

CHAPTER 4

Logic & Program Control Instruction:

Topics - Basic Logic Instruction, Shift & Rotate, Jump Group and Procedures, Machine Control & Miscellaneous Instructions, Basic Interrupt Processing, Hardware Interrupts

Text Book Used – The INTEL Microprocessors; Architecture, Programming and Interfacing By Barry B Brey (8thEdition)

BASIC LOGIC INSTRUCTIONS

The basic logic instructions include AND, OR, Exclusive-OR, and NOT. Another logic instruction is TEST, which is explained in this section of the text because the operation of the TEST instruction is a special form of the AND instruction. Also explained is the NEG instruction, which is similar to the NOT instruction.

Logic operations provide binary bit control in low-level software. The logic instructions allow bits to be set, cleared, or complemented. Low-level software appears in machine language or assembly language form and often controls the I/O devices in a system. All logic instructions affect the flag bits. Logic operations always clear the carry and overflow flags, while the other flags change to reflect the condition of the result. When binary data are manipulated in a register or a memory location, the rightmost bit position is always numbered bit 0. Bit position numbers increase from bit 0 toward the left, to bit 7 for a byte, and to bit 15 for a word. A doubleword (32 bits) uses bit position 31 as its leftmost bit and a quadword (64-bits) uses bit position 63 as its leftmost bit.

AND

The AND operation performs logical multiplication, as illustrated by the truth table in Figure 5–3. Here, two bits, A and B, are ANDed to produce the result X. As indicated by the truth table, X is a logic 1 only when both A and B are logic 1s. For all other input combinations of A and B, X is a logic 0. It is important to remember that 0 AND anything is always 0, and 1 AND 1 is always 1.

FIGURE 5–3 (a) The truth table for the AND operation and (b) the logic symbol of an AND gate.

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

(a)



(b)

The AND instruction can replace discrete AND gates if the speed required is not too great, although this is normally reserved for embedded control applications. (Note that Intel has released the 80386EX embedded controller, which embodies the basic structure of the personal computer system.) With the 8086 microprocessor, the AND instruction often executes in about a microsecond. With newer versions, the execution speed is greatly increased. Take the 3.0 GHz Pentium with its clock time of 1/3 ns that executes up to three instruction per clock (1/9 ns per AND operation). If the circuit that the AND instruction replaces operates at a much slower speed than the microprocessor, the AND instruction is a logical replacement. This replacement can save a considerable amount of money. A single AND gate integrated circuit (74HCT08) costs approximately 40¢, while it costs less than 1/100¢ to store the AND instruction in read-only memory. Note that a logic circuit replacement such as this only appears in control systems based on microprocessors and does not generally find application in the personal computer.

The AND operation clears bits of a binary number. The task of clearing a bit in a binary number is called **masking**. Figure 5–4 illustrates the process of masking. Notice that the leftmost 4 bits clear to 0 because 0 AND anything is 0. The bit positions that AND with 1s do not change. This occurs because if a 1 ANDs with a 1, a 1 results; if a 1 ANDs with a 0, a 0 results.

The AND instruction uses any addressing mode except memory-to-memory and segment register addressing. Table 5–16 lists some AND instructions and comments about their operations. An ASCII-coded number can be converted to BCD by using the AND instruction to mask off the leftmost four binary bit positions. This converts the ASCII 30H to 39H to 0–9. Example 5–25 shows a short program that converts the ASCII contents of BX into BCD. The AND instruction in this example converts two digits from ASCII to BCD simultaneously.

EXAMPLE 5–25

```
0000 BB 3135      MOV BX,3135H      ;load ASCII
0003 81 E3 0F0F    AND BX,0F0FH      ;mask BX
```

OR

The **OR operation** performs logical addition and is often called the *Inclusive-OR* function. The OR function generates a logic 1 output if any inputs are 1. A 0 appears at the output only when all inputs are 0. The truth table for the OR function appears in Figure 5–5. Here, the inputs A and B OR together to produce the X output. It is important to remember that 1 ORed with anything yields a 1.

FIGURE 5–4 The operation of the AND function showing how bits of a number are cleared to zero.

x x x x x x x x	Unknown number
• 0 0 0 0 1 1 1 1	Mask
0 0 0 0 x x x x	Result

TABLE 5–16 Example AND instructions.

Assembly Language	Operation
AND AL,BL	AL = AL and BL
AND CX,DX	CX = CX and DX
AND ECX,EDI	ECX = ECX and EDI
AND RDX,RBP	RDX = RDX and RBP (64-bit mode)
AND CL,33H	CL = CL and 33H
AND DI,4FFFH	DI = DI and 4FFFH
AND ESI,34H	ESI = ESI and 34H
AND RAX,1	RAX = RAX and 1 (64-bit mode)
AND AX,[DI]	The word contents of the data segment memory location addressed by DI are ANDed with AX
AND ARRAY[SI],AL	The byte contents of the data segment memory location addressed by ARRAY plus SI are ANDed with AL
AND [EAX],CL	CL is ANDed with the byte contents of the data segment memory location addressed by ECX

FIGURE 5–5 (a) The truth table for the OR operation and (b) the logic symbol of an OR gate.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

(a)



(b)

In embedded controller applications, the OR instruction can also replace discrete OR gates. This results in considerable savings because a quad, two-input OR gate (74HCT32) costs about 40¢, while the OR instruction costs less than 1/100¢ to store in a read-only memory. Figure 5–6 shows how the OR gate sets (1) any bit of a binary number. Here, an unknown number (XXXX XXXX) ORs with a 0000 1111 to produce a result of XXXX 1111. The rightmost 4 bits set, while the leftmost 4 bits remain unchanged. The OR operation sets any bit; the AND operation clears any bit.

The OR instruction uses any of the addressing modes allowed to any other instruction except segment register addressing. Table 5–17 illustrates several example OR instructions with comments about their operation.

Suppose that two BCD numbers are multiplied and adjusted with the AAM instruction. The result appears in AX as a two-digit unpacked BCD number. Example 5–26 illustrates this multiplication and shows how to change the result into a two-digit ASCII-coded number using the OR instruction. Here, OR AX,3030H converts the 0305H found in AX to 3335H. The OR operation can be replaced with an ADD AX,3030H to obtain the same results.

FIGURE 5–6 The operation of the OR function showing how bits of a number are set to one.

x x x x	x x x x	Unknown number
+	0 0 0 0 1 1 1 1	Mask
<hr/>		
x x x x	1 1 1 1	Result

TABLE 5–17 Example OR instructions.

Assembly Language	Operation
OR AH,BL	AL = AL or BL
OR SI,DX	SI = SI or DX
OR EAX,EBX	EAX = EAX or EBX
OR R9,R10	R9 = R9 or R10 (64-bit mode)
OR DH,0A3H	DH = DH or 0A3H
OR SP,990DH	SP = SP or 990DH
OR EBP,10	EBP = EBP or 10
OR RBP,1000H	RBP = RBP or 1000H (64-bit mode)
OR DX,[BX]	DX is ORed with the word contents of data segment memory location addressed by BX
OR DATES[DI + 2],AL	The byte contents of the data segment memory location addressed by DI plus 2 are ORed with AL

EXAMPLE 5–26

0000 B0 05	MOV	AL,5	;load data
0002 B3 07	MOV	BL,7	
0004 F6 E3	MUL	BL	
0006 D4 0A	AAM		;adjust
0008 0D 3030	OR	AX,3030H	;convert to ASCII

Exclusive-OR

The **Exclusive-OR** instruction (XOR) differs from Inclusive-OR (OR). The difference is that a 1,1 condition of the OR function produces a 1; the 1,1 condition of the Exclusive-OR operation produces a 0. The Exclusive-OR operation excludes this condition; the Inclusive-OR includes it.


Figure 5–7 shows the truth table of the Exclusive-OR function. (Compare this with Figure 5–5 to appreciate the difference between these two OR functions.) If the inputs of the Exclusive-OR function are both 0 or both 1, the output is 0. If the inputs are different, the output is 1. Because of this, the Exclusive-OR is sometimes called a comparator. The XOR instruction uses any addressing mode except segment register addressing. Table 5–18 lists several Exclusive-OR instructions and their operations.

As with the AND and OR functions, Exclusive-OR can replace discrete logic circuitry in embedded applications. The 74HCT86 quad, two-input Exclusive-OR gate is replaced by one XOR instruction. The 74HCT86 costs about 40¢, whereas the instruction costs less than 1/100¢ to store in the memory. Replacing just one 74HCT86 saves a considerable amount of money, especially if many systems are built.

FIGURE 5-7 (a) The truth table for the Exclusive-OR operation and (b) the logic symbol of an Exclusive-OR gate.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)

TABLE 5-18 Example Exclusive-OR instructions.

Assembly Language	Operation
XOR CH,DL	CH = CH xor DL
XOR SI,BX	SI = SI xor BX
XOR EBX,EDI	EBX = EBX xor EDI
XOR RAX,RBX	RAX = RAX xor RBX (64-bit mode)
XOR AH,0EEH	AH = AH xor 0EEH
XOR DI,00DDH	DI = DI xor 00DDH
XOR ESI,100	ESI = ESI xor 100
XOR R12,20	R12 = R12 xor 20 (64-bit mode)
XOR DX,[SI]	DX is Exclusive-ORed with the word contents of the data segment memory location addressed by SI
XOR DEAL[BP+2],AH	AH is Exclusive-ORed with the byte contents of the stack segment memory location addressed by BP plus 2

The Exclusive-OR instruction is useful if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented. Figure 5-8 shows how just part of an unknown quantity can be inverted by XOR. Notice that when a 1 Exclusive-ORs with X, the result is X. If a 0 Exclusive-ORs with X, the result is X. Suppose that the leftmost 10 bits of the BX register must be inverted without changing the rightmost 6 bits. The XOR BX,0FFC0H instruction accomplishes this task. The AND instruction clears (0) bits, the OR instruction sets (1) bits, and now the Exclusive-OR instruction inverts bits. These three instructions allow a program to gain complete control over any bit stored in any register or memory location. This is ideal for control system applications in which equipment must be turned on (1), turned off (0), and toggled from on to off or off to on.

A common use for the Exclusive-OR instruction is to clear a register to zero. For example, the XOR CH,CH instruction clears register CH to 00H and requires 2 bytes of memory to store the instruction. Likewise, the MOV CH, 00H instruction also clears CH to 00H, but requires 3 bytes of memory. Because of this saving, the XOR instruction is often used to clear a register in place of a move immediate.

Example 5-27 shows a short sequence of instructions that clears bits 0 and 1 of CX, sets bits 9 and 10 of CX, and inverts bit 12 of CX. The OR instruction is used to set bits, the AND instruction is used to clear bits, and the XOR instruction inverts bits.

