6

Conditional Processing

This chapter introduces a major item to your assembly language toolchest, giving your programs the ability to make decisions. Nearly every program needs this capability. First, we start by introducing you to the Boolean operations that are the core of all decision statements because they affect the CPU status flags. Then we show how to use conditional jump and loop instructions that interpret CPU status flags. Next, we show how to use the tools from this chapter to implement one of the most fundamental structures in theoretical computer science: the Finite-State Machine. We finish the chapter by demonstrating MASM's built-in logic structures for 32-bit programming.

6.1 Conditional Branching

A programming language that permits decision making lets you alter the flow of control, using a technique known as *conditional branching*. Every IF statement, switch statement, or conditional loop found in high-level languages has built-in branching logic. Assembly language, as primitive as it is, provides all the tools you need for decision-making logic. In this chapter, we will see how the translation works, from high-level conditional statements to low-level implementation code.

Programs that deal with hardware devices must be able to manipulate individual bits in numbers. Individual bits must be tested, cleared, and set. Data encryption and compression also rely on bit manipulation. We will show how to perform these operations in assembly language.

This chapter should answer some basic questions:

- How can I use the boolean operations introduced in Chapter 1 (AND, OR, NOT)?
- How do I write an IF statement in assembly language?
- How are nested-IF statements translated by compilers into machine language?
- How can I set and clear individual bits in a binary number?
- How can I perform simple binary data encryption?
- How are signed numbers differentiated from unsigned numbers in boolean expressions?

This chapter follows a *bottom-up* approach, starting with the binary foundations behind programming logic. Next, you will see how the CPU compares instruction operands, using the CMP instruction and the processor status flags. Finally, we put it all together and show how to use assembly language to implement logic structures characteristic of high-level languages.

6.2 Boolean and Comparison Instructions

In Chapter 1, we introduced the four basic operations of boolean algebra: AND, OR, XOR, and NOT. These operations can be carried out at the binary bit level, using assembly language instructions. These operations are also important at the boolean expression level, in IF statements, for example. First, we will look at the bitwise instructions. The techniques used here could be used to manipulate control bits for hardware devices, implement communication protocols, or encrypt data, just to name a few applications. The Intel instruction set contains the AND, OR, XOR, and NOT instructions, which directly implement boolean operations on binary bits, shown in Table 6-1. In addition, the TEST instruction is a nondestructive AND operation.

Table 6-1 Selected Boolean Instructions.

Operation	Description
AND	Boolean AND operation between a source operand and a destination operand.
OR	Boolean OR operation between a source operand and a destination operand.
XOR	Boolean exclusive-OR operation between a source operand and a destination operand.
NOT	Boolean NOT operation on a destination operand.
TEST	Implied boolean AND operation between a source and destination operand, setting the CPU flags appropriately.

6.2.1 The CPU Status Flags

Boolean instructions affect the Zero, Carry, Sign, Overflow, and Parity flags. Here's a quick review of their meanings:

- The Zero flag is set when the result of an operation equals zero.
- The Carry flag is set when an operation generates a carry out of the highest bit of the destination operand.
- The Sign flag is a copy of the high bit of the destination operand, indicating that it is negative if *set* and positive if *clear*. (Zero is assumed to be positive.)
- The Overflow flag is set when an instruction generates a result that is outside the signed range of the destination operand.
- The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.

6.2.2 AND Instruction

The AND instruction performs a boolean (bitwise) AND operation between each pair of matching bits in two operands and places the result in the destination operand:

```
AND  destination,source
```

The following operand combinations are permitted, although immediate opperands can be no larger than 32 bits:

```
AND reg,reg
AND reg,mem
AND reg,imm
AND mem,reg
AND mem,imm
```

The operands can be 8, 16, 32, or 64 bits, and they must be the same size. For each matching bit in the two operands, the following rule applies: If both bits equal 1, the result bit is 1; otherwise, it is 0. The following truth table from Chapter 1 labels the input bits x and y. The third column shows the value of the expression $\mathbf{x} \wedge \mathbf{y}$:

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

The AND instruction lets you clear 1 or more bits in an operand without affecting other bits. The technique is called bit *masking*, much as you might use masking tape when painting a house to cover areas (such as windows) that should not be painted. Suppose, for example, that a control byte is about to be copied from the AL register to a hardware device. Further, we will assume that the device resets itself when bits 0 and 3 are cleared in the control byte. Assuming that we want to reset the device without modifying any other bits in AL, we can write the following:

```
and AL,11110110b    ; clear bits 0 and 3, leave others unchanged
```

For example, suppose AL is initially set to 10101110 binary. After ANDing it with 11110110, AL equals 10100110:

```
mov al,10101110b
and al,11110110b              ; result in AL = 10100110
```

Flags The AND instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand. For example, suppose the following instruction results in a value of Zero in the EAX register. In that case, the Zero flag will be set:

```
and eax,1Fh
```

Converting Characters to Upper case

The AND instruction provides an easy way to translate a letter from lowercase to uppercase. If we compare the ASCII codes of capital **A** and lowercase **a**, it becomes clear that only bit 5 is different:

```
0 1 1 0 0 0 0 1 = 61h ('a')
0 1 0 0 0 0 0 1 = 41h ('A')
```

The rest of the alphabetic characters have the same relationship. If we AND any character with 11011111 binary, all bits are unchanged except for bit 5, which is cleared. In the following example, all characters in an array are converted to uppercase:

```
.data
array BYTE 50 DUP(?)
.code
    mov   ecx,LENGTHOF array
    mov   esi,OFFSET array
L1: and   BYTE PTR [esi],11011111b    ; clear bit 5
    inc   esi
    loop  L1
```

6.2.3 OR Instruction

The OR instruction performs a boolean OR operation between each pair of matching bits in two operands and places the result in the destination operand:

```
OR  destination,source
```

The OR instruction uses the same operand combinations as the AND instruction:

```
OR reg,reg
OR reg,mem
OR reg,imm
OR mem,reg
OR mem,imm
```

The operands can be 8, 16, 32, or 64 bits, and they must be the same size. For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1. The following truth table (from Chapter 1) describes the boolean expression **x** ∨ **y**:

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

The OR instruction is particularly useful when you need to set 1 or more bits in an operand without affecting any other bits. Suppose, for example, that your computer is attached to a servo motor, which is activated by setting bit 2 in its control byte. Assuming that the AL register contains a control byte in which each bit contains some important information, the following code only sets the bit in position 2:

```
or AL,00000100b                ; set bit 2, leave others unchanged
```

For example, if AL is initially equal to 11100011 binary and then we OR it with 00000100, the result equals 11100111:

```
mov al,11100011b
or  al,00000100b               ; result in AL = 11100111
```

Flags The OR instruction always clears the Carry and Overflow flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand. For example, you can OR a number with itself (or zero) to obtain certain information about its value:

```
or  al,al
```

The values of the Zero and Sign flags indicate the following about the contents of AL:

Zero Flag	Sign Flag	Value in AL Is . . .
Clear	Clear	Greater than zero
Set	Clear	Equal to zero
Clear	Set	Less than zero

6.2.4 Bit-Mapped Sets

Some applications manipulate sets of items selected from a limited-sized universal set. Examples might be employees within a company, or environmental readings from a weather monitoring station. In such cases, binary bits can indicate set membership. Rather than holding pointers or references to objects in a container such as a Java HashSet, an application can use a *bit vector* (or bit map) to map the bits in a binary number to an array of objects.

For example, the following binary number uses bit positions numbered from 0 on the right to 31 on the left to indicate that array elements 0, 1, 2, and 31 are members of the set named **SetX**:

```
SetX = 10000000 00000000 00000000 00000111
```

(The bytes have been separated to improve readability.) We can easily check for set membership by ANDing a particular member's bit position with a 1:

```
mov  eax,SetX
and  eax,10000b                ; is element[4] a member of SetX?
```

If the AND instruction in this example clears the Zero flag, we know that element [4] is a member of SetX.

Set Complement

The complement of a set can be generated using the NOT instruction, which reverses all bits. Therefore, the complement of the SetX that we introduced is generated in EAX using the following instructions:

```
mov  eax,SetX
not  eax                       ; complement of SetX
```

Set Intersection

The AND instruction produces a bit vector that represents the intersection of two sets. The following code generates and stores the intersection of SetX and SetY in EAX:

```
mov  eax,SetX
and  eax,SetY
```

This is how the intersection of SetX and SetY is produced:

```
          10000000000000000000000000000111    (SetX)
AND       10000010101000000000011101100011    (SetY)
--------------------------------------------------------
          10000000000000000000000000000011    (intersection)
```

It is hard to imagine any faster way to generate a set intersection. A larger domain would require more bits than could be held in a single register, making it necessary to use a loop to AND all of the bits together.

Set Union

The OR instruction produces a bit map that represents the union of two sets. The following code generates the union of SetX and SetY in EAX:

```
mov  eax,SetX
or   eax,SetY
```

This is how the union of SetX and SetY is generated by the OR instruction:

```
          10000000000000000000000000000111     (SetX)
OR        10000010101000000000011101100011     (SetY)
-------------------------------------------------------
          10000010101000000000011101100111     (union)
```

6.2.5 XOR Instruction

The XOR instruction performs a boolean exclusive-OR operation between each pair of matching bits in two operands and stores the result in the destination operand:

```
XOR  destination,source
```

The XOR instruction uses the same operand combinations and sizes as the AND and OR instructions. For each matching bit in the two operands, the following applies: If both bits are the same (both 0 or both 1), the result is 0; otherwise, the result is 1. The following truth table describes the boolean expression **x** ⊕ **y**:

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

A bit exclusive-ORed with 0 retains its value, and a bit exclusive-ORed with 1 is toggled (complemented). XOR reverses itself when applied twice to the same operand. The following truth table shows that when bit *x* is exclusive-ORed with bit *y* twice, it reverts to its original value:

x	y	x ⊕ y	(x ⊕ y) ⊕ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

As you will find out in Section 6.3.4, this "reversible" property of XOR makes it an ideal tool for a simple form of symmetric encryption.

