
CHAPTER 5

Arithmetic and Logic Instructions

INTRODUCTION

In this chapter, we examine the arithmetic and logic instructions. The arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, increment, and decrement. The logic instructions include AND, OR, Exclusive-OR, NOT, shifts, rotates, and the logical compare (TEST). This chapter also presents the 80386 through the Core2 instructions XADD, SHRD, SHLD, bit tests, and bit scans. The chapter concludes with a discussion of string comparison instructions, which are used for scanning tabular data and for comparing sections of memory data. Both comparison tasks are performed efficiently with the string scan (SCAS) and string compare (CMPS) instructions.

If you are familiar with an 8-bit microprocessor, you will recognize that the 8086 through the Core2 instruction set is superior to most 8-bit microprocessors because most of the instructions have two operands instead of one. Even if this is your first microprocessor, you will quickly learn that this microprocessor possesses a powerful and easy-to-use set of arithmetic and logic instructions.

CHAPTER OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use arithmetic and logic instructions to accomplish simple binary, BCD, and ASCII arithmetic.
2. Use AND, OR, and Exclusive-OR to accomplish binary bit manipulation.
3. Use the shift and rotate instructions.
4. Explain the operation of the 80386 through the Core2 exchange and add, compare and exchange, double-precision shift, bit test, and bit scan instructions.
5. Check the contents of a table for a match with the string instructions.

5-1

ADDITION, SUBTRACTION, AND COMPARISON

The bulk of the arithmetic instructions found in any microprocessor include addition, subtraction, and comparison. In this section, addition, subtraction, and comparison instructions are illustrated. Also shown are their uses in manipulating register and memory data.

Addition

Addition (ADD) appears in many forms in the microprocessor. This section details the use of the ADD instruction for 8-, 16-, and 32-bit binary addition. A second form of addition, called **add-with-carry**, is introduced with the ADC instruction. Finally, the increment instruction (INC) is presented. Increment is a special type of addition that adds 1 to a number. In Section 5–3, other forms of addition are examined, such as BCD and ASCII. Also described is the XADD instruction, found in the 80486 through the Pentium 4.

Table 5–1 illustrates the addressing modes available to the ADD instruction. (These addressing modes include almost all those mentioned in Chapter 3.) However, because there are more than 32,000 variations of the ADD instruction in the instruction set, it is impossible to list them all in this table. The only types of addition *not* allowed are memory-to-memory and segment register. The segment registers can only be moved, pushed, or popped. Note that, as with all other instructions, the 32-bit registers are available only with the 80386 through the Core2. In the 64-bit mode of the Pentium 4 and Core2, the 64-bit registers are also used for addition.

TABLE 5–1 Example addition instructions.

<i>Assembly Language</i>	<i>Operation</i>
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL
ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location
ADD BYTE PTR [DI],3	A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location
ADD BX,[EAX+2*ECX]	The word contents of the data segment memory location addressed by EAX plus 2 times ECX add to BX with the sum stored in BX
ADD RAX,RBX	RBX adds to RAX with the sum stored in RAX (64-bit mode)
ADD EDX,[RAX+RCX]	The doubleword in EDX is added to the doubleword addressed by the sum of RAX and RCX and the sum is stored in EDX (64-bit mode)

Register Addition. Example 5–1 shows a simple sequence of instructions that uses register addition to add the contents of several registers. In this example, the contents of AX, BX, CX, and DX are added to form a 16-bit result stored in the AX register.

EXAMPLE 5–1

```
0000 03 C3      ADD AX,BX
0002 03 C1      ADD AX,CX
0004 03 C2      ADD AX,DX
```

Whenever arithmetic and logic instructions execute, the contents of the flag register change. Note that the contents of the interrupt, trap, and other flags do not change due to arithmetic and logic instructions. Only the flags located in the rightmost 8 bits of the flag register and the overflow flag change. These rightmost flags denote the result of the arithmetic or a logic operation. Any ADD instruction modifies the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags. The flag bits never change for most of the data transfer instructions presented in Chapter 4.

Immediate Addition. Immediate addition is employed whenever constant or known data are added. An 8-bit immediate addition appears in Example 5-2. In this example, DL is first loaded with 12H by using an immediate move instruction. Next, 33H is added to the 12H in DL by an immediate addition instruction. After the addition, the sum (45H) moves into register DL and the flags change, as follows:

```
Z = 0 (result not zero)
C = 0 (no carry)
A = 0 (no half-carry)
S = 0 (result positive)
P = 0 (odd parity)
O = 0 (no overflow)
```

EXAMPLE 5–2

```
0000 B2 12      MOV DL,12H
0002 80 C2 33    ADD DL,33H
```

Memory-to-Register Addition. Suppose that an application requires memory data to be added to the AL register. Example 5–3 shows an example that adds two consecutive bytes of data, stored at the data segment offset locations NUMB and NUMB+1, to the AL register.

EXAMPLE 5–3

```
0000 BF 0000 R   MOV DI,OFFSET NUMB      ;address NUMB
0003 B0 00       MOV AL,0                ;clear sum
0005 02 05       ADD AL,[DI]              ;add NUMB
0007 02 45 01     ADD AL,[DI+1]           ;add NUMB+1
```

The first instruction loads the destination index register (DI) with offset address NUMB. The DI register, used in this example, addresses data in the data segment beginning at memory location NUMB. After clearing the sum to zero, the ADD AL,[DI] instruction adds the contents of memory location NUMB to AL. Finally, the ADD AL,[DI+I] instruction adds the contents of memory location NUMB plus 1 byte to the AL register. After both ADD instructions execute, the result appears in the AL register as the sum of the contents of NUMB plus the contents of NUMB+1.

Array Addition. Memory arrays are sequential lists of data. Suppose that an array of data (ARRAY) contains 10 bytes, numbered from element 0 through element 9. Example 5–4 shows how to add the contents of array elements 3, 5, and 7 together.

This example first clears AL to 0, so it can be used to accumulate the sum. Next, register SI is loaded with a 3 to initially address array element 3. The `ADD AL,ARRAY[SI]` instruction adds the contents of array element 3 to the sum in AL. The instructions that follow add array elements 5 and 7 to the sum in AL, using a 3 in SI plus a displacement of 2 to address element 5, and a displacement of 4 to address element 7.

EXAMPLE 5-4

```

0000 B0 00          MOV AL,0           ;clear sum
0002 BE 0003        MOV SI,3          ;address element 3
0005 02 84 0000 R   ADD AL,ARRAY[SI]   ;add element 3
0009 02 84 0002 R   ADD AL,ARRAY[SI+2] ;add element 5
000D 02 84 0004 R   ADD AL,ARRAY[SI+4] ;add element 7

```

Suppose that an array of data contains 16-bit numbers used to form a 16-bit sum in register AX. Example 5-5 shows a sequence of instructions written for the 80386 and above, showing the scaled-index form of addressing to add elements 3, 5, and 7 of an area of memory called ARRAY. In this example, EBX is loaded with the address ARRAY, and ECX holds the array element number. Note how the scaling factor is used to multiply the contents of the ECX register by 2 to address words of data. (Recall that words are 2 bytes long.)

EXAMPLE 5-5

```

0000 66|BB 00000000 R   MOV EBX,OFFSET ARRAY   ;address ARRAY
0006 66|B9 00000003      MOV ECX,3           ;address element 3
000C 67&8B 04 4B        MOV AX,[EBX+2*ECX]   ;get element 3
0010 66|B9 00000005      MOV ECX,5           ;address element 5
0016 67&03 04 4B        ADD AX,[EBX+2*ECX]   ;add element 5
001A 66|B0 00000007      MOV ECX,7           ;address element 7
0020 67&03 04 4B        ADD AX,[EBX+2*ECX]   ;add element 7

```

Increment Addition. Increment addition (INC) adds 1 to a register or a memory location. The INC instruction adds 1 to any register or memory location, except a segment register. Table 5-2 illustrates some of the possible forms of the increment instructions available to the 8086-Core2 processors. As with other instructions presented thus far, it is impossible to show all variations of the INC instruction because of the large number available.

With indirect memory increments, the size of the data must be described by using the `BYTE PTR`, `WORD PTR`, `DWORD PTR`, or `QWORD PTR` directives. The reason is that the

TABLE 5-2 Example increment instructions.