FIGURE 5-8 The operation of the Exclusive-OR function showing how bits of a number are inverted.

x x x x x x x x	Unknown number
⊕ 0 0 0 0 1 1 1 1	Mask
x x x x x x x x	Result

EXAMPLE 5-27

0000 81 C9 0600	OR CX,0600H	;set bits 9 and 10
0004 83 E1 FC	AND CX,0FFC0H	;clear bits 0 and 1
0007 81 F1 1000	XOR CX,1000H	;invert bit 12

Test and Bit Test Instructions

The **TEST instruction** performs the AND operation. The difference is that the AND instruction changes the destination operand, whereas the TEST instruction does not. A TEST only affects the condition of the flag register, which indicates the result of the test. The TEST instruction uses the same addressing modes as the AND instruction. Table 5-19 lists some TEST instructions and their operations.

The TEST instruction functions in the same manner as a CMP instruction. The difference is that the TEST instruction normally tests a single bit (or occasionally multiple bits), whereas the CMP instruction tests the entire byte, word, or doubleword. The zero flag (Z) is a logic 1 (indicating a zero result) if the bit under test is a zero, and Z=0 (indicating a nonzero result) if the bit under test is not zero.

Usually the TEST instruction is followed by either the JZ (jump if zero) or JNZ (jump if not zero) instruction. The destination operand is normally tested against immediate data. The value of immediate data is 1 to test the rightmost bit position, 2 to test the next bit, 4 for the next, and so on.

Example 5–28 lists a short program that tests the rightmost and leftmost bit positions of the AL register. Here, 1 selects the rightmost bit and 128 selects the leftmost bit. (Note: A 128 is an 80H.) The JNZ instruction follows each test to jump to different memory locations, depending on the outcome of the tests. The JNZ instruction jumps to the operand address (RIGHT or LEFT in the example) if the bit under test is not zero.

The 80386 through the Pentium 4 processors contain additional test instructions that test single bit positions. Table 5–20 lists the four different bit test instructions available to these microprocessors.

All four forms of the bit test instruction test the bit position in the destination operand selected by the source operand. For example, the BT AX,4 instruction tests bit position 4 in AX. The result of the test is located in the carry flag bit. If bit position 4 is a 1, carry is set; if bit position 4 is a 0, carry is cleared.

The remaining 3-bit test instructions also place the bit under test into the carry flag and change the bit under test afterward. The BTC AX,4 instruction complements bit position 4 after testing it, the BTR AX,4 instruction clears it (0) after the test, and the BTS AX,4 instruction sets it (1) after the test.

Example 5–29 repeats the sequence of instructions listed in Example 5–27. Here, the BTR instruction clears bits in CX, BTS sets bits in CX, and BTC inverts bits in CX.

EXAMPLE 5–28

```
0000 A8 01      TEST AL,1          ;test right bit
0002 75 1C      JNZ RIGHT      ;if set
0004 A8 80      TEST AL,128     ;test left bit
0006 75 38      JNZ LEFT       ;if set
```

TABLE 5–19 Example TEST instructions.

Assembly Language	Operation
TEST DL,DH	DL is ANDed with DH
TEST CX,BX	CX is ANDed with BX
TEST EDX,ECX	EDX is ANDed with ECX
TEST RDX,R15	RDX is ANDed with R15 (64-bit mode)
TEST AH,4	AH is ANDed with 4
TEST EAX,256	EAX is ANDed with 256

TABLE 5–20 Bit test instructions.

Assembly Language	Operation
BT	Tests a bit in the destination operand specified by the source operand
BTC	Tests and complements a bit in the destination operand specified by the source operand
BTR	Tests and resets a bit in the destination operand specified by the source operand
BTS	Tests and sets a bit in the destination operand specified by the source operand

EXAMPLE 5–29

```
0000 0F BA E9 09      BTS CX,9          ;set bit 9
0004 0F BA E9 0A      BTS CX,10         ;set bit 10
0008 0F BA F1 00      BTR CX,0          ;clear bit 0
000C 0F BA F1 01      BTR CX,1          ;clear bit 1
0010 0F BA F9 0C      BTC CX,12         ;complement bit 12
```

NOT and NEG

Logical inversion, or the one's complement (NOT), and arithmetic sign inversion, or the two's complement (NEG), are the last two logic functions presented (except for shift and rotate in the next section of the text). These are two of a few instructions that contain only one operand.

Table 5–21 lists some variations of the NOT and NEG instructions. As with most other instructions, NOT and NEG can use any addressing mode except segment register addressing.

The NOT instruction inverts all bits of a byte, word, or doubleword. The NEG instruction two's complements a number, which means that the arithmetic sign of a signed number changes from positive to negative or from negative to positive. The NOT function is considered logical, and the NEG function is considered an arithmetic operation.

TABLE 5–21 Example NOT and NEG instructions.

<i>Assembly Language</i>	<i>Operation</i>
NOT CH	CH is one's complemented
NEG CH	CH is two's complemented
NEG AX	AX is two's complemented
NOT EBX	EBX is one's complemented
NEG ECX	ECX is two's complemented
NOT RAX	RAX is one's complemented (64-bit mode)
NOT TEMP	The contents of data segment memory location TEMP is one's complemented
NOT BYTE PTR[BX]	The byte contents of the data segment memory location addressed by BX are one's complemented

SHIFT AND ROTATE

Shift and rotate instructions manipulate binary numbers at the binary bit level, as did the AND, OR, Exclusive-OR, and NOT instructions. Shifts and rotates find their most common applications in low-level software used to control I/O devices. The microprocessor contains a complete complement of shift and rotate instructions that are used to shift or rotate any memory data or register.

Shift

Shift instructions position or move numbers to the left or right within a register or memory location. They also perform simple arithmetic such as multiplication by powers of 2^{+n} (left shift) and division by powers of 2^{-n} (right shift). The microprocessor's instruction set contains four different shift instructions: Two are logical shifts and two are arithmetic shifts. All four shift operations appear in Figure 5–9.

Notice in Figure 5–9 that there are two right shifts and two left shifts. The logical shifts move a 0 into the rightmost bit position for a logical left shift and a 0 into the leftmost bit position for a logical right shift. There are also two arithmetic shifts. The arithmetic shift left and logical left shift are identical. The arithmetic right shift and logical right shift are different because the arithmetic right shift copies the sign-bit through the number, whereas the logical right shift copies a 0 through the number.

Logical shift operations function with unsigned numbers, and arithmetic shifts function with signed numbers. Logical shifts multiply or divide unsigned data, and arithmetic shifts multiply or divide signed data. A shift left always multiplies by 2 for each bit position shifted, and a shift right always divides by 2 for each bit position shifted. Shifting a number two places, to the left or right, multiplies or divides by 4.

Table 5–22 illustrates some addressing modes allowed for the various shift instructions. There are two different forms of shifts that allow any register (except the segment register) or memory location to be shifted. One mode uses an immediate shift count, and the other uses register CL to hold the shift count. Note that CL must hold the shift count. When CL is the shift count, it does not change when the shift instruction executes. Note that the shift count is a modulo-32 count, which means that a shift count of 33 will shift the data one place ($33/32 = \text{remainder of } 1$). The same applies to a 64-bit number, but the shift count is modulo-64.

FIGURE 5-9 The shift instructions showing the operation and direction of the shift.

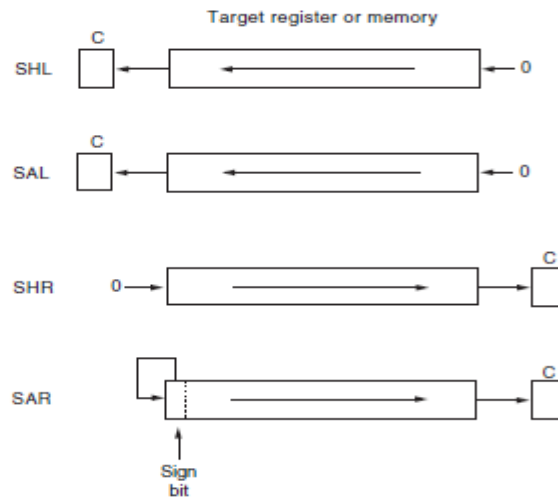


TABLE 5-22 Example shift instructions.

Assembly Language	Operation
SHL AX,1	AX is logically shifted left 1 place
SHR BX,12	BX is logically shifted right 12 places
SHR ECX,10	ECX is logically shifted right 10 places
SHL RAX,50	RAX is logically shifted left 50 places (64-bit mode)
SAL DATA1,CL	The contents of data segment memory location DATA1 are arithmetically shifted left the number of spaces specified by CL
SHR RAX,CL	RAX is logically shifted right the number of spaces specified by CL (64-bit mode)
SAR SI,2	SI is arithmetically shifted right 2 places
SAR EDX,14	EDX is arithmetically shifted right 14 places

Example 5-30 shows how to shift the DX register left 14 places in two different ways. The first method uses an immediate shift count of 14. The second method loads 14 into CL and then uses CL as the shift count. Both instructions shift the contents of the DX register logically to the left 14 binary bit positions or places. Suppose that the contents of AX must be multiplied by 10, as shown in Example 5-31. This can be done in two ways: by the MUL instruction or by shifts and additions. A number is doubled when it shifts left one place. When a number is doubled, and then added to the number times 8, the result is 10 times the number. The number 10 decimal is 1010 in binary. A logic 1 appears in both the 2's and 8's positions. If 2 times the number is added to 8 times the number, the result is 10 times the number. Using this technique, a program can be written to multiply by any constant. This technique often executes faster than the multiply instruction found in earlier versions of the Intel microprocessor.

EXAMPLE 5-30

```

0000 C1 E2 0E          SHL DX,14
                        OR
0003 B1 0E             MOV CL,14
0005 D3 E2             SHL DX,CL

```

EXAMPLE 5-31

```

;Multiply AX by 10 (1010)
;
0000 D1 E0             SHL AX,1           ;AX times 2
0002 8B D8             MOV BX,AX
0004 C1 E0 02          SHL AX,2           ;AX times 8
0007 03 C3             ADD AX,BX          ;AX times 10

;Multiply AX by 18 (10010)
;
0009 D1 E0             SHL AX,1           ;AX times 2
000B 8B D8             MOV BX,AX
000D C1 E0 03          SHL AX,3           ;AX times 16
0010 03 C3             ADD AX,BX          ;AX times 18

;Multiply AX by 5 (101)
;
0012 8B D8             MOV BX,AX
0014 C1 E0 02          SHL AX,2           ;AX times 4
0017 03 C3             ADD AX,BX          ;AX times 5

```

Double-Precision Shifts (80386–Core2 Only). The 80386 and above contain two double precision shifts: SHLD (shift left) and SHRD (shift right). Each instruction contains three operands, instead of the two found with the other shift instructions. Both instructions function with two 16-or 32-bit registers, or with one 16-or 32-bit memory location and a register.

The SHRD AX,BX,12 instruction is an example of the double-precision shift right instruction. This instruction logically shifts AX right by 12 bit positions. The rightmost 12 bits of BX shift into the leftmost 12 bits of AX. The contents of BX remain unchanged by this instruction. The shift count can be an immediate count, as in this example, or it can be found in register CL, as with other shift instructions.

