Part Four The Central Processing Unit

Chapter 12 Instruction Sets: Characteristics and Functions

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of essential characteristics of **machine instructions**.
- Describe the types of operands used in typical machine instruction sets.
- Present an overview of x86 and ARM data types.
- Describe the types of operands supported by typical machine instruction sets.
- Present an overview of x86 and ARM operation types.
- Understand the differences among **big endian**, **little endian**, and **bi-endian**.

Much of what is discussed in this book is not readily apparent to the user or programmer of a computer. If a programmer is using a high-level language, such as Pascal or Ada, very little of the architecture of the underlying machine is visible.

One boundary where the computer designer and the computer programmer can view the same machine is the machine instruction set. From the designer's point of view, the machine instruction set provides the functional requirements for the processor: implementing the processor is a task that in large part involves implementing the machine instruction set. The user who chooses to program in machine language (actually, in assembly language; see Appendix B) becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

A description of a computer's machine instruction set goes a long way toward explaining the computer's processor. Accordingly, we focus on machine instructions in this chapter and the next.

12.1 MACHINE INSTRUCTION CHARACTERISTICS

The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions*. The collection of different instructions that the processor can execute is referred to as the processor's *instruction set.*

Elements of a Machine Instruction

Each instruction must contain the information required by the processor for execution. Figure 12.1, which repeats Figure 3.6, shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction. These elements are as follows:

- **Operation code:** Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or **opcode**.
- **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.

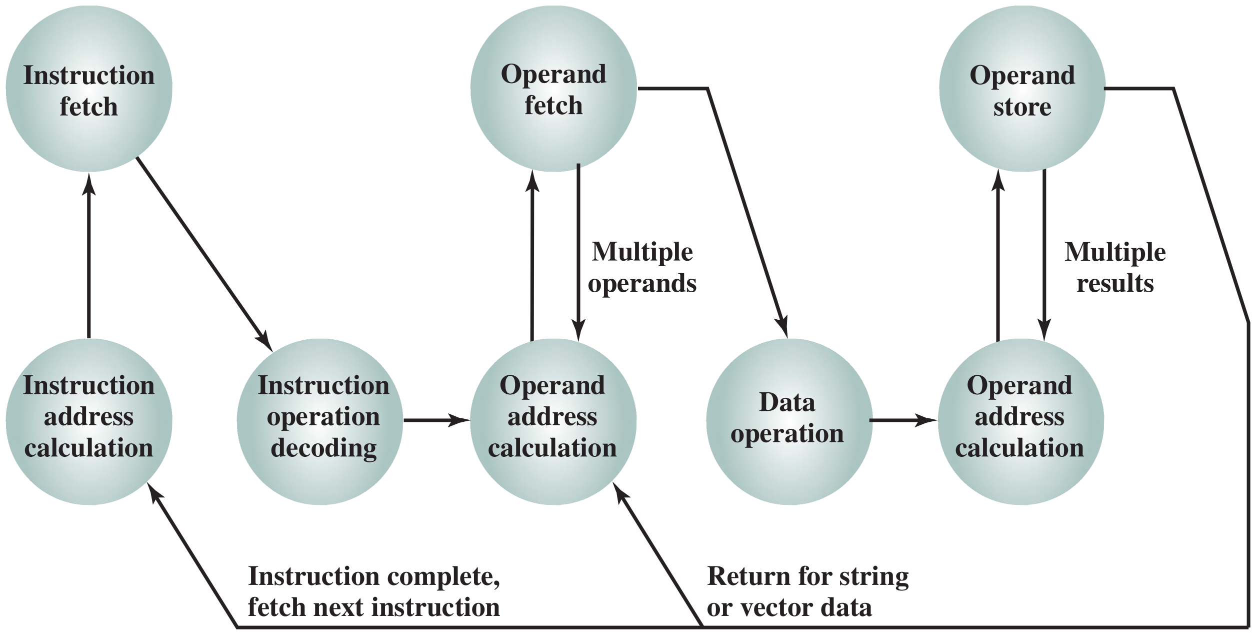

Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

Figure 12.1 Instruction Cycle State Diagram

- **Result operand reference:** The operation may produce a result.
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

The address of the next instruction to be fetched could be either a real address or a virtual address, depending on the architecture. Generally, the distinction is transparent to the instruction set architecture. In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, the main memory or virtual memory address must be supplied. The form in which that address is supplied is discussed in Chapter 13.

Source and result operands can be in one of four areas:

- **Main or virtual memory:** As with next instruction references, the main or virtual memory address must be supplied.
- **Processor register:** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.
- **Immediate:** The value of the operand is contained in a field in the instruction being executed.
- **I/O device:** The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address.

Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the

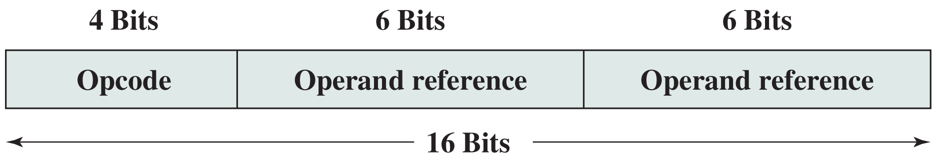


Figure 12.2 A Simple Instruction Format

instruction. A simple example of an instruction format is shown in Figure 12.2. As another example, the IAS instruction format is shown in Figure 2.2. With most instruction sets, more than one format is used. During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a *symbolic representation* of machine instructions. An example of this was used for the IAS instruction set, in Table 1.1.

Opcodes are represented by abbreviations, called *mnemonics,* that indicate the operation. Common examples include

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

Operands are also represented symbolically. For example, the instruction

ADD R, Y

may mean add the value contained in data location Y to the contents of register R. In this example, Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Thus, it is possible to write a machine-language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand. For example, the programmer might begin with a list of definitions:

$$X = 513$$
$$Y = 514$$

and so on. A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Machine-language programmers are rare to the point of nonexistence. Most programs today are written in a high-level language or, failing that, assembly language, which is discussed in Appendix B. However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

$$X = X + Y$$

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions. This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.

With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task. Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language. With this in mind we can categorize instruction types as follows:

- **Data processing:** Arithmetic and logic instructions.
- **Data storage:** Movement of data into or out of register and or memory locations.
- **Data movement:** I/O instructions.
- **Control:** Test and branch instructions.

Arithmetic instructions provide computational capabilities for processing numeric data. *Logic* (Boolean) instructions operate on the bits of a word as bits rather than as numbers; thus, they provide capabilities for processing any other type of data the user may wish to employ. These operations are performed primarily on data in processor registers. Therefore, there must be *memory* instructions for moving data between memory and the registers. *I/O* instructions are needed to transfer programs and data into memory and the results of computations back out to the user. *Test* instructions are used to test the value of a data word or the status of a computation. *Branch* instructions are then used to branch to a different set of instructions depending on the decision made.

We will examine the various types of instructions in greater detail later in this chapter.

Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction.

What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). Thus, we would need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could plausibly be required to contain four address references: two source operands, one destination operand, and the address of the next instruction. In most architectures, many instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter). Most architectures also have a few special-purpose instructions with more operands. For example, the load and store multiple instructions of the ARM architecture, described in Chapter 13, designate up to 17 register operands in a single instruction.

Figure 12.3 compares typical one-, two-, and three-address instructions that could be used to compute Y = (A − B)/[C + (D × E)]. With three addresses, each instruction specifies two source operand locations and a destination operand location. Because we choose not to alter the value of any of the operand locations,

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Figure 12.3 Programs to Execute $Y = \dfrac{A - B}{C + (D \times E)}$

a temporary location, T, is used to store some intermediate results. Note that there are four instructions and that the original expression had five operands.

Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references. With two-address instructions, and for binary operations, one address must do double duty as both an operand and a result. Thus, the instruction SUB Y, B carries out the calculation $Y - B$ and stores the result in Y. The two-address format reduces the space requirement but also introduces some awkwardness. To avoid altering the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation. Our sample program expands to six instructions.

Simpler yet is the one-address instruction. For this to work, a second address must be implicit. This was common in earlier machines, with the implied address being a processor register known as the **accumulator** (AC). The accumulator contains one of the operands and is used to store the result. In our example, eight instructions are needed to accomplish the task.

It is, in fact, possible to make do with zero addresses for some instructions. Zero-address instructions are applicable to a special memory organization called a stack. A stack is a last-in-first-out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers. Thus, zero-address instructions would reference the top two stack elements. Stacks are described in Appendix I. Their use is explored further later in this chapter and in Chapter 13.

Table 12.1 summarizes the interpretations to be placed on instructions with zero, one, two, or three addresses. In each case in the table, it is assumed that the address of the next instruction is implicit, and that one operation with two source operands and one result operand is to be performed.

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in instructions that are more primitive, requiring a less complex processor. It also results in instructions of shorter length. On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs. Also, there is an important threshold between one-address and multiple-address instructions. With one-address instructions, the programmer generally has available only one general-purpose

Table 12.1 Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	$A \leftarrow B$ OP C
2	OP A, B	$A \leftarrow A$ OP B
1	OP A	$AC \leftarrow AC$ OP A
0	OP	$T \leftarrow (T - 1)$ OP T

AC = accumulator
T = top of stack
$(T - 1)$ = second element of stack
A, B, C = memory or register locations

register, the accumulator. With multiple-address instructions, it is common to have multiple general-purpose registers. This allows some operations to be performed solely on registers. Because register references are faster than memory references, this speeds up execution. For reasons of flexibility and ability to use multiple registers, most contemporary machines employ a mixture of two- and three-address instructions.

The design trade-offs involved in choosing the number of addresses per instruction are complicated by other factors. There is the issue of whether an address references a memory location or a register. Because there are fewer registers, fewer bits are needed for a register reference. Also, as we will see in Chapter 13, a machine may offer a variety of addressing modes, and the specification of mode takes one or more bits. The result is that most processor designs involve a variety of instruction formats.

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The design of an instruction set is very complex because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the processor and thus has a significant effect on the implementation of the processor. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

It may surprise you to know that some of the most fundamental issues relating to the design of instruction sets remain in dispute. Indeed, in recent years, the level of disagreement concerning these fundamentals has actually grown. The most important of these fundamental design issues include the following:

- **Operation repertoire:** How many and which operations to provide, and how complex operations should be.
- **Data types:** The various types of data upon which operations are performed.
- **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on.
- **Registers:** Number of processor registers that can be referenced by instructions, and their use.
- **Addressing:** The mode or modes by which the address of an operand is specified.

These issues are highly interrelated and must be considered together in designing an instruction set. This book, of course, must consider them in some sequence, but an attempt is made to show the interrelationships.

Because of the importance of this topic, much of Part Three is devoted to instruction set design. Following this overview section, this chapter examines data types and operation repertoire. Chapter 13 examines addressing modes (which includes a consideration of registers) and instruction formats. Chapter 15 examines the reduced instruction set computer (RISC). RISC architecture calls into question many of the instruction set design decisions traditionally made in commercial computers.

12.2 TYPES OF OPERANDS

Machine instructions operate on data. The most important general categories of data are

- Addresses
- Numbers
- Characters
- Logical data

We shall see, in discussing addressing modes in Chapter 13, that addresses are, in fact, a form of data. In many cases, some calculation must be performed on the operand reference in an instruction to determine the main or virtual memory address. In this context, addresses can be considered to be unsigned integers.

Other common data types are numbers, characters, and logical data, and each of these is briefly examined in this section. Beyond that, some machines define specialized data types or data structures. For example, there may be machine operations that operate directly on a list or a string of characters.

Numbers

All machine languages include numeric data types. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. This is true in two senses. First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating-point numbers, a limit to their precision. Thus, the programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are common in computers:

- Binary integer or binary fixed point
- Binary floating point
- Decimal

We examined the first two in some detail in Chapter 10. It remains to say a few words about decimal numbers.