<i>Assembly Language</i>	<i>Operation</i>
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD PTR[ECX]	Adds 1 to the doubleword contents of the data segment memory location addressed by ECX
INC DATA1	Adds 1 to the contents of data segment memory location DATA1
INC RCX	Adds 1 to RCX (64-bit mode)

assembler program cannot determine if, for example, the INC [DI] instruction is a byte-, word-, or doubleword-sized increment. The INC BYTE PTR [DI] instruction clearly indicates byte-sized memory data; the INC WORD PTR [DI] instruction unquestionably indicates a word-sized memory data; and the INC DWORD PTR [DI] instruction indicates doubleword-sized data. In 64-bit mode operation of the Pentium 4 and Core2, the INC QWORD PTR [RSI] instruction indicates quadword-sized data.

Example 5–6 shows how to modify Example 5–3 to use the increment instruction for addressing NUMB and NUMB + 1. Here, an INC DI instruction changes the contents of register DI from offset address NUMB to offset address NUMB + 1. Both program sequences shown in Examples 5–3 and 5–6 add the contents of NUMB and NUMB + 1. The difference between them is the way that the address is formed through the contents of the DI register using the increment instruction.

EXAMPLE 5–6

```

0000 BF 0000 R    MOV DI,OFFSET NUMB      ;address NUMB
0003 B0 00        MOV AL,0                ;clear sum
0005 02 05        ADD AL,[DI]             ;add NUMB
0007 47           INC DI                   ;increment DI
0008 02 05        ADD AL,[DI]             ;add NUMB+1

```

Increment instructions affect the flag bits, as do most other arithmetic and logic operations. The difference is that increment instructions do not affect the carry flag bit. Carry doesn't change because we often use increments in programs that depend upon the contents of the carry flag. Note that increment is used to point to the next memory element in a byte-sized array of data only. If word-sized data are addressed, it is better to use an ADD DI,2 instruction to modify the DI pointer in place of two INC DI instructions. For doubleword arrays, use the ADD DI,4 instruction to modify the DI pointer. In some cases, the carry flag must be preserved, which may mean that two or four INC instructions might appear in a program to modify a pointer.

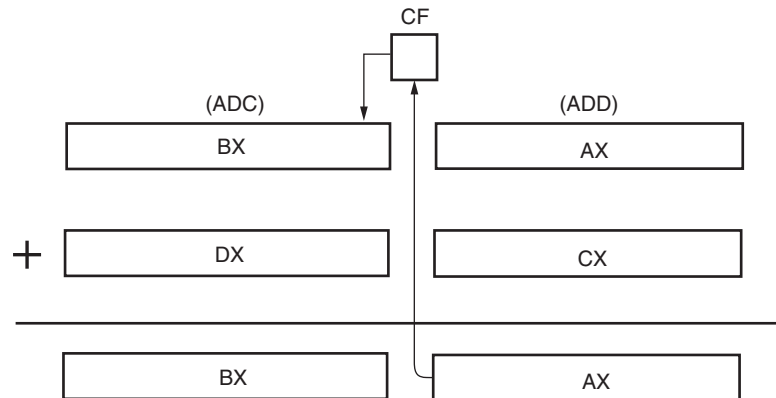
Addition-with-Carry. An addition-with-carry instruction (ADC) adds the bit in the carry flag (C) to the operand data. This instruction mainly appears in software that adds numbers that are wider than 16 bits in the 8086–80286 or wider than 32 bits in the 80386–Core2.

Table 5–3 lists several add-with-carry instructions, with comments that explain their operation. Like the ADD instruction, ADC affects the flags after the addition.

TABLE 5–3 Example add-with-carry instructions.

<i>Assembly Language</i>	<i>Operation</i>
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)
ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with the sum stored in DH
ADC BX,[BP+2]	The word contents of the stack segment memory location addressed by BP plus 2 add to BX with the sum stored in BX
ADC ECX,[EBX]	The doubleword contents of the data segment memory location addressed by EBX add to ECX with the sum stored in ECX

FIGURE 5–1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.



Suppose that a program is written for the 8086–80286 to add the 32-bit number in BX and AX to the 32-bit number in DX and CX. Figure 5–1 illustrates this addition so that the placement and function of the carry flag can be understood. This addition cannot be easily performed without adding the carry flag bit because the 8086–80286 only adds 8- or 16-bit numbers. Example 5–7 shows how the contents of registers AX and CX add to form the least significant 16 bits of the sum. This addition may or may not generate a carry. A carry appears in the carry flag if the sum is greater than FFFFH. Because it is impossible to predict a carry, the most significant 16 bits of this addition are added with the carry flag using the ADC instruction. The ADC instruction adds the 1 or the 0 in the carry flag to the most significant 16 bits of the result. This program adds BX–AX to DX–CX, with the sum appearing in BX–AX.

EXAMPLE 5–7

```
0000 03 C1      ADD AX,CX
0002 13 DA      ADC BX,DX
```

Suppose the same software is rewritten for the 80386 through the Core2, but modified to add two 64-bit numbers in the 32-bit mode. The changes required for this operation are the use of the extended registers to hold the data and modifications of the instructions for the 80386 and above. These changes are shown in Example 5–8, which adds two 64-bit numbers. In the 64-bit mode of the Pentium 4 and Core2, this addition is handled with a single ADD instruction if the location of the operands is changed to RAX and RBX as in the instruction ADD RAX,RBX, which adds RBX to RAX.

EXAMPLE 5–8

```
0000 66|03 C1      ADD EAX,ECX
0003 66|13 DA      ADC EBX,EDX
```

Exchange and Add for the 80486–Core2 Processors. A new type of addition called exchange and add (XADD) appears in the 80486 instruction set and continues through the Core2. The XADD instruction adds the source to the destination and stores the sum in the destination, as with any addition. The difference is that after the addition takes place, the original value of the destination is copied into the source operand. This is one of the few instructions that change the source.

For example, if BL = 12H and DL = 02H, and the XADD BL,DL instruction executes, the BL register contains the sum of 14H and DL becomes 12H. The sum of 14H is generated and the original destination of 12H replaces the source. This instruction functions with any register size and any memory operand, just as with the ADD instruction.

Subtraction

Many forms of subtraction (SUB) appear in the instruction set. These forms use any addressing mode with 8-, 16-, or 32-bit data. A special form of subtraction (decrement, or DEC) subtracts 1 from any register or memory location. Section 5–3 shows how BCD and ASCII data subtract. As with addition, numbers that are wider than 16 bits or 32 bits must occasionally be subtracted. The **subtract-with-borrow instruction** (SBB) performs this type of subtraction. In the 80486 through the Core2 processors, the instruction set also includes a compare and exchange instruction. In the 64-bit mode for the Pentium 4 and Core2, a 64-bit subtraction is also available.

Table 5–4 lists some of the many addressing modes allowed with the subtract instruction (SUB). There are well over 1000 possible subtraction instructions, far too many to list here. About the only types of subtraction not allowed are memory-to-memory and segment register subtractions. Like other arithmetic instructions, the subtract instruction affects the flag bits.

Register Subtraction. Example 5–9 shows a sequence of instructions that perform register subtraction. This example subtracts the 16-bit contents of registers CX and DX from the contents of register BX. After each subtraction, the microprocessor modifies the contents of the flag register. The flags change for most arithmetic and logic operations.

EXAMPLE 5–9

```
0000 2B D9      SUB BX,CX
0002 2B DA      SUB BX,DX
```

Immediate Subtraction. As with addition, the microprocessor also allows immediate operands for the subtraction of constant data. Example 5–10 presents a short sequence of instructions that subtract 44H from 22H. Here, we first load the 22H into CH using an immediate move

TABLE 5–4 Example subtraction instructions.

<i>Assembly Language</i>	<i>Operation</i>
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
SUB RDX,R8	RDX = RDX – R8 (64-bit mode)
SUB DH,6FH	DH = DH – 6FH
SUB AX,0CCCCCH	AX = AX – 0CCCCCH
SUB ESI,2000300H	ESI = ESI – 2000300H
SUB [DI],CH	Subtracts CH from the byte contents of the data segment memory addressed by DI and stores the difference in the same memory location
SUB CH,[BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH
SUB AH,TEMP	Subtracts the byte contents of memory location TEMP from AH and stores the difference in AH
SUB DI,TEMP[ESI]	Subtracts the word contents of the data segment memory location addressed by TEMP plus ESI from DI and stores the difference in DI
SUB ECX,DATA1	Subtracts the doubleword contents of memory location DATA1 from ECX and stores the difference in ECX
SUB RCX,16	RCX = RCX – 16 (64-bit mode)

instruction. Next, the SUB instruction, using immediate data 44H, subtracts 44H from the 22H. After the subtraction, the difference (0DEH) moves into the CH register. The flags change as follows for this subtraction:

Z = 0 (result not zero)
 C = 1 (borrow)
 A = 1 (half-borrow)
 S = 1 (result negative)
 P = 1 (even parity)
 O = 0 (no overflow)

EXAMPLE 5-10

```
0000 B5 22      MOV CH,22H
0002 80 ED 44    SUB CH,44H
```

Both carry flags (C and A) hold borrows after a subtraction instead of carries, as after an addition. Notice in this example that there is no overflow. This example subtracted 44H (+68) from 22H (+34), resulting in a 0DEH (−34). Because the correct 8-bit signed result is −34, there is no overflow in this example. An 8-bit overflow occurs only if the signed result is greater than +127 or less than −128.