The SHLD EBX,ECX,16 instruction shifts EBX left. The leftmost 16 bits of ECX fill the rightmost 16 bits of EBX after the shift. As before, the contents of ECX, the second operand, remain unchanged. This instruction, as well as SHRD, affects the flag bits.

Rotate

Rotate instructions position binary data by rotating the information in a register or memory location, either from one end to another or through the carry flag. They are often used to shift or position numbers that are wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386 through the Core2. The four available rotate instructions appear in Figure 5–10. Numbers rotate through a register or memory location, through the C flag (carry), or through a register or memory location only. With either type of rotate instruction, the programmer can select either a left or a right rotate. Addressing modes used with rotate are the same as those used with shifts. A rotate count can be immediate or located in register CL. Table 5–23 lists some of the possible rotate instructions. If CL is used for a rotate count, it does not change. As with shifts, the count in CL is a modulo-32 count for a 32-bit operation and modulo-64 for a 64-bit operation. Rotate instructions are often used to shift wide numbers to the left or right. The program listed in Example 5–32 shifts the 48-bit number in registers DX, BX, and AX left one binary place. Notice that the least significant 16 bits (AX) shift left first. This moves the leftmost bit of AX into the carry flag bit. Next, the rotate BX instruction rotates carry into BX, and its leftmost bit moves into carry. The last instruction rotates carry into DX, and the shift is complete.

EXAMPLE 5–32

```
0000 D1 E0    SHL AX,1
0002 D1 D3    RCL BX,1
0004 D1 D2    RCL DX,1
```

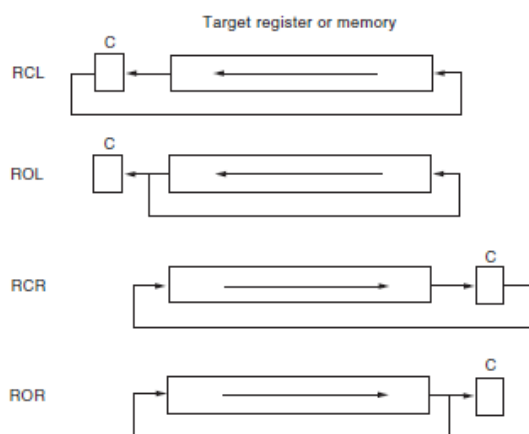


TABLE 5–23 Example rotate instructions.

Assembly Language	Operation
ROL SI,14	SI rotates left 14 places
RCL BL,6	BL rotates left through carry 6 places
ROL ECX,18	ECX rotates left 18 places
ROL RDX,40	RDX rotates left 40 places
RCR AH,CL	AH rotates right through carry the number of places specified by CL
ROR WORD PTR[BP],2	The word contents of the stack segment memory location addressed by BP rotate right 2 places

FIGURE 5–10 The rotate instructions showing the direction and operation of each rotate.

Bit Scan Instructions

Although the bit scan instructions don't shift or rotate numbers, they do scan through a number searching for a 1-bit. Because this is accomplished within the microprocessor by shifting the number, bit scan instructions are included in this section of the text.

The bit scan instructions BSF (bit scan forward) and BSR (bit scan reverse) are available only in the 80386–Pentium 4 processors. Both forms scan through the source number, searching for the first 1-bit. The BSF instruction scans the number from the leftmost bit toward the right, and BSR scans the number from the rightmost bit toward the left. If a 1-bit is encountered, the zero flag is set and the bit position number of the

1-bit is placed into the destination operand. If no 1-bit is encountered (i.e., the number contains all zeros), the zero flag is cleared. Thus, the result is not-zero if no 1-bit is encountered.

For example, if EAX = 60000000H and the BSF EBX,EAX instruction executes, the number is scanned from the leftmost bit toward the right. The first 1-bit encountered is at bit position 30, which is placed into EBX and the zero flag bit is set. If the same value for EAX is used for the BSR instruction, the EBX register is loaded with 29 and the zero flag bit is set.

THE JUMP GROUP

The main program control instruction, **jump** (JMP), allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction. A conditional jump instruction allows the programmer to make decisions based upon numerical tests. The results of numerical tests are held in the flag bits, which are then tested by conditional jump instructions. Another instruction similar to the conditional jump, the conditional set, is explained with the conditional jump instructions in this section.

In this section of the text, all jump instructions are illustrated with their uses in sample programs. Also revisited are the LOOP and conditional LOOP instructions, first presented in Chapter 3, because they are also forms of the jump instruction.

Unconditional Jump (JMP)

Three types of unconditional jump instructions (see Figure 6–1) are available to the microprocessor: short jump, near jump, and far jump. The **short jump** is a 2-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes from the address following the jump. The 3-byte **near jump** allows a branch or jump within $\pm 32\text{K}$ bytes (or anywhere in the current code segment) from the instruction in the current code segment. Remember that segments are cyclic in nature, which means that one location above offset address FFFFH is offset address 0000H. For this reason, if you jump 2 bytes ahead in memory and the instruction pointer addresses offset address FFFFH, the flow continues at offset address 0001H. Thus, a displacement of $\pm 32\text{K}$ bytes allows a jump to any location within the current code segment. Finally, the 5-byte **far jump** allows a jump to any memory location within the real memory system. The short and near jumps are often called **intra-segment jumps**, and the far jumps are often called **inter-segment jumps**.

In the 80386 through the Core2 processors, the near jump is within $\pm 2\text{G}$ if the machine is operated in the protected mode, with a code segment that is 4G bytes long. If operated in the real mode, the near jump is within $\pm 32\text{K}$ bytes. In the protected mode, the 80386 and above use a 32-bit displacement that is not shown in Figure 6–1. If the Pentium 4 is operated in the 64-bit mode, a jump can be to any address in its 1T memory space.

Short Jump. Short jumps are called **relative jumps** because they can be moved, along with their related software, to any location in the current code segment without a change. This is because the jump address is not stored with the opcode. Instead of a jump address, a **distance**, or displacement, follows the opcode. The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and -128. The short jump instruction appears in Figure 6–2. When the microprocessor executes a short jump, the displacement is signextended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment. The short jump instruction branches to this new address for the next instruction in the program.

Example 6–1 shows how short jump instructions pass control from one part of the program to another. It also illustrates the use of a **label** (a symbolic name for a memory address) with the jump instruction. Notice how one jump (JMP SHORT NEXT) uses the SHORT directive to force a short jump, while the other does not. Most assembler programs choose the best form of the jump instruction so the second jump instruction (JMP START) also assembles as a short jump. If the address of the next instruction (0009H) is added to the sign-extended displacement (0017H) of the first jump, the address of NEXT is at location 0017H + 0009H or 0020H.

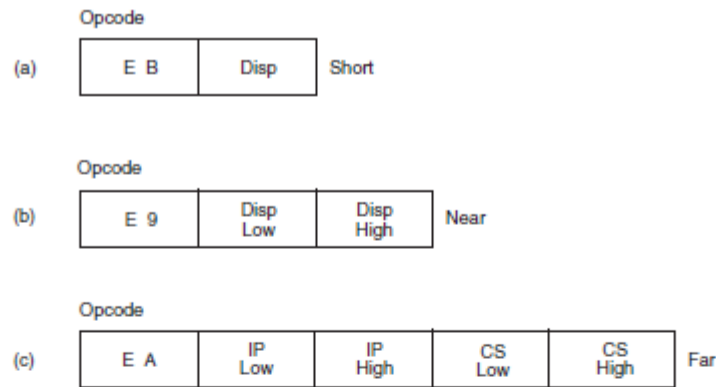


FIGURE 6-1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.

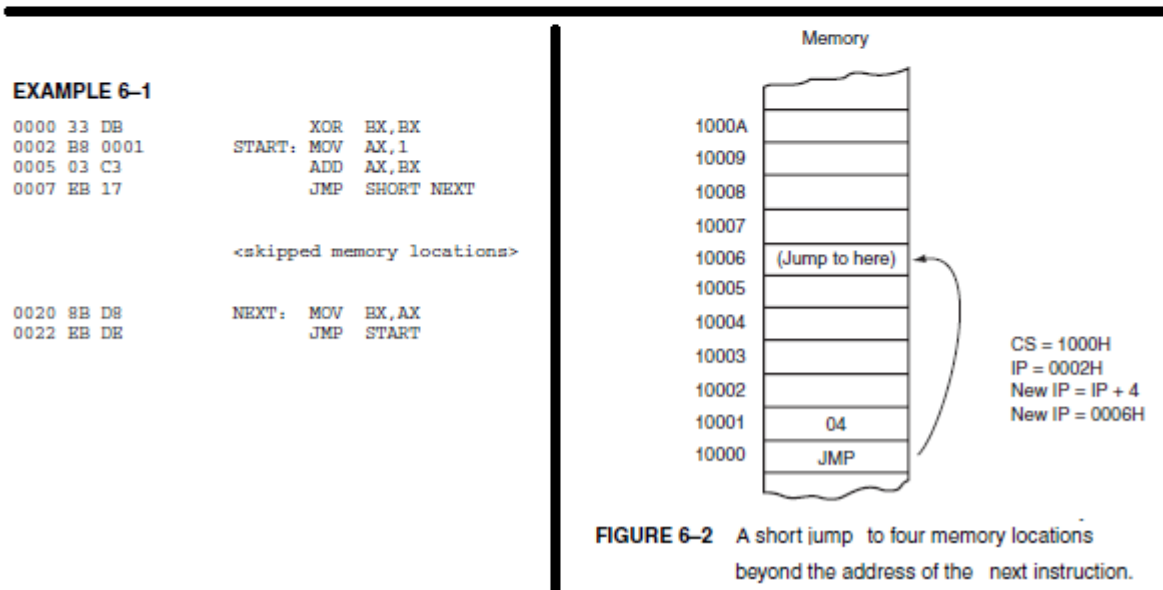


FIGURE 6-2 A short jump to four memory locations beyond the address of the next instruction.

Whenever a jump instruction references an address, a label normally identifies the address. The JMP NEXT instruction is an example; it jumps to label NEXT for the next instruction. It is very rare to use an actual hexadecimal address with any jump instruction, but the assembler supports addressing in relation to the instruction pointer by using the \$+a displacement. For example, the JMP \$+2 instruction jumps over the next two memory locations (bytes) following the JMP instruction. The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it for a jump. If a colon does not follow a label, you cannot jump to it. Note that the only time a colon is used after a label is when the label is used with a jump or call instruction. This is also true in Visual C++.

Near Jump. The near jump is similar to the short jump, except that the distance is farther. A near jump passes control to an instruction in the current code segment located within $\pm 32\text{K}$ bytes from the near jump instruction. The distance is $\pm 2\text{G}$ in the 80386 and above when operated in protected mode. The near jump is a 3-byte instruction that contains an opcode followed by a signed 16-bit displacement. In the 80386 through the Pentium 4 processors, the displacement is 32 bits and the near jump is 5 bytes long. The signed displacement adds to the instruction pointer (IP) to generate the jump address. Because the signed displacement is in the range of $\pm 32\text{K}$, a near jump can jump to any memory location within the current real mode code segment. The protected mode code segment in the 80386 and above can be 4G bytes long, so the 32-bit displacement allows a near jump to any location within $\pm 2\text{G}$ bytes. Figure 6-3 illustrates the operation of the real mode near jump instruction.

