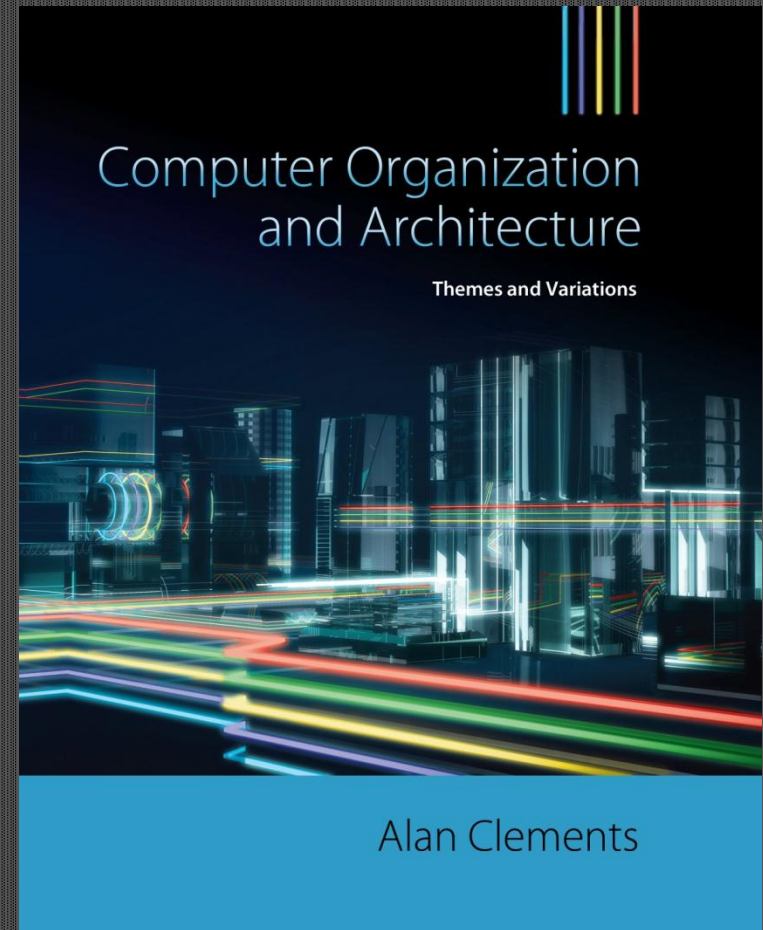


# COMPUTER ORGANIZATION AND ARCHITECTURE

## Chapter 3



# THE INSTRUCTION SET ARCHITECTURE

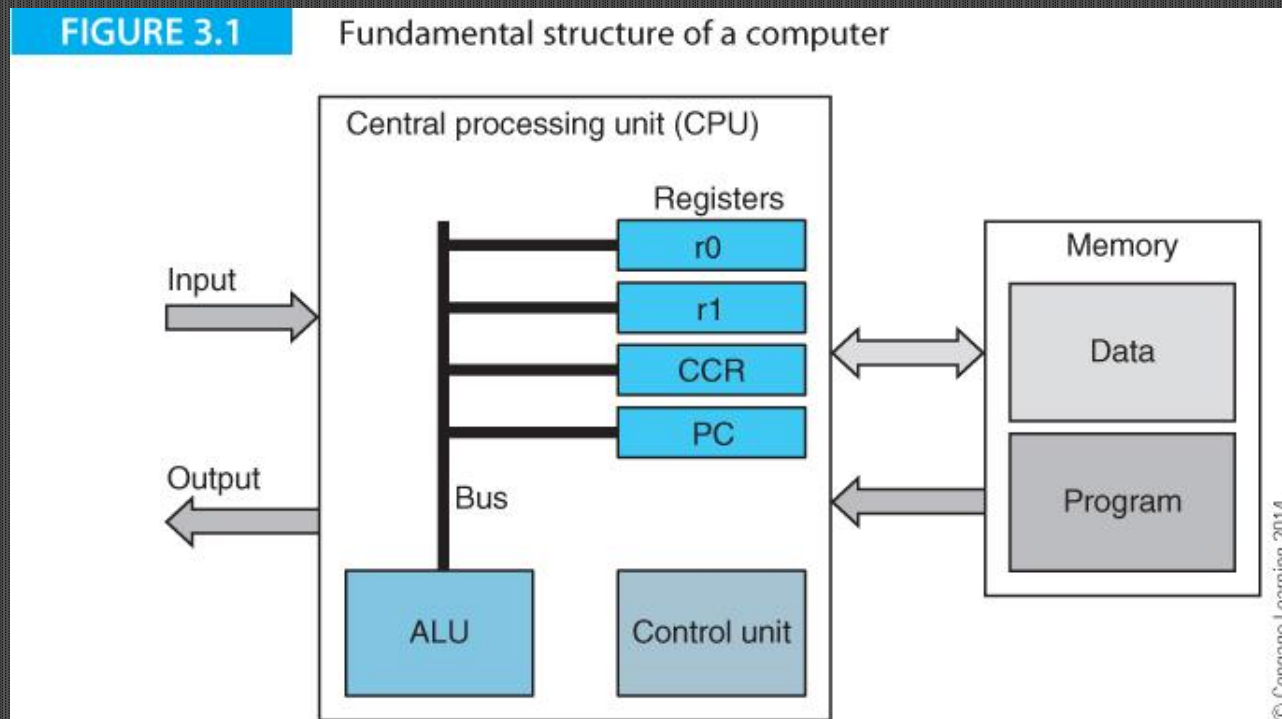
In this set of lectures, we:

- examine the stored program machine and show how an instruction is executed
- introduce instruction formats for *memory-to-memory*, *register-to-memory*, and *register-to-register* operations
- demonstrate how a processor implements *conditional behavior* by selecting one of two alternative actions depending on the result of a test
- describe a set of computer instructions and show how computers access data (*addressing modes*)
- introduce ARM's development system and show how ARM programs are written
- demonstrate how the ARM uses *conditional execution* to implement efficient code.

Figure 3.1 illustrate the structure of a simple hypothetical stored program computer.

The CPU reads instructions from memory and executes them.

Temporary data is stored in registers such as r1 and r2. The PC, program counter, is the register that steps through the program. That is, the PC points at the next instruction to be executed.



## Computer Architecture

The word *architecture* in the expression *computer architecture* is analogous to the same word in the world of building because it indicates *structure* and implies design and planning. Computer architecture describes the structure of a computer from the perspective of the programmer or compiler writer rather than that of the electronic engineer.

The origins of *computer architecture* go back to the early 1960s when each new computer was different from its predecessors and had a unique instruction set. IBM changed computing with the System/360 series, which had a common architecture and instruction set across all models. Each model executed the same instructions, so you could upgrade from a low-cost machine without having to rewrite all your programs. In 1964, this was a radical notion. Forty years later, it is common practice.

# Instruction Formats

A computer executes instructions from 8 bits wide to 80 bits wide.

The instruction format defines the anatomy of an instruction (the number of bits devoted to defining the operation, the number of operands, and the format of operands).

Consider the following examples of instructions. The examples in red show how an instruction might be described in words and below are several examples of actual instructions.

LDR **registerdestination**, memorysource

STR registersource, **memorydestination**

Operation **registerdestination**, registersource1, registersource2

LDR **r1**, 1234

STR r3, **2000**

ADD **r1**, r2, r3

SUB **r3**, r3, r1



## Features

A stored program machine is a computer that has a program in digital form in its main memory. The program counter points to the next instruction to be executed and is incremented after each instruction has been executed.

The program and data are stored in the same memory.

In reality, today's computers store programs and data in separate cache memory. This detail does not affect the following discussion.

A stored program operates in a fetch/execute two-phase mode. In the fetch phase the next instruction is read from memory and decoded.

In the execute phase the instruction is interpreted or executed by the CPU's logic.

Modern computers are pipelined, and fetch and execute operations overlap.

A stored program computer has several registers.

**MAR** The memory address register stores the *address* of the location in main memory that is currently being accessed by a read or write operation.

**MBR** The memory buffer register stores data that has just been read from main memory, or data to be immediately written to main memory.

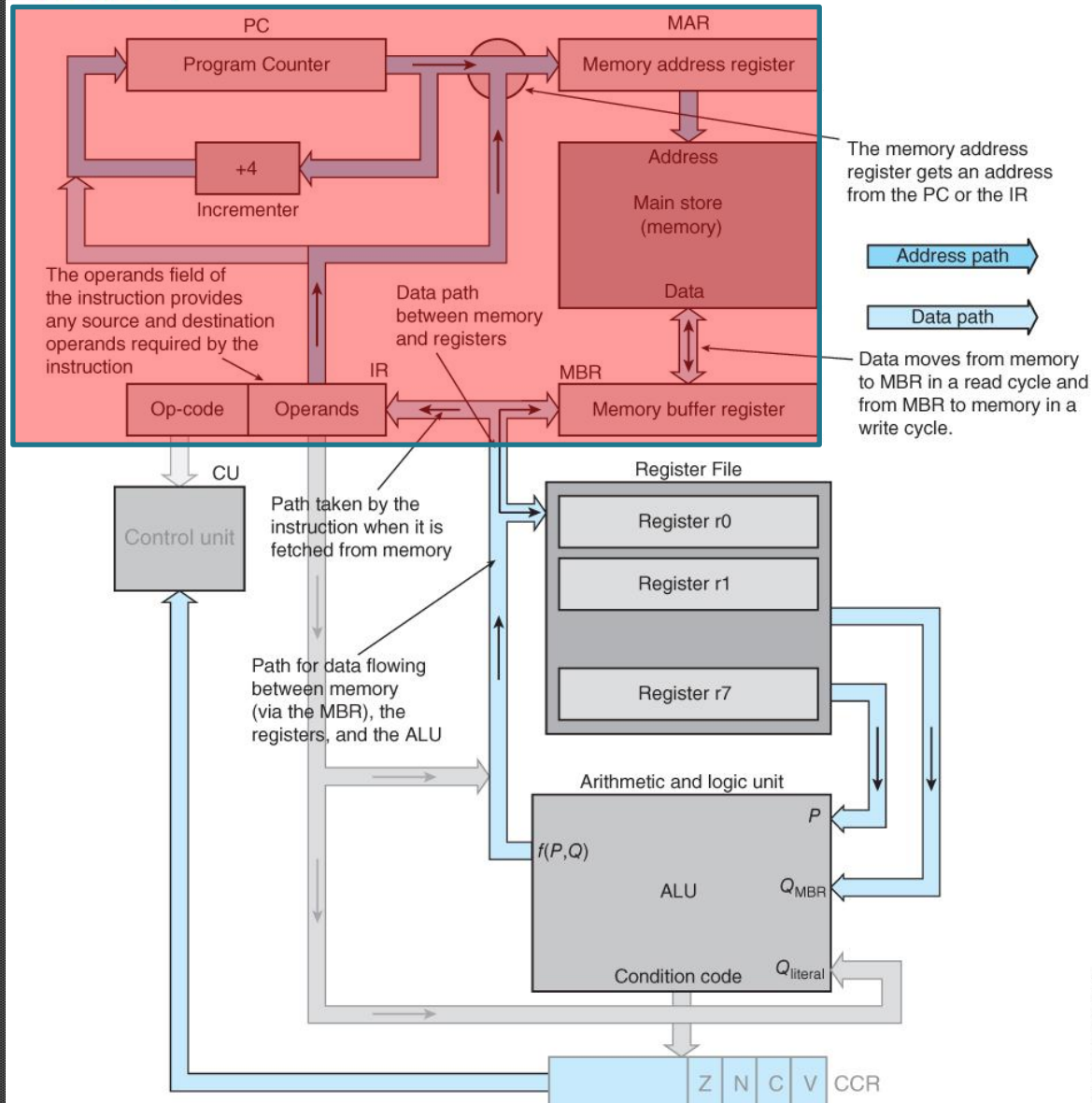
**PC** The program counter contains the address of the next instruction to be executed. Thus, the PC *points* to the location in memory that holds the next instruction.

**IR** The instruction register stores the instruction most recently read from main memory. This is the instruction currently being executed.

**r0 - r7** The register file is a set of eight general-purpose registers r0, r1, r2, ..., r7 that store temporary (working) data, for example, the intermediate results of calculations. A computer requires at least one general-purpose register. Our simple computer has *eight* general-purpose registers.

We are going to use the ARM processor to introduce assembly language and a modern ISA. However, we begin with the description of a very simple hypothetical computer to keep things simple.

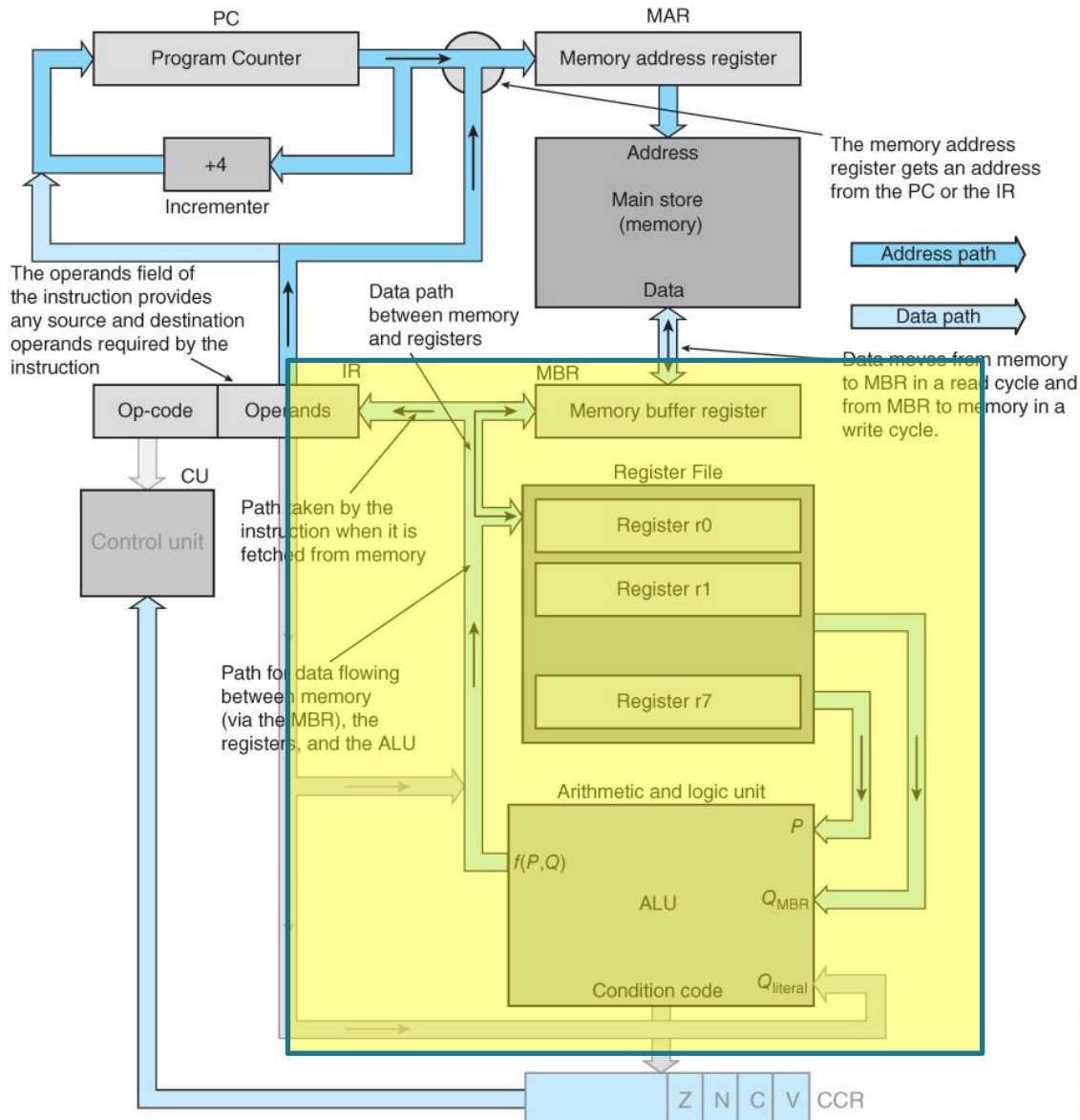


**FIGURE 3.2** Partial structure of a hypothetical stored program machine

## Structure of a Computer

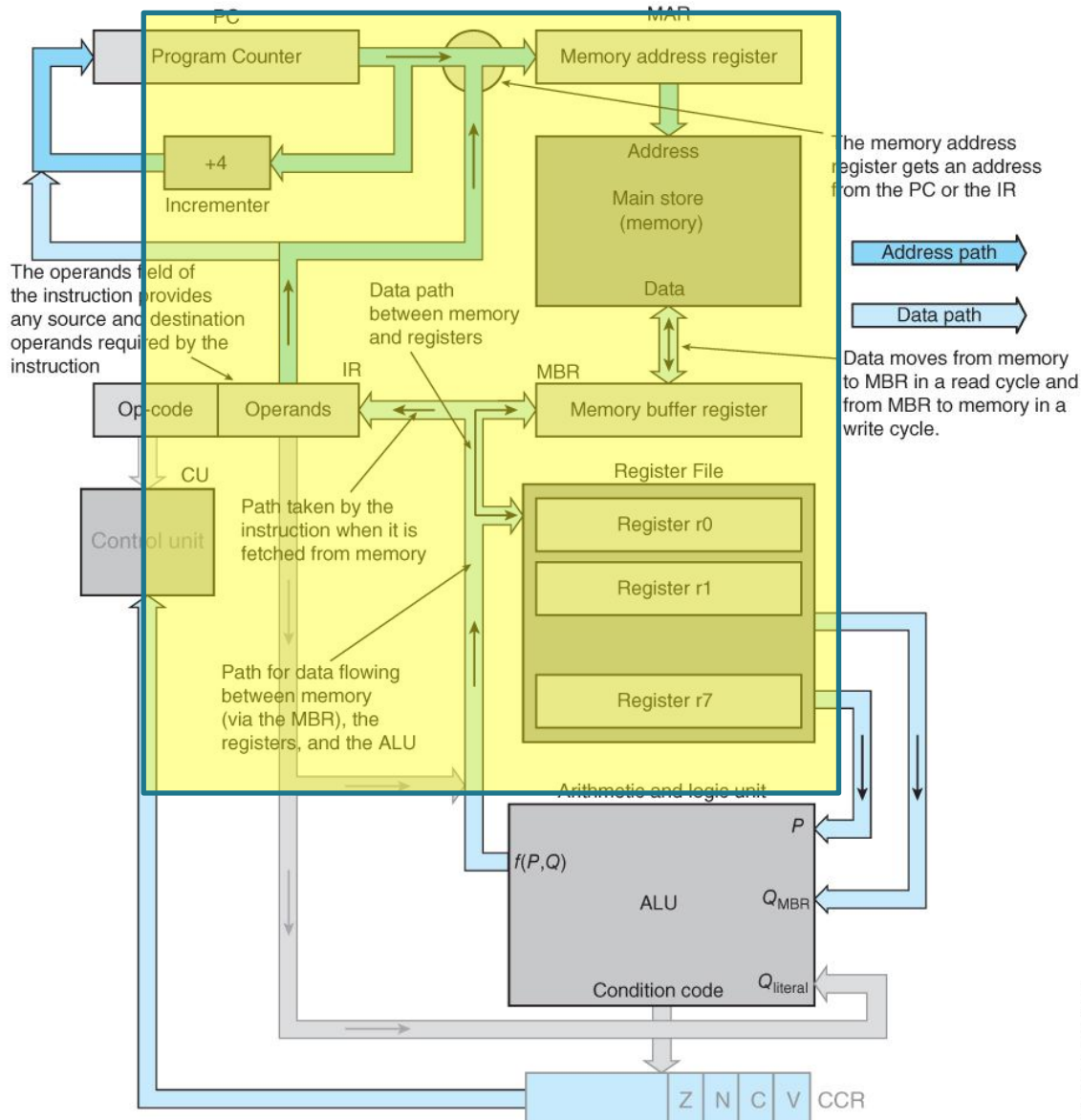
The program counter supplies an address to the MAR which holds it while the instruction is looked up in memory.

The instruction is loaded into the memory buffer register, MBR, and then copied to the instruction register, IR where the op-code is decoded.

**FIGURE 3.2** Partial structure of a hypothetical stored program machine

## Structure of a Computer

In the execute phase, the operands may be read from the register file, transferred to the ALU (arithmetic and logic unit) where they are operated on and then the result passed to the destination register.

**FIGURE 3.2** Partial structure of a hypothetical stored program machine

## Structure of a Computer

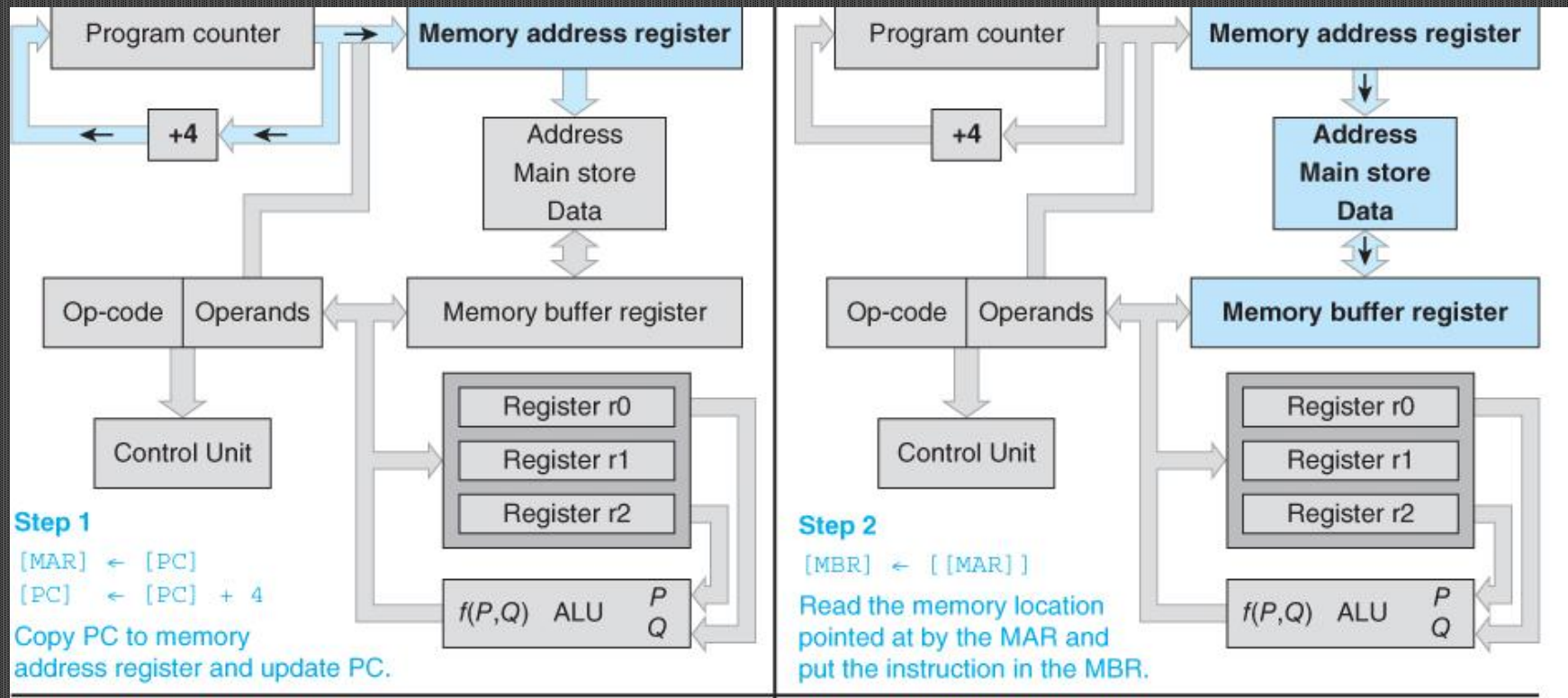
If the operation requires a memory access (e.g., a load or store), the memory address in the instruction register is sent to the memory address register and a read or write operation performed.

