CHAPTER

3

ARM, MOTOROLA, AND INTEL INSTRUCTION SETS

CHAPTER OBJECTIVES

In this chapter, which has three independent parts, you will learn about the following instruction set architectures:

- ARM (Part I)
- Motorola 68000 (Part II)
- Intel IA-32 Pentium (Part III)

The basic ideas of instruction sets, addressing modes, and instruction execution were introduced in Chapter 2. Assembly language representation for machine instructions and programs was used to present a number of program examples. In this chapter, we study how these basic ideas have been implemented in ARM, Motorola 68000, and Intel IA-32 ISAs. The ARM instruction set exemplifies RISC design, and the 68000 and IA-32 instruction sets illustrate the CISC design style. The three parts of this chapter, one for each instruction set, are independent complete units. The generic programs presented in Chapter 2 are coded in each of the three instruction sets. It is important to have a good understanding of the full discussion of basic ideas and programs in Chapter 2 because the corresponding discussions here are more brief. Appendices B, C, and D give concise summaries of the three ISAs, and contain more detail than is provided here in Chapter 3.

PART I
THE ARM EXAMPLE

Advanced RISC Machines (ARM) Limited has designed a family of microprocessors, and it licenses the designs to other companies for chip fabrication and use in computer products and embedded systems. The ARM company is relatively new, having evolved out of the Acorn Computers company that developed processor designs in the early 1980s. The main use for ARM microprocessors is in low-power and low-cost embedded applications such as mobile telephones, communication modems, automotive engine management systems, and hand-held digital assistants [1]. The book by Furber [2] contains a wealth of information on ARM design and implementation; the Clements text [3] uses ARM as a major example, and the book by van Someren and Atack [4] describes assembly language programming for ARM. Detailed information is also available at the ARM web site [5]. All ARM processors share the same basic machine instruction set. The version used here is the one implemented by the ARM7 processor. Later versions added features that are not relevant for the level of discussion in this chapter. In Chapter 11, we describe some of the added features in these later versions of the architecture. The programs from Chapter 2 are presented here in the ARM assembly language in order to illustrate various aspects of the ARM architecture.

3.1 REGISTERS, MEMORY ACCESS, AND DATA TRANSFER

In the ARM architecture, memory is byte addressable, using 32-bit addresses, and the processor registers are 32 bits long. Two operand lengths are used in moving data between the memory and the processor registers: bytes (8 bits) and words (32 bits). Word addresses must be aligned, that is, they must be a multiple of 4. Both little-endian and big-endian memory addressing is supported. (See Section 2.2.2.) The choice is determined by an external input control line to the processor. When a byte is loaded from memory into a processor register or stored from a register into the memory, it is always located in the low-order byte position of the register.

Memory is accessed only by Load and Store instructions. All arithmetic and logic instructions operate only on data in processor registers. This arrangement is a basic

feature of RISC architectures. Its implications for simplicity of processor design and performance will be examined in Chapter 8.

3.1.1 REGISTER STRUCTURE

The processor registers used by application programs are shown in Figure 3.1. There are sixteen 32-bit registers labeled R0 through R15, which consist of fifteen general purpose registers (R0 through R14) and the Program Counter (PC) register, R15. The general purpose registers can hold either memory addresses or data operands. The Current Program Status Register (CPSR), or simply the Status register, holds the condition code flags (N, Z, C, V), interrupt disable flags, and processor mode bits. The information represented by the condition code flags is described in Section 2.4.6. The use of processor mode bits and interrupt disable bits will be described in conjunction with input/output operations and interrupts in Chapter 4. Here, we will assume that the processor is in User mode and is executing an application program.

There are 15 additional general-purpose registers called the *banked* registers. They are duplicates of some of the R0 to R14 registers. They are used when the processor switches into Supervisor or Interrupt modes of operation. Saved copies of the Status register are also available in these nonUser modes. These banked registers and Status register copies will also be discussed in Chapter 4.

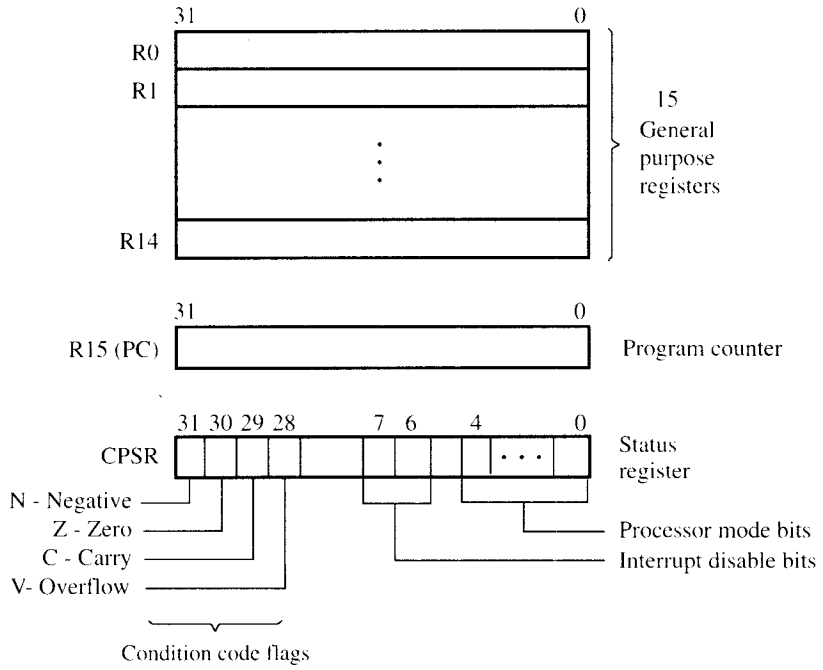


Figure 3.1 ARM register structure.

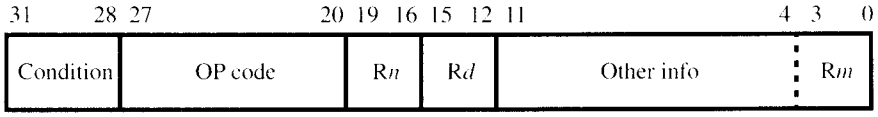


Figure 3.2 ARM instruction format.

3.1.2 MEMORY ACCESS INSTRUCTIONS AND ADDRESSING MODES

Each instruction in the ARM architecture is encoded into a 32-bit word in a reasonably uniform way, typical of RISC designs. Access to memory is provided only by Load and Store instructions. The basic encoding format for these instructions, as well as for the Move, Arithmetic, and Logic instructions is shown in Figure 3.2. More detail is given in Appendix B. An instruction specifies a conditional execution code (Condition), the OP code, two or three registers (Rn, Rd, and Rm), and some other information. If register Rm is not needed, the "Other info" field extends to bit b_0. In a Load instruction, the operand is transferred from the memory into the general-purpose register named in the 4-bit Rd field. In a Store instruction, the operand is transferred from Rd into the memory. If the operand is a byte, it is always located in the low-order byte position of the register, and in a Load instruction, the high-order 24 bits of the register are filled with zeros.

Conditional Execution of Instructions

A distinctive and somewhat unusual feature of ARM processors is that all instructions are conditionally executed, depending on a condition specified in the instruction. The instruction is executed only if the current state of the processor condition code flags satisfies the condition specified in bits b_{31-28} of the instruction. Otherwise, the processor proceeds to the next instruction. One of the conditions is used to indicate that the instruction is always executed. The usefulness of conditional execution will be seen in the examples in Section 3.7. For now, we will ignore this feature and assume that the condition field of the instruction contains the "always executed" code.

Memory Addressing Modes

The basic method for addressing memory operands is to generate the effective address, EA, of the operand by adding a signed offset to the contents of a base register Rn, which is specified in the instruction as shown in Figure 3.2. The magnitude of the offset is either an immediate value, contained in the low-order 12 bits of the instruction, or it is the contents of a third register, Rm, named by the low-order four bits, $b_{3\ 0}$. The sign (direction) of the offset is contained in the OP-code field.

For example, the Load instruction

$$\text{LDR} \quad \text{R}d,[\text{R}n,\#\text{offset}]$$

specifies the offset (expressed as a signed number) in the immediate mode and performs the operation

$$\text{R}d \leftarrow [[\text{R}n] + \text{offset}]$$

Note that the destination register, Rd, is listed first. This is opposite to the order used in Chapter 2. The instruction

$$\text{LDR} \quad \text{R}d,[\text{R}n.\text{R}m]$$

performs the operation

$$\text{R}d \leftarrow [[\text{R}n] + [\text{R}m]]$$

Since the contents of Rm are the magnitude of the offset, Rm must be preceded by a minus sign if a negative offset is desired. In Chapter 2, these two addressing modes were defined as the Index and Base with index modes, respectively. An offset of zero does not have to be specified explicitly. Hence, the instruction

$$\text{LDR} \quad \text{R}d,[\text{R}n]$$

performs the operation

$$\text{R}d \leftarrow [[\text{R}n]]$$

using the addressing mode that was defined as the Indirect mode in Chapter 2.

The OP-code mnemonic LDR specifies that a 32-bit word is loaded from the memory into a register. A byte operand can be loaded into the low-order byte position of a register by using the mnemonic LDRB. The higher order bits are filled with zeros.

Store instructions have the mnemonics STR and STRB. For example, the instruction

$$\text{STR} \quad \text{R}d,[\text{R}n]$$

performs the operation

$$[\text{R}n] \leftarrow [\text{R}d]$$

transferring a word operand into the memory. The STRB instruction transfers the byte contained in the low-order end of Rd.

ARM documents refer to all of these modes, and others that we will describe shortly, as indexed addressing modes. The form that we have used in these first examples is called the Pre-indexed addressing mode because the effective address of the operand is generated by adding the offset to the contents of the base register Rn. The contents of register Rn are not changed. Addressing modes that are similar to the Autodecrement and Autoincrement modes that were discussed in Chapter 2 are also provided. They are called Pre-indexed with writeback and Post-indexed, respectively.

Definitions of all three modes are given as:

Pre-indexed mode — The effective address of the operand is the sum of the contents of the base register Rn and an offset value.

Pre-indexed with writeback mode — The effective address of the operand is generated in the same way as in the Pre-indexed mode, and then the effective address is written back into Rn.

Post-indexed mode — The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn.

Table 3.1 specifies the assembly language syntax for these addressing modes, and

Table 3.1 ARM indexed addressing modes

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[R*n*, #offset]	EA = [R*n*] + offset
Pre-indexed with writeback	[R*n*, #offset]!	EA = [R*n*] + offset; R*n* ← [R*n*] + offset
Post-indexed	[R*n*], #offset	EA = [R*n*]; R*n* ← [R*n*] + offset
With offset magnitude in R*m*:		
Pre-indexed	[R*n*, ± R*m*, shift]	EA = [R*n*] ± [R*m*] shifted
Pre-indexed with writeback	[R*n*, ± R*m*, shift]!	EA = [R*n*] ± [R*m*] shifted; R*n* ← [R*n*] ± [R*m*] shifted
Post-indexed	[R*n*], ± R*m*, shift	EA = [R*n*]; R*n* ← [R*n*] ± [R*m*] shifted
Relative (Pre-indexed with immediate offset)	Location	EA = Location = [PC] + offset

EA = effective address
offset = a signed number contained in the instruction
shift = direction #integer
where direction is LSL for left shift or LSR for right shift, and
integer is a 5-bit unsigned number specifying the shift amount
±R*m* = the offset magnitude in register R*m* can be added to or subtracted from the contents of base register R*n*

gives expressions for the calculation of the effective address, EA, and the writeback operations. The exclamation mark signifies writeback in the Pre-indexed addressing mode. The Post-indexed mode always involves writeback, so the exclamation mark is not needed. Note that pre- and post-indexing are distinguished by the way the square brackets are used. When only the base register is enclosed in square brackets, its contents are used as the effective address. The offset is added to the register contents after the operand is accessed. In other words, post-indexing is specified. This is a generalized form of the Autoincrement addressing mode described in Section 2.5. When both the base register and the offset are placed inside the square brackets, the sum is used as the effective address of the operand, that is, pre-indexing is used. If writeback is to be performed, it must be indicated by the exclamation mark (!). Pre-indexing with writeback is a generalization of the Autodecrement addressing mode discussed in Section 2.5.

In all three addressing modes, the offset may be given as an immediate value in the range ±4095. Alternatively, the magnitude of the offset may be specified as the contents of the third register, R*m*, with the sign (direction) of the offset specified by a ± prefix on the register name. For example, the instruction

LDR R0,[R1,−R2]!

performs the operation

$$R0 \leftarrow [[R1] - [R2]]$$

The effective address of the operand, [R1] − [R2], is then loaded into R1 because writeback is specified by the exclamation mark.

When the offset is given in a register, it may be scaled by a power of 2 by shifting to the right or to the left. This is indicated in the assembly language by placing the shift direction, LSL for left shift or LSR for right shift, and the shift amount, after the register name R*m*, as shown in Table 3.1. The amount of the shift is specified by an immediate value in the range 0 to 31. For example, the contents of R2 in the example above may be multiplied by 16 before being used as an offset as follows:

```
LDR   R0,[R1,−R2,LSL #4]!
```

This instruction will perform the operation

$$R0 \leftarrow [[R1] - 16 \times [R2]]$$

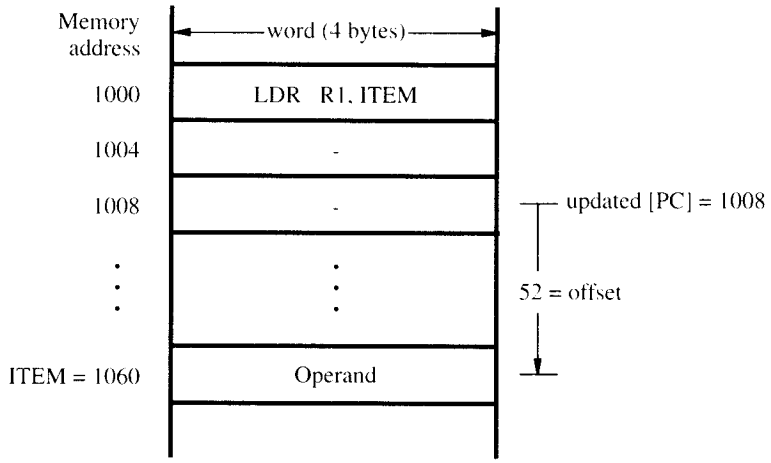
and will then load the effective address into R1.

The Program Counter, PC, may be used as the Base register R*n*. In this case, the Relative addressing mode, as described in Section 2.5, is implemented. The assembler determines the immediate offset as the signed distance between the address of the operand and the contents of the updated PC. When the effective address is calculated at instruction execution time, the contents of the PC will have been updated to the address two words (8 bytes) forward from the instruction containing the Relative addressing mode. The reason for this is related to pipelined execution of instructions, which will be discussed in Chapter 8.

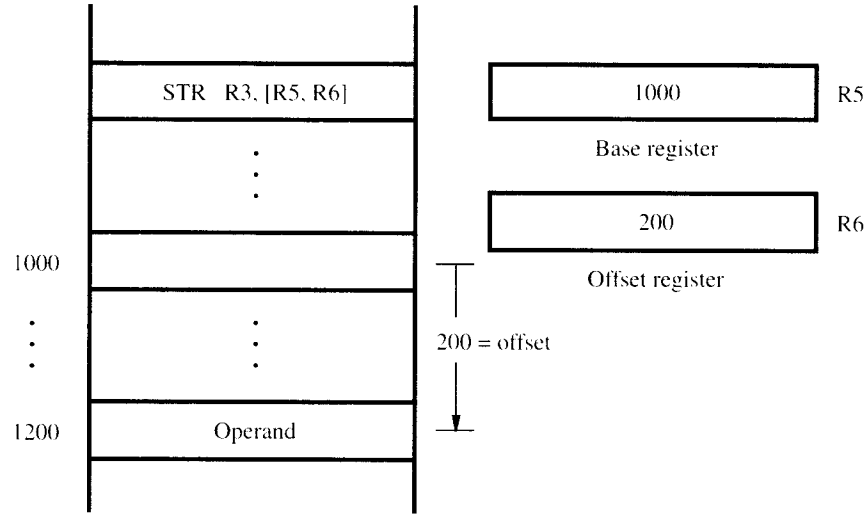
An example of the Relative mode is shown in Figure 3.3*a*. The address of the operand, given symbolically as ITEM in the instruction, is 1060. There is no Absolute addressing mode available in the ARM architecture. Therefore, when the address of an operand is given in this way in the assembly language, the assembler always uses the Relative addressing mode. This is implemented by the Pre-indexed mode with an immediate offset, using PC as the base register. As shown in the figure, the offset calculated by the assembler is 52 because the updated PC will contain 1008 when the offset is added to it during program execution, and the effective address to be generated is 1060 = 1008 + 52. The operand must be within the range of 4095 bytes forward or backward from the updated PC. If the operand address given in the instruction is outside this range, an error is indicated by the assembler and a different addressing mode must be used to access the operand.