The near jump is also relocatable (as was the short jump) because it is also a relative jump. If the code segment moves to a new location in the memory, the distance between the jump instruction and the operand address remains the same. This allows a code segment to be relocated by simply moving it. This

feature, along with the relocatable data segments, makes the Intel family of microprocessors ideal for use in a general-purpose computer system. Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.

Example 6–2 shows the same basic program that appeared in Example 6–1, except that the jump distance is greater. The first jump (JMP NEXT) passes control to the instruction at offset memory location 0200H within the code segment. Notice that the instruction assembles as E9 0200 R. The letter R denotes a **relocatable jump address** of 0200H. The relocatable address of 0200H is for the assembler program’s internal use only. The actual machine language instruction assembles as E9 F6 01, which does not appear in the assembler listing. The actual displacement is 01F6H for this jump instruction. The assembler lists the jump address as 0200 R, so the address is easier to interpret as software is developed. If the linked execution file (.EXE) or command file (.COM) is displayed in hexadecimal code, the jump instruction appears as E9 F6 01.

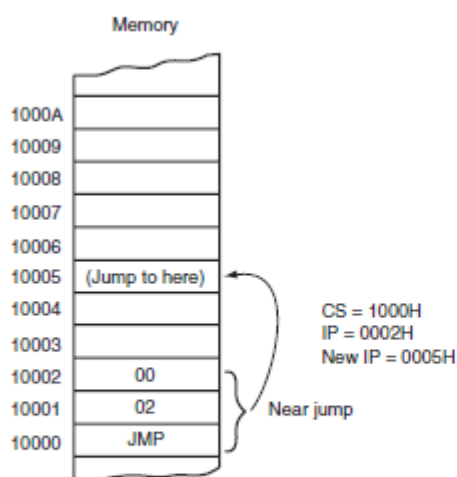


FIGURE 6-3 A near jump that adds the displacement (0002H) to the contents of IP.

EXAMPLE 6-2

```

0000 33 DB          XOR  BX,BX
0002 B8 0001      START: MOV  AX,1
0005 03 C3        ADD  AX,BX
0007 E9 0200 R    JMP  NEXT

<skipped memory locations>

0200 8B DB          NEXT: MOV  BX,AX
0202 E9 0002 R    JMP  START

```

Far Jump. A far jump instruction (see Figure 6–4) obtains a new segment and offset address to accomplish the jump. Bytes 2 and 3 of this 5-byte instruction contain the new offset address; bytes 4 and 5 contain the new segment address. If the microprocessor (80286 through the Core2) is operated in the protected mode, the segment address accesses a descriptor that contains the base address of the far jump segment. The offset address, which is either 16 or 32 bits, contains the offset address within the new code segment.

Example 6–3 lists a short program that uses a far jump instruction. The far jump instruction sometimes appears with the FAR PTR directive, as illustrated. Another way to obtain a far jump is to define a label as a **far label**. A label is far only if it is external to the current code segment or procedure. The JMP UP instruction in the example references a far label. The label UP is defined as a far label by the EXTRN UP:FAR directive. **External labels** appear in programs that contain more than one program file. Another way of defining a label as global is to use a *double colon* (LABEL::) following the label in place of the single colon. This is required inside procedure blocks that are defined as near if the label is accessed from outside the procedure block.

When the program files are joined, the linker inserts the address for the UP label into the JMP UP instruction. It also inserts the segment address in the JMP START instruction. The segment address in JMP FAR PTR START is listed as - - - R for relocatable; the segment address in JMP UP is listed as - - - E for external. In both cases, the - - - is filled in by the linker when it links or joins the program files.

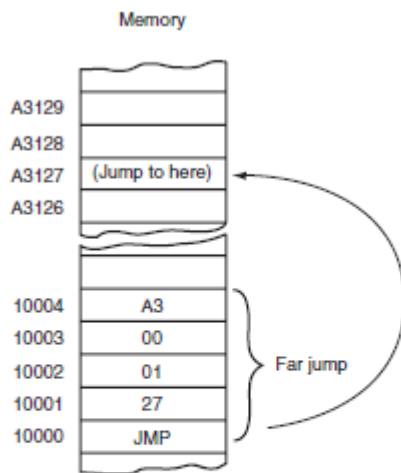


FIGURE 6-4 A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.

EXAMPLE 6-3

```

                                EXTRN  UP:FAR
0000 33 DB                      XOR BX,BX
0002 B8 0001  START:  ADD AX,1
0005 E9 0200 R                JMP NEXT

                                ;<skipped memory locations>

0200 8B D8                      NEXT:  MOV BX,AX
0202 EA 0002 — R              JMP FAR PTR START
0207 EA 0000 — R              JMP UP

```

Jumps with Register Operands. The jump instruction can also use a 16- or 32-bit register as an operand. This automatically sets up the instruction as an **indirect jump**. The address of the jump is in the register specified by the jump instruction. Unlike the displacement associated with the near jump, the contents of the register are transferred directly into the instruction pointer. An indirect jump does not add to the instruction pointer, as with short and near jumps. The JMP AX instruction, for example, copies the contents of the AX register into the IP when the jump occurs. This allows a jump to any location within the current code segment. In the 80386 and above, a JMP EAX instruction also jumps to any location within the current code segment; the difference is that in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed.

EXAMPLE 6-4

```

;Instructions that read 1, 2, or 3 from the keyboard.
;The number is displayed as 1, 2, or 3 using a jump table
;
.MODEL SMALL                      ;select SMALL model
.DATA                             ;start data segment
TABLE: DW ONE                     ;jump table
       DW TWO
       DW THREE
.CODE                             ;start code segment
.STARTUP                          ;start program
TOP:   MOV AH,1                   ;read key into AL
       INT 21H
       SUB AL,31                  ;convert to BCD
       JB TOP                     ;if key < 1
       CMP AL,2
       JA TOP                     ;if key > 3
       MOV AH,0                  ;double key code
       ADD AX,AX
       MOV SI,OFFSET TABLE      ;address TABLE
       ADD SI,AX                  ;form lookup address
       MOV AX,[SI]               ;get ONE, TWO or THREE
       JMP AX                    ;jump to ONE, TWO or THREE
ONE:   MOV DL,'1'                 ;get ASCII 1
       JMP BOT
TWO:   MOV DL,'2'                 ;get ASCII 2
       JMP BOT
THREE: MOV DL,'3'                 ;get ASCII 3
BOT:   MOV AH,2                  ;display number
       INT 21H
.EXIT
END

```

Example 6–4 shows how the JMP AX instruction accesses a jump table in the code segment. This DOS program reads a key from the keyboard and then modifies the ASCII code to 00H in AL for a '1', 01H for a '2', and 02H for a '3'. If a '1', '2', or '3' is typed, AH is cleared to 00H. Because the jump table contains 16-bit offset addresses, the contents of AX are doubled to 0, 2, or 4, so a 16-bit entry in the table can be accessed. Next, the offset address of the start of the jump table is loaded to SI, and AX is added to form the reference to the jump address. The MOV AX,[SI] instruction then fetches an address from the jump table, so the JMP AX instruction jumps to the addresses (ONE, TWO, or THREE) stored in the jump table.

Indirect Jumps Using an Index. The jump instruction may also use the [] form of addressing to directly access the jump table. The jump table can contain offset addresses for near indirect jumps, or segment and offset addresses for far indirect jumps. (This type of jump is also known as a *double-indirect jump* if the register jump is called an *indirect jump*.) The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction. Here Example 6–5 repeats Example 6–4 by using the JMP TABLE [SI] instead of JMP AX. This reduces the length of the program.

The mechanism used to access the jump table is identical with a normal memory reference. The JMP TABLE [SI] instruction points to a jump address stored at the code segment offset location addressed by SI. It jumps to the address stored in the memory at this location. Both the register and indirect indexed jump instructions usually address a 16-bit offset. This means that both types of jumps are near jumps. If a JMP FAR PTR [SI] or JMP TABLE [SI], with TABLE data defined with the DD directive appears in a program, the microprocessor assumes that the jump table contains doubleword, 32-bit addresses (IP and CS).

EXAMPLE 6-5

```

;Instructions that read 1, 2, or 3 from the keyboard.
;The number is displayed as 1, 2, or 3 using a jump table
;
.MODEL SMALL                ;select SMALL model
.DATA                      ;start data segment
TABLE: DW ONE               ;jump table
      DW TWO
      DW THREE

.CODE                      ;start code segment
.STARTUP                  ;start program
TOP:  MOV AH,1              ;read key into AL
      INT 21H
      SUB AL,31             ;convert to BCD
      JB  TOP               ;if key < 1
      CMP AL,2
      JA  TOP               ;if key > 3
      MOV AH,0              ;double key code
      ADD AX,AX
      MOV SI,AX             ;form lookup address
      JMP TABLE[SI]        ;jump to ONE, TWO or THREE
ONE:  MOV DL,'1'             ;get ASCII 1
      JMP BOT
TWO:  MOV DL,'2'             ;get ASCII 2
      JMP BOT
THREE: MOV DL,'3'            ;get ASCII 3
BOT:  MOV AH,2              ;display number
      INT 21H
      .EXIT
      END
0000
0000 002D R
0002 0031 R
0004 0035 R
0000
0017 B4 01
0019 CD 21
001B 2C 31
001D 72 F9
001F 32 02
0021 77 F4
0023 B4 00
0025 03 C0
0027 B5 F0
0029 FF A4 0000 R
002D B2 31
002F EB 06
0031 B2 32
0033 EB 02
0035 B2 33
0037 B4 02
0039 CD 21

```

Conditional Jumps and Conditional Sets

Conditional jump instructions are always short jumps in the 8086 through the 80286 microprocessors. This limits the range of the jump to within +127 bytes and -128 bytes from the location following the conditional jump. In the 80386 and above, conditional jumps are either short or near jumps ($\pm 32\text{K}$). In the 64-bit mode of the Pentium 4, the near jump distance is $\pm 2\text{G}$ for the conditional jumps. This allows these microprocessors to use a conditional jump to any location within the current code segment. Table 6–1 lists all the conditional jump instructions with their test conditions. Note that the Microsoft MASM version 6.x assembler automatically adjusts conditional jumps if the distance is too great.

The conditional jump instructions test the following flag bits: sign (S), zero (Z), carry (C), parity (P), and overflow (O). If the condition under test is true, a branch to the label associated with the jump instruction

occurs. If the condition is false, the next sequential step in the program executes. For example, a JC will jump if the carry bit is set.

The operation of most conditional jump instructions is straightforward because they often test just one flag bit, although some test more than one. Relative magnitude comparisons require more complicated conditional jump instructions that test more than one flag bit.

TABLE 6-1 Conditional jump instructions.