Decrement Subtraction. Decrement subtraction (DEC) subtracts 1 from a register or the contents of a memory location. Table 5-5 lists some decrement instructions that illustrate register and memory decrements.

The decrement indirect memory data instructions require BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR because the assembler cannot distinguish a byte from a word or doubleword when an index register addresses memory. For example, DEC [SI] is vague because the assembler cannot determine whether the location addressed by SI is a byte, word, or doubleword. Using DEC BYTE PTR[SI], DEC WORD PTR[DI], or DEC DWORD PTR[SI] reveals

TABLE 5-5 Example decrement instructions.

<i>Assembly Language</i>	<i>Operation</i>
DEC BH	BH = BH − 1
DEC CX	CX = CX − 1
DEC EDX	EDX = EDX − 1
DEC R14	R14 = R14 − 1 (64-bit mode)
DEC BYTE PTR[DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR[BP]	Subtracts 1 from the word contents of the stack segment memory location addressed by BP
DEC DWORD PTR[EBX]	Subtracts 1 from the doubleword contents of the data segment memory location addressed by EBX
DEC QWORD PTR[RSI]	Subtracts 1 from the quadword contents of the memory location addressed by RSI (64-bit mode)
DEC NUMB	Subtracts 1 from the contents of data segment memory location NUMB

TABLE 5-6 Example subtraction-with-borrow instructions.

<i>Assembly Language</i>	<i>Operation</i>
SBB AH,AL	AH = AH – AL – carry
SBB AX,BX	AX = AX – BX – carry
SBB EAX,ECX	EAX = EAX – ECX – carry
SBB CL,2	CL = CL – 2 – carry
SBB RBP,8	RBP = RBP – 2 – carry (64-bit mode)
SBB BYTE PTR[DI],3	Both 3 and carry subtract from the data segment memory location addressed by DI
SBB [DI],AL	Both AL and carry subtract from the data segment memory location addressed by DI
SBB DI,[BP+2]	Both carry and the word contents of the stack segment memory location addressed by BP plus 2 subtract from DI
SBB AL,[EBX+ECX]	Both carry and the byte contents of the data segment memory location addressed by EBX plus ECX subtract from AL

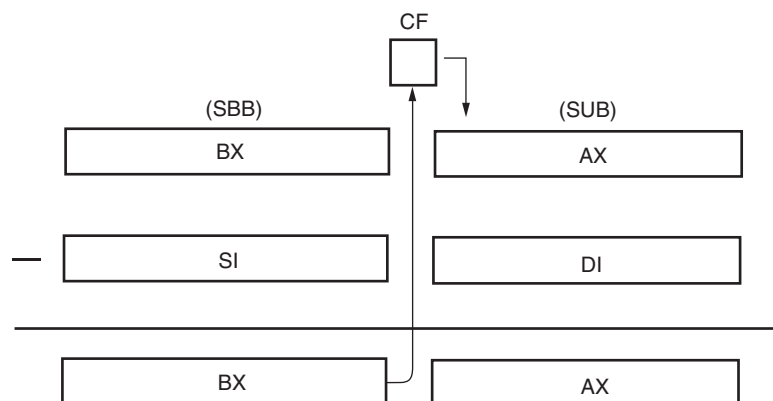
the size of the data to the assembler. In the 64-bit mode, a DEC QWORD PTR[RSI] decrement the 64-bit number stored at the address pointed to by the RSI register.

Subtraction-with-Borrow. A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference. The most common use for this instruction is for subtractions that are wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2. Wide subtractions require that borrows propagate through the subtraction, just as wide additions propagate the carry.

Table 5-6 lists several SBB instructions with comments that define their operations. Like the SUB instruction, SBB affects the flags. Notice that the immediate subtract from memory instruction in this table requires a BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directive.

When the 32-bit number held in BX and AX is subtracted from the 32-bit number held in SI and DI, the carry flag propagates the borrow between the two 16-bit subtractions. The carry flag holds the borrow for subtraction. Figure 5-2 shows how the borrow propagates through the carry flag (C) for this task. Example 5-11 shows how this subtraction is performed by a program. With wide subtraction, the least significant 16- or 32-bit data are subtracted with the SUB

FIGURE 5-2 Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.



instruction. All subsequent and more significant data are subtracted by using the SBB instruction. The example uses the SUB instruction to subtract DI from AX, then uses SBB to subtract-with-borrow SI from BX.

EXAMPLE 5-11

```
0000 2B C7      SUB AX,DI
0002 1B DE      SBB BX,SI
```

Comparison

The comparison instruction (CMP) is a subtraction that changes only the flag bits; the destination operand never changes. A comparison is useful for checking the entire contents of a register or a memory location against another value. A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

Table 5-7 lists a variety of comparison instructions that use the same addressing modes as the addition and subtraction instructions already presented. Similarly, the only disallowed forms of compare are memory-to-memory and segment register compares.

Example 5-12 shows a comparison followed by a conditional jump instruction. In this example, the contents of AL are compared with 10H. Conditional jump instructions that often follow the comparison are JA (jump above) or JB (jump below). If the JA follows the comparison, the jump occurs if the value in AL is above 10H. If the JB follows the comparison, the jump occurs if the value in AL is below 10H. In this example, the JAE instruction follows the comparison. This instruction causes the program to continue at memory location SUBER if the value in AL is 10H or above. There is also a JBE (jump below or equal) instruction that could follow the comparison to jump if the outcome is below or equal to 10H. Later chapters provide additional detail on the comparison and conditional jump instructions.

TABLE 5-7 Example comparison instructions.

<i>Assembly Language</i>	<i>Operation</i>
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP RDI,RSI	RDI – RSI (64-bit mode)
CMP AX,2000H	AX – 2000H
CMP R10W,12H	R10 (word portion) – 12H (64-bit mode)
CMP [DI],CH	CH subtracts from the byte contents of the data segment memory location addressed by DI
CMP CL,[BP]	The byte contents of the stack segment memory location addressed by BP subtracts from CL
CMP AH,TEMP	The byte contents of data segment memory location TEMP subtracts from AH
CMP DI,TEMP[BX]	The word contents of the data segment memory location addressed by TEMP plus BX subtracts from DI
CMP AL,[EDI+ESI]	The byte contents of the data segment memory location addressed by EDI plus ESI subtracts from AL

EXAMPLE 5-12

```
0000 3C 10      CMP AL,10H      ;compare AL against 10H
0002 73 1C      JAE SUBER       ;if AL is 10H or above
```

Compare and Exchange (80486–Core2 Processors Only). The compare and exchange instruction (CMPXCHG), found only in the 80486 through the Core2 instruction sets, compares the destination operand with the accumulator. If they are equal, the source operand is copied into the destination; if they are not equal, the destination operand is copied into the accumulator. This instruction functions with 8-, 16-, or 32-bit data.

The CMPXCHG CX,DX instruction is an example of the compare and exchange instruction. This instruction first compares the contents of CX with AX. If CX equals AX, DX is copied into AX; if CX is not equal to AX, CX is copied into AX. This instruction also compares AL with 8-bit data and EAX with 32-bit data if the operands are either 8- or 32-bit.

In the Pentium–Core2 processors, a CMPXCHG8B instruction is available that compares two quadwords. This is the only new data manipulation instruction provided in the Pentium–Core2 when they are compared with prior versions of the microprocessor. The compare-and-exchange-8-bytes instruction compares the 64-bit value located in EDX:EAX with a 64-bit number located in memory. An example is CMPXCHG8B TEMP. If TEMP equals EDX:EAX, TEMP is replaced with the value found in ECX:EBX; if TEMP does not equal EDX:EAX, the number found in TEMP is loaded into EDX:EAX. The Z (zero) flag bit indicates that the values are equal after the comparison.

This instruction has a bug that will cause the operating system to crash. More information about this flaw can be obtained at www.intel.com. There is also a CMPXCHG16B instruction available to the Pentium 4 when operated in 64-bit mode.