## Fetch/execute cycle in RTL

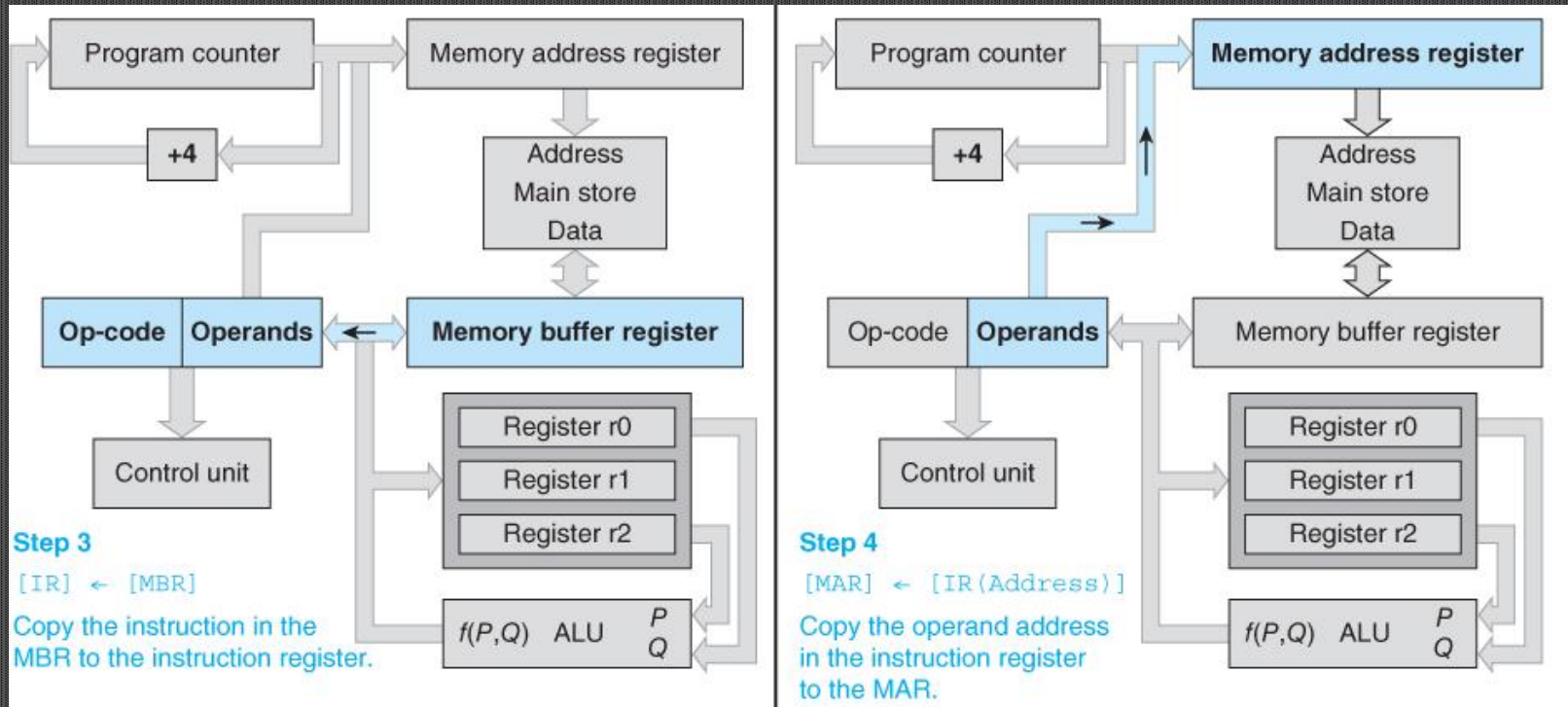
FETCH	$[MAR] \leftarrow [PC]$	;copy PC to MAR
	$[PC] \leftarrow [PC] + 4$	;increment PC
	$[MBR] \leftarrow [[MAR]]$	;read instruction pointed at by MAR
	$[IR] \leftarrow [MBR]$	;copy instruction in MBR to IR
LDR	$[MAR] \leftarrow [IR(\text{address})]$	;copy operand address from IR to MAR
	$[MBR] \leftarrow [[MAR]]$	;read operand value from memory
	$[r1] \leftarrow [MBR]$	;add the operand to register r1

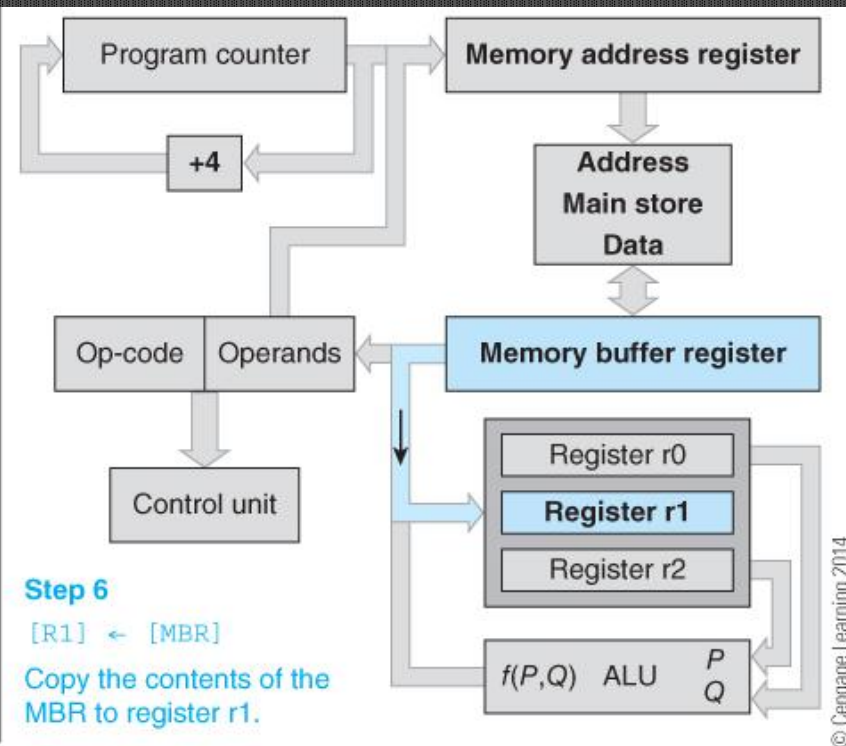
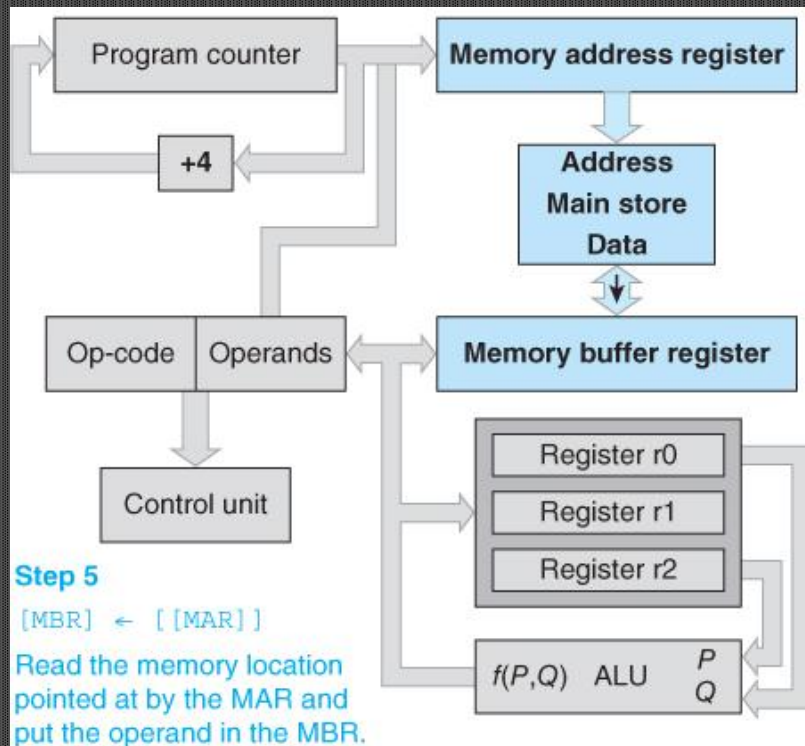


# Fetching and Executing an Instruction









## DEALING WITH CONSTANTS

Suppose we want to load the *number 1234 itself* into register r1.

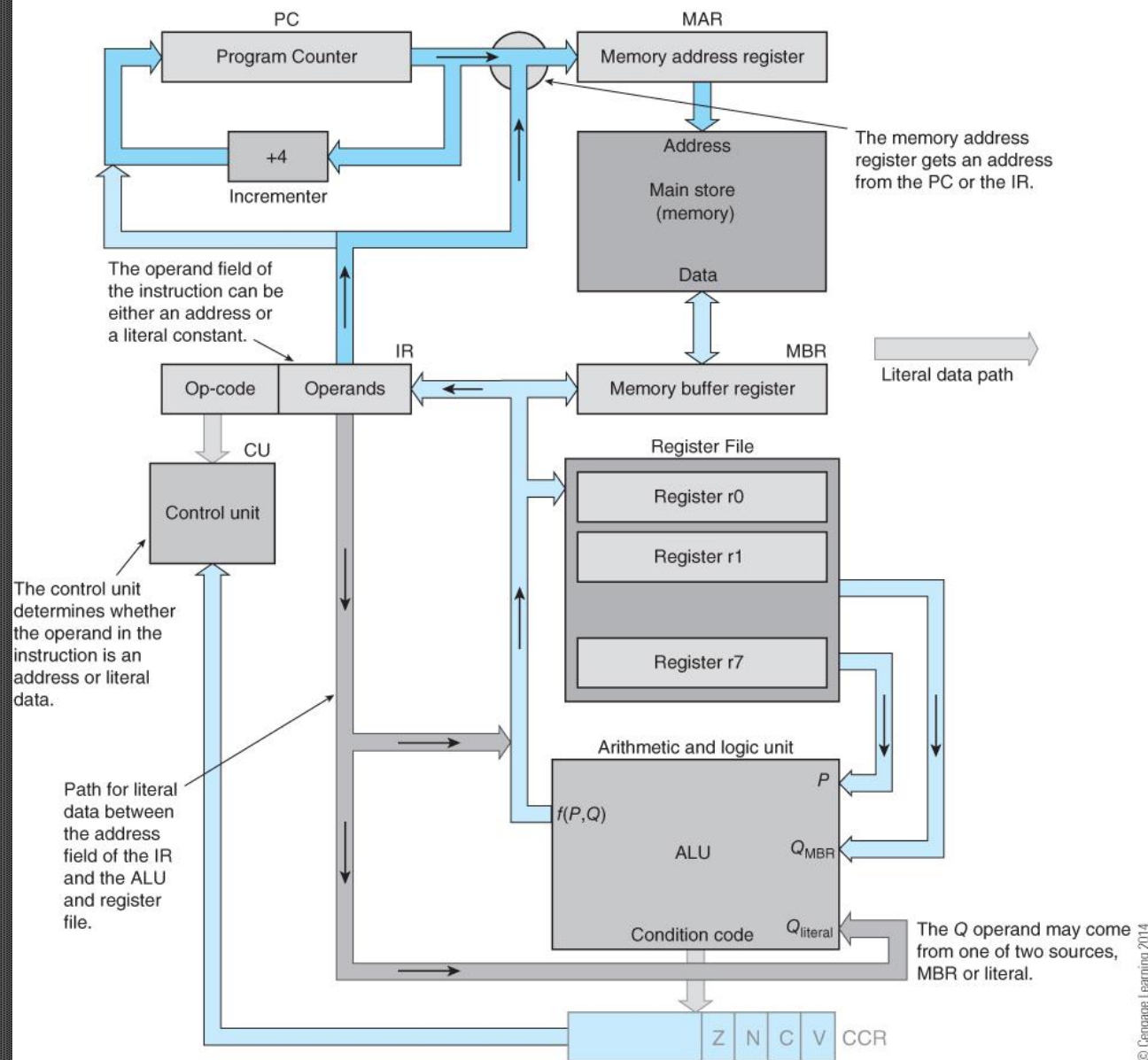
Such a number is called a *literal* operand.

ADD r0,r1,#25 adds the value 25 to contents of r1 and puts sum in r0

Figure 3.4 illustrates the data paths required to implement literal operands.

A path from the instruction register, IR, routes a literal operand to the register file, MBR, and ALU;

When ADD r0,r1,#25 is executed, the operand r1 is routed from the operand field of the IR, rather than from the memory system via the MBR.

**FIGURE 3.4** Information paths for literal operands

## SAMPLE INSTRUCTIONS

LDR <b>r0</b> , <i>address</i>	Load the contents of the memory location at <i>address</i> into register r0.
STR <b>r0</b> , <i>address</i>	Store the contents of register r0 at the specified <i>address</i> in memory.
ADD <b>r0</b> ,r1,r2	Add the contents of register r1 to the contents of register r2 and store the result in register r0.
SUB <b>r0</b> ,r1,r2	Subtract the contents of register r2 from the contents of register r1 and store the result in register r0.
BPL <i>target</i>	If the result of the previous operation was positive, then branch to the instruction at address <i>target</i> .
BEQ <i>target</i>	If the result of the previous operation was zero, then branch to the instruction at address <i>target</i> .
B <i>target</i>	Branch unconditionally to the instruction stored at the memory address <i>target</i> . This executes the instruction at address <i>target</i> .



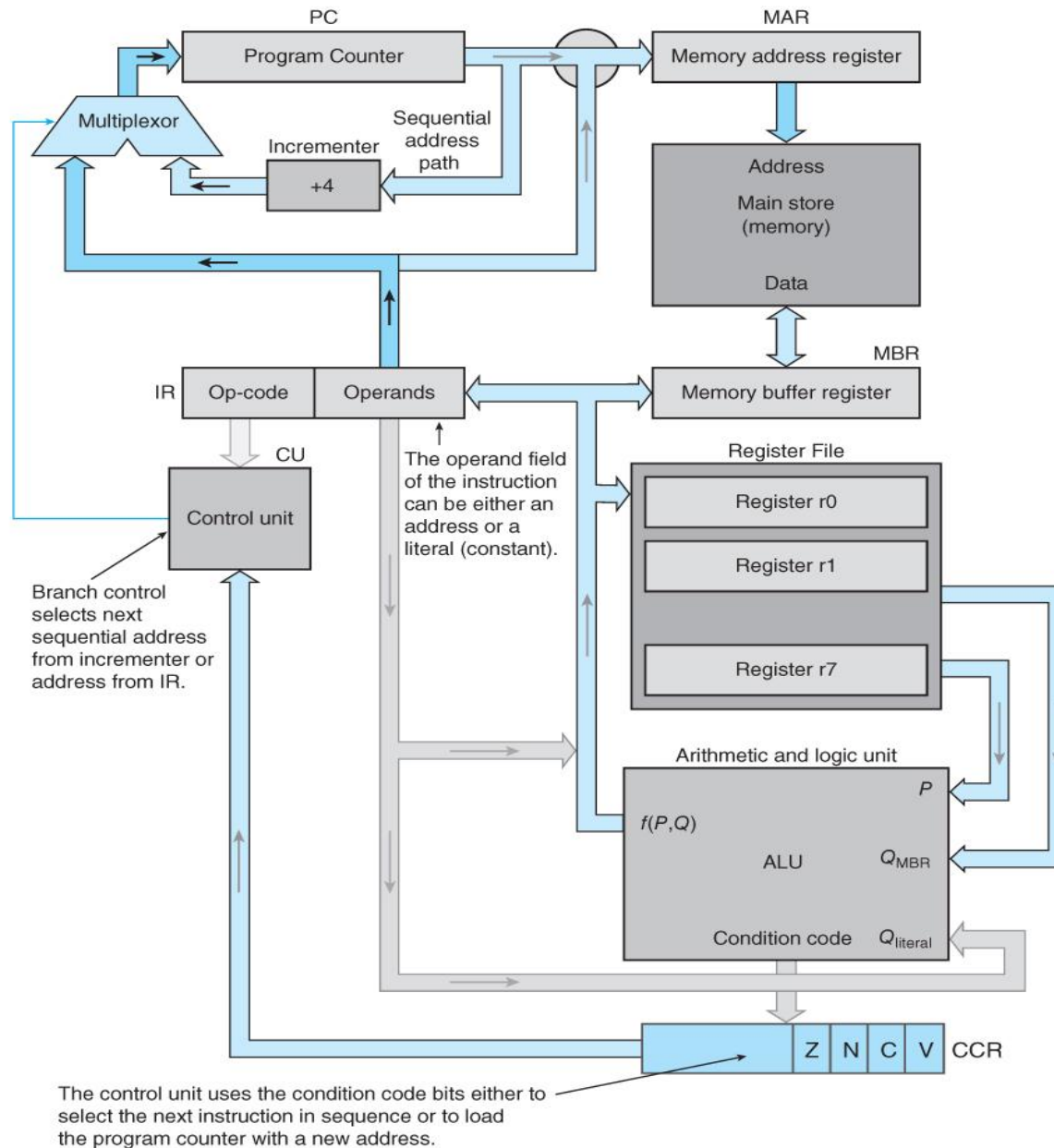
## FLOW CONTROL

*Flow control* refers to any action that modifies the strict instruction-by-instruction sequence of a program.

*Conditional behavior* allows a processor to select one of two possible courses of action.

Figure 3.5 shows the information paths required to implement conditional behavior.

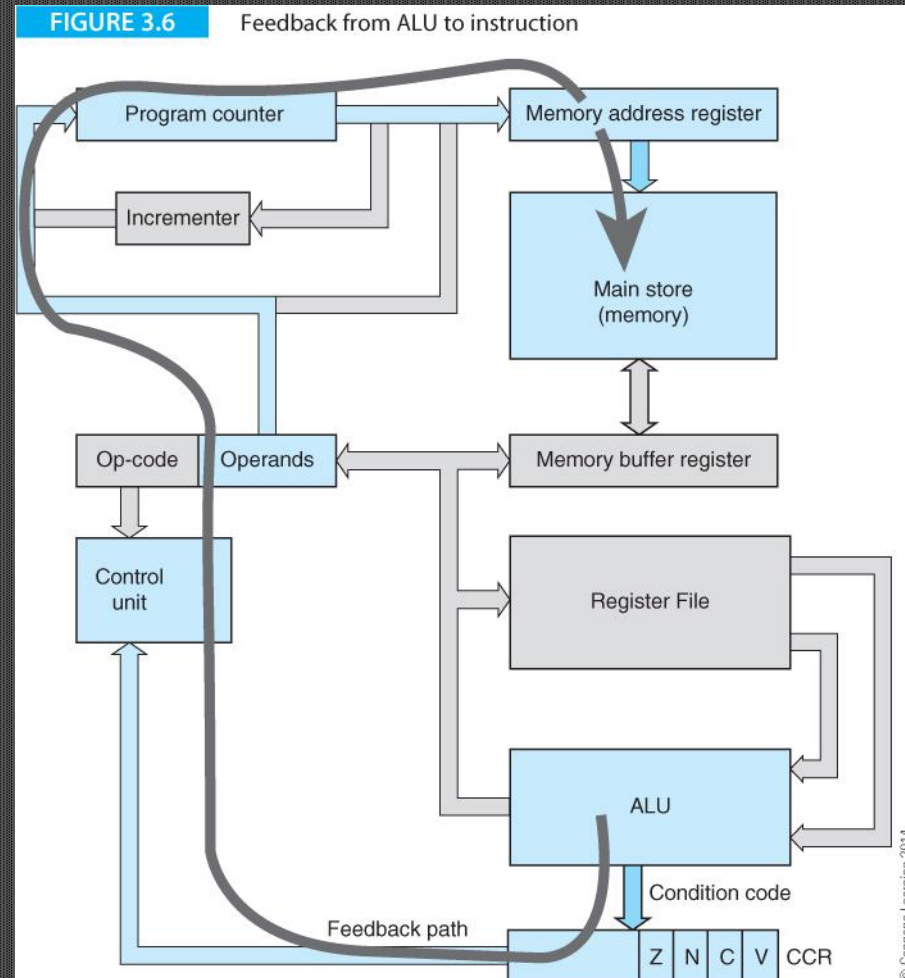
A *conditional instruction* like BEQ results in either continuing program execution normally, or loading the program counter with a new value and executing a *branch* to another region of code.

**FIGURE 3.5** Implementing conditional behavior at the machine level

© Cengage Learning 2014

# FLOW CONTROL

Figure 3.6 illustrate how the result from the ALU can be used to modify the sequence of instructions.



# STATUS BITS (FLAGS)

When the computer performs an operation, it stores *status* or *condition* information in the CCR. The processor records whether the result is zero (Z), negative in two's complement terms (N), generated a carry (C), or arithmetic overflow (V).

```

  11011100
+11000000
 $\hline$ 
 110011100
Z = 0, N = 1
C = 1, V = 0

```

CISC processors, like the Intel IA32 update status flags after each operation.

RISC processors, like the ARM, require the programmer update the status flags.

The ARM does it appending an **S** to the instruction; for example SUBS or ADDS.