Figure 3.3*b* shows an example of the Pre-indexed mode with the offset contained in register R6 and the base value contained in R5. The Store instruction (STR) stores the contents of R3 into memory word location 1200.

The examples shown in Figure 3.4 illustrate the usefulness of the writeback feature in the Post-indexed and Pre-indexed addressing modes. Figure 3.4*a* shows the first three numbers of a list of 25 numbers that are spaced 25 words apart, starting at memory address 1000. They comprise the first row of a 25 × 25 matrix of numbers stored in column order. The first number of the first row of the matrix is stored in word location 1000. The numbers at addresses 1100, 1200, . . . , 3400 are successive numbers of the



(a) Relative addressing mode

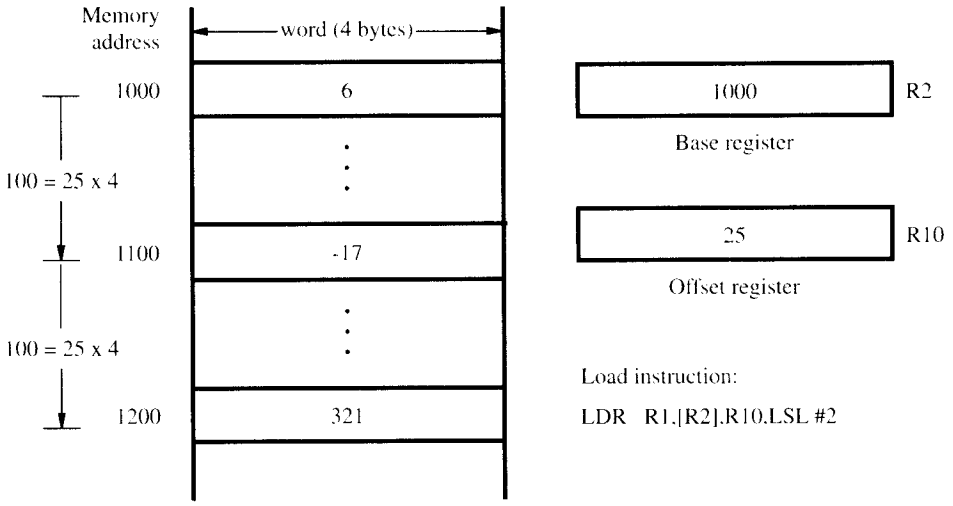


(b) Pre-indexed addressing mode

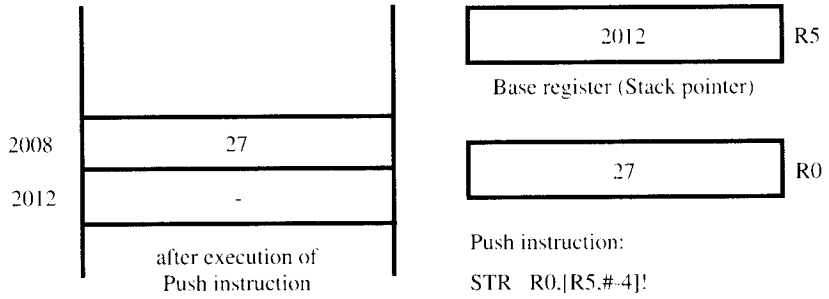
Figure 3.3 Examples of ARM memory addressing modes.

first row. The 25 memory locations 1000, 1004, 1008, . . . , 1096 contain the first column of the matrix.

Successive numbers in the first row of the matrix can be accessed conveniently using the Post-indexed addressing mode with writeback, with the offset contained in a register. Suppose that R2 is used as the base register and that it contains the initial address value 1000. Register R10 is used to hold the offset, and it is loaded with the



(a) Post-indexed addressing with writeback



(b) Pre-indexed addressing with writeback

Figure 3.4 ARM memory addressing modes involving writeback.

value 25. The instruction

LDR R1,[R2],R10,LSL #2

can then be used in a program loop that loads register R1 with successive elements of the first row of the matrix on successive passes through the loop. Let us examine how this works, step by step. The first time that the Load instruction is executed, the effective address is [R2] = 1000. Therefore, the number 6, at this address, is loaded into R1. Then, the writeback operation changes the contents of R2 from 1000 to 1100 so that it points to the second number, −17. It does this by shifting the contents, 25, of

the offset register R10 left by two bit positions and then adding them to the contents of R2. The contents of R10 are not changed in this process. The left shift is equivalent to multiplying 25 by 4, generating the required offset 100. After this offset is added to the contents of R2, the new address 1100 is written back into R2. When the Load instruction is executed on the second pass through the loop, the second number, −17, is loaded into R1. The third number, 321, is loaded into R1 on the third pass, and so on.

This example involved adding the shifted contents of the offset register to the contents of the base register. As indicated in Table 3.1, the shifted offset can also be subtracted from the contents of the base register. Any shift distance in the range 0 through 31 can be selected, and either right or left shifting can be specified.

Figure 3.4*b* shows an example of pushing the contents, 27, of register R0 onto a stack. Register R5 is used as the stack pointer. Initially, it contains the address 2012 of the current TOS (top-of-stack) element. The Pre-indexed addressing mode with writeback, using an immediate offset, can be used to perform the Push operation with the instruction

```
STR   R0,[R5,#−4]!
```

The immediate offset −4 is added to the contents, 2012, of R5 and written back into R5. This new TOS location, 2008, is used as the effective address for the Store operation. The contents, 27, of register R0 are stored at location 2008.

Load/Store Multiple Operands

In addition to the Load and Store instructions for single operands, there are two instructions for loading and storing multiple operands. They are called Block transfer instructions. Any subset of the general purpose registers can be loaded or stored. Only word operands are allowed, and the OP codes used are LDM (Load Multiple) and STM (Store Multiple). The memory operands must be in successive word locations. All of the forms of pre- and post-indexing with and without writeback are available. They operate on a Base register R*n* specified in the instruction. The offset magnitude is always 4 in these instructions so it does not have to be specified explicitly in the instruction. The list of registers must appear in increasing order in the assembly language expression for the instruction. As an example, assume that register R10 is the Base register and that it contains the value 1000 initially. Then, the instruction

```
LDMIA   R10!,{R0,R1,R6,R7}
```

transfers the words from locations 1000, 1004, 1008, and 1012 into registers R0, R1, R6, and R7, leaving the address value 1016 in R10 after the last transfer. The suffix IA in the OP code indicates "Increment After," corresponding to post-indexing. We will discuss the Load/Store Multiple instructions further in Section 3.6 in conjunction with implementing subroutines, where they are used to save and restore registers on a stack in an efficient way.

3.1.3 REGISTER MOVE INSTRUCTIONS

It is often necessary to copy the contents of one register into another register or to load an immediate value into a register. The Move instruction

MOV Rd,Rm

uses the format shown in Figure 3.2 to copy the contents of register Rm into register Rd. An immediate operand in the low-order 8 bits of the instruction can also be loaded into register Rd by the Move instruction. For example,

MOV R0,#76

places the immediate value 76 into register R0. In both forms of the Move instruction, the source operand can be shifted before being placed in the destination register.

3.2 ARITHMETIC AND LOGIC INSTRUCTIONS

The ARM instruction set has a number of instructions for arithmetic and logic operations on operands that are either contained in the general-purpose registers or given as immediate operands in the instruction itself. Memory operands are not allowed for these instructions. There are instructions for different forms of addition and subtraction, and there are two instructions for multiplication. There are instructions for the AND, OR, NOT, XOR, and Bit-Clear logic operations. Instructions such as Compare are provided to set condition code flags based on the results from arithmetic or logic operations on two operands. They do not store the actual results in a register. The format for most of these instructions is shown in Figure 3.2.

3.2.1 ARITHMETIC INSTRUCTIONS

The basic assembly language expression for arithmetic instructions is

OPcode Rd,Rn,Rm

where the operation specified by the OP code is performed using the operands in general-purpose registers Rn and Rm. The result is placed in register Rd. For example, the instruction

ADD R0,R2,R4

performs the operation

R0 ← [R2] + [R4]

and the instruction

SUB R0,R6,R5

performs the operation

$$R0 \leftarrow [R6] - [R5]$$

Instead of being contained in register Rm, the second operand can be given in the Immediate mode. Thus,

```
ADD   R0,R3,#17
```

performs the operation

$$R0 \leftarrow [R3] + 17$$

The immediate value is contained in the 8-bit field in bits b_{7-0} of the instruction.

The second operand can be shifted or rotated before being used in the operation. When a shift or rotation is required, it is specified last in the assembly language expression for the instruction. The instruction

```
ADD   R0,R1,R5,LSL #4
```

operates as follows: The second operand, which is contained in register R5, is shifted left 4 bit positions (equivalent to [R5] $\times$ 16), and it is then added to the contents of register R1; the sum is placed in register R0.

Two versions of a Multiply instruction are provided. The first version multiplies the contents of two registers and places the low-order 32-bits of the product in a third register. The high-order bits of the product, if there are any, are discarded. For example, the instruction

```
MUL   R0,R1,R2
```

performs the operation

$$R0 \leftarrow [R1] \times [R2]$$

The second version specifies a fourth register whose contents are added to the product before storing the result in the destination register. Hence, the instruction

```
MLA   R0,R1,R2,R3
```

performs the operation

$$R0 \leftarrow [R1] \times [R2] + [R3]$$

This is called a Multiply-Accumulate operation. It is often used in numerical algorithms for digital signal processing. We will see an example of this type of application in Section 3.7. The fourth register is encoded in the Other information field of Figure 3.2. There are no provisions made for shifting or rotating any of the operands before they are used in the two Multiply instructions. Some versions of the ARM ISA accommodate double-length products (64 bits). (See Chapter 11.)

Operand Shift Operations

We noted earlier that one of the distinctive features of the ARM instruction set is that all instructions are executed conditionally. Another distinctive feature is the shifting and rotation operations that are incorporated into most instructions. In most

other computer instruction sets, shifting operations are done using separate instructions. This is the case for the Motorola 68000 and the Intel IA-32 processors described in Parts II and III of this chapter. By incorporating shifting and rotation operations into instructions, as needed, the ARM architecture saves code space and can potentially improve execution time performance relative to more conventional processor designs. This feature is implemented using a *barrel shifter* circuit in the data path between the registers and the arithmetic and logic unit in the processor. Details of the shifting and rotation operations available, and their encoding in the instruction format, are given in Appendix B.

3.2.2 LOGIC INSTRUCTIONS

The logic operations AND, OR, XOR, and Bit-Clear are implemented by instructions with the OP codes AND, ORR, EOR, and BIC. They have the same format as the arithmetic instructions. The instruction

AND R*d*,R*n*,R*m*

performs the operation

R*d* ← [R*n*] ∧ [R*m*]

which is a bitwise logical AND between the operands in registers R*n* and R*m*. For example, if register R0 contains the hexadecimal pattern 02FA62CA and R1 contains the pattern 0000FFFF, then the instruction

AND R0,R0,R1

will result in the pattern 000062CA being placed in register R0.

The Bit-Clear instruction (BIC) is closely related to the AND instruction. It complements each bit in operand R*m* before ANDing them with the bits in register R*n*. Using the same R0 and R1 bit patterns as in the above example, the instruction

BIC R0,R0,R1

results in the pattern 02FA0000 being placed in R0.

The Move Negative instruction, with the OP-code mnemonic MVN, complements the bits of the source operand and places the result in R*d*. If the contents of R3 are the hexadecimal pattern 0F0F0F0F, then the instruction

MVN R0,R3

places the result F0F0F0F0 in register R0.

```
LDR     R0,POINTER      Load address LOC into R0.
LDRB    R1,[R0]         Load ASCII characters
LDRB    R2,[R0,#1]        into R1 and R2.
AND     R2,R2,#&F       Clear high-order 28 bits of R2.
ORR     R2,R2,R1,LSL #4 Or [R1] shifted left into [R2].
STRB    R2,PACKED       Store packed BCD digits
                          into PACKED.
```

Figure 3.5 An ARM program for packing two 4-bit decimal digits into a byte.

Digit-Packing Program

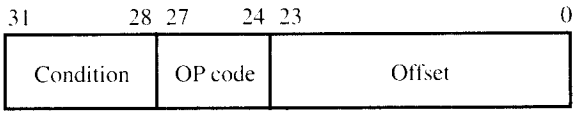
Figure 3.5 shows an ARM program for packing two 4-bit decimal digits into a memory byte location. The generic version of this program is shown in Figure 2.31 and is described in Section 2.10.2. The decimal digits, represented in ASCII code, are stored in byte locations LOC and LOC + 1. The program packs the corresponding 4-bit BCD codes into a single byte location PACKED.

The first Load instruction in the program in Figure 3.5 assumes that the address LOC is stored in memory at address POINTER. As we will see in Section 3.4, an assembler directive can be used to place LOC in POINTER. This method of loading the address LOC into R0 is needed because a 32-bit address cannot be included as an immediate operand in an instruction. Location POINTER points to the BCD digit characters stored in successive byte locations. The two ASCII characters containing the BCD digits in their low-order four bits are loaded into the low-order byte positions of registers R1 and R2 by the next two Load instructions. The And instruction clears the high-order 28 bits of R2 to zero, leaving the second BCD digit in the four low-order bit positions. The Or instruction then shifts the first BCD digit in R1 to the left four positions and places it to the left of the second BCD digit in R2. The packed digits in the low-order byte of R2 are then stored into PACKED.

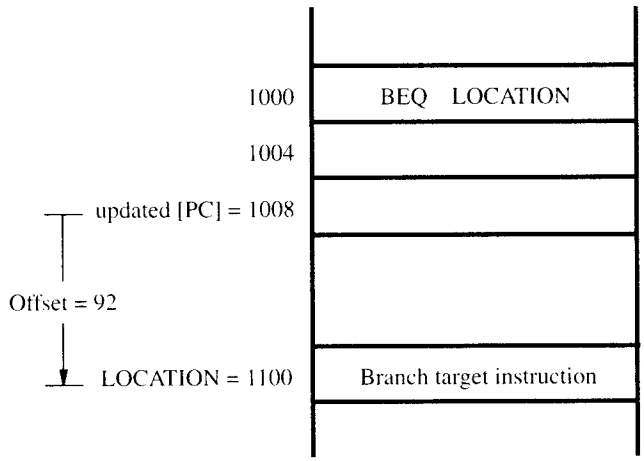
3.3 BRANCH INSTRUCTIONS

Conditional branch instructions contain a signed, 2's-complement, 24-bit offset that is added to the updated contents of the Program Counter to generate the branch target address. The format for the branch instructions is shown in Figure 3.6*a*, and an example is given in Figure 3.6*b*. The BEQ instruction (Branch if Equal to 0) causes a branch if the Z flag is set to 1.

The condition to be tested to determine whether or not branching should take place is specified in the high-order 4 bits, b_{31-28}, of the instruction word. A Branch instruction is executed in the same way as any other ARM instruction, that is, it is executed only if the current state of the condition code flags corresponds to the condition specified in the Condition field of the instruction.



(a) Instruction format



(b) Determination of a branch target address

Figure 3.6 ARM branch instructions.

At the time that the branch target address is computed, the contents of the PC have been updated to contain the address of the instruction that is two words beyond the Branch instruction itself. If the Branch instruction is at address location 1000, and the branch target address is 1100, as shown in Figure 3.6*b*, then the offset has to be 92 because the contents of the updated PC will be 1000 + 8 = 1008 when address 1100 is computed.

3.3.1 SETTING CONDITION CODES

Some instructions, such as Compare, given by

$$\text{CMP} \quad \text{R}n,\text{R}m$$

which performs the operation

$$[\text{R}n] - [\text{R}m]$$

have the sole purpose of setting the condition code flags based on the result of the subtraction operation. On the other hand, the arithmetic and logic instructions affect the condition code flags only if explicitly specified to do so by a bit in the OP-code

```
        LDR   R1,N          Load count into R1.
        LDR   R2,POINTER    Load address NUM1 into R2.
        MOV   R0,#0         Clear accumulator R0.
LOOP    LDR   R3,[R2],#4    Load next number into R3.
        ADD   R0,R0,R3      Add number into R0.
        SUBS  R1,R1,#1      Decrement loop counter R1.
        BGT   LOOP          Branch back if not done.
        STR   R0,SUM        Store sum.
```

Figure 3.7 An ARM program for adding numbers.

field. This is indicated by appending the suffix S to the assembly language OP-code mnemonic. For example, the instruction

```
ADDS  R0,R1,R2
```

sets the condition code flags, but

```
ADD   R0,R1,R2
```

does not.