<i>Assembly Language</i>	<i>Tested Condition</i>	<i>Operation</i>
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = 0	Jump if greater than
JGE	S = 0	Jump if greater than or equal
JL	S != 0	Jump if less than
JLE	Z = 1 or S != 0	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign (positive)
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow
JP or JPE	P = 1	Jump if parity or jump if parity even
JS	S = 1	Jump if sign (negative)
JCXZ	CX = 0	Jump if CX is zero
JECXZ	ECX = 0	Jump if ECX equals zero
JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

Because both signed and unsigned numbers are used in programming, and because the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons. Figure 6-5 shows the order of both signed and unsigned 8-bit numbers. The 16- and 32-bit numbers follow the same order as the 8-bit numbers, except that they are larger. Notice that an FFH (255) is above the 00H in the set of unsigned numbers, but an FFH (-1) is less than 00H for signed numbers. Therefore, an unsigned FFH is above 00H, but a signed FFH is less than 00H.

When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions. The terms *greater than* and *less than* refer to signed numbers. When unsigned numbers are compared, use the JA, JB, JAB, JBE, JE, and JNE instructions. The terms *above* and *below* refer to unsigned numbers.

The remaining conditional jumps test individual flag bits, such as overflow and parity. Notice that JE has an alternative opcode JZ. All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test. (The alternates appear in Appendix B with the instruction set listing.) For example, the JA instruction (jump if above) has the alternative JNBE (jump if not below or equal). A JA functions exactly as a JNBE, but a JNBE is awkward in many cases when compared to a JA.

The conditional jump instructions all test flag bits except for JCXZ (jump if CX = 0) and JECXZ (jump if ECX = 0). Instead of testing flag bits, JCXZ directly tests the contents of the CX register without affecting the flag bits, and JECXZ tests the contents of the ECX register. For the JCXZ instruction, if CX = 0, a jump occurs, and if CX != 0, no jump occurs.

Unsigned numbers		Signed numbers	
255	FFH	+127	7FH
254	FEH	+126	7EH
...		...	
132	84H	+2	02H
131	83H	+1	01H
130	82H	+0	00H
129	81H	-1	FFH
128	80H	-2	FEH
...		...	
4	04H	-124	84H
3	03H	-125	83H
2	02H	-126	82H
1	01H	-127	81H
0	00H	-128	80H

FIGURE 6-5 Signed and unsigned numbers follow different orders.

Likewise for the JECXZ instruction, if ECX = 0, a jump occurs; if ECX != 0, no jump occurs. In the Pentium 4 or Core2 operated in the 64-bit mode, the JRCXZ instruction jumps if RCX = 0.

A program that uses JCXZ appears in Example 6-6. Here, the SCASB instruction searches a table for 0AH. Following the search, a JCXZ instruction tests CX to see if the count has reached zero. If the count is zero, the 0AH is not found in the table. The carry flag is used in this example to pass the not found condition back to the calling program. Another method used to test to see if the data are found is the JNE instruction. If JNE replaces JCXZ, it performs the same function. After the SCASB instruction executes, the flags indicate a not-equal condition if the data were not found in the table.

EXAMPLE 6-6

```

;Instructions that search a table of 100H bytes for 0AH
;The offset address of TABLE is assumed to be in SI
;
0017 B9 0064      MOV CX,100          ;load counter
001A B0 0A        MOV AL,0AH         ;load AL with 0AH
001C FC          CLD                 ;auto-increment
001D F2/AE        REPNE SCASB        ;search for 0AH
001F F9          STC                 ;set carry if found
0020 E3 01        JCXZ NOT_FOUND     ;if not found
0022              NOT_FOUND

```

The Conditional Set Instructions. In addition to the conditional jump instructions, the 80386 through the Core2 processors also contain conditional set instructions. The conditions tested by conditional jumps are put to work with the conditional set instructions. The conditional set instructions set a byte to either 01H or clear a byte to 00H, depending on the outcome of the condition under test. Table 6-2 lists the available forms of the conditional set instructions.

These instructions are useful where a condition must be tested at a point much later in the program. For example, a byte can be set to indicate that the carry is cleared at some point in the program by using the SETNC MEM instruction. This instruction places 01H into memory location MEM if carry is cleared, and 00H into MEM if carry is set. The contents of MEM can be tested at a later point in the program to determine if carry is cleared at the point where the SETNC MEM instruction executed.

TABLE 6-2 Conditional set instructions.

<i>Assembly Language</i>	<i>Tested Condition</i>	<i>Operation</i>
SETA	Z = 0 and C = 0	Set if above
SETAE	C = 0	Set if above or equal
SETB	C = 1	Set if below
SETBE	Z = 1 or C = 1	Set if below or equal
SETC	C = 1	Set if carry
SETE or SETZ	Z = 1	Set if equal or set if zero
SETG	Z = 0 and S = 0	Set if greater than
SETGE	S = 0	Set if greater than or equal
SETL	S != 0	Set if less than
SETLE	Z = 1 or S != 0	Set if less than or equal
SETNC	C = 0	Set if no carry
SETNE or SETNZ	Z = 0	Set if not equal or set if not zero
SETNO	O = 0	Set if no overflow
SETNS	S = 0	Set if no sign (positive)
SETNP or SETPO	P = 0	Set if no parity or set if parity odd
SETO	O = 1	Set if overflow
SETP or SETPE	P = 1	Set if parity or set if parity even
SETS	S = 1	Set if sign (negative)

LOOP

The LOOP instruction is a combination of a decrement CX and the JNZ conditional jump. In the 8086 through the 80286 processors, LOOP decrements CX; if CX != 0, it jumps to the address indicated by the label. If CX becomes 0, the next sequential instruction executes. In the 80386 and above, LOOP decrements either CX or ECX, depending upon the instruction mode. If the 80386 and above operate in the 16-bit instruction mode, LOOP uses CX; if operated in the 32-bit instruction mode, LOOP uses ECX. This default is changed by the LOOPW (using CX) and LOOPD (using ECX) instructions in the 80386 through the Core2. In the 64-bit mode, the loop counter is in RCX and is 64 bits wide.

Example 6-7 shows how data in one block of memory (BLOCK1) add to data in a second block of memory (BLOCK2), using LOOP to control how many numbers add. The LODSW and STOSW instructions access the data in BLOCK1 and BLOCK2. The ADD AX, ES:[DI] instruction accesses the data in BLOCK2 located in the extra segment. The only reason that BLOCK2 is in the extra segment is that DI addresses extra segment data for the STOSW instruction. The .STARTUP directive only loads DS with the address of the data segment.

In this example, the extra segment also addresses data in the data segment, so the contents of DS are copied to ES through the accumulator. Unfortunately, there is no direct move from segment register to segment register instruction.

EXAMPLE 6-7

```

;A program that sums the contents of BLOCK1 and BLOCK2
;and stores the results on top of the data in BLOCK2.
;
.MODEL SMALL                                ;select SMALL model
.DATA                                       ;start data segment
0000 0064[ BLOCK1 DW 100 DUP(?)           ;100 words for BLOCK1
        ]
00C8 0064[ BLOCK2 DW 100 DUP(?)           ;100 words for BLOCK2
        ]
0000 .CODE                                ;start code segment
        .STARTUP                          ;start program
0017 8C D8      MOV AX,DS                  ;overlap DS and ES
0019 8E C0      MOV ES,AX
001B FC        CLD                         ;select auto-increment
001C B9 0064    MOV CX,100                 ;load counter
001F BE 0000 R  MOV SI,OFFSET BLOCK1      ;address BLOCK1
0022 BF 00C8 R  MOV DI,OFFSET BLOCK2      ;address BLOCK2
0025 AD         LODSW                      ;load AX with BLOCK1
0026 26:03 05   L1: ADD AX,ES:[DI]         ;add BLOCK2
0029 AB         STOSW                      ;save answer
002A E2 F9     LOOP L1                    ;repeat 100 times
        .EXIT
END

```

Conditional LOOPS. As with REP, the LOOP instruction also has conditional forms: LOOPE and LOOPNE. The LOOPE (**loop while equal**) instruction jumps if CX != 0 while an equal condition exists. It will exit the loop if the condition is not equal or if the CX register decrements to 0. The LOOPNE (**loop while not equal**) instruction jumps if CX != 0 while a not-equal condition exists. It will exit the loop if the condition is equal or if the CX register decrements to 0. In the 80386 through the Core2 processors, the conditional LOOP instruction can use either CX or ECX as the counter. The LOOPEW/LOOPED or LOOPNEW/LOOPNE instructions override the instruction mode if needed. Under 64-bit operation, the loop counter uses RCX and is 64 bits in width. As with the conditional repeat instructions, alternates exist for LOOPE and LOOPNE. The LOOPE instruction is the same as LOOPZ, and the LOOPNE instruction is the same as LOOPNZ. In most programs, only the LOOPE and LOOPNE apply.

CONTROLLING THE FLOW OF THE PROGRAM

It is much easier to use the assembly language statements .IF, .ELSE, .ELSEIF, and .ENDIF to control the flow of the program than it is to use the correct conditional jump statement. These statements always indicate a special assembly language command to MASM. Note that the control flow assembly language statements beginning with a period are only available to MASM version 6.xx, and not to earlier versions of the assembler such as 5.10. Other statements developed in this chapter include the .REPEAT-.UNTIL and .WHILE-.ENDW statements. These statements (**the dot commands**) do not function when using the Visual C++ inline assembler.

Example 6–8(a) shows how these statements are used to control the flow of a program by testing AL for the ASCII letters A through F. If the contents of AL are A through F, 7 is subtracted from AL.

Accomplishing the same task using the Visual C++ inline assembler is usually handled in C++ rather than in assembly language. Example 6–8(b) shows the same task using the inline assembler in Visual C++ and conditional jumps in assembly language. It also shows how to use a label in an assembly block in Visual C++. This illustrates that it is more difficult to accomplish the same task without the dot commands. Never use uppercase for assembly language commands with the inline assembler because some of them are reserved by C++ and will cause problems.

EXAMPLE 6–8 (a)

```
.IF AL >= 'A' && AL <= 'F'
    SUB AL, 7
.ENDIF
SUB AL, 30H
```

EXAMPLE 6–8 (b)

```
char temp;
_asm{
    mov     al,temp
    cmp     al,41h
    jnb     Later
    cmp     al,46h
    ja      Later
    sub     al,7
Later:
    sub     al,30h
    mov     temp,al
}
```

In Example 6–8(a) notice how the && symbol represents the AND function in the .IF statement. There is no .if in Example 6–8(b) because the same operation was performed by using a few compare (CMP) instructions to accomplish the same task. See Table 6–3 for a complete list of relational operators used with the .IF statement. Note that many of these conditions (such as &&) are also used by many high-level languages such as C/C++.

