EXAMPLE 4–2

```
.MODEL SMALL
.STACK 200H ;set stack size
```

If the stack is not specified by using either method, a warning will appear when the program is linked. The warning may be ignored if the stack size is 128 bytes or fewer. The system automatically assigns (through DOS) at least 128 bytes of memory to the stack. This memory section is located in the program segment prefix (PSP), which is appended to the beginning of each program file. If you use more memory for the stack, you will erase information in the PSP that is critical to the operation of your program and the computer. This error often causes the computer program to crash. If the TINY memory model is used, the stack is automatically located at the very end of the segment, which allows for a larger stack area.

4–3 LOAD-EFFECTIVE ADDRESS

There are several load-effective address instructions in the microprocessor instruction set. The LEA instruction loads any 16-bit register with the address, as determined by the addressing mode selected for the instruction. The LDS and LES variations load any 16-bit register with the offset address retrieved from a memory location, and then load either DS or ES with a segment address retrieved from memory.

LEA

The LEA instruction loads a 16- or 32-bit register with the offset address of the data specified by the operand. As the first example in Table 4–9 shows, the operand address NUMB is loaded into register AX, not the contents of address NUMB.

By comparing LEA with MOV, it is observed that LEA BX,[DI] loads the offset address specified by [DI] (contents of DI) into the BX register; MOV BX,[DI] loads the data stored at the memory location addressed by [DI] into register BX.

Earlier in the text, several examples are presented by using the OFFSET directive. The OFFSET directive performs the same function as an LEA instruction if the operand is a displacement. For example, the MOV BX,OFFSET LIST performs the same function as LEA BX,LIST. Both instructions load the offset address of memory location LIST into the BX register. See Example 4–3 for a short program that loads SI with the address of DATA1 and DI with the address of DATA2. It then exchanges the contents of these memory locations. Note that the LEA and MOV with OFFSET instructions are both the same length (three bytes).

ABLE 4–9 Load-effective address instructions.

Assembly Language	*Operation*
LEA AX,NUMB	Loads AX with the address of NUMB
LEA EAX,NUMB	Loads EAX with the address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST	Loads DS and EDI with the 48-bit contents of data segment memory location LIST
LES BX,CAT	Loads ES and BX with the 32-bit contents of data segment memory location CAT

EXAMPLE 4–3

```
                          .MODEL SMALL               ;select SMALL model
0000                      .DATA                      ;start of DATA segment
0000  2000      DATA1     DW     2000H               ;define DATA1
0002  3000      DATA2     DW     3000H               ;define DATA2
0000                      .CODE                      ;start of CODE segment
                          .STARTUP                   ;start of program
0017  BE 0000 R           LEA    SI,DATA1            ;address DATA1 with SI
001A  BF 0002 R           MOV    DI,OFFSET DATA2     ;address DATA2 with DI

001D  8B 1C               MOV    BX,[SI]             ;exchange DATA1 with DATA2
001F  8B 0D               MOV    CX,[DI]
0021  89 0C               MOV    [SI],CX
0023  89 1D               MOV    [DI],BX
                          .EXIT                      ;exit to DOS
                          END                        ;end of file
```

But why is the LEA instruction available if the OFFSET directive accomplishes the same task? First, OFFSET only functions with simple operands such as LIST. It may not be used for an operand such as [DI], LIST [SI], and so on. The OFFSET directive is more efficient than the LEA instruction for simple operands. It takes the microprocessor longer to execute the LEA BX,LIST instruction than the MOV BX,OFFSET LIST. The 80486 microprocessor, for example, requires two clocks to execute the LEA BX,LIST instruction and only one clock to execute MOV BX,OFFSET LIST. The reason that the MOV BX,OFFSET LIST instruction executes faster is because the assembler calculates the offset address of LIST, while the microprocessor calculates the LEA instruction. The MOV BX,OFFSET LIST instruction is actually assembled as a move immediate instruction and is more efficient.

Suppose that the microprocessor executes an LEA BX,[DI] instruction and DI contains a 1000H. Because DI contains the offset address, the microprocessor transfers a copy of DI into BX. A MOV BX,DI instruction performs this task in less time and is often preferred to the LEA BX,[DI] instruction.