3.3.2 A LOOP PROGRAM FOR ADDING NUMBERS

Figure 3.7 shows a loop program for adding a list of numbers, patterned after the program in Figure 2.16. The load and store operations performed by the first, second, and last instructions use the Relative addressing mode. This assumes that the memory locations N, POINTER, and SUM are within the range reachable by the offset relative to the PC. Memory location POINTER contains the address NUM1 of the first of the numbers to be added, N contains the number of entries in the list, and SUM is used to store the sum. The Post-indexed addressing mode with writeback in the first instruction of the loop mirrors the use of the Autoincrement addressing mode in Figure 2.16.

3.4 ASSEMBLY LANGUAGE

The ARM assembly language has assembler directives to reserve storage space, assign numerical values to address labels and constant symbols, define where program and data blocks are to be placed in memory, and specify the end of the source program text. These facilities were described in general in Section 2.6.1.

We illustrate some of the ARM directives in Figure 3.8, which gives a complete source program for the program of Figure 3.7. The AREA directive, which uses the argument CODE or DATA, indicates the beginning of a block of memory that contains

	Memory address label	Operation	Addressing or data information
Assembler directives		AREA	CODE
		ENTRY	
Statements that generate machine instructions		LDR	R1,N
		LDR	R2,POINTER
		MOV	R0,#0
	LOOP	LDR	R3,[R2],#4
		ADD	R0,R0,R3
		SUBS	R1,R1,#1
		BGT	LOOP
		STR	R0,SUM
Assembler directives		AREA	DATA
	SUM	DCD	0
	N	DCD	5
	POINTER	DCD	NUM1
	NUM1	DCD	3,-17,27,-12,322
		END	

Figure 3.8 ARM assembly language source program for the program in Figure 3.7.

either program instructions or data. The AREA directive actually requires more parameters to be specified, but they are not relevant for the level of discussion here. The ENTRY directive specifies that program execution is to begin at the following LDR instruction.

In the data area, which follows the code area, the DCD directives are used to label and initialize the data operands. The word locations SUM and N are initialized to 0 and 5, respectively, by the first two DCD directives. The address NUM1 is placed in the pointer location POINTER by the next DCD directive. The last DCD directive specifies that the five numbers to be added are placed in successive memory locations, starting at NUM1.

Constants in hexadecimal notation have a & prefix, and constants in base n, for n between two and nine, are denoted as n_xxx. For example, 2_101100 denotes a binary constant. Base ten constants do not need a prefix.

An EQU directive can be used to define symbolic names for constants. For example, the statement

```
TEN  EQU  10
```

allows TEN to be used in a program instead of the decimal constant 10. When a number of registers are used in a program, it is convenient to use symbolic names for them that relate to their usage. The RN directive is used for this purpose. For example,

```
COUNTER  RN  3
```

establishes the name COUNTER for register R3. The register names R0 to R15, PC (for R15), and LR (for R14) are predefined by the assembler.

3.4.1 PSEUDO-INSTRUCTIONS

An alternative way of loading the address NUM1 into register R2 in Figure 3.8 is also provided in the assembly language. The *pseudo-instruction*

```
ADR  Rd,ADDRESS
```

loads the 32-bit value ADDRESS into R*d*. This instruction is not an actual machine instruction. The assembler chooses appropriate real machine instructions to implement pseudo-instructions. For example, the combination of the machine instruction

```
LDR  R2,POINTER
```

and the data declaration directive

```
POINTER  DCD  NUM1
```

that is used in Figure 3.8 is one way to implement the pseudo-instruction

```
ADR  R2,NUM1
```

which would be placed at the position of the LDR instruction in the program. In this case, the assembler would need to allocate an appropriate data area for the DCD declaration.

A more efficient way to implement the ADR instruction is possible in this particular example, and it is the one that would be chosen by the assembler. When the address value to be loaded by the ADR instruction is within 255 bytes of the current contents of the PC (R15), the instruction

```
ADD  Rd,R15,#offset
```

can be used to implement the ADR pseudo-instruction. If this is done in the example program, the location POINTER is not needed. The assembler implements the ADR pseudo-instruction with the real machine instruction

```
ADD  R2,R15,#28
```

because NUM1 is 28 bytes beyond the updated PC when the ADD instruction is executed. This assumes that the data area immediately follows the STR instruction. This is not actually true because an instruction to return control to the operating system must follow the STR instruction, but it has been omitted.

3.5 I/O OPERATIONS

The ARM architecture uses memory-mapped I/O as described in Section 2.7. Reading a character from a keyboard or sending a character to a display can be done using program-controlled I/O as described in that section.

Suppose that bit 3 in each of the device status registers INSTATUS (keyboard) and OUTSTATUS (display) contains the respective control flags SIN and SOUT. Also assume that the keyboard DATAIN and display DATAOUT registers are located at addresses INSTATUS + 4 and OUTSTATUS + 4, immediately following the status register locations. The read and write wait loops can then be implemented as follows. Assume that address INSTATUS has been loaded into register R1. The instruction sequence

```
READWAIT  LDR   R3,[R1]
          TST   R3,#8
          BEQ   READWAIT
          LDRB  R3,[R1,#4]
```

reads a character into register R3 when a key has been pressed on the keyboard. The test (TST) instruction performs the bitwise logical AND operation on its two operands and sets the condition code flags based on the result. The immediate operand 8 has a single one in the bit 3 position. Therefore, the result of the TST operation will be zero if bit 3 of INSTATUS is zero and will be nonzero if bit 3 is one, signifying that a character is available in DATAIN. The BEQ instruction branches back to READWAIT if the result is zero, looping until a key is pressed, which sets bit 3 of INSTATUS to one.

Assuming that address OUTSTATUS has been loaded into register R2, the instruction sequence

```
WRITEWAIT  LDR   R4,[R2]
           TST   R4,#8
           BEQ   WRITEWAIT
           STRB  R3,[R2,#4]
```

sends the character in register R3 to the DATAOUT register when the display is ready to receive it.

These two routines can be used to read a line of characters from a keyboard, store them in the memory, and echo them back to a display, as shown in the program in Figure 3.9. This program is patterned after the generic program in Figure 2.20. Register R0 is assumed to contain the address of the first byte in the memory area where the line is to be stored. Registers R1 through R4 have the same usage as in the READWAIT and WRITEWAIT loops described above. The first Store instruction (STRB) stores the character read from the keyboard into the memory. The Post-indexed addressing mode with writeback is used in this instruction to step through the memory area, analogous to the use of the Autoincrement addressing mode in Figure 2.20. The Test if Equal (TEQ)

```
READ    LDR     R3,[R1]      Load [INSTATUS] and
        TST     R3,#8          wait for character.
        BEQ     READ
        LDRB    R3,[R1,#4]   Read the character and
        STRB    R3,[R0],#1     store it in memory.
ECHO    LDR     R4,[R2]      Load [OUTSTATUS] and
        TST     R4,#8          wait for display
        BEQ     ECHO           to be ready.
        STRB    R3,[R2,#4]   Send character to display.
        TEQ     R3,#CR       If not carriage return,
        BNE     READ           read more characters.
```

Figure 3.9 An ARM program that reads a line of characters and displays it.

instruction tests whether or not the two operands are equal and sets the Z condition code flag accordingly.

3.6 SUBROUTINES

A Branch and Link (BL) instruction is used to call a subroutine. It operates in the same way as other branch instructions, with one added step. The return address, which is the address of the next instruction after the BL instruction, is loaded into register R14, which acts as a link register. Since subroutines may be nested, the contents of the link register must be saved on a stack by the subroutine. Register R13 is normally used as the pointer for this stack.

Figure 3.10 shows the program of Figure 3.7 rewritten as a subroutine. Parameters are passed through registers. The calling program passes the size of the number list and the address of the first number to the subroutine in registers R1 and R2; and the subroutine passes the sum back to the calling program in register R0. The subroutine also uses register R3. Therefore, its contents, along with the contents of the link register R14, are saved on the stack by the STMFD instruction. The suffix FD in this instruction specifies that the stack grows toward lower addresses and that the stack pointer R13 is to be predecremented before pushing words onto the stack. The LDMFD instruction restores the contents of register R3 and pops the saved return address into the PC (R15), performing the return operation automatically.

Figure 3.11*a* shows the program of Figure 3.7 rewritten as a subroutine with parameters passed on the stack. The parameters NUM1 and *n* are pushed onto the stack by the first four instructions of the calling program. We assume that NUM1 is contained in memory location POINTER. Registers R0 to R3 serve the same purpose inside the subroutine as in Figure 3.7. Their contents are saved on the stack by the first instruction of the subroutine along with the return address in R14. The contents of the stack at

Calling program

```
        LDR     R1,N
        LDR     R2,POINTER
        BL      LISTADD
        STR     R0,SUM
        .
        .
        .
```

Subroutine

```
LISTADD STMFD   R13!,{R3,R14}   Save R3 and return address in R14 on
                                stack, using R13 as the stack pointer.
        MOV     R0,#0
LOOP    LDR     R3,[R2],#4
        ADD     R0,R0,R3
        SUBS    R1,R1,#1
        BGT     LOOP
        LDMFD   R13!,{R3,R15}   Restore R3 and load return address
                                into PC (R15).
```

Figure 3.10 Program of Figure 3.7 written as an ARM subroutine; parameters passed through registers.

various times are shown in Figure 3.11*b*. After the parameters have been pushed and the Call instruction (BL) has been executed, the top of the stack is at level 2. It is at level 3 after all registers have been saved by the first instruction of the subroutine. The next two instructions load the parameters into registers R1 and R2 using offsets of 20 and 24 bytes into the stack, which reach to n and NUM1, respectively, from level 3. When the sum has been accumulated in R0, it is inserted into the stack by the Store instruction (STR), overwriting NUM1.

The last example of subroutines is the case of handling nested calls. Figure 3.12 shows the ARM code for the program of Figure 2.28. The stack frames corresponding to the first and second subroutines are shown in Figure 3.13. Register R12 is used as the frame pointer. Symbolic names are used for some of the registers in this example to aid program readability. Registers R12 (frame pointer), R13 (stack pointer), R14 (link register), and R15 (program counter), are labeled as FP, SP, LR, and PC, respectively. The assembler directive RN can be used to define these names.

The structure of the calling program and the subroutines is the same as in Figure 2.28. Aspects that are specific to ARM are as follows. Both the return address and the old contents of the frame pointer are saved on the stack by the first instruction in each subroutine. The second instruction sets the frame pointer to point to its saved value, as shown in Figure 3.13. This is consistent with the frame pointer position in Figures 2.27 and 2.29. The parameters are then referenced at offsets of 8, 12, and so on, as usual.

(Assume top of stack is at level 1 below.)

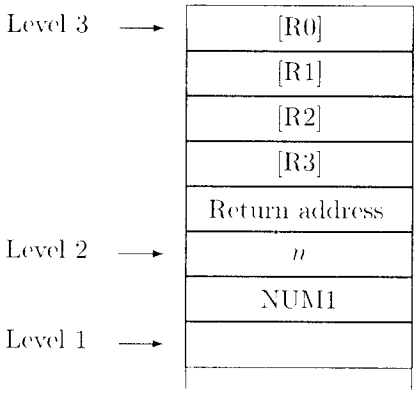
Calling program

```
        LDR     R0.POINTER      Push NUM1
        STR     R0.[R13.#-4]!     on stack.
        LDR     R0.N            Push n
        STR     R0.[R13.#-4]!     on stack.
        BL      LISTADD
        LDR     R0.[R13.#4]     Move the sum into
        STR     R0.SUM            memory location SUM.
        ADD     R13.R13.#8      Remove parameters from stack.
        :
```

Subroutine

```
LISTADD STMFD   R13!.{R0-R3.R14} Save registers.
        LDR     R1.[R13.#20]    Load parameters
        LDR     R2.[R13.#24]      from stack.
        MOV     R0.#0
LOOP    LDR     R3.[R2].#4
        ADD     R0.R0.R3
        SUBS    R1.R1.#1
        BGT     LOOP
        STR     R0.[R13.#24]    Place sum on stack.
        LDMFD   R13!.{R0-R3.R15} Restore registers and return.
```

(a) Calling program and subroutine



(b) Top of stack at various times

Figure 3.11 Program of Figure 3.7 written as an ARM subroutine; parameters passed on the stack.

Memory location		Instructions		Comments
Calling program				
		⋮		
2000		LDR	R10.PARAM2	Place parameters on stack.
2004		STR	R10.[SP.#−4]!	
2008		LDR	R10.PARAM1	
2012		STR	R10.[SP.#−4]!	
2016		BL	SUB1	
2020		LDR	R10.[SP]	Store SUB1 result.
2024		STR	R10.RESULT	
2028		ADD	SP.SP.#8	Remove parameters from stack.
2032		next instruction		
		⋮		
First subroutine				
2100	SUB1	STMFD	SP!.{R0−R3.FP.LR}	Save registers.
2104		ADD	FP.SP.#16	Load frame pointer.
2108		LDR	R0.[FP.#8]	Load parameters.
2112		LDR	R1.[FP.#12]	
		⋮		
		LDR	R2.PARAM3	Place parameter on stack.
		STR	R2.[SP.#−4]!	
2160		BL	SUB2	
2164		LDR	R2.[SP].#4	Pop SUB2 result into R2.
		⋮		
		STR	R3.[FP.#8]	Place result on stack.
		LDMFD	SP!.{R0−R3.FP.PC}	Restore registers and return.
Second subroutine				
3000	SUB2	STMFD	SP!.{R0.R1.FP.LR}	Save registers.
		ADD	FP.SP.#8	Load frame pointer.
		LDR	R0.[FP.#8]	Load parameter.
		⋮		
		STR	R1.[FP.#8]	Place result on stack.
		LDMFD	SP!.{R0.R1.FP.PC}	Restore registers and return.

Figure 3.12 Nested subroutines in ARM assembly language.

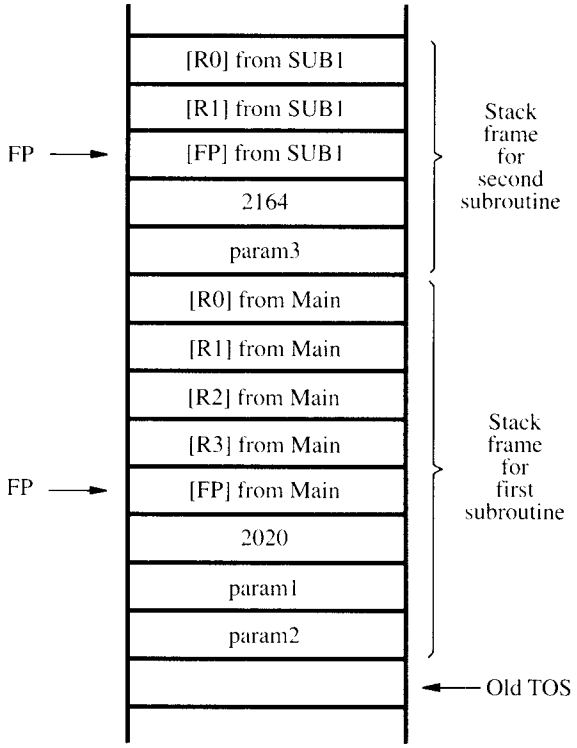


Figure 3.13 ARM stack frames for Figure 3.12.

The last instruction in each subroutine restores the old value of the frame pointer as well as the values of the other registers used, and pops the return address from the stack into the PC.

3.7 PROGRAM EXAMPLES

In this section, we give ARM versions of the programs for dot product, byte sorting, and linked-list operations that were described in Chapter 2. The programs are patterned after the generic programs shown in Figures 2.33, 2.34, 2.37, and 2.38. We will describe only those aspects of the ARM code that differ from the generic versions used in Chapter 2.

3.7.1 VECTOR DOT PRODUCT PROGRAM

The first two instructions in Figure 3.14 load the addresses of the A and B vectors into registers R1 and R2. They are the ADR pseudo-instructions described in Section 3.4.1. If AVEC and BVEC are close enough to the program, an Add instruction using the current

```
        ADR   R1,AVEC       R1 points to vector A.
        ADR   R2,BVEC       R2 points to vector B.
        LDR   R3,N          R3 is the loop counter.
        MOV   R0,#0         R0 accumulates the dot product.
LOOP    LDR   R4,[R1],#4    Load A component.
        LDR   R5,[R2],#4    Load B component.
        MLA   R0,R4,R5,R0   Multiply components and
                              accumulate into R0.
        SUBS  R3,R3,#1      Decrement the counter.
        BNE   LOOP          Branch back if not done.
        STR   R0,DOTPROD    Store dot product.
```

Figure 3.14 An ARM dot-product program.

value of the PC can be used to generate the addresses. The Relative addressing mode is used to access the contents of N and DOTPROD, and the Post-indexed addressing mode with writeback is used in the first two instructions of the loop. The Multiply-Accumulate instruction (MLA) performs the necessary arithmetic operations. It multiplies the vector elements in R4 and R5 and accumulates their product into R0.

