CHAPTER

9

Instruction Set Architecture

Up to this point, much of what we have studied has focused on digital system design, with computer components serving as examples. In this chapter, we will study more specialized material, dealing with instruction set architecture for general-purpose computers. We will examine the operations that the instructions perform and focus particularly on how the operands are obtained and where the results are stored. We will contrast two distinct classes of architectures: reduced instruction set computers (RISCs) and complex instruction set computers (CISCs). We will classify elementary instructions into three categories: data transfer, data manipulation, and program control. In each of these categories, we elaborate on typical elementary instructions.

Central to the material presented here are the general-purpose parts of the generic computer at the beginning of Chapter 1, including the central processing unit (CPU) and the accompanying floating-point unit (FPU). Since a small general-purpose microprocessor may be present for controlling keyboard and monitor functions, these components are also involved. Aside from addressing used to access memory and I/O components, the concepts studied apply less to other areas of the computer. Increasingly, however, small CPUs have appeared more frequently in the I/O components.

9-1 Computer Architecture Concepts

The binary language in which instructions are defined and stored in memory is referred to as *machine language*. A symbolic language that replaces binary opcodes and addresses with symbolic names and that provides other features helpful to the programmer is referred to as *assembly language*. The logical structure of computers

is normally described in assembly-language reference manuals. Such manuals explain various internal elements of the computer that are of interest to the programmer, such as processor registers. The manuals list all hardware-implemented instructions, specify the symbolic names and binary code format of the instructions, and provide a precise definition of each instruction. In the past, this information represented the *architecture* of the computer. A computer was composed of its architecture, plus a specific *implementation* of that architecture. The implementation was separated into two parts: the organization and the hardware. The *organization* consists of structures such as datapaths, control units, memories, and the buses that interconnect them. *Hardware* refers to the logic, the electronic technologies employed, and the various physical design aspects of the computer.

As computer designers pushed for higher and higher performance, and as increasingly more of the computer resided within a single IC, the relationships among architecture, organization, and hardware became so intertwined that a more integrated viewpoint became necessary. According to this new viewpoint, architecture as previously defined is more restrictively called *instruction set architecture* (*ISA*), the structure of a particular hardware implementation of the ISA is referred to as the *microarchitecture* or *computer organization*, and the term *architecture* is used to encompass the whole of the computer, including instruction set architecture, organization, and hardware. This unified view enables intelligent design trade-offs to be made that are apparent only in a tightly coupled design process. These trade-offs have the potential for producing better computer designs.

In this chapter, we focus on instruction set architecture. In the next, we will look at two distinct instruction set architectures, with a focus on implementation using two somewhat different architectures.

A computer usually has a variety of instructions and multiple instruction formats. It is the function of the control unit to decode each instruction and provide the control signals needed to process it. Simple examples of instructions and instruction formats were presented in Section 8-7. We now expand this presentation by introducing typical instructions found in commercial general-purpose computers. We also investigate the various instruction formats that may be encountered in a typical computer, with an emphasis on the addressing of operands. The format of an instruction is depicted in a rectangular box symbolizing the bits of the binary instruction. The bits are divided into groups called *fields*. The following are typical fields found in instruction formats:

1. An *opcode field*, which specifies the operation to be performed.
2. An *address field*, which provides either a memory address or an address that selects a processor register.
3. A *mode field*, which specifies the way the address field is to be interpreted.

Other special fields are sometimes employed under certain circumstances—for example, a field that gives the number of positions to shift in a shift-type instruction or an operand field in an immediate operand instruction.

Basic Computer Operation Cycle

In order to comprehend the various addressing concepts to be presented in the next two sections, we need to understand the basic operation cycle of the computer. The computer's control unit is designed to execute each instruction of a program in the following sequence of steps:

1. Fetch the instruction from memory into the instruction register in the control unit.
2. Decode the instruction.
3. Locate the operands used by the instruction.
4. Fetch operands from memory (if necessary).
5. Execute the operation in processor registers.
6. Store the results in the proper place.
7. Go back to Step 1 to fetch the next instruction.

As explained in Section 8-7, a register in the computer called the *program counter* (*PC*) keeps track of the instructions in the program stored in memory. The *PC* holds the address of the instruction to be executed next and is incremented each time a word is read from the program in memory. The decoding done in Step 2 determines the operation to be performed and the addressing mode or modes of the instruction. The operands in Step 3 are located from the addressing modes and the address fields of the instruction. The computer executes the instruction, storing the result, and returns to Step 1 to fetch the next instruction in sequence.

Register Set

The *register set* consists of all registers in the CPU that are accessible to the programmer. These registers are typically those mentioned in assembly-language programming reference manuals. In the simple CPUs we have dealt with so far, the register set has consisted of the programmer-accessible portion of the register file and the *PC*. The CPUs can also contain other registers, such as the instruction register, registers in the register file that are accessible only to hardware controls and/or microprograms, and pipeline registers. These registers, however, are not directly accessible to the programmer and, as a consequence, are not a part of the register set, which represents the stored information in the CPU that is visible to the programmer through the instructions. Thus, the register set has a considerable influence on instruction set architecture.

The register set for a realistic CPU is quite complex. In this chapter, we add two registers to the set we have used thus far: the *processor status register* (*PSR*) and the *stack pointer* (*SP*). The processor status register contains flip-flops that are selectively set by status values *C*, *N*, *V*, and *Z* from the ALU and shifter. These stored status bits are used to make decisions that determine the program flow, based on ALU results, shifter results, or the contents of registers. The stored status bits in the

processor status register are also referred to as the *condition codes* or the *flags*. Additional bits in the *PSR* will be discussed when we cover associated concepts in this chapter.

9-2 Operand Addressing

Consider an instruction such as ADD, which specifies the addition of two operands to produce a result. Suppose that the result of the addition is treated as just another operand. Then the ADD instruction has three operands: the addend, the augend, and the result. An operand residing in memory is specified by its address. An operand residing in a processor register is specified by a register address, a binary code of n bits that specifies one of at most 2^n registers in the register file. Thus, a computer with 16 processor registers, say, $R0$ through $R15$, has in its instructions one or more register address fields of four bits. The binary code 0101, for example, designates register $R5$.

Some operands, however, are not explicitly addressed, because their location is specified either by the opcode of the instruction or by an address assigned to one of the other operands. In such a case, we say that the operand has an *implied address*. If the address is implied, then there is no need for a memory or register address field for the operand in the instruction. On the other hand, if an operand has an address in the instruction, then we say that the operand is explicitly addressed or has an *explicit address*.

The number of operands explicitly addressed for a data-manipulation operation such as ADD is an important factor in defining the instruction set architecture for a computer. An additional factor is the number of such operands that can be explicitly addressed in memory by the instruction. These two factors are so important in defining the nature of instructions that they act a means of distinguishing different instruction set architectures. They also govern the length of computer instructions.

We begin by illustrating simple programs with different numbers of explicitly addressed operands per instruction. Since the explicitly addressed operands have up to three memory or register addresses per instruction, we label the instructions as having three, two, one, or zero addresses. Note that, of the three operands needed for an instruction such as ADD, the addresses of all operands not having an address in the instruction are implied.

To illustrate the influence of the number of operands on computer programs, we will evaluate the arithmetic statement

$$X = (A + B)(C + D)$$

using three, two, one, and zero address instructions. We assume that the operands are in memory addresses symbolized by the letters A, B, C, and D and must not be changed by the program. The result is to be stored in memory at a location with address X. The initial arithmetic operations to be used in the instructions are addition, subtraction, and multiplication, with mnemonics ADD, SUB, and MUL, respectively. Further, three operations needed to transfer data during the evaluation are move, load, and store, denoted by MOVE, LD, and ST, respectively. LD

moves an operand from memory to a register and ST from a register to memory. Depending on the addresses permitted, MOVE can transfer data between registers, between memory locations, or from memory to register or register to memory.

Three-Address Instructions

A program that evaluates $X = (A + B)(C + D)$ using three-address instructions is as follows (a register transfer statement is shown for each instruction):

ADD T1, A, B	$M[T1] \leftarrow M[A] + M[B]$
ADD T2, C, D	$M[T2] \leftarrow M[C] + M[D]$
MUL X, T1, T2	$M[X] \leftarrow M[T1] \times M[T2]$

The symbol $M[A]$ denotes the operand stored in memory at the address symbolized by A. The symbol $\times$ designates multiplication. T1 and T2 are temporary storage locations in memory.

This same program can use registers as the temporary storage locations:

ADD R1, A, B	$R1 \leftarrow M[A] + M[B]$
ADD R2, C, D	$R2 \leftarrow M[C] + M[D]$
MUL X, R1, R2	$M[X] \leftarrow R1 \times R2$

Use of registers reduces the data memory accesses required from nine to five. An advantage of the three-address format is that it results in short programs for evaluating expressions. A disadvantage is that the binary-coded instructions require more bits to specify three addresses, particularly if they are memory addresses.

Two-Address Instructions

For two-address instructions, each address field can again specify either a possible register or a memory address. The first operand address listed in the symbolic instruction also serves as the implied address to which the result of the operation is transferred. The program is as follows:

MOVE T1, A	$M[T1] \leftarrow M[A]$
ADD T1, B	$M[T1] \leftarrow M[T1] + M[B]$
MOVE X, C	$M[X] \leftarrow M[C]$
ADD X, D	$M[X] \leftarrow M[X] + M[D]$
MUL X, T1	$M[X] \leftarrow M[X] \times M[T1]$

If a temporary storage register R1 is available, it can replace T1. Note that this program takes five instructions instead of the three used by the three-address instruction program.

One-Address Instructions

To perform instructions such as ADD, a computer with one-address instructions uses an implied address—such as a register called an *accumulator*, *ACC*—for obtaining one of the operands and as the location of the result. The program to evaluate the arithmetic statement is as follows:

LD	A	$ACC \leftarrow M[A]$
ADD	B	$ACC \leftarrow ACC + M[B]$
ST	X	$M[X] \leftarrow ACC$
LD	C	$ACC \leftarrow M[C]$
ADD	D	$ACC \leftarrow ACC + M[D]$
MUL	X	$ACC \leftarrow ACC \times M[X]$
ST	X	$M[X] \leftarrow ACC$

All operations are done between the *ACC* register and a memory operand. In this case, the number of instructions in the program has increased to seven and the number of memory data accesses is also seven.