Another example is LEA SI,[BX + DI]. This instruction adds BX to DI and stores the sum in the SI register. The sum generated by this instruction is a modulo-64K sum. If BX = 1000H and DI = 2000H, the offset address moved into SI is 3000H. If BX = 1000H and DI = FF00H, the offset address is 0F00H instead of 10F00H. Notice that the second result is a modulo-64K sum of 0F00H. (A **modulo-64K sum** drops any carry out of the 16-bit result.)

LDS, LES,

The LDS and LES instructions load any 16-bit register with an offset address, and the DS, ES segment register with a segment address. These instructions use any of the memory-addressing modes to access a 32-bit section of memory that contains both the segment and offset address. The 32-bit section of memory contains a 16-bit offset and segment address.

Figure 4–15 illustrates an example LDS BX,[DI] instruction. This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers. The LDS and LES instructions obtain a new far address from memory. The offset address appears first, followed by the segment address. This format is used for storing all 32-bit memory addresses.

4–4 STRING DATA TRANSFERS

There are three string data transfer instructions: LODS, STOS, and MOVS. Each string instruction allows data transfers that are either a single byte, or word (or if repeated, a block of bytes, or words). Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions.

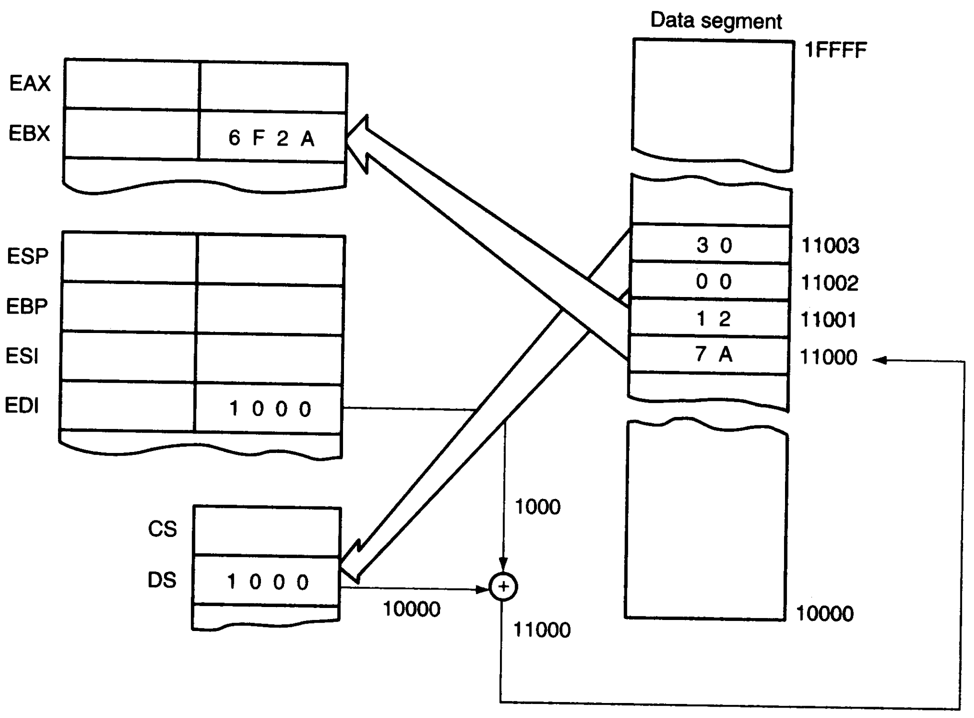


FIGURE 4–15 The LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H. This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.

The Direction Flag

The direction flag (D) (located in the flag register) selects the auto-increment (D = 0) or the auto-decrement (D = 1) operation for the DI and SI registers during string operations. The direction flag is used only with the string instructions. The CLD instruction clears the D flag (D = 0) and the STD instruction sets it (D = 1). Therefore, the CLD instruction selects the auto-increment mode (D = 0) and STD selects the auto-decrement mode (D = 1).