5-2**MULTIPLICATION AND DIVISION**

Only modern microprocessors contain multiplication and division instructions. Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions. Because microprocessor manufacturers were aware of this inadequacy, they incorporated multiplication and division instructions into the instruction sets of the newer microprocessors. The Pentium–Core2 processors contain special circuitry that performs a multiplication in as little as one clocking period, whereas it took over 40 clocking periods to perform the same multiplication in earlier Intel microprocessors.

Multiplication

Multiplication is performed on bytes, words, or doublewords, and can be signed integer (IMUL) or unsigned integer (MUL). Note that only the 80386 through the Core2 processors multiply 32-bit doublewords. The product after a multiplication is always a double-width product. If two 8-bit numbers are multiplied, they generate a 16-bit product; if two 16-bit numbers are multiplied, they generate a 32-bit product; and if two 32-bit numbers are multiplied, a 64-bit product is generated. In the 64-bit mode of the Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product.

Some flag bits (overflow and carry) change when the multiply instruction executes and produce predictable outcomes. The other flags also change, but their results are unpredictable and therefore are unused. In an 8-bit multiplication, if the most significant 8 bits of the result are zero, both C and O flag bits equal zero. These flag bits show that the result is 8 bits wide (C = 0) or 16 bits wide (C = 1). In a 16-bit multiplication, if the most significant 16-bits part of

TABLE 5–8 Example 8-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

the product is 0, both C and O clear to zero. In a 32-bit multiplication, both C and O indicate that the most significant 32 bits of the product are zero.

8-Bit Multiplication. With 8-bit multiplication, the multiplicand is always in the AL register, whether signed or unsigned. The multiplier can be any 8-bit register or any memory location. Immediate multiplication is not allowed unless the special signed immediate multiplication instruction, discussed later in this section, appears in a program. The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL. An example is the MUL BL instruction, which multiplies the unsigned contents of AL by the unsigned contents of BL. After the multiplication, the unsigned product is placed in AX—a double-width product. Table 5–8 illustrates some 8-bit multiplication instructions.

Suppose that BL and CL each contain two 8-bit unsigned numbers, and these numbers must be multiplied to form a 16-bit product stored in DX. This procedure cannot be accomplished by a single instruction because we can only multiply a number times the AL register for an 8-bit multiplication. Example 5–13 shows a short program that generates $DX = BL \times CL$. This example loads register BL and CL with example data 5 and 10. The product, a 50, moves into DX from AX after the multiplication by using the MOV DX,AX instruction.

EXAMPLE 5–13

```

0000 B3 05      MOV  BL,5           ;load data
0002 B1 0A      MOV  CL,10
0004 8A C1      MOV  AL,CL         ;position data
0006 F6 E3      MUL  BL           ;multiply
0008 8B D0      MOV  DX,AX        ;position product

```

For signed multiplication, the product is in binary form, if positive, and in two's complement form, if negative. These are the same forms used to store all positive and negative signed numbers used by the microprocessor. If the program of Example 5–13 multiplies two signed numbers, only the MUL instruction is changed to IMUL.

16-Bit Multiplication. Word multiplication is very similar to byte multiplication. The difference is that AX contains the multiplicand instead of AL, and the 32-bit product appears in DX–AX instead of AX. The DX register always contains the most significant 16 bits of the product, and AX contains the least significant 16 bits. As with 8-bit multiplication, the choice of the multiplier is up to the programmer. Table 5–9 shows several different 16-bit multiplication instructions.

A Special Immediate 16-Bit Multiplication. The 8086/8088 microprocessors could not perform immediate multiplication; the 80186 through the Core2 processors can do so by using a special version of the multiply instruction. Immediate multiplication must be signed multiplication, and the instruction format is different because it contains three operands. The first operand is the 16-bit destination register; the second operand is a register or memory location

TABLE 5–9 Example 16-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
MUL CX	AX is multiplied by CX; the unsigned product is in DX–AX
IMUL DI	AX is multiplied by DI; the signed product is in DX–AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX–AX

that contains the 16-bit multiplicand; and the third operand is either 8-bit or 16-bit immediate data used as the multiplier.

The IMUL CX,DX,12H instruction multiplies 12H times DX and leaves a 16-bit signed product in CX. If the immediate data are 8 bits, they sign-extend into a 16-bit number before the multiplication occurs. Another example is IMUL BX,NUMBER,1000H, which multiplies NUMBER times 1000H and leaves the product in BX. Both the destination and multiplicand must be 16-bit numbers. Although this is immediate multiplication, the restrictions placed upon it limit its utility, especially the fact that it is a signed multiplication and the product is 16 bits wide.

32-Bit Multiplication. In the 80386 and above, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers. As with 8- and 16-bit multiplication, 32-bit multiplication can be signed or unsigned by using the IMUL and MUL instructions. With 32-bit multiplication, the contents of EAX are multiplied by the operand specified with the instruction. The product (64 bits wide) is found in EDX–EAX, where EAX contains the least significant 32 bits of the product. Table 5–10 lists some of the 32-bit multiplication instructions found in the 80386 and above instruction set.

64-Bit Multiplication. The result of a 64-bit multiplication in the Pentium 4 appears in the RDX:RAX register pair as a 128-bit product. Although multiplication of this size is relatively rare, the Pentium 4 and Core2 can perform it on both signed and unsigned numbers. Table 5–11 shows a few examples of this high precision multiplication.

TABLE 5–10 Example 32-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
MUL ECX	EAX is multiplied by ECX; the unsigned product is in EDX–EAX
IMUL EDI	EAX is multiplied by EDI; the signed product is in EDX–EAX
MUL DWORD PTR[ESI]	EAX is multiplied by the doubleword contents of the data segment memory location address by ESI; the unsigned product is in EDX–EAX

TABLE 5–11 Example 64-bit multiplication instructions.

<i>Assembly Language</i>	<i>Operation</i>
MUL RCX	RAX is multiplied by RCX; the unsigned product is in RDX–RAX
IMUL RDI	RAX is multiplied by RDI; the signed product is in RDX–RAX
MUL QWORD PTR[RSI]	RAX is multiplied by the quadword contents of the memory location address by RSI; the unsigned product is in RDX–RAX

Division

As with multiplication, division occurs on 8- or 16-bit numbers in the 8086–80286 microprocessors, and on 32-bit numbers in the 80386 and above microprocessor. These numbers are signed (IDIV) or unsigned (DIV) integers. The dividend is always a double-width dividend that is divided by the operand. This means that an 8-bit division divides a 16-bit number by an 8-bit number; a 16-bit division divides a 32-bit number by a 16-bit number; and a 32-bit division divides a 64-bit number by a 32-bit number. There is no immediate division instruction available to any microprocessor. In the 64-bit mode of the Pentium 4 and Core2, a 64-bit division divides a 128-bit number by a 64-bit number.

None of the flag bits change predictably for a division. A division can result in two different types of errors; one is an attempt to divide by zero and the other is a divide overflow. A divide overflow occurs when a small number divides into a large number. For example, suppose that $AX = 3000$ and that it is divided by 2. Because the quotient for an 8-bit division appears in AL, the result of 1500 causes a divide overflow because the 1500 does not fit into AL. In either case, the microprocessor generates an interrupt if a divide error occurs. In most systems, a divide error interrupt displays an error message on the video screen. The divide error interrupt and all other interrupts for the microprocessor are explained in Chapter 6.

8-Bit Division. An 8-bit division uses the AX register to store the dividend that is divided by the contents of any 8-bit register or memory location. The quotient moves into AL after the division with AH containing a whole number remainder. For a signed division, the quotient is positive or negative; the remainder always assumes the sign of the dividend and is always an integer. For example, if $AX = 0010H$ (+16) and $BL = 0FDH$ (−3) and the IDIV BL instruction executes, $AX = 01FBH$. This represents a quotient of −5 (AL) with a remainder of 1 (AH). If, on the other hand, a −16 is divided by +3, the result will be a quotient of −5 (AL) with a remainder of −1 (AH). Table 5–12 lists some of the 8-bit division instructions.

With 8-bit division, the numbers are usually 8 bits wide. This means that one of them, the dividend, must be converted to a 16-bit wide number in AX. This is accomplished differently for signed and unsigned numbers. For the unsigned number, the most significant 8 bits must be cleared to zero (zero-extended). The MOVZX instruction described in Chapter 4 can be used to zero-extend a number in the 80386 through the Core2 processors. For signed numbers, the least significant 8 bits are sign-extended into the most significant 8 bits. In the microprocessor, a special instruction sign-extends AL into AH, or converts an 8-bit signed number in AL into a 16-bit signed number in AX. The CBW (**convert byte to word**) instruction performs this conversion. In the 80386 through the Core2, a MOVSX instruction (see Chapter 4) sign-extends a number.