3.7.2 BYTE-SORTING PROGRAM

Figure 3.15 shows the byte-sorting program. It follows the same structure as used in the program in Figure 2.34*b*. The address LIST of the first byte is loaded into register R4. It is used in the second to the last Compare instruction to determine when the inner loop (based on the k index) terminates. Correspondingly, R5 contains the address LIST + 1 and is used in the last Compare instruction to determine when the outer loop (based on the j index) terminates. The base register R2 is used to step the j index backward from the end of the list in the outer loop. Register R3 steps the k index backward through each sublist in the inner loop. The Pre-indexed addressing mode with writeback is used to load LIST(j) bytes into register R0 and to load LIST(k) bytes into R1 in the outer and inner loops, respectively.

The conditional execution feature of the ARM instruction set is used to advantage in the inner loop when LIST(k) must be interchanged with LIST(j). The three-instruction sequence STR, STR, MOV is only executed if LIST(k) is greater than LIST(j), as indicated by the GT suffixes. The forward conditional branch to NEXT in the generic program in Figure 2.34*b* is not needed in the ARM program.

3.7.3 LINKED-LIST INSERTION AND DELETION SUBROUTINES

The insertion and deletion subroutine programs in Figures 3.16 and 3.17 mirror the structure of the corresponding programs in Figures 2.37 and 2.38 quite closely. The forward conditional branches used in the generic programs are not needed in the ARM

```
for   (j = n-1: j > 0: j = j - 1)
      { for ( k = j-1: k >= 0: k = k - 1 )
          { if   (LIST[k] > LIST[j])
                 {  TEMP = LIST[k]:
                    LIST[k] = LIST[j]:
                    LIST[j] = TEMP:
                 }
          }

      }
```

(a) C-language program for sorting

```
        ADR     R4.LIST      Load list pointer register R4.
        LDR     R10.N          and initialize outer loop base
        ADD     R2.R4.R10      register R2 to LIST + n.
        ADD     R5.R4.#1     Load LIST + 1 into R5.
OUTER   LDRB    R0.[R2.#-1]! Load LIST(j) into R0.
        MOV     R3.R2        Initialize inner loop base register
                               R3 to LIST + n - 1.
INNER   LDRB    R1.[R3.#-1]! Load LIST(k) into R1.
        CMP     R1.R0        Compare LIST(k) to LIST(j).
        STRGTB  R1.[R2]      If LIST(k) > LIST(j), interchange
        STRGTB  R0.[R3]        LIST(k) and LIST(j), and
        MOVGT   R0.R1          move (new) LIST(j) into R0.
        CMP     R3.R4        If k > 0, repeat
        BNE     INNER          inner loop.
        CMP     R2.R5        If j > 1, repeat
        BNE     OUTER          outer loop.
```

(b) ARM program implementation

Figure 3.15 An ARM byte-sorting program.

programs. This is a result of the use of conditional execution of instruction blocks, as done in the byte-sorting program in Figure 3.15. Parameters are passed through registers in both ARM subroutines.

Register mnemonics are used to reflect register usage, instead of the usual R*i* notation. The assembler directive RN can be used to define the equivalences. As in the programs in Figures 2.37 and 2.38, RHEAD contains the address of the first record in

Subroutine

```
INSERTION  CMP     RHEAD.#0                 Check if list empty.
           MOVEQ   RHEAD.RNEWREC            If empty, insert new
           MOVEQ   PC.R14                     record as head.
           LDR     R0.[RHEAD]               If not empty, check if
           LDR     R1.[RNEWREC]               new record becomes
           CMP     R0.R1                      new head, and
           STRGT   RHEAD.[RNEWREC.#4]         insert if yes.
           MOVGT   RHEAD.RNEWREC
           MOVGT   PC.R14
           MOV     RCURRENT.RHEAD           If new record goes after
LOOP       LDR     RNEXT.[RCURRENT.#4]        current head,
           CMP     RNEXT.#0                   find where.
           STREQ   RNEWREC.[RCURRENT.#4]    New record becomes new tail.
           MOVEQ   PC.R14
           LDR     R0.[RNEXT]               Go further?
           CMP     R0.R1
           MOVLT   RCURRENT.RNEXT           Yes, then loop back.
           BLT     LOOP
           STR     RNEXT.[RNEWREC.#4]       Otherwise, insert new record
           STR     RNEWREC.[RCURRENT.#4]      between current and
           MOV     PC.R14                     next records.
```

Figure 3.16 An ARM subroutine for inserting a new record into a linked list.

Subroutine

```
DELETION   LDR     R0.[RHEAD]               Check if record to be
           CMP     R0.RIDNUM                  deleted is the head.
           LDREQ   RHEAD.[RHEAD.#4]         If yes, delete
           MOVEQ   PC.R14                     and return.
           MOV     RCURRENT.RHEAD           Otherwise, continue search.
LOOP       LDR     RNEXT.[RCURRENT.#4]      Is next record the one
           LDR     R0.[RNEXT]               to be deleted?
           CMP     R0.RIDNUM
           LDREQ   R0.[RNEXT.#4]            If yes, delete
           STREQ   R0.[RCURRENT.#4]           and return.
           MOVEQ   PC.R14
           MOV     RCURRENT.RNEXT           Otherwise, loop back
           B       LOOP                       to continue search.
```

Figure 3.17 An ARM subroutine for deleting a record from a linked list.

the list. RNEWREC contains the address of the new record to be inserted. RIDNUM contains the ID number of the record to be deleted. The two registers RCURRENT and RNEXT contain link addresses that are used by the subroutines to walk through the list to find the insertion or deletion positions.

The insertion subroutine in Figure 3.16, patterned after the subroutine in Figure 2.37, has the following structure. The first three instructions insert the new record as the head (and tail) of a previously empty list. Recall that the new record is assumed to initially have zero in its link field. The third instruction in this block performs the return operation from the subroutine to the calling program. The next six instructions determine whether or not the new record should become the new head of the existing list. The list is ordered by increasing ID numbers. Therefore, if the ID number in the first word of the current head record is greater than the ID number of the new record, then the new record becomes the new head of the list. The conditionally executed STRGT and MOVGT instructions perform the appropriate link address operations if this is the case. Otherwise, the remaining part of the subroutine determines where the new record should be inserted in the list after the current head, including the possibility that the new record becomes the tail.

The deletion subroutine is shown in Figure 3.17. If the record to be deleted is the head of the list, the first four instructions discover this, delete it, and return. Otherwise, the remainder of the subroutine uses registers RCURRENT and RNEXT to move through the list looking for the record. The LDREQ/STREQ pair of instructions delete the record when it is found to be the one pointed to by RNEXT.

As with the generic programs in Figures 2.37 and 2.38, the insertion subroutine in Figure 3.16 assumes that the ID number of the new record does not match that of any record already in the list, and the deletion subroutine in Figure 3.17 assumes that there exists a record in the list with an ID number that does match the contents of RIDNUM. Problems 3.23 and 3.24 consider how the subroutines should be changed to signal an error if the assumptions are not true.

PART II
THE 68000 EXAMPLE

In this second part of Chapter 3, we describe the basic architecture of processors in Motorola's 680X0 family by discussing the 68000 ISA. The family includes several processors that provide different performance levels. All members of the family have the same basic architecture, but later members have additional features that enhance their performance. We use the 68000 here because it is somewhat simpler to describe, yet it portrays the salient features of the entire family. We do not provide a comprehensive description of the 68000. For such information, the reader can consult manufacturer's information [6]. Instead, we concentrate on the most important aspects of the 68000, giving sufficient detail to enable the reader to prepare, assemble, and run simple programs. The distinguishing features of various members of the 680X0 family, as well as some of the features introduced for performance enhancement, are described in Chapter 11. The programs from Chapter 2 are presented here in the 68000 assembly language to illustrate various aspects of the 68000 architecture.

3.8 REGISTERS AND ADDRESSING

The 68000 processor is characterized by a 16-bit external word length because the processor chip has 16 data pins for connection to the memory. However, data are manipulated inside the processor in registers that contain 32 bits. The more advanced models of this family are the 68020, 68030, and 68040 processors, which come in larger chip packages and have 32 external data pins. Thus, they can deal with data both internally and externally in 32-bit quantities. Tabak [7] covers these members of the 680X0 family, emphasizing the 68040.

3.8.1 THE 68000 REGISTER STRUCTURE

The 68000 register structure, shown in Figure 3.18, has 8 data registers and 8 address registers, each 32 bits long. The data registers serve as general-purpose accumulators and as counters.

The 68000 instructions deal with operands of three different lengths. A 32-bit operand is said to occupy a *long word*, a 16-bit operand constitutes a word, and an 8-bit operand is a byte. When an instruction uses a byte or a word operand in a register, the operand is in the low-order bit positions of the register. In most cases, such instructions do not affect the remaining high-order bits of the register, but some instructions extend the sign of a shorter operand into the high-order bits.

The address registers hold information used in determining the addresses of memory operands. This information may be given in either long word or word sizes. When the address of a given memory location is in an address register, the register serves as a pointer to that location. Both address and data registers can also be used as index registers. One address register, A7, has the special function of being the processor stack pointer. The role of this register is discussed in Section 3.13.

The address registers and address calculations involve 32 bits. However, in the 68000, only the least-significant 24 bits of an address are used externally to access the memory. The 68020, 68030, and 68040 processors have 32 external address lines as well as 32 data lines.

The last register shown in Figure 3.18 is the processor *status register*, SR. It contains five condition code bits, which are described in Section 3.11.1; three interrupt bits, which are discussed in Chapter 4; and two mode-select bits, which are explained in Section 3.13.

3.8.2 ADDRESSING

The memory of a 68000 computer is organized in 16-bit words and is byte addressable. Two consecutive words can be interpreted as a single 32-bit long word. Memory addresses are assigned as shown in Figure 3.19. A word must be aligned on an even boundary, that is, its address must be an even number. The big-endian address assignment is used. The byte in the high-order position of a word has the same address as the word, whereas the byte in the low-order position has the next higher address.

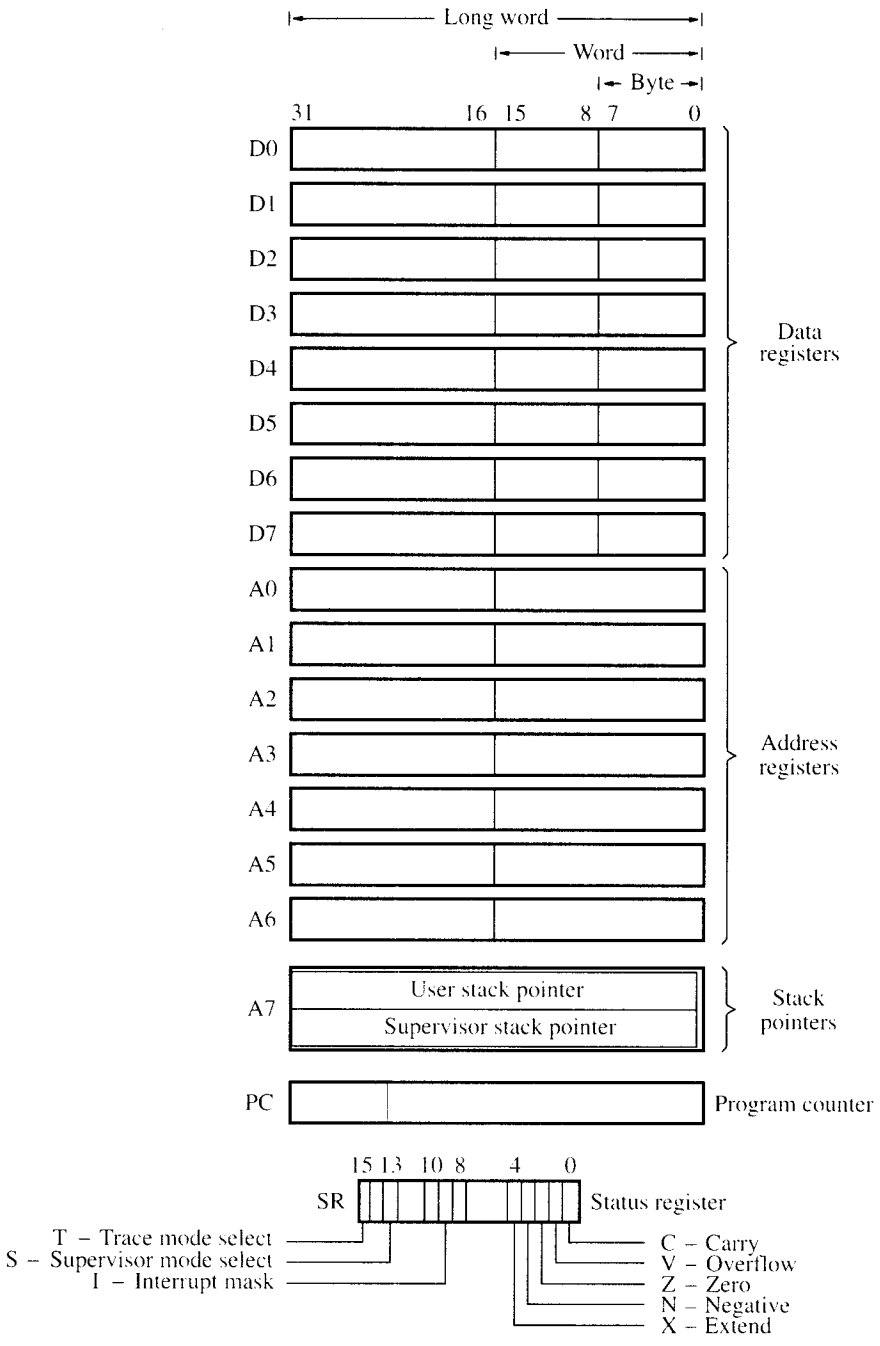


Figure 3.18 The 68000 register structure.

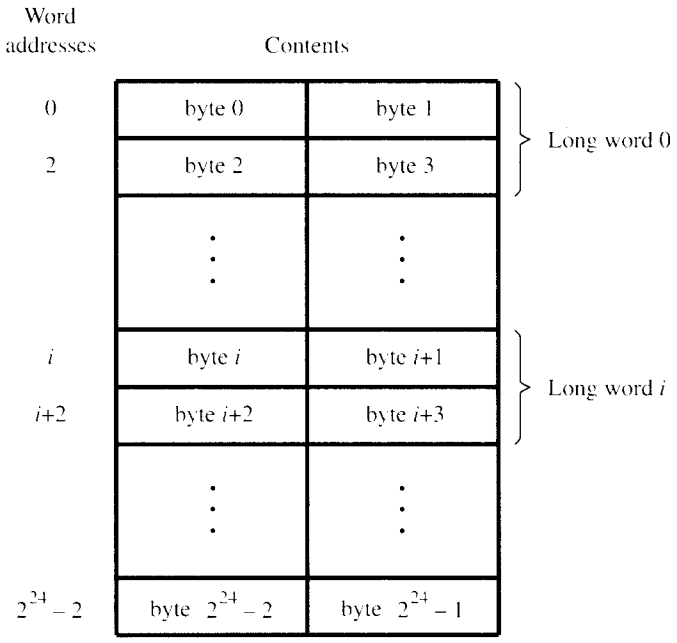


Figure 3.19 Map of addressable locations in the 68000.

Since the 68000 generates 24-bit addresses, its addressable space is 2^{24} (16,777,216 or 16M) bytes. This addressable space may be thought of as consisting of 512 (2^9) *pages* of 32K (2^{15}) bytes each. Thus, hexadecimal addresses 0 to 7FFF constitute page 0, addresses 8000 to FFFF make up page 2, and so on. The last page consists of addresses FF8000 to FFFFFF.

The 68000 has several addressing modes, including those discussed in Section 2.5. Many of the 68000's instructions fit into a 16-bit word, but some require additional words for extra addressing information. The first word of an instruction is the OP-code word, which specifies the operation to be performed and gives some addressing information. The rest of the addressing information, if any, is given in subsequent words. The available addressing modes are defined as follows:

Immediate mode — The operand is contained in the instruction. Four sizes of operands can be specified. Byte, word, and long-word operands are given following the OP-code word. The fourth size consists of very small numbers that can be included directly in the OP-code word of some instructions.