Flags The XOR instruction always clears the Overflow and Carry flags. XOR modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand.

Checking the Parity Flag Parity checking is a function performed on a binary number that counts the number of 1 bits contained in the number; if the resulting count is even, we say that the data has even parity; if the count is odd, the data has odd parity. In x86 processors, the Parity

flag is set when the lowest byte of the destination operand of a bitwise or arithmetic operation has even parity. Conversely, when the operand has odd parity, the flag is cleared. An effective way to check the parity of a number without changing its value is to exclusive-OR the number with zero:

```
mov  al,10110101b              ; 5 bits = odd parity
xor  al,0                      ; Parity flag clear (odd)
mov  al,11001100b              ; 4 bits = even parity
xor  al,0                      ; Parity flag set (even)
```

Visual Studio uses PE = 1 to indicate even parity, and PE = 0 to indicate odd parity.

16-Bit Parity You can check the parity of a 16-bit integer by performing an exclusive-OR between the upper and lower bytes:

```
mov  ax,64C1h                  ; 0110 0100 1100 0001
xor  ah,al                     ; Parity flag set (even)
```

Imagine the set bits (bits equal to 1) in each register as being members of an 8-bit set. The XOR instruction zeroes all bits belonging to the intersection of the sets. XOR also forms the union between the remaining bits. The parity of this union will be the same as the parity of the entire 16-bit integer.

What about 32-bit values? If we number the bytes from B_0 through B_3, we can calculate the parity as B_0 XOR B_1 XOR B_2 XOR B_3.

6.2.6 NOT Instruction

The NOT instruction toggles (inverts) all bits in an operand. The result is called the *one's complement*. The following operand types are permitted:

```
NOT reg
NOT mem
```

For example, the one's complement of F0h is 0Fh:

```
mov  al,11110000b
not  al                        ; AL = 00001111b
```

Flags No flags are affected by the NOT instruction.

6.2.7 TEST Instruction

The TEST instruction performs an implied AND operation between each pair of matching bits in two operands and sets the Sign, Zero, and Parity flags based on the value assigned to the destination operand. The only difference between TEST and AND is that TEST does not modify the destination operand. The TEST instruction permits the same operand combinations as the AND instruction. TEST is particularly valuable for finding out whether individual bits in an operand are set.

Example: Testing Multiple Bits The TEST instruction can check several bits at once. Suppose we want to know whether bit 0 or bit 3 is set in the AL register. We can use the following instruction to find this out:

```
test al,00001001b;              test bits 0 and 3
```

(The value 00001001 in this example is called a *bit mask*.) From the following example data sets, we can infer that the Zero flag is set only when all tested bits are clear:

```
0 0 1 0 0 1 0 1  <- input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 1  <- result: ZF = 0

0 0 1 0 0 1 0 0  <- input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 0  <- result: ZF = 1
```

Flags The TEST instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in the same way as the AND instruction.

6.2.8 CMP Instruction

Having examined all of the bitwise instructions, let's now turn to instructions used in logical (boolean) expressions. The most common boolean expressions involve some type of comparison. The following pseudocode snippets support this idea:

```
if A > B ...
while X > 0 and X < 200  ...
if check_for_error( N ) = true
```

In x86 assembly language we use the CMP instruction to compare integers. Character codes are also integers, so they work with CMP as well. Floating-point values require specialized comparison instructions, which we cover in Chapter 12.

The CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand. Neither operand is modified:

```
CMP destination,source
```

CMP uses the same operand combinations as the AND instruction.

Flags The CMP instruction changes the Overflow, Sign, Zero, Carry, Auxiliary Carry, and Parity flags according to the value the destination operand would have had if actual subtraction had taken place. When two unsigned operands are compared, the Zero and Carry flags indicate the following relations between operands:

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

When two signed operands are compared, the Sign, Zero, and Overflow flags indicate the following relations between operands:

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

CMP is a valuable tool for creating conditional logic structures. When you follow CMP with a conditional jump instruction, the result is the assembly language equivalent of an IF statement.

Examples Let's look at three code fragments showing how flags are affected by the CMP instruction. When AX equals 5 and is compared to 10, the Carry flag is set because subtracting 10 from 5 requires a borrow:

```
mov  ax,5
cmp  ax,10                    ; ZF = 0 and CF = 1
```

Comparing 1000 to 1000 sets the Zero flag because subtracting the source from the destination produces zero:

```
mov  ax,1000
mov  cx,1000
cmp  cx,ax                    ; ZF = 1 and CF = 0
```

Comparing 105 to 0 clears both the Zero and Carry flags because subtracting 0 from 105 generates a positive, nonzero value.

```
mov  si,105
cmp  si,0                     ; ZF = 0 and CF = 0
```

6.2.9 Setting and Clearing Individual CPU Flags

How can you easily set or clear the Zero, Sign, Carry, and Overflow flags? There are several ways, some of which require modifying the destination operand. To set the Zero flag, TEST or AND an operand with Zero; to clear the Zero flag, OR an operand with 1:

```
test  al,0                    ; set Zero flag
and   al,0                    ; set Zero flag
or    al,1                    ; clear Zero flag
```

TEST does not modify the operand, whereas AND does. To set the Sign flag, OR the highest bit of an operand with 1. To clear the Sign flag, AND the highest bit with 0:

```
or   al,80h                   ; set Sign flag
and  al,7Fh                   ; clear Sign flag
```

To set the Carry flag, use the STC instruction; to clear the Carry flag, use CLC:

```
stc                           ; set Carry flag
clc                           ; clear Carry flag
```

To set the Overflow flag, add two positive values that produce a negative sum. To clear the Overflow flag, OR an operand with 0:

```
mov  al,7Fh                    ; AL = +127
inc  al                        ; AL = 80h (-128), OF=1
or   eax,0                     ; clear Overflow flag
```

6.2.10 Boolean Instructions in 64-Bit Mode

For the most part, 64-bit instructions work exactly the same in 64-Bit mode as they do in 32-bit mode. For example, if the source operand is a constant whose size is less than 32 bits and the destination is a 64-bit register or memory operand, all bits in the destination operand are affected:

```
.data
allones QWORD 0FFFFFFFFFFFFFFFFh
.code
    mov  rax,allones           ; RAX = FFFFFFFFFFFFFFFF
    and  rax,80h               ; RAX = 0000000000000080
    mov  rax,allones           ; RAX = FFFFFFFFFFFFFFFF
    and  rax,8080h             ; RAX = 0000000000008080
    mov  rax,allones           ; RAX = FFFFFFFFFFFFFFFF
    and  rax,808080h           ; RAX = 0000000000808080
```

But when the source operand is a 32-bit constant or register, only the lower 32 bits of the destination operand are affected. In the following example, only the lower 32 bits of RAX are modified:

```
mov  rax,allones               ; RAX = FFFFFFFFFFFFFFFF
and  rax,80808080h             ; RAX = FFFFFFFF80808080
```

The same results are true when the destination operand is a memory operand. Clearly, 32-bit operands are a special case that you must consider separately from other operand sizes.

6.2.11 Section Review

1. Write a single instruction using 16-bit operands that clears the high 8 bits of AX and does not change the low 8 bits.
2. Write a single instruction using 16-bit operands that sets the high 8 bits of AX and does not change the low 8 bits.
3. Write a single instruction (other than NOT) that reverses all the bits in EAX.
4. Write instructions that set the Zero flag if the 32-bit value in EAX is even and clear the Zero flag if EAX is odd.
5. Write a single instruction that converts an uppercase character in AL to lowercase but does not modify AL if it already contains a lowercase letter.

6.3 Conditional Jumps

6.3.1 Conditional Structures

There are no explicit high-level logic structures in the x86 instruction set, but you can implement them using a combination of comparisons and jumps. Two steps are involved in executing a conditional statement: First, an operation such as CMP, AND, or SUB modifies the CPU status flags. Second, a conditional jump instruction tests the flags and causes a branch to a new address. Let's look at a couple of examples.

Example 1 The CMP instruction in the following example compares EAX to Zero. The JZ (Jump if zero) instruction jumps to label **L1** if the Zero flag was set by the CMP instruction:

```
        cmp   eax,0
        jz    L1                    ; jump if ZF = 1
        .
        .
L1:
```

Example 2 The AND instruction in the following example performs a bitwise AND on the DL register, affecting the Zero flag. The JNZ (jump if not Zero) instruction jumps if the Zero flag is clear:

```
        and   dl,10110000b
        jnz   L2                    ; jump if ZF = 0
        .
        .
L2:
```

6.3.2 J*cond* Instruction

A *conditional jump instruction* branches to a destination label when a status flag condition is true. Otherwise, if the flag condition is false, the instruction immediately following the conditional jump is executed. The syntax is as follows:

```
Jcond destination
```

cond refers to a flag condition identifying the state of one or more flags. The following examples are based on the Carry and Zero flags:

JC	Jump if carry (Carry flag set)
JNC	Jump if not carry (Carry flag clear)
JZ	Jump if zero (Zero flag set)
JNZ	Jump if not zero (Zero flag clear)

CPU status flags are most commonly set by arithmetic, comparison, and boolean instructions. Conditional jump instructions evaluate the flag states, using them to determine whether or not jumps should be taken.

Using the CMP Instruction Suppose you want to jump to label L1 when EAX equals 5. In the next example, if EAX equals 5, the CMP instruction sets the Zero flag; then, the JE instruction jumps to L1 because the Zero flag is set:

```
cmp  eax,5
je   L1                           ; jump if equal
```

(The JE instruction always jumps based on the value of the Zero flag.) If EAX were not equal to 5, CMP would clear the Zero flag, and the JE instruction would not jump.