## EXAMPLE OF A CONDITIONAL OPERATION

```
        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 be zero or it may not be zero.

**BEQ onZero** forces a branch 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.

This implements: if zero then  $r1 = r2 + r3$  else  $r1 = r2 - r3$ .



# EXAMPLE OF A CONDITIONAL OPERATION

$X = P - Q$

IF  $X \geq 0$  THEN  $X = P + 5$   
 ELSE  $X = P + 20$

	LDR	r0,P	;Load r0 with the contents of location P
	LDR	r1,Q	;Load r1 with the contents of location Q
	SUBS	r2,r0,r1	;Subtract the contents of Q from P
			;to get $X = P - Q$
	BPL	THEN	;IF $X \geq 0$ 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 ;the three operands P, Q, X. The memory ;locations are 36, 40, and 44, respectively.
Q	DCD	9	
X	DCD		

# EXAMPLE OF A CONDITIONAL OPERATION

LDR r0,P ;Load r0 with the contents of memory location P  
 LDR r1,Q ;Load r1 with the contents of memory location Q

**SUBS r2,r0,r1 ;Subtract Q from P to get  $X = P - Q$**   
**BPL THEN ;IF  $X \geq 0$  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  
 Q DCD 9  
 X DCD

;These three lines reserve memory space for  
 the three operands P, Q, X. The memory  
 locations are 36, 40, and 44, respectively.

**Here's where the test  
 and conditional  
 branch take place**

# EXAMPLE OF A CONDITIONAL OPERATION

```

LDR  r0,P      ;Load r0 with the contents of memory location P
LDR  r1,Q      ;Load r1 with the contents of memory location Q
SUBS r2,r0,r1  ;Subtract the contents of Q from P to get X = P - Q
BPL  THEN      ;IF X ≥ 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

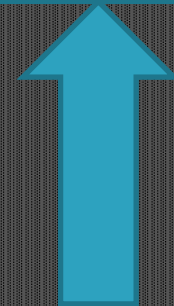
```

```

P      DCD      12
Q      DCD      9
X      DCD

```

;These three lines reserve memory space for  
;the three operands P, Q, X. The memory  
;locations are 36, 40, and 44, respectively.



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

# EXAMPLE OF A CONDITIONAL OPERATION

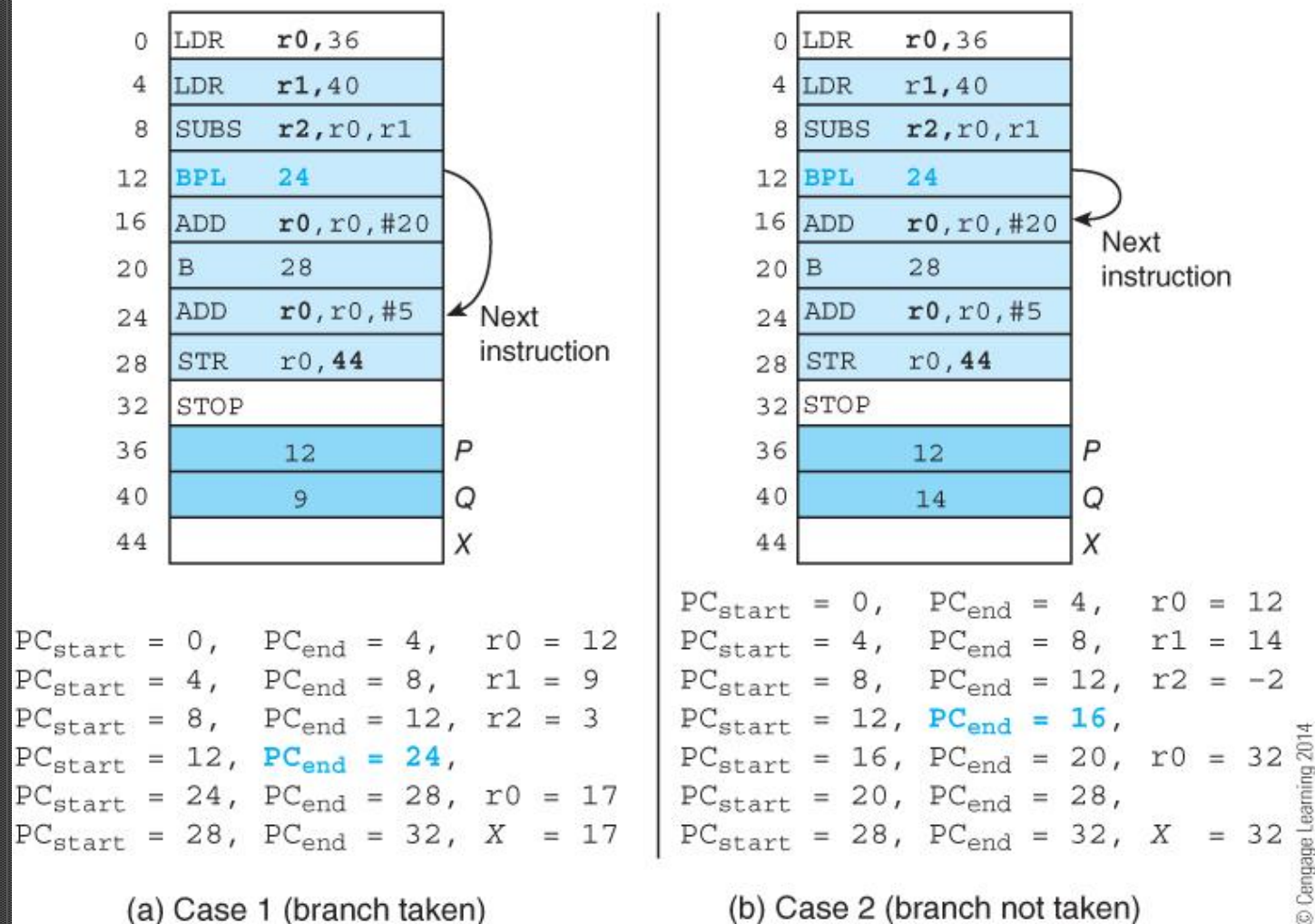
This sequence of assembly-language instructions can be expressed in RTL notation:

	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
ELSE	ADD	r0,r0,#20	;[r0] $\leftarrow$ [r0] + 20
	B	EXIT	;[PC] $\leftarrow$ EXIT
THEN	ADD	r0,r0,#5	;[r0] $\leftarrow$ [r0] + 5
EXIT	STR	r0,X	;[X] $\leftarrow$ [r0]

Case 1: P = 12, Q = 9, and the branch is *taken*

Case 2: P = 12, Q = 14, and the branch is *not taken*

**FIGURE 3.7** Illustration of conditional execution





# Consider the code needed to calculate

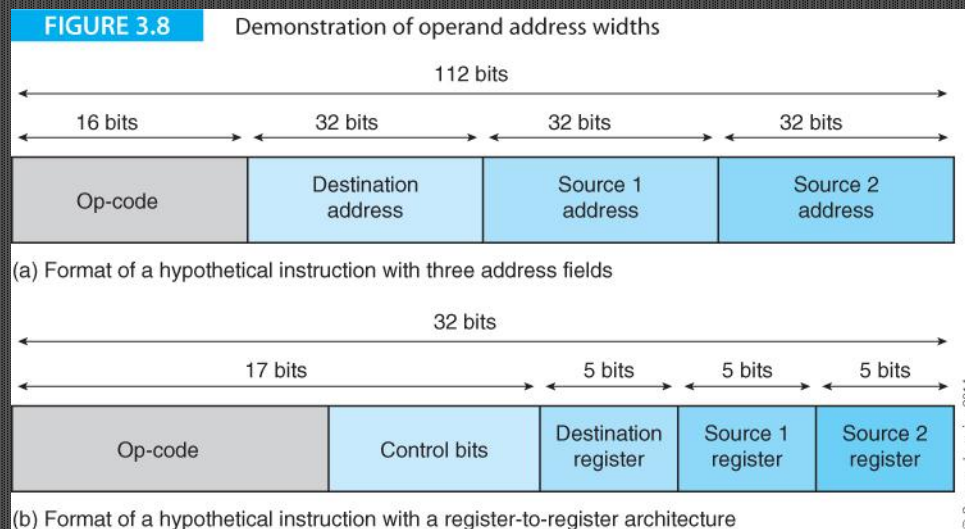
$$1 + 2 + 3 + 4 + \dots + 20$$

```
LDR    r0,#1      ;Put 1 in register r0 (the counter)
LDR    r1,#0      ;Put 0 in register r1 (the sum)
Next  ADD    r1,r1,r0 ;REPEAT: Add current count to sum
      ADD    r0,r0,#1 ; Add 1 to the counter
      CMP    r0,#21  ; Have we added all 20 numbers?
      BNE    Next    ;UNTIL we have made 20 iterations
      STOP                ;If we have THEN stop
```

Figure 3.8a illustrates an instruction that implements ADD A,B,C where A, B, and C are 32-bit memory addresses. The width is 112 bits which is unfeasibly large.

Figure 3.8b illustrates the format of a hypothetical RISC processor with a register-to-register format that can execute ADD R1,R2,R3 where the registers are chosen from 32 possible registers (requiring a 5-bit register address field).

Such a format is used by most 32-bit RISC processors with small variations.



# GENERAL-PURPOSE REGISTERS

Registers are usually the same width as the fundamental word of a computer (but not always so).

The ARM processor has 32-bit registers, a 32-bit program counter, and its basic wordlength is 32 bits wide.

Some computers have dedicated registers – different registers have different functions.

Some computers have entirely general-purpose registers (they all behave identically).

The ARM has general-purpose registers but two have special hardware-defined functions and cannot be used by the programmer for general-purpose data processing.

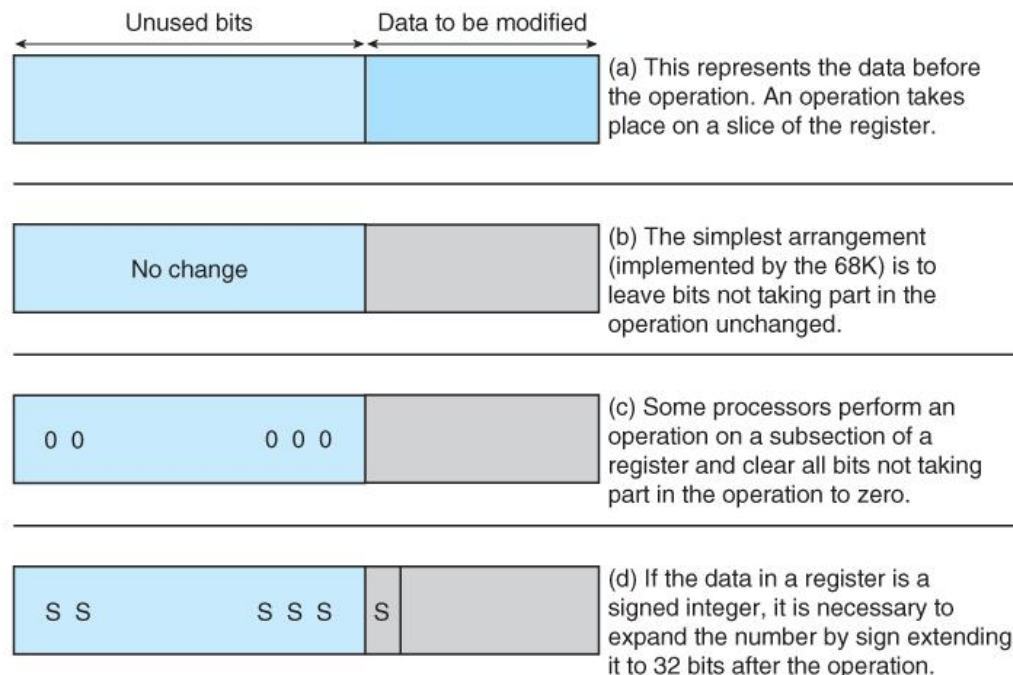
# Data Extension

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?

This is processor dependent. Some set the unused bits to 0, some leave the unused bits unchanged, and some sign-extend the 16-bit word to 32-bits.

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

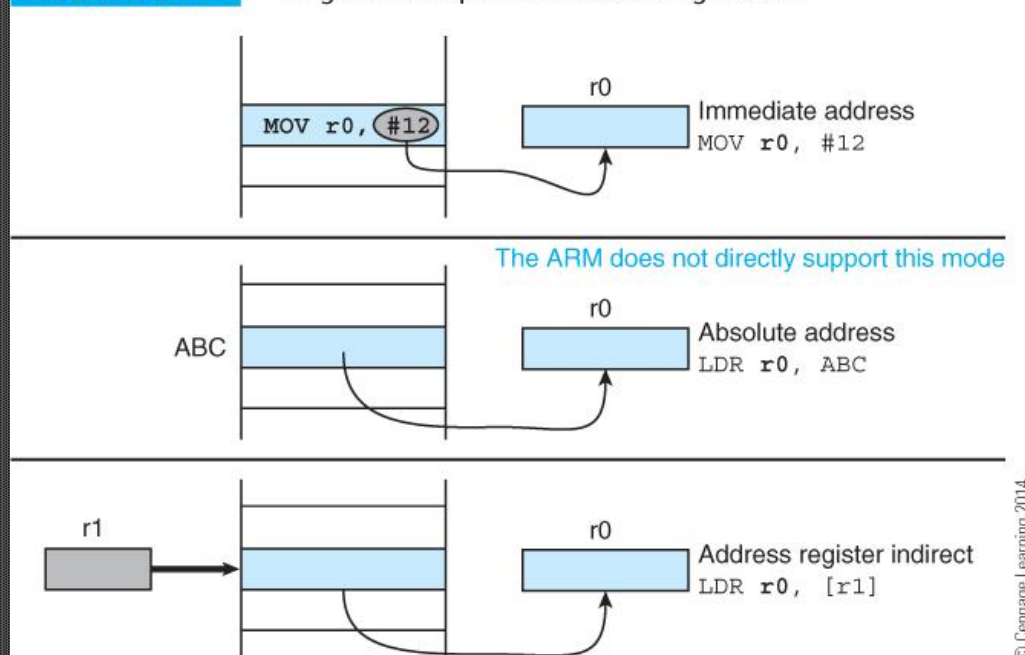


# ADDRESSING MODES

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)
- 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 operand is in memory and the destination operand is in a register

**Register-to-memory** The source operand is in a register and the destination operand is in memory

**Register-to-register** Both operands are in registers.

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

RISC processors like the ARM and MIPS allow only register-to-register data-processing operations. RISC processor have a special LAD and a special STORE instruction to transfer data between memory and a register.

## Program Counter Relative Addressing

Register indirect addressing allows you to specify the location of an operand with respect to a register.

LDR **r0**, [r1, #16] specifies that the operand is 16 bytes on from r1.

Suppose that we use r15, the PC, to generate an address and write

LDR **r0**, [PC, #16].

The operand is 16 bytes on from the PC or  $8 + 16 = 24$  bytes from the current instruction (The ARM's PC is always 8 bytes on from the current instruction).

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

If you relocate the program and its data elsewhere in memory, the relative offset does not change.

# OP-CODES AND INSTRUCTIONS

Computers can have three-address, two-address, one-address, and zero-address instructions.

CISC processors typically have two address instructions where one address is memory and one a register.

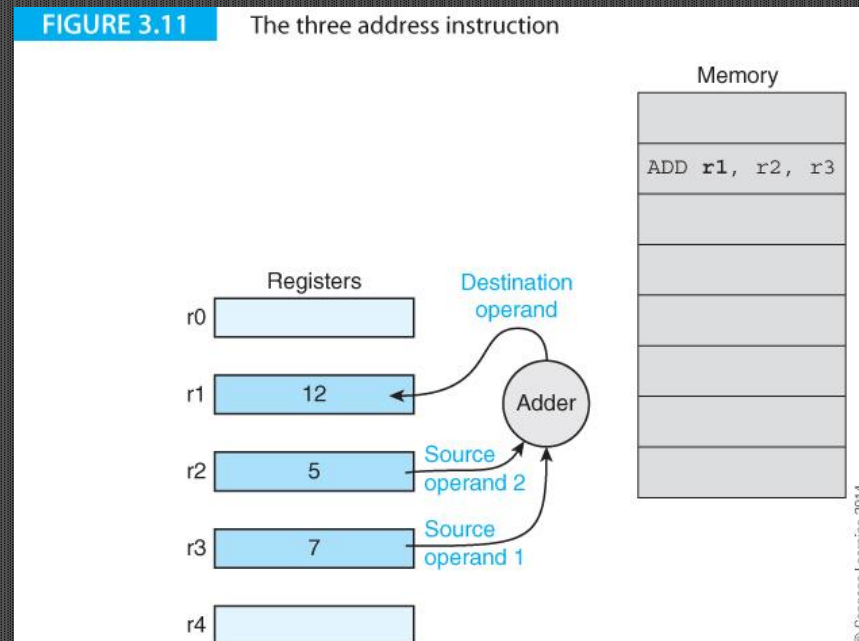
RISC processors typically have a three-address data processing instruction where the three operand addresses are registers. They also have two dedicated two-address instructions, LOAD and STORE.

# THE INSTRUCTION SET ARCHITECTURE

## Sample address formats

Operands	Instruction	Effect
Three	ADD P,Q,R	Add Q to R and put the result in P
Two	ADD P,Q	Add Q to P and put the result in P
One	ADD P	Add P to accumulator and put result in the acc
Zero	ADD	Pop top two items off the stack, add them and push result

**FIGURE 3.11** The three address instruction



## Two Address Machines

A CISC has a *two-address* instruction format. You can execute  $Q \leftarrow P + Q$ . One operand appears *twice*, first as a source and then as a destination.

The price of a two-operand instruction format is the destruction by overwriting of one of the source operands.

Typically, the operands are either two registers or one register and a memory location; for example, the 68K ADD instruction can be written:

Instruction	RTL definition	Mode
ADD D0,D1	$[D1] \leftarrow [D1] + [D0]$	Register-to-register
ADD P,D2	$[D2] \leftarrow [D2] + [P]$	Memory-to-register
ADD D7,P	$[P] \leftarrow [P] + [D7]$	Register-to-memory



## One Address Machines

A one address machine specifies just one operand in the instruction.

The second operand is a fixed register called an *accumulator* that doesn't have to be specified.

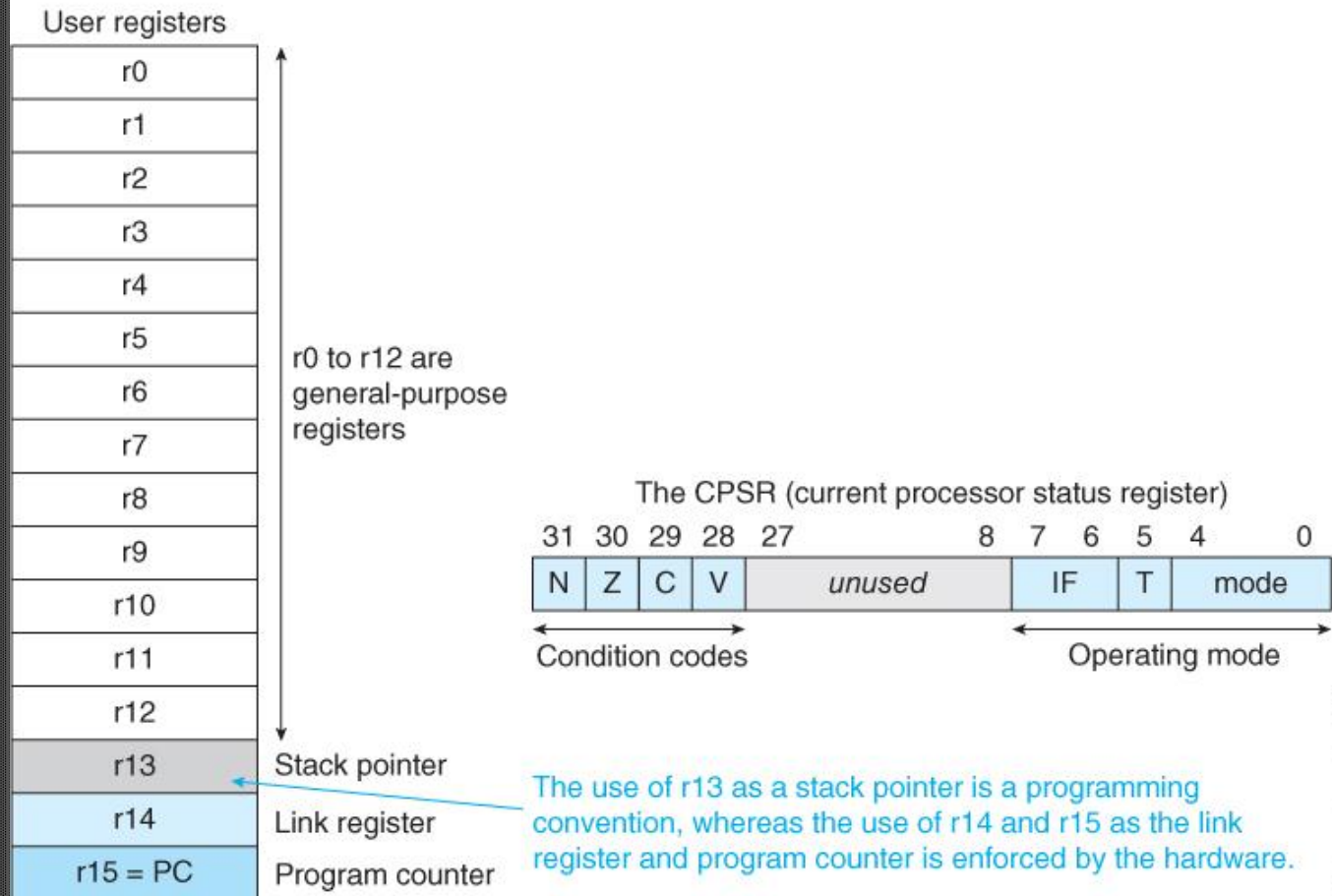
For example, the operation one-address instruction ADD P means  $[A] \leftarrow [A] + [P]$ . The notation  $[A]$  indicates the contents of the accumulator.

The simple operation  $R = P + Q$  can be implemented by the following fragment of 8-bit code from a first-generation 6800 8-bit processor.

```
LDA P    ;load accumulator with P
ADD Q    ;add Q to accumulator
STA R    ;store accumulator in R
```

# THE ARM REGISTERS

**FIGURE 3.12** ARM register set



## Zero Address Machines

A zero address machine uses instructions that do not have an address at all.

A zero address machine operates on data that is at the top of a stack  
zero address machines are normally referred to as *stack machines*.

The code used to evaluate the expression  $Z = (A + B) * (C - D)$  might be written as:

PUSH A	Push A on stack
PUSH B	Push B on stack
ADD	Add top two items and push A+B on the stack
PUSH C	Push C on the stack
PUSH D	Push D on the stack
SUB	Subtract top two items and push C - D on the stack
MUL	Multiply top two items on stack (C - D), (A + B) push result
POP Z	Pull the top item off the stack (the result)

## Zero Address Machines

Stack machines can handle Boolean logic. Consider if  $(A < B)$  or  $(C = D)$ . This can be expressed as:

PUSH A	Push A on stack
PUSH B	Push B on stack
LT	Pull A and B and perform comparison. Push true or false
PUSH C	Push C
PUSH D	Push D
EQ	Push C and D and test for equality. Push true or false
OR	Pull top two Boolean values off stack. Perform OR push result.

The Boolean value on the stack can be used with a branch on true or a branch on false command as in the case of any other computer.

## One-and-a-half address machines

A CISC machine is called a *one-and-a-half address* machine because one operand is an address in memory and the other is a register. This 68K code demonstrates the evaluation of the expression  $(A+B)(C-D)$ .

```
MOVE A,D0    ;Load A from memory into register D0
ADD  B,D0    ;Add B from memory into register D0
MOVE C,D1    ;Load C from memory into register D1
SUB  D,D1    ;Subtract D from memory from register D1
MULU D0,D1   ;Multiply register D1 by D0
MOVE D1,X    ;Store register D1 in memory location X
```

Compare with the following code of an accumulator-based machine:

```
LDA A        ;Load A from memory into the accumulator
ADD B        ;Add B from memory into the accumulator
STA P        ;Store the accumulator in memory location P
LDA C        ;Load C from memory into the accumulator
SUB D        ;Subtract D from memory from the accumulator
MUL P        ;Multiply the accumulator by P from memory
STA X        ;Store the accumulator in memory location X
```



# ARM REGISTER SET

14 general-purpose registers r0 to r13.

r14 stores a subroutine return address

r15 contains the program counter.

Sixteen registers require a 4-bit address which saves three bits per instruction over RISC processors with 32-register architectures (5-bit address).

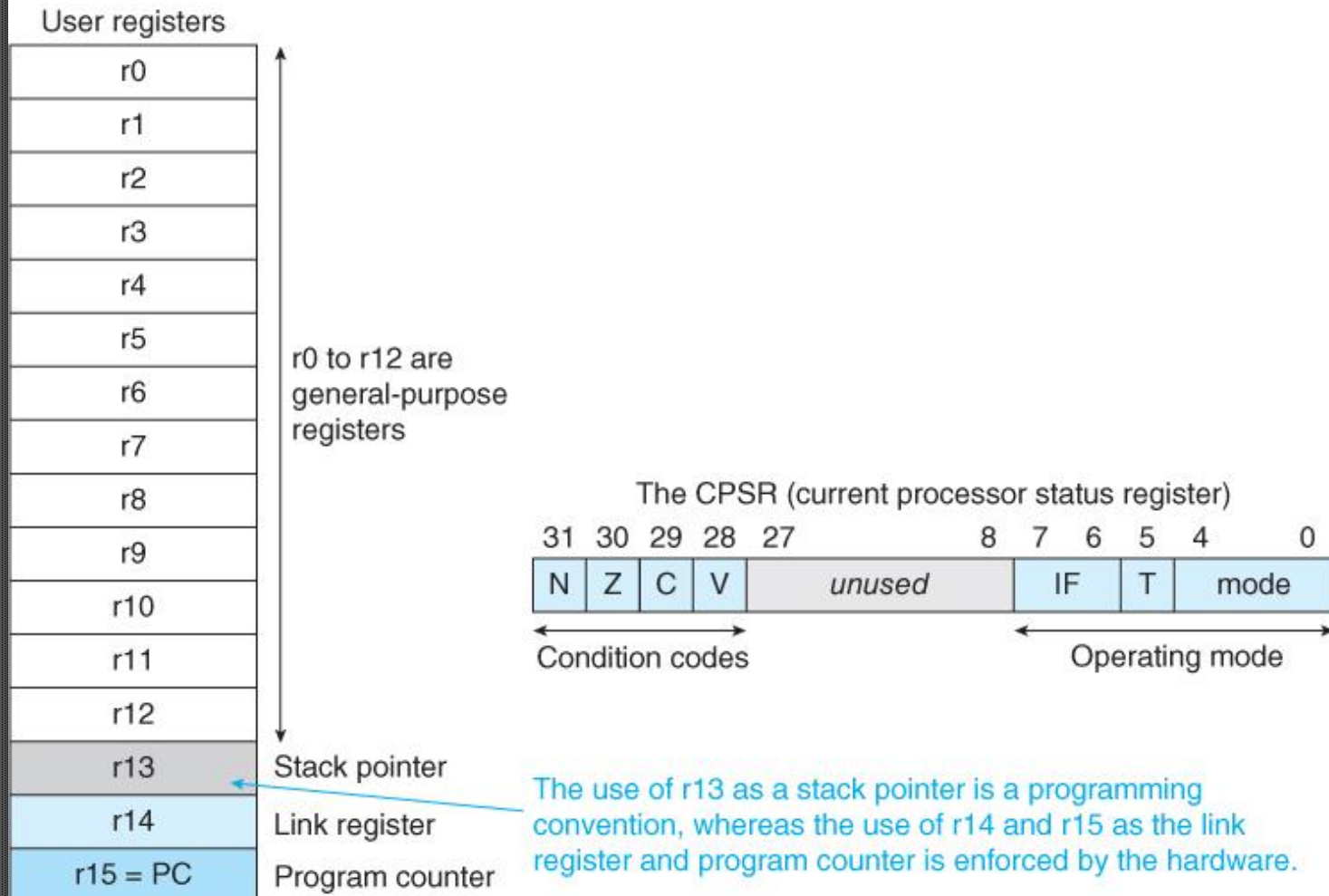
Register r13 is *reserved* for use by the programmer as the stack pointer.

The ARM's *current program status register* (CPSR) contains Z (zero), N (negative), C (carry) and V (overflow) flag bits

ARM processors have a rich instruction set  
Consider ADD **r1**,r2,r3,LSL r4 and MLA **r1**,r2,r3,r4.

# 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	$[PC] \leftarrow \text{label} \text{ (jump to label)}$

© 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.