Zero-Address Instructions

To perform an ADD instruction with zero addresses, all three addresses in the instruction must be implied. A conventional way of achieving this goal is to use a *stack*, which is a mechanism or structure that stores information such that the item stored last is the first retrieved. Because of its "last-in, first-out" nature, a stack is also called a *last-in, first-out* (*LIFO*) queue. The operation of a computer stack is analogous to that of a stack of trays or plates, in which the last tray placed on top of the stack is the first to be taken off. Data-manipulation operations such as ADD are performed on the stack. The word at the top of the stack is referred to as TOS. The word below it is TOS_{-1}. When one or more words are used as operands for an operation, they are removed from the stack. The word below them then becomes the new TOS. When a resulting word is produced, it is placed on the stack and becomes the new TOS. Thus, TOS and a few locations below it are the implied addresses for operands, and TOS is the implied address for the result. For example, the instruction that specifies an addition is simply

ADD

The resulting register transfer action is $TOS \leftarrow TOS + TOS_{-1}$. Thus, there are no registers or register addresses used for data-manipulation instructions in a stack architecture. Memory addressing, however, is used in such architectures for data transfers. For instance, the instruction

PUSH X

results in $TOS \leftarrow M[X]$, a transfer of the word in address X in memory to the top of the stack. A corresponding operation,

POP X

results in $M[X] \leftarrow TOS$, a transfer of the entry at the top of the stack to address X in memory.

The program for evaluating the sample arithmetic statement for the zero-address situation is as follows:

PUSH	A	$TOS \leftarrow M[A]$
PUSH	B	$TOS \leftarrow M[B]$
ADD		$TOS \leftarrow TOS + TOS_{-1}$
PUSH	C	$TOS \leftarrow M[C]$
PUSH	D	$TOS \leftarrow M[D]$
ADD		$TOS \leftarrow TOS+TOS_{-1}$
MUL		$TOS \leftarrow TOS \times TOS_{-1}$
POP	X	$M[X] \leftarrow TOS$

This program requires eight instructions—one more than the number required by the previous one-address program. However, it uses addressed memory locations or registers only for PUSH and POP and not to execute data-manipulation instructions involving ADD and MUL. Note that memory data accesses may be necessary, however, depending upon the stack implementation. Often, stacks utilize a fixed number of registers near the top of the stack. If a given program can be executed only within these stack locations, memory data accesses are necessary for fetching the initial operands and storing the final result only. But, if the program requires more temporary, intermediate storage, additional data accesses to memory are required.

Addressing Architectures

The programs just presented change if the number of addresses to the memory in the instructions is restricted or if the memory addresses are restricted to specific instructions. These restrictions, combined with the number of operands addressed, define addressing architectures. We can illustrate such architectures with the evaluation of an arithmetic statement in a three-address architecture that has all of the accesses to memory. Such an addressing scheme is called a *memory-to-memory architecture*. This architecture has only control registers, such as the program counter in the CPU. All operands come directly from memory, and all results are sent directly to memory. The formats of data transfer and manipulation instructions contain from one to three address fields, all of which are used for memory addresses. For the previous example, three instructions are required, but if an extra word must appear in the instruction for each memory address, then up to four memory reads are required to

fetch each instruction. Including the fetching of operands and storing of results, the program to perform the arithmetic operation would require 21 accesses to memory. If memory accesses take more than one clock cycle, the execution time would be in excess of 21 clock periods. Thus, even though the instruction count is low, the execution time is potentially high. Also, providing the capability for all operations to access memory increases the complexity of the control structures and may lengthen the clock cycle. Thus, this memory-to-memory architecture is typically not used in new designs.

In contrast, the three-address *register-to-register* or *load/store architecture*, which allows only one memory address and restricts its use to load and store types of instructions, is typical in modern processors. Such an architecture requires a sizeable register file, since all data manipulation instructions use register operands. With this architecture, the program to evaluate the sample arithmetic statement is as follows:

LD	R1, A	$R1 \leftarrow M[A]$
LD	R2, B	$R2 \leftarrow M[B]$
ADD	R3, R1, R2	$R3 \leftarrow R1 + R2$
LD	R1, C	$R1 \leftarrow M[C]$
LD	R2, D	$R2 \leftarrow M[D]$
ADD	R1, R1, R2	$R1 \leftarrow R1 + R2$
MUL	R1, R1, R3	$R1 \leftarrow R1 \times R3$
ST	X, R1	$M[X] \leftarrow R1$

Note that the instruction count increases to eight compared to three for the three-address, memory-to-memory case. Note also that the operations are the same as those for the stack case, except for the need for register addresses. By using registers, the number of accesses to memory for instructions, addresses, and operands is reduced from 21 to 18. If addresses can be obtained from registers instead of memory, as discussed in the next section, this number can be further reduced.

Variations on the previous two addressing architectures include three-address instructions and two-address instructions with one or two of the addresses to memory. The program lengths and number of memory accesses tend to be intermediate between the previous two architectures. An example of a two-address instruction with a single memory address allowed is

ADD	R1,A	$R1 \leftarrow R1 + M[A]$

This *register-memory* type of architecture remains common among the current instruction set architectures, primarily to provide compatibility with older software using a specific architecture.

The program with one-address instructions illustrated previously gives the *single-accumulator architecture*. Since this architecture has no register file, its single address is for accessing memory. It requires 21 accesses to memory to evaluate the

sample arithmetic statement. In more complex programs, significant additional memory accesses would be needed for temporary storage locations in memory. Because of its large number of memory accesses, this architecture is inefficient and consequently, is restricted to use in CPUs for simple, low-cost applications that do not require high performance.

The zero-address instruction case using a stack supports the concept of a *stack architecture*. Data-manipulation instructions such as ADD use no address, since they are performed on the top few elements of the stack. Single memory-address load and store operations, as shown in the program to evaluate the sample arithmetic statement, are used for data transfer. Since most of the stack is located in memory, as discussed earlier, one or more hidden memory accesses may be required for each stack operation. As register-register and load/store architectures have made strong performance advances, the high frequency of memory accesses in stack architectures has made them unattractive. However, stack architectures have begun to borrow technological advances from these other architectures. These architectures store substantial numbers of stack locations in the processor chip and handle transfers between these locations and the memory transparently. Stack architectures are particularly useful for rapid interpretation of high-level language programs in which the intermediate code representation uses stack operations.

Stack architectures are compatible with a very efficient approach to expression processing which uses postfix notation rather than the traditional infix notation to which we are accustomed. The infix expression

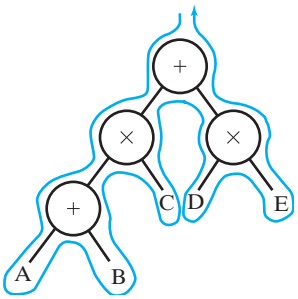
$$(A + B) \times C + (D \times E)$$

with the operators between the operands can be written as the postfix expression

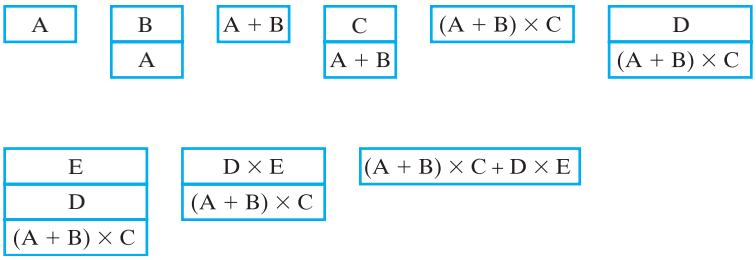
$$A\ B + C \times D\ E \times +$$

Postfix notation is called reverse Polish notation (RPN), honoring the Polish mathematician Jan Lukasiewicz, who proposed prefix (the reverse of postfix) notation; prefix was also known as Polish notation.

Conversion of $(A + B) \times C + (D \times E)$ to RPN can be achieved graphically, as shown in Figure 9-1. When the path shown traversing the graph passes a



☐ **FIGURE 9-1**
Graph for Example of Conversion from Infix to RPN



□ **FIGURE 9-2**
Stack Activity for Execution of Example Stack Program

variable, that variable is entered into the RPN expression. When the path passes an operation for the final time, the operation is entered into the RPN expression.

It is very easy to develop a program for an RPN expression. Whenever a variable is encountered, it is pushed onto the stack. Whenever an operation is encountered, it is executed on the implicit address TOS, or addresses TOS and TOS_{-1}, with the result placed in the new TOS. The program for the example RPN expression is

```
PUSH A
PUSH B
ADD
PUSH C
MUL
PUSH D
PUSH E
MUL
ADD
```

The execution of the program is illustrated by the successive stack states shown in Figure 9-2. As an operand is pushed on the stack, the stack grows by one stack location. When an operation is performed, the operand in the TOS is popped off and temporarily stored in a register. The operation is applied to the stored operand and the new TOS operand, and the result replaces the TOS operand.

9-3 Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on data stored in computer registers or memory words. How the operands are selected during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. The address of the operand produced by the application of such a rule is called the *effective address*. Computers use addressing-mode techniques to accommodate one or both of the following provisions:

1. To give programming flexibility to the user via pointers to memory, counters for loop control, indexing of data, and relocation of programs.
2. To reduce the number of bits in the address fields of the instruction.

The availability of various addressing modes gives the experienced programmer the ability to write programs that require fewer instructions. The effect, however, on throughput and execution time must be carefully weighed. For example, the presence of more complex addressing modes may actually result in lower throughput and longer execution time. Also, most machine-executable programs are produced by compilers that often do not use complex addressing modes effectively.

In some computers, the addressing mode of the instruction is specified by a distinct binary code. Other computers use a common binary code that designates both the operation and the addressing mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes two or more addressing modes are combined in one instruction.

An example of an instruction format with a distinct addressing-mode field is shown in Figure 9-3. The opcode specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is, it may designate a memory address or a processor register. Moreover, as discussed in the previous section, the instruction may have more than one address field. In that case, each address field is associated with its own particular addressing mode.