In the following example, the JL instruction jumps to label L1 because AX is less than 6:

```
mov  ax,5
cmp  ax,6
jl   L1                          ; jump if less
```

In the following example, the jump is taken because AX is greater than 4:

```
mov  ax,5
cmp  ax,4
jg   L1                          ; jump if greater
```

6.3.3 Types of Conditional Jump Instructions

The x86 instruction set has a large number of conditional jump instructions. They are able to compare signed and unsigned integers and perform actions based on the values of individual CPU flags. The conditional jump instructions can be divided into four groups:

- Jumps based on specific flag values
- Jumps based on equality between operands or the value of (E)CX
- Jumps based on comparisons of unsigned operands
- Jumps based on comparisons of signed operands

Table 6-2 shows a list of jumps based on the Zero, Carry, Overflow, Parity, and Sign flags.

Table 6-2 Jumps Based on Specific Flag Values.

Mnemonic	Description	Flags / Registers
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Equality Comparisons

Table 6-3 lists jump instructions based on evaluating equality. In some cases, two operands are compared; in other cases, a jump is taken based on the value of CX, ECX, or RCX. In the table, the notations *leftOp* and *rightOp* refer to the left (destination) and right (source) operands in a CMP instruction:

```
CMP leftOp,rightOp
```

The operand names reflect the ordering of operands for relational operators in algebra. For example, in the expression X < Y, X is called *leftOp* and Y is called *rightOp*.

Table 6-3 Jumps Based on Equality.

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0
JRCXZ	Jump if RCX = 0 (64-bit mode)

Although the JE instruction is equivalent to JZ (jump if Zero) and JNE is equivalent to JNZ (jump if not Zero), it's best to select the mnemonic (JE or JZ) that best indicates your intention to either compare two operands or examine a specific status flag.

Following are code examples that use the JE, JNE, JCXZ, and JECXZ instructions. Examine the comments carefully to be sure that you understand why the conditional jumps were (or were not) taken.

Example 1:

```
mov  edx,0A523h
cmp  edx,0A523h
jne  L5                          ; jump not taken
je   L1                          ; jump is taken
```

Example 2:

```
mov  bx,1234h
sub  bx,1234h
jne  L5                          ; jump not taken
je   L1                          ; jump is taken
```

Example 3:

```
mov    cx,0FFFFh
inc    cx
jcxz   L2                        ; jump is taken
```

Example 4:

```
xor    ecx,ecx
jecxz  L2                        ; jump is taken
```

Unsigned Comparisons

Jumps based on comparisons of unsigned numbers are shown in Table 6-4. The operand names reflect the order of operands, as in the expression (*leftOp* < *rightOp*). The jumps in Table 6-4 are only meaningful when comparing unsigned values. Signed operands use a different set of jumps.

Signed Comparisons

Table 6-5 displays a list of jumps based on signed comparisons. The following instruction sequence demonstrates the comparison of two signed values:

```
mov  al,+127                     ; hexadecimal value is 7Fh
cmp  al,-128                     ; hexadecimal value is 80h
ja   IsAbove                     ; jump not taken, because 7Fh < 80h
jg   IsGreater                   ; jump taken, because +127 > -128
```

The JA instruction, which is designed for unsigned comparisons, does not jump because unsigned 7Fh is smaller than unsigned 80h. The JG instruction, on the other hand, is designed for signed comparisons—it jumps because +127 is greater than −128.

Table 6-4 Jumps Based on Unsigned Comparisons.

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* ≥ *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* ≤ *rightOp*)
JNA	Jump if not above (same as JBE)

Table 6-5 Jumps Based on Signed Comparisons.

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* ≥ *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* ≤ *rightOp*)
JNG	Jump if not greater (same as JLE)

In the following code examples, examine the comments to be sure you understand why the jumps were (or were not) taken:

Example 1

```
mov   edx,-1
cmp   edx,0
jnl   L5                    ; jump not taken (-1 >= 0 is false)
jnle  L5                    ; jump not taken (-1 > 0 is false)
jl    L1                    ; jump is taken (-1 < 0 is true)
```

Example 2

```
mov   bx,+32
cmp   bx,-35
jng   L5                    ; jump not taken (+32 <= -35 is false)
jnge  L5                    ; jump not taken (+32 < -35 is false)
jge   L1                    ; jump is taken (+32 >= -35 is true)
```

Example 3

```
mov  ecx,0
cmp  ecx,0
jg   L5                        ; jump not taken (0 > 0 is false)
jnl  L1                        ; jump is taken (0 >= 0 is true)
```

Example 4

```
mov  ecx,0
cmp  ecx,0
jl   L5                        ; jump not taken (0 < 0 is false)
jng  L1                        ; jump is taken (0 <= 0 is true)
```

6.3.4 Conditional Jump Applications

Testing Status Bits One of the things assembly language does best is bit testing. Often, we do not want to change the values of the bits we're testing, but we do want to modify the values of CPU status flags. Conditional jump instructions often use these status flags to determine whether or not to transfer control to code labels. Suppose, for example, that an 8-bit memory operand named **status** contains status information about an external device attached to the computer. The following instructions jump to a label if bit 5 is set, indicating that the device is offline:

```
mov   al,status
test  al,00100000b             ; test bit 5
jnz   DeviceOffline
```

The following statements jump to a label if any of the bits 0, 1, or 4 are set:

```
mov   al,status
test  al,00010011b             ; test bits 0,1,4
jnz   InputDataByte
```

Jumping to a label if bits 2, 3, and 7 are all set requires both the AND and CMP instructions:

```
mov   al,status
and   al,10001100b             ; mask bits 2,3,7
cmp   al,10001100b             ; all bits set?
je    ResetMachine             ; yes: jump to label
```

Larger of Two Integers The following code compares the unsigned integers in EAX and EBX and moves the larger of the two to EDX:

```
  mov  edx,eax                 ; assume EAX is larger
  cmp  eax,ebx                 ; if EAX is >= EBX
  jae  L1                      ;    jump to L1
  mov  edx,ebx                 ; else move EBX to EDX
L1:                            ; EDX contains the larger integer
```

Smallest of Three Integers The following instructions compare the unsigned 16-bit values in the variables V1, V2, and V3 and move the smallest of the three to AX:

```
.data
V1 WORD ?
V2 WORD ?
V3 WORD ?
.code
    mov   ax,V1               ; assume V1 is smallest
    cmp   ax,V2               ; if AX <= V2
    jbe   L1                  ;   jump to L1
    mov   ax,V2               ; else move V2 to AX
L1: cmp   ax,V3               ; if AX <= V3
    jbe   L2                  ;   jump to L2
    mov   ax,V3               ; else move V3 to AX
L2:
```

Loop until Key Pressed In the following 32-bit code, a loop runs continuously until the user presses a standard alphanumeric key. The *ReadKey* method from the Irvine32 library sets the Zero flag if no key is present in the input buffer:

```
.data
char BYTE ?
.code
L1: mov  eax,10               ; create 10 ms delay
    call Delay
    call ReadKey              ; check for key
    jz   L1                   ; repeat if no key
    mov  char,AL              ; save the character
```

The foregoing code inserts a 10-millisecond delay in the loop to give MS-Windows time to process event messages. If you omit the delay, keystrokes may be ignored.

Application: Sequential Search of an Array

A common programming task is to search for values in an array that meet some criteria. For example, the following program looks for the first nonzero value in an array of 16-bit integers. If it finds one, it displays the value; otherwise, it displays a message stating that a nonzero value was not found:

```
; Scanning an Array                    (ArrayScan.asm)
; Scan an array for the first nonzero value.

INCLUDE Irvine32.inc

.data
intArray  SWORD  0,0,0,0,1,20,35,-12,66,4,0
;intArray SWORD  1,0,0,0              ; alternate test data
;intArray SWORD  0,0,0,0              ; alternate test data
;intArray SWORD  0,0,0,1              ; alternate test data
noneMsg  BYTE "A non-zero value was not found",0
```

This program contains alternate test data that are currently commented out. Uncomment each of these lines to test the program with different data configurations.

```
.code
main PROC
     mov   ebx,OFFSET intArray       ; point to the array
     mov   ecx,LENGTHOF intArray     ; loop counter

L1:  cmp   WORD PTR [ebx],0          ; compare value to zero
     jnz   found                     ; found a value
     add   ebx,2                     ; point to next
     loop  L1                        ; continue the loop
     jmp   notFound                  ; none found

found:                               ; display the value
     movsx eax,WORD PTR[ebx]         ; sign-extend into EAX
     call  WriteInt
     jmp   quit

notFound:                            ; display "not found" message
     mov   edx,OFFSET noneMsg
     call  WriteString

quit:
     call Crlf
     exit
main ENDP
END main
```

Application: Simple String Encryption

The XOR instruction has an interesting property. If an integer X is XORed with Y and the resulting value is XORed with Y again, the value produced is X:

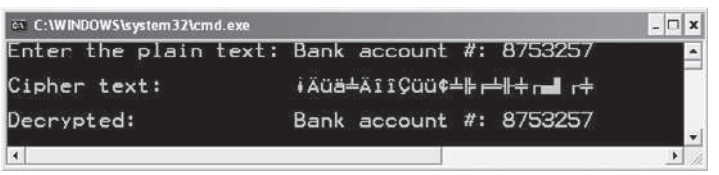
$$((X \otimes Y) \otimes Y) = X$$

This reversible property of XOR provides an easy way to perform a simple form of data encryption: A *plain text* message is transformed into an encrypted string called *cipher text by* XORing each of its characters with a character from a third string called a *key*. The intended viewer can use the key to decrypt the cipher text and produce the original plain text.