Whenever a string instruction transfers a byte, the contents of DI and/or SI increment or decrement by 1. If a word is transferred, the contents of DI and/or SI increment or decrement by 2. Only the actual registers used by the string instruction increment or decrement. For example, the STOSB instruction uses the DI register to address a memory location. When STOSB executes, only DI increments or decrements without affecting SI. The same is true of the LODSB instruction, which uses the SI register to address memory data. LODSB only increments/decrements SI without affecting DI.

DI and SI

During the execution of a string instruction, memory accesses occur through either or both of the DI and SI registers. The DI offset address accesses data in the extra segment for all string instructions that use it. The SI offset address accesses data, by default, in the data segment. The segment assignment of SI may be changed with a segment override prefix, as described later in this chapter. The DI segment assignment is always in the extra segment when a string instruction executes. This assignment cannot be changed. The reason that one pointer addresses data in the

TABLE 4–10 Forms of the LODS instruction.

Assembly Language	*Operation*
LODSB	AL = DS:[SI]; SI = SI ± 1
LODSW	AX = DS:[SI]; SI = SI ± 2
LODSD	EAX = DS:[SI]; SI = SI ± 4
LODS LIST	AL = DS:[SI]; SI = SI ± 1 (if LIST is a byte)
LODS DATA1	AX = DS:[SI], SI = SI ± 2 (if DATA1 is a word)

Note: The segment can be overridden with a segment override prefix as in LODS ES:DATA4.

extra segment and the other in the data segment is so the MOVS instruction can move 64K bytes of data from one segment of memory to another.

LODS

The LODS instruction loads AL, or AX, with data stored at the data segment offset address indexed by the SI register. After loading AL with a byte, AX with a word, or EAX with a doubleword, the contents of SI increment, if D = 0 or decrement, if D = 1. A 1 is added to or subtracted from SI for a byte-sized LODS, a 2 is added or subtracted for a word-sized LODS.

Table 4–10 lists the permissible forms of the LODS instruction. The LODSB **(loads a byte)** instruction causes a byte to be loaded into AL, the LODSW **(loads a word)** instruction causes a word to be loaded into AX. Although rare, as an alternative to LODSB, and LODSW, the LODS instruction may be followed by a byte-, word-sized operand to select a byte, or word transfer. Operands are often defined as bytes with DB and as words with DW. The DB pseudo-operation defines byte(s) and the DW pseudo-operation defines word(s).

Figure 4–16 shows the effect of executing the LODSW instruction if the D flag = 0, SI = 1000H, and DS = 1000H. Here, a 16-bit number stored at memory locations 11000H and 11001H moves into AX. Because D = 0 and this is a word transfer, the contents of SI increment by 2 after AX loads with memory data.

STOS

The STOS instruction stores AL, or AX at the extra segment memory location addressed by the DI register. Table 4–11 lists all forms of the STOS instruction. As with LODS, a STOS instruction may be appended with a B, or W for byte, or word transfers. The STOSB **(stores a byte)** instruction stores the byte in AL at the extra segment memory location addressed by DI. The STOSW **(stores a word)** instruction stores AX in the extra segment memory location addressed by DI. After the byte (AL), or word (AX) is stored, the contents of DI increments or decrements.

TABLE 4–11 Forms of the STOS instruction.

Assembly Language	*Operation*
STOSB	ES:[DI] = AL; DI = DI ± 1
STOSW	ES:[DI] = AX; DI = DI ± 2
STOSD	ES:[DI] = EAX; DI = DI ± 4
STOS LIST	ES:[DI] = AL; DI = DI ± 1 (if list is a byte)
STOS DATA3	ES:[DI] = AX; DI = DI ± 2 (if DATA3 is a word)