Although all internal computer operations are binary in nature, the human users of the system deal with decimal numbers. Thus, there is a necessity to convert from decimal to binary on input and from binary to decimal on output. For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is **packed decimal**.[1]

[1]Textbooks often refer to this as binary coded decimal (BCD). Strictly speaking, BCD refers to the encoding of each decimal digit by a unique 4-bit sequence. Packed decimal refers to the storage of BCD-encoded digits using one byte for each two digits.

With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, 0 = 000, 1 = 0001, . . . , 8 = 1000, and 9 = 1001. Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used. To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits. Thus, the code for 246 is 0000 0010 0100 0110. This code is clearly less compact than a straight binary representation, but it avoids the conversion overhead. Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits. Standard sign values are 1100 for positive (+) and 1101 for negative (−).

Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers. The algorithms are quite similar to those described in Section 9.3 but must take into account the decimal carry operation.

Characters

A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used character code in the International Reference Alphabet (IRA), referred to in the United States as the American Standard Code for Information Interchange (ASCII; see Appendix H). Each character in this code is represented by a unique 7-bit pattern; thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent *control* characters. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Note in Table H.1 (Appendix H) that for the IRA bit pattern 011XXXX, the digits 0 through 9 are represented by their binary equivalents, 0000 through 1001, in the rightmost 4 bits. This is the same code as packed decimal. This facilitates conversion between 7-bit IRA and 4-bit packed decimal representation.

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is used on IBM mainframes. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

Logical Data

Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be *logical* data.

There are two advantages to the bit-oriented view. First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false). With logical data, memory can be used most efficiently for this storage. Second, there are occasions when we wish to manipulate the bits of a data item. For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations. Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

Note that, in the preceding examples, the same data are treated sometimes as logical and other times as numerical or text. The "type" of a unit of data is determined by the operation being performed on it. While this is not normally the case in high-level languages, it is almost always the case with machine language.

12.3 INTEL x86 AND ARM DATA TYPES

x86 Data Types

The x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quadword), and 128 (double quadword) bits in length. To allow maximum flexibility in data structures and efficient memory utilization, words need not be aligned at even-numbered addresses; doublewords need not be aligned at addresses evenly divisible by 4; quadwords need not be aligned at addresses evenly divisible by 8; and so on. However, when data are accessed across a 32-bit bus, data transfers take place in units of doublewords, beginning at addresses divisible by 4. The processor converts the request for misaligned values into a sequence of requests for the bus transfer. As with all of the Intel 80x86 machines, the x86 uses the little-endian style; that is, the least significant byte is stored in the lowest address (see Appendix 12A for a discussion of endianness).

The byte, word, doubleword, quadword, and double quadword are referred to as general data types. In addition, the x86 supports an impressive array of specific data types that are recognized and operated on by particular instructions. Table 12.2 summarizes these types.

Figure 12.4 illustrates the x86 numerical data types. The signed integers are in twos complement representation and may be 16, 32, or 64 bits long. The floating-point type actually refers to a set of types that are used by the floating-point unit and operated on by floating-point instructions. The floating-point representations conform to the IEEE 754 standard.

The packed SIMD (single-instruction-multiple-data) data types were introduced to the x86 architecture as part of the extensions of the instruction set to optimize performance of multimedia applications. These extensions include MMX (multimedia extensions) and SSE (streaming SIMD extensions). The basic concept is that multiple operands are packed into a single referenced memory item and that these multiple operands are operated on in parallel. The data types are as follows:

- **Packed byte and packed byte integer:** Bytes packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.
- **Packed word and packed word integer:** 16-bit words packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.

Table 12.2 x86 Data Types

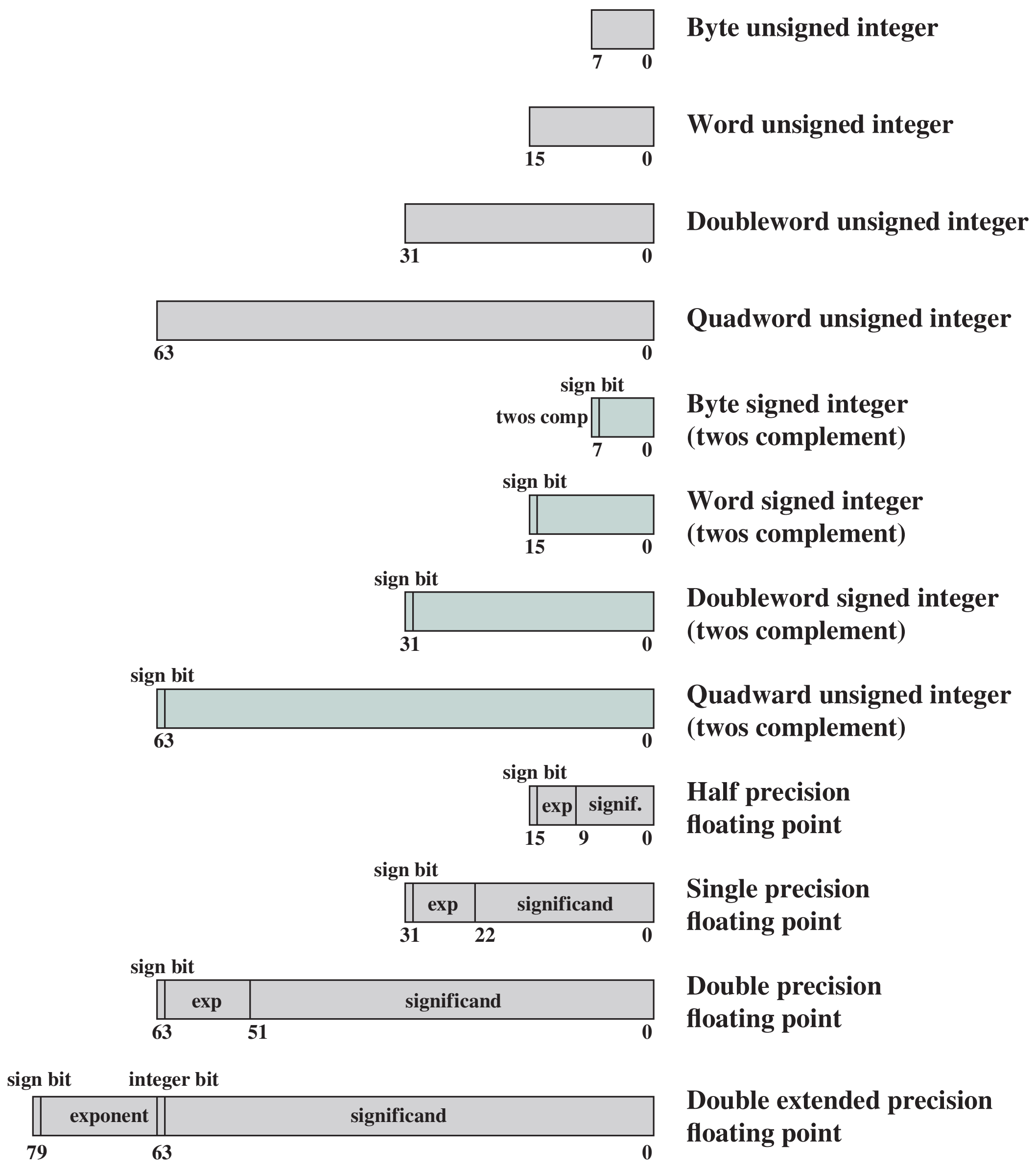
Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2^{23} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{23} - 1$ bytes.
Floating point	See Figure 12.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types.

- **Packed doubleword and packed doubleword integer:** 32-bit doublewords packed into a 64-bit quadword or 128-bit double quadword, interpreted as a bit field or as an integer.
- **Packed quadword and packed quadword integer:** Two 64-bit quadwords packed into a 128-bit double quadword, interpreted as a bit field or as an integer.
- **Packed single-precision floating-point and packed double-precision floating-point:** Four 32-bit floating-point or two 64-bit floating-point values packed into a 128-bit double quadword.

ARM Data Types

ARM processors support data types of 8 (byte), 16 (halfword), and 32 (word) bits in length. Normally, halfword access should be halfword aligned and word accesses should be word aligned. For nonaligned access attempts, the architecture supports three alternatives.

- Default case:
  – The address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.



Byte unsigned integer
7 0
Word unsigned integer
15 0
Doubleword unsigned integer
31 0
Quadword unsigned integer
63 0
sign bit
twos comp
Byte signed integer (twos complement)
7 0
sign bit
Word signed integer (twos complement)
15 0
sign bit
Doubleword signed integer (twos complement)
31 0
sign bit
Quadward unsigned integer (twos complement)
63 0
sign bit
exp
signif.
Half precision floating point
15 9 0
sign bit
exp
significand
Single precision floating point
31 22 0
sign bit
exp
significand
Double precision floating point
63 51 0
sign bit
integer bit
exponent
significand
Double extended precision floating point
79 63 0

Figure 12.4 x86 Numeric Data Formats

– Load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two, or three bytes depending on the value of the two least significant address bits.

- **Alignment checking:** When the appropriate control bit is set, a data abort signal indicates an alignment fault for attempting unaligned access.
- **Unaligned access:** When this option is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer.

For all three data types (byte, halfword, and word) an unsigned interpretation is supported, in which the value represents an unsigned, nonnegative integer. All three data types can also be used for twos complement signed integers.

The majority of ARM processor implementations do not provide floating-point hardware, which saves power and area. If floating-point arithmetic is required in such processors, it must be implemented in software. ARM does support an optional floating-point coprocessor that supports the single- and double-precision floating point data types defined in IEEE 754.

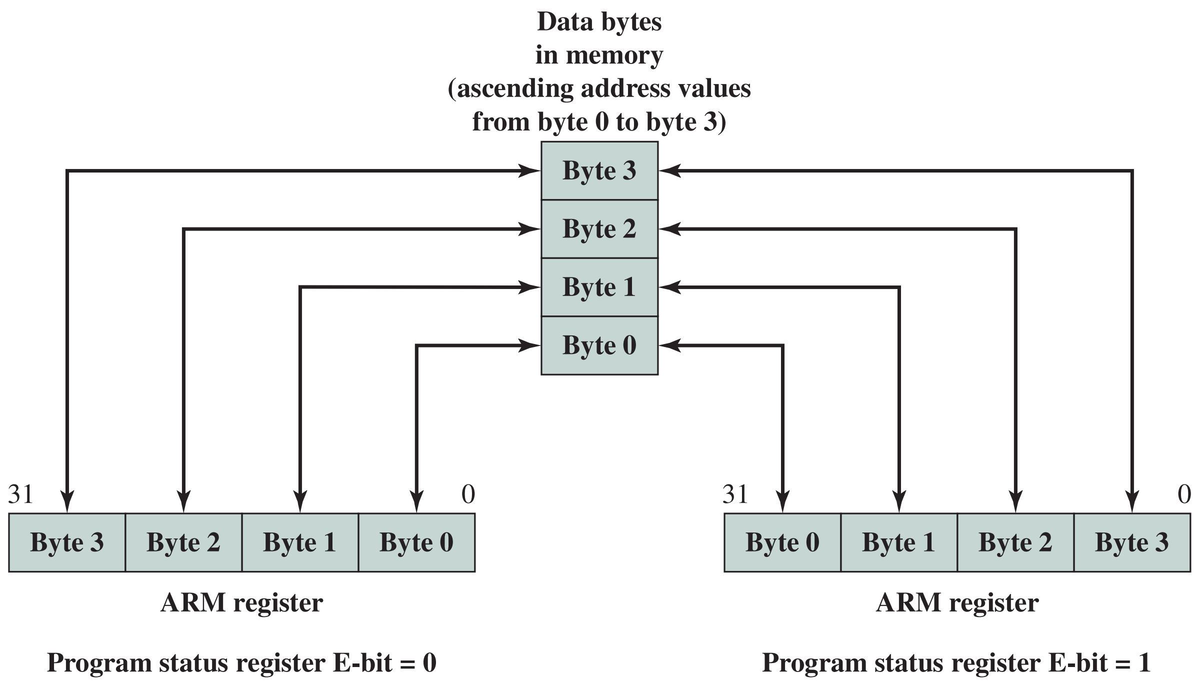


Figure 12.5 ARM Endian Support—Word Load/Store with E-Bit

ENDIAN SUPPORT A state bit (E-bit) in the system control register is set and cleared under program control using the SETEND instruction. The E-bit defines which endian to load and store data. Figure 12.5 illustrates the functionality associated with the E-bit for a word load or store operation. This mechanism enables efficient dynamic data load/store for system designers who know they need to access data structures in the opposite endianness to their OS/environment. Note that the address of each data byte is fixed in memory. However, the byte lane in a register is different.

