P, Q, and X calculated?

start
LDR and SUBS
BPL — Y
ADD 20
ADD 5
STR
end

25

Example 2: Conditional Operation

Case 1:   P = 12, Q = 9, and hence the conditional branching is *taken*
(i.e., will branch to THEN)
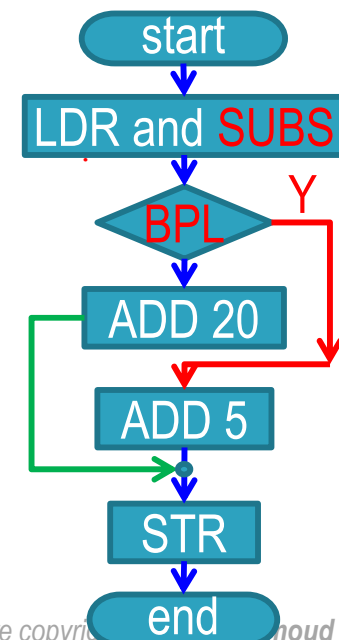
0	LDR	r0,36	
4	LDR	r1,40	
8	SUBS	r2,r0,r1	
12	BPL	24	
16	ADD	r0,r0,#20	
20	B	28	
24	ADD	r0,r0,#5	Next
28	STR	r0,44	instruction
32	STOP		
36	12		P
40	9		Q
44			X

	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	

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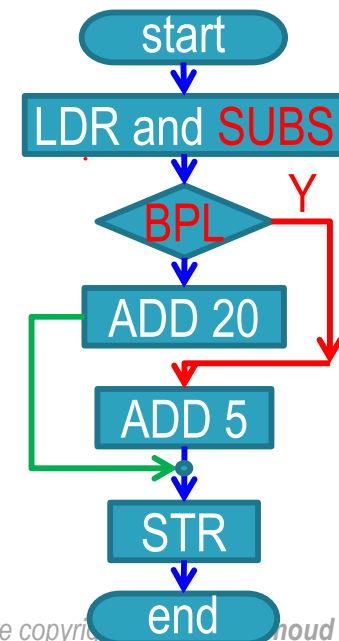