Example 6–9 shows another example of the conditional .IF directive that converts all ASCII-coded letters to uppercase. First, the keyboard is read without echo using DOS INT 21H function 06H, and then the .IF statement converts the character into uppercase, if needed. In this example, the logical AND function (&&) is used to determine if the character is in lowercase. If it is lowercase, 20H is subtracted, converting to uppercase. This program reads a key from the keyboard and converts it to uppercase before displaying it. Notice also how the program terminates when the control C key (ASCII = 03H) is typed. The .LISTALL directive causes all assembler generated statements to be listed, including the label @Startup generated by

the .STARTUP directive. The .EXIT directive also is expanded by .LISTALL to show the use of the DOS INT 21H function 4CH, which returns control to DOS.

Operator	Function
==	Equal or the same as
!=	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
&	Bit test
!	Logical inversion
&&	Logical AND
	Logical OR
	Or

TABLE 6-3 Relational operators used with the .IF statement in assembly language.

EXAMPLE 6-9

```

;A DOS program that reads the keyboard and converts all
;lowercase data to uppercase before displaying it.
;
;This program is terminated with a control-C
;
.MODEL TINY                                ;select tiny model
.LISTALL                                  ;list all statements
.CODE                                     ;start code segment
.STARTUP                                  ;start program
0000
0100 * @Startup
0100 B4 06 MAIN1: MOV AH,6                ;read key without echo
0102 B2 FF MOV DL,0FFH
0104 CD 21 INT 21H
0106 74 F8 JE MAIN1                    ;if no key
0108 3C 03 CMP AL,3                    ;test for control-C
010A 74 10 JE MAIN2                    ;if control-C

        .IF AL >= 'a' && AL <= 'z'

010C 3C 61 * cmp al,'a'
010E 72 06 * jnb @C0001
0110 3C 7A * cmp al,'z'
0112 77 02 * ja @C0001

0114 2C 20 SUB AL,20H

        .ENDIF

0116 * @C0001:
0116 8A D0 MOV DL,AL                    ;echo character to display
0118 CD 21 INT 21H
011A EB E4 JMP MAIN1                    ;repeat
011C
011C B4 4C * MAIN2: .EXIT MOV AH,4CH
011E CD 21 * INT 21H
END

```

In this program, a lowercase letter is converted to uppercase by the use of the .IF AL >= 'a' && AL <= 'z' statement. If AL contains a value that is greater than or equal to a lowercase a, and less than or equal to a lowercase z (a value of a through z), the statement between the .IF and .ENDIF executes.

This statement (SUB AL,20H) subtracts 20H from the lowercase letter to change it to an uppercase letter. Notice how the assembler program implements the .IF statement (see lines that begin with *). The label @C0001 is an assembler-generated label used by the conditional jump statements placed in the program by the .IF statement. Another example that uses the conditional .IF statement appears in Example 6-10. This program reads a key from the keyboard, and then converts it to hexadecimal code. This program is not listed in expanded form.

In this example, the .IF AL >='a' && AL <='f' statement causes the next instruction (SUB AL,57H) to execute if AL contains letters a through f, converting them to hexadecimal.

If it is not between letters a and f, the next .ELSEIF statement tests it for the letters A through F. If it is the letters A through F, 37H is subtracted from AL. If neither condition is true, 30H is subtracted from AL before AL is stored at data segment memory location TEMP. The same conversion can be performed in a C++ function as illustrated in the program snippet of Example 6-10(b).

EXAMPLE 6-10(a)

```

;A DOS program that reads key and stores its hexadecimal
;value in memory location TEMP
;
.MODEL SMALL                                ;select small model
.DATA                                     ;start data segment
TEMP DB ?                                ;define TEMP
.CODE                                     ;start code segment
.STARTUP                                  ;start program
0000 MOV AH,1                            ;read keyboard
0019 CD 21 INT 21H
0023 2C 57 .IF AL >= 'a' && AL <= 'f' SUB AL,57H ;if lowercase
002F 2C 37 .ELSEIF .IF AL >= 'A' && AL <= 'F' SUB AL,37H ;if uppercase
0033 2C 30 .ELSE SUB AL,30H ;otherwise
0035 A2 0000 R .ENDIF MOV TEMP,AL ;save it in TEMP
.EXIT
END

```

EXAMPLE 6-10(b)

```

char Convert(char temp)
{
    if ( temp >= 'a' && temp <= 'f' )
        temp -= 0x57;
    else if ( temp >= 'A' && temp <= 'F' )
        temp -= 0x37;
    else
        temp -= 0x30;
    return temp;
}

```


EXAMPLE 6-12

```
;A DOS program that reads a character string from the
;keyboard and then displays it again.
;

.MODEL SMALL                ;select small model
.DATA                      ;start data segment
0000 0D 0A                 MES DB 13,10 ;return and line feed
0002 0100[                 BUF DB 256 DUP(?) ;character string buffer
    00
    ]
0000                      ;start code segment
.CODE
.STARTUP                  ;start program
0017 8C D8                MOV AX,DX ;overlap DS with ES
0019 8C C0                MOV ES,AX
001B FC                  CLD ;select auto-increment
001C BF 0002 R            MOV DI,OFFSET BUF ;address buffer

.REPEAT                  ;repeat until enter

001F * 0C0001:
001F B4 01                MOV AH,1 ;read key
0021 CD 21                INT 21H
0023 AA                  STOSB ;store key code

.UNTIL AL == 0DH

0025 3C 0D                * cmp al,0dh
0027 75 F7                * jne 0C0001
0028 C6 45 FF 24          MOV BYTE PTR[DI-1],'a'
002C BA 0000 R            MOV DX,OFFSET MES
002E B4 09                MOV AH,9
0031 CD 21                INT 21H ;display MES

.EXIT
END
```

EXAMPLE 6-13

```
012C B9 0064             MOV CX,100 ;set count
012F BF 00C8 R            MOV DI,OFFSET THREE ;address arrays
0132 BE 0000 R            MOV SI,OFFSET ONE
0135 BB 0064 R            MOV BX,OFFSET TWO

.REPEAT

0138 * 0C0001:
0138 AC                  LODSB
0139 02 07              ADD AL,[BX]
013B AA                  STOSB
013C 43                  INC BX

.UNTILCXZ

013D E2 F9 *            LOOP 0C0001
```

There is also an .UNTILCXZ instruction available that uses the LOOP instruction to check CX for a repeat loop. The .UNTILCXZ instruction uses the CX register as a counter to repeat a loop a fixed number of times. Example 6-13 shows a sequence of instructions that uses the .UNTILCXZ instruction used to add the contents of byte-sized array ONE to bytesized array TWO. The sums are stored in array THREE. Note that each array contains 100 bytes of data, so the loop is repeated 100 times. This example assumes that array THREE is in the extra segment, and that arrays ONE and TWO are in the data segment. Notice how the LOOP instruction is inserted for the .UNTILCXZ.

PROCEDURES

The procedure (subroutine, method, or **function**) is an important part of any computer system's architecture. A procedure is a group of instructions that usually performs one task. A procedure is a reusable section of the software that is stored in memory once, but used as often as necessary.

This saves memory space and makes it easier to develop software. The only disadvantage of a procedure is that it takes the computer a small amount of time to link to the procedure and return from it. The CALL instruction links to the procedure, and the RET (**return**) instruction returns from the procedure.

The stack stores the return address whenever a procedure is called during the execution of a program. The CALL instruction pushes the address of the instruction following the CALL (**return address**) on the stack. The RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

With the assembler, there are specific rules for storing procedures. A procedure begins with the PROC directive and ends with the ENDP directive. Each directive appears with the name of the procedure. This programming structure makes it easy to locate the procedure in a program listing. The PROC directive is followed by the type of procedure: NEAR or FAR. Example 6-16 shows how the assembler uses the definition of both a near (intrasegment) and far (intersegment) procedure. In MASM version 6.x, the NEAR or FAR type can be followed by the USES statement. The USES statement allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure. The USES statement is also illustrated in Example 6-14.

When these first two procedures are compared, the only difference is the opcode of the return instruction. The near return instruction uses opcode C3H and the far return uses opcode CBH. A near return removes a 16-bit number from the stack and places it into the instruction pointer to return from the procedure in the

current code segment. A far return removes a 32-bit number from the stack and places it into both IP and CS to return from the procedure to any memory location.

Procedures that are to be used by all software (global) should be written as far procedures. Procedures that are used by a given task (local) are normally defined as near procedures. Most procedures are near procedures.

EXAMPLE 6-14

```

0000          SUMS  PROC  NEAR
0000 03 C3      ADD  AX, BX
0002 03 C1      ADD  AX, CX
0004 03 C2      ADD  AX, DX
0006 C3        RET
0007          SUMS  ENDP

0007          SUMS1 PROC  FAR
0007 03 C3      ADD  AX, BX
0009 03 C1      ADD  AX, CX
000B 03 C2      ADD  AX, DX
000D CB        RET
000E          SUMS1 ENDP

000E          SUMS3 PROC  NEAR USE BX CX DX
0011 03 C3      ADD  AX, BX
0013 03 C1      ADD  AX, CX
0015 03 C2      ADD  AX, DX
                RET
001B          SUMS  ENDP

```

CALL

The CALL instruction transfers the flow of the program to the procedure. The CALL instruction differs from the jump instruction because a CALL saves a return address on the stack. The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.

Near CALL. The near CALL instruction is 3 bytes long; the first byte contains the opcode, and the second and third bytes contain the displacement, or distance of $\pm 32K$ in the 8086 through the 80286 processors. This is identical to the form of the near jump instruction. The 80386 and above use a 32-bit displacement, when operating in the protected mode, that allows a distance of $\pm 2G$ bytes. When the near CALL executes, it first pushes the offset address of the next instruction onto the stack. The offset address of the next instruction appears in the instruction pointer (IP or EIP). After saving this return address, it then adds the displacement from bytes 2 and 3 to the IP to transfer control to the procedure. There is no short CALL instruction. A variation on the opcode exists as CALLN, but this should be avoided in favor of using the PROC statement to define the CALL as near.

Why save the IP or EIP on the stack? The instruction pointer always points to the next instruction in the program. For the CALL instruction, the contents of IP/EIP are pushed onto the stack, so program control passes to the instruction following the CALL after a procedure ends.

Figure 6-6 shows the return address (IP) stored on the stack and the call to the procedure.

Far CALL. The far CALL instruction is like a far jump because it can call a procedure stored in any memory location in the system. The far CALL is a 5-byte instruction that contains an opcode followed by the next value for the IP and CS registers. Bytes 2 and 3 contain the new contents of the IP, and bytes 4 and 5 contain the new contents for CS.

The far CALL instruction places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5 of the instruction. This allows the far CALL to call a procedure located anywhere in the memory and return from that procedure. Figure 6-7 shows how the far CALL instruction calls a far procedure. Here, the contents of IP and CS are pushed onto the stack. Next, the program branches to the procedure. A variant of the far call exists as CALLF, but this should be avoided in favor of defining the type of call instruction with the PROC statement.

In the 64-bit mode a far call is to any memory location and the information placed onto the stack is an 8-byte number. Likewise, the far return instruction also retrieves an 8-byte return address from the stack and places it into RIP.