TABLE 5–12 Example 8-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
DIV CL	AX is divided by CL; the unsigned quotient is in AL and the unsigned remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the signed remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP; the unsigned quotient is in AL and the unsigned remainder is in AH

EXAMPLE 5-14

```

0000 A0 0000 R      MOV  AL,NUMB      ;get NUMB
0003 B4 00          MOV  AH,0        ;zero-extend
0005 F6 36 0002 R   DIV  NUMB1       ;divide by NUMB1
0009 A2 0003 R      MOV  ANSQ,AL     ;save quotient
000C 88 26 0004 R   MOV  ANSR,AH     ;save remainder

```

Example 5-14 illustrates a short program that divides the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1. Here, the quotient is stored in location ANSQ and the remainder is stored in location ANSR. Notice how the contents of location NUMB are retrieved from memory and then zero-extended to form a 16-bit unsigned number for the dividend.

16-Bit Division. Sixteen-bit division is similar to 8-bit division, except that instead of dividing into AX, the 16-bit number is divided into DX–AX, a 32-bit dividend. The quotient appears in AX and the remainder appears in DX after a 16-bit division. Table 5-13 lists some of the 16-bit division instructions.

As with 8-bit division, numbers must often be converted to the proper form for the dividend. If a 16-bit unsigned number is placed in AX, DX must be cleared to zero. In the 80386 and above, the number is zero-extended by using the MOVZX instruction. If AX is a 16-bit signed number, the CWD (**convert word to doubleword**) instruction sign-extends it into a signed 32-bit number. If the 80386 and above is available, the MOVSX instruction can also be used to sign-extend a number.

EXAMPLE 5-15

```

0000 B8 FF9C        MOV  AX,-100      ;load a -100
0003 B9 0009        MOV  CX,9         ;load +9
0006 99            CWD                ;sign-extend
0007 F7 F9          IDIV  CX

```

Example 5-15 shows the division of two 16-bit signed numbers. Here, –100 in AX is divided by +9 in CX. The CWD instruction converts the –100 in AX to –100 in DX–AX before the division. After the division, the results appear in DX–AX as a quotient of –11 in AX and a remainder of –1 in DX.

32-Bit Division. The 80386 through the Pentium 4 processors perform 32-bit division on signed or unsigned numbers. The 64-bit contents of EDX–EAX are divided by the operand specified by the instruction, leaving a 32-bit quotient in EAX and a 32-bit remainder in EDX. Other than the size of the registers, this instruction functions in the same manner as the 8- and 16-bit divisions. Table 5-14 shows some 32-bit division instructions. The CDQ (**convert doubleword to quadword**) instruction is used before a signed division to convert the 32-bit contents of EAX into a 64-bit signed number in EDX–EAX.

TABLE 5-13 Example 16-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
DIV CX	DX–AX is divided by CX; the unsigned quotient is AX and the unsigned remainder is in DX
IDIV SI	DX–AX is divided by SI; the signed quotient is in AX and the signed remainder is in DX
DIV NUMB	DX–AX is divided by the word contents of data segment memory NUMB; the unsigned quotient is in AX and the unsigned remainder is in DX

TABLE 5–14 Example 32-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
DIV ECX	EDX–EAX is divided by ECX; the unsigned quotient is in EAX and the unsigned remainder is in EDX
IDIV DATA4	EDX–EAX is divided by the doubleword contents in data segment memory location DATA4; the signed quotient is in EAX and the signed remainder is in EDX
DIV DWORD PTR[EDI]	EDX–EAX is divided by the doubleword contents of the data segment memory location addressed by EDI; the unsigned quotient is in EAX and the unsigned remainder is in EDX

The Remainder. What is done with the remainder after a division? There are a few possible choices. The remainder could be used to round the quotient or just dropped to truncate the quotient. If the division is unsigned, rounding requires that the remainder be compared with half the divisor to decide whether to round up the quotient. The remainder could also be converted to a fractional remainder.

EXAMPLE 5–16

```

0000 F6 F3          DIV  BL          ;divide
0002 02 E4          ADD  AH,AH       ;double remainder
0004 3A E3          CMP   AH,BL      ;test for rounding
0006 72 02          JB   NEXT        ;if OK
0008 FE C0          INC   AL         ;round
000A                NEXT:

```

Example 5–16 shows a sequence of instructions that divide AX by BL and round the unsigned result. This program doubles the remainder before comparing it with BL to decide whether to round the quotient. Here, an INC instruction rounds the contents of AL after the comparison.

Suppose that a fractional remainder is required instead of an integer remainder. A fractional remainder is obtained by saving the quotient. Next, the AL register is cleared to zero. The number remaining in AX is now divided by the original operand to generate a fractional remainder.

EXAMPLE 5–17

```

0000 B8 000D        MOV  AX,13       ;load 13
0003 B3 02          MOV  BL,2        ;load 2
0005 F6 F3          DIV  BL          ;13/2
0007 A2 0003 R      MOV  ANSQ,AL     ;save quotient
000A B0 00          MOV  AL,0        ;clear AL
000C F6 F3          DIV  BL          ;generate remainder
000E A2 0004 R      MOV  ANSR,AL     ;save remainder

```

Example 5–17 shows how 13 is divided by 2. The 8-bit quotient is saved in memory location ANSQ, and then AL is cleared. Next, the contents of AX are again divided by 2 to generate a fractional remainder. After the division, the AL register equals 80H. This is 10000000_2 . If the binary point (radix) is placed before the leftmost bit of AL, the fractional remainder in AL is 0.10000000_2 or 0.5 decimal. The remainder is saved in memory location ANSR in this example.

64-Bit Division. The Pentium 4 processor operated in 64-bit mode performs 64-bit division on signed or unsigned numbers. The 64-bit division uses the RDX:RAX register pair to hold the dividend and the quotient is found in RAX and the remainder is in RDX after the division. Table 5–15 illustrates a few 64-bit division instructions.

TABLE 5–15 Example 64-bit division instructions.

<i>Assembly Language</i>	<i>Operation</i>
DIV RCX	RDX–RAX is divided by RCX; the unsigned quotient is in RAX and the unsigned remainder is in RDX
IDIV DATA4	RDX–RAX is divided by the quadword contents in memory location DATA4; the signed quotient is in RAX and the signed remainder is in RDX
DIV QWORD PTR[RDI]	RDX–RAX is divided by the quadword contents of the memory location addressed by RDI; the unsigned quotient is in RAX and the unsigned remainder is in RDX

5-3**BCD AND ASCII ARITHMETIC**

The microprocessor allows arithmetic manipulation of both BCD (**binary-coded decimal**) and ASCII (**American Standard Code for Information Interchange**) data. This is accomplished by instructions that adjust the numbers for BCD and ASCII arithmetic.

The BCD operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic. The ASCII operations are performed on ASCII data used by many programs. In many cases, BCD or ASCII arithmetic is rarely used today, but some of the operations can be used for other purposes.

None of the instructions detailed in this section of the chapter function in the 64-bit mode of the Pentium 4 or Core2. In the future it appears that the BCD and ASCII instruction will become obsolete.

BCD Arithmetic

Two arithmetic techniques operate with BCD data: addition and subtraction. The instruction set provides two instructions that correct the result of a BCD addition and a BCD subtraction. The DAA (**decimal adjust after addition**) instruction follows BCD addition, and the DAS (**decimal adjust after subtraction**) follows BCD subtraction. Both instructions correct the result of the addition or subtraction so that it is a BCD number.

For BCD data, the numbers always appear in the packed BCD form and are stored as two BCD digits per byte. The adjustment instructions function only with the AL register after BCD addition and subtraction.

DAA Instruction. The DAA instruction follows the ADD or ADC instruction to adjust the result into a BCD result. Suppose that DX and BX each contain 4-digit packed BCD numbers. Example 5–18 provides a short sample program that adds the BCD numbers in DX and BX, and stores the result in CX.

EXAMPLE 5–18

```

0000 BA 1234      MOV DX,1234H      ;load 1234 BCD
0003 BB 3099      MOV BX,3099H      ;load 3099 BCD
0006 8A C3        MOV AL,BL         ;sum BL and DL
0008 02 C2        ADD AL,DL
000A 27           DAA
000B 8A C8        MOV CL,AL         ;answer to CL
000D 9A C7        MOV AL,BH         ;sum BH, DH and carry
000F 12 C6        ADC AL,DH
0011 27           DAA
0012 8A E8        MOV CH,AL         ;answer to CH

```

Because the DAA instruction functions only with the AL register, this addition must occur 8 bits at a time. After adding the BL and DL registers, the result is adjusted with a DAA instruction before being stored in CL. Next, add BH and DH registers with carry; the result is then adjusted with DAA before being stored in CH. In this example, a 1234 is added to 3099 to generate a sum of 4333, which moves into CX after the addition. Note that 1234 BCD is the same as 1234H.