Absolute mode — The absolute address of an operand is given in the instruction, following the OP code. There are two versions of this mode — long and short. In the long mode, a 24-bit address is specified explicitly. In the short mode, a 16-bit value is given in the instruction to be used as the low-order 16 bits of an address. The sign bit of this value is extended to provide the high-order eight bits of the address. Since the sign bit is either 0 or 1, it follows that in the short

mode only two pages of the addressable space can be accessed. These are the 0 page and the FF8 page, each containing 32K bytes.

Register mode — The operand is in a processor register specified in the instruction.

Register indirect mode — The effective address of the operand is in an address register specified in the instruction.

Autoincrement mode — The effective address of the operand is in an address register, A*n*, specified in the instruction. After the operand is accessed, the contents of A*n* are incremented by 1, 2, or 4, depending on whether a byte, a word, or a long-word operand, respectively, is involved.

Autodecrement mode — The contents of an address register, A*n*, specified in the instruction are decremented by 1, 2, or 4, depending on whether a byte, a word, or a long-word operand, respectively, is involved. The effective address of the operand is the decremented contents of A*n*.

Basic index mode — A 16-bit signed offset and an address register, A*n*, are specified in the instruction. The sum of this offset and the contents of A*n* is the effective address of the operand.

Full index mode — An 8-bit signed offset, an address register A*n*, and an index register R*k* (either an address or a data register) are given in the instruction. The effective address of the operand is the sum of the offset and the contents of registers A*n* and R*k*. Either all 32 bits or the sign-extended low-order 16 bits of R*k* are used in the derivation of the address.

Basic relative mode — This is the same mode as the basic index mode except that the program counter is used instead of an address register, A*n*.

Full relative mode — This is the same mode as the full index mode except that the program counter is used instead of an address register, A*n*.

The addressing modes and their assembler syntax are summarized in Table 3.2.

Note that there are two versions of the index mode. The basic index mode corresponds to the mode depicted in Figure 2.13. The full index mode involves the contents of two registers and an offset constant given in the instruction. The size of the offset constant is 16 bits in the basic mode and 8 bits in the full mode.

In the full index mode, the second register, R*k*, can be used in two ways: either all 32 bits are used or only the low-order 16 bits are used. The two possibilities are indicated to the assembler by appending a size indicator — L for a long word or W for a word — to the name of the register, for example, D1.L or D1.W. The latter is the default size if no indicator is given. When a 16-bit word is used in the computation of a 32-bit effective address, this word is sign extended.

In either of the two index modes, the program counter may be used in place of the address register. The resulting addressing modes are called the relative modes because the effective address is specified in terms of the distance between the operand and the instruction that refers to it. Consider the instruction

```
ADD  100(PC,A1),D0
```

Table 3.2 68000 addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Absolute Short	Value	EA = Sign Extended WValue
Absolute Long	Value	EA = Value
Register	Rn	EA = R_n that is, Operand = [R_n]
Register Indirect	(An)	EA = [A_n]
Autoincrement	(An)+	EA = [A_n]; Increment A_n
Autodecrement	−(An)	Decrement A_n; EA = [A_n]
Indexed basic	WValue(An)	EA = WValue + [A_n]
Indexed full	BValue(An,Rk.S)	EA = BValue + [A_n] +[R_k]
Relative basic	WValue(PC) or Label	EA = WValue + [PC]
Relative full	BValue(PC,Rk.S) or Label (Rk)	EA = BValue + [PC] + [R_k]

EA = effective address
Value = a number given either explicitly or represented by a label
BValue = an 8-bit Value
WValue = a 16-bit Value
A_n = an address register
R_n = an address or a data register
S = a size indicator: W for sign-extended 16-bit word and L for 32-bit long word

When encoded in machine form, this instruction consists of two words. The OP-code word specifies that this is an Add instruction, that the destination register is data register D0, and that the full relative addressing mode is used for the source operand. The second word, also called the *extension word*, specifies that register A1 is used as the index register and it contains the offset value 100 encoded in 8 bits.

Assume that the preceding instruction is stored in location 1000 and that register A1 contains the value 6, as shown in Figure 3.20. When the OP-code word of this instruction has been fetched and while it is being decoded by the processor, the program counter points to the extension word, which means that the program counter contains the value 1002. Therefore, the effective address of the source operand is

$$\begin{aligned} EA &= [PC] + [A1] + 100 \\ &= 1002 + 6 + 100 \\ &= 1108 \end{aligned}$$

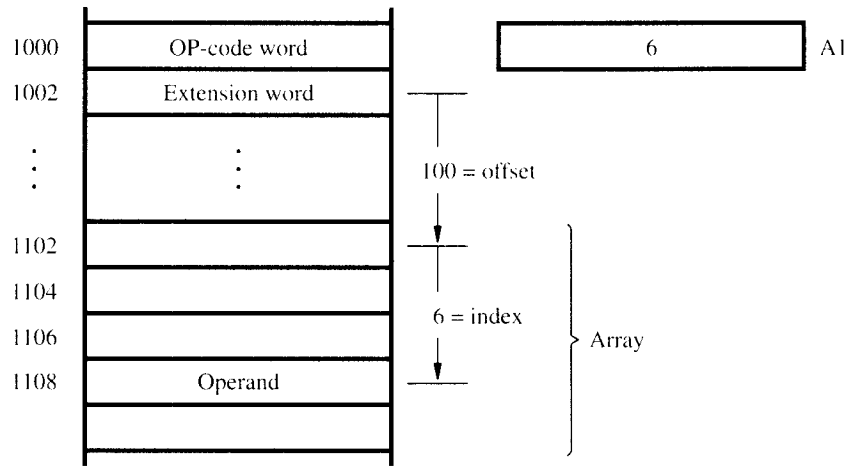


Figure 3.20 An example of 68000 full relative mode for the instruction ADD 100(PC,A1),D0.

Figure 3.20 suggests how this addressing mode can be used to access an entry in an array. The offset value specifies the distance between the first entry in the array and the instruction. Then the index register gives the distance between that point and the desired operand, which is the fourth word in the array.

We have written the relative mode in an explicit format. Most assemblers allow this mode to be specified in a simpler way. First, the assembler must be informed that relative addressing is to be used in a given section of the program through an appropriate assembler directive. Next, after the name ARRAY has been assigned the value 1102, the instruction in Figure 3.20 can be written as

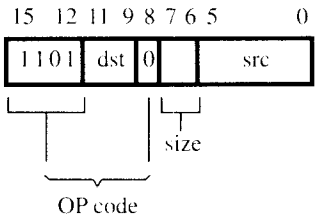
```
ADD   ARRAY(A1),D0
```

The assembler interprets this specification of the source operand as being in the full relative mode, and it computes the offset as indicated in the figure. The assembler does not know, and does not need to know, what the contents of register A1 will be when the instruction is executed. For example, this instruction may be inside a program loop, in which case A1 could be used to access successive elements of the array.

The full relative mode is limited by the fact that the offset is a 2's-complement 8-bit number, thus restricting its values to the range −128 to +127 bytes.

3.9 INSTRUCTIONS

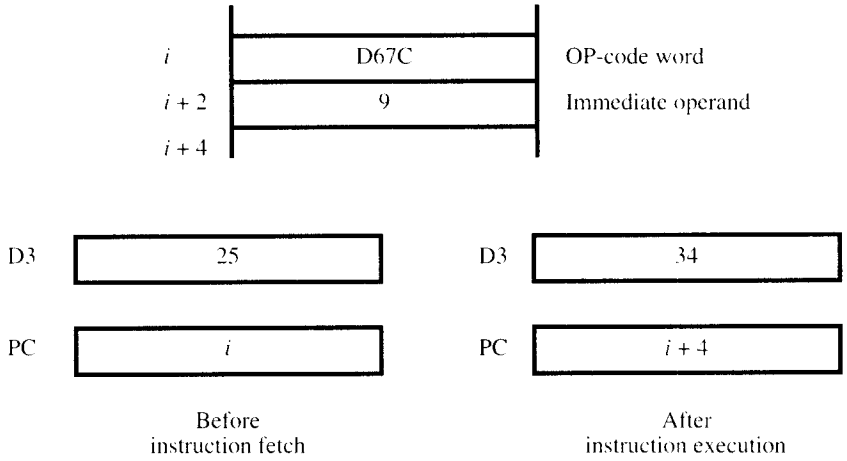
The 68000 ISA provides an extensive set of instructions, most of which can operate on any of the three possible sizes of operands. The instruction set is summarized in Appendix C. All addressing modes can be used in a uniform way with most instructions. Instruction sets that exhibit this feature are said to be *orthogonal*.



(a) Format of the OP-code word for an ADD instruction

Binary	1101	0110	0111	1100
Hex	D	6	7	C

(b) Encoding of the OP-code word



(c) Consequences of the execution

Figure 3.21 The 68000 instruction ADD #9,D3.

The 68000 has both one-operand and two-operand instructions. A two-operand instruction is written as

OP src,dst

where the operation OP is performed using the source and destination operands. The result is placed in the destination location. An example is given in Figure 3.21, which

shows the instruction

ADD #9,D3

This instruction performs the action

dst ← [src] + [dst]

which results in adding the value 9 to the contents of register D3 and storing the result back in D3.

Figure 3.21*a* depicts the general format of the ADD instruction. Either the source operand or the destination operand must be in a data register, D*n*. The second operand may be in a register or a memory location. The allowable combinations are given in Table C.4. Since at least one of the two operands is always in one of the eight data registers, a 3-bit field suffices to identify it. The other operand is specified according to Table C.1. In our example, the destination register D3 is represented by the binary pattern 011 in bits 9 through 11, and the immediate source operand is identified with the pattern 111100 in bits 0 through 5.

The desired operand size is indicated in the 2-bit size field. In our example, the size of the operands is not stated explicitly in the assembly language statement, in which case the assembler assumes the default value of a 16-bit word. According to Table C.3, word-size operands are denoted by the pattern 01.

From the discussion above, it follows that the OP-code word for our ADD instruction is 1101011001111100, which is represented by the hex number D67C, as indicated in Figure 3.21*b*.

The immediate source operand, 9, is given in the word following the OP-code word, as shown in Figure 3.21*c*. Before fetching the instruction, the program counter points to the OP-code word at address *i*. As each word is fetched from the memory, the contents of the PC are incremented by 2. Thus, when execution of the instruction is completed, the PC points to the OP-code word of the next instruction at address $i + 4$.

A similar instruction using the same format is the Subtract instruction, SUB, which performs the operation

dst ← [dst] − [src]

As Table C.4 shows, the ADD and SUB instructions allow considerable flexibility in specifying one of the two operands. However, the second operand must be in a data register. Most other two-operand instructions have the same type of restriction. The only instruction in which both the source and the destination operands may be specified in terms of most of the addressing modes is the Move instruction, MOVE, which performs the action

dst ← [src]

Let us now consider a simple routine for the task C ← [A] + [B], shown in Figure 2.8. The required task can be performed as follows

```
MOVE   A,D0
ADD    B,D0
MOVE   D0,C
```

These instructions may be stored in the memory of a 68000 computer, as shown in Figure 3.22. The figure shows hexadecimal values for the addresses and operands. The operands are assumed to be 16 bits long, and their addresses are specified in the absolute mode. Note that the long version of the absolute mode is needed because the desired addresses cannot be represented in 16 bits. The high-order 16 bits of a 32-bit address are placed in the lower address word and the low-order 16 bits in the higher address word, according to the convention shown in Figure 3.19.

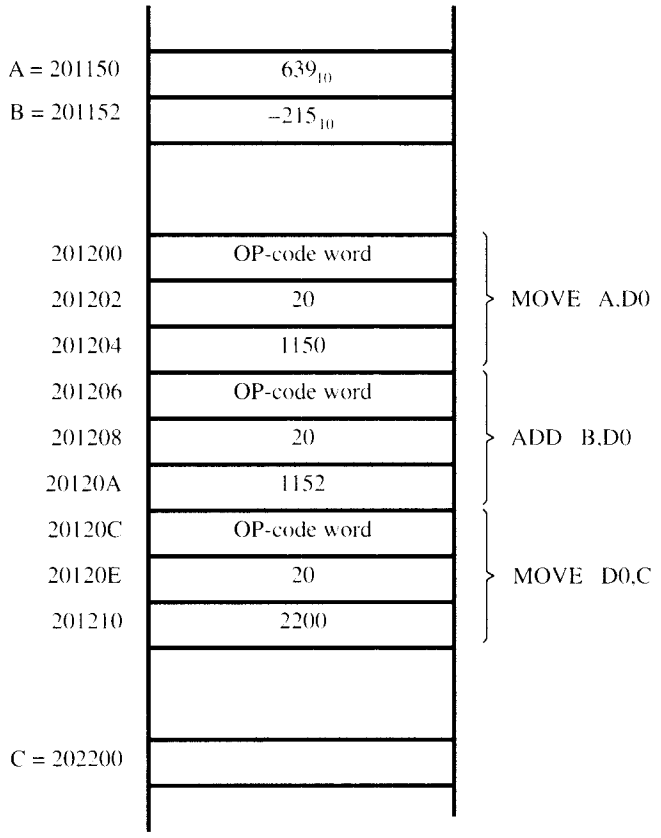


Figure 3.22 A 68000 routine for C ← [A] + [B].

3.10 ASSEMBLY LANGUAGE

The discussion of assembly languages in Section 2.6 applies generally to the 68000 assembly language. Some minor differences and additions are explained here.

Because 68000 instructions can deal with three different sizes of operands, the assembler instructions must indicate the desired size. This is done by appending the size indicator to the operation mnemonic. The size indicator is L for long word, W for word, and B for byte. Thus, if an Add instruction is to operate on long-word operands, its operation mnemonic is written as ADD.L. When no size indication is given, the operand size is taken to be one word. This means that the instructions ADD.W #20,D1 and ADD #20,D1 are identical.

Numbers in a source program are assumed to be in decimal representation unless marked with the prefix $ for hexadecimal or % for binary. Alphanumeric characters placed between single quotes are replaced by the assembler with their ASCII codes. Several characters may be specified in a string between quotes. For example, a valid character string is 'STRING3'.

All of the assembler directives discussed in Section 2.6 can be used with only slight differences in notation. The starting address of a block of instructions or data is specified with the ORG directives. The EQU directives equates names with numerical values. Data constants are inserted into an object program using the DC (Define Constant) directives. The size indicator is appended to specify the size of the data items, and several items may be defined in one directives. For example, the directives

```
        ORG    100
PLACE   DC.B   23,$4F,%10110101
```

	Memory address label	Operation	Addressing or data information
Assembler directives	C	EQU	$202200
		ORG	$201150
	A	DC.W	639
	B	DC.W	−215
		ORG	$201200
Statements that generate machine instructions		MOVE	A,D0
		ADD	B,D0
		MOVE	D0,C
Assembler directive		END	

Figure 3.23 68000 assembly language representation for the routine in Figure 3.22.

result in hex values 17 (23_{10}), 4F, and B5 being loaded into memory locations 100, 101, and 102, respectively. The label PLACE is assigned the value 100.

A block of memory can be reserved for data by means of the DS (Define Storage) directive. For instance, the directive

```
ARRAY   DS.L   200
```

reserves 200 long words and associates the name ARRAY with the address of the first long word.

A simple example of a 68000 assembly language program that corresponds to Figure 3.22 is given in Figure 3.23.

3.11 PROGRAM FLOW CONTROL

Branch instructions are needed to implement program structures such as **if** statements and loops. In general, a branch instruction tests a branch condition and then, depending on the result, causes execution to proceed along one of two possible paths. The conditions tested relate to the result of a recently performed operation.

3.11.1 CONDITION CODE FLAGS

The 68000 has five condition code flags, stored in the status register shown in Figure 3.18. In addition to the N, Z, V, and C flags described in Section 2.4.6, the 68000 has a fifth flag, X (extend). It is set in the same way as the C flag, but it is not affected by as many instructions. This apparent duplication is convenient when dealing with multiple-precision operations, which we will discuss in Chapter 6.

Table C.4 in Appendix C shows which flags are affected by each instruction. The C and X flags are set to 1 if a carry occurs from the most-significant bit position as a result of performing an add operation. The C and X flags are set to 1 if no carry occurs as a result of performing a subtract operation, signifying a borrow signal. Since operands can be specified in any of three possible lengths, these two flags depend on the carry-out from bit positions 7, 15, and 31, for byte, word, and long-word operands, respectively. The MOVE instruction sets the N and Z flags according to the operand moved and clears the C and V flags. MOVE does not affect the X flag unless the destination specified is the status register itself.