start → LDR and SUBS → BPL → ADD 20 → ADD 5 → STR → end

26

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	P
40	14	Q
44		X

Next instruction

	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	14	
X	DCD	0	

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

start → LDR and SUBS → BPL → ADD 20 → ADD 5 → STR → end

27

Example 3: Conditional Operation

❑ Consider the code needed to calculate 1 + 2 + 3 + 4 + … + 20

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
```

makes it stop at 20

CMP means Subtract the two and set flags.

not bold → there is no storage

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

  o *general-purpose registers*

  o *special-purpose* (dedicated) registers

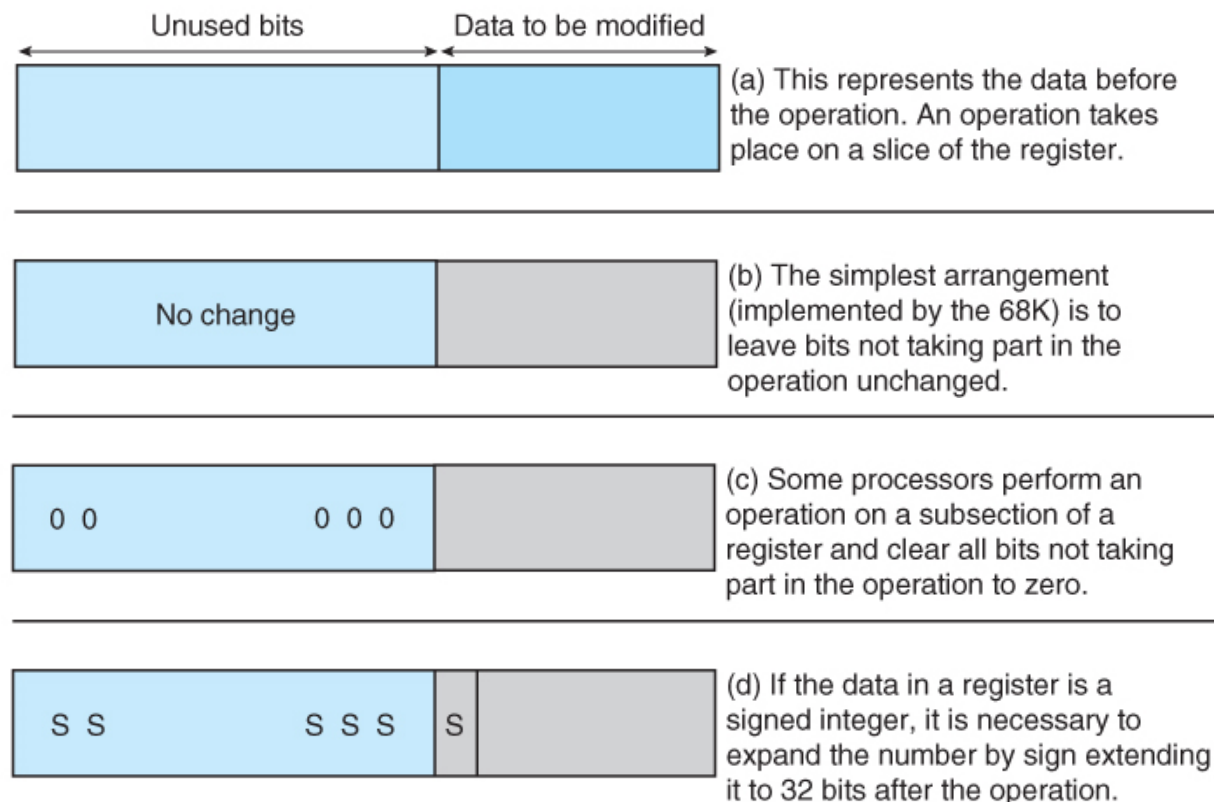❑ Registers usually have the same width as the fundamental word of a computer.

❑ The *ARM* processors have

  o general-purpose registers, and

  o two special purpose registers (have special hardware-defined functions)

29

Data Extension

❑ Sometimes registers hold data values smaller than their actual length
  o for example, a 16-bit (halfword in a 32-bit word register).

❑ What happens to the other bits? (*processor dependent*)
  o some leave the unused bits unchanged,
  o some set the unused bits to 0, and
  o some sign-extend the 16-bit halfword to 32-bits (*two's complement*)

FIGURE 3.9   Operations on a subsection of a register

Unused bits   Data to be modified

(a) This represents the data before the operation. An operation takes place on a slice of the register.

No change

(b) The simplest arrangement (implemented by the 68K) is to leave bits not taking part in the operation unchanged.

0 0   0 0 0

(c) Some processors perform an operation on a subsection of a register and clear all bits not taking part in the operation to zero.

S S   S S S   S

(d) If the data in a register is a signed integer, it is necessary to expand the number by sign extending it to 32 bits after the operation.

© Cengage Learning 2014

30

Addressing Modes

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

FIGURE 3.10   Progressive sequence of addressing modes

r0

MOV r0, #12

Immediate address
MOV r0, #12