12.4 TYPES OF OPERATIONS

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

Table 12.3 (based on [HAYE98]) lists common instruction types in each category. This section provides a brief survey of these various types of operations, together with a brief discussion of the actions taken by the processor to execute a particular type of operation (summarized in Table 12.4). The latter topic is examined in more detail in Chapter 14.

Table 12.3 Common Instruction Set Operations

Type	Operation Name	Description
Data transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT	(complement) Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

Type	Operation Name	Description
Input/output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

Table 12.4 Processor Actions for Various Types of Operations

Data transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Data Transfer

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things. First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands, the mode of addressing for each operand must be specified. This latter point is discussed in Chapter 13.

The choice of data transfer instructions to include in an instruction set exemplifies the kinds of trade-offs the designer must make. For example, the general location (memory or register) of an operand can be indicated in either the specification of the opcode or the operand. Table 12.5 shows examples of the most common IBM EAS/390 data transfer instructions. Note that there are variants to indicate

Table 12.5 Examples of IBM EAS/390 Data Transfer Operations

Operation Mnemonic	Name	Number of Bits Transferred	Description
L	Load	32	Transfer from memory to register
LH	Load Halfword	16	Transfer from memory to register
LR	Load	32	Transfer from register to register
LER	Load (short)	32	Transfer from floating-point register to floating-point register
LE	Load (short)	32	Transfer from memory to floating-point register
LDR	Load (long)	64	Transfer from floating-point register to floating-point register
LD	Load (long)	64	Transfer from memory to floating-point register
ST	Store	32	Transfer from register to memory
STH	Store Halfword	16	Transfer from register to memory
STC	Store Character	8	Transfer from register to memory
STE	Store (short)	32	Transfer from floating-point register to memory
STD	Store (long)	64	Transfer from floating-point register to memory

the amount of data to be transferred (8, 16, 32, or 64 bits). Also, there are different instructions for register to register, register to memory, memory to register, and memory to memory transfers. In contrast, the VAX has a move (MOV) instruction with variants for different amounts of data to be moved, but it specifies whether an operand is register or memory as part of the operand. The VAX approach is somewhat easier for the programmer, who has fewer mnemonics to deal with. However, it is also somewhat less compact than the IBM EAS/390 approach because the location (register versus memory) of each operand must be specified separately in the instruction. We will return to this distinction when we discuss instruction formats in Chapter 13.

In terms of processor action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the processor simply causes data to be transferred from one register to another; this is an operation internal to the processor. If one or both operands are in memory, then the processor must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode (discussed in Chapter 13).
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers. Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions; for example,

- **Absolute:** Take the absolute value of the operand.
- **Negate:** Negate the operand.
- **Increment:** Add 1 to the operand.
- **Decrement:** Subtract 1 from the operand.

The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU. Figure 3.5 illustrates the movements involved in both data transfer and arithmetic operations. In addition, of course, the ALU portion of the processor performs the desired operation.

Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as "bit twiddling." They are based upon Boolean operations (see Chapter 11).

Some of the basic logical operations that can be performed on Boolean or binary data are shown in Table 12.6. The NOT operation inverts a bit. AND, OR, and Exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test.

These logical operations can be applied bitwise to *n*-bit logical data units. Thus, if two registers contain the data

$$(R1) = 10100101$$

$$(R2) = 00001111$$

then

$$(R1) \text{ AND } (R2) = 00000101$$

Table 12.6 Basic Logical Operations

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P = Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

where the notation (X) means the contents of location X. Thus, the AND operation can be used as a *mask* that selects certain bits in a word and zeros out the remaining bits. As another example, if two registers contain

$$(R1) = 10100101$$
$$(R2) = 11111111$$

then

$$(R1)\ \text{XOR}\ (R2) = 01011010$$

With one word set to all 1s, the XOR operation inverts all of the bits in the other word (ones complement).

In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in Figure 12.6. With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out is lost. On the other end, a 0 is shifted in. Logical shifts are useful primarily for isolating fields within a word. The 0s that are shifted into a word displace unwanted information that is shifted off the other end.

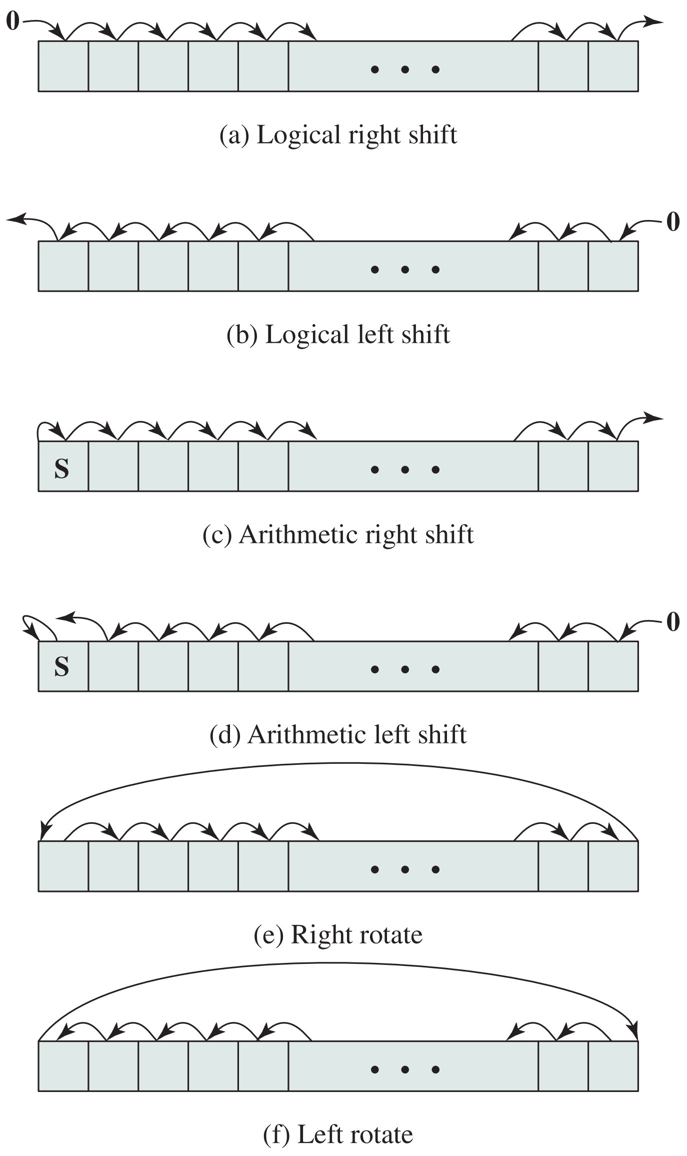


Figure 12.6 Shift and Rotate Operations

As an example, suppose we wish to transmit characters of data to an I/O device 1 character at a time. If each memory word is 16 bits in length and contains two characters, we must *unpack* the characters before they can be sent. To send the two characters in a word;

1. Load the word into a register.
2. Shift to the right eight times. This shifts the remaining character to the right half of the register.
3. Perform I/O. The I/O module reads the lower-order 8 bits from the data bus.

The preceding steps result in sending the left-hand character. To send the right-hand character;

1. Load the word again into the register.
2. AND with 0000000011111111. This masks out the character on the left.
3. Perform I/O.

The **arithmetic shift** operation treats the data as a signed integer and does not shift the sign bit. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained. These operations can speed up certain arithmetic operations. With numbers in twos complement notation, a right arithmetic shift corresponds to a division by 2, with truncation for odd numbers. Both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow. If overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Because of the potential for overflow, many processors do not include this instruction, including PowerPC and Itanium. Others, such as the IBM EAS/390, do offer the instruction. Curiously, the x86 architecture includes an arithmetic left shift but defines it to be identical to a logical left shift.

Rotate, or cyclic shift, operations preserve all of the bits being operated on. One use of a rotate is to bring each bit successively into the leftmost bit, where it can be identified by testing the sign of the data (treated as a number).

As with arithmetic operations, logical operations involve ALU activity and may involve data transfer operations. Table 12.7 gives examples of all of the shift and rotate operations discussed in this subsection.

Table 12.7 Examples of Shift and Rotate Operations

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

Conversion

Conversion instructions are those that change the format or operate on the format of data. An example is converting from decimal to binary. An example of a more complex editing instruction is the EAS/390 Translate (TR) instruction. This instruction can be used to convert from one 8-bit code to another, and it takes three operands:

TR R1 (L), R2

The operand R2 contains the address of the start of a table of 8-bit codes. The L bytes starting at the address specified in R1 are translated, each byte being replaced by the contents of a table entry indexed by that byte. For example, to translate from EBCDIC to IRA, we first create a 256-byte table in storage locations, say, 1000-10FF hexadecimal. The table contains the characters of the IRA code in the sequence of the binary representation of the EBCDIC code; that is, the IRA code is placed in the table at the relative location equal to the binary value of the EBCDIC code of the same character. Thus, locations 10F0 through 10F9 will contain the values 30 through 39, because F0 is the EBCDIC code for the digit 0, and 30 is the IRA code for the digit 0, and so on through digit 9. Now suppose we have the EBCDIC for the digits 1984 starting at location 2100 and we wish to translate to IRA. Assume the following:

- Locations 2100–2103 contain F1 F9 F8 F4.
- R1 contains 2100.
- R2 contains 1000.

Then, if we execute

TR R1 (4), R2

locations 2100–2103 will contain 31 39 38 34.

Input/Output

Input/output instructions were discussed in some detail in Chapter 7. As we saw, there are a variety of approaches taken, including isolated programmed I/O, memory-mapped programmed I/O, DMA, and the use of an I/O processor. Many implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words.

System Control

System control instructions are those that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the use of the operating system.

Some examples of system control operations are as follows. A system control instruction may read or alter a control register; we discuss control registers in Chapter 14. Another example is an instruction to read or modify a storage protection key, such as is used in the EAS/390 memory system. Yet another example is access to process control blocks in a multiprogramming system.

Transfer of Control

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

There are a number of reasons why transfer-of-control operations are required. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.
2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds. For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested. If the number is negative, the computation is not performed, but an error condition is reported.
3. To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

We now turn to a discussion of the most common transfer-of-control operations found in instruction sets: branch, **skip**, and **procedure call**.

BRANCH INSTRUCTIONS A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed. Most often, the instruction is a **conditional branch** instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual). A branch instruction in which the branch is always taken is an **unconditional branch**.

There are two common ways of generating the condition to be tested in a conditional branch instruction. First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. This code can be thought of as a short user-visible register. As an example, an arithmetic operation (ADD, SUBTRACT, and so on) could set a 2-bit condition code with one of the following four values: 0, positive, negative, overflow. On such a machine, there could be four different conditional branch instructions:

BRP X	Branch to location X if result is positive.
BRN X	Branch to location X if result is negative.
BRZ X	Branch to location X if result is zero.
BRO X	Branch to location X if overflow occurs.