DAS Instruction. The DAS instruction functions as does the DAA instruction, except that it follows a subtraction instead of an addition. Example 5-19 is the same as Example 5-18, except that it subtracts instead of adds DX and BX. The main difference in these programs is that the DAA instructions change to DAS, and the ADD and ADC instructions change to SUB and SBB instructions.

EXAMPLE 5-19

0000 BA 1234	MOV DX,1234H	;load 1234 BCD
0003 BB 3099	MOV BX,3099H	;load 3099 BCD
0006 8A C3	MOV AL,BL	;subtract DL from BL
0008 2A C2	SUB AL,DL	
000A 2F	DAS	
000B 8A C8	MOV CL,AL	;answer to CL
000D 9A C7	MOV AL,BH	;subtract DH
000F 1A C6	SBB AL,DH	
0011 2F	DAS	
0012 8A E8	MOV CH,AL	;answer to CH

ASCII Arithmetic

The ASCII arithmetic instructions function with ASCII-coded numbers. These numbers range in value from 30H to 39H for the numbers 0–9. There are four instructions used with ASCII arithmetic operations: AAA (**ASCII adjust after addition**), AAD (**ASCII adjust before division**), AAM (**ASCII adjust after multiplication**), and AAS (**ASCII adjust after subtraction**). These instructions use register AX as the source and as the destination.

AAA Instruction. The addition of two one-digit ASCII-coded numbers will not result in any useful data. For example, if 31H and 39H are added, the result is 6AH. This ASCII addition (1 + 9) should produce a two-digit ASCII result equivalent to a 10 decimal, which is a 31H and a 30H in ASCII code. If the AAA instruction is executed after this addition, the AX register will contain a 0100H. Although this is not ASCII code, it can be converted to ASCII code by adding 3030H to AX which generates 3130H. The AAA instruction clears AH if the result is less than 10, and adds 1 to AH if the result is greater than 10.

EXAMPLE 5-20

0000 B8 0031	MOV AX,31H	;load ASCII 1
0003 04 39	ADD AL,39H	;add ASCII 9
0005 37	AAA	;adjust sum
0006 05 3030	ADD AX,3030H	;answer to ASCII

Example 5-20 shows the way ASCII addition functions in the microprocessor. Please note that AH is cleared to zero before the addition by using the MOV AX,31H instruction. The operand of 0031H places 00H in AH and 31H into AL.

AAD Instruction. Unlike all other adjustment instructions, the AAD instruction appears before a division. The AAD instruction requires that the AX register contain a two-digit unpacked BCD number (not ASCII) before executing. After adjusting the AX register with AAD, it is divided by an unpacked BCD number to generate a single-digit result in AL with any remainder in AH.

Example 5–21 illustrates how 72 in unpacked BCD is divided by 9 to produce a quotient of 8. The 0702H loaded into the AX register is adjusted by the AAD instruction to 0048H. Notice that this converts a two-digit unpacked BCD number into a binary number so it can be divided with the binary division instruction (DIV). The AAD instruction converts the unpacked BCD numbers between 00 and 99 into binary.

EXAMPLE 5–21

```

0000 B3 09          MOV    BL,9           ;load divisor
0002 B8 0702        MOV    AX,702H        ;load dividend
0005 D5 0A          AAD                 ;adjust
0007 F6 F3          DIV    BL             ;divide

```

AAM Instruction. The AAM instruction follows the multiplication instruction after multiplying two one-digit unpacked BCD numbers. Example 5–22 shows a short program that multiplies 5 times 5. The result after the multiplication is 0019H in the AX register. After adjusting the result with the AAM instruction, AX contains 0205H. This is an unpacked BCD result of 25. If 3030H is added to 0205H, it has an ASCII result of 3235H.

EXAMPLE 5–22

```

0000 B0 05          MOV    AL,5           ;load multiplicand
0002 B1 03          MOV    CL,3           ;load multiplier
0004 F6 E1          MUL    CL
0006 DA 0A          AAM                 ;adjust

```

The AAM instruction accomplishes this conversion by dividing AX by 10. The remainder is found in AL, and the quotient is in AH. Note that the second byte of the instruction contains 0AH. If the 0AH is changed to another value, AAM divides by the new value. For example, if the second byte is changed to 0BH, the AAM instruction divides by 11. This is accomplished with DB 0D4H, 0BH in place of AAM, which forces the AMM instruction to multiply by 11.

One side benefit of the AAM instruction is that AAM converts from binary to unpacked BCD. If a binary number between 0000H and 0063H appears in the AX register, the AAM instruction converts it to BCD. For example, if AX contains a 0060H before AAM, it will contain 0906H after AAM executes. This is the unpacked BCD equivalent of 96 decimal. If 3030H is added to 0906H, the result changes to ASCII code.

Example 5–23 shows how the 16-bit binary content of AX is converted to a four-digit ASCII character string by using division and the AAM instruction. Note that this works for numbers between 0 and 9999. First DX is cleared and then DX–AX is divided by 100. For example, if $AX = 245_{10}$, $AX = 2$ and $DX = 45$ after the division. These separate halves are converted to BCD using AAM, and then 3030H is added to convert to ASCII code.

EXAMPLE 5–23

```

0000 33 D2          XOR    DX,DX          ;clear DX
0002 B9 0064        MOV    CX,100         ;divide DX-AX by 100
0005 F7 F1          DIV    CX
0007 D4 0A          AAM                 ;convert to BCD
0009 05 3030        ADD    AX,3030H       ;convert to ASCII
000C 92             XCHG    AX,DX         ;repeat for remainder
000D D4 0A          AAM
000F 05 3030        ADD    AX,3030H

```

Example 5–24 uses the DOS 21H function AH = 02H to display a sample number in decimal on the video display using the AAM instruction. Notice how AAM is used to convert AL into BCD. Next, ADD AX,3030H converts the BCD code in AX into ASCII for display with DOS INT 21H. Once the data are converted to ASCII code, they are displayed by loading DL

with the most significant digit from AH. Next, the least significant digit is displayed from AL. Note that the DOS INT 21H function calls change AL.

EXAMPLE 5-24

```

;A program that displays the number in AL, loaded
;with the first instruction (48H).
;
0000      .MODEL TINY          ;select tiny model
          .CODE                ;start code segment
          .STARTUP             ;start program
0100 B0 48      MOV AL,48H      ;load test data
0102 B4 00      MOV AH,0        ;clear AH
0104 D4 0A      AAM              ;convert to BCD
0106 05 3030    ADD AX,3030H    ;convert to ASCII
0109 8A D4      MOV DL,AH       ;display most-significant digit
010B B4 02      MOV AH,2
010D 50      PUSH AX
010E CD 21      INT 21H
0110 58      POP AX
0111 8A D0      MOV DL,AL       ;display least-significant digit
0113 CD 21      INT 21H
          .EXIT                ;exit to DOS
          END

```

AAS Instruction. Like other ASCII adjust instructions, AAS adjusts the AX register after an ASCII subtraction. For example, suppose that 35H subtracts from 39H. The result will be 04H, which requires no correction. Here, AAS will modify neither AH nor AL. On the other hand, if 38H is subtracted from 37H, then AL will equal 09H and the number in AH will decrement by 1. This decrement allows multiple-digit ASCII numbers to be subtracted from each other.

5-4**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

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.

<i>Assembly Language</i>	<i>Operation</i>
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)

B OR together to produce the X output. It is important to remember that 1 ORed with anything yields a 1.

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.

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.

<i>Assembly Language</i>	<i>Operation</i>
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

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.

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

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.

<i>Assembly Language</i>	<i>Operation</i>
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

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.

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 \bar{x} \bar{x} \bar{x} \bar{x}	Result

EXAMPLE 5-27

```

0000 81 C9 0600      OR   CX,0600H      ;set bits 9 and 10
0004 83 E1 FC        AND   CX,0FFFCH     ;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.

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

```

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.

TABLE 5-19 Example TEST instructions.

<i>Assembly Language</i>	<i>Operation</i>
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.

<i>Assembly Language</i>	<i>Operation</i>
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

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 5–29

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.

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

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.

5-5

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.