The ARM does not directly support this mode

ABC

r0

Absolute address
LDR r0, ABC

r1

r0

Address register indirect
LDR r0, [r1]

© Cengage Learning 2014

31
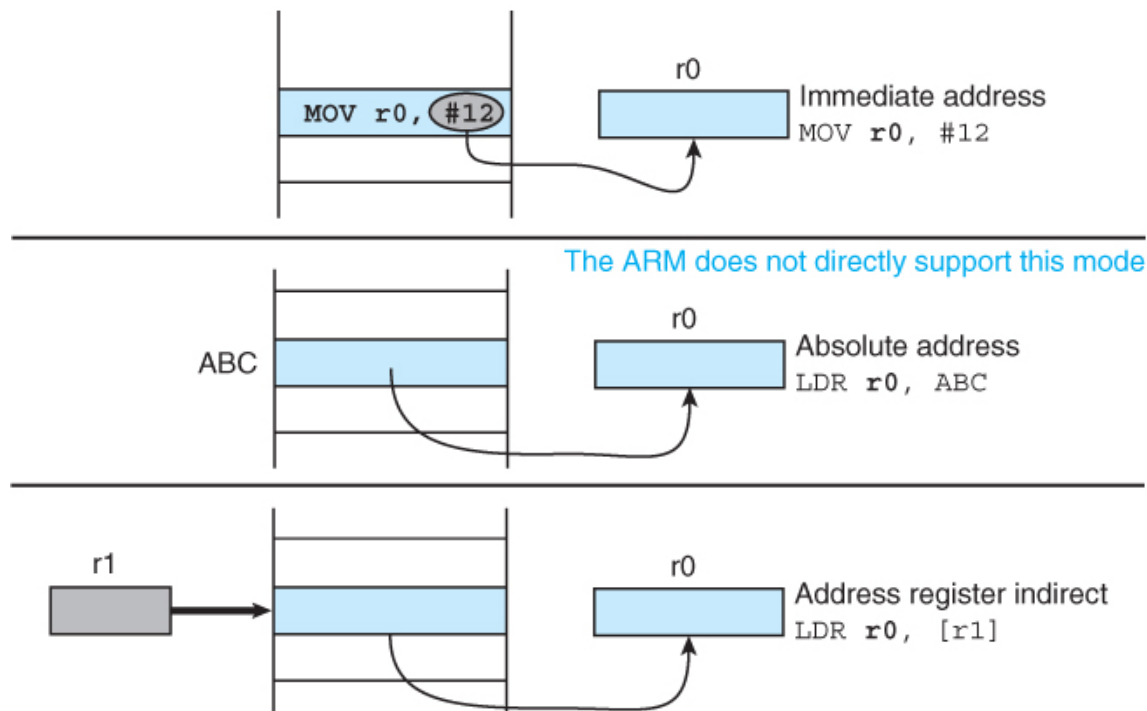
Instruction Types

❑ *Memory-to-register*
   o The source operands are in memory and → *expensive.*
   o the destination operand is in a register
❑ *Register-to-memory*
   o The source operands are in registers and
   o the destination operand is in memory
❑ *Register-to-register*
   o Both operands are in registers. *not that expensive.*

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 from memory to a register, or from a register to memory, respectively, using **Register indirect addressing mode**.*

32

Operands and Instructions

❑ **CISC** processors *typically* have
  o **Two-address** instructions, where
    ▪ **one** address is **memory** and the **other** is a **register**.

❑ **RISC** processors *typically* have a
  o **three-address** data processing instruction, where
    ▪ **the three** operand addresses **are registers**.

  o They *also have* **two** special *two-address* instructions,
    ▪ **LOAD** and **STORE**.

33

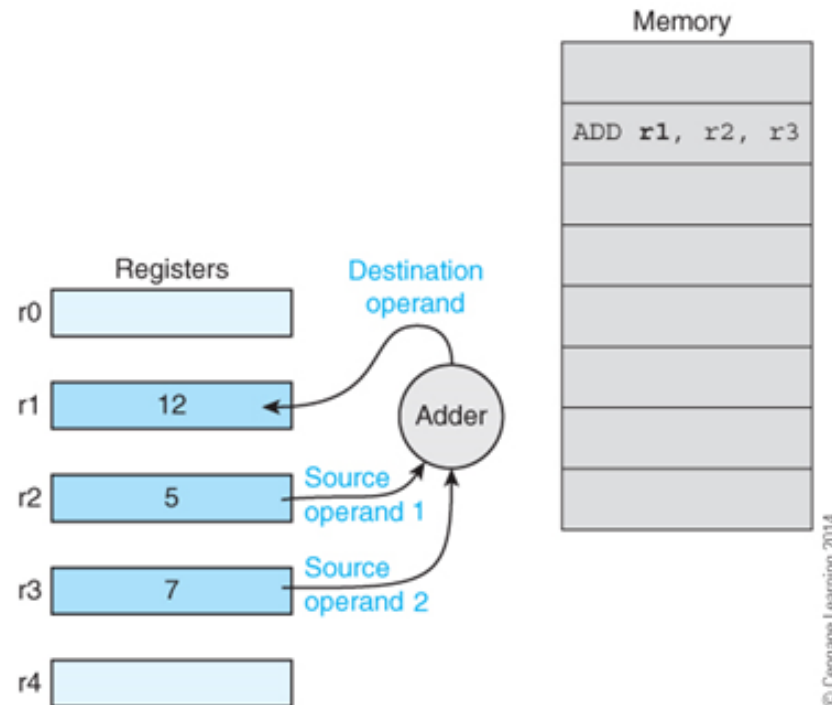# Three Address Machines

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

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

FIGURE 3.11   The three address instruction

34

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
   o   Acorn Computers,
   o   Apple Computers, and
   o   VLSI Technology.

❑ The *1st-generation* of *ARM* was   8-bit microprocessors.   *[handwritten: 1 bite]*

❑ The *2nd-generation* of *ARM* was 32-bit microprocessors.   *[handwritten: 4 bites]*
   o   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 1st-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.

[handwritten: even if less than 32 bits, fill by insignificant digits.]

35

ARM Register Set

❑ The **ARM** processor has
  o 16 *32-bit* registers (r0, r1, r2, …,r11, 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*
    ▪ r11 is *reserved* for **use by the programmer** as the *frame pointer*

by convention, but not wrong to use another register

❑ Sixteen registers require a 4-bit address.