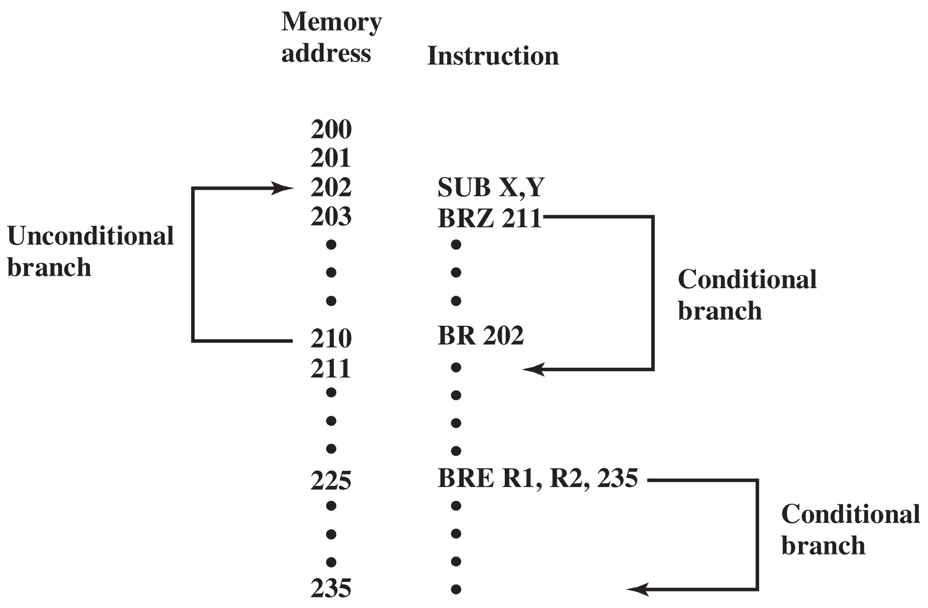


Figure 12.7 Branch Instructions

In all of these cases, the result referred to is the result of the most recent operation that set the condition code.

Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction. For example,

BRE R1, R2, X Branch to X if contents of R1 = contents of R2.

Figure 12.7 shows examples of these operations. Note that a branch can be either *forward* (an instruction with a higher address) or *backward* (lower address). The example shows how an unconditional and a conditional branch can be used to create a repeating loop of instructions. The instructions in locations 202 through 210 will be executed repeatedly until the result of subtracting Y from X is 0.

SKIP INSTRUCTIONS Another form of transfer-of-control instruction is the skip instruction. The skip instruction includes an implied address. Typically, the skip implies that one instruction be skipped; thus, the implied address equals the address of the next instruction plus one instruction length. Because the skip instruction does not require a destination address field, it is free to do other things. A typical example is the increment-and-skip-if-zero (ISZ) instruction. Consider the following program fragment:

```
301
 .
 .
 .
309  ISZ  R1
310  BR   301
311
```

In this fragment, the two transfer-of-control instructions are used to implement an iterative loop. R1 is set with the negative of the number of iterations to be performed. At the end of the loop, R1 is incremented. If it is not 0, the program branches back to the beginning of the loop. Otherwise, the branch is skipped, and the program continues with the next instruction after the end of the loop.

PROCEDURE CALL INSTRUCTIONS Perhaps the most important innovation in the development of programming languages is the *procedure.* A procedure is a self-contained computer program that is incorporated into a larger program. At any point in the program the procedure may be invoked, or *called.* The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The two principal reasons for the use of procedures are economy and modularity. A procedure allows the same piece of code to be used many times. This is important for economy in programming effort and for making the most efficient use of storage space in the system (the program must be stored). Procedures also allow large programming tasks to be subdivided into smaller units. This use of *modularity* greatly eases the programming task.

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

Figure 12.8a illustrates the use of procedures to construct a program. In this example, there is a main program starting at location 4000. This program includes a call to procedure PROC1, starting at location 4500. When this call instruction is encountered, the processor suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location 4500. Within PROC1, there are two calls to PROC2 at location 4800. In each case, the execution of PROC1

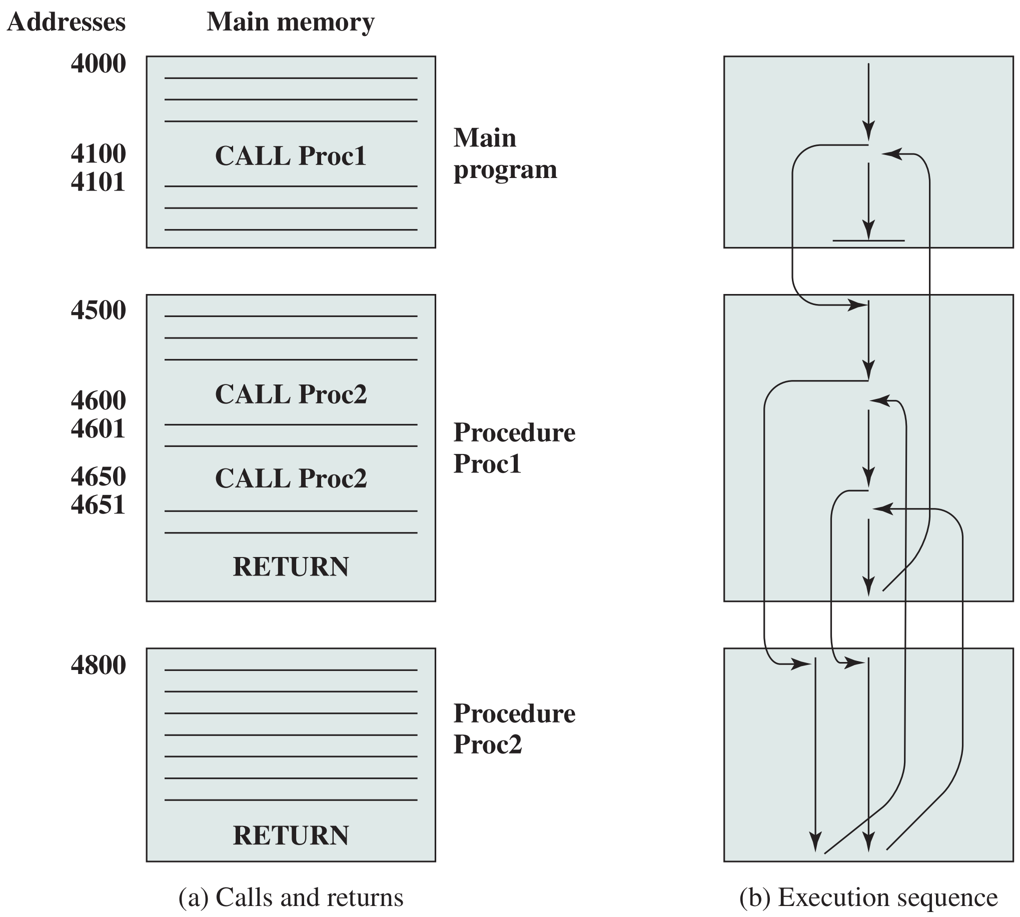


Figure 12.8 Nested Procedures

is suspended and PROC2 is executed. The RETURN statement causes the processor to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction. This behavior is illustrated in Figure 12.8b.

Three points are worth noting:

1. A procedure can be called from more than one location.
2. A procedure call can appear in a procedure. This allows the *nesting* of procedures to an arbitrary depth.
3. Each procedure call is matched by a return in the called program.

Because we would like to be able to call a procedure from a variety of points, the processor must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

- Register
- Start of called procedure
- Top of stack

Consider a machine-language instruction CALL X, which stands for *call procedure at location X*. If the register approach is used, CALL X causes the following actions:

$$\begin{aligned} RN &\leftarrow PC + \Delta \\ PC &\leftarrow X \end{aligned}$$

where RN is a register that is always used for this purpose, PC is the program counter, and Δ is the instruction length. The called procedure can now save the contents of RN to be used for the later return.

A second possibility is to store the return address at the start of the procedure. In this case, CALL X causes

$$\begin{aligned} X &\leftarrow PC + \Delta \\ PC &\leftarrow X + 1 \end{aligned}$$

This is quite handy. The return address has been stored safely away.

Both of the preceding approaches work and have been used. The only limitation of these approaches is that they complicate the use of *reentrant* procedures. A **reentrant procedure** is one in which it is possible to have several calls open to it at the same time. A recursive procedure (one that calls itself) is an example of the use of this feature (see Appendix M). If parameters are passed via registers or memory for a reentrant procedure, some code must be responsible for saving the parameters so that the registers or memory space are available for other procedure calls.

A more general and powerful approach is to use a stack (see Appendix I for a discussion of stacks). When the processor executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack. Figure 12.9 illustrates the use of the stack.

In addition to providing a return address, it is also often necessary to pass parameters with a procedure call. These can be passed in registers. Another possibility is to store the parameters in memory just after the CALL instruction. In this case, the return must be to the location following the parameters. Again, both of

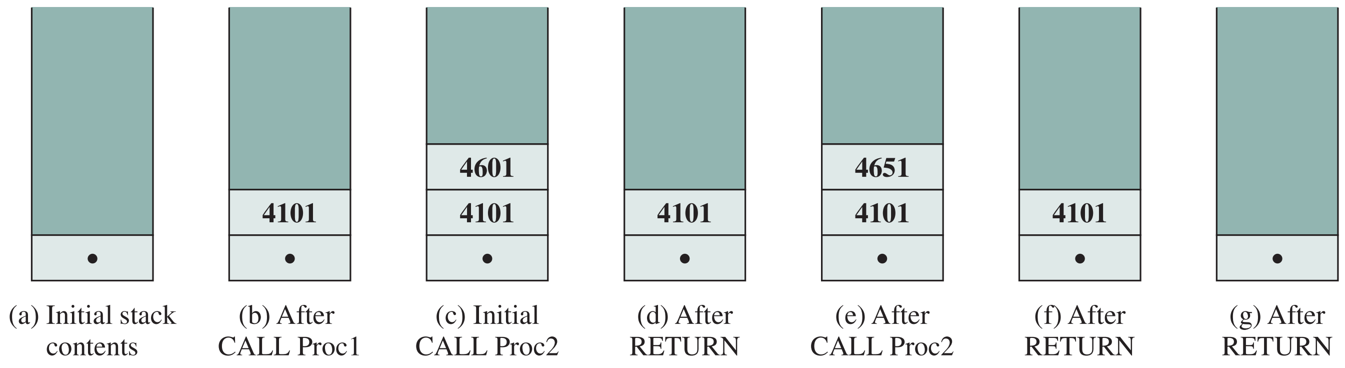


Figure 12.9 Use of Stack to Implement Nested Subroutines of Figure 12.8

these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of parameters in memory makes it difficult to exchange a variable number of parameters. Both approaches prevent the use of reentrant procedures.