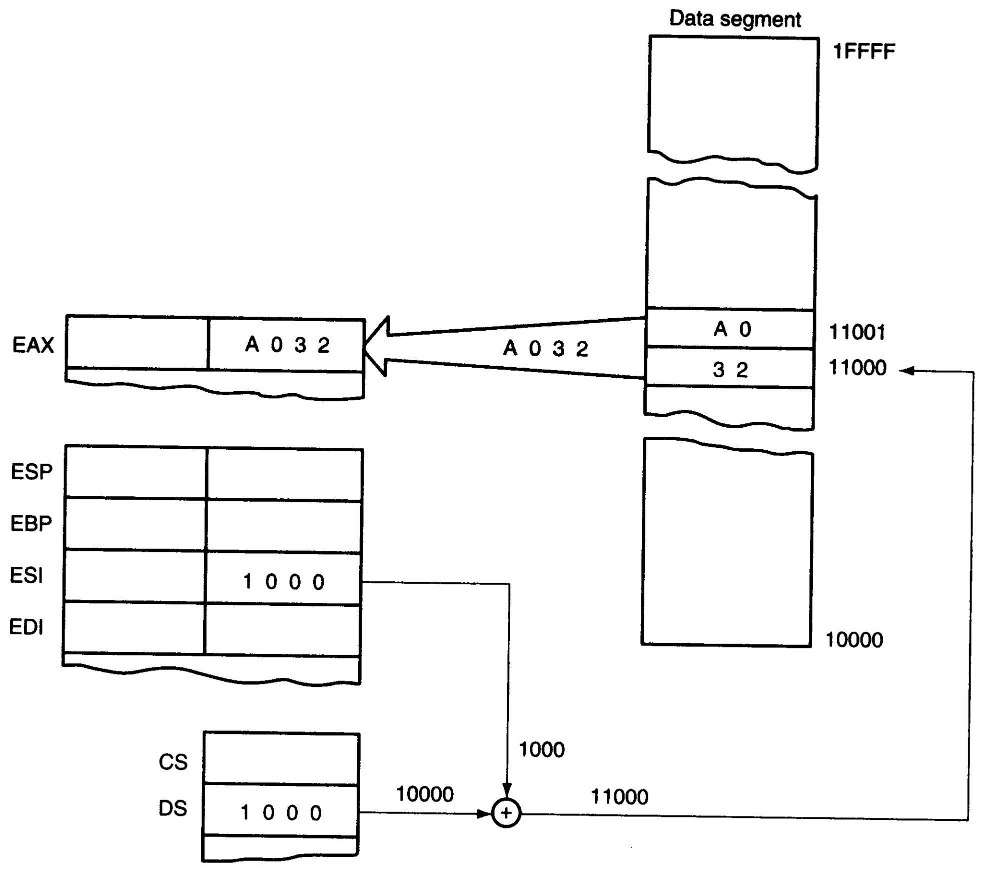


FIGURE 4–16 The operation of the LODSW instruction if DS = 1000H, D = 0, 11000H = 32, and 11001H = A0. This instruction is shown after AX is loaded from memory, but before SI increments by 2.

STOS with a REP. The **repeat prefix** (REP) is added to any string data transfer instruction, except the LODS instruction. It doesn't make any sense to perform a repeated LODS operation. The REP prefix causes CX to decrement by 1 each time the string instruction executes. After CX decrements, the string instruction repeats. If CX reaches a value of 0, the instruction terminates and the program continues with the next sequential instruction. Thus, if CX is loaded with a 100, and a REP STOSB instruction executes, the microprocessor automatically repeats the STOSB instruction 100 times. Because the DI register is automatically incremented or decremented after each datum is stored, this instruction stores the contents of AL in a block of memory instead of a single byte of memory.

Suppose that the STOSW instruction is used to clear the video text display (see Example 4–4). This is accomplished by addressing video text memory that begins at memory location B800:0000. Each character position on the 25-line-by-80-character per line display comprises two bytes. The first byte contains the ASCII-coded character, and the second contains the color and attributes of the character. In this example, AL is the ASCII-coded space (20H) and AH is the color code for white text on a black background (07H). Notice how this program uses a count of 25 * 80 and the REP STOSW instruction to clear the screen with ASCII spaces.

The operands in a program can be modified by using arithmetic or logic operators such as multiplication (*). Other operators appear in Table 4–12.

TABLE 4–12 Common operand operators.