Review Slide 3 in Chapter 2.

Condition Code Register

❑ The **ARM**'s *current program status register* (**CPSR**) contains
  o Condition codes (bits number 31, 30, 29, and 28)
    N (negative), Z (zero), C (carry) and V (overflow) flag bits
  o Operating mode (bits number 0–7)
    Might talk about them later

if put s at end of instruction, will set these flags.

❑ **ARM** processors have a rich instruction set

Convention: use last few processors for special registers.

36

consider registers memory array.

each with 4 bites.

ARM Register Set

FIGURE 3.12   ARM register set

User registers

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15 = PC

r0 to r12 are
general-purpose
registers

r13 — Stack pointer
r14 — Link register
r15 = PC — Program counter

The CPSR (current processor status register)

31	30	29	28	27	8	7	6	5	4	0
N	Z	C	V	unused		I	F	T	mode	

Condition codes   Operating mode

The use of r13 as a stack pointer is a programming convention whereas the use of r14 and r15 as the link register and program counter is enforced by the hardware.

© Cengage Learning 2014

37

Typical ARM Instructions

TABLE 3.1	ARM Data Processing	Data Transfer,	and Compare Instructions

Instruction	ARM Mnemonic	Definition
Addition	ADD r0,r1,r2	[r0] ← [r1] + [r2]
Subtraction	SUB r0,r1,r2	[r0] ← [r1] - [r2]
AND	AND r0,r1,r2	[r0] ← [r1] · [r2]
OR	ORR r0,r1,r2	[r0] ← [r1] + [r2]
Exclusive OR	EOR r0,r1,r2	[r0] ← [r1] ⊕ [r2]
Multiply	MUL r0,r1,r2	[r0] ← [r1] x [r2]
Register-to-register move	MOV r0,r1	[r0] ← [r1]
Compare	CMP r1,r2	[r1] - [r2]
Branch on zero to label	BEQ label	[PC] ← label (jump to label)

(handwritten annotation) don't automatically set flags unless S is added after.

(handwritten annotation) Comparing to → set flags: destination omitted.

© Cengage Learning 2014

38

ARM Assembly Language

❑ ARM instructions are written in the form

(optional)

used to identify location ← {Label} Op-code **operand1**, operand2, operand3 {;comment}

must be in Column 0 *ADD, SUB etc.* *usually destination register only* *register, constant etc.*

❑ 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.

❑ 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
```