```
Test_5 ADD r0,r1,r2 ;calculate TotalTime = Time + NewTime  
      SUBS r7,#1    ;Decrement loop counter  
      BEQ Test_5    ;IF zero THEN goto Test_5
```

The Label field is a user-defined label that can be used by other instructions to refer to that line.

Any text following a semicolon is regarded as a comment field and is ignored by the assembler.

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 = 1 to 10 (count down)
Next MUL r2,r1,r1	; square number
MLA r0,r2,r1,r0	; cube number and add 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 and implements the appropriate algorithm. It is not yet a program that we can run.

We have to specify where the code goes in memory.

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

(CODE WHITE, ASSEMBLER DIRECTIVES RED)

AREA ARMtest, CODE, READONLY  
ENTRY

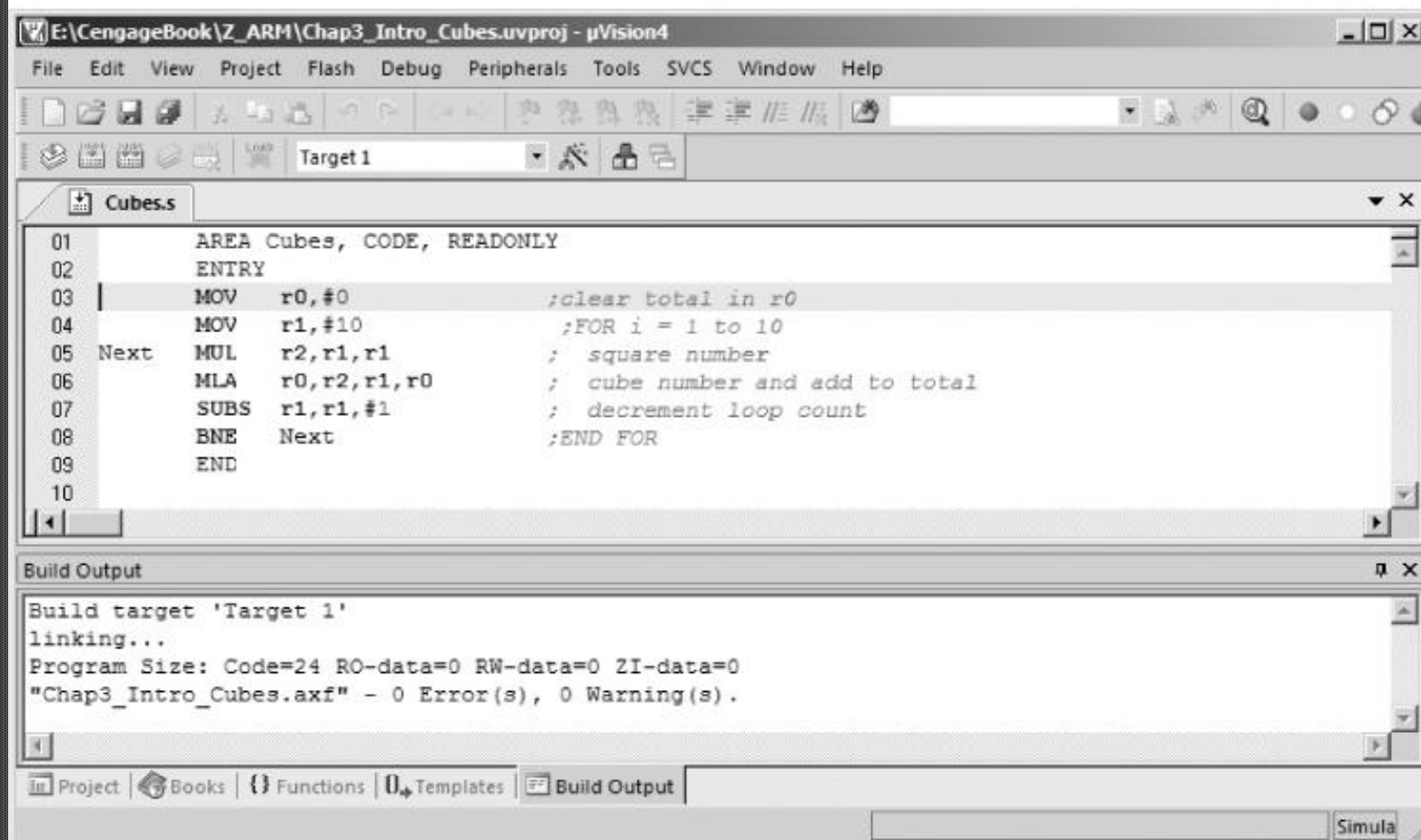
```
Next  MOV  r0,#0          ;clear total in r0
      MOV  r1,#10        ;FOR i = 1 to 10
      MUL  r2,r1,r1      ; square number
      MLA  r0,r2,r1,r0    ; cube number and add to total
      SUBS r1,r1,#1      ; decrement loop count
      BNE  Next          ;END FOR
```

END

# Snapshot of the Display of an ARM Development System

FIGURE 3.13

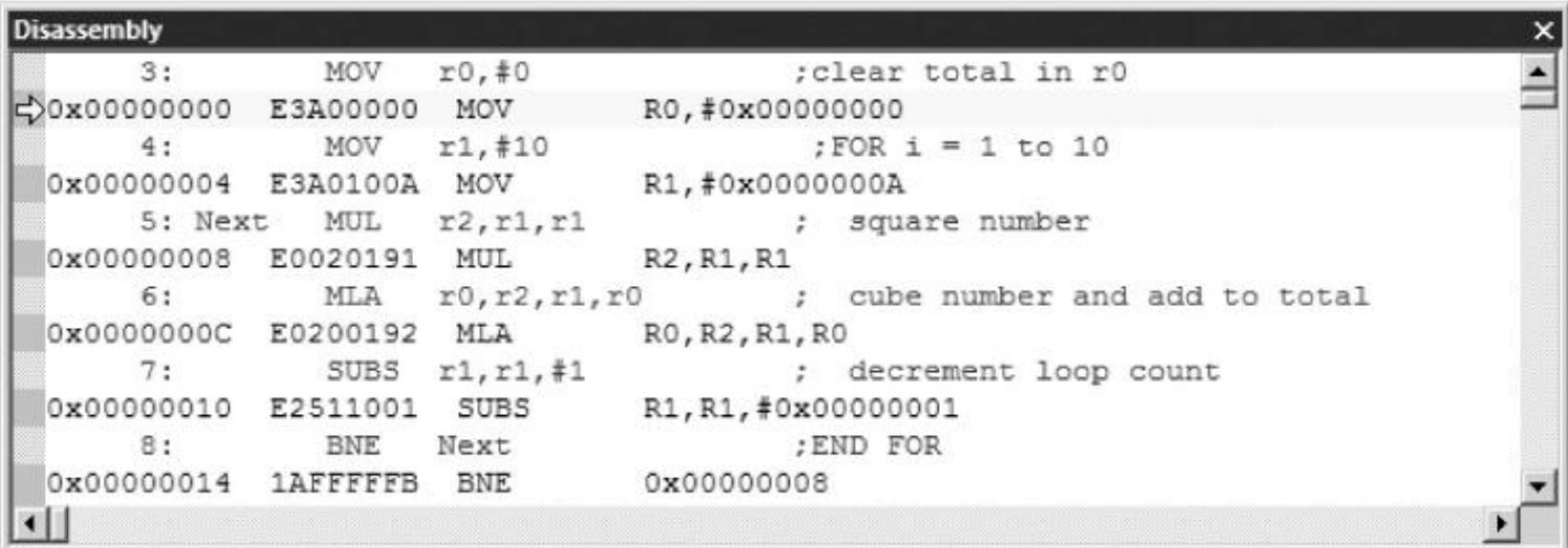
Assembling an assembly language program using Kiel's ARM IDE



This is the Disassembly Window that shows memory contents as both hexadecimal values and code.

**FIGURE 3.14**

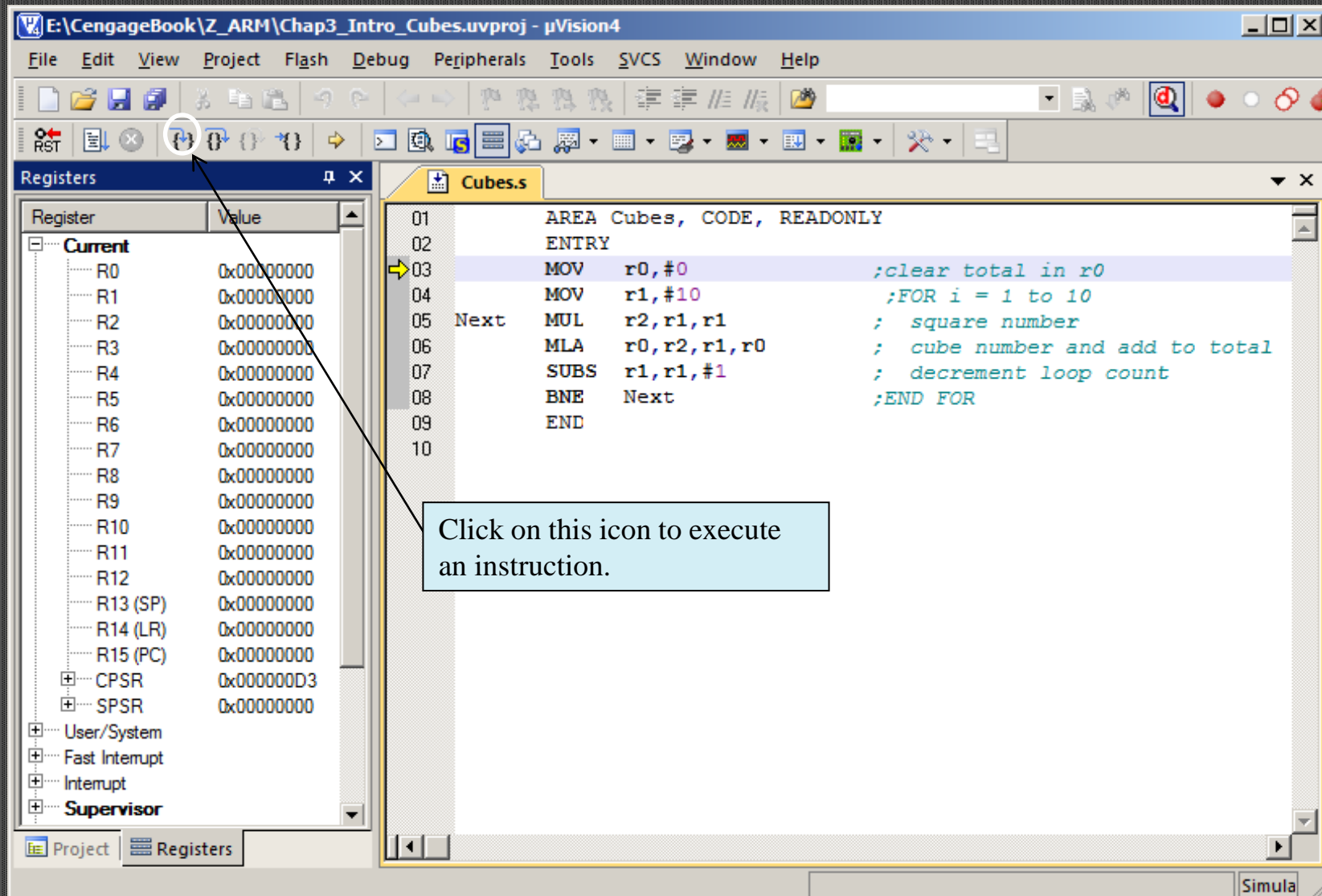
The disassembly window with the hexadecimal code generated by the program



```

Disassembly
3:      MOV    r0,#0                ;clear total in r0
⇒0x00000000 E3A00000 MOV          R0,#0x00000000
4:      MOV    r1,#10              ;FOR i = 1 to 10
0x00000004 E3A0100A MOV          R1,#0x0000000A
5: Next  MUL    r2,r1,r1           ; square number
0x00000008 E0020191 MUL          R2,R1,R1
6:      MLA    r0,r2,r1,r0         ; cube number and add to total
0x0000000C E0200192 MLA          R0,R2,R1,R0
7:      SUBS   r1,r1,#1            ; decrement loop count
0x00000010 E2511001 SUBS        R1,R1,#0x00000001
8:      BNE    Next                ;END FOR
0x00000014 1AFFFFFFB BNE         0x00000008
  
```

# Executing a program



The screenshot shows the uVision4 IDE interface. The title bar indicates the project is 'E:\CengageBook\Z\_ARM\Chap3\_Intro\_Cubes.uvproj - uVision4'. The menu bar includes File, Edit, View, Project, Flash, Debug, Peripherals, Tools, SVCS, Window, and Help. The toolbar contains various icons for file operations, debugging, and simulation. The 'Registers' window on the left lists registers R0 through R15, all with a value of 0x00000000, along with CPSR (0x000000D3) and SPSR (0x00000000). The main window displays the assembly code for '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
10
```

A callout box points to the 'Execute' icon in the toolbar, stating: "Click on this icon to execute an instruction."

At the bottom right, there is a 'Simula' button.

# Executing a program

The screenshot shows the uVision4 IDE with the project file `E:\CengageBook\Z_ARM\Chap3_Intro_Cubes.uvproj`. The **Registers** window on the left displays the current state of the ARM registers. The **Current** register set includes R0 through R15, CPSR, and SPSR. R2 is highlighted with a value of `0x00000064`, and R15 (PC) is highlighted with a value of `0x0000000C`. The main window shows the assembly code for `Cubes.s`. The code is as follows:

```

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
10

```

The execution is paused at line 06, which is highlighted in blue. The **Simula** button is visible at the bottom right of the IDE window.



The following slide demonstrates some assembly language directives (in red). These directives are:

EQU	equate	Equate a name to a value
DCD	define constant	Set up a 32-bit constant in memory
DCW	define constant	Set up a 16-bit constant in memory
DCB	define constant	Set up an 8-bit constant in memory
END		The physical end of the code
ENTRY		Starting point for execution
AREA		Names the region of code or data
ALIGN		Ensures that instructions are correctly aligned on 32-bit boundaries

**AREA Directives, CODE, READONLY****ENTRY**

```

MOV  r6,#XX      ;load r6 with 5 (i.e., XX)
LDR  r7,P1        ;load r7 with the contents of 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  DCD  0x12345678 ;store hex 32-bit value 1345678
P3  DCB  25        ;store the byte 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 = "Hello"
Strg2 = "X2", &0C, &0A
Z3   DCW  0xABCD
      END

```

## PSEUDOINSTRUCTIONS

A pseudo instruction is an operation that the programmer can use when writing code. The actual instruction does not exist. The assembler, generates suitable code to carry out the same action.

For example, you can't write `MOV r0,#0x1234567` to load register r0 with the 32-bit value 0x01234567 because an instruction is only 32 bits long in total.

The pseudoinstruction `ADR rdestination,label`, loads the 32-bit address of the line 'label' into a register.

The following fragment demonstrates the use of the ADR pseudoinstruction.

```
ADR  r1,MyArray    ;set up r1 to point to MyArray
...
LDR  r3,[r1]        ;read an element using the pointer
```

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

`ADR r1,MyArray` loads register r1 with the 32-bit address of MyArray using the appropriate code generated by the assembler. The programmer does not have to know how the assembler generates suitable code to implement the ADR.

Another useful pseudoinstruction is `LDR rd, = value`. The compiler generates the code that allows register `rd` to be loaded with the stated value; for example,

`LDR r0, = 0x12345678`

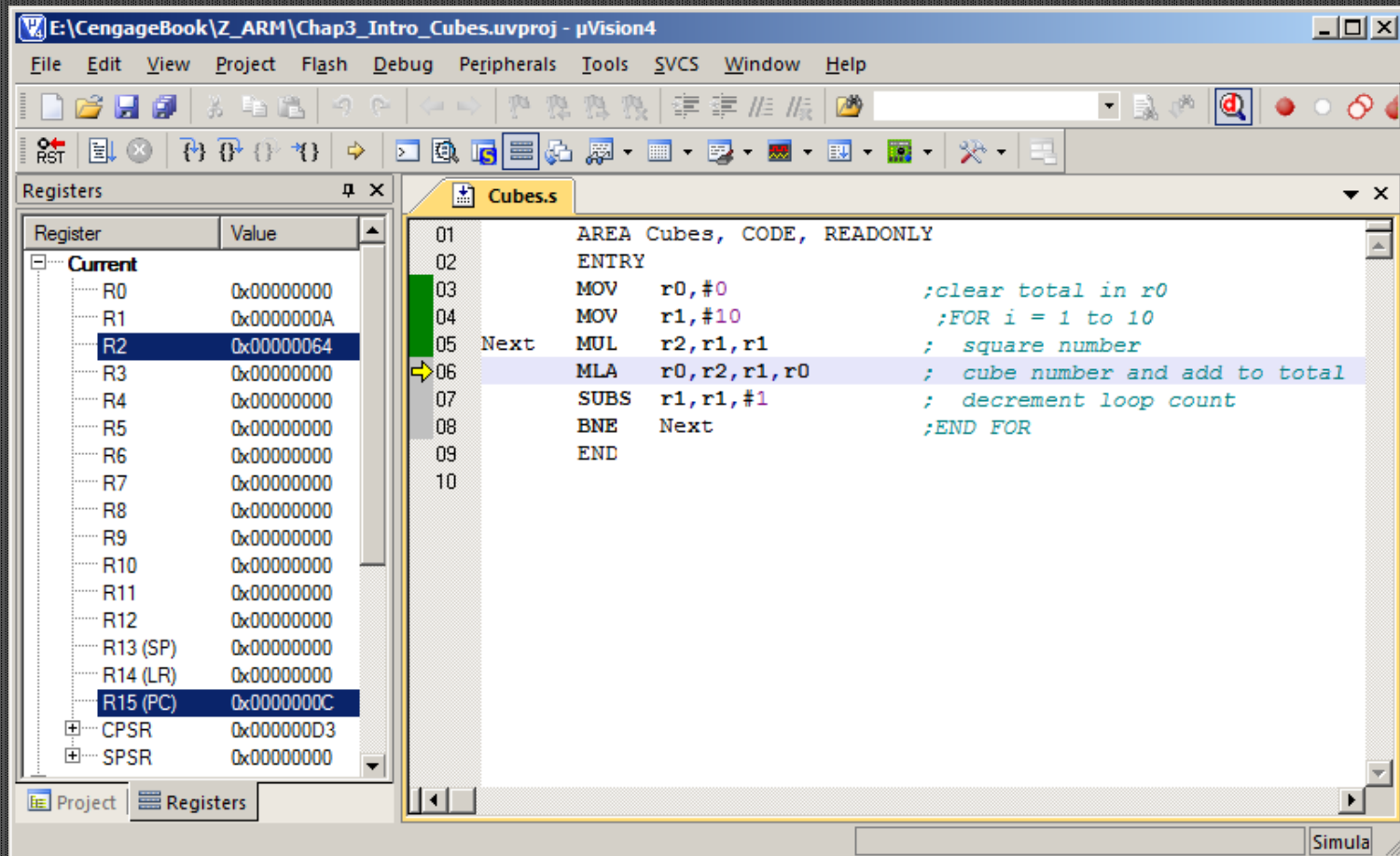
loads `r0` with  $12345678_{16}$ .

The assembler uses a `MOV` or `MVN` instruction if it can, or it uses an `LDR r0,[pc,#offset]` instruction to access the appropriate constant  $12345678_{16}$  that is stored in a so-called *literal pool* or *constant pool* somewhere in memory.

All this is done automatically.

## Executing Code in a Development System

This is the snapshot of the development system. It shows the code in source form and the contents of registers.





# Snapshot of a Debugger showing memory locations

The screenshot displays the ARM Debugger interface with the following components:

- Execution Window - CALC.S:** Shows assembly code for a calculator program. The current instruction is at line 23: `CMP r0, #'y'`. The code includes macros for writing and reading characters, and logic to calculate the sum of digits in a string.
- User Registers:** A table showing the values of 15 registers (r0-r14), the program counter (pc), and the current CPSR. r0 contains 0xffffffff, r1 contains 0x00000a44, and the pc is 0x00008080.
- Memory Window: 0xA000:** Displays a memory dump starting at address 0x009fe8, showing a sequence of zeros.
- Active Console Window:** Shows the program's output: `1234+`, `25=`, and `1259`.
- Taskbar:** Shows the Windows taskbar with the Start button and several open applications, including the debugger and a file explorer.

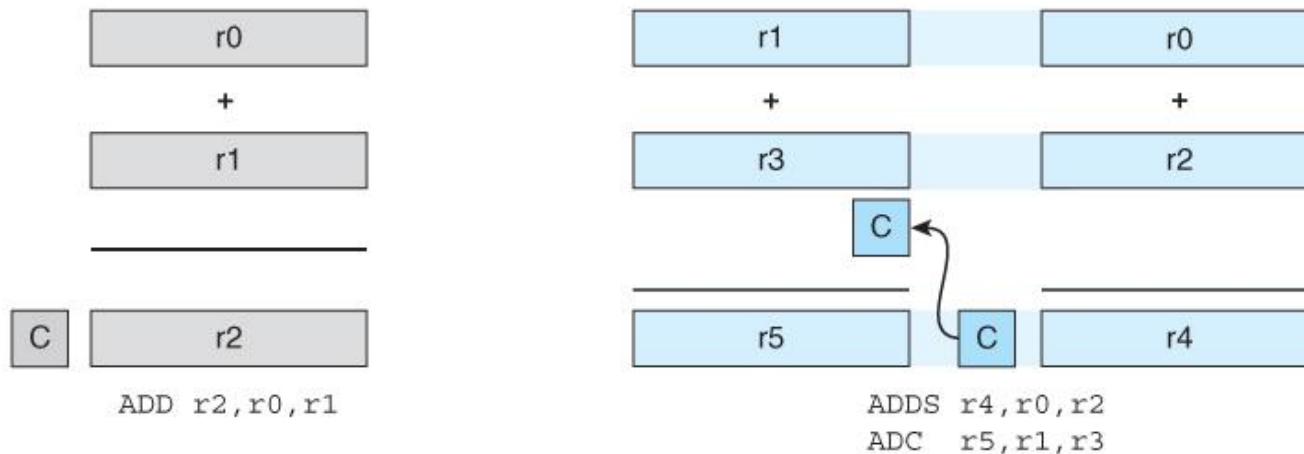
## Data-Processing Instructions

Addition	ADD
Subtraction	SUB
Negation	NEG
Comparison	CMP
Multiplication	MUL
Shifting	LSL, LSR, ASL, ASR, ROL, ROR

Note: The ARM does not have an explicit shift instruction but combines a shift with other operations.