Implied Mode

Although most addressing modes modify the address field of the instruction, one mode needs no address field at all: the implied mode. In this mode, the operand is specified implicitly in the definition of the opcode. It is the implied mode that provides the location for the two-operand-plus-result operations when fewer than three addresses are contained in the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, any instruction that uses an accumulator without a second operand is an implied-mode instruction. For example, data-manipulation instructions in a stack computer, such as ADD, are implied-mode instructions, since the operands are implied to be on top of stack.

Immediate Mode

In the immediate mode, the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address

☐ **FIGURE 9-3**
Instruction Format with Mode Field

field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful, for example, for initializing registers to a constant value.

Register and Register-Indirect Modes

Earlier, we mentioned that the address field of the instruction may specify either a memory location or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode. In this mode, the operands are in registers that reside within the processor of the computer. The particular register is selected by a register address field in the instruction format.

In the register-indirect mode, the instruction specifies a register in the processor whose content gives the address of the operand in memory. In other words, the selected register contains the memory address of the operand, rather than the operand itself. Before using a register-indirect mode instruction, the programmer must ensure that the memory address is available in the processor register. A reference to the register is then equivalent to specifying a memory address. The advantage of register-indirect mode is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

An autoincrement or autodecrement mode is similar to the register-indirect mode, except that the register is incremented or decremented after (or before) its address value is used to access memory. When the address stored in the register refers to an array of data in memory, it is convenient to increment the register after each access to the array. This can be achieved by using a separate register-increment instruction. However, because it is such a common requirement, some computers incorporate an autoincrement mode that increments the content of the register containing the address after the memory data are accessed.

In the following instruction, an autoincrement mode is used to add the constant value 3 to the elements of an array addressed by register $R1$:

$$\text{ADD} \qquad (\text{R1})+, 3 \qquad M[R1] \leftarrow M[R1]+3, R1 \leftarrow R1+1$$

$R1$ is initialized to the address of the first element in the array. Then the ADD instruction is repeatedly executed until the addition of 3 to all elements of the array has occurred. The register transfer statement accompanying the instruction shows the addition of 3 to the memory location addressed by $R1$ and the incrementing of $R1$ in preparation for the next execution of the ADD on the next element in the array.

Direct Addressing Mode

In the direct addressing mode, the address field of the instruction gives the address of the operand in memory in a data-transfer or data-manipulation instruction. An example of a data-transfer instruction is shown in Figure 9-4. The instruction in memory consists of two words. The first, at address 250, has the opcode for "load to *ACC*" and a mode field specifying a direct address. The second word of the instruction, at address 251, contains the address field, symbolized by ADRS, and is equal to 500. The *PC* holds the address of the instruction, which is brought from memory using two

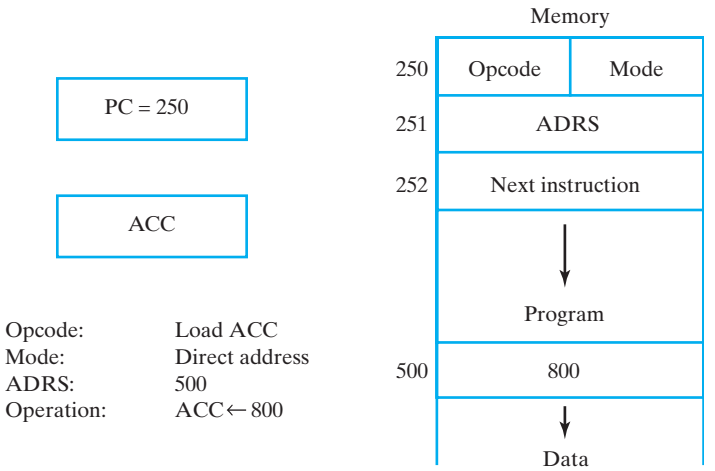


FIGURE 9-4
Example Demonstrating Direct Addressing for a Data-Transfer Instruction

memory accesses. Simultaneously with or after the completion of the first access, the *PC* is incremented to 251. Then the second access for ADRS occurs and the *PC* is again incremented. The execution of the instruction results in the operation

$$ACC \leftarrow M[\text{ADRS}]$$

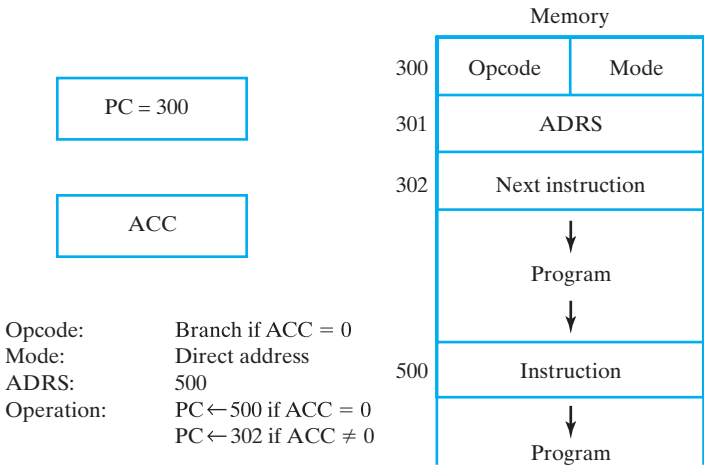
Since ADRS = 500 and M[500] = 800, the *ACC* receives the number 800. After the instruction is executed, the *PC* holds the number 252, which is the address of the next instruction in the program.

Now consider a branch-type instruction, as shown in Figure 9-5. If the contents of *ACC* equal 0, control branches to ADRS; otherwise, the program continues with the next instruction in sequence. When $ACC = 0$, the branch to address 500 is accomplished by loading the value of the address field ADRS into the *PC*. Control then continues with the instruction at address 500. When $ACC \neq 0$, no branch occurs, and the *PC*, which was incremented twice during the fetch of the instruction, holds the address 302, the address of the next instruction in sequence.

Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes, it is useful to distinguish between the address part of the instruction, as given in the address field, and the address used by the control when executing the instruction. Recall that we refer to the latter as the effective address.

Indirect Addressing Mode

In the indirect addressing mode, the address field of the instruction gives the address at which the effective address is stored in memory. The control unit fetches the instruction from memory and uses the address part to access memory again in order



☐ **FIGURE 9-5**
Example Demonstrating Direct Addressing in a Branch Instruction

to read the effective address. Consider the instruction "load to *ACC*" given in Figure 9-4. If the mode specifies an indirect address, the effective address is stored in *M*[ADRS]. Since ADRS = 500 and *M*[ADRS] = 800, the effective address is 800. This means that the operand loaded into the *ACC* is the one found in memory at address 800 (not shown in the figure).

Relative Addressing Mode

Some addressing modes require that the address field of the instruction be added to the content of a specified register in the CPU in order to evaluate the effective address. Often, the register used is the *PC*. In the relative addressing mode, the effective address is calculated as follows:

$$\text{Effective address} = \text{Address part of the instruction} + \text{Contents of } PC$$

The address part of the instruction is considered to be a signed number that can be either positive or negative. When this number is added to the contents of the *PC*, the result produces an effective address whose position in memory is relative to the address of the next instruction in the program.

To clarify this with an example, let us assume that the *PC* contains the number 250 and the address part of the instruction contains the number 500, as in Figure 9-6, with the mode field specifying a relative address. The instruction at location 250 is read from memory during the fetch phase of the operation cycle, and the *PC* is incremented by 1 to 251. Since the instruction has a second word, the control unit reads the address field into a control register, and the *PC* is incremented to 252. The computation of the effective address for the relative addressing mode is 252 + 500 = 752. The result is that the operand associated with the instruction is 500 locations away, relative to the location of the next instruction.

Relative addressing is often used in branch-type instructions when the branch address is in a location close to the instruction word. Relative addressing produces

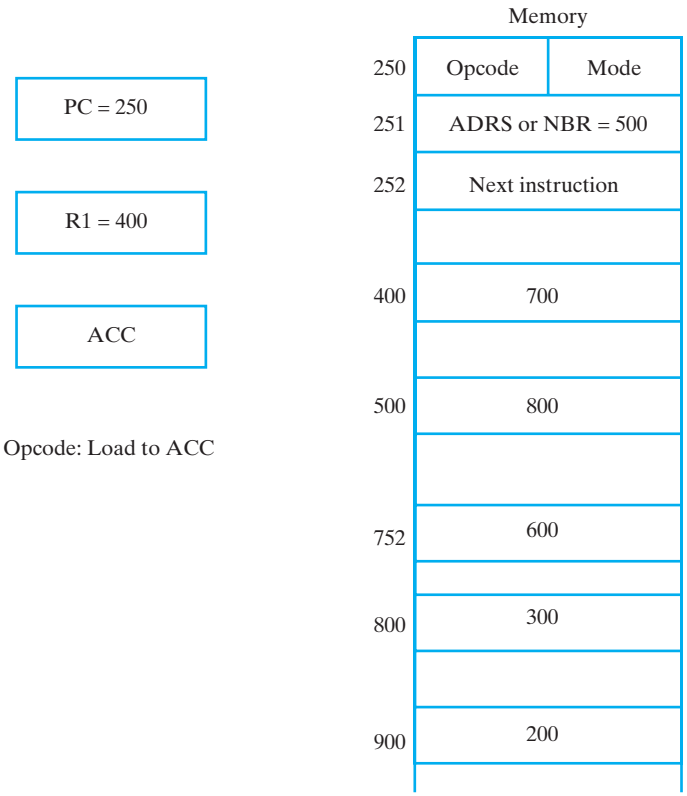


FIGURE 9-6
Numerical Example for Addressing Modes

more compact instructions, since the relative address can be specified with fewer bits than are required to designate the entire memory address. This permits the relative address field to be included in the same instruction word as the opcode.

Indexed Addressing Mode

In the indexed addressing mode, the content of an index register is added to the address part of the instruction to obtain the effective address. The index register may be a special CPU register or simply a register in a register file. We illustrate the use of indexed addressing by considering an array of data in memory. The address field of the instruction defines the beginning address of the array. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the register. Any operand in the array can be accessed with the same instruction, provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands.