Example Program We will demonstrate a simple program that uses *symmetric encryption*, a process by which the same key is used for both encryption and decryption. The following steps occur in order at runtime:

1. The user enters the plain text.
2. The program uses a single-character key to encrypt the plain text, producing the cipher text, which is displayed on the screen.
3. The program decrypts the cipher text, producing and displaying the original plain text.

Here is sample output from the program:



Program Listing Here is a complete program listing:

```
; Encryption Program                    (Encrypt.asm)

INCLUDE Irvine32.inc
KEY = 239                       ; any value between 1-255
BUFMAX = 128                    ; maximum buffer size

.data
sPrompt  BYTE "Enter the plain text:",0
sEncrypt BYTE "Cipher text:    ",0
sDecrypt BYTE "Decrypted:      ",0
buffer   BYTE  BUFMAX+1 DUP(0)
bufSize  DWORD ?

.code
main PROC
    call  InputTheString        ; input the plain text
    call  TranslateBuffer       ; encrypt the buffer
    mov   edx,OFFSET sEncrypt   ; display encrypted message
    call  DisplayMessage
    call  TranslateBuffer       ; decrypt the buffer
    mov   edx,OFFSET sDecrypt   ; display decrypted message
    call  DisplayMessage
    exit
main ENDP

;-----------------------------------------------------
InputTheString PROC
;
; Prompts user for a plaintext string. Saves the string
; and its length.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------
    pushad                      ; save 32-bit registers
    mov   edx,OFFSET sPrompt    ; display a prompt
    call  WriteString
    mov   ecx,BUFMAX            ; maximum character count
    mov   edx,OFFSET buffer     ; point to the buffer
    call  ReadString            ; input the string
    mov   bufSize,eax           ; save the length
    call  Crlf
    popad
    ret
InputTheString ENDP

;-----------------------------------------------------
DisplayMessage PROC
;
; Displays the encrypted or decrypted message.
; Receives: EDX points to the message
; Returns:  nothing
;-----------------------------------------------------
```

```
    pushad
    call  WriteString
    mov   edx,OFFSET buffer    ; display the buffer
    call  WriteString
    call  Crlf
    call  Crlf
    popad
    ret
DisplayMessage ENDP

;-----------------------------------------------------
TranslateBuffer PROC
;
; Translates the string by exclusive-ORing each
; byte with the encryption key byte.
; Receives: nothing
; Returns: nothing
;-----------------------------------------------------

    pushad
    mov   ecx,bufSize          ; loop counter
    mov   esi,0                ; index 0 in buffer
L1:
    xor   buffer[esi],KEY      ; translate a byte
    inc   esi                  ; point to next byte
    loop  L1
    popad
    ret
TranslateBuffer ENDP
END main
```

You should never encrypt important data with a single-character encryption key, because it can be too easily decoded. Instead, the chapter exercises suggest that you use an encryption key containing multiple characters to encrypt and decrypt the plain text.

6.3.5 Section Review

1. Which jump instructions follow unsigned integer comparisons?
2. Which jump instructions follow signed integer comparisons?
3. Which conditional jump instruction is equivalent to JNAE?
4. Which conditional jump instruction is equivalent to the JNA instruction?
5. Which conditional jump instruction is equivalent to the JNGE instruction?
6. *(Yes/No):* Will the following code jump to the label named **Target**?

```
mov ax,8109h
cmp ax,26h
jg  Target
```

6.4 Conditional Loop Instructions

6.4.1 LOOPZ and LOOPE Instructions

The LOOPZ (loop if zero) instruction works just like the LOOP instruction except that it has one additional condition: the Zero flag must be set in order for control to transfer to the destination label. The syntax is

```
LOOPZ destination
```

The LOOPE (loop if equal) instruction is equivalent to LOOPZ, and they share the same opcode. They perform the following tasks:

```
ECX = ECX - 1
if ECX > 0 and ZF = 1, jump to destination
```

Otherwise, no jump occurs, and control passes to the next instruction. LOOPZ and LOOPE do not affect any of the status flags. In 32-bit mode, ECX is the loop counter register, and in 64-bit mode, RCX is the counter.

6.4.2 LOOPNZ and LOOPNE Instructions

The LOOPNZ (loop if not zero) instruction is the counterpart of LOOPZ. The loop continues while the unsigned value of ECX is greater than zero (after being decremented) and the Zero flag is clear. The syntax is

```
LOOPNZ destination
```

The LOOPNE (loop if not equal) instruction is equivalent to LOOPNZ, and they share the same opcode. They perform the following tasks:

```
ECX = ECX - 1
if ECX > 0 and ZF = 0, jump to destination
```

Otherwise, nothing happens, and control passes to the next instruction.

Example The following code excerpt (from *Loopnz.asm*) scans each number in an array until a nonnegative number is found (when the sign bit is clear). Notice that we push the flags on the stack before the ADD instruction because ADD will modify the flags. Then the flags are restored by POPFD just before the LOOPNZ instruction executes:

```
.data
array  SWORD  -3,-6,-1,-10,10,30,40,4
sentinel SWORD  0
.code
    mov    esi,OFFSET array
    mov    ecx,LENGTHOF array

L1: test   WORD PTR [esi],8000h       ; test sign bit
    pushfd                            ; push flags on stack
    add    esi,TYPE array             ; move to next position
    popfd                             ; pop flags from stack
    loopnz  L1                        ; continue loop
```

```
        jnz     quit                        ; none found
        sub     esi,TYPE array              ; ESI points to value
quit:
```

If a nonnegative value is found, ESI is left pointing at it. If the loop fails to find a positive number, it stops when ECX equals zero. In that case, the JNZ instruction jumps to label **quit**, and ESI points to the sentinel value (0), located in memory immediately following the array.

6.4.3 Section Review

1. *(True/False):* The LOOPE instruction jumps to a label when (and only when) the Zero flag is clear.
2. *(True/False):* In 32-bit mode, the LOOPNZ instruction jumps to a label when ECX is greater than zero and the Zero flag is clear.
3. *(True/False):* The destination label of a LOOPZ instruction must be no farther than −128 or +127 bytes from the instruction immediately following LOOPZ.
4. Modify the LOOPNZ example in Section 6.4.2 so that it scans for the first negative value in the array. Change the array initializers so they begin with positive values.
5. *Challenge:* The LOOPNZ example in Section 6.4.2 relies on a sentinel value to handle the possibility that a positive value might not be found. What might happen if you removed the sentinel?

6.5 Conditional Structures

We define a *conditional structure* to be one or more conditional expressions that trigger a choice between different logical branches. Each branch causes a different sequence of instructions to execute. No doubt you have already used conditional structures in a high-level programming language. But you may not know how language compilers translate conditional structures into low-level machine code. Let's find out how that is done.

6.5.1 Block-Structured IF Statements

An IF structure impli that a boolean expression is followed by two lists of statements; one performed when the expression is true, and another performed when the expression is false:

```
if( boolean-expression )
  statement-list-1
else
  statement-list-2
```

The **else** portion of the statement is optional. In assembly language, we code this structure in steps. First, we evaluate the boolean expression in such a way that one of the CPU status flags is affected. Second, we construct a series of jumps that transfer control to the two lists of statements, based on the value of the relevant CPU status flag.

Example 1 In the following C++ code, two assignment statements are executed if **op1** is equal to **op2**:

```
if( op1 == op2 )
{
     X = 1;
     Y = 2;
}
```

We translate this IF statement into assembly language with a CMP instruction followed by conditional jumps. Because **op1** and **op2** are memory operands (variables), one of them must be moved to a register before executing CMP. The following code implements the IF statement as efficiently as possible by allowing the code to "fall through" to the two MOV instructions that we want to execute when the boolean condition is true:

```
    mov   eax,op1
    cmp   eax,op2              ; op1 == op2?
    jne   L1                   ; no: skip next
    mov   X,1                  ; yes: assign X and Y
    mov   Y,2
L1:
```

If we implemented the == operator using JE, the resulting code would be slightly less compact (six instructions rather than five):

```
    mov   eax,op1
    cmp   eax,op2              ; op1 == op2?
    je    L1                   ; yes: jump to L1
    jmp   L2                   ; no: skip assignments
L1: mov   X,1                  ; assign X and Y
    mov   Y,2
L2:
```

> As you see from the foregoing example, the same conditional structure can be translated into assembly language in multiple ways. When examples of compiled code are shown in this chapter, they represent only what a hypothetical compiler might produce.

Example 2 In the NTFS file storage system, the size of a disk cluster depends on the disk volume's overall capacity. In the following pseudocode, we set the cluster size to 4,096 if the volume size (in the variable named **terrabytes**) is less than 16 TBytes. Otherwise, we set the cluster size to 8,192:

```
clusterSize = 8192;
if terrabytes < 16
  clusterSize = 4096;
```

Here's a way to implement the pseudocode in assembly language:

```
    mov    clusterSize,8192    ; assume larger cluster
    cmp    terrabytes, 16      ; smaller than 16 TB?
    jae    next
    mov    clusterSize,4096    ; switch to smaller cluster
next:
```

Example 3 The following pseudocode statement has two branches:

```
if op1 > op2
    call Routine1
else
    call Routine2
end if
```

In the following assembly language translation of the pseudocode, we assume that **op1** and **op2** are signed doubleword variables. When comparing variables, one must be moved to a register:

```
      mov  eax,op1                ; move op1 to a register
      cmp  eax,op2                ; op1 > op2?
      jg   A1                     ; yes: call Routine1
      call Routine2               ; no: call Routine2
      jmp  A2                     ; exit the IF statement
A1:   call Routine1
A2:
```

White Box Testing

Complex conditional statements may have multiple execution paths, making them hard to debug by inspection (looking at the code). Programmers often implement a technique known as *white box testing*, which verifies a subroutine's inputs and corresponding outputs. White box testing requires you to have a copy of the source code. You assign a variety of values to the input variables. For each combination of inputs, you manually trace through the source code and verify the execution path and outputs produced by the subroutine. Let's see how this is done in assembly language by implementing the following nested-IF statement:

```
if op1 == op2
  if X > Y
    call Routine1
  else
    call Routine2
  end if
else
  call Routine3
end if
```

Following is a possible translation to assembly language, with line numbers added for reference. It reverses the initial condition (op1 == op2) and immediately jumps to the ELSE portion. All that is left to translate is the inner IF-ELSE statement:

```
1:       mov   eax,op1
2:       cmp   eax,op2                ; op1 == op2?
3:       jne   L2                     ; no: call Routine3

; process the inner IF-ELSE statement.
4:       mov   eax,X
5:       cmp   eax,Y                  ; X > Y?
6:       jg    L1                     ; yes: call Routine1
7:       call  Routine2               ; no: call Routine2
8:       jmp   L3                     ; and exit
9:   L1: call  Routine1               ; call Routine1
10:      jmp   L3                     ; and exit
11:  L2: call  Routine3
12:  L3:
```

Table 6-6 shows the results of white box testing of the sample code. In the first four columns, test values have been assigned to op1, op2, X, and Y. The resulting execution paths are verified in columns 5 and 6.