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

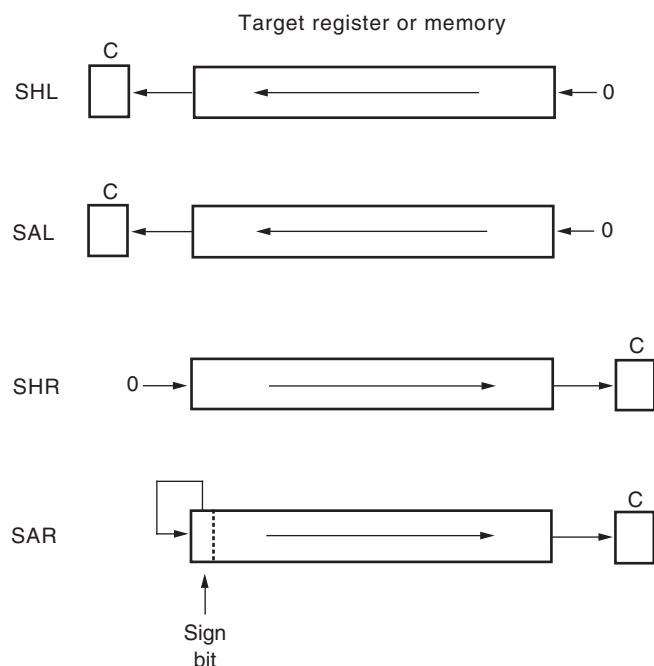


TABLE 5–22 Example shift instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

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.

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.

EXAMPLE 5–30

```

0000 C1 E2 0E          SHL DX,14

                        or

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

```

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 5-31

```

;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

```

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

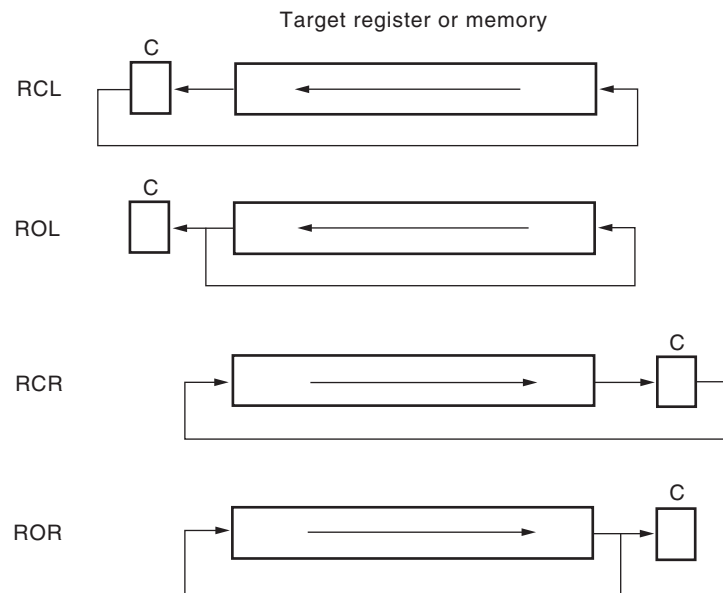


TABLE 5–23 Example rotate instructions.

<i>Assembly Language</i>	<i>Operation</i>
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

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.

5-6

STRING COMPARISONS

As illustrated in Chapter 4, the string instructions are very powerful because they allow the programmer to manipulate large blocks of data with relative ease. Block data manipulation occurs with the string instructions `MOVS`, `LODS`, `STOS`, `INS`, and `OUTS`. In this section, additional string instructions that allow a section of memory to be tested against a constant or against another section of memory are discussed. To accomplish these tasks, use the `SCAS` (**string scan**) or `CMPS` (**string compare**) instructions.

SCAS

The `SCAS` (string scan instruction) compares the `AL` register with a byte block of memory, the `AX` register with a word block of memory, or the `EAX` register (80386-Core2) with a doubleword block of memory. The `SCAS` instruction subtracts memory from `AL`, `AX`, or `EAX` without affecting either the register or the memory location. The opcode used for byte comparison is `SCASB`, the opcode used for the word comparison is `SCASW`, and the opcode used for a doubleword comparison is `SCASD`. In all cases, the contents of the extra segment memory location addressed by `DI` is compared with `AL`, `AX`, or `EAX`. Recall that this default segment (`ES`) cannot be changed with a segment override prefix.

Like the other string instructions, `SCAS` instructions use the direction flag (`D`) to select either auto-increment or auto-decrement operation for `DI`. They also repeat if prefixed by a conditional repeat prefix.

Suppose that a section of memory is 100 bytes long and begins at location `BLOCK`. This section of memory must be tested to see whether any location contains `00H`. The program in Example 5–33 shows how to search this part of memory for `00H` using the `SCASB` instruction. In this example, the `SCASB` instruction has an `REPNE` (**repeat while not equal**) prefix. The `REPNE` prefix causes the `SCASB` instruction to repeat until either the `CX` register reaches 0, or until an equal condition exists as the outcome of the `SCASB` instruction's comparison. Another conditional repeat prefix is `REPE` (**repeat while equal**). With either repeat prefix, the contents of `CX` decrements without affecting the flag bits. The `SCASB` instruction and the comparison it makes change the flags.

EXAMPLE 5–33

0000 BF 0011 R	MOV	DI, OFFSET BLOCK	;address data
0003 FC	CLD		;auto-increment
0004 B9 0064	MOV	CX, 100	;load counter
0007 32 C0	XOR	AL, AL	;clear AL
0009 F2/AE	REPNE	SCASB	

Suppose that you must develop a program that skips ASCII-coded spaces in a memory array. (This task appears in the procedure listed in Example 5–34.) This procedure assumes that the `DI` register already addresses the ASCII-coded character string and that the length of the string is 256 bytes or fewer. Because this program is to skip spaces (`20H`), the `REPE` prefix is used with a `SCASB` instruction. The `SCASB` instruction repeats the comparison, searching for a `20H`, as long as an equal condition exists.

EXAMPLE 5–34

0000 FC	CLD	;auto-increment
0001 B9 0100	MOV	CX, 256 ;load counter
0004 B0 20	MOV	AL, 20H ;get space
0006 F3/AE	REPE	SCASB

CMPS

The CMPS (compare strings instruction) always compares two sections of memory data as bytes (CMPSB), words (CMPSW), or doublewords (CMPSD). Note that only the 80386 through Core2 can use doublewords. In the Pentium 4 or Core2 operated in 64-bit mode, a CMPSQ instruction uses quadwords. The contents of the data segment memory location addressed by SI are compared with the contents of the extra segment memory location addressed by DI. The CMPS instruction increments or decrements both SI and DI. The CMPS instruction is normally used with either the REPE or REPNE prefix. Alternates to these prefixes are REPZ (repeat while zero) and REPNZ (repeat while not zero), but usually the REPE or REPNE prefixes are used in programming.

Example 5–35 illustrates a short procedure that compares two sections of memory searching for a match. The CMPSB instruction is prefixed with REPE. This causes the search to continue as long as an equal condition exists. When the CX register becomes 0 or an unequal condition exists, the CMPSB instruction stops execution. After the CMPSB instruction ends, the CX register is 0 or the flags indicate an equal condition when the two strings match. If CX is not 0 or the flags indicate a not-equal condition, the strings do not match.

EXAMPLE 5–35

0000 BE 0075 R	MOV SI,OFFSET LINE	;address LINE
0003 BF 007F R	MOV DI,OFFSET TABLE	;address TABLE
0006 FC	CLD	;auto-increment
0007 B9 000A	MOV CX,10	;load counter
000A F3/A6	REPE CMPSB	;search

5–7

SUMMARY

1. Addition (ADD) can be 8, 16, 32, or 64 bits. The ADD instruction allows any addressing mode except segment register addressing. Most flags (C, A, S, Z, P, and O) change when the ADD instruction executes. A different type of addition, add-with-carry (ADC), adds two operands and the contents of the carry flag (C). The 80486 through the Core2 processors have an additional instruction (XADD) that combines an addition with an exchange.
2. The increment instruction (INC) adds 1 to the byte, word, or doubleword contents of a register or memory location. The INC instruction affects the same flag bits as ADD except the carry flag. The BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives appear with the INC instruction when the contents of a memory location are addressed by a pointer.
3. Subtraction (SUB) is a byte, word, doubleword, or quadword and is performed on a register or a memory location. The only form of addressing not allowed by the SUB instruction is segment register addressing. The subtract instruction affects the same flags as ADD and subtracts carry if the SBB form is used.
4. The decrement (DEC) instruction subtracts 1 from the contents of a register or a memory location. The only addressing modes not allowed with DEC are immediate or segment register addressing. The DEC instruction does not affect the carry flag and is often used with BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR.
5. The comparison (CMP) instruction is a special form of subtraction that does not store the difference; instead, the flags change to reflect the difference. Comparison is used to compare an entire byte or word located in any register (except segment) or memory location.