As well as a simple ADD instruction that adds two 32-bit values, ARM has an ADC (add with carry) that adds to 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 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`.

© Cengage Learning 2014

## COMPARISON

CMP Q,P which evaluates  $Q - P$  but does not store the result;

```

    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
```

The multiply instruction, **MUL Rd,Rm,Rs**, calculates the product of two 32-bit signed integers in 32-bit registers Rm and Rs, then deposits the result in 32-bit register Rd, which stores the 32 lower-order bits of the 64-bit product.

```
MOV  r0,#121      ;load r0 with 121
MOV  r1,#96       ;load r1 with 96
MUL  r2,r0,r1     ;r2 = r0 x r1
```

you can't use the *same* register to specify both the destination Rd and the operand Rm, because ARM's implementation uses Rd as a temporary register during multiplication. This is a feature of the ARM processor.

ARM has a *multiply and accumulate* instruction, **MLA**, that performs a multiplication and adds the product to a running total. MLA instruction has a four-operand form: **MLA Rd,Rm,Rs,Rn**, whose RTL definition is  $[Rd] = [Rm] \times [Rs] + [Rn]$ . A 32-bit by 32-bit multiplication is truncated to the lower-order 32 bits.



ARM's *multiply and accumulate* supports the calculation of an inner product by performing one multiplication and addition per instruction. The inner product is used in multimedia applications; for example, if vector **a** consists of  $n$  components  $a_1, a_2, \dots, a_n$  and vector **b** consists of the  $n$  components  $b_1, b_2, \dots, b_n$ , then the *inner product* of **a** and **b** is the scalar value

$$s = \mathbf{a} \cdot \mathbf{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n.$$

```

MOV    r4, #n           ;r4 is the loop counter
MOV    r3, #0           ;clear the inner product
ADR    r5, Vector1      ;r5 points to vector 1
ADR    r6, Vector2      ;r6 points to vector 2

Loop   LDR    r0, [r5], #4 ;REPEAT read a component of A and
                           ;update the pointer
        LDR    r1, [r6], #4 ; get the second element
        MLA    r3, r0, r1, r3 ; add new product term to the total
                           ;(r3 = r3 + r0·r1)
        SUBS   r4, r4, #1  ; decrement the loop counter
                           ; (and remember to set the CCR)
        BNE    Loop       ;UNTIL all done

```

## BITWISE LOGICAL OPERATIONS

Instruction	Operation	Final value in r2
AND r2,r1,r0	11001010.00001111	00001010
OR r2,r1,r0	11001010+00001111	11001111
NOT r2,r1	<u>11001010</u>	00110101
EOR r2,r1,r0	11001010 $\oplus$ 00001111	11000101

Although ARM lacks an explicit NOT instruction, you can perform a NOT by using an EOR with the second operand equal to FFFFFFFF<sub>16</sub> (32 1's in a register) because the value of  $x \oplus 1$  is NOT  $x$ . A NOT operation can also be implemented with the *move negated instruction* MVN, that copies the *logical complement* of a value into a register.

Suppose that register r0 contains the 8 bits bbbbbbxx, register r1 contains the bits bbbyyybb and register r2 contains the bits zzzbbbbbb, where x, y, and z represent the bits of desired fields and the b's are unwanted bits. We wish to pack these bits to get the final value zzzzyyyxx. We can achieve this by:

```
ANDr0,r0,#2_00000011 ;Mask r0 to two bits xx
ANDr1,r1,#2_00011100  ;Mask r1 to three bits yyy
ANDr2,r2,#2_11100000  ;Mask r2 to three bits zzz
OR  r0,r0,r1           ;Merge r1 and r0 to get 000yyyxx
OR  r0,r0,r2           ;Merge r2 and r0 to get zzzzyyyxx
```

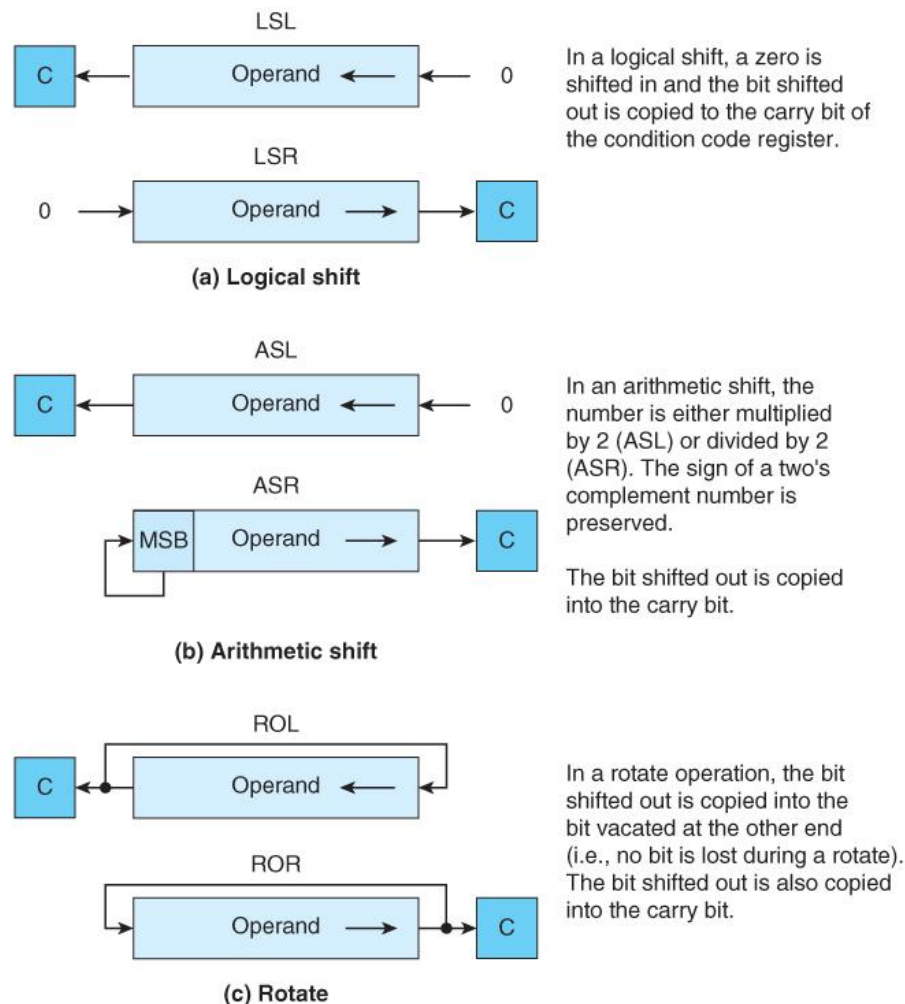
Shift operations move bits one or more places left or right.  
Logical shifts insert a 0 in the vacated position.

### Examples of logical shifts

Source string	Direction	Number of shifts	Destination string
0110011111010111	Left	1	1100111110101110
0110011111010111	Left	2	1001111101011100
0110011111010111	Left	3	0011111010111000
0110011111010111	Right	1	0011001111101011
0110011111010111	Right	2	0001100111110101
0110011111010111	Right	3	0000110011111010

Arithmetic shifts replicate the sign-bit during a right shift  
Circular shifts treat the register as a ring and the bit shifted out of one end is shifted in the other end.

**FIGURE 3.23** Shift operations





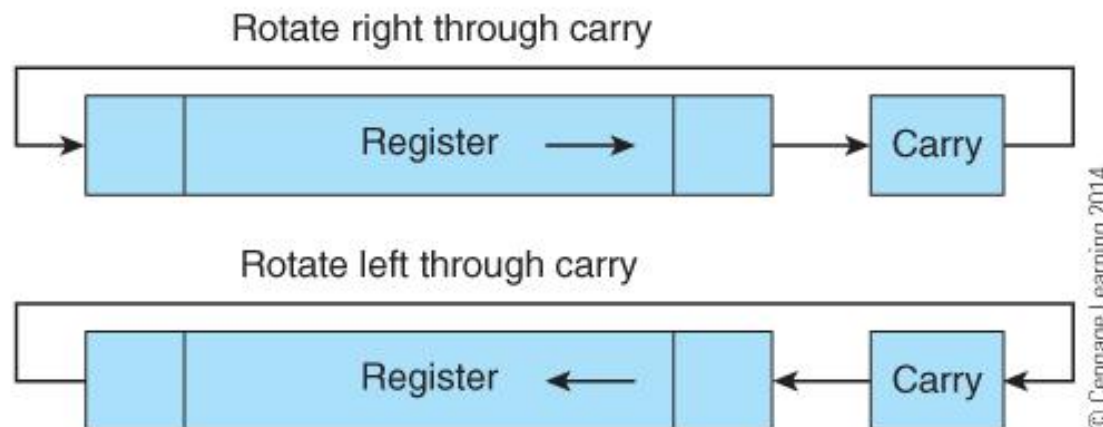
The rotate through carry instruction (sometimes called extended shift) included the carry register in the shift path.

The carry bit is shifted into the bit of the word vacated, and the bit of the word shifted out is shifted into the carry.

In eight bits, if the carry  $C = 1$  and the word to be shifted is 01101110, a rotate left through carry would give

11011101 and carry = 0

**FIGURE 3.24** The rotate through carry



## IMPLEMENTING A SHIFT OPERATION ON THE ARM

ARM combines shifting with other data processing operations, because the second operand can be shifted before it is used. Consider:

**ADD r0,r1,r2, LSL #1**

A logical shift left is applied to the contents of r2 before they are added to the contents of r1. This operation is equivalent to

$$[r0] \leftarrow [r1] + [r2] \times 2.$$

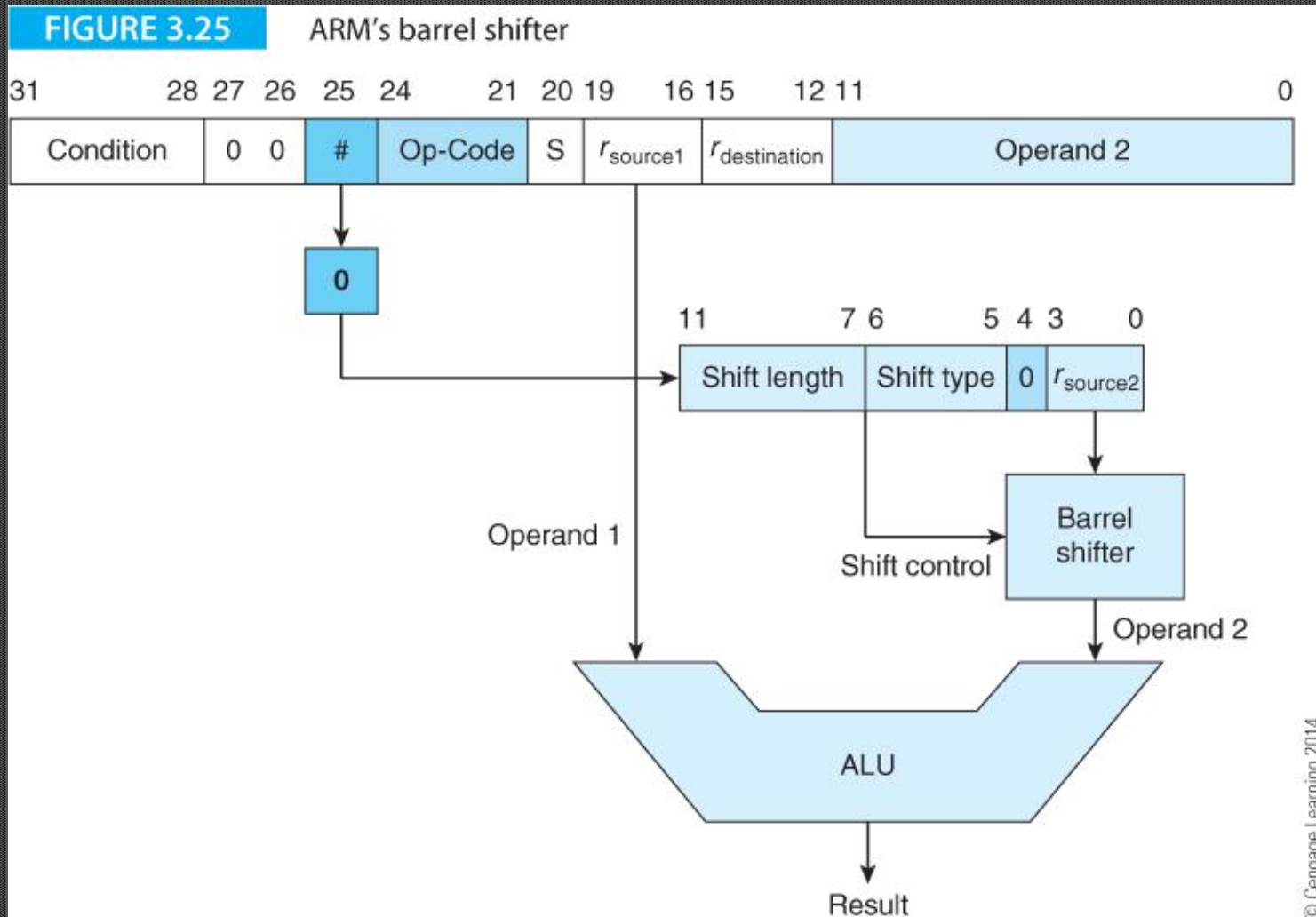
To apply a shift operation to a register without any other data processing, you can a move

**MOV r3,r3 LSL #1.**

You can perform *dynamic shifts*. Consider **MOV r4,r3, LSL r1**, which moves the contents of r3 left by the value in r1 before putting the result in r4.

Suppose a number in r0 is of the form 0.00000010101111... and you want to normalize it to 0.101... If register r1 contains the exponent, we can execute **MOV r0,r0,LSL r1** to perform the normalization operation in a single cycle.

Figure 3.25 illustrates the structure of instructions with shifted operands and shows how the various fields control the shifter and the ALU.

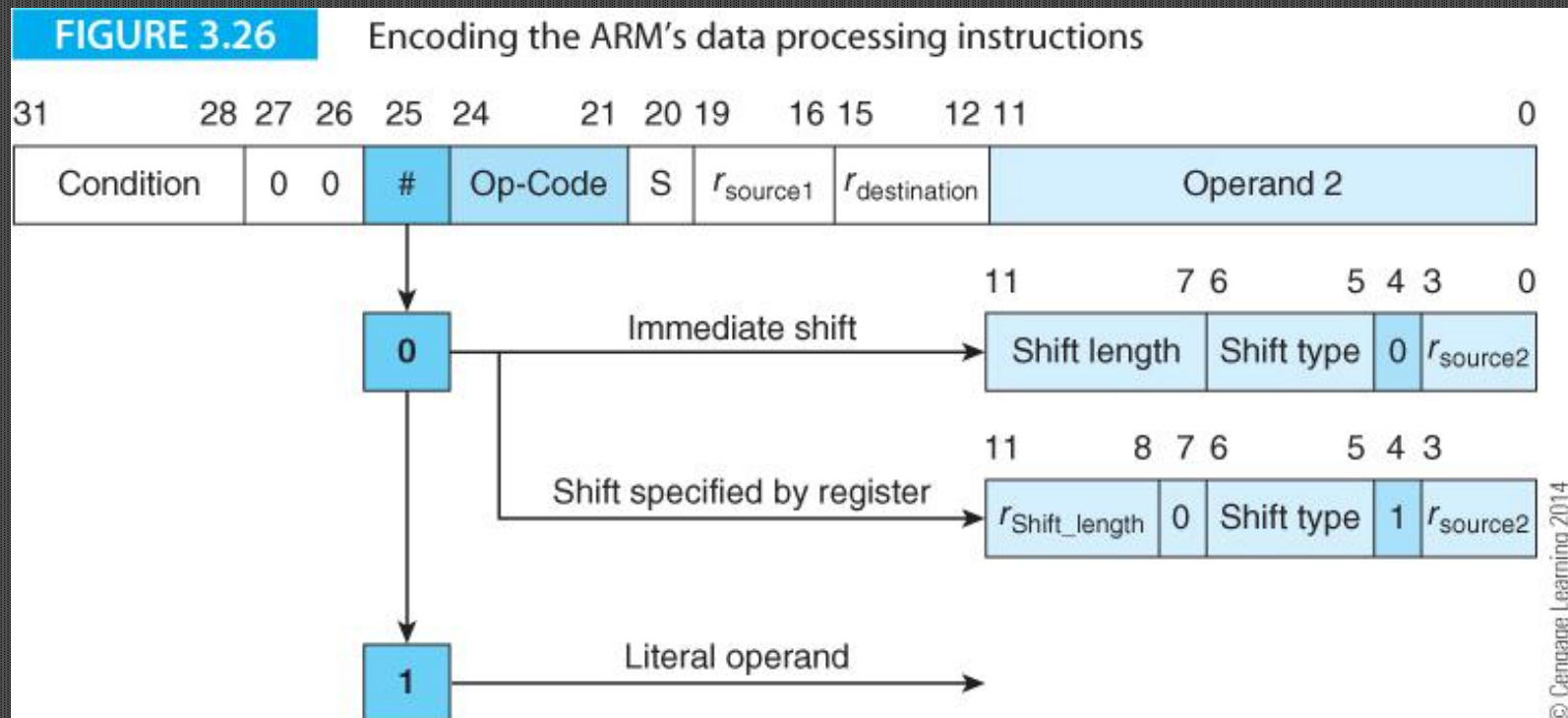


ARM implements only the following five shifts (the programmer can synthesize the rest).

LSL	logical shift left	
LSR	logical shift right	
ASR	arithmetic shift right	
ROR	rotate right	
RRX	rotate right through carry	(one shift)

Other shift operations have to be synthesized by the programmer.

Figure 3.26 illustrates the structure of the ARM's data processing instructions and demonstrates how bit 25 is used to control the nature of the second source operand.





# UNCONDITIONAL BRANCH

ARM's unconditional branch instruction has the form B target, where target denotes the *branch target address* (BTA, the address of the next instruction to be executed). The following fragment of code demonstrates how the unconditional branch is used.

```
..    do this        Some code
..    then that      Some other code
      B      Next    Now skip past next instructions
..                                     ...the code being skipped past
..                                     ...the code being skipped past
Next ..          Target address for the branch
```

## CONDITIONAL BRANCH

```
IF (X == Y)
  THEN Y = Y + 1;
  ELSE Y = Y + 2
```

A test is performed and one of two courses of action is carried out depending on the outcome. We can translate this as:

```
    CMP r1,r2      ; r1 contains y and r2 contains x: compare them
    BNE Plus2     ;if not equal then branch to the else part
    ADD r1,r1,#1   ;if equal fall through to here and add one to y
    B leave        ;now skip past the else part
Plus2 ADD r1,r1,#2 ;ELSE part add 2 to y
leave ...          ;continue from here
```

The *conditional branch* instruction tests flag bits in the processor's condition code register, then takes the branch if the tested condition is true. There are eight possible conditional branches based on the state of a single bit (four that branch on true and four that branch on false).

# ARM's BRANCHES

TABLE 3.2

ARM's Conditional Execution and Branch Control Mnemonics

Encoding	Mnemonic	Branch on Flag Status	Execute on condition
0000	EQ	Z set	Equal (i.e., zero)
0001	NE	Z clear	Not equal (i.e., not zero)
0010	CS	C set	Unsigned higher or same
0011	CC	C clear	Unsigned lower
0100	MI	N set	Negative
0101	PL	N clear	Positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear or Z set	Unsigned lower or same
1010	GE	N set and V set, or N clear and V clear	Greater or equal
1011	LT	N set and V clear, or N clear and V set	Less than
1100	GT	Z clear, and either N set and V set, or N clear and V clear	Greater than
1101	LE	Z set, or N set and V clear, or N clear and V set	Less than or equal
1110	AL		Always (default)
1111	NV		Never (reserved)

## BRANCHING AND LOOP CONSTRUCTS

Nothing illustrates the concept of flow control better than the classic loop constructs that are at the core of so-called structured programming. The following demonstrate the structure of the FOR, WHILE and UNTIL loops.

### The FOR loop

```
      MOV    r0, #10      ;set up the loop counter
Loop  code ...           ;body of the loop

      SUBS   r0, r0, #1   ;decrement loop counter, set flags
      BNE   Loop         ;continue until count zero
Post loop ...           ;fall through on zero count
```

## CONDITIONAL EXECUTION

One of ARM's most unusual features is that each instruction is *conditionally executed*. We can associate an instruction with a logical condition.

If the stated condition is true, the instruction is executed. Otherwise it is bypassed (*annulled* or *squashed*).

The assembly language programmer indicates the conditional execution mode by appending the appropriate condition to a mnemonic; for example,

**ADDEQ** r1,r2,r3

specifies that the addition is performed only if the Z-bit is set because a previous result was zero. The RTL form of this operation is

IF Z = 1 THEN  $[r1] \leftarrow [r2] + [r3]$



## CONDITIONAL EXECUTION

There is nothing to stop you combining conditional execution and shifting because the branch and shift fields of an instruction are independent. You can write

**ADDCC r1,r2,r3, LSL r4**

which is interpreted as IF  $C = 0$  THEN  $[r1] \leftarrow [r2] + [r3] \times 2^{[r4]}$

ARM's conditional execution mode makes it easy to implement conditional operations in a high-level language.

Consider the following fragment of C code.

```
if (P == Q) X = P - Y ;
```

If we assume that r1 contains P, r2 contains Q, r3 contains X, and r4 contains Y, then we can write

```
CMP    r1,r2        ;compare P == Q  
SUBEQ  r3,r1,r4      ;if (P == Q) then r3 = r1 - r4
```

Notice how this operation is implemented without using a branch by squashing instructions we don't wish to execute rather than branching round them. In this case the subtraction is squashed if the comparison is false

Now consider a more complicated example of a C construct with a compound predicate:

```
if ((a == b) && (c == d)) e++;
```

```
CMP    r0,r1      ;compare a == b
CMPEQ  r2,r3      ;if a == b then test c == d
ADDEQ  r4,r4,#1   ;if a == b AND c == d THEN increment e
```

The first line, `CMP r0,r1`, compares `a` and `b`.

The next line, `CMPEQ r2,r3`, executes a conditional comparison only if the result of the first line was true (i.e., `a == b`).

The third line, `ADDEQ r4,r4,#1`, is executed only if the previous line was true (i.e., `c == d`) to implement the `e++`.