Table 6-6 Testing the Nested IF Statement.

op1	op2	X	Y	Line Execution Sequence	Calls
10	20	30	40	1, 2, 3, 11, 12	Routine3
10	20	40	30	1, 2, 3, 11, 12	Routine3
10	10	30	40	1, 2, 3, 4, 5, 6, 7, 8, 12	Routine2
10	10	40	30	1, 2, 3, 4, 5, 6, 9, 10, 12	Routine1

6.5.2 Compound Expressions

Logical AND Operator

Assembly language easily implements compound boolean expressions containing AND operators. Consider the following pseudocode, in which the values being compared are assumed to be unsigned integers:

```
if (a1 > b1) AND (b1 > c1)
    X = 1
end if
```

Short-Circuit Evaluation The following is a straightforward implementation using *short-circuit* evaluation, in which the second expression is not evaluated if the first expression is false. This is the norm for high-level languages:

```
    cmp   al,bl              ; first expression...
    ja    L1
    jmp   next
L1: cmp   bl,cl              ; second expression...
    ja    L2
    jmp   next
L2: mov   X,1                ; both true: set X to 1
next:
```

We can reduce the code to five instructions by changing the initial JA instruction to JBE:

```
    cmp   al,bl              ; first expression...
    jbe   next               ; quit if false
    cmp   bl,cl              ; second expression
    jbe   next               ; quit if false
    mov   X,1                ; both are true
next:
```

The 29% reduction in code size (seven instructions down to five) results from letting the CPU fall through to the second CMP instruction if the first JBE is not taken.

Logical OR Operator

When a compound expression contains subexpressions joined by the OR operator, the overall expression is true if any of the subexpressions is true. Let's use the following pseudocode as an example:

```
if (a1 > b1) OR (b1 > c1)
   X = 1
```

In the following implementation, the code branches to L1 if the first expression is true; otherwise, it falls through to the second CMP instruction. The second expression reverses the > operator and uses JBE instead:

```
     cmp   al,bl               ; 1: compare AL to BL
     ja    L1                  ; if true, skip second expression
     cmp   bl,cl               ; 2: compare BL to CL
     jbe   next                ; false: skip next statement
L1:  mov   X,1                 ; true: set X = 1
next:
```

For a given compound expression, there are multiple ways the expression can be implemented in assembly language.

6.5.3 WHILE Loops

A WHILE loop tests a condition first before performing a block of statements. As long as the loop condition remains true, the statements are repeated. The following loop is written in C++:

```
while( val1 < val2 )
{
     val1++;
     val2--;
}
```

When implementing this structure in assembly language, it is convenient to reverse the loop condition and jump to **endwhile** if a condition becomes true. Assuming that **val1** and **val2** are variables, we must copy one of them to a register at the beginning and restore the variable's value at the end:

```
     mov   eax,val1            ; copy variable to EAX
beginwhile:
     cmp   eax,val2            ; if not (val1 < val2)
     jnl   endwhile            ;   exit the loop
     inc   eax                 ; val1++;
     dec   val2                ; val2--;
     jmp   beginwhile          ; repeat the loop
endwhile:
     mov   val1,eax            ; save new value for val1
```

EAX is a proxy (substitute) for **val1** inside the loop. References to **val1** must be through EAX. JNL is used, implying that **val1** and **val2** are signed integers.

Example: IF statement Nested in a Loop

High-level languages are particularly good at representing nested control structures. In the following C++ code, an IF statement is nested inside a WHILE loop. It calculates the sum of all array elements greater than the value in **sample**:

```
int array[] = {10,60,20,33,72,89,45,65,72,18};
int sample = 50;
int ArraySize = sizeof array / sizeof sample;
int index = 0;
int sum = 0;
while( index < ArraySize )
{
   if( array[index] > sample )
   {
      sum += array[index];
   }
   index++;
}
```

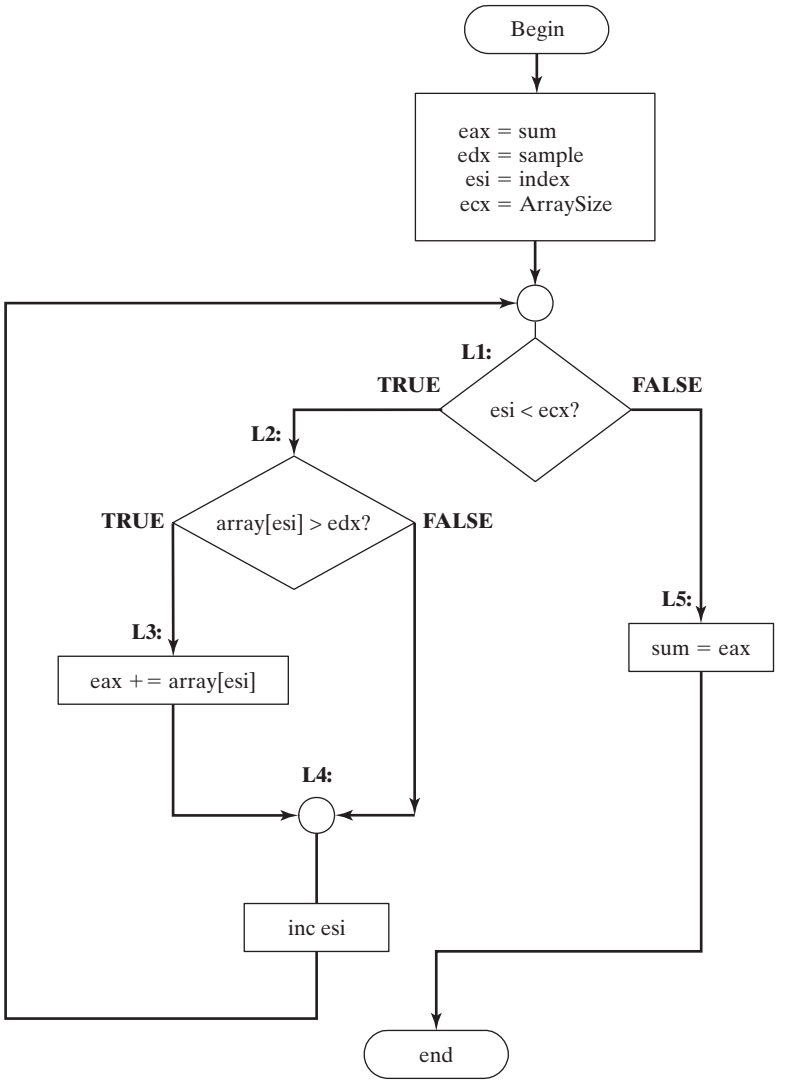
Before coding this loop in assembly language, let's use the flowchart in Fig. 6-1 to describe the logic. To simplify the translation and speed up execution by reducing the number of memory accesses, registers have been substituted for variables. EDX = sample, EAX = sum, ESI = index, and ECX = ArraySize (a constant). Label names have been added to the shapes.

Assembly Code The easiest way to generate assembly code from a flowchart is to implement separate code for each flowchart shape. Note the direct correlation between the flowchart labels and labels used in the following source code (see *Flowchart.asm*):

```
.data
sum DWORD 0
sample DWORD 50
array DWORD 10,60,20,33,72,89,45,65,72,18
ArraySize = ($ - Array) / TYPE array

.code
main PROC
     mov   eax,0                ; sum
     mov   edx,sample
     mov   esi,0                ; index
     mov   ecx,ArraySize

L1:  cmp   esi,ecx              ; if esi < ecx
     jl    L2
     jmp   L5

L2:  cmp   array[esi*4], edx    ; if array[esi] > edx
     jg    L3
     jmp   L4
L3:  add   eax,array[esi*4]

L4:  inc   esi
     jmp   L1

L5:  mov   sum,eax
```

A review question at the end of Section 6.5 will give you a chance to improve this code.

Figure 6–1 Loop containing IF statement.



6.5.4 Table-Driven Selection

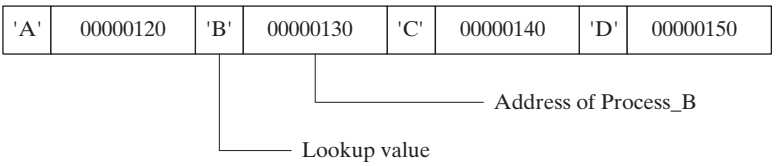
Table-driven selection is a way of using a table lookup to replace a multiway selection structure. To use it, you must create a table containing lookup values and the offsets of labels or procedures, and then you must use a loop to search the table. This works best when a large number of comparisons are made.