Some computers dedicate one CPU register to function solely as an index register. This register is addressed implicitly when an index-mode instruction is used. In

computers with many processor registers, any CPU register can be used as an index register. In such a case, the index register to be used must be specified with a register field within the instruction format.

A specialized variation of the index mode is the base-register mode. In this mode, the contents of a base register are added to the address part of the instruction to obtain the effective address. This is similar to indexed addressing, except that the register is called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way addresses are computed: an index register is assumed to hold an index number that is relative to the address field of the instruction; a base register is assumed to hold a base address, and the address field of the instruction gives a displacement relative to the base address.

Summary of Addressing Modes

In order to show the differences among the various modes, we investigate the effect of the addressing mode on the instruction shown in Figure 9-6. The instruction in addresses 250 and 251 is "load to *ACC*," with the address field ADRS (or an operand NBR) equal to 500. The *PC* has the number 250 for fetching this instruction. The content of a processor register *R*1 is 400, and the *ACC* receives the result after the instruction is executed. In the direct mode, the effective address is 500, and the operand to be loaded into the *ACC* is 800. In the immediate mode, the operand 500 is loaded into the *ACC*. In the indirect mode, the effective address is 800, and the operand is 300. In the relative mode, the effective address is 500 + 252 = 752, and the operand is 600. In the index mode, the effective address is 500 + 400 = 900, assuming that *R*1 is the index register. In the register mode, the operand is in *R*1, and 400 is loaded into the *ACC*. In the register-indirect mode, the effective address is the contents of *R*1, and the operand loaded into the *ACC* is 700.

Table 9-1 lists the value of the effective address and the operand loaded into the *ACC* for the seven addressing modes. The table also shows the operation with a

□ TABLE 9-1
Symbolic Convention for Addressing Modes

			Refers to Figure 9-6	
Addressing Mode	**Symbolic Convention**	**Register Transfer**	**Effective Address**	**Contents of *ACC***
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register-indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

register transfer statement and a symbolic convention for each addressing mode. LDA is the symbol for the load-to-accumulator opcode. In the direct mode, we use the symbol ADRS for the address part of the instruction. The # symbol precedes the operand NBR in the immediate mode. The symbol ADRS enclosed in square brackets symbolizes an indirect address, which some compilers or assemblers designate with the symbol @. The symbol $ before the address makes the effective address relative to the *PC*. An index-mode instruction is recognized by the symbol of a register placed in parentheses after the address symbol. The register mode is indicated by giving the name of the processor register following LDA. In the register-indirect mode, the name of the register that holds the effective address is enclosed in parentheses.

9-4 Instruction Set Architectures

Computers provide a set of instructions to permit computational tasks to be carried out. The instruction sets of different computers differ in several ways from each other. For example, the binary code assigned to the opcode field varies widely for different computers. Likewise, although a standard exists (see Reference 2), the symbolic name given to instructions varies for different computers. In comparison to these minor differences, however, there are two major types of instruction set architectures that differ markedly in the relationship of hardware to software: *Complex instruction set computers* (CISCs) provide hardware support for high-level language operations and have compact programs; *reduced instruction set computers* (RISCs) emphasize simple instructions and flexibility that, when combined, provide higher throughput and faster execution. These two architectures can be distinguished by considering the properties that characterize their instruction sets.

A *RISC architecture* has the following properties:

1. Memory accesses are restricted to load and store instructions, and data manipulation instructions are register-to-register.
2. Addressing modes are limited in number.
3. Instruction formats are all of the same length.
4. Instructions perform elementary operations.

The goal of a RISC architecture is high throughput and fast execution. To achieve these goals, accesses to memory, which typically take longer than other elementary operations, are to be avoided, except for fetching instructions. A result of this view is the need for a relatively large register file. Because of the fixed instruction length, limited addressing modes, and elementary operations, the control unit of a RISC is comparatively simple and is typically hardwired. In addition, the underlying organization is universally a pipelined design, as covered in Chapter 10.

A purely *CISC architecture* has the following properties:

1. Memory access is directly available to most types of instructions.
2. Addressing modes are substantial in number.
3. Instruction formats are of different lengths.
4. Instructions perform both elementary and complex operations.

The goal of the CISC architecture is to match more closely the operations used in programming languages and to provide instructions that facilitate compact programs and conserve memory. In addition, efficiencies in performance may result through a reduction in the number of instruction fetches from memory, compared with the number of elementary operations performed. Because of the high memory accessibility, the register files in a CISC may be smaller than in a RISC. Also, because of the complexity of the instructions and the variability of the instruction formats, microprogrammed control is more likely to be used. In the quest for speed, the microprogrammed control in newer designs is likely to be controlling a pipelined datapath. CISC instructions are converted to a sequence of RISC-like operations that are processed by the RISC-like pipeline, as discussed in detail in Chapter 10.

Actual instruction set architectures range between those which are purely RISC and those which are purely CISC. Nevertheless, there is a basic set of elementary operations that most computers include among their instructions. In this chapter, we will focus primarily on elementary instructions that are included in both CISC and RISC instruction sets. Most elementary computer instructions can be classified into three major categories: (1) data-transfer instructions, (2) data-manipulation instructions, and (3) program-control instructions.

Data-transfer instructions cause transfer of data from one location to another without changing the binary information content. Data-manipulation instructions perform arithmetic, logic, and shift operations. Program-control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer. In addition to the basic instruction set, a computer may have other instructions that provide special operations for particular applications.

9-5 Data-Transfer Instructions

Data-transfer instructions move data from one place in the computer to another without changing the data. Typical transfers are between memory and processor registers, between processor registers and input and output registers, and among the processor registers themselves.

Table 9-2 gives a list of eight typical data-transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol, the assembly-language abbreviation recommended by an IEEE standard (Reference 2). Different computers, however, may use different mnemonics for the same instruction name. The load instruction is used to designate a transfer from memory to a processor register. The store instruction designates a transfer from a processor register into a memory word. The move instruction is used in computers with multiple processor registers to designate a transfer from one register to another. It is also used for data transfer between registers and memory and between two memory words.

The exchange instruction exchanges information between two registers, between a register and a memory word, or between two memory words. The push and pop instructions are for stack operations described next.

Stack Instructions

The stack architecture introduced earlier possesses features that facilitate a number of data-processing and control tasks. A stack is used in some electronic calculators

TABLE 9-2
Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOVE
Exchange	XCH
Push	PUSH
Pop	POP
Input	IN
Output	OUT

and computers for the evaluation of arithmetic expressions. Unfortunately, because of the negative effects on performance of having the stack reside primarily in memory, a stack in a computer typically handles only state information related to procedure calls and returns and interrupts, as explained in Sections 9-8 and 9-9.

The stack instructions push and pop transfer data between a memory stack and a processor register or memory. The *push* operation places a new item onto the top of the stack. The *pop* operation removes one item from the stack so that the stack pops up. However, nothing is really physically pushed or popped in the stack. Rather, the memory stack is essentially a portion of a memory address space accessed by an address that is always incremented or decremented before or after the memory access. The register that holds the address for the stack is called a *stack pointer* (*SP*) because its value always points to TOS, the item at the Top Of Stack. Push and pop operations are implemented by decrementing or incrementing the stack pointer.

Figure 9-7 shows a portion of a memory organized as a stack that grows from higher to lower addresses. The stack pointer, *SP*, holds the binary address of the item that is currently on top of the stack. Three items are presently stored in the stack: *A*, *B*,

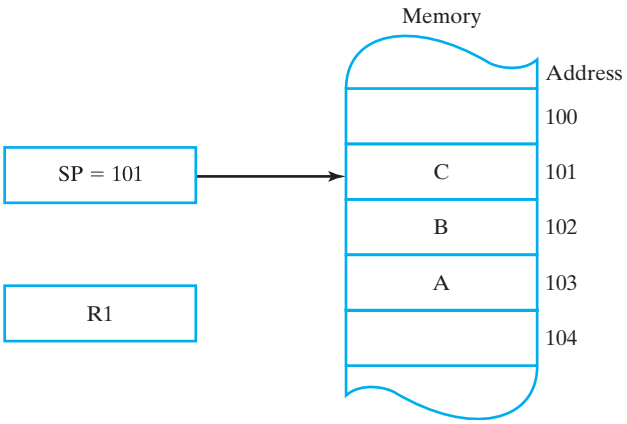


FIGURE 9-7
Memory Stack

and *C*, in consecutive addresses 103, 102, and 101, respectively. Item *C* is on top of the stack, so *SP* contains 101. To remove the top item, the stack is popped by reading the item at address 101 and incrementing *SP*. Item *B* is now on top of the stack, since *SP* contains address 102. To insert a new item, the stack is pushed by first decrementing *SP* and then writing the new item on top of the stack using *SP* as the memory address. Note that item *C* has been read out of the stack, but is not physically removed from it. This does not matter as far as the stack operation is concerned, because when the stack is pushed, a new item is written over it regardless of what was there before.

We assume that the items in the stack communicate with a data register *R*1 or a memory location *X*. A new item is placed on the stack with the push operation sequence:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow R1$$

The stack pointer is decremented so that it points at the address of the next word. A memory write microoperation inserts the word from *R*1 onto the top of the stack. Note that *SP* holds the address of the top of the stack and that *M*[*SP*] denotes the memory word specified by the address presently in *SP*. An item is deleted from the stack with the pop operation pair:

$$R1 \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

The top item is read from the stack into *R*1, and the stack pointer is incremented to point at the next item in the stack, which is the new top of the stack. The two microoperations described in this case can be in parallel.

The two microoperations needed for either the push or the pop operation are an access to memory through *SP* and an update of *SP*. In Figure 9-7, the stack grows by decreasing the memory address. By contrast, a stack may be constructed to grow by increasing the memory address. In such a case, *SP* is incremented for the push operation and decremented for the pop operation. A stack may also be constructed so that *SP* points to the next empty location above the top of the stack. In that case, the order of execution of the microoperations must be modified.

A stack pointer is loaded with an initial value, which must be the bottom address of an assigned stack in memory. From then on, *SP* is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the processor can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

The final pair of data transfer instructions, input and output, depend on the type of input-output used, as described next.