You can also handle some testing with multiple conditions.  
Consider:

```
if (a == b) e = e + 4;  
if (a < b)  e = e + 7;  
if (a > b)  e = e + 12;
```

We can use conditional execution to implement this as

```
CMP    r0,r1      ;compare a == b  
ADDEQ  r4,r4,#4   ;if a == b then e = e + 4  
ADDLE  r4,r4,#7   ;if a < b  then e = e + 7  
ADDGT  r4,r4,#12  ;if a > b  then e = e + 12
```

## ADDRESSING MODES

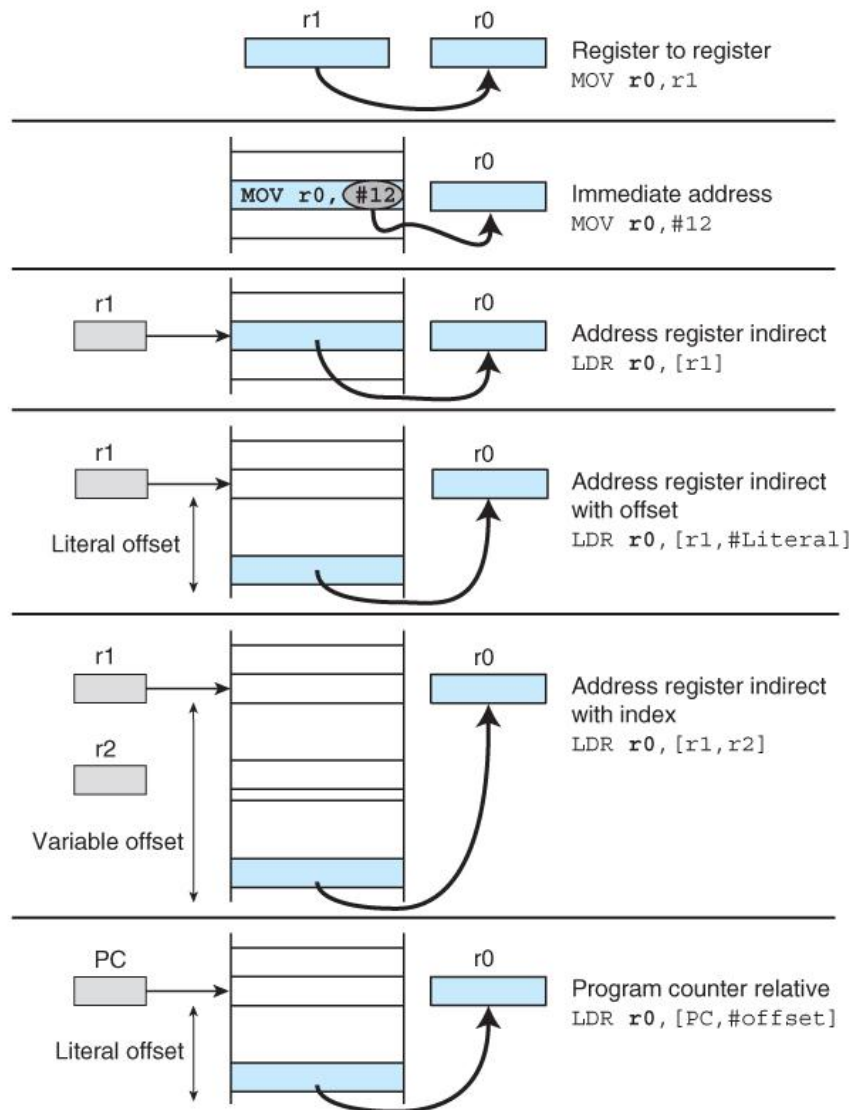
<i>Mnemonic</i>	<i>RTL form</i>	<i>Description</i>
ADD <b>r0</b> ,r1,#Q	$[r0] \leftarrow [r1] + Q$	<i>Literal</i> : Add the integer Q to contents of register r1
LDR <b>r0</b> ,Mem	$[r0] \leftarrow [\text{Mem}]$	<i>Absolute</i> : Load contents of memory location Mem into register r0. This addressing mode is not supported by ARM but is supported by all CISC processors
LDR <b>r0</b> ,[r1]	$[r0] \leftarrow [[r1]]$	<i>Register Indirect</i> : Load r0 with the contents of the memory location pointed at by r1

The ARM lacks a simple memory direct (i.e., absolute) addressing mode and 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.



# Concepts of Addressing Modes

**FIGURE 3.27** Summary of addressing modes that can be used by ARM



© Cengage Learning 2014

# Handling Literals

ARM is able to use literal operands.

ADD r0,r1,#7 adds 7 to r1 and puts the result in r0.

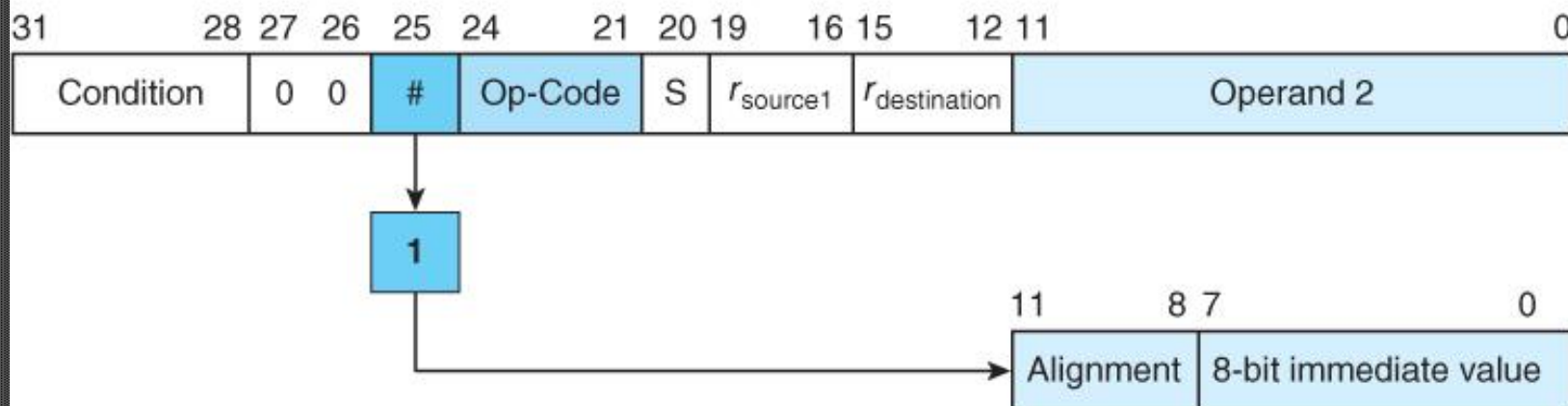
MOV r3,#25 moves 25 into r3.

Literals are 12 bit values in the range 0 to 4095.

Literals can be scaled by a power of 2 (an unusual feature of the ARM).

Figure 3.28 illustrate the format of ARM's instructions with a literal operand.

**FIGURE 3.28** Diagram of ARM's literal operand encoding

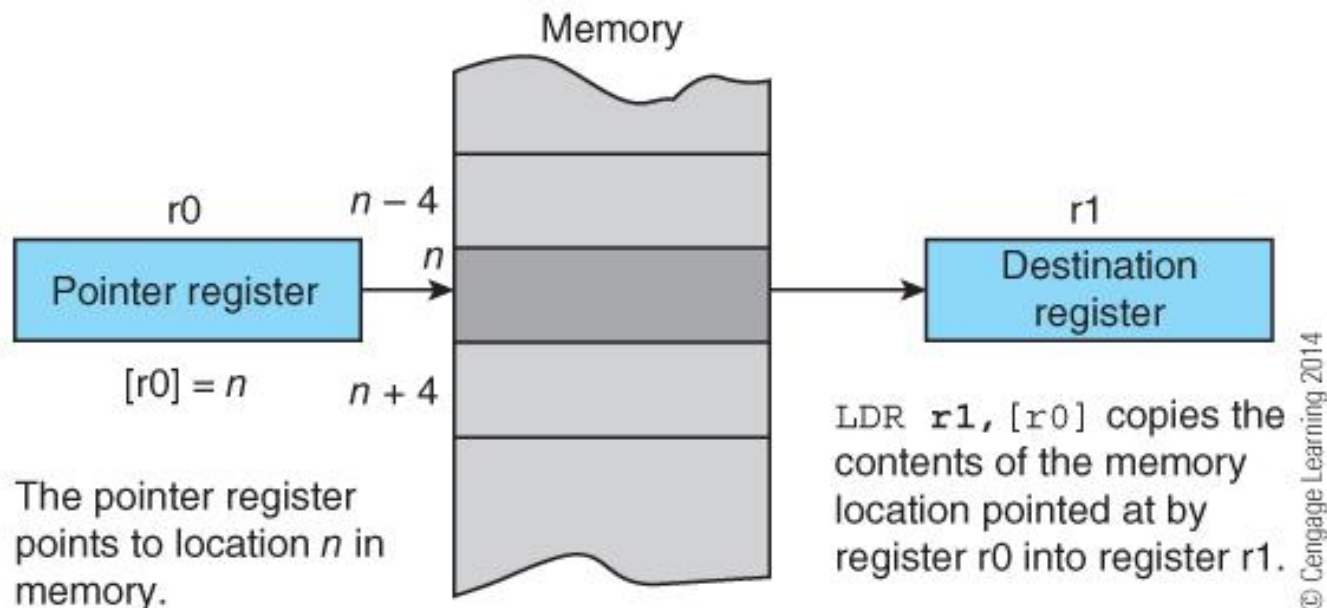


# Register Indirect Addressing

In register indirect addressing, the location of an operand is given by the contents of a register.

All computers support some form of register indirect addressing.

**FIGURE 3.31** Register indirect addressing



In register indirect addressing, the location of an operand is given by the contents of a register. All computers support some form of register indirect addressing. This is also called:

- Indexed
- Pointer-based

The ARM indicates register indirect addressing by means of square brackets; for example,

`LDR r1,[r0]` ;load r1 with the contents of the memory location pointed  
;at by r0

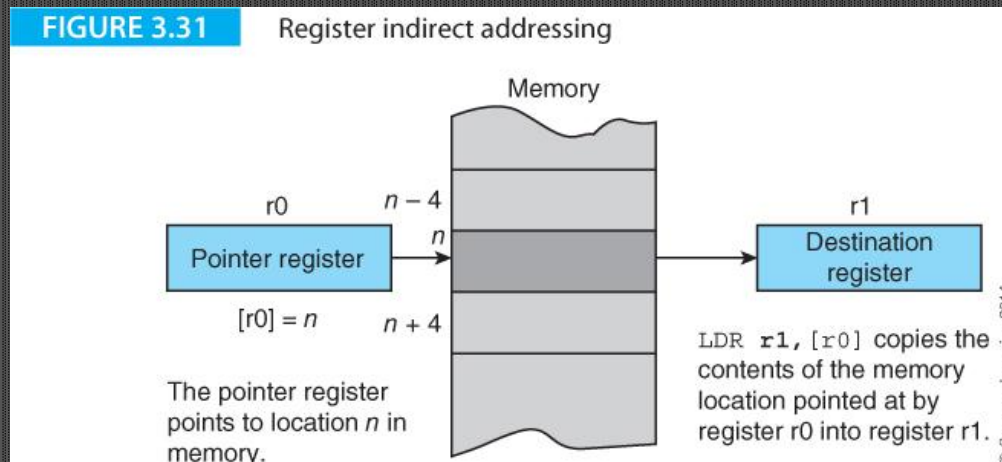
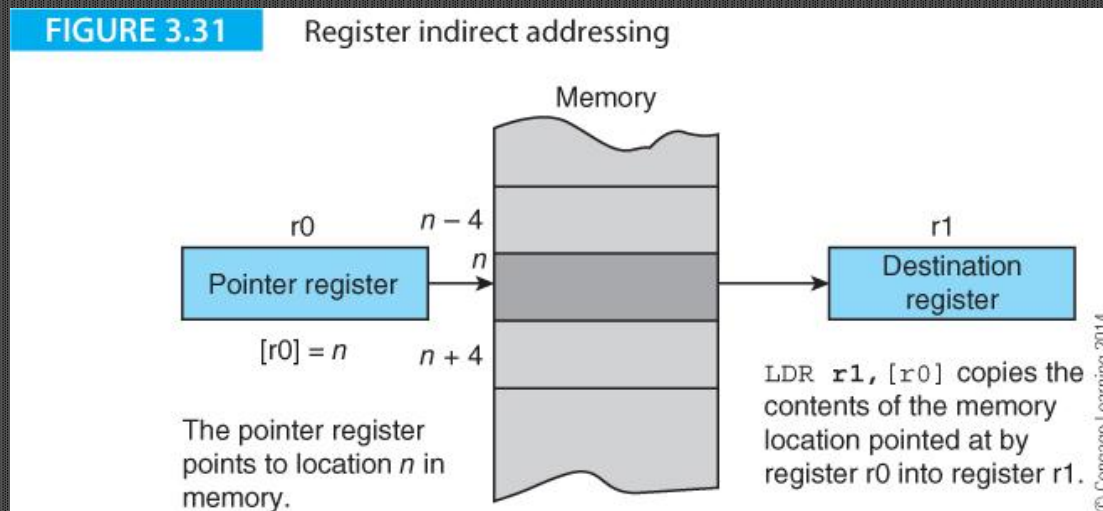


Figure 3.31 shows the execution of  
`LDR r1,[r0]` ;load r1 with the contents of the memory location pointed  
;at by r0



Consider what happens if we next execute

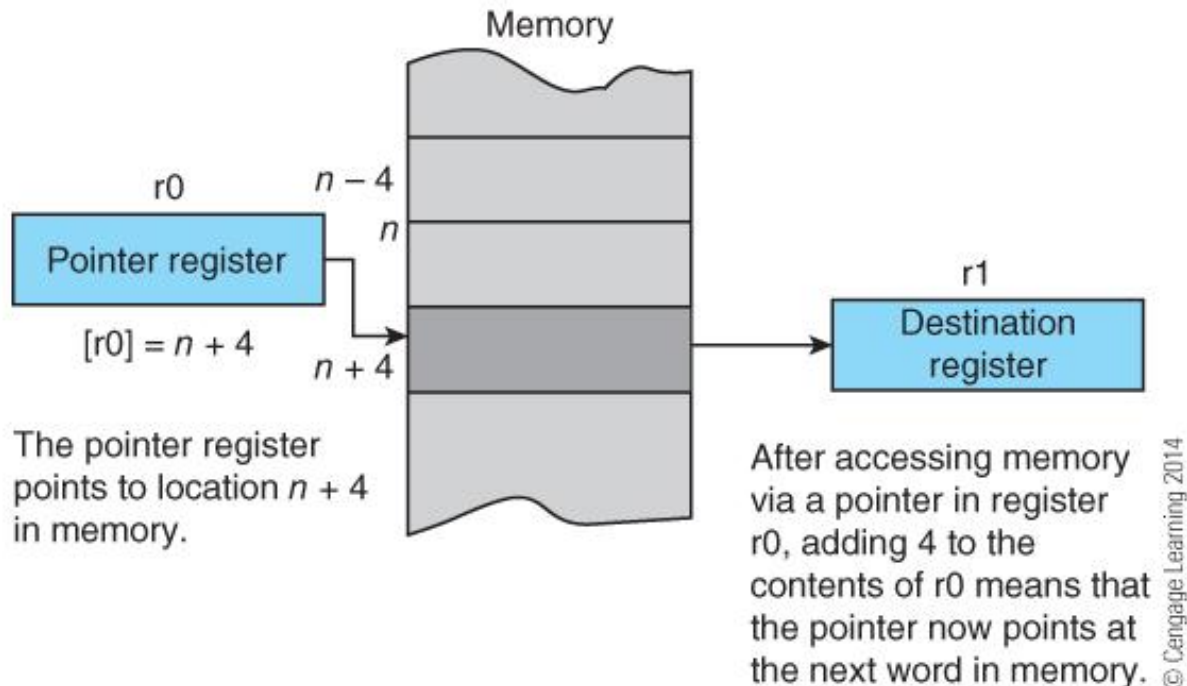
`ADD r0,r0,#4` ;Add 4 to the contents of register `r4`  
;(i.e., increment the pointer by one word)



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 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



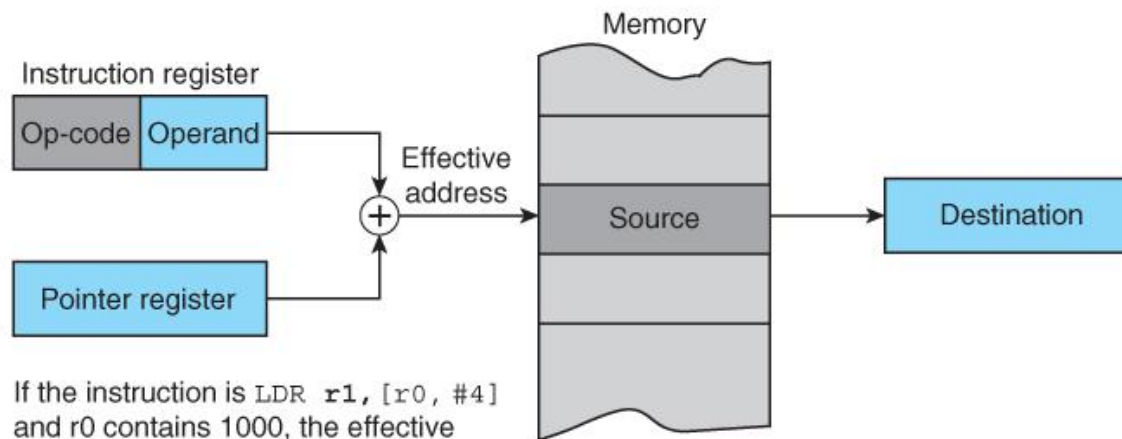
## Register Indirect Addressing with an Offset

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* coded into the load/store instruction.

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 contents of the pointer register r1 plus offset 4; that is, the operand is 4 bytes on from the address specified by the pointer.

**FIGURE 3.33** Register indirect addressing with an offset



If the instruction is `LDR r1, [r0, #4]` and r0 contains 1000, the effective address of the source operand is  $1000 + 4 = 1004$ .

The following fragment of code demonstrates the use of offsets to implement array access. Because the offset is a constant, it cannot be changed at runtime.

```
Sun    EQU 0           ;offsets for days of the week
Mon    EQU 4
Tue    EQU 8
.
Sat    EQU 24

      ADR r0, week      ;r0 points to array week
      LDR r2,[r0,#Tue]  ;read the data for Tuesday into r2

Week   DCD              ;data for day 1 (Sunday)
       DCD              ;data for day 2 (Monday)
       DCD              ;data for day 3 (Tuesday)
       DCD              ;data for day 4 (Wednesday)
       DCD              ;data for day 5 (Thursday)
       DCD              ;data for day 6 (Friday)
       DCD              ;data for day 7 (Saturday)
```

Snapshot of the program using register indirect addressing with an offset.

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

- Registers Window:** Shows the current state of registers. R15(PC) is highlighted with a value of 0x00000014.
- Disassembly Window:** Shows the assembly code being executed. Instruction 15 (NOP) is highlighted.
- Source Window:** Shows the assembly code for DaysOfWeek.asm. Instructions 10 through 15 are highlighted.

**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
<b>R15 (PC)</b>	<b>0x00000014</b>
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
<b>Supervisor</b>	
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 Window (DaysOfWeek.asm):**

```

01      AREA DaysOfWeek, CODE, READONLY
02      EQU 0              ;0 - offsets for days of the week
03      EQU 4              ;4
04      EQU 8              ;8
05      EQU 0xC            ;12
06      EQU 0x10           ;16
07      EQU 0x14           ;20
08      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
  
```

# Register Indirect Addressing with Base and Index Registers

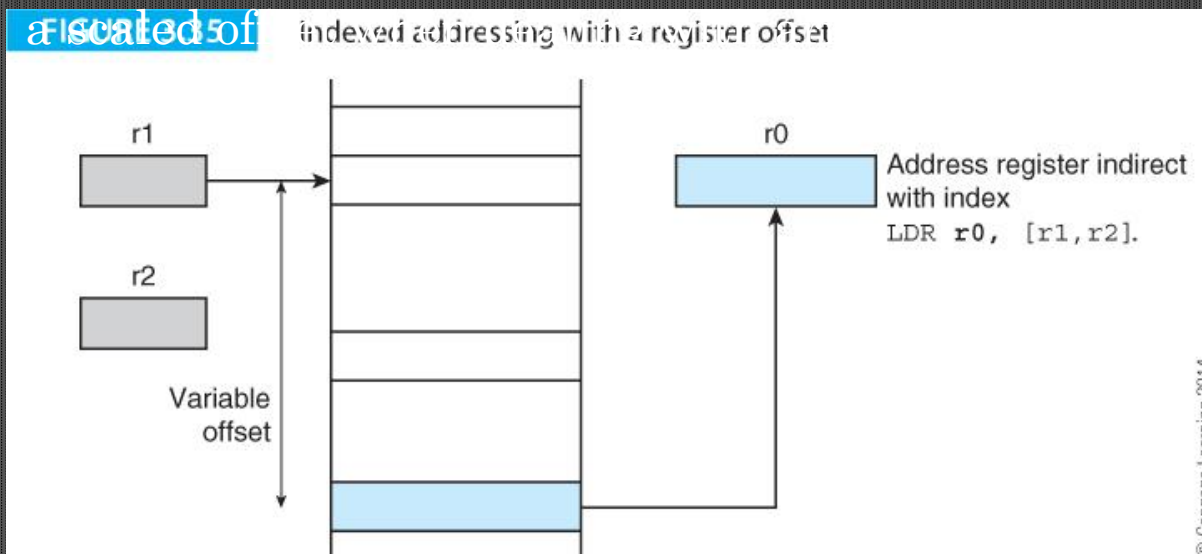
You can specify the offset as a second register so that you can use a dynamic offset that can be modified at runtime (See Figure 3.35).

**LDR r2,[r0,r1]**       $:[r2] \leftarrow [[r0] + [r1]]$  load r2 with the location  
;pointed at by r0 plus r1

**LDR r2,[r0,r1,LSL #2]**       $:[r2] \leftarrow [[r0] + 4 \times [r1]]$  Scale r1 by 4

In the second example, register r1 is scaled by 4. This allows you to use

a scaled of Indexed addressing with a register offset





# Pre-indexing (register indirect with a constant/literal)

The screenshot displays the ARM Debugger interface with the following components:

- Execution Window - ARM\_CH3\_INDEXADDRESS.S:**

```

1      AREA INDEXING, CODE, READONLY
2      ENTRY
3      i      EQU      2*4          ;let's make i = 2 for this test
4      ADR     r0,X                ;register r0 points at array X
5      LDR     r1,[r0,#i]          ;get element i
6
7      X      DCD     0x01234567    ;store hex 32-bit value 01234567
8      DCD     0x89ABCDEF
9      DCD     0xABABABAB
10     DCD     0xFFAABBCC
11     END

```
- User Registers:**

r0	0x00008088
r1	0xabababab
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
pc	0x00008088
cpsr	%nzcvi0t User32
- Memory Window: 0x8088 (1):**

0x008088	0x01234567
0x00808c	0x89abcdef
0x008090	0xabababab
0x008094	0xffaabbcc
0x008098	0x00000000
0x00809c	0x00000000
0x0080a0	0x00000000
0x0080a4	0x00000000
0x0080a8	0x00000000
0x0080ac	0x00000000

Step completed

## AUTOINDEXING PRE-INDEXED ADDRESSING MODE

Elements in an array or similar data structure are frequently accessed sequentially. Auto-indexing addressing modes in which the pointer is automatically adjusted to point at the next element *before* or *after* it is used have been implemented.

ARM implements two auto-indexing modes by adding the offset to the base (i.e., pointer register).

ARM's *autoindexing pre-indexed* addressing mode is indicated by appending the suffix “!” to the effective address. Consider the following ARM instruction:

LDR r0,[r1,#8]!      ;load r0 with the word pointed at by register r1  
                         ; plus 8 *then* 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

## EXAMPLE OF PRE-INDEXED ADDRESSING MODE

This *auto-indexing mode* does not incur additional execution time, because it is performed in parallel with memory access.

Consider this example of the addition of two arrays.

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

Register	Value
<b>Current</b>	
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
<b>R15 (PC)</b>	<b>0x00000028</b>
+ CPSR	0x600000D3
+ SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
<b>Supervisor</b>	
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:	
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

## AUTOINDEXING POST-INDEXING MODE

Autoindexing *post-indexing* addressing first accesses the operand at the location pointed to by the base register, then increments the base register.

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

Post-indexing is denoted by placing the offset *outside* the square. The RTL definition of this instruction is:

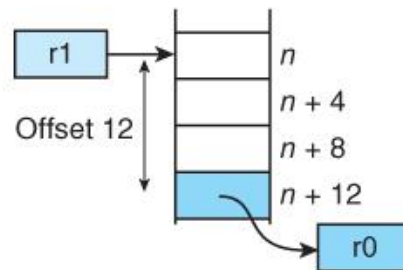
$[r0] \leftarrow [[r1]]$	Access the memory address in base register r1
$[r1] \leftarrow [r1] + 8$	Update pointer (base register) by adding offset



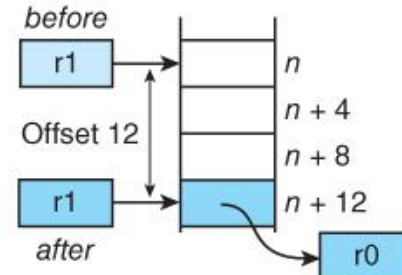
# AUTOINDEXING POST-INDEXING MODE

Autoindexing *post-indexing* addressing first accesses the operand at the location

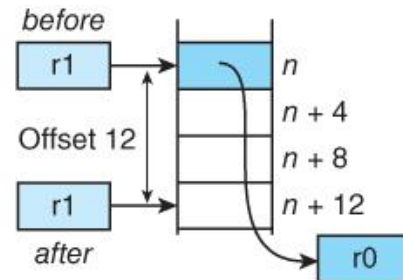
**FIGURE 3.38** Register indirect addressing with offset



- (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.