For example, the following is part of a table containing single-character lookup values and addresses of procedures:

```
.data
CaseTable BYTE   'A'            ; lookup value
    DWORD Process_A             ; address of procedure
    BYTE  'B'
    DWORD Process_B
     (etc.)
```

Let's assume Process_A, Process_B, Process_C, and Process_D are located at addresses 120h, 130h, 140h, and 150h, respectively. The table would be arranged in memory as shown in Fig. 6–2.

Figure 6–2 Table of procedure offsets.



Example Program In the following example program (*ProcTable.asm*), the user inputs a character from the keyboard. Using a loop, the character is compared to each entry in a lookup table. The first match found in the table causes a call to the procedure offset stored immediately after the lookup value. Each procedure loads EDX with the offset of a different string, which is displayed during the loop:

```
; Table of Procedure Offsets              (ProcTable.asm)

; This program contains a table with offsets of procedures.
; It uses the table to execute indirect procedure calls.

INCLUDE Irvine32.inc
.data
CaseTable BYTE 'A'                    ; lookup value
          DWORD   Process_A           ; address of procedure
EntrySize = ($ - CaseTable)
           BYTE 'B'
           DWORD   Process_B
           BYTE 'C'
           DWORD   Process_C
           BYTE 'D'
           DWORD   Process_D
NumberOfEntries = ($ - CaseTable) / EntrySize
prompt BYTE "Press capital A,B,C,or D: ",0
```

Define a separate message string for each procedure:

```
msgA BYTE "Process_A",0
msgB BYTE "Process_B",0
msgC BYTE "Process_C",0
msgD BYTE "Process_D",0
```

```
.code
main PROC
     mov   edx,OFFSET prompt          ; ask user for input
     call  WriteString
     call  ReadChar                   ; read character into AL
     mov   ebx,OFFSET CaseTable       ; point EBX to the table
     mov   ecx,NumberOfEntries        ; loop counter
L1:
     cmp   al,[ebx]                   ; match found?
     jne   L2                         ; no: continue
     call  NEAR PTR [ebx + 1]         ; yes: call the procedure
```

This CALL instruction calls the procedure whose address is stored in the memory location referenced by EBX+1. An indirect call such as this requires the NEAR PTR operator.

```
     call  WriteString                ; display message
     call  Crlf
     jmp   L3                         ; exit the search
L2:
     add   ebx,EntrySize              ; point to the next entry
     loop  L1                         ; repeat until ECX = 0

L3:
     exit
main ENDP
```

Each of the following procedures moves a different string offset to EDX:

```
Process_A PROC
     mov   edx,OFFSET msgA
     ret
Process_A ENDP

Process_B PROC
     mov   edx,OFFSET msgB
     ret
Process_B ENDP

Process_C PROC
     mov   edx,OFFSET msgC
     ret
Process_C ENDP

Process_D PROC
     mov   edx,OFFSET msgD
     ret
Process_D ENDP
END main
```

The table-driven selection method involves some initial overhead, but it can reduce the amount of code you write. A table can handle a large number of comparisons, and it can be more

easily modified than a long series of compare, jump, and CALL instructions. A table can even be reconfigured at runtime.

6.5.5 Section Review

Notes: In all compound expressions, use short-circuit evaluation. Assume that **val1** and **X** are 32-bit variables.

1. Implement the following pseudocode in assembly language:

```
if ebx > ecx
   X = 1
```

2. Implement the following pseudocode in assembly language:

```
if edx <= ecx
   X = 1
else
   X = 2
```

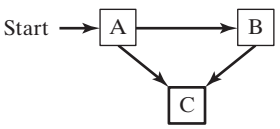
3. In the program from Section 6.5.4, why is it better to let the assembler calculate NumberOfEntries rather than assigning a constant such as NumberOfEntries = 4?
4. *Challenge:* Rewrite the code from Section 6.5.3 so it is functionally equivalent, but uses fewer instructions.

6.6 Application: Finite-State Machines

A *finite-state machine* (FSM) is a machine or program that changes state based on some input. It is fairly simple to use a graph to represent an FSM, which contains squares (or circles) called *nodes* and lines with arrows between the circles called *edges (or arcs)*.

A simple example is shown in Figure 6-3. Each node represents a program state, and each edge represents a transition from one state to another. One node is designated as the *initial state*, shown in our diagram with an incoming arrow. The remaining states can be labeled with numbers or letters. One or more states are designated as *terminal states*, shown by a thick border around the square. A terminal state represents a state in which the program might stop without producing an error. A FSM is a specific instance of a more general type of structure called a *directed graph*. The latter is a set of nodes connected by edges having specific directions.

Figure 6–3 Simple finite-state machine.



6.6.1 Validating an Input String

Programs that read input streams often must validate their input by performing a certain amount of error checking. A programming language compiler, for instance, can use a FSM to scan source programs and convert words and symbols into *tokens*, which are usually keywords, arithmetic operators, and identifiers.

When using a FSM to check the validity of an input string, you usually read the input character by character. Each character is represented by an edge (transition) in the diagram. A FSM detects illegal input sequences in one of two ways:

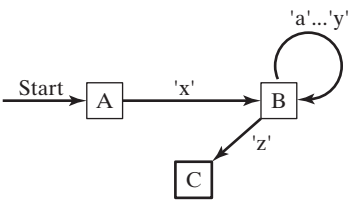
- The next input character does not correspond to any transitions from the current state.
- The end of input is reached and the current state is a nonterminal state.

Character String Example Let's check the validity of an input string according to the following two rules:

- The string must begin with the letter "x" and end with the letter "z."
- Between the first and last characters, there can be zero or more letters within the range {'a'...'y'}.

The FSM diagram in Fig. 6-4 describes this syntax. Each transition is identified with a particular type of input. For example, the transition from state A to state B can only be accomplished if the letter **x** is read from the input stream. A transition from state B to itself is accomplished by the input of any letter of the alphabet except **z**. A transition from state B to state C occurs only when the letter **z** is read from the input stream.

FIGURE 6–4 FSM for string.



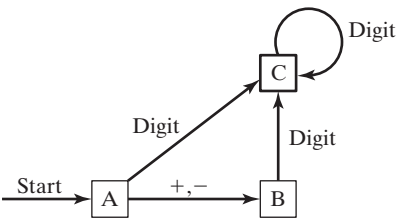
If the end of the input stream is reached while the program is in state A or B, an error condition results because only state C is marked as a terminal state. The following input strings would be recognized by this FSM:

```
xaabcdefgz
xz
xyyqqrrstuvz
```

6.6.2 Validating a Signed Integer

A FSM for parsing a signed integer is shown in Fig. 6-5. Input consists of an optional leading sign followed by a sequence of digits. There is no maximum number of digits implied by the diagram.

FIGURE 6–5 Signed decimal integer FSM.



Finite-state machines are easily translated into assembly language code. Each state in the diagram (A, B, C, . . .) is represented in the program by a label. The following actions are performed at each label:

1. A call to an input procedure reads the next character from input.
2. If the state is a terminal state, check to see whether the user has pressed the Enter key to end the input.
3. One or more compare instructions check for each possible transition leading away from the state. Each comparison is followed by a conditional jump instruction.

For example, at state A, the following code reads the next input character and checks for a possible transition to state B:

```
StateA:
     call  Getnext              ; read next char into AL
     cmp   al,'+'               ; leading + sign?
     je    StateB               ; go to State B
     cmp   al,'-'               ; leading - sign?
     je    StateB               ; go to State B
     call  IsDigit              ; ZF = 1 if AL contains a digit
     jz    StateC               ; go to State C
     call  DisplayErrorMsg      ; invalid input found
     jmp   Quit
```

Let's examine this code in more detail. First, it calls *Getnext* to read the next character from the console input into the AL register. The code will check for a leading + or – sign. It begins by comparing the value in AL to a "+" character. If the character matches, a jump is taken to the label named *StateB:*

```
StateA:
     call  Getnext              ; read next char into AL
     cmp   al,'+'               ; leading + sign?
     je    StateB               ; go to State B
```

At this point, we should look again at Fig. 6-5, and see that the transition from state A to state B can only be made if a + or – character is read from input. Therefore, the code must also check for the minus sign:

```
     cmp   al,'-'               ; leading - sign?
     je    StateB               ; go to State B
```

If a transition to state B is not possible, we can check the AL register for a digit, which would cause a transition to state C. The call to the *IsDigit* procedure (from the book's link library) sets the Zero flag if AL contains a digit:

```
     call  IsDigit              ; ZF = 1 if AL contains a digit
     jz    StateC               ; go to State C
```

Finally, there are no other possible transitions away from state A. If the character in AL has not been found to be a leading sign or digit, the program calls *DisplayErrorMsg* (which displays an error message on the console) and then jumps to the label named Quit:

```
     call  DisplayErrorMsg      ; invalid input found
     jmp   Quit
```

The label *Quit* marks the exit point of the program, at the end of the main procedure:

```
Quit:
     call  Crlf
     exit
main ENDP
```