→ *empty lines ignored*

put spaces.

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

39

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)
        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.
   o 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:
   o ***executable instructions*** that are executed by the computer and
   o ***assembler directives*** that tell the assembler something about the *environment*.

40

Structure of an ARM Program

identify an Area *will be Code*
called Cubes.   ↓   ↓

AREA Cubes, CODE, READONLY
ENTRY ← *initialize the PC to first instruction.*
        like main in C

```
        MOV   r0,#0        ;clear total in r0
        MOV   r1,#10       ;FOR i = 10 to 1
Next    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

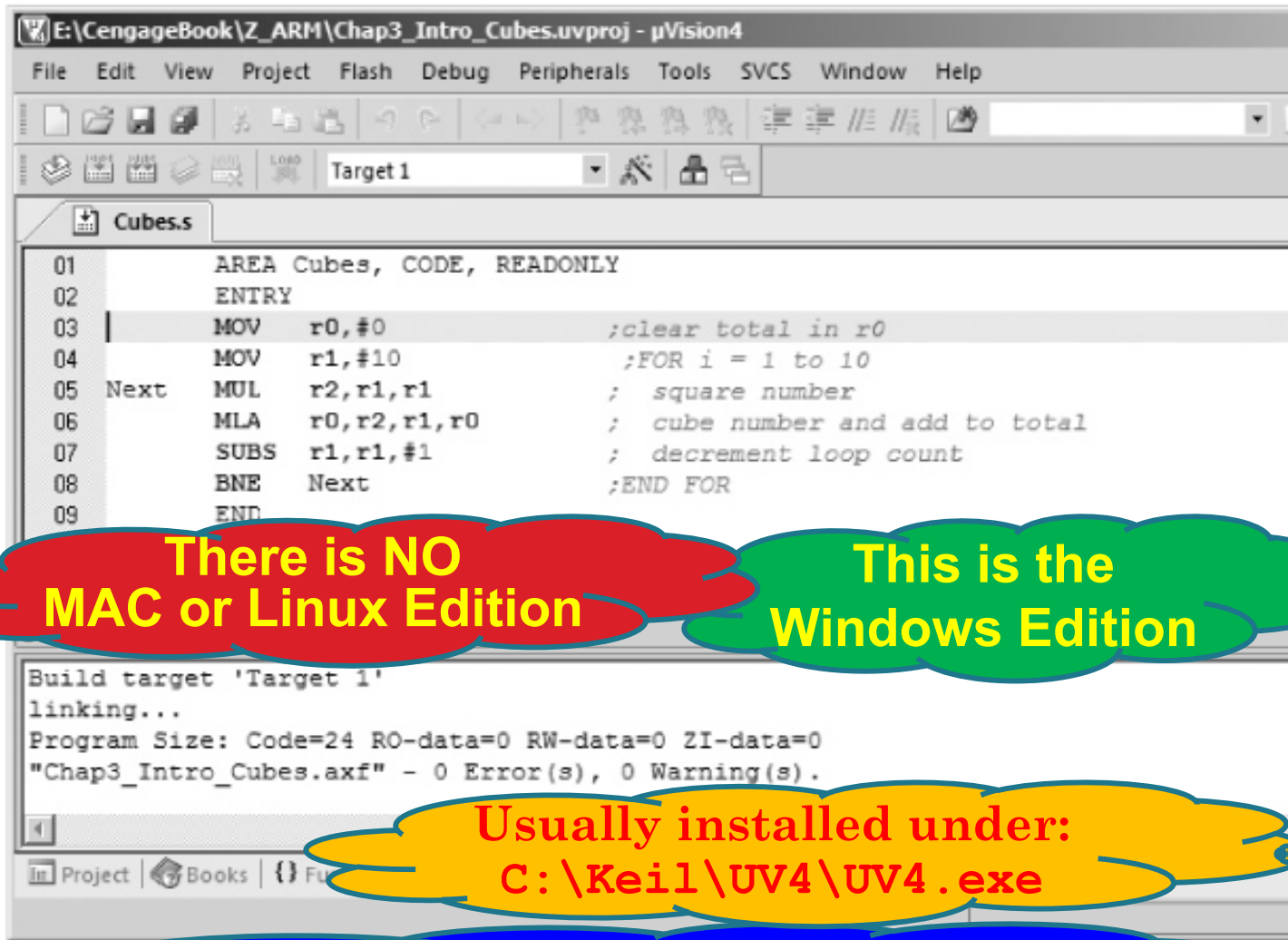*executable instructions*

assembler directive

one entry, one end.
can have many areas.

41

Snapshot of the Display of an ARM Development System

We use version 4.

FIGURE 3.13   Assembling an assembly language program using Kiel's ARM IDE


```
E:\CengageBook\Z_ARM\Chap3_Intro_Cubes.uvproj - µVision4

File   Edit   View   Project   Flash   Debug   Peripherals   Tools   SVCS   Window   Help

Target 1

Cubes.s

01          AREA  Cubes, CODE, READONLY
02          ENTRY
03          MOV   r0,#0              ;clear total in r0
04          MOV   r1,#10             ;FOR i = 1 to 10
05   Next   MUL   r2,r1,r1           ;  square number
06          MLA   r0,r2,r1,r0        ;  cube number and add to total
07          SUBS  r1,r1,#1           ;  decrement loop count
08          BNE   Next               ;END FOR
09          END
```

There is NO MAC or Linux Edition

This is the Windows Edition