Operator	*Example*	*Comment*
+	MOV AL,6+3	Copies 9 into AL
–	MOV AL,8–2	Copies 6 into AL
*	MOV AL,4*3	Copies 12 into AL
/	MOV AX,12/5	Copies 2 into AX (remainder is lost)
MOD	MOV AX, 12 MOD 7	Copies 5 into AX (quotient is lost)
AND	MOV AX,12 AND 4	Copies 4 into AX (1100 AND 0100 = 0100)
OR	MOV AX,12 OR 1	Copies 13 into AX (1100 OR 0001 = 1101)
NOT	MOV AL,NOT 1	Copies 254 into AL (0000 0001 NOT equals 1111 1110 or 254)

EXAMPLE 4–4

```
                    .MODEL TINY                     ;select TINY model
0000                .CODE                           ;start of CODE segment
                    .STARTUP                        ;start of program
0100  FC                    CLD                     ;select increment mode
0101  B8 B800               MOV     AX,0B800H       ;address segment B800
0104  8E C0                 MOV     ES,AX

0106  BF 0000               MOV     DI,0            ;address offset 0000
0109  B9 07D0               MOV     CX,25*80        ;load count
010C  B8 0720               MOV     AX,0720H        ;load data

010F  F3/AB                 REP     STOSW           ;clear the screen
                    .EXIT                           ;exit to DOS
                    END                             ;end of file
```

The REP prefix precedes the STOSW instruction in both assembly language and hexadecimal machine language. In machine language, the F3H is the REP prefix and ABH is the STOSW opcode.

If the value loaded to AX is changed to 0731H, the video display fills with white ones on a black background. If AX is changed to 0132H, the video display fills with blue twos on a black background. By changing the value loaded to AX, the display can be filled with any character and any color combination. More information on accessing the video display appears in a later chapter.

MOVS

One of the more useful string data transfer instructions is MOVS because it transfers data from one memory location to another. This is the only memory-to-memory transfer allowed in the 8086–Pentium 4 microprocessors. The MOVS instruction transfers a byte, or word from the data segment location addressed by SI to the extra segment location addressed by DI. As with the other string instructions, the pointers then increment or decrement, as dictated by the direction flag. Table 4–13 lists all the permissible forms of the MOVS instruction. Note that only the source operand (SI), located in the data segment, may be overridden so that another segment may be used. The destination operand (DI) must always be located in the extra segment.

Suppose that the video display needs to be scrolled up one line. Because we now know the location of the video display, a repeated MOVSW instruction can be used to scroll the video display up a line. Example 4–5 lists a short program that addresses the video text display, beginning at location B800:0000 with the DS:SI register combination and location B800:00A0 with the ES:DI register combination. Next, the REP MOVSW instruction is executed 24 * 80 times to scroll the display up a line. This is followed by a sequence that addresses the last line of the display so it can be cleared. The last line is cleared in this example by storing spaces on a black background. The last line could be cleared by changing only the ASCII code to a space, without modifying the attribute, by reading the

TABLE 4–13 Forms of the MOVS instruction.

Assembly Language	*Operation*
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVS BYTE1,BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2, SI = SI ± 2 (if WORD1 and WORD2 are words)

code and attribute into a register. Once in a register, the code is modified, and both the code and attribute are stored in memory.

EXAMPLE 4–5

```
                  .MODEL TINY                         ;select TINY model
0000              .CODE                               ;indicate start of CODE segment
                  .STARTUP                            ;indicate start of program
0100  FC                 CLD                          ;select increment
0101  B8 B800            MOV      AX,0B800H           ;load ES and DS with B800
0104  8E C0              MOV      ES,AX
0106  8E D8              MOV      DS,AX

0108  BE 00A0            MOV      SI,160              ;address line 1
010B  BF 0000            MOV      DI,0                ;address line 0
010E  B9 0780            MOV      CX,24*80            ;load count
0111  F3/A5              REP      MOVSW               ;scroll screen

0113  BF 0F00            MOV      DI,24*80*2          ;clear bottom line
0116  B9 0050            MOV      CX,80
0119  B8 0720            MOV      AX,0720H
011C  F3/AB              REP      STOSW
                  .EXIT                               ;exit to DOS
                  END                                 ;end of file
```

4–5 MISCELLANEOUS DATA TRANSFER INSTRUCTIONS

Don't be fooled by the term *miscellaneous;* these instructions are used in programs. The data transfer instructions detailed in this section are XCHG, LAHF, SAHF, XLAT, IN, and OUT. Because the miscellaneous instructions are not used as often as a MOV instruction, they have been grouped together and represented in this section.