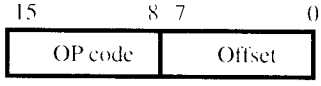
3.11.2 BRANCH INSTRUCTIONS

A conditional branch instruction causes program execution to continue with the instruction at the branch target address if the branch condition is met. This address is determined from the branch offset in the operand field. Otherwise, if the branch condition is not met, the instruction that immediately follows the branch instruction is executed. The 68000 provides branch instructions with two types of offset. In the first type, a short offset of 8 bits is included in the OP-code word. These instructions can be

used when the branch target is within +127 or −128 bytes of the value in the program counter at the time the branch address is computed. Recall that the PC contents are incremented as each word is fetched from the memory, which means that the offset defines the distance from the word that follows the branch instruction OP-code word. In the second type, a 16-bit offset is specified in the word that follows the OP-code word. This provides for a much greater range (±32K) within which the branch target can be located. In this case, the offset is the distance from the extension word to the branch target.

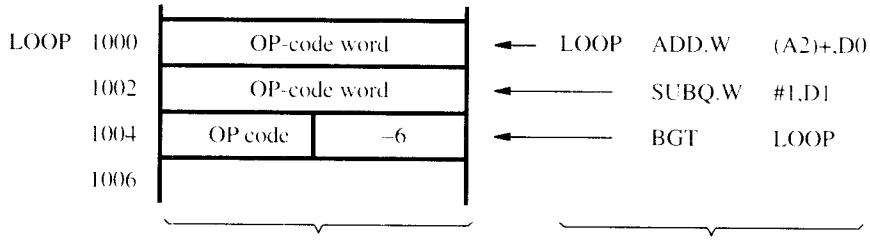
Figure 3.24 illustrates the use of a short-offset branch instruction. It shows how the program loop in Figure 2.16 can be implemented using a 68000 processor. Note that the program in Figure 2.16 uses a Decrement instruction. Since the 68000 does not have such an instruction, we have used the Subtract Quick instruction, SUBQ, which subtracts the immediate operand 1 from the contents of register D1. A 3-bit immediate operand is included within the OP-code word of the SUBQ instruction; thus, only one word is needed to represent the instruction.

The 68000 has 16 conditional branch instructions, each with 8- and 16-bit offsets. It also has an unconditional branch instruction, BRA, where the branch is always taken. Tables C.5 and C.6 give the details of these instructions.



Branch address = [updated PC] + offset

(a) Short-offset branch instruction format



[PC] = 1006 when branch address is computed
Branch address = 1006 − 6 = 1000

(b) Example of using a branch instruction in the loop of Figure 2.16

Figure 3.24 68000 short-offset branch instructions.

```
        MOVE.L    N,D1          N contains n, the number of entries
                                  to be added, and D1 is used as
                                  a counter that determines how
                                  many times to execute the loop.
        MOVEA.L   #NUM1,A2      A2 is used as a pointer to the list entries.
                                  It is initialized to NUM1, the address
                                  of the first entry.
        CLR.L     D0            D0 is used to accumulate the sum.
LOOP    ADD.W     (A2)+,D0      Successive numbers are added in D0.
        SUBQ.L    #1,D1         Decrement the counter.
        BGT       LOOP          If [D1] ≠ 0, execute the loop again.
        MOVE.L    D0,SUM        Store the sum in SUM.
```

Figure 3.25 A 68000 program for the addition program in Figure 2.16.

Figure 3.25 shows a 68000 program for the program in Figure 2.16. It uses data registers D0 and D1 to accumulate the sum and to act as a counter, respectively, and uses address register A2 to point to the operands as they are fetched from the memory. Note that an address register is used because, in the Autoincrement addressing mode, only address registers are allowed.

Decrement and Branch Instructions

In addition to the normal branch instructions, the 68000 has a set of more complex branch instructions that incorporate a counting mechanism. Such a facility is useful for implementing loop control. These instructions are written in the format

DBcc D*n*,LABEL

where the suffix cc denotes a branch condition. For example, if GT is used in place of cc, the resultant instruction, DBGT, is the Decrement and Branch unless Greater Than instruction. The full set of possible branch conditions is given in Table C.6. The way the branch condition is used in these instructions is opposite to the way it is used in other branch instructions. The action is as follows:

- If the condition specified by cc is satisfied, then the instruction that immediately follows the DBcc instruction is executed next.
- If the condition specified by cc is not satisfied, then the least-significant 16 bits of register D*n* are decremented by 1. If the result is equal to −1, the instruction that follows the DBcc instruction is executed next. If the result is not equal to −1, a branch is made to the instruction at location LABEL.

The DBcc instructions are more powerful than normal branch instructions because the decision on whether the branch is to be taken depends on two conditions rather than one. If the same action were specified using normal branch instructions, it would be

necessary to use a sequence of three instructions: first, a branch instruction that tests the cc condition; next, an instruction that decrements the contents of the counter register; and finally, another branch instruction that causes a branch based on the result of the decrement operation. For example, the instructions

```
DBcc   D3,LOOP
next instruction
```

are equivalent to the sequence

```
       Bcc     NEXT
       SUBQ    #1,D3
       BGE     LOOP
NEXT   next instruction
```

A useful way of thinking about the DBcc instructions is to view them as providing convenient means for loop control where early exit from the loop occurs when a given condition is met. The number of times that the loop can be executed is limited by the contents of the counter register, which is D3 in the preceding example.

One DBcc instruction, DBF (Decrement and Branch if False), uses a test condition that is always false. Thus, the decision on whether a branch is to be made is based solely on the result of decrementing the counter register. This instruction is useful when a loop is always executed a predetermined number of times. It is even given a second name, DBRA (Decrement and Branch Always).

To demonstrate the usefulness of decrement and branch instructions, the program of Figure 3.25 can be rewritten using the DBRA instruction, as shown in Figure 3.26. Register D1 is initialized to the value n in Figure 3.25. However, because the DBRA instruction causes a branch when the counter register contains a value equal to or greater than zero, register D1 is initialized to the value $n - 1$ in Figure 3.26. The total number of instructions in the two programs is the same, but the program in Figure 3.26 takes less time to execute because of the shorter loop.

```
       MOVE.L   N,D1          Put n - 1 into the
       SUBQ.L   #1,D1           counter register D1.
       MOVEA.L  #NUM1,A2
       CLR.L    D0
LOOP   ADD.W    (A2)+,D0
       DBRA     D1,LOOP       Loop back until [D1]= -1.
       MOVE.L   D0,SUM
```

Figure 3.26 An alternative 68000 program for the program in Figure 3.25.

3.12 I/O OPERATIONS

The 68000 processor requires that all status and data buffers in the interfaces of I/O devices be addressable as if they were memory locations. This means that program-controlled I/O in a 68000 computer can be achieved as described in the general discussion in Section 2.7.

Assume that bit b_3 of the keyboard status register INSTATUS contains the input control flag SIN. An input operation from the keyboard is accomplished with the instruction sequence

```
READWAIT  BTST.W   #3,INSTATUS
          BEQ      READWAIT
          MOVE.B   DATAIN,D1
```

The bit test instruction, BTST, tests the state of one bit of the destination operand and sets condition code flag Z to be the complement of the bit tested. The position of the bit to be tested, b_3 in our example, is specified by the first operand.

Assuming that bit b_3 in the display status register OUTSTATUS contains the output control flag SOUT, the character in register D1 can be sent to the display by the instruction sequence

```
WRITEWAIT  BTST.W   #3,OUTSTATUS
           BEQ      WRITEWAIT
           MOVE.B   D1,DATAOUT
```

A 68000 program that reads one line of characters from a keyboard, stores them in the memory, and echoes them back to the display is shown in Figure 3.27. This

```
        MOVEA.L   #LOC,A1          Initialize pointer register A1 to contain the
                                   address of the first location in memory
                                   where the characters are to be stored.
READ    BTST.W    #3, INSTATUS     Wait for a character to be entered
        BEQ       READ             in the keyboard buffer DATAIN.
        MOVE.B    DATAIN,(A1)      Transfer the character from DATAIN into
                                   the memory (this clears SIN to 0).
ECHO    BTST.W    #3,OUTSTATUS     Wait for the display to become ready.
        BEQ       ECHO
        MOVE.B    (A1),DATAOUT     Move the character just read to the output
                                   buffer register (this clears SOUT to 0).
        CMPI.B    #CR,(A1)+        Check if the character just read is CR
                                   (carriage return). If it is not CR, then
        BNE       READ             branch back and read another character.
                                   Also, increment the pointer to store the
                                   next character.
```

Figure 3.27 A 68000 program that reads a line of characters and displays it.

program is patterned after the program in Figure 2.20. It assumes that a line ends when the return key is pressed. The characters are stored in memory byte locations starting with location LOC.

3.13 STACKS AND SUBROUTINES

A stack can be implemented, as explained in Section 2.8, using any of the address registers as a pointer. The Autoincrement and Autodecrement addressing modes facilitate this process. One specific register, register A7, is designated as the processor stack pointer, and the stack this register points to is called the *processor stack*. This is the stack used in all stack operations that the processor performs automatically, as in the case of subroutine linkage.

Figure 3.18 shows two different 32-bit registers called A7. The 68000 provides for two different modes of operation, called the user and supervisor modes. Each mode has its own version of the processor stack pointer, A7. In the *supervisor mode*, the processor can execute all machine instructions. In the *user mode*, some instructions, called *privileged* instructions, cannot be executed. Application programs are normally run in the user mode, and the system software uses the supervisor mode. Bit S in the processor status register determines which of the two modes is active, and, hence, which of the two A7 registers is used.

A Branch-to-Subroutine (BSR) instruction is used to call a subroutine. It is implemented in the same way as any other branch instruction, but it also causes the contents of the program counter to be pushed onto the stack. Its branch target is the first instruction in the subroutine. When the subroutine is completed, a Return-from-Subroutine (RTS) instruction is used to return to the calling program. It pops the return address at the top of the stack into the program counter. The BSR and RTS instructions allow the subroutine linkage mechanism, described in general terms in Section 2.9, to be implemented.

Figure 3.28 shows how the program in Figure 3.26 can be written as a subroutine, passing parameters through registers. The list address and the number of entries in the list are passed to the subroutine using registers A2 and D1. After performing the addition, the subroutine returns the sum in register D0.

Figure 3.29 shows how the program in Figure 3.26 can be written as a subroutine, passing parameters on the processor stack pointed to by address register A7. The MOVEM (Move multiple registers) instructions save and restore registers A2, D1, and D0. The order in which these registers are stored on the stack is shown in Figure 3.29*b*. The first MOVEM instruction, which uses the Autodecrement addressing mode, pushes the specified registers onto the stack. The second MOVEM instruction uses the Autoincrement addressing mode to pop the stored values off the stack and store them into the registers in the reverse order.

Consider now the case of nested subroutines, in which one subroutine calls another, as shown in Figure 2.28. Figure 3.30 gives the 68000 code for this example, and the stack frames for subroutines SUB1 and SUB2 are shown in Figure 3.31. The main program calls subroutine SUB1. Before the call instruction BSR is executed, the main program pushes parameters param2 and param1 onto the stack for use by SUB1.

Calling program

```
        MOVEA.L   #NUM1,A2    Put the address NUM1 in A2.
        MOVE.L    N,D1        Put the number of entries n in D1.
        BSR       LISTADD     Call subroutine LISTADD.
        MOVE.L    D0,SUM      Store the sum in SUM.
        next instruction
        .
        .
        .
```

Subroutine

```
LISTADD SUBQ.L    #1,D1       Adjust count to n - 1.
        CLR.L     D0
LOOP    ADD.W     (A2)+,D0    Accumulate sum in D0.
        DBRA      D1,LOOP
        RTS
```

Figure 3.28 Program of Figure 3.26 written as a 68000 subroutine; parameters passed through registers.

The subroutine begins by creating its own frame on the stack. The special instruction:

LINK A*i*,#disp

sets up register A*i* as the frame pointer by performing the following operations:

1. It pushes the contents of register A*i* onto the processor stack.
2. It copies the contents of the processor stack pointer, A7, into register A*i*.
3. It adds the specified displacement value to register A7.

If the displacement value is a negative number, it will cause the top of the stack to move upward (to a lower address location), thus creating an empty space on the stack which the subroutine can use for local variables. These variables can be accessed using indexed addressing with the frame pointer register A*i*. At the end of the subroutine, the UNLK (Unlink) instruction reverses the actions of the LINK instruction. It loads A7 from A*i*, thus lowering the top of the stack to its position before adding the displacement value. Then it pops the original contents of register A*i* off the stack and back into A*i*.

In the example in Figure 3.30, we have assumed that the subroutines can perform their tasks using only registers, so they do not require work space on the stack. Hence, each subroutine begins with the instruction

LINK A6,#0

(Assume top of stack is at level 1 below.)

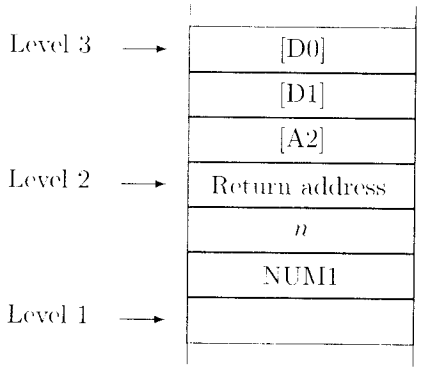
Calling program

```
        MOVE.L    #NUM1.-(A7)      Push parameters onto stack.
        MOVE.L    N.-(A7)
        BSR       LISTADD
        MOVE.L    4(A7).SUM        Save result.
        ADDI.L    #8.A7            Restore top of stack.
        :
```

Subroutine

```
LISTADD MOVEM.L   D0-D1/A2.-(A7)   Save registers D0, D1, and A2.
        MOVE.L    16(A7).D1        Initialize counter to n.
        SUBQ.L    #1.D1            Adjust count to use DBRA.
        MOVEA.L   20(A7).A2        Initialize pointer to the list.
        CLR.L     D0               Initialize sum to 0.
LOOP    ADD.W     (A2)+.D0         Add entry from list.
        DBRA      D1.LOOP
        MOVE.L    D0.20(A7)        Put result on the stack.
        MOVEM.L   (A7)+.D0-D1/A2   Restore registers.
        RTS
```

(a) Calling program and subroutine



(b) Stack contents at different times

Figure 3.29 Program of Figure 3.26 written as a 68000 subroutine; parameters passed on the stack.

Memory location		Instructions		Comments
Calling program				
		⋮		
2000		MOVE.L	PARAM2.–(A7)	Place parameters on stack.
2006		MOVE.L	PARAM1.–(A7)	
2012		BSR	SUB1	
2014		MOVE.L	(A7).RESULT	Store result.
2020		ADDI.L	#8.A7	Restore stack level.
2024		next instruction		
		⋮		
First subroutine				
2100	SUB1	LINK	A6.#0	Set frame pointer.
2104		MOVEM.L	D0–D2/A0.–(A7)	Save registers.
		MOVEA.L	8(A6).A0	Load parameters.
		MOVE.L	12(A6).D0	
		⋮		
		MOVE.L	PARAM3.–(A7)	Place a parameter on stack.
2160		BSR	SUB2	
2164		MOVE.L	(A7)+.D1	Pop result from SUB2 into D1.
		⋮		
		MOVE.L	D2.8(A6)	Place result on stack.
		MOVEM.L	(A7)+.D0–D2/A0	Restore registers.
		UNLK	A6	Restore frame pointer.
		RTS		Return.
Second subroutine				
3000	SUB2	LINK	A6.#0	Set frame pointer.
		MOVEM.L	D0–D1.–(A7)	Save registers.
		MOVE.L	8(A6).D0	Load parameter.
		⋮		
		MOVE.L	D1.8(A6)	Place result on stack.
		MOVEM.L	(A7)+.D0–D1	Restore registers.
		UNLK	A6	Restore frame pointer.
		RTS		Return.

Figure 3.30 Nested subroutines in 68000 assembly language.