A more flexible approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure. The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a *stack frame.*

An example is provided in Figure 12.10. The example refers to procedure P in which the local variables *x*1 and *x*2 are declared, and procedure Q, which P can call and in which the local variables *y*1 and *y*2 are declared. In this figure, the return

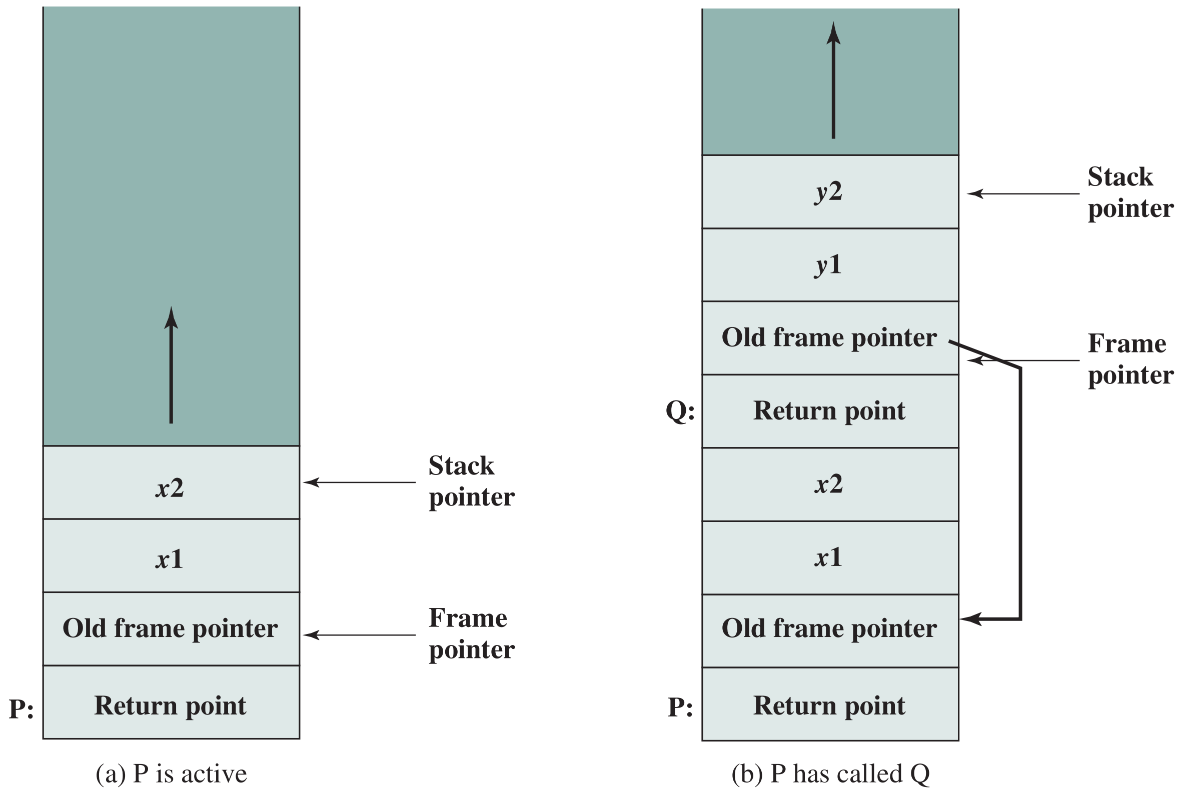


Figure 12.10 Stack Frame Growth Using Sample Procedures P and Q

point for each procedure is the first item stored in the corresponding stack frame. Next is stored a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable.

12.5 INTEL x86 AND ARM OPERATION TYPES

x86 Operation Types

The x86 provides a complex array of operation types, including a number of specialized instructions. The intent was to provide tools for the compiler writer to produce optimized machine language translation of high-level language programs. Most of these are the conventional instructions found in most machine instruction sets, but several types of instructions are tailored to the x86 architecture and are of particular interest. Appendix A of [CART06] lists the x86 instructions, together with the operands for each and the effect of the instruction on the condition codes. Appendix B of the NASM assembly language manual [NASM12] provides a more detailed description of each x86 instruction. Both documents are available at box.com/COA10e.

CALL/RETURN INSTRUCTIONS The x86 provides four instructions to support procedure call/return: CALL, ENTER, LEAVE, RETURN. It will be instructive to look at the support provided by these instructions. Recall from Figure 12.10 that a common means of implementing the procedure call/return mechanism is via the use of stack frames. When a new procedure is called, the following must be performed upon entry to the new procedure:

- Push the return point on the stack.
- Push the current frame pointer on the stack.
- Copy the stack pointer as the new value of the frame pointer.
- Adjust the stack pointer to allocate a frame.

The CALL instruction pushes the current instruction pointer value onto the stack and causes a jump to the entry point of the procedure by placing the address of the entry point in the instruction pointer. In the 8088 and 8086 machines, the typical procedure began with the sequence

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, space_for_locals
```

where EBP is the frame pointer and ESP is the stack pointer. In the 80286 and later machines, the ENTER instruction performs all the aforementioned operations in a single instruction.

The ENTER instruction was added to the instruction set to provide direct support for the compiler. The instruction also includes a feature for support of what are called nested procedures in languages such as Pascal, COBOL, and Ada (not found in C or FORTRAN). It turns out that there are better ways of handling nested procedure calls for these languages. Furthermore, although the ENTER instruction saves a few bytes of memory compared with the PUSH, MOV, SUB sequence (4 bytes versus 6 bytes), it actually takes longer to execute (10 clock cycles versus 6 clock cycles). Thus, although it may have seemed a good idea to the instruction set designers to add this feature, it complicates the implementation of the processor while providing little or no benefit. We will see that, in contrast, a RISC approach to processor design would avoid complex instructions such as ENTER and might produce a more efficient implementation with a sequence of simpler instructions.

MEMORY MANAGEMENT Another set of specialized instructions deals with memory segmentation. These are privileged instructions that can only be executed from the operating system. They allow local and global segment tables (called descriptor tables) to be loaded and read, and for the privilege level of a segment to be checked and altered.

The special instructions for dealing with the on-chip cache were discussed in Chapter 4.

STATUS FLAGS AND CONDITION CODES Status flags are bits in special registers that may be set by certain operations and used in conditional branch instructions. The term *condition code* refers to the settings of one or more status flags. In the x86 and many other architectures, status flags are set by arithmetic and compare operations. The compare operation in most languages subtracts two operands, as does a subtract operation. The difference is that a compare operation only sets status flags, whereas a subtract operation also stores the result of the subtraction in the destination operand. Some architectures also set status flags for data transfer instructions.

Table 12.8 lists the status flags used on the x86. Each flag, or combinations of these flags, can be tested for a conditional jump. Table 12.9 shows the condition codes (combinations of status flag values) for which conditional jump opcodes have been defined.

Several interesting observations can be made about this list. First, we may wish to test two operands to determine if one number is bigger than another. But this will depend on whether the numbers are signed or unsigned. For example, the 8-bit number 11111111 is bigger than 00000000 if the two numbers are interpreted

Table 12.8 x86 Status Flags

Status Bit	Name	Description
C	Carry	Indicates carrying or borrowing out of the left-most bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
P	Parity	Parity of the least-significant byte of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
A	Auxiliary Carry	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation. Used in binary-coded decimal arithmetic.
Z	Zero	Indicates that the result of an arithmetic or logic operation is 0.
S	Sign	Indicates the sign of the result of an arithmetic or logic operation.
O	Overflow	Indicates an arithmetic overflow after an addition or subtraction for twos complement arithmetic.

Table 12.9 x86 Condition Codes for Conditional Jump and SETcc Instructions

Symbol	Condition Tested	Comment
A, NBE	C = 0 AND Z = 0	Above; Not below or equal (greater than, unsigned)
AE, NB, NC	C = 0	Above or equal; Not below (greater than or equal, unsigned); Not carry
B, NAE, C	C = 1	Below; Not above or equal (less than, unsigned); Carry set
BE, NA	C = 1 OR Z = 1	Below or equal; Not above (less than or equal, unsigned)
E, Z	Z = 1	Equal; Zero (signed or unsigned)
G, NLE	[(S = 1 AND O = 1) OR (S = 0 AND O = 0)]AND[Z = 0]	Greater than; Not less than or equal (signed)
GE, NL	(S = 1 AND O = 1) OR (S = 0 AND O = 0)	Greater than or equal; Not less than (signed)
L, NGE	(S = 1 AND O = 0) OR (S = 0 AND O = 0)	Less than; Not greater than or equal (signed)
LE, NG	(S = 1 AND O = 0) OR (S = 0 AND O = 1) OR (Z = 1)	Less than or equal; Not greater than (signed)
NE, NZ	Z = 0	Not equal; Not zero (signed or unsigned)
NO	O = 0	No overflow
NS	S = 0	Not sign (not negative)
NP, PO	P = 0	Not parity; Parity odd
O	O = 1	Overflow
P	P = 1	Parity; Parity even
S	S = 1	Sign (negative)

as unsigned integers ($255 > 0$) but is less if they are considered as 8-bit twos complement numbers ($-1 < 0$). Many assembly languages therefore introduce two sets of terms to distinguish the two cases: If we are comparing two numbers as signed integers, we use the terms *less than* and *greater than;* if we are comparing them as unsigned integers, we use the terms *below* and *above.*

A second observation concerns the complexity of comparing signed integers. A signed result is greater than or equal to zero if (1) the sign bit is zero and there is no overflow (S = 0 AND O = 0), or (2) the sign bit is one and there is an overflow. A study of Figure 10.4 should convince you that the conditions tested for the various signed operations are appropriate.

x86 SIMD INSTRUCTIONS In 1996, Intel introduced MMX technology into its Pentium product line. MMX is set of highly optimized instructions for multimedia tasks. There are 57 new instructions that treat data in a SIMD (single-instruction, multiple-data) fashion, which makes it possible to perform the same operation, such as addition or multiplication, on multiple data elements at once. Each instruction typically takes a single clock cycle to execute. For the proper application, these fast parallel operations can yield a speedup of two to eight times over comparable algorithms that do not use the MMX instructions [ATKI96]. With the introduction of 64-bit x86 architecture, Intel has expanded this extension to include double quadword (128 bits) operands and floating-point operations. In this subsection, we describe the MMX features.

The focus of MMX is multimedia programming. Video and audio data are typically composed of large arrays of small data types, such as 8 or 16 bits, whereas conventional instructions are tailored to operate on 32- or 64-bit data. Here are some examples: In graphics and video, a single scene consists of an array of pixels,[2] and there are 8 bits for each pixel or 8 bits for each pixel color component (red, green, blue). Typical audio samples are quantized using 16 bits. For some 3D graphics algorithms, 32 bits are common for basic data types. To provide for parallel operation on these data lengths, three new data types are defined in MMX. Each data type is 64 bits in length and consists of multiple smaller data fields, each of which holds a fixed-point integer. The types are as follows:

- **Packed byte:** Eight bytes packed into one 64-bit quantity.
- **Packed word:** Four 16-bit words packed into 64 bits.
- **Packed doubleword:** Two 32-bit doublewords packed into 64 bits.

Table 12.10 lists the MMX instruction set. Most of the instructions involve parallel operation on bytes, words, or doublewords. For example, the PSLLW instruction performs a left logical shift separately on each of the four words in the packed word operand; the PADDB instruction takes packed byte operands as input and performs parallel additions on each byte position independently to produce a packed byte output.

One unusual feature of the new instruction set is the introduction of **saturation arithmetic** for byte and 16-bit word operands. With ordinary unsigned arithmetic, when an operation overflows (i.e., a carry out of the most significant bit), the extra bit is truncated. This is referred to as wraparound, because the effect of the truncation can be, for example, to produce an addition result that is smaller than the two input operands. Consider the addition of the two words, in hexadecimal, F000h and 3000h. The sum would be expressed as

$$\begin{array}{rcr} \text{F000h} & = & 1111\ 0000\ 0000\ 0000 \\ +\ \text{3000h} & = & \underline{0011\ 0000\ 0000\ 0000} \\ & & 10010\ 0000\ 0000\ 0000 = \text{2000h} \end{array}$$

If the two numbers represented image intensity, then the result of the addition is to make the combination of two dark shades turn out to be lighter. This is typically not what is intended. With saturation arithmetic, if addition results in overflow or subtraction results in underflow, the result is set to the largest or smallest value representable. For the preceding example, with saturation arithmetic, we have

$$\begin{array}{rcr} \text{F000h} & = & 1111\ 0000\ 0000\ 0000 \\ +\ \text{3000h} & = & \underline{0011\ 0000\ 0000\ 0000} \\ & & 10010\ 0000\ 0000\ 0000 \\ & & 1111\ 1111\ 1111\ 1111 = \text{FFFFh} \end{array}$$

[2]A pixel, or picture element, is the smallest element of a digital image that can be assigned a gray level. Equivalently, a pixel is an individual dot in a dot-matrix representation of a picture.

Table 12.10 MMX Instruction Set

Category	Instruction	Description
Arithmetic	PADD [B, W, D]	Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.
	PADDS [B, W]	Add with saturation.
	PADDUS [B, W]	Add unsigned with saturation.
	PSUB [B, W, D]	Subtract with wraparound.
	PSUBS [B, W]	Subtract with saturation.
	PSUBUS [B, W]	Subtract unsigned with saturation.
	PMULHW	Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.
	PMULLW	Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.
	PMADDWD	Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.
Comparison	PCMPEQ [B, W, D]	Parallel compare for equality; result is mask of 1s if true or 0s if false.
	PCMPGT [B, W, D]	Parallel compare for greater than; result is mask of 1s if true or 0s if false.
Conversion	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS [WB, DW]	Pack words into bytes, or doublewords into words, with signed saturation.
	PUNPCKH [BW, WD, DQ]	Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register.
	PUNPCKL [BW, WD, DQ]	Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register.
Logical	PAND	64-bit bitwise logical AND
	PNDN	64-bit bitwise logical AND NOT
	POR	64-bit bitwise logical OR
	PXOR	64-bit bitwise logical XOR
Shift	PSLL [W, D, Q]	Parallel logical left shift of packed words, doublewords, or quadword by amount specified in MMX register or immediate value.
	PSRL [W, D, Q]	Parallel logical right shift of packed words, doublewords, or quadword.
	PSRA [W, D]	Parallel arithmetic right shift of packed words, doublewords, or quadword.
Data transfer	MOV [D, Q]	Move doubleword or quadword to/from MMX register.
Statemgt	EMMS	Empty MMX state (empty FP registers tag bits).

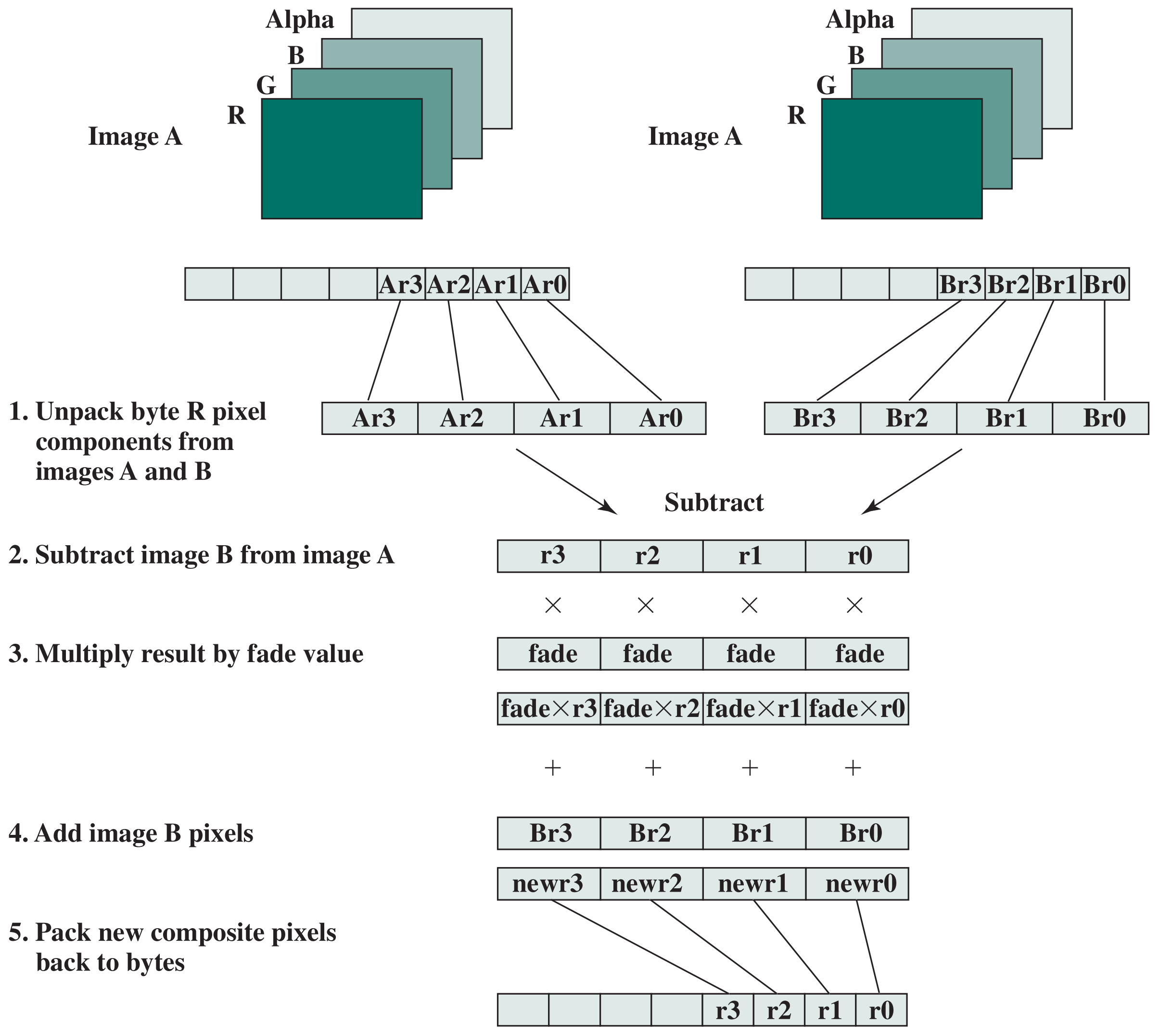
Note: If an instruction supports multiple data types [byte (B), word (W), doubleword (D), quadword (Q)], the data types are indicated in brackets.

To provide a feel for the use of MMX instructions, we look at an example, taken from [PELE97]. A common video application is the fade-out, fade-in effect, in which one scene gradually dissolves into another. Two images are combined with a weighted average:

$$\text{Result\_pixel} = \text{A\_pixel} \times \text{fade} + \text{B\_pixel} \times (1 - \text{fade})$$

This calculation is performed on each pixel position in A and B. If a series of video frames is produced while gradually changing the fade value from 1 to 0 (scaled appropriately for an 8-bit integer), the result is to fade from image A to image B.

Figure 12.11 shows the sequence of steps required for one set of pixels. The 8-bit pixel components are converted to 16-bit elements to accommodate the MMX 16-bit multiply capability. If these images use 640 × 480 resolution, and the dissolve technique uses all 255 possible values of the fade value, then the total number of



MMX code sequence performing this operation:

```
pxor        mm7, mm7      ;zero out mm7
movq        mm3, fad_val  ;load fade value replicated 4 times
movd        mm0, imageA   ;load 4 red pixel components from image A
movd        mm1, imageB   ;load 4 red pixel components from image B
punpckblw   mm0, mm7      ;unpack 4 pixels to 16 bits
punpckblw   mm1, mm7      ;unpack 4 pixels to 16 bits
psubw       mm0, mm1      ;subtract image B from image A
pmulhw      mm0, mm3      ;multiply the subtract result by fade values
padddw      mm0, mm1      ;add result to image B
packuswb    mm0, mm7      ;pack 16-bit results back to bytes
```

Figure 12.11 Image Compositing on Color Plane Representation

instructions executed using MMX is 535 million. The same calculation, performed without the MMX instructions, requires 1.4 billion instruction executions [INTE98].

ARM Operation Types

The ARM architecture provides a large collection of operation types. The following are the principal categories:

- **Load and store instructions:** In the ARM architecture, only load and store instructions access memory locations; arithmetic and logical instructions are performed only on registers and immediate values encoded in the instruction. This limitation is characteristic of RISC design and it is explored further in Chapter 15. The ARM architecture supports two broad types of instruction that load or store the value of a single register, or a pair of registers, from or to memory: (1) load or store a 32-bit word or an 8-bit unsigned byte, and (2) load or store a 16-bit unsigned halfword, and load and sign extend a 16-bit halfword or an 8-bit byte.
- **Branch instructions:** ARM supports a branch instruction that allows a conditional branch forwards or backwards up to 32 MB. A subroutine call can be performed by a variant of the standard branch instruction. As well as allowing a branch forward or backward up to 32 MB, the Branch with Link (BL) instruction preserves the address of the instruction after the branch (the return address) in the LR (R14). Branches are determined by a 4-bit condition field in the instruction.
- **Data-processing instructions:** This category includes logical instructions (AND, OR, XOR), add and subtract instructions, and test and compare instructions.
- **Multiply instructions:** The integer multiply instructions operate on word or halfword operands and can produce normal or long results. For example, there is a multiply instruction that takes two 32-bit operands and produces a 64-bit result.
- **Parallel addition and subtraction instructions:** In addition to the normal data processing and multiply instructions, there are a set of parallel addition and subtraction instructions, in which portions of two operands are operated on in parallel. For example, ADD16 adds the top halfwords of two registers to form the top halfword of the result and adds the bottom halfwords of the same two registers to form the bottom halfword of the result. These instructions are useful in image processing applications, similar to the x86 MMX instructions.
- **Extend instructions:** There are several instructions for unpacking data by sign or zero extending bytes to halfwords or words, and halfwords to words.
- **Status register access instructions:** ARM provides the ability to read and also to write portions of the status register.

CONDITION CODES The ARM architecture defines four condition flags that are stored in the program status register: N, Z, C, and V (Negative, Zero, Carry and oVerflow), with meanings essentially the same as the S, Z, C, and V flags in the

Table 12.11 ARM Conditions for Conditional Instruction Execution

Code	Symbol	Condition Tested	Comment
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS/HS	C = 1	Carry set/unsigned higher or same
0011	CC/LO	C = 0	Carry clear/unsigned lower
0100	MI	N = 1	Minus/negative
0101	PL	N = 0	Plus/positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 AND Z = 0	Unsigned higher
1001	LS	C = 0 OR Z = 1	Unsigned lower or same
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)]	Signed greater than or equal
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	Signed less than
1100	GT	(Z = 0) AND (N = V)	Signed greater than
1101	LE	(Z = 1) OR (N ≠ V)	Signed less than or equal
1110	AL	—	Always (unconditional)
1111	—	—	This instruction can only be executed unconditionally

x86 architecture. These four flags constitute a condition code in ARM. Table 12.11 shows the combination of conditions for which conditional execution is defined.

There are two unusual aspects to the use of condition codes in ARM:

1. All instructions, not just branch instructions, include a condition code field, which means that virtually all instructions may be conditionally executed. Any combination of flag settings except 1110 or 1111 in an instruction's condition code field signifies that the instruction will be executed only if the condition is met.

2. All data processing instructions (arithmetic, logical) include an S bit that signifies whether the instruction updates the condition flags.

The use of conditional execution and conditional setting of the condition flags helps in the design of shorter programs that use less memory. On the other hand, all instructions include 4 bits for the condition code, so there is a trade-off in that fewer bits in the 32-bit instruction are available for opcode and operands. Because the ARM is a RISC design that relies heavily on register addressing, this seems to be a reasonable trade-off.

12.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

accumulator	jump	procedure call
address	little endian	procedure return
arithmetic shift	logical shift	push
bi-endian	machine instruction	reentrant procedure
big endian	operand	rotate
branch	operation	skip
conditional branch	packed decimal	stack
instruction set	pop	

Review Questions

12.1 What are the typical elements of a machine instruction?
12.2 What types of locations can hold source and destination operands?
12.3 If an instruction contains four addresses, what might be the purpose of each address?
12.4 List and briefly explain five important instruction set design issues.
12.5 What types of operands are typical in machine instruction sets?
12.6 What is the relationship between the IRA character code and the packed decimal representation?
12.7 What is the difference between an arithmetic shift and a logical shift?
12.8 Why are transfer of control instructions needed?
12.9 List and briefly explain two common ways of generating the condition to be tested in a conditional branch instruction.
12.10 What is meant by the term *nesting of procedures*?
12.11 List three possible places for storing the return address for a **procedure return**.
12.12 What is a reentrant procedure?
12.13 What is **reverse Polish notation**?
12.14 What is the difference between big endian and little endian?

Problems

12.1 Show in hex notation:
a. The packed decimal format for 23.
b. The ASCII characters 23.

12.2 For each of the following packed decimal numbers, show the decimal value:
a. 0111 0011 0000 1001
b. 0101 1000 0010
c. 0100 1010 0110

12.3 A given microprocessor has words of 1 byte. What is the smallest and largest integer that can be represented in the following representations:
a. Unsigned.
b. Sign-magnitude.
c. Ones complement.
d. Twos complement.

e. Unsigned packed decimal.
f. Signed packed decimal.

12.4 Many processors provide logic for performing arithmetic on packed decimal numbers. Although the rules for decimal arithmetic are similar to those for binary operations, the decimal results may require some corrections to the individual digits if binary logic is used.

Consider the decimal addition of two unsigned numbers. If each number consists of N digits, then there are $4N$ bits in each number. The two numbers are to be added using a binary adder. Suggest a simple rule for correcting the result. Perform addition in this fashion on the numbers 1698 and 1786.

12.5 The tens complement of the decimal number X is defined to be $10^N - X$, where N is the number of decimal digits in the number. Describe the use of ten's complement representation to perform decimal subtraction. Illustrate the procedure by subtracting $(0326)_{10}$ from $(0736)_{10}$.

12.6 Compare zero-, one-, two-, and three-address machines by writing programs to compute

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

for each of the four machines. The instructions available for use are as follows:

0 Address	**1 Address**	**2 Address**	**3 Address**
PUSH M	LOAD M	MOVE $(X \leftarrow Y)$	MOVE $(X \leftarrow Y)$
POP M	STORE M	ADD $(X \leftarrow X + Y)$	ADD $(X \leftarrow Y + Z)$
ADD	ADD M	SUB $(X \leftarrow X - Y)$	SUB $(X \leftarrow Y - Z)$
SUB	SUB M	MUL $(X \leftarrow X \times Y)$	MUL $(X \leftarrow Y \times Z)$
MUL	MUL M	DIV $(X \leftarrow X/Y)$	DIV $(X \leftarrow Y/Z)$
DIV	DIV M		

12.7 Consider a hypothetical computer with an instruction set of only two n-bit instructions. The first bit specifies the opcode, and the remaining bits specify one of the 2^{n-1} n-bit words of main memory. The two instructions are as follows:

SUBS X	Subtract the contents of location X from the accumulator, and store the result in location X and the accumulator.
JUMP X	Place address X in the program counter.

A word in main memory may contain either an instruction or a binary number in twos complement notation. Demonstrate that this instruction repertoire is reasonably complete by specifying how the following operations can be programmed:
a. Data transfer: Location X to accumulator, accumulator to location X.
b. Addition: Add contents of location X to accumulator.
c. Conditional branch.
d. Logical OR.
e. I/O Operations.

12.8 Many instruction sets contain the instruction NOOP, meaning no operation, which has no effect on the processor state other than incrementing the program counter. Suggest some uses of this instruction.

12.9 In Section 12.4, it was stated that both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow, and if overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Demonstrate that these statements are true for 5-bit twos complement integers.

12.10 In what way are numbers rounded using arithmetic right shift (e.g., round toward $+\infty$, round toward $-\infty$, toward zero, away from 0)?

12.11 Suppose a stack is to be used by the processor to manage procedure calls and returns. Can the program counter be eliminated by using the top of the stack as a program counter?

12.12 The x86 architecture includes an instruction called Decimal Adjust after Addition (DAA). DAA performs the following sequence of instructions:

```
if ((AL AND 0FH) >9) OR (AF = 1)  then
      AL ← AL + 6;
      AF ← 1;
else
      AF ← 0;
endif;
if (AL > 9FH) OR (CF = 1)  then
      AL ← AL + 60H;
      CF ← 1;
else
      CF ← 0;
endif.
```

"H" indicates hexadecimal. AL is an 8-bit register that holds the result of addition of two unsigned 8-bit integers. AF is a flag set if there is a carry from bit 3 to bit 4 in the result of an addition. CF is a flag set if there is a carry from bit 7 to bit 8. Explain the function performed by the DAA instruction.

12.13 The x86 Compare instruction (CMP) subtracts the source operand from the destination operand; it updates the status flags (C, P, A, Z, S, O) but does not alter either of the operands. The CMP instruction can be used to determine if the destination operand is greater than, equal to, or less than the source operand.

a. Suppose the two operands are treated as unsigned integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.

b. Suppose the two operands are treated as twos complement signed integers. Show which status flags are relevant to determine the relative size of the two integer and what values of the flags correspond to greater than, equal to, or less than.

c. The CMP instruction may be followed by a conditional Jump (Jcc) or Set Condition (SETcc) instruction, where cc refers to one of the 16 conditions listed in Table 12.11. Demonstrate that the conditions tested for a signed number comparison are correct.

12.14 Suppose we wished to apply the x86 CMP instruction to 32-bit operands that contained numbers in a floating-point format. For correct results, what requirements have to be met in the following areas?

a. The relative position of the significand, sign, and exponent fields.

b. The representation of the value zero.

c. The representation of the exponent.

d. Does the IEEE format meet these requirements? Explain.

12.15 Many microprocessor instruction sets include an instruction that tests a condition and sets a destination operand if the condition is true. Examples include the SETcc on the x86, the Scc on the Motorola MC68000, and the Scond on the National NS32000.

a. There are a few differences among these instructions:

- SETcc and Scc operate only on a byte, whereas Scond operates on byte, word, and doubleword operands.
- SETcc and Scond set the operand to integer one if true and to zero if false. Scc sets the byte to all binary ones if true and all zeros if false. What are the relative advantages and disadvantages of these differences?

b. None of these instructions set any of the condition code flags, and thus an explicit test of the result of the instruction is required to determine its value. Discuss whether condition codes should be set as a result of this instruction.

c. A simple IF statement such as IF a > b THEN can be implemented using a numerical representation method, that is, making the Boolean value manifest, as opposed to a *flow of control* method, which represents the value of a Boolean expression by a point reached in the program. A compiler might implement IF a > ssb THEN with the following x86 code:

```
        SUB     CX, CX    ;set register CX to 0
        MOV     AX, B     ;move contents of location B to register AX
        CMP     AX, A     ;compare contents of register AX and location A
        JLE     TEST      ;jump if A ≤ B
        INC     CX        ;add 1 to contents of register CX
TEST    JCXZ    OUT       ;jump if contents of CX equal 0
THEN            OUT
```

The result of (A > B) is a Boolean value held in a register and available later on, outside the context of the flow of code just shown. It is convenient to use register CX for this, because many of the branch and loop opcodes have a built-in test for CX.

Show an alternative implementation using the SETcc instruction that saves memory and execution time. (*Hint:* No additional new x86 instructions are needed, other than the SETcc.)

d. Now consider the high-level language statement:

$$A: = (B > C) \text{ OR } (D = F)$$

A compiler might generate the following code:

```
      MOV       EAX, B    ;move contents of location B to register EAX
      CMP       EAX, C    ;compare contents of register EAX and location C
      MOV       BL, 0     ;0 represents false
      JLE       N1        ;jump if (B ≤ C)
      MOV       BL, 1     ;1 represents false
N1    MOV       EAX, D
      CMP       EAX, F
      MOV       BH, 0
      JNE       N2
      MOV       BH, 1
N2    OR        BL, BH
```

Show an alternative implementation using the SETcc instruction that saves memory and execution time.

12.16 Suppose that two registers contain the following hexadecimal values: AB0890C2, 4598EE50. What is the result of adding them using MMX instructions:

a. packed byte.

b. packed word.

Assume saturation arithmetic is not used.

12.17 Appendix I points out that there are no stack-oriented instructions in an instruction set if the stack is to be used only by the processor for such purposes as procedure handling. How can the processor use a stack for any purpose without stack-oriented instructions?

12.18 Mathematical formulas are usually expressed in what is known as infix notation, in which a binary operator appears between the operands. An alternative technique is known as **reverse Polish**, or **postfix**, notation, in which the operator follows its two operands. See Appendix I for more details. Convert the following formulas from reverse Polish to infix:

a. $AB + C + D \times$
b. $AB/CD/ +$
c. $ABCDE + \times \times /$
d. $ABCDE + F/ + G - H/ \times +$

12.19 Convert the following formulas from infix to reverse Polish:

a. $A + B + C + D + E$
b. $(A + B) \times (C + D) + E$
c. $(A \times B) + (C \times D) + E$
d. $(A - B) \times (((C - D \times E)/F)/G) \times H$

12.20 Convert the expression $A + B - C$ to postfix notation using Dijkstra's algorithm. Show the steps involved. Is the result equivalent to $(A + B) - C$ or $A + (B - C)$? Does it matter?

12.21 Using the algorithm for converting infix to postfix defined in Appendix I, show the steps involved in converting the expression of Figure I.3 into postfix. Use a presentation similar to Figure I.5.

12.22 Show the calculation of the expression in Figure I.5, using a presentation similar to Figure I.4.

12.23 Redraw the little-endian layout in Figure 12.13 so that the bytes appear as numbered in the big-endian layout. That is, show memory in 64-bit rows, with the bytes listed left to right, top to bottom.

12.24 For the following data structures, draw the big-endian and little-endian layouts, using the format of Figure 12.13, and comment on the results.

a.
```
struct {
   double i;     //0x1112131415161718
} s1;
```
b.
```
struct {
   int i;        //0x11121314
   int j;        //0x15161718
} s2;
```
c.
```
struct {
   short i;      //0x1112
   short j;      //0x1314
   short k;      //0x1516
   short l;      //0x1718
} s3;
```

12.25 The IBM Power architecture specification does not dictate how a processor should implement little-endian mode. It specifies only the view of memory a processor must have when operating in little-endian mode. When converting a data structure from big endian to little endian, processors are free to implement a true byte-swapping mechanism or to use some sort of an address modification mechanism. Current Power processors are all default big-endian machines and use address modification to treat data as little-endian.

Consider the structure s defined in Figure 12.13. The layout in the lower-right portion of the figure shows the structure s as seen by the processor. In fact, if structures is compiled in little-endian mode, its layout in memory is shown in Figure 12.12.

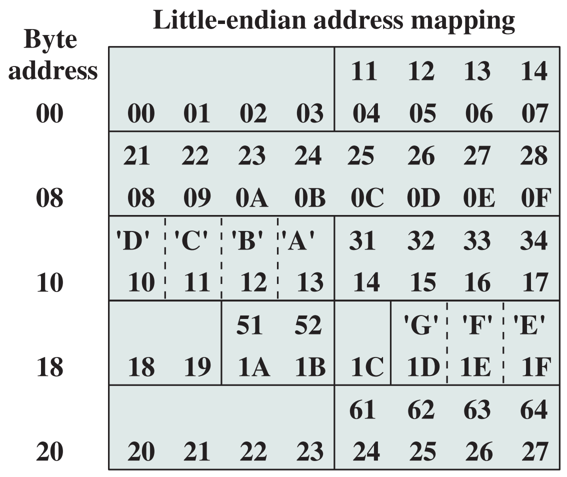


Figure 12.12 Power Architecture Little-Endian Structures in Memory

Explain the mapping that is involved, describe an easy way to implement the mapping, and discuss the effectiveness of this approach.

12.26 Write a small program to determine the endianness of machine and report the results. Run the program on a computer available to you and turn in the output.

12.27 The MIPS processor can be set to operate in either big-endian or little-endian mode. Consider the Load Byte Unsigned (LBU) instruction, which loads a byte from memory into the low-order 8 bits of a register and fills the high-order 24 bits of the register with zeros. The description of LBU is given in the MIPS reference manual using a register-transfer language as