XCHG

The XCHG (**exchange**) instruction exchanges the contents of a register with the contents of any other register or memory location. The XCHG instruction cannot exchange segment registers or memory-to-memory data. Exchanges are byte-, word-, or doubleword-sized (80386 and above), and use any addressing mode discussed in Chapter 3, except immediate addressing. Table 4–14 shows some examples of the XCHG instruction.

The XCHG instruction, using the 16-bit AX register with another 16-bit register, is the most efficient exchange. This instruction occupies one byte of memory. Other XCHG instructions require two or more bytes of memory, depending on the addressing mode selected.

TABLE 4–14 Forms of the XCHG instruction.

Assembly Language	*Operation*
XCHG AL, CL	Exchanges the contents of AL with CL
XCHG CX, BP	Exchanges the contents of CX with BP
XCHG AL, DATA2	Exchanges the contents of AL with data segment memory location DATA2

When using a memory addressing mode and the assembler, it doesn't matter which operand addresses memory. The XCHG AL, [DI] instruction is identical to the XCHG [DI], AL instruction, as far as the assembler is concerned.

LAHF and SAHF

The LAHF and SAHF instructions are seldom used because they were designed as bridge instructions. These instructions allowed 8085 (an early 8-bit microprocessor) software to be translated into 8086 software by a translation program. Because any software that required translation was probably completed many years ago, these instructions have little application today. The LAHF instruction transfers the rightmost eight bits of the flag register into the AH register. The SAHF instruction transfers the AH register into the rightmost eight bits of the flag register.

XLAT

The XLAT (**translate**) instruction converts the contents of the AL register into a number stored in a memory table. This instruction performs the direct table lookup technique often used to convert one code to another. An XLAT instruction first adds the contents of AL to BX to form a memory address within the data segment. It then copies the contents of this address into AL. This is the only instruction that adds an 8-bit number to a 16-bit number.

Suppose that a 7-segment LED display lookup table is stored in memory at address TABLE. The XLAT instruction then translates the BCD number in AL to a 7-segment code in AL. Example 4–6 provides a short program that converts from a BCD code to a 7-segment code. Figure 4–17 shows the operation of this example program if TABLE = 1000H, DS = 1000H, and the initial value of AL = 05H (a 5 BCD). After the translation, AL = 6DH.

EXAMPLE 4–6

```
                      ;Using an XLAT to convert from BCD to 7-segment code
                      ;
                      .MODEL SMALL                            ;select SMALL model
0000                  .DATA                                   ;start of DATA segment
0000  3F 06 5B 4F     TABLE  DB    3FH,6,5BH,4FH              ;7-segment lookup table
0004  66 6D 7D 27            DB    66H,6DH,7DH,27H
0008  7F 6F                  DB    7FH,6FH
000A  00              CODE7  DB    ?                          ;reserve for result
0000                  .CODE                                   ;start of CODE segment
                      .STARTUP                                ;start of program
0017  B0 04                  MOV   AL,4                       ;load test data
0019  BB 0000 R              MOV   BX,OFFSET TABLE            ;address lookup table
001C  D7                     XLAT                             ;convert to 7-segment
001D  A2 000A R              MOV   CODE7,AL                   ;save 7-segment code
                      .EXIT                                   ;exit to DOS
                             END                              ;end of file
```

IN and OUT

Table 4–15 lists the forms of the IN and OUT instructions, which perform I/O operations. Notice that the contents of AL, or AX are transferred only between the I/O device and the microprocessor. An IN instruction transfers data from an external I/O device to AL, or AX; an OUT transfers data from AL, or AX to an external I/O device.