Independent versus Memory-Mapped I/O

Input and output (I/O) instructions transfer data between processor registers and input and output devices. These instructions are similar to load and store instructions, except that the transfers are to and from external registers instead of memory words. The computer has a number of input and output ports, with one or more ports

dedicated to communication with a specific input or output device. A *port* is typically a register with input and/or output lines attached to the device. The particular port is chosen by an address, in a manner similar to the way an address selects a word in memory. Input and output instructions include an address field in their format, for specifying the particular port selected for the transfer of data.

Port addresses are assigned in two ways. In the *independent I/O system*, the address ranges assigned to memory and I/O ports are independent from each other. The computer has distinct input and output instructions, as listed in Table 9-2, containing a separate address field that is interpreted by the control and used to select a particular I/O port. Independent I/O addressing isolates memory and I/O selection, so that the memory address range is not affected by the port address assignment. For this reason, the method is also referred to as an *isolated I/O configuration*.

In contrast to independent I/O, *memory-mapped I/O* assigns a subrange of the memory addresses for addressing I/O ports. There are no separate addresses for handling input and output transfers, since I/O ports are treated as memory locations in one common address range. Each I/O port is regarded as a memory location, similar to a memory word. Computers that adopt the memory-mapped scheme have no distinct input or output instructions, because the same instructions are used for manipulating both memory and I/O data. For example, the load and store instructions used for memory transfer are also used for I/O transfer, provided that the address associated with the instruction is assigned to an I/O port and not to a memory word. The advantage of this scheme is the simplicity that results with the same set of instructions serving for both memory and I/O access.

9-6 Data-Manipulation Instructions

Data-manipulation instructions perform operations on data and provide the computational capabilities of the computer. In a typical computer, they are usually divided into three basic types:

1. Arithmetic instructions.
2. Logical and bit-manipulation instructions.
3. Shift instructions.

A list of elementary data-manipulation instructions looks very much like the list of microoperations given in Chapter 8. However, an instruction is typically processed by executing a *sequence* of one or more microinstructions. A microoperation is an elementary operation executed by the hardware of the computer under the control of the control unit. In contrast, an instruction may involve several elementary operations that fetch the instruction, bring the operands from appropriate processor registers, and store the result in the specified location.

Arithmetic Instructions

The four basic arithmetic instructions are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. A list of typical arithmetic instructions is given in Table 9-3. The increment instruction adds one to

the value stored in a register or memory word. A common characteristic of the increment operation, when executed on a computer word, is that a binary number of all 1s produces a result of all 0s when incremented. The decrement instruction subtracts one from a value stored in a register or memory word. When decremented, a number of all 0s produces a number of all 1s.

The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the opcode. An arithmetic instruction may specify unsigned or signed integers, binary or decimal numbers, or floating-point data. The arithmetic operations with binary integers were presented in Chapters 1 and 3. The floating-point representation is used for scientific calculations and is presented in the next section.

The number of bits in any register is finite; therefore, the results of arithmetic operations are of finite precision. Most computers provide special instructions to facilitate double-precision arithmetic. A carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition with two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two operands and a borrow that may have resulted from a previous operation.

The subtract reverse instruction reverses the order of the operands, performing $B - A$ instead of $A - B$. The negate instruction performs the 2s complement of a signed number, which is equivalent to multiplying the number by -1.

Logical and Bit-Manipulation Instructions

Logical instructions perform binary operations on words stored in registers or memory words. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. Logical instructions consider each bit of the operand separately and treat it as a binary variable. By proper application of the logical

☐ **TABLE 9-3**
Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Subtract reverse	SUBR
Negate	NEG

instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory.

Some typical logical and bit-manipulation instructions are listed in Table 9-4. The clear instruction causes the specific operand to be replaced by 0s. The set instruction causes the operand to be replaced by 1s. The complement instruction inverts all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of two operands. Although logical instructions perform Boolean operations, when used on words they often are viewed as performing bit-manipulation operations. Three bit-manipulation operations are possible. A selected bit can be cleared to 0, set to 1, or complemented. The three logical instructions are usually applied to do just that.

The AND instruction is used to clear a bit or a selected group of bits of an operand to 0. For any Boolean variable X, the relationship $X \cdot 0 = 0$ dictates that a binary variable ANDed with a 0 produces a 0; and similarly, the relationship $X \cdot 1 = X$ dictates that the variable does not change when ANDed with a 1. Therefore, the AND instruction is used to selectively clear bits of an operand by ANDing the operand with a word that has 0s in the bit positions that must be cleared and 1s in the bit positions that must remain the same. The AND instruction is also called a *mask* because, by inserting 0s, it masks a selected portion of an operand. AND is also sometimes referred to as a *bit clear* instruction.

The OR instruction is used to set a bit or a selected group of bits of an operand to 1. For any Boolean variable X, the relationship $X + 1 = 1$ dictates that a binary variable ORed with a 1 produces a 1; similarly, the relationship $X + 0 = X$ dictates that the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing the operand with a word with 1s in the bit positions that must be set to 1. The OR instruction is sometimes called a *bit set* instruction.

The XOR instruction is used to selectively complement bits of an operand. This is because of the Boolean relationships $X \oplus 1 = \overline{X}$ and $X \oplus 0 = X$. A binary variable is complemented when XORed with a 1, but does not change value when

☐ **TABLE 9-4**
Typical Logical and Bit-Manipulation Instructions

Name	Mnemonic
Clear	CLR
Set	SET
Complement	NOT
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC

XORed with a 0. The XOR instruction is sometimes called a *bit complement* instruction.

Other bit-manipulation instructions included in Table 9-4 can clear, set, or complement the carry bit. Additional instructions can clear, set, or complement other status bits or flag bits in a similar manner.

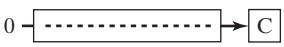
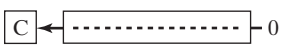
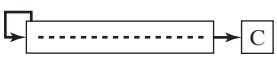
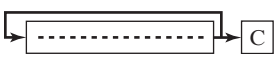
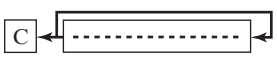
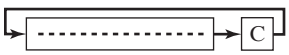
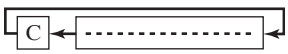
Shift Instructions

Instructions to shift the content of a single operand are provided in several varieties. Shifts are operations in which the bits of the operand are moved to the left or to the right. The incoming bit shifted in at the end of the word determines the type of shift. Instead of using just a 0, as for sl and sr in Chapter 8, here we add further possibilities. The shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations.

Table 9-5 lists four types of shift instructions, both right and left versions. The small diagrams shown in the right column show the bit movement for each of the shifts in the Intel IA-32 ISA. In all cases, the outgoing bit is copied into the carry status bit *C*. The logical shifts insert 0 into the incoming bit position during the shift. Arithmetic shifts conform to the rules for shifting 2s complement signed numbers. The arithmetic shift right instruction uses sign extension, filling the leftmost position with its own value during the shift. The arithmetic shift left instruction inserts 0 into the incoming bit in the rightmost position and is identical to the logical shift left instruction.

The rotate instructions produce a circular shift: the values shifted out of the outgoing bit are rotated back into the incoming bit. The rotate-with-carry instructions treat the carry bit as an extension of the register whose word is being rotated.

☐ **TABLE 9-5**
Typical Shift Instructions

Name	Mnemonic	Diagram
Logical shift right	SHR	
Logical shift left	SHL	
Arithmetic shift right	SHRA	
Arithmetic shift left	SHLA	
Rotate right	ROR	
Rotate left	ROL	
Rotate right with carry	RORC	
Rotate left with carry	ROLC	

Thus, a rotate left with carry transfers the carry bit into the incoming bit in the rightmost bit position of the register, transfers the outgoing bit from the leftmost bit of the register into the carry, and shifts the entire register to the left.

Most computers have a multiple-field format for the shift instruction that provides for shifting multiple, rather than just one, bit positions. One field contains the opcode, and another contains the number of positions that an operand is to be shifted. A shift instruction may include the following five fields:

OP REG TYPE RL COUNT

OP is the opcode field for specifying a shift, and REG is a register address that specifies the location of the operand. TYPE is a 2-bit field that specifies one of the four types of shifts (logical, arithmetic, rotate, and rotate with carry), while RL is a 1-bit field that specifies whether a shift is to the right or the left. COUNT is a k-bit field that specifies shifts of up to $2^k - 1$ positions. With such a format, it is possible to specify the type of shift, the direction of the shift, and the number of positions to be shifted, all in one instruction.

Note that for shifts of greater than one position, the filling of the positions vacated by the shift is consistent with the diagrams shown in Table 9-5. In the Intel IA-32 ISA, in addition to the use of the carry bit C, the N and Z condition code bits are also set based on the shift results. The overflow bit, V, is defined only for 1-bit shifts.

9-7 Floating-Point Computations

In many scientific calculations, the range of numbers is very large. In a computer, the way to express such numbers is in floating-point notation. The floating-point number has two parts, one containing the sign of the number and a *fraction* (sometimes called a *mantissa*) and the other designating the position of the radix point in the number and called the *exponent*. For example, the decimal number +6132.789 is represented in floating-point notation as

Fraction	Exponent
+.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+.6132789 \times 10^{+4}$. Decimal floating-point numbers are interpreted as representing a number in the form

$$F \times 10^E$$

where F is the fraction and E the exponent. Only the fraction and the exponent are physically represented in computer registers; radix 10 and the decimal point of the fraction are assumed and are not shown explicitly. A floating-point binary number is represented in a similar manner, except that it uses radix 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as

Fraction	Exponent
01001110	000100

The fraction has a 0 in the leftmost position to denote a plus. The binary point of the fraction follows the sign bit, but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$F \times 2^E = +(0.1001110)_2 \times 2^{+4}$$

A floating-point number is said to be *normalized* if the most significant digit of the fraction is nonzero. For example, the decimal fraction 0.350 is normalized, but 0.0035 is not. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit; it is usually represented in floating-point by all 0s in both the fraction and the exponent.