```
mem ← LoadMemory(...)
byte ← $\text{VirtualAddress}_{1..0}$
if CONDITION then
        GPR[rt] ← $0^{24} \| \text{mem}_{31 - 8 \times \text{byte} .. 24 - 8 \times \text{byte}}$
else
        GPR[rt] ← $0^{24} \| \text{mem}_{7 + 8 \times \text{byte} .. 8 \times \text{byte}}$
endif
```

where *byte* refers to the two low-order bits of the effective address and *mem* refers to the value loaded from memory. In the manual, instead of the word CONDITION, one of the following two words is used: BigEndian, LittleEndian. Which word is used?

12.28 Most, but not all, processors use big- or little-endian bit ordering within a byte that is consistent with big- or little-endian ordering of bytes within a multibyte scalar. Let us consider the Motorola 68030, which uses big-endian byte ordering. The documentation of the 68030 concerning formats is confusing. The user's manual explains that the bit ordering of bit fields is the opposite of bit ordering of integers. Most bit field operations operate with one endian ordering, but a few bit field operations require the opposite ordering. The following description from the user's manual describes most of the bit field operations:

> A bit operand is specified by a base address that selects one byte in memory (the base byte), and a bit number that selects the one bit in this byte. The most significant bit is bit seven. A bit field operand is specified by: **(1)** a base address that selects one byte in memory; **(2)** a bit field offset that indicates the leftmost (base) bit of the bit field in relation to the most significant bit of the base byte; and **(3)** a bit field width that determines how many bits to the right of the base byte are in the bit field. The most significant bit of the base byte is bit field offset 0, the least significant bit of the base byte is bit field offset 7.

Do these instructions use big-endian or little-endian bit ordering?

APENDIX 12A LITTLE-, BIG-, AND BI-ENDIAN

An annoying and curious phenomenon relates to how the bytes within a word and the bits within a byte are both referenced and represented. We look first at the problem of byte ordering and then consider that of bits.

Byte Ordering

The concept of endianness was first discussed in the literature by Cohen [COHE81]. With respect to bytes, endianness has to do with the byte ordering of multibyte scalar values. The issue is best introduced with an example. Suppose we have the 32-bit hexadecimal value 12345678 and that it is stored in a 32-bit word in byte-addressable memory at byte location 184. The value consists of 4 bytes, with the least significant byte containing the value 78 and the most significant byte containing the value 12. There are two obvious ways to store this value:

Address	Value
184	12
185	34
186	56
187	78

Address	Value
184	78
185	56
186	34
187	12

The mapping on the left stores the most significant byte in the lowest numerical byte address; this is known as **big endian** and is equivalent to the left-to-right order of writing in Western culture languages. The mapping on the right stores the least significant byte in the lowest numerical byte address; this is known as **little endian** and is reminiscent of the right-to-left order of arithmetic operations in arithmetic units.[3] For a given multibyte scalar value, big endian and little endian are byte-reversed mappings of each other.

The concept of endianness arises when it is necessary to treat a multiple-byte entity as a single data item with a single address, even though it is composed of smaller addressable units. Some machines, such as the Intel 80x86, x86, VAX, and Alpha, are little-endian machines, whereas others, such as the IBM System 370/390, the Motorola 680x0, Sun SPARC, and most RISC machines, are big endian. This presents problems when data are transferred from a machine of one endian type to the other and when a programmer attempts to manipulate individual bytes or bits within a multibyte scalar.

The property of endianness does not extend beyond an individual data unit. In any machine, aggregates such as files, data structures, and arrays are composed of multiple data units, each with endianness. Thus, conversion of a block of memory from one style of endianness to the other requires knowledge of the data structure.

Figure 12.13 illustrates how endianness determines addressing and byte order. The C structure at the top contains a number of data types. The memory layout in the

[3]The terms *big endian* and *little endian* come from Part I, Chapter 4 of Jonathan Swift's *Gulliver's Travels*. They refer to a religious war between two groups, one that breaks eggs at the big end and the other that breaks eggs at the little end.

```
struct{
  int     a;     //0x1112_1314                    word
  int     pad;   //
  double  b;     //0x2122_2324_2526_2728          doubleword
  char*   c;     //0x3132_3334                    word
  char    d[7];  //'A','B','C','D','E','F','G'    byte array
  short   e;     //0x5152                         halfword
  int     f;     //0x6162_6364                    word
} s;
```

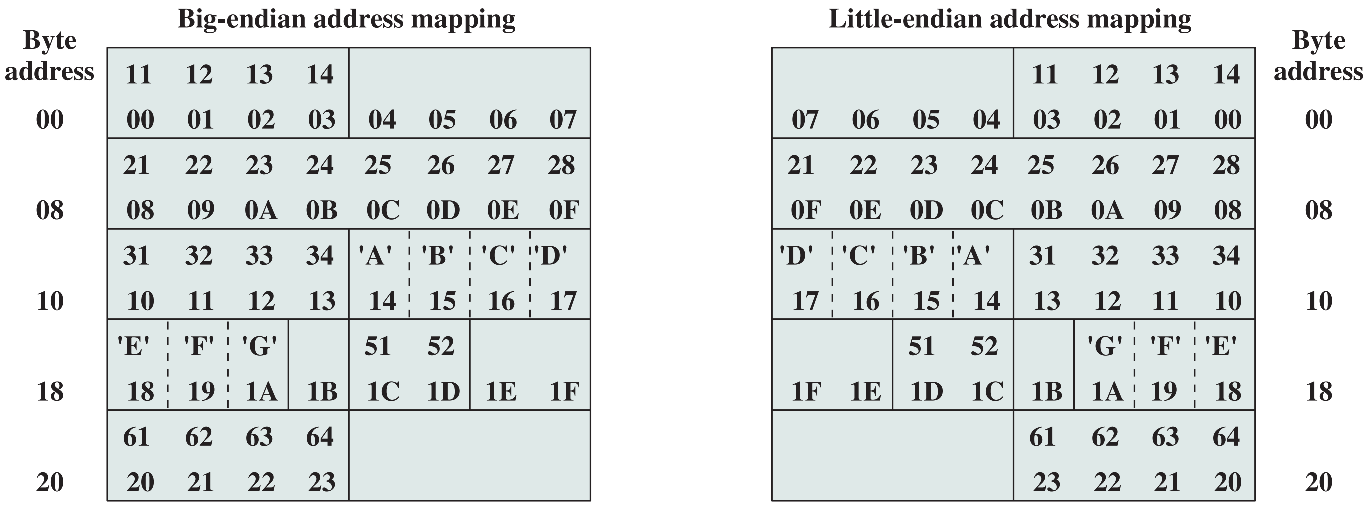


Figure 12.13 Example C Data Structure and Its Endian Maps

lower left results from compilation of that structure for a big-endian machine, and that in the lower right for a little-endian machine. In each case, memory is depicted as a series of 64-bit rows. For the big-endian case, memory typically is viewed left to right, top to bottom, whereas for the little-endian case, memory typically is viewed as right to left, top to bottom. Note that these layouts are arbitrary. Either scheme could use either left to right or right to left within a row; this is a matter of depiction, not memory assignment. In fact, in looking at programmer manuals for a variety of machines, a bewildering collection of depictions is to be found, even within the same manual.

```
struct{
   int a; //0x1112_1314                            word
   int pad; //
   double b; //0x2122_2324_2526_2728               doubleword
   char* c; //0x3132_3334                          word
   char d[7]; //'A','B','C','D','E','F','G'        byte array
   short e; //0x5152                               halfword
   int f; //0x6162_6364                            word
} s;
```

We can make several observations about this data structure:

- Each data item has the same address in both schemes. For example, the address of the doubleword with hexadecimal value 2122232425262728 is 08.
- Within any given multibyte scalar value, the ordering of bytes in the little-endian structure is the reverse of that for the big-endian structure.

- Endianness does not affect the ordering of data items within a structure. Thus, the four-character word c exhibits byte reversal, but the seven-character byte array d does not. Hence, the address of each individual element of d is the same in both structures.

The effect of endianness is perhaps more clearly demonstrated when we view memory as a vertical array of bytes, as shown in Figure 12.14.

There is no general consensus as to which is the superior style of endianness.[4] The following points favor the big-endian style:

- **Character-string sorting:** A big-endian processor is faster in comparing integer-aligned character strings; the integer ALU can compare multiple bytes in parallel.

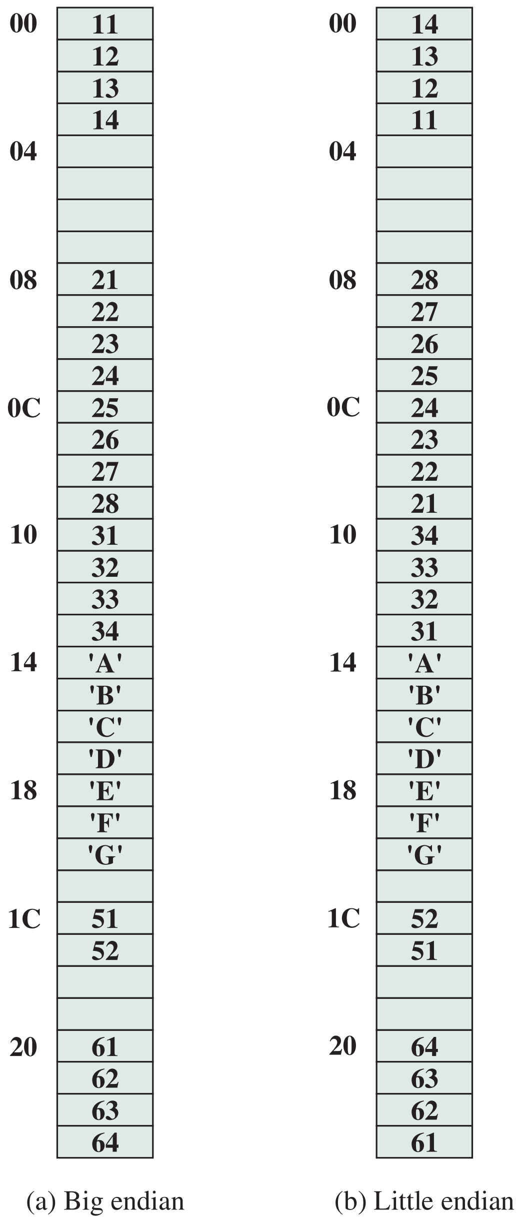


Figure 12.14 Another View of Figure 12.13

[4]The prophet revered by both groups in the Endian Wars of *Gulliver's Travels* had this to say. "All true Believers shall break their Eggs at the convenient End." Not much help!

- **Decimal/IRA dumps:** All values can be printed left to right without causing confusion.
- **Consistent order:** Big-endian processors store their integers and character strings in the same order (most significant byte comes first).

The following points favor the little-endian style:

- A big-endian processor has to perform addition when it converts a 32-bit integer address to a 16-bit integer address, to use the least significant bytes.
- It is easier to perform higher-precision arithmetic with the little-endian style; you don't have to find the least-significant byte and move backward.

The differences are minor and the choice of endian style is often more a matter of accommodating previous machines than anything else.

The PowerPC is a bi-endian processor that supports both big-endian and little-endian modes. The bi-endian architecture enables software developers to choose either mode when migrating operating systems and applications from other machines. The operating system establishes the endian mode in which processes execute. Once a mode is selected, all subsequent memory loads and stores are determined by the memory-addressing model of that mode. To support this hardware feature, 2 bits are maintained in the machine state register (MSR) maintained by the operating system as part of the process state. One bit specifies the endian mode in which the kernel runs; the other specifies the processor's current operating mode. Thus, mode can be changed on a per-process basis.

Bit Ordering

In ordering the bits within a byte, we are immediately faced with two questions:

1. Do you count the first bit as bit zero or as bit one?
2. Do you assign the lowest bit number to the byte's least significant bit (little endian) or to the bytes most significant bit (big endian)?

These questions are not answered in the same way on all machines. Indeed, on some machines, the answers are different in different circumstances. Furthermore, the choice of big- or little-endian bit ordering within a byte is not always consistent with big- or little-endian ordering of bytes within a multibyte scalar. The programmer needs to be concerned with these issues when manipulating individual bits.

Another area of concern is when data are transmitted over a bit-serial line. When an individual byte is transmitted, does the system transmit the most significant bit first or the least significant bit first? The designer must make certain that incoming bits are handled properly. For a discussion of this issue, see [JAME90].