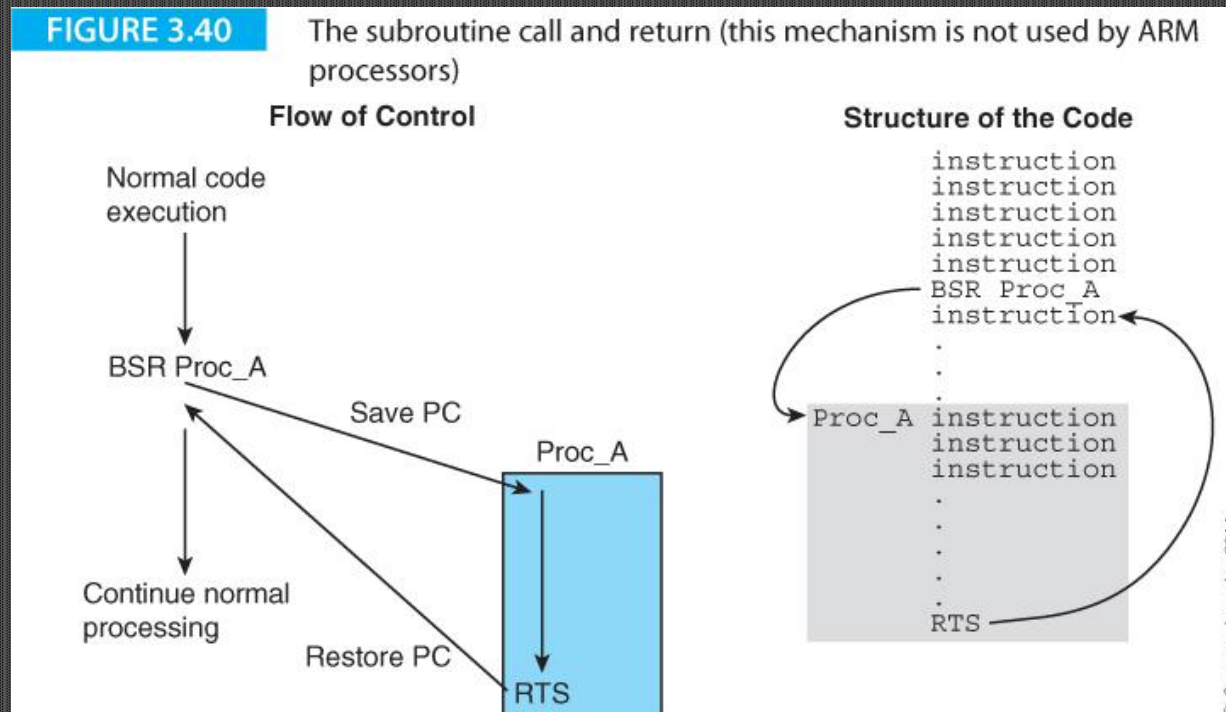
- (c) `LDR r0, [r1], #12`  
Effective address specified by base register. Operand accessed at effective address. Offset added to base register after the access.

## Subroutine Call and Return

The instruction `BSR Proc_A` *calls* subroutine `Proc_A`.

The processor saves the address of the next instruction to be executed in a safe place, and loads the program counter with the address of the first instruction in the subroutine.

At the end of the subroutine a *return from subroutine instruction*, `RTS`, causes the processor to return to the point immediately following the subroutine call.



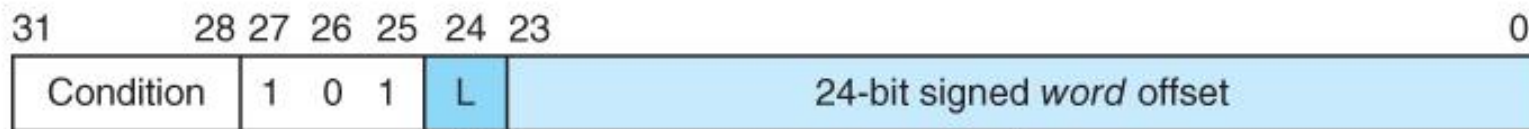
## ARM SUPPORT FOR SUBROUTINES

ARM processors do not provide a fully automatic subroutine call/return mechanism like CISC processors.

ARM's *branch with link* instruction, BL, automatically saves the return address in register r14.

The branch instruction (Figure 3.41) has an 8-bit op-code with a 24-bit signed program counter relative offset. The 24-bit offset is shifted left twice to convert the word-offset address to a byte address, sign-extended to 32 bits, added to the program counter.

**FIGURE 3.41** Encoding ARM's branch and branch-with-link instructions



The L-bit is 0 for a branch instruction and 1 for a branch with link instruction.

The 24-bit word offset is shifted left twice to create a 26-bit byte offset.

© Cengage Learning 2014

## ARM SUPPORT FOR SUBROUTINES

The branch with link instruction behaves like the corresponding branch instruction but also copies the return address (i.e., address of the next instruction to be executed following a return) into the link register r14.

If you execute:

```
BL    Sub_A    ;branch to "Sub_A" with link  
                ;save return address in r14
```

the ARM executes a branch to the target address specified by the label Sub\_A.

It also copies the program counter held in register r15 into the link register r14 to preserve the return address.

At the end of the subroutine you return by transferring the return address in r14 to the program counter by:

```
MOV    pc,lr    ;we can also write this MOV r15,r14
```

Suppose that you want to evaluate **if  $x > 0$  then  $x = 16x + 1$  else  $x = 32x$**  several times in a program. Assuming that  $x$  is in  $r0$ , we can write :

```
Func1 CMP      r0,#0          ;test for  $x > 0$ 
      MOVGT    r0,r0, LSL #4  ;if  $x > 0$   $x = 16x$ 
      ADDGT    r0,r0,#1       ;if  $x > 0$  then  $x = 16x + 1$ 
      MOVLT    r0,r0, LSL #5  ;ELSE if  $x < 0$  THEN  $x = 32x$ 
      MOV      pc,lr          :return by restoring saved PC
```

We've made use of conditional execution here. Consider the following application of the subroutine.

```
LDR    r0,[r4]          ; get P
BL     Func1             ;  $P = (\text{if } P > 0 \text{ then } 16P + 1 \text{ else } 32P)$  First call
STR    r0,[r4]          ; save P

LDR    r0,[r5,#20]      ; get Q
BL     Func1             ;  $Q = (\text{if } Q > 0 \text{ then } 16Q + 1 \text{ else } 32Q)$  Second call
STR    r0,[r5,#20]      ; save P
```



We used dummy data for the calls; first with  $P = 3$  and then with  $Q = -1$  ( $\text{FFFFFFFF}_{16}$ ). At the end of execution memory locations  $P$  and  $Q$  contain the expected values of 49 ( $31_{16}$ ) and -32 ( $\text{FFFFFFE0}_{16}$ ). These two values are stored after the data at addresses 0x4C and 0x50, respectively. We used indexed addressing with displacement to store the results in memory e.g., `STR r4,[r0,#8]`.

FIGURE 3.42 Demonstrating a subroutine call

The screenshot shows the ARM assembler interface with the following components:

- Assembly Code (BL\_instruction.asm):**

```

01 AREA BL_instruction, CODE, READWRITE
02 ENTRY
03
04 ADR r4,P ;register r4 points at P
05 ADR r5,Q ;register r5 points at Q
06
07 LDR r0,[r4] ; get P
08 BL Func1 ; P = (if P > 0 then 16P + 1 else 32P)
09 STR r0,[r4,#8] ; save P
10 ;
11 ; some code
12 ;
13 LDR r0,[r5] ; get Q
14 BL Func1 ; Q = (if Q > 0 then 16Q + 1 else 32Q)
15 STR r0,[r5,#8] ; save P
16
17 MOV r0,#0x18 ; angel_SWReason_ReportException
18 LDR r1,=0x20026 ; ADP_Stopped_ApplicationExit
19 SVC #0x123456 ; ARM semihosting (formerly SWI)
20
21
22 Func1 CMP r0,#0 ;test for x > 0
23 MOVGT r0,r0,LSL #4 ;if x > 0 x = 16x
24 ADDGT r0,r0,#1 ;if x > 0 then x = 16x + 1
25 MOVLT r0,r0,LSL #5 ;ELSE if x < 0 THEN x = 32x
26 MOV pc,r14 ;return by restoring saved PC
27
28 AREA BL_instruction, DATA, READWRITE
29 DCD 0x00000003 ;P = 3
30 DCD 0xFFFFFFFF ;Q = -1
31 SPACE 8
32

```
- Registers Window:**

Register	Value
R0	0x00000018
R1	0x0020026
R2	0x00000000
R3	0x00000000
R4	0x00000044
R5	0x00000048
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x0000001C
R15 (PC)	0x00000028
CPSR	0x40000003
SPSR	0x00000000
- Memory 1 Window:**

Address	Value
0x00000044	00 00 00 03 FF FF FF FF
0x0000004C	00 00 00 31 FF FF FF E0
0x00000054	00 00 00 00 00 00 00 00
0x0000005C	00 00 00 00 00 00 00 00

Arrows indicate that the 'Source data' (0x31) is at address 0x4C and the 'Result' (0xE0) is at address 0x5C.

```

AREA BL_instruction, CODE, READWRITE
ENTRY

ADR    r4,P           ;register r4 points at P
ADR    r5,Q           ;register r5 points at Q

LDR    r0,[r4]        ; get P
BL     Func1          ; P = (if P > 0 then 16P + 1 else 32P)
STR    r0,[r4,#8]     ; save P
;
; some code
;
LDR    r0,[r5]        ; get Q
BL     Func1          ; Q = (if Q > 0 then 16Q + 1 else 32Q)
STR    r0,[r5,#8]     ; save P

MOV    r0, #0x18      ; angel_SWIreason_ReportException
LDR    r1, =0x20026    ; ADP_Stopped_ApplicationExit
SVC    #0x123456      ; ARM semihosting (formerly SWI)

```

```

Func1  CMP    r0,#0    ;test for x > 0
MOVGT  r0,r0, LSL #4   ;if x > 0 x = 16x
ADDGT  r0,r0,#1        ;if x > 0 then x = 16x + 1
MOVLT  r0,r0, LSL #5   ;ELSE if x < 0 THEN x = 32x
MOV    pc,r14         ;return by restoring saved PC

```

```

P
Q
AREA BL_instruction, DATA, READWRITE
DCD    0x00000003      ;P = 3
DCD    0xFFFFFFFF      ;Q = -1
SPACE  8

```

Memory 1	
Address:	0x44
0x00000044:	00 00 00 03 FF FF FF FF
0x0000004C:	00 00 00 31 FF FF FF E0
0x00000054:	00 00 00 00 00 00 00 00

Registers	
Register	Value
<b>Current</b>	
R0	0x00000018
<b>R1</b>	<b>0x00020026</b>
R2	0x00000000
R3	0x00000000
R4	0x00000044
R5	0x00000048
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x0000001C
<b>R15 (PC)</b>	<b>0x00000028</b>
+ CPSR	0xA00000D3
+ SPSR	0x00000000
+ User/System	
+ Fast Interrupt	
+ Interrupt	
+ <b>Supervisor</b>	
+ Abort	
+ Undefined	
+ Internal	
PC \$	0x00000028
Mode	Supervisor
States	36
Sec	0.00000000

# THE STACK

The stack is a data structure, a *last in first out queue*, LIFO, in which items enter at one end and leave in the reverse order.

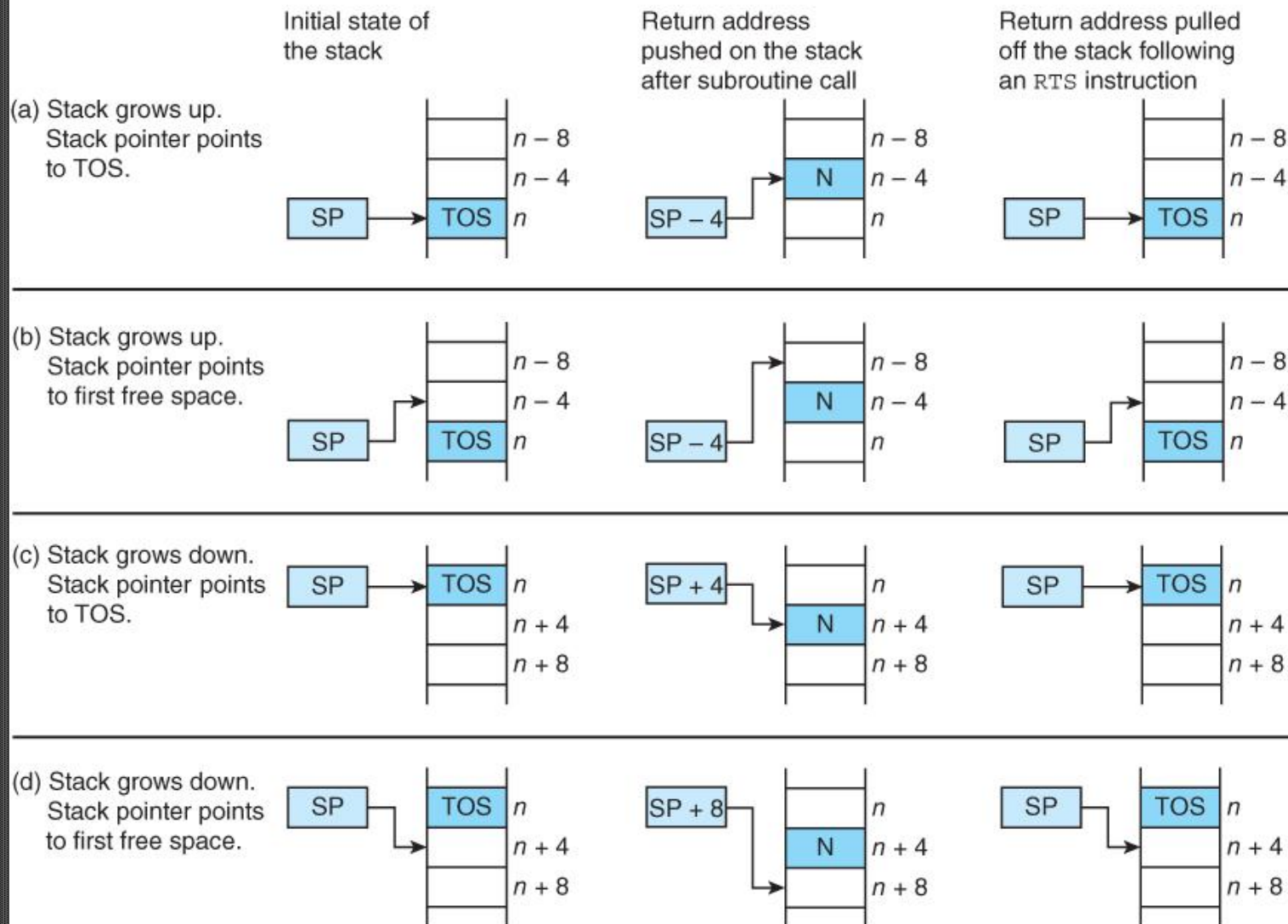
Stacks in microprocessors are implemented by using a *stack pointer* to point to the top of the stack in memory.

As items are added to the stack (pushed), the stack pointer is moved up, and as items are removed from the stack (pulled or popped) the stack pointer is moved down.

Figure 3.45 demonstrates four ways of constructing a stack. The two design decisions you have to make when implementing a stack are whether the stack grows *up toward low* memory as items are pushed or whether the stack grows *down toward high* memory as items are pushed.

TOS means *top of stack* and indicates the next item on the stack.

Figure 3.45 shows the stack being used to store a return address after a subroutine call.

**FIGURE 3.45****Possible stack structures**

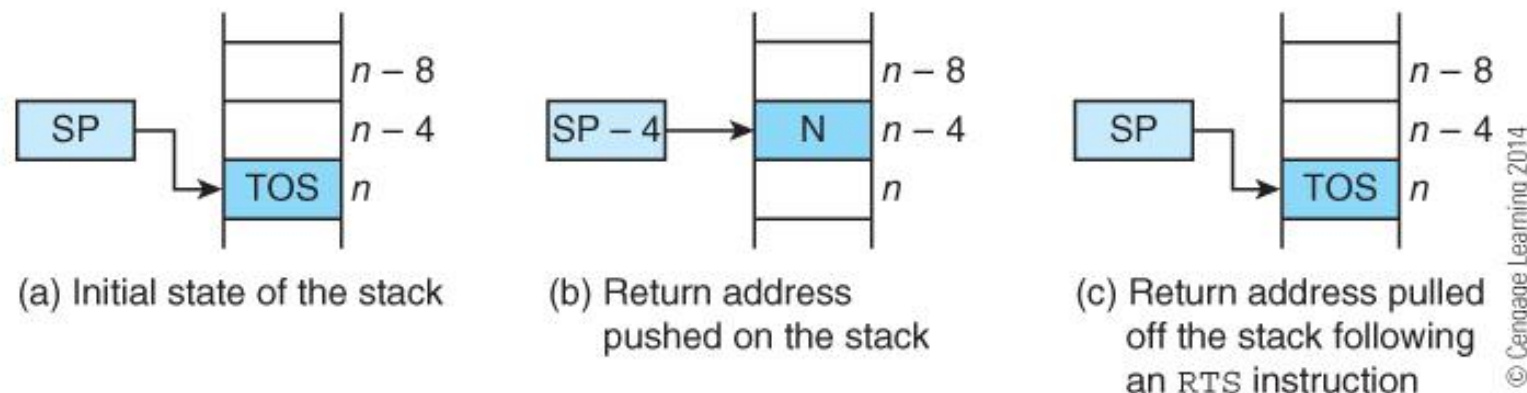
TOS = top of stack (the element at the top of the stack). By convention, up refers to the top of the page and lower addresses. This can be confusing. These diagrams assume byte-addressed memory and that each element on the stack is 32 bits (4 bytes).



An important application of the stack is to save return addresses after a subroutine call.

CISC processors maintain the stack automatically. RISC processors force the programmer to maintain the stack.

**FIGURE 3.46** Using the stack to save a return address





## SUBROUTINE CALL AND RETURN

A subroutine call can be implemented by pushing the return address on the stack and then jumping to the branch target address. Typically, this operation is implemented by JSR target or BSR target by CISC processors.

Because the ARM does not implement this operation, you could synthesize this instruction by:

```
;assume that the stack grows towards low addresses and  
;the SP points ;at the next item on the stack.  
SUB r13,r13,#4 ;pre-decrement the stack pointer  
STR r15,[r13]; ;push the return address on the stack  
B Target ;jump to the target address  
... ;return here
```

Once the code or body of the subroutine has been executed, a *return from subroutine* instruction, RTS, is executed and the program counter restored to the point it was at after the BSR Proc\_A instruction had been fetched. The effect of RTS instruction is

```
RTS: [PC] ← [[SP]]    ;Copy the return address on the stack to the PC  
     [SP] ← [SP] + 4   ;Adjust the stack pointer
```

In Figure 3.46 the stack moves up by 4 because each address occupies four bytes. Because the ARM does not support a stack-based subroutine return mechanism, you would have to write:

```
LDR  r12,[r13],#+4    ; get saved PC and post-increment stack pointer  
SUB  r15,[r12],#4     ;fix PC and load into r15 to return
```

# ARM subroutine call and return

**Register List:**

Register	Value
R0	0x00000018
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000010
R13 (SP)	0x00000074
R14 (LR)	0x00000000
<b>R15 (PC)</b>	<b>0x00000018</b>
CPSR	0x00000000
SPSR	0x00000000

**Assembly Code (Disassembled):**

```

4:      SUB    r13,r13,#4      ;pre-decrement the stack pointer
0x00000004 E24DD004 SUB      R13,R13,#0x00000004
5:      STR    r15,[r13];      ;push the return address on the stack
0x00000008 E58DF000 STR      PC,[R13]
6:      B      Target          ;jump to the target address
0x0000000C EA000003 B        0x00000020
7:      MOV    r0,#0xFF        ;return here (this is a dummy operation for tracing)
0x00000010 E3A000FF MOV      R0,#0x000000FF
8:      MOV    r0,#0x18        ;angel_SWIreason_ReportException
0x00000014 E3A00018 MOV      R0,#0x00000018
9:      LDR    r1,=0x20026      ;ADP_Stopped_ApplicationExit
0x00000018 E59F1010 LDR      R1,[PC,#0x0010]
10:     SVC    #0x123456        ;ARM semihosting (formerly SWI)
0x0000001C EF123456 SWI      0x00123456
11: Target MOV    r1,#0xAA      ;just a dummy operation
0x00000020 E3A010AA MOV      R1,#0x000000AA
12:     LDR    r12,[r13],#4     ;get the PC and post-increment the stack pointer
0x00000024 E49DC004 LDR      R12,[R13],#0x0004
13:     SUB    r12,r12,#4        ;modify the restored PC (because it is 4 too big)
0x00000028 E24CC004 SUB      R12,R12,#0x00000004
14:     MOV    r15,r12          ;put the return address in the PC to return
0x0000002C E1A0F00C MOV      PC,R12
  
```

**Memory 1 Window:**

Address	Value
0x00000048	00 00 00 00
0x0000004C	00 00 00 00
0x00000050	00 00 00 00
0x00000054	00 00 00 00
0x00000058	00 00 00 00
0x0000005C	00 00 00 00
0x00000060	00 00 00 00
0x00000064	00 00 00 00
0x00000068	00 00 00 00
0x0000006C	00 00 00 00
0x00000070	00 00 00 14
0x00000074	00 00 00 00
0x00000078	00 00 00 00