An additional comparison instruction (CMPXCHG), which is a combination of comparison and exchange instructions, is found in the 80486–Core2 processors. In the Pentium–Core2 processors, the CMPXCHG8B instruction compares and exchanges quadword data. In the 64-bit Pentium 4 and Core2, a COMPXCHG16B instruction is available.

6. Multiplication is byte, word, or doubleword, and it can be signed (IMUL) or unsigned (MUL). The 8-bit multiplication always multiplies register AL by an operand with the product found in AX. The 16-bit multiplication always multiplies register AX by an operand with the product found in DX–AX. The 32-bit multiply always multiplies register EAX by an operand with the product found in EDX–EAX. A special IMUL immediate instruction exists on the 80186–Core2 processors that contains three operands. For example, the IMUL BX,CX,3 instruction multiplies CX by 3 and leaves the product in BX. In the Pentium 4 and Core2 with 64-bit mode enabled, multiplication is 64 bits.
7. Division is byte, word, or doubleword, and it can be signed (IDIV) or unsigned (DIV). For an 8-bit division, the AX register divides by the operand, after which the quotient appears in AL and the remainder appears in AH. In the 16-bit division, the DX–AX register divides by the operand, after which the AX register contains the quotient and DX contains the remainder. In the 32-bit division, the EDX–EAX register is divided by the operand, after which the EAX register contains the quotient and the EDX register contains the remainder. Note that the remainder after a signed division always assumes the sign of the dividend.
8. BCD data add or subtract in packed form by adjusting the result of the addition with DAA or the subtraction with DAS. ASCII data are added, subtracted, multiplied, or divided when the operations are adjusted with AAA, AAS, AAM, and AAD. These instructions do not function in the 64-bit mode.
9. The AAM instruction has an interesting added feature that allows it to convert a binary number into unpacked BCD. This instruction converts a binary number between 00H–63H into unpacked BCD in AX. The AAM instruction divides AX by 10, and leaves the remainder in AL and quotient in AH. These instructions do not function in the 64-bit mode.
10. The AND, OR, and Exclusive-OR instructions perform logic functions on a byte, word, or doubleword stored in a register or memory location. All flags change with these instructions, with carry (C) and overflow (O) cleared.
11. The TEST instruction performs the AND operation, but the logical product is lost. This instruction changes the flag bits to indicate the outcome of the test.
12. The NOT and NEG instructions perform logical inversion and arithmetic inversion. The NOT instruction one's complements an operand, and the NEG instruction two's complements an operand.
13. There are eight different shift and rotate instructions. Each of these instructions shifts or rotates a byte, word, or doubleword register or memory data. These instructions have two operands: The first is the location of the data shifted or rotated, and the second is an immediate shift or rotate count or CL. If the second operand is CL, the CL register holds the shift or rotate count. In the 80386 through the Core2 processors, two additional double-precision shifts (SHRD and SHLD) exist.
14. The scan string (SCAS) instruction compares AL, AX, or EAX with the contents of the extra segment memory location addressed by DI.
15. The string compare (CMPS) instruction compares the byte, word, or doubleword contents of two sections of memory. One section is addressed by DI in the extra segment, and the other is addressed by SI in the data segment.
16. The SCAS and CMPS instructions repeat with the REPE or REPNE prefixes. The REPE prefix repeats the string instruction while an equal condition exists, and the REPNE repeats the string instruction while a not-equal condition exists.

5-8**QUESTIONS AND PROBLEMS**

1. Select an ADD instruction that will:
 - (a) add BX to AX
 - (b) add 12H to AL
 - (c) add EDI and EBP
 - (d) add 22H to CX
 - (e) add the data addressed by SI to AL
 - (f) add CX to the data stored at memory location FROG
 - (g) add 234H to RCX
2. What is wrong with the ADD RCX,AX instruction?
3. Is it possible to add CX to DS with the ADD instruction?
4. If AX = 1001H and DX = 20FFH, list the sum and the contents of each flag register bit (C, A, S, Z, and O) after the ADD AX,DX instruction executes.
5. Develop a short sequence of instructions that adds AL, BL, CL, DL, and AH. Save the sum in the DH register.
6. Develop a short sequence of instructions that adds AX, BX, CX, DX, and SP. Save the sum in the DI register.
7. Develop a short sequence of instructions that adds ECX, EDX, and ESI. Save the sum in the EDI register.
8. Develop a short sequence of instructions that adds RCX, RDX, and RSI. Save the sum in the R12 register.
9. Select an instruction that adds BX to DX, and also adds the contents of the carry flag (C) to the result.
10. Choose an instruction that adds 1 to the contents of the SP register.
11. What is wrong with the INC [BX] instruction?
12. Select a SUB instruction that will:
 - (a) subtract BX from CX
 - (b) subtract 0EEH from DH
 - (c) subtract DI from SI
 - (d) subtract 3322H from EBP
 - (e) subtract the data address by SI from CH
 - (f) subtract the data stored 10 words after the location addressed by SI from DX
 - (g) subtract AL from memory location FROG
 - (h) subtract R9 from R10
13. If DL = 0F3H and BH = 72H, list the difference after BH is subtracted from DL and show the contents of the flag register bits.
14. Write a short sequence of instructions that subtracts the numbers in DI, SI, and BP from the AX register. Store the difference in register BX.
15. Choose an instruction that subtracts 1 from register EBX.
16. Explain what the SBB [DI-4],DX instruction accomplishes.
17. Explain the difference between the SUB and CMP instruction.
18. When two 8-bit numbers are multiplied, where is the product found?
19. When two 16-bit numbers are multiplied, what two registers hold the product? Show the registers that contain the most and least significant portions of the product.
20. When two numbers multiply, what happens to the O and C flag bits?
21. Where is the product stored for the MUL EDI instruction?
22. Write a sequence of instructions that cube the 8-bit number found in DL. Load DL with a 5 initially, and make sure that your result is a 16-bit number.

23. What is the difference between the IMUL and MUL instructions?
24. Describe the operation of the IMUL BX,DX,100H instruction.
25. When 8-bit numbers are divided, in which register is the dividend found?
26. When 16-bit numbers are divided, in which register is the quotient found?
27. When 64-bit numbers are divided, in which register is the quotient found?
28. What errors are detected during a division?
29. Explain the difference between the IDIV and DIV instructions.
30. Where is the remainder found after an 8-bit division?
31. Where is the quotient found after a 64-bit division?
32. Write a short sequence of instructions that divides the number in BL by the number in CL and then multiplies the result by 2. The final answer must be a 16-bit number stored in the DX register.
33. Which instructions are used with BCD arithmetic operations?
34. Explain how the AAM instruction converts from binary to BCD.
35. Which instructions are used with ASCII arithmetic operations?
36. Develop a sequence of instructions that converts the unsigned number in AX (values of 0–65535) into a 5-digit BCD number stored in memory, beginning at the location addressed by the BX register in the data segment. Note that the most significant character is stored first and no attempt is made to blank leading zeros.
37. Develop a sequence of instructions that adds the 8-digit BCD number in AX and BX to the 8-digit BCD number in CX and DX. (AX and CX are the most significant registers. The result must be found in CX and DX after the addition.)
38. Does the AAM instruction function in the 64-bit mode?
39. Select an AND instruction that will:
 - (a) AND BX with DX and save the result in BX
 - (b) AND 0EAH with DH
 - (c) AND DI with BP and save the result in DI
 - (d) AND 1122H with EAX
 - (e) AND the data addressed by BP with CX and save the result in memory
 - (f) AND the data stored in four words before the location addressed by SI with DX and save the result in DX
 - (g) AND AL with memory location WHAT and save the result at location WHAT
40. Develop a short sequence of instructions that clears (0) the three leftmost bits of DH without changing the remainder of DH and stores the result in BH.
41. Select an OR instruction that will:
 - (a) OR BL with AH and save the result in AH
 - (b) OR 88H with ECX
 - (c) OR DX with SI and save the result in SI
 - (d) OR 1122H with BP
 - (e) OR the data addressed by RBX with RCX and save the result in memory
 - (f) OR the data stored 40 bytes after the location addressed by BP with AL and save the result in AL
 - (g) OR AH with memory location WHEN and save the result in WHEN
42. Develop a short sequence of instructions that sets (1) the rightmost 5 bits of DI without changing the remaining bits of DI. Save the results in SI.
43. Select the XOR instruction that will:
 - (a) XOR BH with AH and save the result in AH
 - (b) XOR 99H with CL
 - (c) XOR DX with DI and save the result in DX
 - (d) XOR 1A23H with RSP