Floating-point representation increases the range of numbers that can be accommodated in a given register. Consider a computer with 48-bit registers. Since one bit must be reserved for the sign, the range of signed integers will be $\pm(2^{47} - 1)$, which is approximately $\pm 10^{14}$. The 48 bits can be used to represent a floating-point number, with one bit for the sign, 35 bits for the fraction, and 12 bits for the exponent. The largest positive or negative number that can be accommodated is thus

$$\pm(1 - 2^{-35}) \times 2^{+2047}$$

This number is derived from a fraction that contains 35 1s, and an exponent with a sign bit and 11 1s. The maximum exponent is $2^{11} - 1$, or 2047. The largest number that can be accommodated is approximately equivalent to decimal 10^{615}. Although a much larger range is represented, there are still only 48 bits in the representation. As a consequence, exactly the same number of numbers are represented. Hence, the range is traded for the precision of the numbers, which is reduced from 48 bits to 35 bits.

Arithmetic Operations

Arithmetic operations with floating-point numbers are more complicated than with integer numbers, and their execution takes longer and requires more complex hardware. Adding and subtracting two numbers require that the radix points be aligned, since the exponent parts must be equal before adding or subtracting the fractions. The alignment is done by shifting one fraction and correspondingly adjusting its exponent until it is equal to the other exponent. Consider the sum of the following floating-point numbers:

$$.5372400 \times 10^2$$
$$+ .1580000 \times 10^{-1}$$

It is necessary that the two exponents be equal before the fractions can be added. We can either shift the first number three positions to the left or shift the second number three positions to the right. When the fractions are stored in registers, shifting to the left causes a loss of the most significant digits. Shifting to the right causes a loss of the least

significant digits. The second method is preferable because it only reduces the precision, whereas the first method may cause an error. The usual alignment procedure is to shift the fraction with the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the fractions can be added:

$$\begin{array}{r} .5372400 \times 10^2 \\ +\ .0001580 \times 10^2 \\ \hline .5373980 \times 10^2 \end{array}$$

When two normalized fractions are added, the sum may contain an overflow digit. An overflow can be corrected by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros in the fraction, as shown in the following example:

$$\begin{array}{r} .56780 \times 10^5 \\ -\ .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

A floating-point number that has a 0 in the most significant position of the fraction is not normalized. To normalize the number, it is necessary to shift the fraction to the left and decrement the exponent until a nonzero digit appears in the first position. In the preceding example, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers, a normalization procedure is performed after each operation to ensure that all results are in normalized form.

Floating-point multiplication and division do not require an alignment of the fractions. Multiplication can be performed by multiplying the two fractions and adding the exponents. Division is accomplished by dividing the fractions and subtracting the exponents. In the examples shown, we used decimal numbers to demonstrate arithmetic operations on floating-point numbers. The same procedure applies to binary numbers, except that the base of the exponent is 2 instead of 10.

Biased Exponent

The sign and fraction part of a floating-point number is usually a signed-magnitude representation. The exponent representation employed in most computers is known as a *biased exponent*. The bias is an excess number added to the exponent so that, internally, all exponents become positive. As a consequence, the sign of the exponent is removed from being a separate entity.

Consider, for example, the range of decimal exponents from -99 to $+99$. This is represented by two digits and a sign. If we use an excess 99 bias, then the biased exponent e will be equal to $e = E + 99$, where E is the actual exponent. For $E = -99$, we have $e = -99 + 99 = 0$; and for $E = +99$, we have $e = 99 + 99 = 198$. In this way, the biased exponent is represented in a register as a positive number in the range from 000 to 198. Positive-biased exponents have a range of numbers from 099 to 198. Subtraction of the bias, 99, gives the positive values from 0 to +99. Negative-biased

exponents have a range from 098 to 000. Subtraction of 99 gives the negative values from −1 to −99.

The advantage of biased exponents is that the resulting floating-point numbers contain only positive exponents. It is then simpler to compare the relative magnitude between two numbers without being concerned with the signs of their exponents. Another advantage is that the most negative exponent converts to a biased exponent with all 0s. The floating-point representation of zero is then a zero fraction and a zero biased exponent, which is the smallest possible exponent.

Standard Operand Format

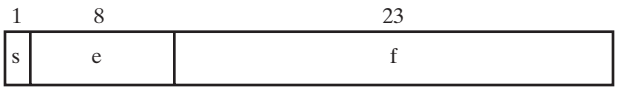
Arithmetic instructions that perform operations with floating-point data often use the suffix F. Thus, ADDF is an add instruction with floating-point numbers. There are two standard formats for representing a floating-point operand: the single-precision data type, consisting of 32 bits, and the double-precision data type, consisting of 64 bits. When both types of data are available, the single-precision instruction mnemonic uses an FS suffix, and the double precision uses FL (for "floating-point long").

The format of the IEEE standard (see Reference 3) single-precision floating-point operand is shown in Figure 9-8. It consists of 32 bits. The sign bit *s* designates the sign for the fraction. The biased exponent *e* contains 8 bits and uses an excess 127 number. The fraction *f* consists of 23 bits. The binary point is assumed to be immediately to the left of the most significant bit of the *f* field. In addition, an implied 1 bit is inserted to the left of the binary point, which in effect expands the number to 24 bits representing a value from 1.0_2 to $1.11\ldots\ 1_2$. The component of the binary floating-point number that consists of a leading bit to the left of the implied binary point, together with the fraction in the field, is called the *significand*. Following are some examples of field values and the corresponding significands:

f Field	Significand	Decimal Equivalent
100 . . . 0	1.100 . . . 0	1.50
010 . . . 0	1.010 . . . 0	1.25
000 . . . 0	1.000 . . . 0*	1.00*

* Assuming the exponent is not equal to 00 . . . 0.

Even though the *f* field by itself may not be normalized, the significand is always normalized, because it has a nonzero bit in the most significant position. Since normalized numbers must have a nonzero most significant bit, this 1 bit is not included explicitly in the format, but must be inserted by the hardware during arithmetic computations. The exponent field uses an excess 127 bias value for



□ **FIGURE 9-8**
IEEE Floating-Point Operand Format

normalized numbers. The range of valid exponents is from −126 (represented as 00000001) through +127 (represented as 11111110). The maximum (11111111) and minimum (00000000) values for the *e* field are reserved to indicate exceptional conditions. Table 9-6 shows the biased and actual values of some exponents.

Normalized numbers are numbers that can be expressed as floating-point operands in which the *e* field is neither all 0s nor all 1s. The value of the number is derived from the three fields in the format of Figure 9-8 using the formula

$$(-1)^s 2^{e-127} \times (1.f)$$

The most positive normalized number that can be obtained has a 0 for the sign bit for a positive sign, a biased exponent equal to 254, and an *f* field with 23 1s. This gives an exponent $E = 254 - 127 = 127$. The significand is equal to $1 + 1 - 2^{-23} = 2 - 2^{-23}$. The maximum positive number that can be accommodated is

$$+2^{127} \times (2 - 2^{-23})$$

The smallest positive normalized number has a biased exponent equal to 00000001 and a fraction of all 0s. The exponent is $E = 1 - 127 = -126$, and the significand is equal to 1.0. The smallest positive number that can be accommodated is $+2^{-126}$. The corresponding negative numbers are the same, except that the sign bit is negative. As mentioned before, exponents with all 0s or all 1s (decimal 255) are reserved for the following special conditions:

1. When $e = 255$ and $f = 0$, the number represents plus or minus infinity. The sign is determined from the sign bit *s*.
2. When $e = 255$ and $f \neq 0$, the representation is considered to be *not a number*, or NaN, regardless of the sign value. NaNs are used to signify invalid operations, such as the multiplication of zero by infinity.
3. When $e = 0$ and $f = 0$, the number denotes plus or minus zero.

☐ **TABLE 9-6**
Evaluating Biased Exponents

Exponent *E* in decimal	Biased exponent $e = E + 127$	
	Decimal	Binary
−126	−126 + 127 = 1	00000001
−001	−001 + 127 = 126	01111110
000	000 + 127 = 127	01111111
+001	001 + 127 = 128	10000000
+126	126 + 127 = 253	11111101
+127	127 + 127 = 254	11111110

4. When $e = 0$ and $f \neq 0$, the number is said to be denormalized. This is the name given to numbers with a magnitude less than the minimum value that is represented in the normalized format.

9-8 Program Control Instructions

The instructions of a program are stored in successive memory locations. When processed by the control, the instructions are read from consecutive memory locations and executed one by one. Each time an instruction is fetched from memory, the *PC* is incremented so that it contains the address of the next instruction in sequence. In contrast, a program control instruction, when executed, may change the address value in the *PC* and cause the flow of control to be altered. The change in the *PC* as a result of the execution of a program control instruction causes a break in the sequence of execution of instructions. This is an important feature of digital computers, since it provides control over the flow of program execution and a capability of branching to different program segments, depending on previous computations.

Some typical program control instructions are listed in Table 9-7. The branch and jump instructions are often used interchangeably to mean the same thing, although sometimes they are used to denote different addressing modes. For example, the jump may use direct or indirect addressing, whereas the branch uses relative addressing. The branch (or jump) is usually a one-address instruction. When executed, the branch instruction causes a transfer of the effective address into the *PC*. Since the *PC* contains the address of the instruction to be executed next, the next instruction will be fetched from the location specified by the effective address.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified effective address without any conditions. The conditional branch instruction specifies a condition that must be met in order for the branch to occur, such as the value in a specified register being negative. If the condition is met, the *PC* is loaded with the effective address, and the next instruction is taken from this address. If the condition is not met, the *PC* is not changed, and the next instruction is taken from the next location in sequence.

The call and return instructions are used in conjunction with procedures. Their performance and implementation are discussed later in this section.

□ **TABLE 9-7**
Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Call procedure	CALL
Return from procedure	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

The compare instruction performs a comparison via a subtraction, with the difference not retained. Instead, the comparison causes a conditional branch, changes the contents of a register, or sets or resets stored status bits. Similarly, the test instruction performs the logical AND of two operands without retaining the result and executes one of the actions listed for the compare instruction.