Complete Finite-State Machine Program The following program implements the signed integer FSM from Fig. 6-5:

```
; Finite State Machine              (Finite.asm)

INCLUDE Irvine32.inc

ENTER_KEY = 13
.data
InvalidInputMsg BYTE "Invalid input",13,10,0

.code
main PROC
     call Clrscr

StateA:
     call  Getnext              ; read next char into AL
     cmp   al,'+'               ; leading + sign?
     je    StateB               ; go to State B
     cmp   al,'-'               ; leading - sign?
     je    StateB               ; go to State B
     call  IsDigit              ; ZF = 1 if AL contains a digit
     jz    StateC               ; go to State C
     call  DisplayErrorMsg      ; invalid input found
     jmp   Quit

StateB:
     call  Getnext              ; read next char into AL
     call  IsDigit              ; ZF = 1 if AL contains a digit
     jz    StateC
     call  DisplayErrorMsg      ; invalid input found
     jmp   Quit

StateC:
     call  Getnext              ; read next char into AL
     call  IsDigit              ; ZF = 1 if AL contains a digit
     jz    StateC
     cmp   al,ENTER_KEY         ; Enter key pressed?
     je    Quit                 ; yes: quit
     call  DisplayErrorMsg      ; no: invalid input found
     jmp   Quit

Quit:
     call  Crlf
     exit
main ENDP

;-----------------------------------------------
```

```
Getnext PROC
;
; Reads a character from standard input.
; Receives: nothing
; Returns: AL contains the character
;-----------------------------------------------
     call  ReadChar              ; input from keyboard
     call  WriteChar             ; echo on screen
     ret
Getnext ENDP
;-----------------------------------------------
DisplayErrorMsg PROC
;
; Displays an error message indicating that
; the input stream contains illegal input.
; Receives: nothing.
; Returns: nothing
;-----------------------------------------------
     push  edx
     mov   edx,OFFSET InvalidInputMsg
     call  WriteString
     pop   edx
     ret
DisplayErrorMsg ENDP
END main
```

IsDigit Procedure The Finite-State Machine sample program calls the *IsDigit* procedure, which belongs to the book's link library. Let's look at the source code for IsDigit. It receives the AL register as input, and the value it returns is the setting of the Zero flag:

```
;-----------------------------------------------------------------------
IsDigit PROC
;
; Determines whether the character in AL is a valid decimal digit.
; Receives: AL = character
; Returns: ZF = 1 if AL contains a valid decimal digit; otherwise, ZF = 0.
;-----------------------------------------------------------------------
     cmp   al,'0'
     jb    ID1                   ; ZF = 0 when jump taken
     cmp   al,'9'
     ja    ID1                   ; ZF = 0 when jump taken
     test  ax,0                  ; set ZF = 1
ID1: ret
IsDigit ENDP
```

Before examining the code in IsDigit, we can review the set of hexadecimal ASCII codes for decimal digits, shown in the following table. Because the values are contiguous, we need only to check for the starting and ending range values:

Character	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code (hex)	30	31	32	33	34	35	36	37	38	39

In the IsDigit procedure, the first two instructions compare the character in the AL register to the ASCII code for the digit 0. If the numeric ASCII code of the character is less than the ASCII code for 0, the program jumps to the label ID1:

```
cmp  al,'0'
jb   ID1                          ; ZF = 0 when jump taken
```

But one may ask, if JB transfers control to the label named ID1, how do we know the state of the Zero flag? The answer lies in the way CMP works—it carries out an implied subtraction of the ASCII code for Zero (30h) from the character in the AL register. If the value in AL is smaller, the Carry flag is set, and the Zero flag is clear. (You may want to step through this code with a debugger to verify this fact.) The JB instruction is designed to transfer control to a label when CF = 1 and ZF = 0.

Next, the code in the IsDigit procedure compares AL to the ASCII code for the digit 9. If the value is greater, the code jumps to the same label:

```
cmp  al,'9'
ja   ID1                          ; ZF = 0 when jump taken
```

If the ASCII code for the character in AL is larger than the ASCII code of the digit 9 (39h), the Carry flag and Zero flag are cleared. That is exactly the flag combination that causes the JA instruction to transfer control to its target label.

If neither jump is taken (JA or JB), we assume that the character in AL is indeed a digit. Therefore, we insert an instruction that is guaranteed to set the Zero flag. To test any value with zero means to perform an implied AND with all zero bits. The result must be zero:

```
test  ax,0                        ; set ZF = 1
```

The JB and JA instructions we looked at earlier in IsDigit jumped to a label that was just beyond the TEST instruction. So if those jumps are taken, the Zero flag will be clear. Here is the complete procedure one more time:

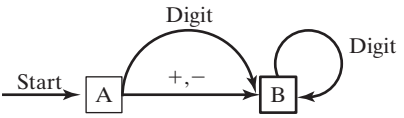
```
Isdigit PROC
    cmp  al,'0'
    jb   ID1                      ; ZF = 0 when jump taken
    cmp  al,'9'
    ja   ID1                      ; ZF = 0 when jump taken
    test ax,0                     ; set ZF = 1
ID1: ret
Isdigit ENDP
```

In real-time or high-performance applications, programmers often take advantage of hardware characteristics to fully optimize their code. The IsDigit procedure is an example of this approach because it uses the flag settings of JB, JA, and TEST to return what is essentially a Boolean result.

6.6.3 Section Review

1. A finite-state machine is a specific application of what type of data structure?
2. In a finite-state machine diagram, what do the nodes represent?
3. In a finite-state machine diagram, what do the edges represent?

4. In the signed integer finite-state machine (Section 6.6.2), which state is reached when the input consists of "+5"?
5. In the signed integer finite-state machine (Section 6.6.2), how many digits can occur after a minus sign?
6. What happens in a finite-state machine when no more input is available and the current state is a nonterminal state?
7. Would the following simplification of a signed decimal integer finite-state machine work just as well as the one shown in Section 6.6.2? If not, why not?



6.7 Conditional Control Flow Directives

In 32-bit mode, MASM includes a number of high-level c*onditional control flow directives* that help to simplify the coding of conditional statements. Unfortunately, they cannot be used in 64-bit mode. Before assembling your code, the assembler performs a preprocessing step. In this step, it recognizes directives such as .CODE, .DATA, as well as directives that can be used for conditional control flow. Table 6-7 lists the directives.

Table 6-7 Conditional Control Flow Directives.

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF *condition*	Generates code that tests *condition* and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF *condition*	Generates code that executes the block of statements if *condition* is true.
.REPEAT	Generates code that repeats execution of the block of statements until *condition* becomes true
.UNTIL *condition*	Generates code that repeats the block of statements between .REPEAT and .UNTIL until *condition* becomes true
.UNTILCXZ	Generates code that repeats the block of statements between .REPEAT and .UNTILCXZ until *CX* equals zero
.WHILE *condition*	Generates code that executes the block of statements between .WHILE and .ENDW as long as *condition* is true

6.7.1 Creating IF Statements

The .IF, .ELSE, .ELSEIF, and .ENDIF directives make it easy for you to code multiway branching logic. They cause the assembler to generate CMP and conditional jump instructions in the background, which appear in the output listing file (*progname*.lst). This is the syntax:

```
.IF condition1
    statements
[.ELSEIF condition2
    statements ]
[.ELSE
    statements ]
.ENDIF
```

The square brackets show that .ELSEIF and .ELSE are optional, whereas .IF and .ENDIF are required. A *condition* is a boolean expression involving the same operators used in C++ and Java (such as <, >, ==, and !=). The expression is evaluated at runtime. The following are examples of valid conditions, using 32-bit registers and variables:

```
eax > 10000h
val1 <= 100
val2 == eax
val3 != ebx
```

The following are examples of compound conditions:

```
(eax > 0) && (eax > 10000h)
(val1 <= 100) || (val2 <= 100)
(val2 != ebx) && !CARRY?
```

A complete list of relational and logical operators is shown in Table 6-8.

Table 6-8 Runtime Relational and Logical Operators.

Operator	Description
expr1 == *expr2*	Returns true when *expr1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 ≥ *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 ≤ *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

Before using MASM conditional directives, be sure you thoroughly understand how to implement conditional branching instructions in pure assembly language. In addition, when a program containing decision directives is assembled, inspect the listing file to make sure the code generated by MASM is what you intended.

Generating ASM Code When you use high-level directives such as .IF and .ELSE, the assembler writes code for you. For example, let's write an .IF directive that compares EAX to the variable **val1**:

```
mov eax,6
.IF eax > val1
  mov result,1
.ENDIF
```

val1 and **result** are assumed to be 32-bit unsigned integers. When the assembler reads the foregoing lines, it expands them into the following assembly language instructions, which you can view if you run the program in the Visual Studio debugger, right-click, and select *Go to Disassembly.*

```
    mov   eax,6
    cmp   eax,val1
    jbe   @C0001              ; jump on unsigned comparison
    mov   result,1
@C0001:
```

The label name @C0001 was created by the assembler. This is done in a way that guarantees that all labels within same procedure are unique.

To control whether or not MASM-generated code appears in the source listing file, you can configure the Project properties in Visual Studio. Here's how: from the Project menu, select *Project Properties*, select *Microsoft Macro Assembler*, select *Listing File*, and set *Enable Assembly Generated Code Listing* to Yes.

6.7.2 Signed and Unsigned Comparisons

When you use the .IF directive to compare values, you must be aware of how MASM generates conditional jumps. If the comparison involves an unsigned variable, an unsigned conditional jump instruction is inserted in the generated code. This is a repeat of a previous example that compares EAX to **val1**, an unsigned doubleword:

```
.data
val1 DWORD   5
result DWORD ?
.code
    mov eax,6
    .IF eax > val1
      mov result,1
    .ENDIF
```

The assembler expands this using the JBE (unsigned jump) instruction:

```
mov eax,6
cmp eax,val1
```

```
    jbe @C0001                  ; jump on unsigned comparison
    mov result,1
@C0001:
```

Comparing a Signed Integer If an .IF directive compares a signed variable, however, a signed conditional jump instruction is inserted into the generated code. For example, **val2**, is a signed doubleword:

```
.data
val2 SDWORD -1
result DWORD ?
.code
    mov eax,6
    .IF eax > val2
      mov result,1
    .ENDIF
```

Consequently, the assembler generates code using the JLE instruction, a jump based on signed comparisons:

```
    mov eax,6
    cmp eax,val2
    jle @C0001                  ; jump on signed comparison
    mov result,1
@C0001:
```

Comparing Registers The question we might then ask is, what happens if two registers are compared? Clearly, the assembler cannot determine whether the values are signed or unsigned:

```
    mov eax,6
    mov ebx,val2
    .IF eax > ebx
       mov result,1
    .ENDIF
```

The following code is generated, showing that the assembler defaults to an unsigned comparison (note the use of the JBE instruction):

```
    mov  eax,6
    mov  ebx,val2
    cmp  eax, ebx
    jbe  @C0001
    mov  result,1
@C0001:
```

6.7.3 Compound Expressions

Many compound boolean expressions use the logical OR and AND operators. When using the .IF directive, the || symbol is the logical OR operator:

```
.IF expression1 || expression2
    statements
.ENDIF
```

Similarly, the && symbol is the logical AND operator:

```
.IF expression1 && expression2
    statements
.ENDIF
```

The logical OR operator will be used in the next program example.