The CALL instruction also can reference far pointers if the instruction appears as CALL FAR PTR [4*EBX] or as CALL TABLE [4*EBX], if the data in the table are defined as doubleword data with the DD directive. These instructions retrieve a 32-bit address (4 bytes long) from the data segment memory location addressed by EBX and use it as the address of a far procedure.

RET

The return instruction (RET) removes a 16-bit number (**near return**) from the stack and places it into IP, or removes a 32-bit number (**far return**) and places it into IP and CS. The near and far return instructions are both defined in the procedure's PROC directive, which automatically selects the proper return instruction. With the 80386 through the Pentium 4 processors operating in the protected mode, the far return removes 6 bytes from the stack. The first 4 bytes contain the new value for EIP and the last 2 contain the new value for CS. In the 80386 and above, a protected mode near return removes 4 bytes from the stack and places them into EIP.

When IP/EIP or IP/EIP and CS are changed, the address of the next instruction is at a new memory location. This new location is the address of the instruction that immediately follows the most recent CALL to a procedure. Figure 6–8 shows how the CALL instruction links to a procedure and how the RET instruction returns in the 8086–Core2 operating in the real mode.

There is one other form of the return instruction, which adds a number to the contents of the stack pointer (SP) after the return address is removed from the stack. A return that uses an immediate operand is ideal for use in a system that uses the C/C++ or PASCAL calling conventions.

(This is true even though the C/C++ and PASCAL calling conventions require the caller to remove stack data for many functions.) These conventions push parameters on the stack before calling a procedure. If the parameters are to be discarded upon return, the return instruction contains a number that represents the number of bytes pushed to the stack as parameters.

Example 6–17 shows how this type of return erases the data placed on the stack by a few pushes. The RET 4 adds a 4 to SP after removing the return address from the stack. Because the PUSH AX and PUSH BX together place 4 bytes of data on the stack, this return effectively deletes AX and BX from the stack. This type of return rarely appears in assembly language programs, but it is used in high-level programs to clear stack data after a procedure. Notice how parameters are addressed on the stack by using the BP register, which by default addresses the stack segment. Parameter stacking is common in procedures written for C++ or PASCAL by using the C++ or PASCAL calling conventions.

As with the CALLN and CALLF instructions, there are also variants of the return instruction: RETN and RETF. As with the CALLN and CALLF instructions, these variants should also be avoided in favor of using the PROC statement to define the type of call and return.

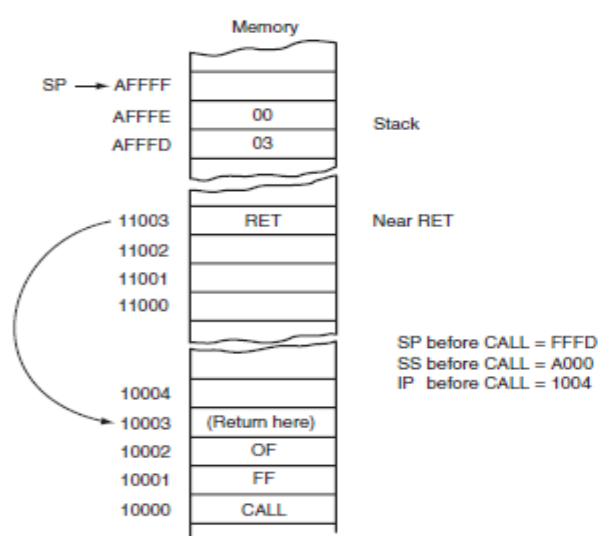


FIGURE 6–8 The effect of a near return instruction on the stack and instruction pointer.

EXAMPLE 6–17

```

0000 B8 001E      MOV    AX,30
0003 BB 0028      MOV    BX,40
0006 50           PUSH   AX
0007 53           PUSH   BX
0008 E8 0066      CALL   ADDM
                    ;stack parameter 1
                    ;stack parameter 2
                    ;add stack parameters

0071             ADDM   PROC   NEAR
0071 55           PUSH   BP
0072 8B EC        MOV    BP,SP
0074 8B 46 04     MOV    AX,[BP+4]
0077 03 46 06     ADD    AX,[BP+6]
007A 5D           POP     BP
007B C2 0004     RET     4
007E             ADDM   ENDP
                    ;save BP
                    ;address stack with BP
                    ;get parameter 1
                    ;add parameter 2
                    ;restore BP
                    ;return, dump parameters

```

INTRODUCTION TO INTERRUPTS

An interrupt is either a **hardware-generated CALL** (externally derived from a hardware signal) or a **software-generated CALL** (internally derived from the execution of an instruction or by some other internal event). At times, an internal interrupt is called an *exception*. Either type interrupts the program by calling an **interrupt service procedure** (ISP) or interrupt handler.

This section explains software interrupts, which are special types of CALL instructions. This section describes the three types of software interrupt instructions (INT, INTO, and INT 3), provides a map of the interrupt vectors, and explains the purpose of the special interrupt return instruction (IRET).

Interrupt Vectors

An **interrupt vector** is a 4-byte number stored in the first 1024 bytes of the memory (00000H–003FFH) when the microprocessor operates in the real mode. In the protected mode, the vector table is replaced by an interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts. There are 256 different interrupt vectors, and each vector contains the address of an interrupt service procedure. Table 6–4 lists the interrupt vectors, with a brief description and the memory location of each vector for the real mode. Each vector contains a value for IP and CS that forms the address of the interrupt service procedure. The first 2 bytes contain the IP, and the last 2 bytes contain the CS.

TABLE 6–4 Interrupt vectors defined by Intel.

Number	Address	Microprocessor	Function
0	0H–3H	All	Divide error
1	4H–7H	All	Single-step
2	8–BH	All	NMI pin
3	CH–FH	All	Breakpoint
4	10H–13H	All	Interrupt on overflow
5	14H–17H	80186–Core2	Bound instruction
6	18H–1BH	80186–Core2	Invalid opcode
7	1CH–1FH	80186–Core2	Coprocessor emulation
8	20H–23H	80386–Core2	Double fault
9	24H–27H	80386	Coprocessor segment overrun
A	28H–2BH	80386–Core2	Invalid task state segment
B	2CH–2FH	80386–Core2	Segment not present
C	30H–33H	80386–Core2	Stack fault
D	34H–37H	80386–Core2	General protection fault (GPF)
E	38H–3BH	80386–Core2	Page fault
F	3CH–3FH	—	Reserved
10	40H–43H	80286–Core2	Floating-point error
11	44H–47H	80486SX	Alignment check interrupt
12	48H–4BH	Pentium–Core2	Machine check exception
13–1F	4CH–7FH	—	Reserved
20–FF	80H–3FFH	—	User interrupts

Intel reserves the first 32 interrupt vectors for the present and future microprocessor products. The remaining interrupt vectors (32–255) are available for the user. Some of the reserved vectors are for errors that occur during the execution of software, such as the divide error interrupt. Some vectors are reserved for the coprocessor. Still others occur for normal events in the system. In a personal computer, the reserved vectors are used for system functions, as detailed later in this section. Vectors 1–6, 7, 9, 16, and 17 function in the real mode and protected mode; the remaining vectors function only in the protected mode.

Interrupt Instructions

The microprocessor has three different interrupt instructions that are available to the programmer: INT, INTO, and INT 3. In the real mode, each of these instructions fetches a vector from the vector table, and then calls the procedure stored at the location addressed by the vector. In the protected mode, each of these instructions fetches an interrupt descriptor from the interrupt descriptor table. The descriptor specifies the address of the interrupt service procedure. The interrupt call is similar to a far CALL instruction because it places the return address (IP/EIP and CS) on the stack.

INTs. There are 256 different software interrupt instructions (INTs) available to the programmer. Each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH). For example, the INT 100 uses interrupt vector 100, which appears at memory address 190H–193H. The address of the interrupt vector is determined by multiplying the interrupt type number by 4. For example, the INT 10H instruction calls the

interrupt service procedure whose address is stored beginning at memory location 40H (10H × 4) in the real mode. In the protected mode, the interrupt descriptor is located by multiplying the type number by 8 instead of 4 because each descriptor is 8 bytes long. Each INT instruction is 2 bytes long. The first byte contains the opcode, and the second byte contains the vector type number. The only exception to this is INT 3, a 1-byte special software interrupt used for breakpoints.

Whenever a software interrupt instruction executes, it (1) pushes the flags onto the stack, (2) clears the T and I flag bits, (3) pushes CS onto the stack, (4) fetches the new value for CS from the interrupt vector, (5) pushes IP/EIP onto the stack, (6) fetches the new value for IP/EIP from the vector, and (7) jumps to the new location addressed by CS and IP/EIP.

The INT instruction performs as a far CALL except that it not only pushes CS and IP onto the stack, but it also pushes the flags onto the stack. The INT instruction performs the operation of a PUSHF, followed by a far CALL instruction.

Notice that when the INT instruction executes, it clears the interrupt flag (I), which controls the external hardware interrupt input pin INTR (interrupt request). When I = 0, the microprocessor disables the INTR pin; when I = 1, the microprocessor enables the INTR pin. Software interrupts are most commonly used to call system procedures because the address of the system function need not be known. The system procedures are common to all system and application software. The interrupts often control printers, video displays, and disk drives. Besides relieving the program from remembering the address of the system call, the INT instruction replaces a far CALL that would otherwise be used to call a system function. The INT instruction is 2 bytes long, whereas the far CALL is 5 bytes long. Each time that the INT instruction replaces a far CALL, it saves 3 bytes of memory in a program. This can amount to a sizable saving if the INT instruction often appears in a program, as it does for system calls.

IRET/IRETD. The interrupt return instruction (IRET) is used only with software or hardware interrupt service procedures. Unlike a simple return instruction (RET), the IRET instruction will (1) pop stack data back into the IP, (2) pop stack data back into CS, and (3) pop stack data back into the flag register. The IRET instruction accomplishes the same tasks as the POPF, followed by a far RET instruction.

Whenever an IRET instruction executes, it restores the contents of I and T from the stack. This is important because it preserves the state of these flag bits. If interrupts were enabled before an interrupt service procedure, they are automatically re-enabled by the IRET instruction because it restores the flag register.

In the 80386 through the Core2 processors, the IRETD instruction is used to return from an interrupt service procedure that is called in the protected mode. It differs from the IRET because it pops a 32-bit instruction pointer (EIP) from the stack. The IRET is used in the real mode and the IRETD is used in the protected mode.

INT 3. An INT 3 instruction is a special software interrupt designed to function as a breakpoint. The difference between it and the other software interrupts is that INT 3 is a 1-byte instruction, while the others are 2-byte instructions. It is common to insert an INT 3 instruction in software to interrupt or break the flow of the software. This function is called a breakpoint. A breakpoint occurs for any software interrupt, but because INT 3 is 1 byte long, it is easier to use for this function. Breakpoints help to debug faulty software.

INTO. Interrupt on overflow (INTO) is a conditional software interrupt that tests the overflow flag (O). If O = 0, the INTO instruction performs no operation; if O = 1 and an INTO instruction executes, an interrupt occurs via vector type number 4.