Based on their three possible actions, compare and test instructions are viewed to be of three distinct types, depending upon the way in which conditional decisions are handled. The first type executes the entire decision as a single instruction. For example, the contents of two registers can be compared and a branch or jump taken if the contents are equal. Since two register addresses and a memory address are involved, such an instruction requires three addresses. The second type of compare and test instruction also uses three addresses, all of which are register addresses. Considering the same example, if the contents of the first two registers are equal, a 1 is placed in the third register. If the contents are not equal, a 0 is placed in the third register. These two types of instructions avoid the use of stored status bits. In the first case, no such bit is required, and in the second case, a register is used to simulate its presence. The third type of compare and test has compare and test operations that set or reset stored status bits. Branch or jump instructions are then used to conditionally change the program sequence. This third type of compare and test instruction is discussed in the next subsection.

Conditional Branch Instructions

A conditional branch instruction is a branch instruction that may or may not cause a transfer of control, depending on the value of stored bits in the processor status register, *PSR*. Each conditional branch instruction tests a different combination of status bits for a condition. If the condition is true, control is transferred to the effective address. If the condition is false, the program continues with the next instruction.

Table 9-8 gives a list of conditional branch instructions that depend directly on the bits in the *PSR*. In most cases, the instruction mnemonic is constructed with the letter B (for "branch") and a letter for the name of the status bit. The letter N (for

☐ **TABLE 9-8**
Conditional Branch Instructions Relating to Status Bits in the PSR

Branch Condition	Mnemonic	Test Condition
Branch if zero	BZ	$Z = 1$
Branch if not zero	BNZ	$Z = 0$
Branch if carry	BC	$C = 1$
Branch if no carry	BNC	$C = 0$
Branch if minus	BN	$N = 1$
Branch if plus	BNN	$N = 0$
Branch if overflow	BV	$V = 1$
Branch if no overflow	BNV	$V = 0$

"not") is included if the status bit is tested for a 0 condition. Thus, BC is a branch if carry = 1, and BNC is a branch if carry = 0.

The zero status bit Z is used to check whether the result of an ALU operation or shift is equal to zero. The carry bit C is used to check the carry after the addition or the borrow after the subtraction of two operands in the ALU. It is also used in conjunction with shift instructions to check the value of the outgoing bit. The sign bit N reflects the state of the leftmost bit of the output from the ALU or shift. $N = 0$ denotes a positive sign and $N = 1$ a negative sign. These instructions can be used to check the value of the leftmost bit, whether it represents a sign or not. The overflow bit V is used in conjunction with arithmetic and shift operations with both signed and unsigned numbers.

As stated previously, the compare instruction performs a subtraction of two operands, say, $A - B$. The result of the operation is not transferred into a destination register, but the status bits are affected. The status bits provide information about the relative magnitude between A and B. Some computers provide special branch instructions that can be applied after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers are considered to be unsigned or signed.

The relative magnitude between two unsigned binary numbers A and B can be determined by subtracting $A - B$ and checking the C and Z status bits. Most commercial computers consider the C status bit as a carry after addition and a borrow after subtraction. A borrow occurs when $A < B$, because the most significant position must borrow a bit to complete the subtraction. A borrow does not occur if $A \geq B$, because the difference $A - B$ is positive. The condition for borrowing is the inverse of the condition for carrying when the subtraction is done by taking the 2s complement of B. Computers that use the C status bit as a borrow after a subtraction, complement the output carry after adding the 2s complement of the subtrahend and call this bit a borrow. The technique is typically applied to all instructions that use subtraction within the functional unit, not just the subtract instruction. For example, it applies to compare instructions.

The conditional branch instructions for unsigned numbers are listed in Table 9-9. It is assumed that a previous instruction has updated status bits C and Z after a subtraction $A - B$ or some other similar instruction. The words "above," "below," and

□ **TABLE 9-9**
Conditional Branch Instructions for Unsigned Numbers

Branch Condition	Mnemonic	Condition	Status Bits*
Branch if above	BA	$A > B$	$C + Z = 0$
Branch if above or equal	BAE	$A \geq B$	$C = 0$
Branch if below	BB	$A < B$	$C = 1$
Branch if below or equal	BBE	$A \leq B$	$C + Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

*Note that C here is a borrow bit.

☐ **TABLE 9-10**
Conditional Branch Instructions for Signed Numbers

Branch Condition	Mnemonic	Condition	Status Bits
Branch if greater	BG	$A > B$	$(N \oplus V) + Z = 0$
Branch if greater or equal	BGE	$A \geq B$	$N \oplus V = 0$
Branch if less	BL	$A < B$	$N \oplus V = 1$
Branch if less or equal	BLE	$A \leq B$	$(N \oplus V) + Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

"equal" are used to denote the relative magnitude between two unsigned numbers. The two numbers are equal if $A = B$. This is determined from the zero status bit Z, which is equal to 1 because $A = B$. A is below B and the borrow $C = 1$ when $A - B = 0$. For A to be below or equal to B ($A < B$), we must have $C = 1$ or $Z = 1$. The relationship ($A \leq B$), is the inverse of $A > B$ and is detected from the complemented condition of the status bits. Similarly, $A \leq B$ is the inverse of $A \geq B$, and $A \neq B$ is the inverse of $A = B$.

The conditional branch instructions for signed numbers are listed in Table 9-10. Again, it is assumed that a previous instruction has updated the status bits N, V, and Z after a subtraction $A - B$. The words "greater," "less," and "equal" are used to denote the relative magnitude between two signed numbers. If $N = 0$, the sign of the difference is positive, and A must be greater than or equal to B, provided that $V = 0$, indicating that no overflow occurred. An overflow causes a sign reversal, as discussed in Section 3-11. This means that if $N = 1$ and $V = 1$, there was a sign reversal, and the result should have been positive, which makes A greater than or equal to B. Therefore, the condition $A \geq B$ is true if both N and V are equal to 0 or both are equal to 1. This is the complement of the exclusive-OR operation.

For A to be greater than but not equal to B ($A > B$), the result must be positive and nonzero. Since a zero result gives a positive sign, we must ensure that the Z bit is 0 to exclude the possibility that $A = B$. Note that the condition $(N \oplus V) + Z = 0$ means that both the exclusive-OR operation and the Z bit must be equal to 0. The other two conditions in the table can be derived in a similar manner. The conditions BE (branch on equal) and BNE (branch on not equal) given for unsigned numbers apply to signed numbers as well and can be determined from $Z = 1$ and $Z = 0$, respectively.

Procedure Call and Return Instructions

A *procedure* is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a procedure may be called to perform its function many times at various points in the program, a procedure may be called to perform its function many times at various points in the program. Each time the procedure is called, a branch is made to the beginning of the procedure to start executing its

set of instructions. After the procedure has been executed, a branch is made again to return to the main program. A procedure is also referred to as a *subroutine*.

The instruction that transfers control to a procedure is known by different names, including call procedure, call subroutine, jump to subroutine, branch to subroutine, and branch and link. We will refer to the routine containing the procedure call as the *calling procedure*. The calling procedure is often referred to as the *caller*, and the procedure being called is often referred to as the *callee*. The call procedure instruction has a one-address field and performs two operations. First, it stores the value of the *PC*, which is the address following that of the call procedure instruction, in a temporary location. This address is called the *return address*, and the corresponding instruction is the *continuation point* in the calling procedure. Second, the address in the call procedure instruction—the address of the first instruction in the procedure—is loaded into the *PC*. When the next instruction is fetched, it comes from the called procedure.

The final instruction in every procedure must be a return to the calling procedure. The return instruction takes the address that was stored by the call procedure instruction and places it back in the *PC*. This results in a transfer of program execution back to the continuation point in the calling procedure.

Different computers use different temporary locations for storing the return address. Some computers store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The advantage of using a stack for the return address is that, when a succession of procedures are called, the sequential return address can be pushed onto the stack. The return instruction causes the stack to pop, and the contents of the top of the stack are then transferred to the *PC*. In this way, a return is always to the program that last called the procedure. A procedure call instruction using a stack is implemented with the following microoperation sequence:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$PC \leftarrow$ Effective address	Transfer control to procedure

The return instruction is implemented by popping the stack and transferring the return address to the *PC*:

$PC \leftarrow M[SP]$	Transfer return address to *PC*
$SP \leftarrow SP + 1$	Increment stack pointer

By using a procedure stack, all return addresses are automatically stored by the hardware in the memory stack. Thus, the programmer does not have to be concerned about managing the return addresses for procedures called from within procedures.

In addition to storing the return address, the program must also properly manage any parameter values transferred to the procedure and result values returned to the calling procedure, as well as temporary values stored in registers required by either the procedure or calling procedure. The method used by a programming language or compiler to ensure that the values are properly managed is commonly known as a *calling convention*. The calling convention will typically specify how parameter values are provided to the procedure, how result values are returned to the calling procedure, which registers may be overwritten by the procedure, and

which registers must be preserved by the procedure so that their values can be used by the calling procedure after the procedure returns control to it. A combination of registers and the stack is often used as part of the calling convention to pass parameter values to the procedure and return values to the calling procedure. The stack can also be used to preserve register values across the procedure call.

9-9 Program Interrupt

A program interrupt is used to handle a variety of situations that require a departure from the normal program sequence. A program interrupt transfers control from a program that is currently running to another service program as a result of an externally or internally generated request. Control returns to the original program after the service program is executed. In principle, the interrupt procedure is similar to a call procedure, except in three respects:

1. The interrupt is usually initiated at an unpredictable point in the program by an external or internal signal, rather than the execution of an instruction.
2. The address of the service program that processes the interrupt request is determined by a hardware procedure, rather than the address field of an instruction.
3. In response to an interrupt, it is necessary to store information that defines all or part of the contents of the register set, rather than storing only the program counter.

After the computer has been interrupted and the appropriate service program executed, the computer must return to exactly the same state that it was in before the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened. The state of the computer at the end of an execution of an instruction is determined from the contents of the register set. In addition to containing the condition codes, the *PSR* can specify what interrupts are allowed to occur and whether the computer is operating in user or system mode. Most computers have a resident operating system that controls and supervises all other programs. When the computer is executing a program that is part of the operating system, the computer is placed in system mode, in which certain instructions are privileged and can be executed in the system mode only. The computer is in user mode when it executes user programs, in which case it cannot execute the privileged instructions. The mode of the computer at any given time is determined from a special status bit or bits in the *PSR*.