```
Build target 'Target 1'
linking...
Program Size: Code=24 RO-data=0 RW-data=0 ZI-data=0
"Chap3_Intro_Cubes.axf" - 0 Error(s), 0 Warning(s).
```

Project | Books | {} Fu

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

This is MicroVision 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
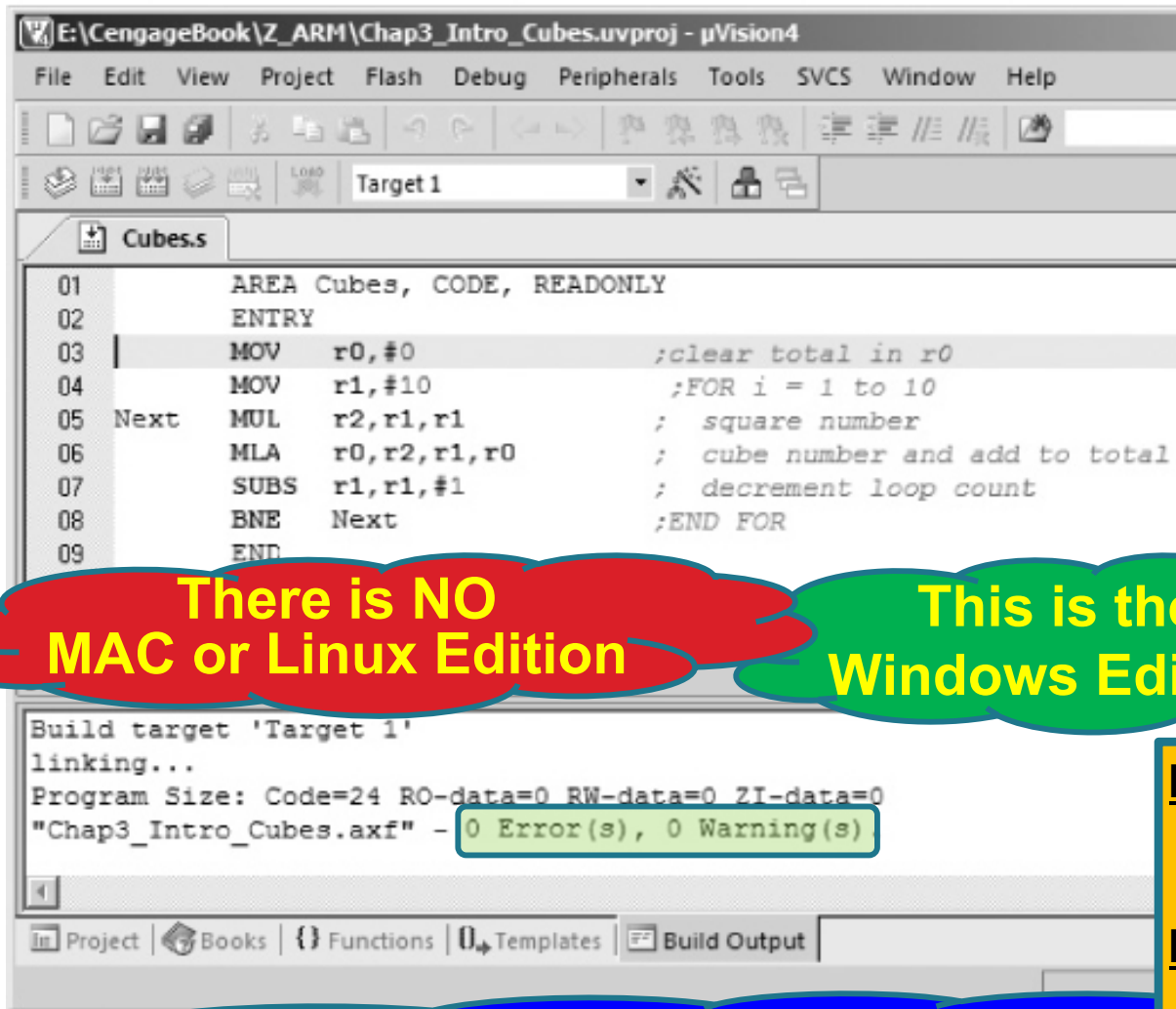   **Save**
        Enter file name
        (to simplify things, use
        .s as an extension
    Save

Snapshot of the Display of an ARM Development System

FIGURE 3.13   Assembling an assembly language program

```
E:\CengageBook\Z_ARM\Chap3_Intro_Cubes.uvproj - µVision4
File   Edit   View   Project   Flash   Debug   Peripherals   Tools   SVCS   Window   Help

Target 1

Cubes.s
01          AREA  Cubes, CODE, READONLY
02          ENTRY
03          MOV   r0,#0              ;clear total in r0
04          MOV   r1,#10             ;FOR i = 1 to 10
05   Next   MUL   r2,r1,r1           ;  square number
06          MLA   r0,r2,r1,r0        ;  cube number and add to total
07          SUBS  r1,r1,#1           ;  decrement loop count
08          BNE   Next               ;END FOR
09          END
```

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.

There is NO MAC or Linux Edition

This is the Windows Edition

```
Build target 'Target 1'
linking...
Program Size: Code=24 RO-data=0 RW-data=0 ZI-data=0
"Chap3_Intro_Cubes.axf" - 0 Error(s), 0 Warning(s)
```

Project Books {} Functions 0 Templates Build Output

Debug
 Start/Stop Debug Session
 Ok
Debug
 Step, or simply press F11

This is MicroVision 4, not 5.

43