The INTO instruction appears in software that adds or subtracts signed binary numbers. With these operations, it is possible to have an overflow. Either the JO instruction or INTO instruction detects the overflow condition.

An Interrupt Service Procedure. Suppose that, in a particular system, a procedure is required to add the contents of DI, SI, BP, and BX and then save the sum in AX. Because this is a common task in this system, it may occasionally be worthwhile to develop the task as a software interrupt. Realize that interrupts are usually reserved for system events and this is merely an example showing how an interrupt service procedure appears. Example 6–18 shows this software interrupt. The main difference between this procedure and a normal far procedure is that it ends with the IRET instruction instead of the RET instruction, and the contents of the flag register are saved on the stack during its execution. It is also important to save all registers that are changed by the procedure using USES.

EXAMPLE 6-18

			INTS	PROC	FAR	USES	AX
0000				ADD		AX, BX	
0000 03 C3				ADD		AX, BP	
0002 03 05				ADD		AX, DI	
0004 03 C7				ADD		AX, SI	
0006 03 C6				IRET			
0008 CF				ENDP			
0009			INTS				

Interrupt Control

Although this section does not explain hardware interrupts, two instructions are introduced that control the INTR pin. The **set interrupt flag** instruction (STI) places a 1 into the I flag bit, which enables the INTR pin. The **clear interrupt flag** instruction (CLI) places a 0 into the I flag bit, which disables the INTR pin. The STI instruction enables INTR and the CLI instruction disables INTR. In a software interrupt service procedure, hardware interrupts are enabled as one of the first steps. This is accomplished by the STI instruction. The reason interrupts are enabled early in an interrupt service procedure is that just about all of the I/O devices in the personal computer are interrupt-processed. If the interrupts are disabled too long, severe system problems result.

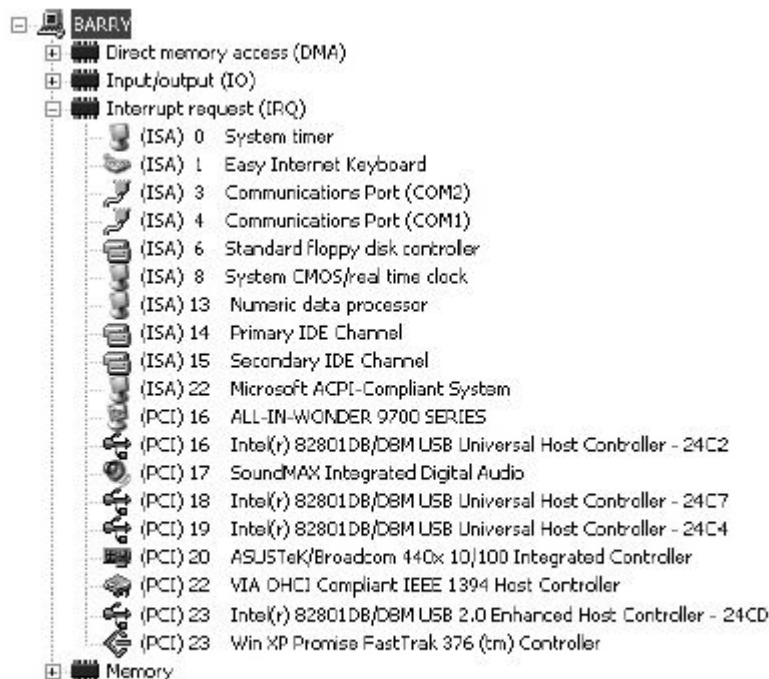


FIGURE 6-9 Interrupts in a typical personal computer.

Interrupts in the Personal Computer

The interrupts found in the personal computer differ somewhat from the ones presented in Table 6-4. The reason that they differ is that the original personal computers are 8086/8088-based systems. This meant that they only contained Intel-specified interrupts 0-4. This design has been carried forward so that newer systems are compatible with the early personal computers.

Access to the protected mode interrupt structure in use by Windows is accomplished through kernel functions Microsoft provides and cannot be directly addressed. Protected mode interrupts use an interrupt descriptor table, which is beyond the scope of the text at this point. Protected mode interrupts are discussed completely in later chapters. Figure 6-9 illustrates the interrupts available in the author's computer. The interrupt assignments are viewable in the control panel of Windows under Performance and Maintenance by clicking on System and selecting Hardware and then Device Manager. Now click on View and select Device by Type and finally Interrupts.

64-Bit Mode Interrupts

The 64-bit system uses the IRETQ instruction to return from an interrupt service procedure. The main difference between IRET/IRETD and the IRETQ instruction is that IRETQ retrieves an 8-byte return address from the stack. The IRETQ instruction also retrieves the 32-bit EFLAG register from the stack and places it

into the RFLAG register. It appears that Intel has no plans for using the leftmost 32 bits of the RFLAG register. Otherwise, 64-bit mode interrupts are the same as 32-bit mode interrupts.

MACHINE CONTROL AND MISCELLANEOUS INSTRUCTIONS

The last category of real mode instructions found in the microprocessor is the machine control and miscellaneous group. These instructions provide control of the carry bit, sample the BUSY/TEST pin, and perform various other functions. Because many of these instructions are used in hardware control, they need only be explained briefly at this point.

Controlling the Carry Flag Bit

The carry flag (C) propagates the carry or borrow in multiple-word/doubleword addition and subtraction. It also can indicate errors in assembly language procedures. Three instructions control the contents of the carry flag: STC (set carry), CLC (clear carry), and CMC (complement carry). Because the carry flag is seldom used except with multiple-word addition and subtraction, it is available for other uses. The most common task for the carry flag is to indicate an error upon return from a procedure. Suppose that a procedure reads data from a disk memory file. This operation can be successful, or an error such as file-not-found can occur. Upon return from this procedure, if C = 1, an error has occurred; if C = 0, no error occurred. Most of the DOS and BIOS procedures use the carry flag to indicate error conditions. This flag is not available in Visual C/C++ for use with C++.

WAIT

The WAIT instruction monitors the hardware \overline{BUSY} pin on the 80286 and 80386, and the \overline{TEST} pin on the 8086/8088. The name of this pin was changed beginning with the 80286 microprocessor from \overline{TEST} to \overline{BUSY} . If the WAIT instruction executes while the \overline{BUSY} pin = 1, nothing happens and the next instruction executes. If the \overline{BUSY} pin = 0 when the WAIT instruction executes, the microprocessor waits for the \overline{BUSY} pin to return to a logic 1. This pin inputs a busy condition when at a logic 0 level.

The $\overline{BUSY}/\overline{TEST}$ pin of the microprocessor is usually connected to the \overline{BUSY} pin of the 8087 through the 80387 numeric coprocessors. This connection allows the microprocessor to wait until the coprocessor finishes a task. Because the coprocessor is inside an 80486 through the Core2, the \overline{BUSY} pin is not present in these microprocessors.

HLT

The halt instruction (HLT) stops the execution of software. There are three ways to exit a halt: by an interrupt, by a hardware reset, or during a DMA operation. This instruction normally appears in a program to wait for an interrupt. It often synchronizes external hardware interrupts with the software system. Note that DOS and Windows both use interrupts extensively, so HLT will not halt the computer when operated under these operating systems.

NOP

When the microprocessor encounters a no operation instruction (NOP), it takes a short time to execute. In early years, before software development tools were available, a NOP, which performs absolutely no operation, was often used to pad software with space for future machine language instructions. If you are developing machine language programs, which are extremely rare, it is recommended that you place 10 or so NOPS in your program at 50-byte intervals. This is done in case you need to add instructions at some future point. A NOP may also find application in time delays to waste time. Realize that a NOP used for timing is not very accurate because of the cache and pipelines in modern microprocessors.

LOCK Prefix

The LOCK prefix appends an instruction and causes the \overline{LOCK} pin to become a logic 0. The \overline{LOCK} pin often disables external bus masters or other system components. The LOCK prefix causes the \overline{LOCK} pin to activate for only the duration of a locked instruction. If more than one sequential instruction is locked, the \overline{LOCK} pin remains a logic 0 for the duration of the sequence of locked instructions. The LOCK:MOV AL,[SI] instruction is an example of a locked instruction.

ESC

The escape (ESC) instruction passes instructions to the floating-point coprocessor from the microprocessor. Whenever an ESC instruction executes, the microprocessor provides the memory address, if required, but otherwise performs a NOP. Six bits of the ESC instruction provide the opcode to the coprocessor and begin executing a coprocessor instruction. The ESC opcode never appears in a program as ESC and in itself is considered obsolete as an opcode. In its place are a set of coprocessor instructions (FLD, FST, FMUL, etc.) that assemble as ESC instructions for the coprocessor. More detail is provided in Chapter 13, which details the 8087–Core2 numeric coprocessors.

BOUND

The BOUND instruction, first made available in the 80186 microprocessor, is a comparison instruction that may cause an interrupt (vector type number 5). This instruction compares the contents of any 16-bit or 32-bit register against the contents of two words or doublewords of memory: an upper and a lower boundary. If the value in the register compared with memory is not within the upper and lower boundary, a type 5 interrupt ensues. If it is within the boundary, the next instruction in the program executes.

For example, if the BOUND SI,DATA instruction executes, word-sized location DATA contains the lower boundary, and word-sized location DATA+2 bytes contains the upper boundary.

If the number contained in SI is less than memory location DATA or greater than memory location DATA+2 bytes, a type 5 interrupt occurs. Note that when this interrupt occurs, the return address points to the BOUND instruction, not to the instruction following BOUND. This differs from a normal interrupt, where the return address points to the next instruction in the program.

ENTER and LEAVE

The ENTER and LEAVE instructions, first made available to the 80186 microprocessor, are used with stack frames, which are mechanisms used to pass parameters to a procedure through the stack memory. The stack frame also holds local memory variables for the procedure. Stack frames provide dynamic areas of memory for procedures in multiuser environments. The ENTER instruction creates a stack frame by pushing BP onto the stack and then loading BP with the uppermost address of the stack frame. This allows stack frame variables to be accessed through the BP register. The ENTER instruction contains two operands: The first operand specifies the number of bytes to reserve for variables on the stack frame, and the second specifies the level of the procedure.

Suppose that an ENTER 8,0 instruction executes. This instruction reserves 8 bytes of memory for the stack frame and the zero specifies level 0. Figure 6–10 shows the stack frame set up by this instruction. Note that this instruction stores BP onto the top of the stack. It then subtracts 8 from the stack pointer, leaving 8 bytes of memory space for temporary data storage. The uppermost location of this 8-byte temporary storage area is addressed by BP. The LEAVE instruction reverses this process by reloading both SP and BP with their prior values. The ENTER and LEAVE instructions were used to call C++ functions in Windows 3.1, but since then, CALL has been used in modern versions of Windows for C++ functions.

FIGURE 6–10 The stack frame created by the ENTER 8,0 instruction. Notice that BP is stored beginning at the top of the stack frame. This is followed by an 8-byte area called a stack frame.