Some computers store only the program counter when responding to an interrupt. In such computers, the program that performs the data processing for servicing the interrupt must include instructions to store the essential contents of the register set. Other computers store the entire register set automatically in response to an interrupt. Some computers have two sets of processor registers, so that when the program switches from user to system mode in response to an interrupt, it is not necessary to store the contents of processor registers, because each computer mode employs its own set of registers.

The hardware procedure for processing interrupts is very similar to the execution of a procedure call instruction. The contents of the register set of the processor are temporarily stored in memory, typically by being pushed onto a memory stack, and the address of the first instruction of the interrupt service program is loaded into the *PC*. The address of the service program is chosen by the hardware. Some computers assign one memory location for the beginning address of the service program:

the service program must then determine the source of the interrupt and proceed to service it. Other computers assign a separate memory location for each possible interrupt source. Sometimes, the interrupt source hardware itself supplies the address of the service routine. In any case, the computer must possess some form of hardware procedure for selecting a branch address for servicing the interrupt.

Most computers will not respond to an interrupt until the instruction that is in the process of being executed is completed. Then, just before going to fetch the next instruction, the control checks for any interrupt signals. If an interrupt has occurred, control goes to a hardware interrupt cycle. During this cycle, the contents of some part or all of the register set are pushed onto the stack. The branch address for the particular interrupt is then transferred to the *PC*, and the control goes to fetch the next instruction, which is the beginning of the interrupt service routine. The last instruction in the service routine is a return from the interrupt instruction. When this return is executed, the stack is popped to retrieve the return address, which is transferred to the *PC* as well as any stored contents of the rest of the register set, which are transferred back to the appropriate registers.

Types of Interrupts

The three major types of interrupts that cause a break in the normal execution of a program are as follows:

1. External interrupts.
2. Internal interrupts.
3. Software interrupts.

External interrupts come from input or output devices, from timing devices, from a circuit monitoring the power supply, or from any other external source. Conditions that cause external interrupts are an input or output device requesting a transfer of data, an external device completing a transfer of data, the time-out of an event, or an impending power failure. A time-out interrupt may result from a program that is in an endless loop and thus exceeds its time allocation. A power-failure interrupt may have as its service program a few instructions that transfer the complete contents of the register set of the processor into a nondestructive memory such as a disk in the few milliseconds before power ceases.

Internal interrupts arise from the invalid or erroneous use of an instruction or data. Internal interrupts are also called *traps*. Examples of interrupts caused by internal conditions are an arithmetic overflow, an attempt to divide by zero, an invalid opcode, a memory stack overflow, and a protection violation. A *protection violation* is an attempt to address an area of memory that is not supposed to be accessed by the currently executing program. The service programs that process internal interrupts determine the corrective measure to be taken in each case.

External and internal interrupts are initiated by the hardware of the computer. By contrast, a *software interrupt* is initiated by executing an instruction. The software interrupt is a special call instruction that behaves like an interrupt rather than a procedure call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. Typical use of the software interrupt is associated

with a system call instruction. This instruction provides a means for switching from user mode to system mode. Certain operations in the computer may be performed by the operating system only in system mode. For example, a complex input or output procedure is done in system mode. In contrast, a program written by a user must run in user mode. When an input or output transfer is required, the user program causes a software interrupt, which stores the contents of the *PSR* (with the mode bit set to "user"), loads new *PSR* contents (with the mode bit set to "system"), and initiates the execution of a system program. The calling program must pass information to the operating system in order to specify the particular task that is being requested.

An alternative term for an interrupt is an *exception*, which may apply only to internal interrupts or to all interrupts, depending on the particular computer manufacturer. As an illustration of the use of the two terms, what one programmer calls interrupt-handling routines may be referred to as exception-handling routines by another programmer.

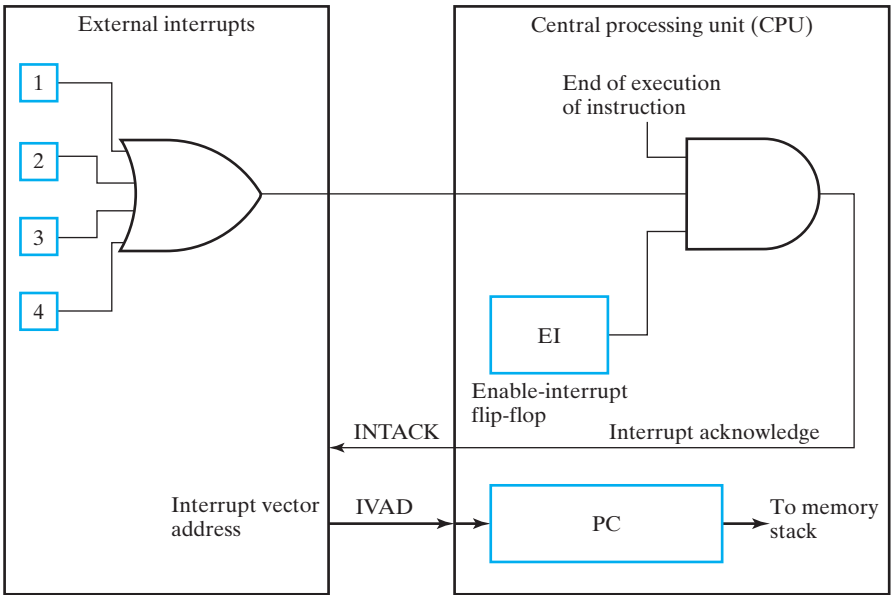
Processing External Interrupts

External interrupts may have single or multiple interrupt input lines. If there are more interrupt sources than there are interrupt inputs in the computer, two or more sources are ORed to form a common line. An interrupt signal may originate at any time during program execution. To ensure that no information is lost, the computer usually acknowledges the interrupt only after the execution of the current instruction is completed and only if the state of the processor warrants it.

Figure 9-9 shows a simplified external interrupt configuration. Four external interrupt sources are ORed to form a single interrupt input signal. Within the CPU is an enable-interrupt flip-flop (*EI*) that can be set or reset with two program instructions: enable interrupt (ENI) and disable interrupt (DSI). When *EI* is 0, the interrupt signal is neglected. When *EI* is 1 and the CPU is at the end of executing an instruction, the computer acknowledges the interrupt by enabling the interrupt acknowledge output *INTACK*. The interrupt source responds to *INTACK* by providing an interrupt vector address *IVAD* to the CPU. The program-controlled *EI* flip-flop allows the programmer to decide whether to use the interrupt facility. If a DSI instruction to reset *EI* has been inserted in the program, it means that the programmer does not want the program to be interrupted. The execution of an ENI instruction to set *EI* indicates that the interrupt facility will be active while the program is running.

The computer responds to an interrupt request signal if $EI = 1$ and execution of the present instruction is completed. Typical microinstructions that implement the interrupt are as follows:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PSR$	Store processor status word on stack
$EI \leftarrow 0$	Reset enable – interrupt flip – flop
$INTACK \leftarrow 1$	Enable interrupt acknowledge
$PC \leftarrow IVAD$	Transfer interrupt vector address to *PC*
	Go to fetch phase.



□ **FIGURE 9-9**
Example of External Interrupt Configuration

The return address available in the *PC* is pushed onto the stack, and the *PSR* contents are pushed onto the stack. *EI* is reset to disable further interrupts. The program that services the interrupt can set *EI* with an instruction whenever it is appropriate to enable other interrupts. The CPU assumes that the external source will provide an *IVAD* in response to an *INTACK*. The *IVAD* is taken as the address of the first instruction of the program that services the interrupt. Obviously, a program must be written for that purpose and stored in memory.

The return from an interrupt is done with an instruction at the end of the service program that is similar to a return from a procedure. The stack is popped, and the return address is transferred to the *PC*. Since the *EI* flip-flop is usually included in the *PSR*, the value of *EI* for the original program is returned to *EI* when the old value of the *PSR* is returned. Thus, the interrupt system is enabled or disabled for the original program, as it was before the interrupt occurred.

9-10 CHAPTER SUMMARY

In this chapter, we defined the concepts of instruction set architecture and the components of an instruction and explored the effects on programs of the maximum address count per instruction, using both memory addresses and register addresses. This led to the definitions of four types of addressing architecture: memory-to-memory, register-to-register, single-accumulator, and stack. Addressing modes specify how the information in an instruction is interpreted in determining the effective address of an operand.

Reduced instruction set computers (RISCs) and complex instruction set computers (CISCs) are two broad categories of instruction set architecture. A RISC has

as its goals high throughput and fast execution of instructions. In contrast, a CISC attempts to closely match the operations used in programming languages and facilitates more compact programs.

Three categories of elementary instructions are data transfer, data manipulation, and program control. In elaborating data transfer instructions, the concept of the memory stack appears. Transfers between the CPU and I/O are addressed by two different methods: independent I/O, with a separate address space, and memory-mapped I/O, which uses part of the memory address space. Data manipulation instructions fall into three classes: arithmetic, logical, and shift. Floating-point formats and operations handle broader ranges of operand values for arithmetic operations.

Program control instructions include basic unconditional and conditional transfers of control, the latter may or may not use condition codes. Procedure calls and returns permit programs to be broken up into procedures that perform useful tasks. Interruption of the normal sequence of program execution is based on three types of interrupts: external, internal, and software. Also referred to as exceptions, interrupts require special processing actions upon the initiation of routines to service them and upon returns to execution of the interrupted programs.

References

1. Hennessy, J. L. and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 5th ed. Amsterdam: Elsevier, 2011.
2. *IEEE Standard for Microprocessor Assembly Language* (IEEE Std 694-1985). New York: The Institute of Electrical and Electronics Engineers.
3. *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985). New York: The Institute of Electrical and Electronics Engineers.
4. *The Intel 64 and IA-32 Architectures Software Developer's Manual*, Vols. 2A and 2B. Intel Corporation, 1997–2006.
5. Mano, M. M. *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.
6. Mano, M. M. *Computer System Architecture*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.
7. Patterson, D. A. and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Amsterdam: Elsevier, 2013.

Problems

The plus (+) indicates a more advanced problem and the asterisk (*) indicates that a solution is available on the Companion Website for the text.

9-1. Based on operations illustrated in Section 9-2, write a program to evaluate the arithmetic expression

$$X = (A + B - C) \times (D - E)$$