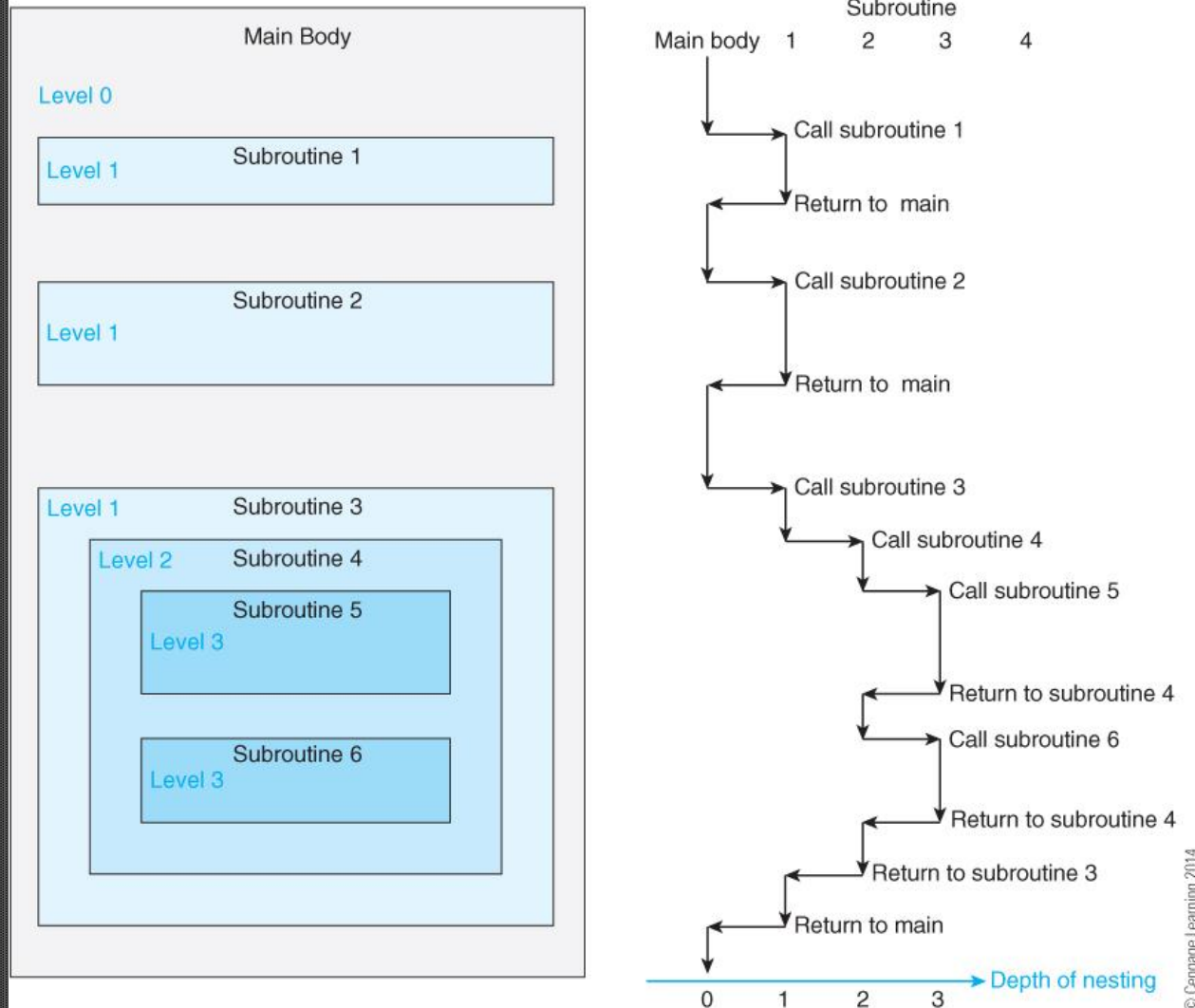
**SubroutineCall.asm Source Code:**

```

01 AREA SubroutineCall, CODE, READWRITE
02 ENTRY
03 ADR    r13,Stack            ;set up the stack pointer
04 SUB    r13,r13,#4           ;pre-decrement the stack pointer
05 STR    r15,[r13];          ;push the return address on the stack
06 B      Target              ;jump to the target address
07 MOV    r0,#0xFF            ;return here (this is a dummy operation for tracing)
08 MOV    r0,#0x18            ;angel_SWIreason_ReportException
09 LDR    r1,=0x20026          ;ADP_Stopped_ApplicationExit
10 SVC    #0x123456           ;ARM semihosting (formerly SWI)
11 Target MOV    r1,#0xAA      ;just a dummy operation
12 LDR    r12,[r13],#4         ;get the PC and post-increment the stack pointer
13 SUB    r12,r12,#4          ;modify the restored PC (because it is 4 too big)
14 MOV    r15,r12            ;put the return address in the PC to return
15
16 AREA SubroutineCall, DATA, READWRITE
17 SPACE 64                  ;reserve room for the stack to grow up
18 Stack DCD    0x00000000    ;base of the stack
  
```

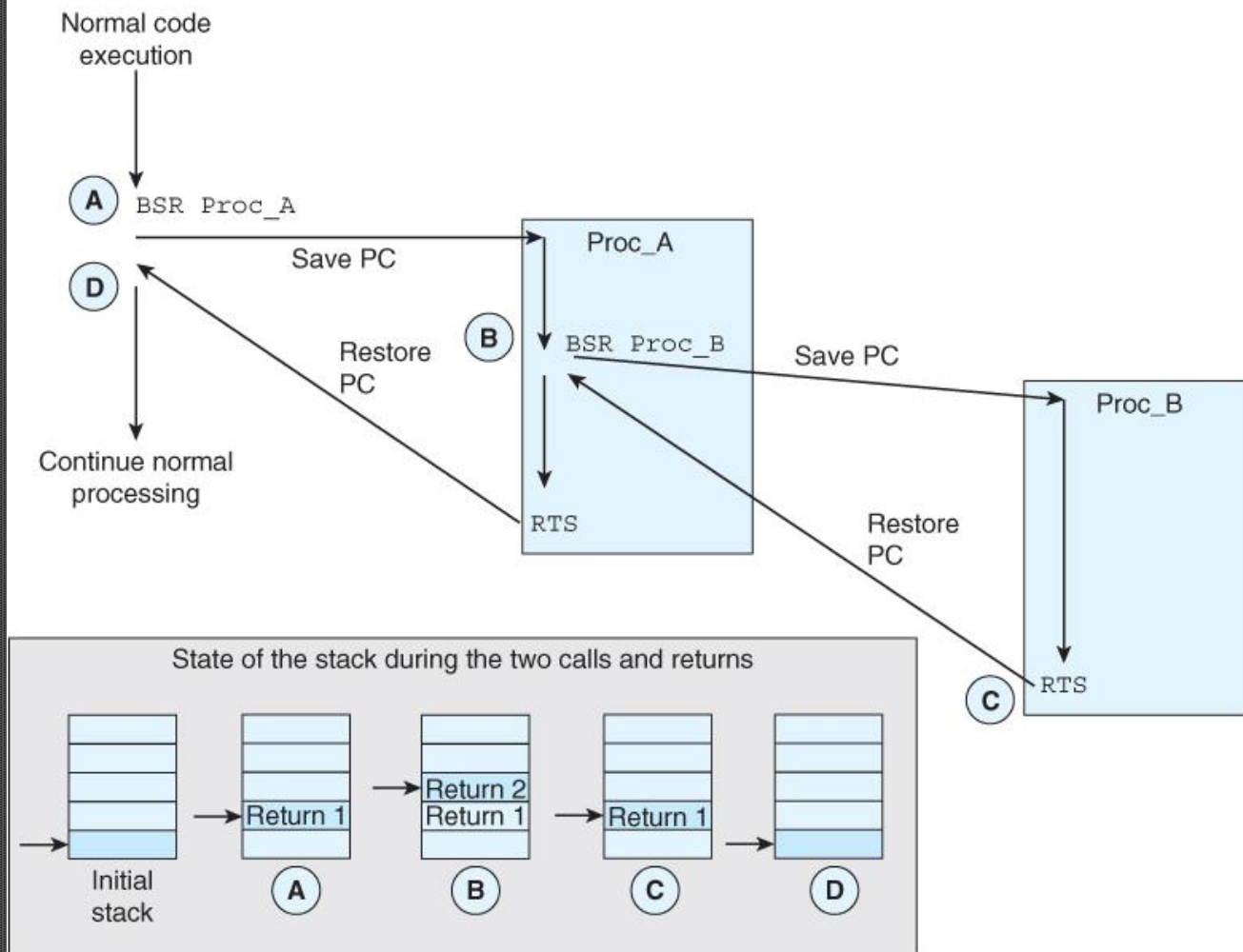
# Nested subroutines

**FIGURE 3.48** An example of nested subroutines



# Example of nested subroutine

**FIGURE 3.49** The stack and nested subroutines (CISC processors)





## LEAF ROUTINES

A leaf routine doesn't call another routine; it's at the end of the tree. If you call a leaf routine with BL, the return address is saved in link register r14. A return to the calling point is made with a MOV pc,lr.

If the routine is not a leaf routine, you cannot call another routine without first saving the link register.

```

        BL    XYZ                ;call a simple leaf routine
        .
        BL    XYZ1               ;call a routine that calls a nested routine
        .
XYZ      ...                    ;code (this is the leaf routine)
        .
        MOV   pc,lr              ;copy link register into PC and return
XYZ1     STMFD sp!,{r0-r4,lr}    ;save working registers and link register
        .
        BL    XYZ                ;call XZY – overwrites the old link register
        .
        LDMFD sp!,{r0-r4,pc}    ;restore registers and force a return

```

Subroutine XYZ is a leaf subroutine that does not call a nested subroutine and, therefore, we don't have to worry about the link register, r14, and we can return by executing **MOV pc,lr**.

Subroutine XYZ1 contains a call to a nested subroutine and we have to save the link register in order to return from XYZ1.

The simplest way of saving the link register is to push it on the stack. In this case we use a *store multiple registers* instruction and also save registers r0 to r4.

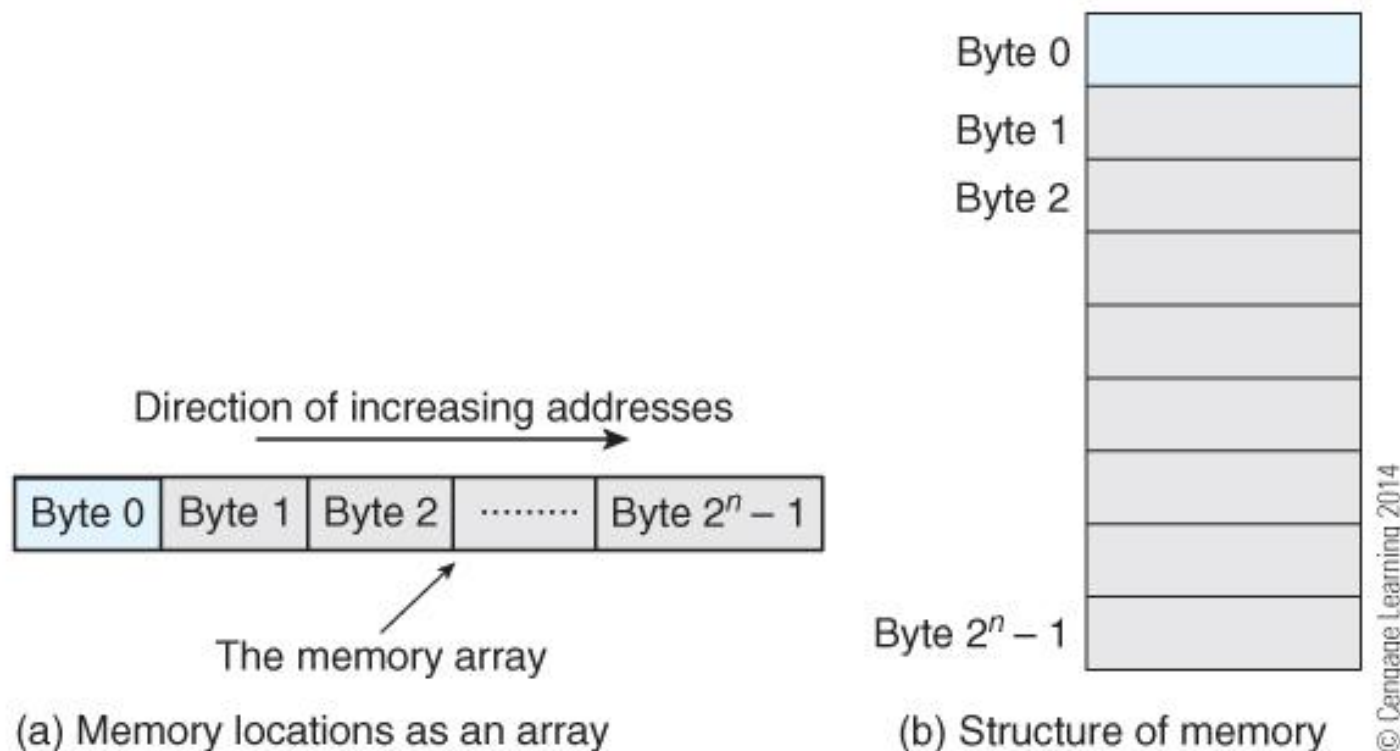
When return from XYZ1, we restore the registers and load the saved r14 (the return address in the link register) into the program counter.

# DATA ORGANIZATION AND ENDIANISM

Figure 3.50 shows how bytes in memory are numbered from 0 to  $2^n - 1$ . Word numbering is universal and the first word in memory *word 0* and the last word,  $2^n - 1$ .

**FIGURE 3.50**

Numbering the bytes of a memory array



*Bit* numbering can vary between processors. Figure 3.51a shows right-to-left numbering, with the least-significant digit on the right.

Microprocessors (ARM, Intel) number the bits of a word from the least-significant bit (lsb) which is bit 0, to the most-significant bit (e.g., msb) which is bit  $m - 1$ , in the same way.

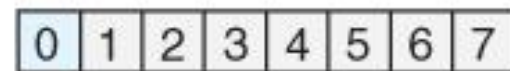
Some microprocessors, (PowerPC) reverse this scheme, as illustrated in Figure 3.51b.

**FIGURE 3.51**

Numbering the bits of a byte



(a) Bit numbering with the least-significant bit at the right



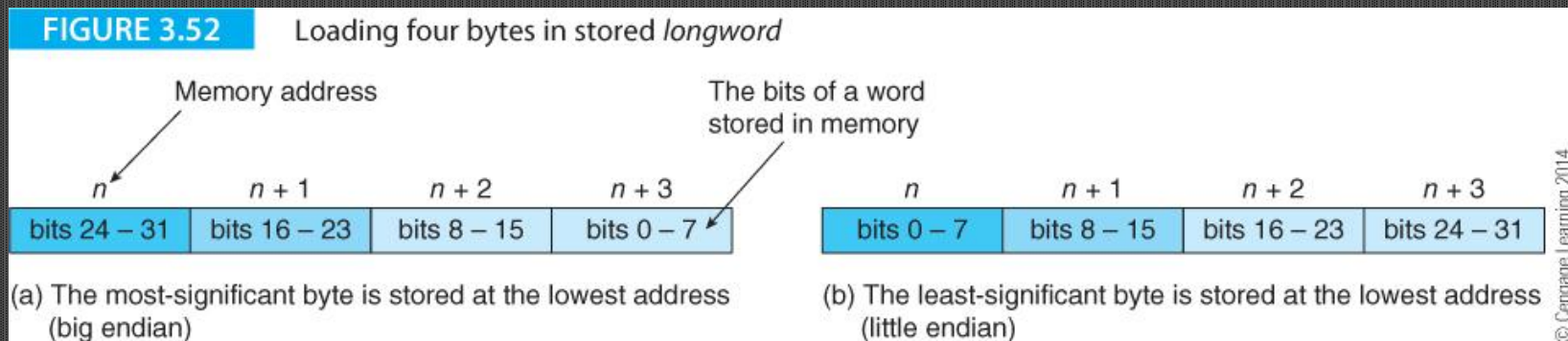
(b) Bit numbering with the least-significant bit at the left

© Cengage Learning 2014

As well as the way in which we organize the bits of a byte, we have to consider the way in which we organize the individual bytes of a word.

Figure 3.52 demonstrates that we can number the bytes of a word in two ways. We can either put the most-significant byte at the *highest byte address* of the word or we can put the most-significant byte at the *lowest address* in a word.

The ordering is called *big endian* if the most-significant element goes in at the lowest address, and *little endian* if it goes in at the highest address.





## BLOCK MOVE INSTRUCTIONS

The following conventional ARM code demonstrates how to load four registers from memory.

```
ADR r0,DataToGo ; load r0 with the address of the data area
LDR r1,[r0],#4   ; load r1 with the word pointed at by r0
                  ; and update pointer
LDR r2,[r0],#4   ; load r2 with word pointed at by r0
                  ; and update the pointer
LDR r3,[r0],#4   ; and so forth for remaining registers r3 and r5...

LDR r5,[r0],#4
```

ARM has a *block move to memory* instruction, STM, and a *block move from memory*, LDM that can copy groups of registers to and from memory. Both these block move instructions take a suffix to describe *how* the data is accessed.

Conceptually, a block move is easy to understand, because it's simply a *'copy the contents of these registers to memory'* or vice versa.

Let's start by moving the contents of registers r1, r2, r3, and r5, into sequential memory locations with

STMIA r0!,{r1-r3, r5} ;note the syntax of this and all block

This instruction copies registers r1 to r3, and r5, into sequential memory locations, using r0 as a pointer with auto-indexing (indicated by the ! suffix).

The suffix IA indicates that index register r0 is *incremented after* each transfer, with data transfer in order of increasing addresses.

Although ARM's block mode instructions have several variations, *ARM always stores the lowest numbered register at the lowest address*, followed by the next lowest numbered register at the next higher address, and so on (e.g., r1 then r2, r3, and r5 in the preceding example).

# Executing STMIA r0!,{r1-r3, r5}

The screenshot shows the uVision4 IDE with the following components:

- Registers Window:**

Register	Value
R0	0x00000048
R1	0x11111111
R2	0x22222222
R3	0x33333333
R4	0x00000000
R5	0x55555555
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000018
CPSR	0x000000D3
SPSR	0x00000000
- Assembly Window (MoveMultiple.asm):**

```

01 AREA MoveMultiple, CODE, READWRITE
02 ENTRY
03 LDR r1,=0x11111111 ;Let's set up some nice simple values that we can track
04 LDR r2,=0x22222222
05 LDR r3,=0x33333333
06 LDR r5,=0x55555555
07 ADR r0, Stack ;let's have a stack to put data on
08 STMIA r0!,{r1-r3, r5} ;Now move the data
09 MOV r0, #0x18 ;angel_SWIreason_ReportException
10 LDR r1, =0x20026 ;ADP_Stopped_ApplicationExit
11 SVC #0x123456 ;ARM semihosting (formerly SWI)
12 SPACE 20 ;leave 20 spaces
13 Stack SPACE 20 ;leave 20 more spaces
14 END

```
- Memory Window (Memory 1):**

Address	Value
0x0000001C	E5 9F 10 38
0x00000020	EF 12 34 56
0x00000024	00 00 00 00
0x00000028	00 00 00 00
0x0000002C	00 00 00 00
0x00000030	00 00 00 00
0x00000034	00 00 00 00
0x00000038	11 11 11 11
0x0000003C	22 22 22 22
0x00000040	33 33 33 33
0x00000044	55 55 55 55
0x00000048	00 00 00 00
0x0000004C	11 11 11 11
0x00000050	22 22 22 22
0x00000054	33 33 33 33
0x00000058	55 55 55 55

**Callouts:**

- Registers preloaded with data:** Points to the initial values of R1, R2, R3, and R5 in the Registers window.
- r0 contains 0x48 which is the value after the data has been stored:** Points to the value of R0 in the Registers window.
- Registers saved on the stack:** Points to the memory locations 0x00000038-0x00000058 in the Memory window, which contain the values of R1, R2, R3, and R5.
- Registers loaded in the constant pool before execution:** Points to the memory locations 0x0000004C-0x00000058 in the Memory window, which contain the values of R1, R2, R3, and R5.

# BLOCK MOVES AND STACK OPERATIONS

ARM's block move instruction is versatile because it supports four possible stack modes. The differences among these modes are the *direction* in which the stack grows (up or *ascending* and down or *descending*) and whether the stack pointer points at the item currently at the top of the stack or the next free item on the stack. CISC processors with hardware stack support generally provide only one fixed stack mode. The ARM's literature uses four terms to describe stacks:

1. DF *descending full* Figure 3.52a
2. AF *ascending full* Figure 3.52b
3. DE *descending empty* Figure 3.52c
4. AE *ascending empty* Figure 3.52d

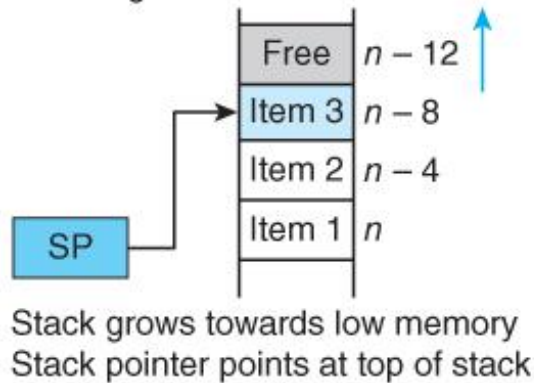
ARM uses the terms ascending and descending to describe the growth of the stack toward higher or lowers addresses, respectively and NOT whether it grows up or down on the page.

A stack is described as *full* if the stack pointer points to the top element of the stack. If the stack pointer points to the next free element above the top of the stack, then the stack is called *empty*.

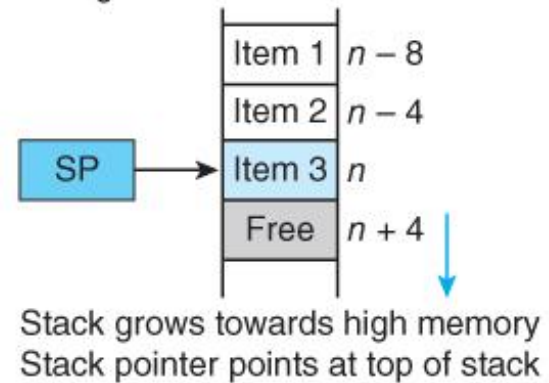


**FIGURE 3.59** ARM's four stack modes

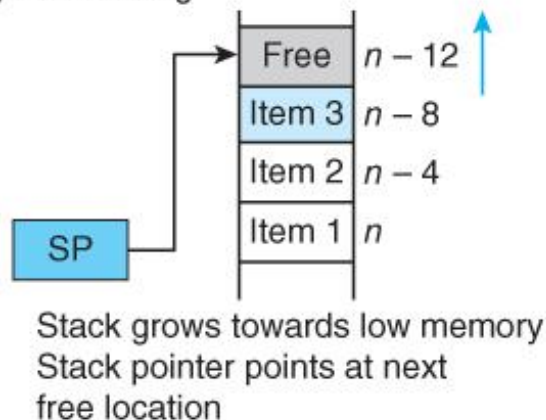
(a) Stack full descending



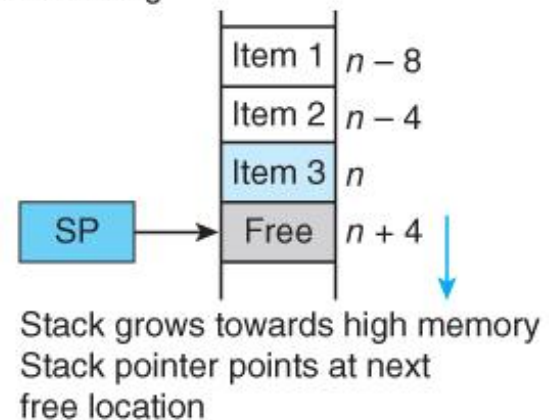
(b) Stack full ascending



(c) Stack empty descending



(d) Stack empty ascending





ARM has *two* ways of describing stacks, which can be a little confusing at first. A stack operation can be described either by *what* it does or *how* it does it.

The most popular stack points at the top item on the stack and which grows towards lower addresses.

This is a *full descending stack*, FD (the type used in this text).

We can write **STMFD sp!,{r0,r1}** when pushing r0 and r1 on the stack, and we can write **LDMFD sp!,{r0,r1}** when popping r0 and r1 off the stack.

A full descending stack is *implemented* by first decrementing the pointer and then storing data at that address (push data) or by reading data at the stack address and then incrementing the pointer (pull data).

## APPLICATIONS OF BLOCK MOVE INSTRUCTIONS

One of the most important applications of the ARM's block move instructions is in saving registers on entering a subroutine and restoring registers before returning from a subroutine. Consider the following ARM code:

```
BL      test                ;call test, save return address in r14
.  
test STMFD r13!,{r0-r4,r10} ;subroutine test, save working registers  
.  
      body of code  
.  
LDMFD r13!,{r0-r4,r10} ;subroutine completes, restore the registers  
MOV    pc,r14             ;copy the return address in r14 to the PC
```

We can reduce the size of this code because the instruction **MOV pc,r14** is redundant.

If you are using a block move to restore registers from the stack, you can also include the program counter. We can write:

```
test STMFD r13!,{r0-r4,r10,r14}  ;save working registers
                                   ; and return address in r14
:
LDMFD r13!,{r0-r4,r10,r15} ;restore working registers
                           ;and put r14 in the PC
```

At the beginning of the subroutine we push the link register r14 containing the return address onto the stack, and then at the end we pull the saved registers, including the value of the return address which is placed in the PC, to effect the return.

The block move provides a convenient means of copying data between memory regions.

In the next example we copy 256 words from Table 1 to Table 2.

The block move instruction allows us to move eight registers at once, as the following code illustrates:

	ADR	r0,Table1	; r0 points to source (note pseudo-op ADR)
	ADR	r1,Table2	; r1 points to the destination
	MOV	r2,#32	; 32 blocks of 8 = 256 words to move
Loop	LDRFD	r0!,{r3-r10}	; REPEAT Load 8 registers in r3 to r10
	STRFD	r1!,{r3-r10}	; store the registers at their destination
	SUBS	r2,r2,#1	; decrement loop counter
	BNE	Loop	; UNTIL all 32 blocks of 8 registers moved