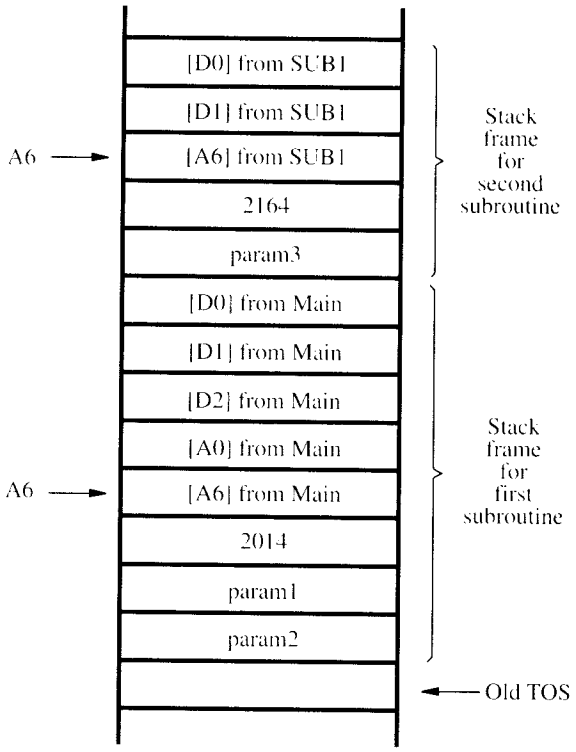


Figure 3.31 68000 stack frames for Figure 3.30.

which defines register A6 as the stack frame pointer and leaves A7 pointing to the location where the old contents of A6 are stored. This instruction performs the same operations as the first two Move instructions at the beginning of the subroutines in Figure 2.28. In each case, it is followed by a MOVEM instruction, which saves the registers needed by the subroutine on the stack.

The remainder of the program in Figure 3.30 is a straightforward implementation of the program in Figure 2.28, using 68000 instructions. As the program in Figure 3.30 is executed, the resulting items are placed on the stack as shown in Figure 3.31. The main program pushes two parameters on the stack, and then the BSR instruction causes the return address, 2014, to be pushed onto the stack. Note that the BSR instruction fits in one word at location 2012 because the offset to SUB1 is small enough to be represented in 8 bits. The LINK and MOVEM instructions in SUB1 save the contents of the frame pointer, A6, and four other registers.

Before subroutine SUB1 calls SUB2, it pushes one parameter, param3, onto the stack. The return address, 2164, is pushed onto the stack by the BSR instruction. The BSR instruction occupies two words because the offset to the target address is larger than can be represented in 8 bits. When each subroutine completes its task, it restores the saved register contents and then returns. After control is returned to the main program,

the result placed on the stack by SUB1 (overwriting param1) is stored in memory location RESULT. Then, the stack pointer A7 is returned to its original value, pointing to the old TOS element in Figure 3.31, by the ADDI.L instruction.

3.14 LOGIC INSTRUCTIONS

In previous sections, we described instructions that move operands and perform arithmetic operations such as addition or subtraction. The operands involved in these instructions have a fixed length of 32, 16, or 8 bits. In some applications it is necessary to manipulate other sizes of data, perhaps only individual bits, and perform logic operations on these data. The 68000 has several instructions for such purposes. In particular, the 68000 has instructions that perform logical AND, OR, and XOR operations as well as instructions that shift and rotate operands in several different ways.

To illustrate the use of logic instructions, let us consider two examples. Suppose that register D1 contains some 32-bit binary pattern, and we want to determine if the pattern in bit positions b_{18} through b_{14} is 11001. This can be done using the instructions

```
AND.L   #$7C000,D1
CMPI.L  #$64000,D1
BEQ     YES
```

The first instruction performs the logical AND of individual bits of the source and destination operands, leaving the result in register D1. The hex number 7C000 has ones in bit positions b_{18} through b_{14} and zeros elsewhere. Thus, as a result of the AND operation, the five bits in positions b_{18} through b_{14} in register D1 retain their original values and the remaining bits are cleared to 0. The subsequent Compare instruction tests whether these five bits correspond to the desired pattern.

Digit-Packing Program

As a second example, consider again the BCD digit-packing program shown in Figure 2.31. The 68000 code for this routine is shown in Figure 3.32. The two ASCII bytes are brought into registers D0 and D1. The LSL instruction shifts the byte in

```
MOVEA.L   #LOC,A0       A0 points to data.
MOVE.B    (A0)+,D0      Load first byte into D0.
LSL.B     #4,D0         Shift left by 4 bit positions.
MOVE.B    (A0),D1       Load second byte into D1.
ANDI.B    #$F,D1        Clear high-order 4 bits to zero.
OR.B      D0,D1         Concatenate the digits.
MOVE.B    D1,PACKED     Store the result.
```

Figure 3.32 Use of 68000 logic instructions in packing BCD digits.

D0 four bit positions to the left, filling the low-order four bits with zeros. The first entry in the operand field of this instruction is a count that indicates the number of bit positions by which the operand is to be shifted. Table C.4 shows that the count may also be specified in another data register. Hence, the same effect can be achieved with

```
LSL.B   D2,D0
```

if the contents of D2 have been set to 4 earlier. The ANDI instruction sets the high-order four bits of the second byte to 0. Finally, the 4-bit patterns that are the desired BCD codes are combined in D1 with the OR instruction and stored in memory location PACKED.

3.15 PROGRAM EXAMPLES

In this section, we give the 68000 version of the programs for dot product, byte sorting, and linked-list operations that were described in Chapter 2.

3.15.1 VECTOR DOT PRODUCT PROGRAM

The program in Figure 2.33 computes the dot product of two vectors, AVEC and BVEC. The 68000 version is shown in Figure 3.33. The two programs are identical except for the use of the DBRA instruction to control the loop. Using this instruction necessitates reducing the contents of the count register D0 by 1, as explained in Section 3.11.2.

Note that the MULS instruction multiplies two signed 16-bit numbers and produces a 32-bit product. We have assumed that the vector elements are represented in 16-bit words and that the dot product fits in 16 bits. All addresses are treated as 32-bit values.

```
        MOVEA.L   #AVEC,A1      Address of first vector.
        MOVEA.L   #BVEC,A2      Address of second vector.
        MOVE      N,D0          Number of elements.
        SUBQ      #1,D0         Adjust count to use DBRA.
        CLR       D1            Use D1 as accumulator.
LOOP    MOVE      (A1)+,D2      Get element from vector A.
        MULS      (A2)+,D2      Multiply element from vector B.
        ADD       D2,D1         Accumulate product.
        DBRA      D0,LOOP
        MOVE      D1,DOTPROD
```

Figure 3.33 A 68000 program for computing the dot product of two vectors.

3.15.2 BYTE-SORTING PROGRAM

Now let us consider the program to sort a list of bytes given in Figure 2.34. This program performs an in-place sort using the straight selection algorithm to put a list of characters in alphabetic order. The list is stored in memory locations LIST through LIST + n − 1, with each character represented in the ASCII code and occupying one byte. The value n is a 16-bit value stored at address N.

The C-language program for this task is reproduced in Figure 3.34*a*. The 68000 implementation of this task is given in Figure 3.34*b*. This program closely parallels the program in Figure 2.34*b*. The programs differ in some minor respects, as follows.

```
for   (j = n-1; j > 0; j = j - 1)
      { for ( k = j-1; k >= 0; k = k - 1 )
          { if   (LIST[k] > LIST[j])
                 {  TEMP = LIST[k];
                    LIST[k] = LIST[j];
                    LIST[j] = TEMP;
                 }
          }

      }
```

(a) C-language program for sorting

```
        MOVEA.L   #LIST.A1      Pointer to the start of the list.
        MOVE      N.D1          Initialize outer loop
        SUBQ      #1.D1           index j in D1.
OUTER   MOVE      D1.D2         Initialize inner loop
        SUBQ      #1.D2           index k in D2.
        MOVE.B    (A1.D1).D3    Current maximum value in D3.
INNER   CMP.B     D3.(A1.D2)    If LIST(k) ≤ [D3],
        BLE       NEXT          do not exchange.
        MOVE.B    (A1.D2).(A1.D1)  Interchange LIST(k)
        MOVE.B    D3.(A1.D2)        and LIST(j) and load
        MOVE.B    (A1.D1).D3        new maximum into D3.
NEXT    DBRA      D2.INNER      Decrement counters k and j
        SUBQ      #1.D1           and branch back
        BGT       OUTER           if not finished.
```

(b) 68000 program implementation

Figure 3.34 A 68000 byte-sorting program.

The MOVE instruction of the 68000 allows both the source and the destination operands to be in the memory. Hence, when interchanging two entries, the value of LIST(k) is copied directly into LIST(j). This eliminates the need for the temporary register R4 used in Figure 2.34 and leads to a slight reorganization of the program instructions.

Another difference is that we have used the DBRA instruction to terminate the inner loop in the program because the index k runs down to 0. Note that it is not possible to use the DBRA instruction in the outer loop because the final value for j is 1 rather than 0.

3.15.3 LINKED-LIST INSERTION AND DELETION SUBROUTINES

Figure 3.35 gives a 68000 subroutine to insert a record in linked list. This program is identical to the program in Figure 2.37. Note that the CMPA version of the Compare instruction is used to compare address values, and the CMP version is used for data values.

A 68000 program to delete a record from a linked list is given in Figure 3.36. This program corresponds directly to the program given in Figure 2.38.

```
Subroutine
INSERTION   CMPA.L   #0,A0        A0 is RHEAD.
            BGT      HEAD
            MOVEA.L  A1,A0        A1 is RNEWREC.
            RTS
HEAD        CMP.L    (A0),(A1)    Compare ID of new record to head.
            BGT      SEARCH
            MOVE.L   A0,4(A1)     New record becomes head.
            MOVEA.L  A1,A0
            RTS
SEARCH      MOVEA.L  A0,A2        A2 is RCURRENT.
LOOP        MOVEA.L  4(A2),A3     A3 is RNEXT.
            CMPA.L   #0,A3
            BEQ      TAIL
            CMP.L    (A3),(A1)
            BLT      INSERT
            MOVEA.L  A3,A2        Go to next record.
            BRA      LOOP
INSERT      MOVE.L   A2,4(A1)
TAIL        MOVE.L   A1,4(A2)
            RTS
```

Figure 3.35 A 68000 subroutine to insert a record in a linked list.

Subroutine

```
DELETION  CMP.L    (A0),D1     D1 is RIDNUM.
          BGT      SEARCH
          MOVEA.L  4(A0),A0    Delete head record.
          RTS
SEARCH    MOVEA.L  A0,A2       A2 is RCURRENT.
LOOP      MOVEA.L  4(A2),A3    A3 is RNEXT.
          CMP.L    (A3),D1
          BEQ      DELETE
          MOVEA.L  A3,A2
          BRA      LOOP
DELETE    MOVE.L   4(A3),D2    D2 is RTEMP.
          MOVE.L   D2,4(A2)
          RTS
```

Figure 3.36 A 68000 subroutine to delete a record from a linked list.

As with the generic subroutines in Figures 2.37 and 2.38, the insertion subroutine in Figure 3.35 assumes that the ID number of the new record does not match that of any record already in the list, and the deletion subroutine in Figure 3.36 assumes that there exists a record in the list with an ID number that does match the contents of RIDNUM. Problems 3.49 and 3.50 consider how the subroutines should be changed to signal an error if the assumptions are not true.

PART III
THE IA-32 PENTIUM EXAMPLE

The Intel Corporation uses the generic name Intel Architecture (IA) for processors in its product line. We will describe IA processors that operate with 32-bit memory addresses and 32-bit data operands. They are referred to as IA-32 processors, and the most recent is the Pentium series. The first IA-32 processor was the 80386, introduced in 1985. Since then, the 80486 (1989), Pentium (1993), Pentium Pro (1995), Pentium II (1997), Pentium III (1999), and Pentium 4 (2000) have been implemented. These processors have increasing levels of performance, achieved through a number of architectural and microelectronic technology improvements. The evolution of the IA family will be explained in Chapter 11. The latest members of the family have specialized instructions for handling multimedia graphical information and for vector data processing. These aspects of the instruction set will be considered briefly here and also in Chapter 11. The IA-32 instruction set is very large. We will restrict our attention to the basic instructions and addressing modes. Detailed information on the IA-32 instruction set architecture and assembly language can be found at the Intel web site [8] and in the books by Brey [9], Dandamudi [10], and Tabak [7].

3.16 REGISTERS AND ADDRESSING

In the IA-32 architecture, memory is byte addressable using 32-bit addresses, and instructions operate on data operands of 8 or 32 bits. These operand sizes are called byte and doubleword in Intel terminology. A 16-bit operand was called a word in earlier 16-bit Intel processors. Little-endian addressing is used, as described in Section 2.2.2. Multiple-byte data operands may start at any byte address location. They need not be aligned with any particular address boundaries in the memory.

3.16.1 IA-32 REGISTER STRUCTURE

The processor registers are shown in Figure 3.37. While there are some exceptions, the eight 32-bit registers labeled R0 through R7 are general-purpose registers that can be used to hold either data operands or addressing information. There are eight floating-point registers for holding doubleword or quadword (64 bits) floating-point data operands. The floating-point registers have an extension field to provide a total length of 80 bits, not shown in Figure 3.37. The extra bits are used for increased accuracy while floating-point numbers are operated on in the processor. Chapter 6 provides a discussion of floating-point number representation and operations. This topic is not discussed here in Chapter 3.

IA-32 architectures are based on a memory model that associates different areas of the memory, called *segments*, with different usages. The *code segment* holds the instructions of a program. The *stack segment* contains the processor stack, and four *data segments* are provided for holding data operands. The six segment registers shown in Figure 3.37 contain selector values that are used in locating these segments in the memory address space. The detailed function of these registers will be explained in Chapter 11 where we discuss the IA family. Here, we will not need to know these details. A 32-bit address in the IA-32 architecture can be presumed to access memory locations anywhere in the program, processor stack, or data areas.

The two registers shown at the bottom of Figure 3.37 are the Instruction Pointer, which serves as the program counter and contains the address of the next instruction to be executed, and the Status Register, which holds the condition code flags (CF, ZF, SF, OF). These flags contain information about the results of arithmetic operations, as will be discussed in Section 3.19. The program execution mode bits (IOPL, IF, TF) are associated with input/output operations and interrupts, discussed in Chapter 4.

The IA-32 general-purpose registers allow for compatibility with the registers of earlier 8-bit and 16-bit Intel processors. In those processors, some restrictions applied to the specific usage of the different registers in programs. Figure 3.38 shows the association between the IA-32 registers and the registers in earlier processors. The eight general-purpose registers are grouped into three different types: data registers for holding operands, and pointer and index registers for holding addresses and address indices used to determine the effective address of a memory operand.

In Intel's original 8-bit processors, the data registers were called A, B, C, and D. In later 16-bit processors, these registers were labeled AX, BX, CX, and DX. The high- and low-order bytes in each register are identified by suffixes H and L. For example,

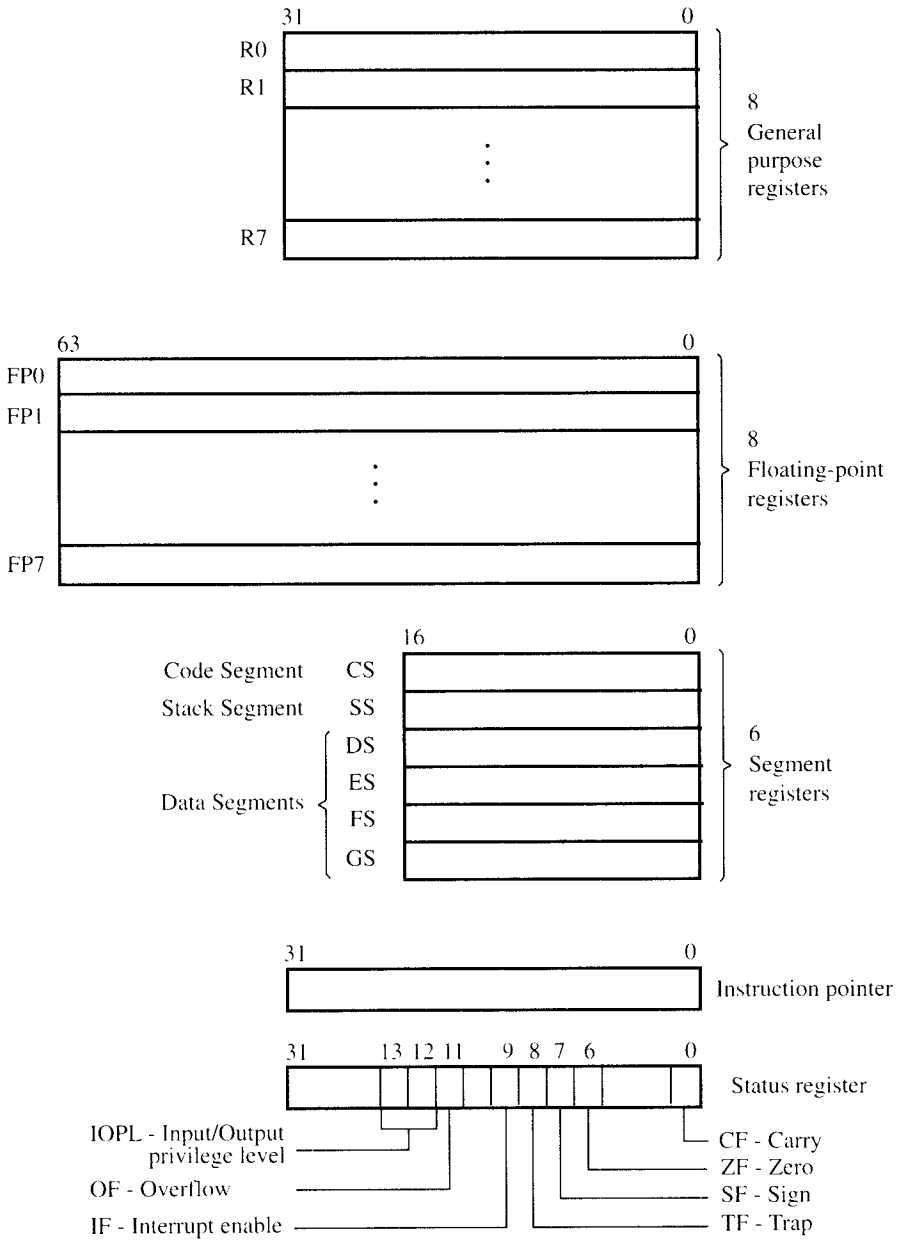


Figure 3.37 IA-32 register structure.