SetCursorPosition Example

The **SetCursorPosition** procedure, shown in the next example, performs range checking on its two input parameters, DH and DL (see *SetCur.asm*). The Y-coordinate (DH) must be between 0 and 24. The X-coordinate (DL) must be between 0 and 79. If either is found to be out of range, an error message is displayed:

```
SetCursorPosition PROC
; Sets the cursor position.
; Receives: DL = X-coordinate, DH = Y-coordinate.
; Checks the ranges of DL and DH.
; Returns: nothing
;-----------------------------------------------
.data
BadXCoordMsg BYTE "X-Coordinate out of range!",0Dh,0Ah,0
BadYCoordMsg BYTE "Y-Coordinate out of range!",0Dh,0Ah,0

.code
    .IF (dl < 0) || (dl > 79)

        mov  edx,OFFSET BadXCoordMsg
        call WriteString
        jmp  quit
    .ENDIF

    .IF (dh < 0) || (dh > 24)

        mov  edx,OFFSET BadYCoordMsg
        call WriteString
        jmp  quit

    .ENDIF

    call Gotoxy
quit:
    ret
SetCursorPosition ENDP
```

The following code is generated by MASM when it preprocesses SetCursorPosition:

```
.code
; .IF (dl < 0) || (dl > 79)

    cmp   dl, 000h
    jb    @C0002
    cmp   dl, 04Fh
    jbe   @C0001
```

```
@C0002:
     mov   edx,OFFSET BadXCoordMsg
     call  WriteString
     jmp   quit

; .ENDIF

@C0001:
; .IF (dh < 0) || (dh > 24)

     cmp   dh, 000h
     jb    @C0005
     cmp   dh, 018h
     jbe   @C0004

@C0005:
     mov   edx,OFFSET BadYCoordMsg
     call  WriteString
     jmp   quit

; .ENDIF

@C0004:
     call  Gotoxy
quit:
     ret
```

College Registration Example

Suppose a college student wants to register for courses. We will use two criteria to determine whether or not the student can register: The first is the person's grade average, based on a 0 to 400 scale, where 400 is the highest possible grade. The second is the number of credits the person wants to take. A multiway branch structure can be used, involving .IF, .ELSEIF, and .ENDIF. The following shows an example (see *Regist.asm*):

```
.data
TRUE = 1
FALSE = 0
gradeAverage  WORD 275          ; test value
credits       WORD 12           ; test value
OkToRegister  BYTE ?
.code
     mov OkToRegister,FALSE
     .IF gradeAverage > 350
        mov OkToRegister,TRUE
     .ELSEIF (gradeAverage > 250) && (credits <= 16)
        mov OkToRegister,TRUE
     .ELSEIF (credits <= 12)
        mov OkToRegister,TRUE
     .ENDIF
```

Table 6-9 lists the corresponding code generated by the assembler, which you can view by looking at the *Dissassembly* window of the Microsoft Visual Studio debugger. (It has been cleaned up here a bit to make it easier to read.) MASM-generated code will appear in the source listing file if you use the /Sg command-line option when assembling programs. The size of a

defined constants (such as TRUE or FALSE in the current code example) is 32-bits. Therefore, when a constant is moved to a BYTE address, MASM inserts the BYTE PTR operator.

Table 6-9 Registration Example, MASM-Generated Code.

```
        mov  byte ptr OkToRegister,FALSE
        cmp  word ptr gradeAverage,350
        jbe  @C0006
        mov  byte ptr OkToRegister,TRUE
        jmp  @C0008
@C0006:
        cmp  word ptr gradeAverage,250
        jbe  @C0009
        cmp  word ptr credits,16
        ja   @C0009
        mov  byte ptr OkToRegister,TRUE
        jmp  @C0008
@C0009:
        cmp  word ptr credits,12
        ja   @C0008
        mov  byte ptr OkToRegister,TRUE
@C0008:
```

6.7.4 Creating Loops with .REPEAT and .WHILE

The .REPEAT and .WHILE directives offer alternatives to writing your own loops with CMP and conditional jump instructions. They permit the conditional expressions listed earlier in Table 6-8. The .REPEAT directive executes the loop body before testing the runtime condition following the .UNTIL directive:

```
.REPEAT
    statements
.UNTIL condition
```

The .WHILE directive tests the condition before executing the loop:

```
.WHILE condition
    statements
.ENDW
```

Examples: The following statements display the values 1 through 10 using the .WHILE directive. The counter register (EAX) is initialized to zero before the loop. Then, in the first statement inside the loop, EAX is incremented. The .WHILE directive branches out of the loop when EAX equals 10.

```
mov eax,0
.WHILE eax < 10
    inc  eax
    call WriteDec
    call Crlf
.ENDW
```

The following statements display the values 1 through 10 using the .REPEAT directive:

```
mov eax,0
.REPEAT
     inc  eax
     call WriteDec
     call Crlf
.UNTIL eax == 10
```

Example: Loop Containing an IF Statement

Earlier in this chapter, in Section 6.5.3, we showed how to write assembly language code for an IF statement nested inside a WHILE loop. Here is the pseudocode:

```
while( op1 < op2 )
{
     op1++;
     if( op1 == op3 )
       X = 2;
     else
       X = 3;
}
```

The following is an implementation of the pseudocode using the .WHILE and .IF directives. Because **op1**, **op2**, and **op3** are variables, they are moved to registers to avoid having two memory operands in any one instruction:

```
.data
X   DWORD 0
op1 DWORD 2                    ; test data
op2 DWORD 4                    ; test data
op3 DWORD 5                    ; test data
.code
     mov eax,op1
     mov ebx,op2
     mov ecx,op3
     .WHILE eax < ebx
       inc eax
       .IF eax == ecx
          mov X,2
       .ELSE
          mov X,3
       .ENDIF
     .ENDW
```

6.8 Chapter Summary

The AND, OR, XOR, NOT, and TEST instructions are called *bitwise instructions* because they work at the bit level. Each bit in a source operand is matched to a bit in the same position of the destination operand:

- The AND instruction produces 1 when both input bits are 1.
- The OR instruction produces 1 when at least one of the input bits is 1.
- The XOR instruction produces 1 only when the input bits are different.

- The TEST instruction performs an implied AND operation on the destination operand, setting the flags appropriately. The destination operand is not changed.
- The NOT instruction reverses all bits in a destination operand.

The CMP instruction compares a destination operand to a source operand. It performs an implied subtraction of the source from the destination and modifies the CPU status flags accordingly. CMP is usually followed by a conditional jump instruction that transfers control to a code label.

Four types of conditional jump instructions are shown in this chapter:

- Table 6-2 contains examples of jumps based on specific flag values, such as JC (jump carry), JZ (jump zero), and JO (jump overflow).
- Table 6-3 contains examples of jumps based on equality, such as JE (jump equal), JNE (jump not equal), and JECXZ (jump if ECX = 0), and JRCXZ (jump if RCX = 0).
- Table 6-4 contains examples of conditional jumps based on comparisons of unsigned integers, such as JA (jump if above), JB (jump if below), and JAE (jump if above or equal).
- Table 6-5 contains examples of jumps based on signed comparisons, such as JL (jump if less) and JG (jump if greater).

In 32-bit mode, the LOOPZ (LOOPE) instruction repeats when the Zero flag is set and ECX is greater than Zero. The LOOPNZ (LOOPNE) instruction repeats when the Zero flag is clear and ECX is greater than zero. In 64-bit mode, the RCX register is used by the LOOPZ and LOOPNZ instructions.

Encryption is a process that encodes data, and *decryption* is a process that decodes data. The XOR instruction can be used to perform simple encryption and decryption.

Flowcharts are an effective tool for visually representing program logic. You can easily write assembly language code, using a flowchart as a model. It is helpful to attach a label to each flowchart symbol and use the same label in your assembly source code.

A *finite-state machine* (FSM) is an effective tool for validating strings containing recognizable characters such as signed integers. It is relatively easy to implement a FSM in assembly language if each state is represented by a label.

The .IF, .ELSE, .ELSEIF, and .ENDIF directives evaluate runtime expressions and greatly simplify assembly language coding. They are particularly useful when coding complex compound boolean expressions. You can also create conditional loops, using the .WHILE and .REPEAT directives.

6.9 Key Terms

6.9.1 Terms

bit-mapped set
bit mask
bit vector
boolean expression
cipher text
compound expression
conditional branching
conditional control flow directives
conditional structure
decryption
directed graph
edge
encryption
finite-state machine (FSM)

initial state
key (encryption)
logical AND operator
logical OR operator
masking (bits)
node
plain text
set complement
set intersection
set union
short-circuit evaluation
symmetric encryption
terminal state
table-driven selection
white box testing

6.9.2 Instructions, Operators, and Directives

AND
.BREAK
CMP
.CONTINUE
.ELSE
.ELSEIF
.ENDIF
.ENDW
.IF
JA
JAE
JB
JBE
JC
JE
JECXZ
JRCXZ
JG
JGE
JL
JLE
JP
JS
JZ
JNA
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNL
JNP
JNS
JNZ
LOOPE
LOOPNE
LOOPZ
LOOPNZ
NOT
OR
.REPEAT
TEST
.UNTIL
.UNTILCXZ
.WHILE
XOR