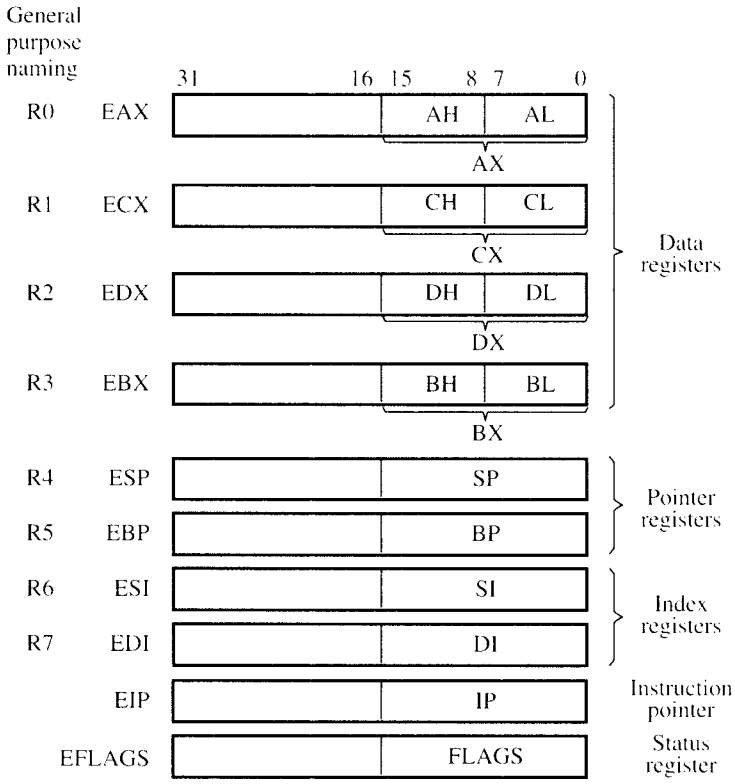


Figure 3.38 Compatibility of the IA-32 register structure with earlier Intel processor register structures.

the two bytes in register AX are referred to as AH and AL. In IA-32 processors, the prefix E is used to identify the corresponding "extended" 32-bit registers: EAX, EBX, ECX, and EDX. The E-prefix labeling is also used for the other 32-bit registers shown in Figure 3.38. They are the extended versions of the corresponding 16-bit registers used in earlier processors.

This register labeling is used in Intel technical documents and in other descriptions of Intel processors. The reason that the historical labeling has been retained is that Intel has maintained upward compatibility over its processor line. That is, programs in machine language representation developed for the earlier 16-bit processors will run correctly on current IA-32 processors without change if the processor state is set to do so. We will use the E-prefix register labeling in giving examples of assembly language programs because these mnemonics are used in current versions of the assembly language for IA-32 processors. The AL, BL, etc. labeling will also be used for byte operands when they are operated on in the low-order eight bits of the corresponding 32-bit register.

The IA-32 processor state can be switched dynamically between 32-bit operation and 16-bit operation during program execution on an instruction by instruction basis by the use of instruction prefix bytes. We will discuss this feature in Chapter 11.

3.16.2 IA-32 ADDRESSING MODES

The IA-32 architecture has a large and flexible set of addressing modes. They are designed to access individual data items or data items that are members of an ordered list that begins at a specified memory address. We give full definitions for these modes, along with the way that they are expressed in assembly language.

The basic modes, which are available in most processors, have been described in Section 2.5. They are: Immediate, Absolute, Register, and Register indirect. Intel uses the term Direct for the Absolute mode, so we will do the same here. There are also several addressing modes that provide more flexibility in accessing data operands in the memory. The most flexible mode described in Section 2.5 is the Index mode that has the general notation X(Ri, Rj). The effective address of the operand, EA, is calculated as

$$EA = [Ri] + [Rj] + X$$

where Ri and Rj are general-purpose registers and X is a constant. The registers Ri and Rj are called *base* and *index* registers, respectively, and the constant X is called a *displacement*. The IA-32 addressing modes include this mode and simpler variations of it.

The full set of IA-32 addressing modes is defined as follows:

Immediate mode — The operand is contained in the instruction. It is a signed 8-bit or 32-bit number, with the length being specified by a bit in the OP code of the instruction. This bit is 0 for the short version and 1 for the long version.

Direct mode — The memory address of the operand is given by a 32-bit value in the instruction.

Register mode — The operand is contained in one of the eight general-purpose registers specified in the instruction.

Register indirect mode — The memory address of the operand is contained in one of the eight general-purpose registers specified in the instruction.

Base with displacement mode — An 8-bit or 32-bit signed displacement and one of the eight general-purpose registers to be used as a base register are specified in the instruction. The effective address of the operand is the sum of the contents of the base register and the displacement.

Index with displacement mode — A 32-bit signed displacement, one of the eight general-purpose registers to be used as an index register, and a scale factor of 1, 2, 4, or 8, are specified in the instruction. To obtain the effective address of the operand, the contents of the index register are multiplied by the scale factor and then added to the displacement.

Base with index mode — Two of the eight general-purpose registers and a scale factor of 1, 2, 4, or 8, are specified in the instruction. The registers are used as base and index registers, and the effective address of the operand is calculated as follows. The contents of the index register are multiplied by the scale factor and added to the contents of the base register.

Base with index and displacement mode — An 8-bit or 32-bit signed displacement, two of the eight general-purpose registers, and a scale factor of 1, 2, 4, or 8, are specified in the instruction. The registers are used as base and index registers,

and the effective address of the operand is calculated as follows. The contents of the index register are multiplied by the scale factor and then added to the contents of the base register and the displacement.

The IA-32 addressing modes and the way that they are expressed in assembly language are given in Table 3.3. The calculation of the effective address of the operand is also specified in the table. As specified in the footnotes, register ESP cannot be used as an index register. As we will see later, it is used as the processor stack pointer. We will now give some examples of how the addressing modes are used to access operands.

In two-operand instructions, the source (src) and destination (dst) operands are specified in the assembly language in the order

OPcode dst,src

This ordering is the same as in the ARM architecture described in Part I of this chapter, but it is opposite to that used in Chapter 2 and in the Motorola 68000 processor described in Part II of this chapter. For example, the Move instruction

MOV dst,src

performs the operation

dst ← [src]

Table 3.3 IA-32 addressing modes

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] × S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value == an 8- or 32-bit signed number
Location == a 32-bit address
Reg, Reg1, Reg2 == one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register
Disp == an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.
S == a scale factor of 1, 2, 4, or 8

It is convenient to use the Move instruction to illustrate the IA-32 addressing modes. The instruction

```
MOV  EAX,25
```

uses the Immediate addressing mode to move the decimal value 25 into the EAX register. A number given in this form using the digits 0 through 9 is assumed to be in decimal notation. The suffixes B and H are used to specify binary and hexadecimal numbers, respectively. For example, the instruction

```
MOV  EAX,3FA00H
```

moves the hex number 3FA00 into EAX.

The instruction

```
MOV  EAX,LOCATION
```

uses the Direct addressing mode to move the doubleword at the memory location specified by the address label LOCATION into register EAX. This assumes that LOCATION has been defined as an address label for a memory location in the data declaration section of the assembly language program. We will see how to do this in Section 3.18. If LOCATION represents the address 1000, then this instruction moves the doubleword at 1000 into EAX.

The distinction between the Immediate addressing mode and the Direct addressing mode in IA-32 assembly language programs warrants some discussion because it can be somewhat confusing. Consider the following case: It is sometimes useful to give symbolic names to numerical constants that are used as immediate operands. The assembler directive

```
NUMBER  EQU  25
```

is used to associate the symbolic name NUMBER with the decimal value 25, as described in Section 2.6.1. If this is done, the instruction

```
MOV  EAX,NUMBER
```

is interpreted by the assembler to mean that NUMBER is an immediate operand to be moved into register EAX. On the other hand, if NUMBER is defined as an address label, this instruction is interpreted as using the Direct addressing mode.

In many assembly languages, this potentially confusing situation is avoided by using a special symbol, such as #, as a prefix to denote the Immediate addressing mode. In the IA-32 assembly language, square brackets can be used to explicitly indicate the Direct addressing mode, as in the instruction

```
MOV  EAX,[LOCATION]
```

However, the brackets are not needed if LOCATION has been defined as an address label.

When it is necessary to treat an address label as an immediate operand, the assembler directive OFFSET is used. For example, the instruction

```
MOV  EBX, OFFSET LOCATION
```

moves the value of the address label LOCATION, for example 1000, using the Immediate addressing mode, into the EBX register. The EBX register can then be used in the Register indirect mode in the instruction

MOV EAX,[EBX]

to move the contents of the memory location whose address, LOCATION, is contained in register EBX into register EAX. The word OFFSET as an assembler directive is chosen to indicate that an address is always considered as a relative distance away from the starting point of the memory segment containing the location. In all of these examples, the Register mode has been used to specify the destination.

These examples have illustrated the basic addressing modes: Immediate, Direct, Register, and Register indirect. The remaining four addressing modes provide more flexibility in accessing data operands in the memory.

The Base with displacement mode is illustrated in Figure 3.39*a*. Register EBP is used as the base register. A doubleword operand that is 60 byte locations away from the base address of 1000, that is, at address 1060, can be moved into register EAX by the instruction

MOV EAX,[EBP + 60]

The IA-32 instructions and addressing modes can be used to operate on byte operands as well as doubleword operands. For example, still assuming that the base register EBP contains the address 1000, the byte operand at address 1010 can be loaded into the low-order byte position in the EAX register by the instruction

MOV AL,[EBP + 10]

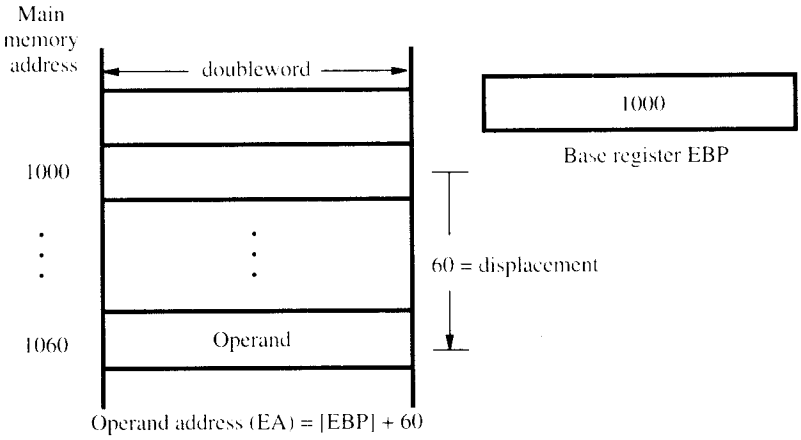
The assembler selects the version of the Move OP code that is used for byte data because the destination, AL, is the low-order byte position of the EAX register.

The addressing mode that provides the most flexibility is the Base with index and displacement mode. An example is shown in Figure 3.39*b*, using EBP and ESI as the base and index registers. This example shows how the mode is used to access a particular doubleword operand in a list of doubleword operands. The list begins at a displacement of 200 away from the base address 1000. Using a scale factor of 4 on the index register contents, successive doubleword operands at addresses 1200, 1204, 1208, ... can be accessed by using the sequence of indices 0, 1, 2, ... in the index register ESI. In the example shown in the figure, the doubleword at address 1360 (that is, $1000 + 200 + 4 \times 40$) is accessed when the index register contains 40. This operand can be loaded into register EAX by the instruction

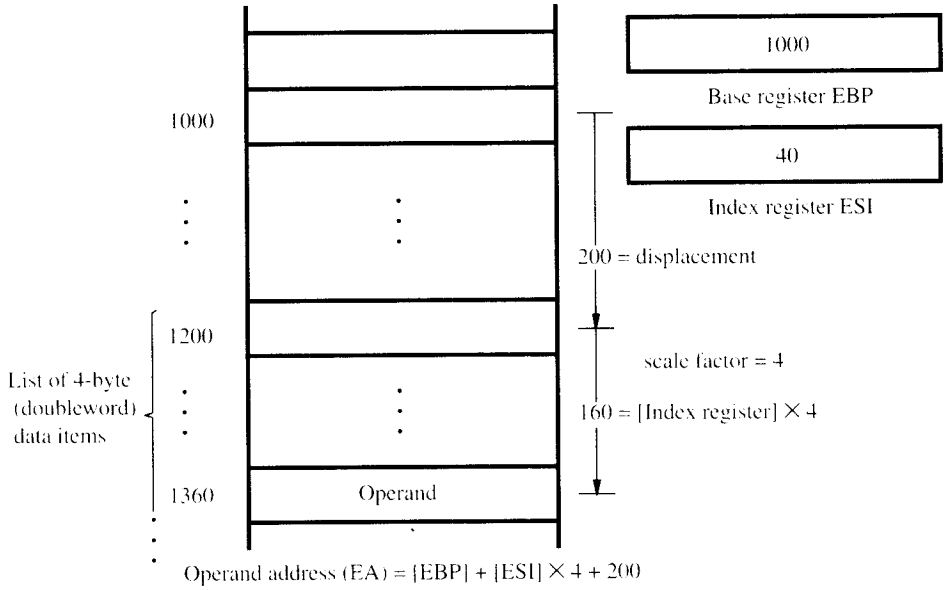
MOV EAX,[EBP + ESI*4 + 200]

The use of a scale factor of 4 in this addressing mode makes it easy to access successive doubleword operands of the list in a program loop by simply incrementing the index register by 1 on each pass through the loop. Having discussed these two modes in some detail, the closely related Index with displacement mode and Base with index mode should be easy to understand.

Before leaving this discussion of addressing modes, a comment on an aspect of the Table 3.3 entries is useful. It would appear that the Base with displacement mode is



(a) Base with displacement mode, expressed as [EBP + 60]



(b) Base with displacement and index mode, expressed as [EBP + ESI * 4 + 200]

Figure 3.39 Examples of addressing modes in the IA-32 architecture.

redundant because the same effect could be obtained by using the Index with displacement mode with a scale factor of 1. But the former mode is useful because it is encoded with one less byte. In addition, the displacement size in the Index with displacement mode can only be 32 bits.

A discussion of how the addressing modes are encoded into machine instructions is included in the next section. More detail is provided in Appendix D.

3.17 IA-32 INSTRUCTIONS

The IA-32 instruction set is extensive. It is encoded in a variable-length instruction format that does not have a fully regular layout. We will examine the instruction format in Section 3.17.1. Most of the IA-32 instructions have either one or two operands. In the two-operand case, only one of the operands can be in the memory. The other must be in a processor register. In addition to the usual instructions for moving data between the memory and the processor registers, and for performing arithmetic operations, the instruction set includes a number of different logical and shift/rotate operations on data. Byte string instructions are included for nonnumeric data processing. Push and pop operations for manipulating the processor stack are directly supported in the instruction set.

We will begin by introducing a small set of instructions and show how they can be used in a simple complete program. The instruction

ADD dst,src

performs the operation

$$dst \leftarrow [dst] + [src]$$

and, as we have seen earlier,

MOV dst,src

performs the operation

$$dst \leftarrow [src]$$

Suppose that two data operands are in registers EAX and EBX. Their sum can be computed in EAX and stored in the memory at location SUM by the two-instruction sequence

```
ADD   EAX,EBX
MOV   SUM,EAX
```

Since only one operand can be in memory in the two-operand instructions, the operation

$$C \leftarrow [A] + [B]$$

involving three memory locations A, B, and C, can be performed using the instruction