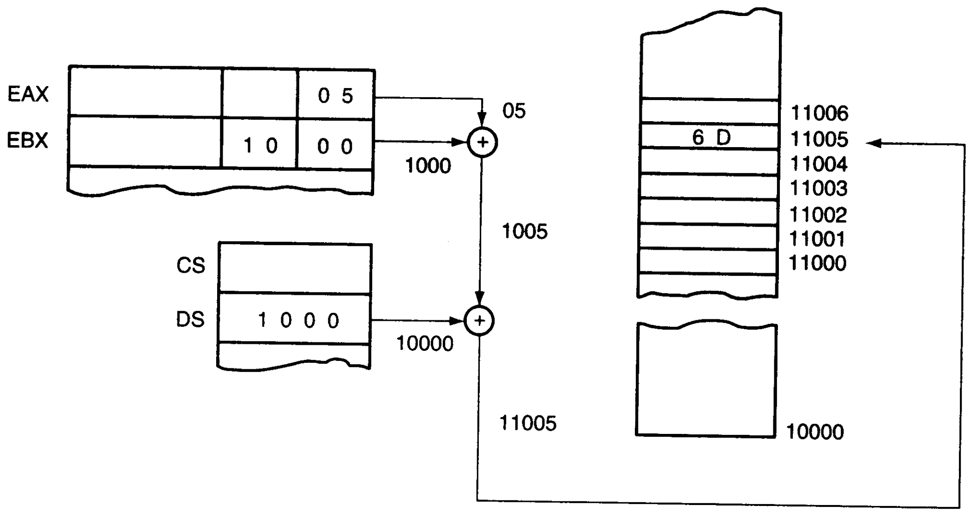


FIGURE 4–17 The operation of the XLAT instruction at the point just before 6DH is loaded into AL.

TABLE 4–15 IN and OUT instructions.

Assembly Language	*Operation*
IN AL,p8	8-bits are input to AL from I/O port p8
IN AX,p8	16-bits are input to AX from I/O port p8
IN AL,DX	8-bits are input to AL from I/O port DX
IN AX,DX	16-bits are input to AX from I/O port DX
OUT p8,AL	8-bits are output from AL to I/O port p8
OUT p8,AX	16-bits are output from AX to I/O port p8
OUT DX,AL	8-bits are output from AL to I/O port DX
OUT DX,AX	16-bits are output from AX to I/O port DX

Note: p8 = an 8-bit I/O port number and DX = the 16-bit port address held in DX.

Two forms of I/O device (port) addressing exist for IN and OUT: fixed-port and variable-port. *Fixed-port addressing* allows data transfer between AL or AX using an 8-bit I/O port address. It is called fixed-port addressing because the port number follows the instruction's opcode. Often, instructions are stored in a ROM. A fixed-port instruction stored in a ROM has its port number permanently fixed because of the nature of read-only memory. A fixed-port address stored in a RAM can be modified, but such a modification does not conform to good programming practices.

The port address appears on the address bus during an I/O operation. For the 8-bit fixed-port I/O instructions, the 8-bit port address is zero-extended into a 16-bit address. For example, if the IN AL,6AH instruction executes, data from I/O address 6AH are input to AL. The address appears as a 16-bit 006AH on pins A0–A15 of the address bus. Note that Intel reserves the last 16 I/O ports for use with some of its peripheral components.

Variable-port addressing allows data transfers between AL, AX, and a 16-bit port address. It is called *variable-port addressing* because the I/O port number is stored in register DX, which can be changed (varied) during the execution of a program. The 16-bit I/O port address appears on the address bus pin connections A0–A15. The IBM PC uses a 16-bit port address to access its I/O space. The I/O space for a PC is located at I/O port 0000H–03FFH. Note that some plug-in adapter cards may use I/O addresses above 03FFH.

Figure 4–18 illustrates the execution of the OUT 19H,AX instruction, which transfers the contents of AX to I/O port 19H. Notice that the I/O port number appears as a 0019H on the 16-bit address bus and that the data from AX appears on the data bus of the microprocessor. The system control signal IOWC (I/O write control) is a logic zero to enable the I/O device.

A short program that clicks the speaker in the personal computer appears in Example 4–7. The speaker is controlled by accessing I/O port 61H. If the rightmost two bits of this port are set (11) and then cleared (00), a click is heard on the speaker. Note that this program uses a logical OR instruction to set these two bits and a logical AND instruction to clear them. These logic operation instructions are described in Chapter 5. The MOV CX,4000H instruction, followed by the LOOP L1 instruction, is used as a time delay. If the count is increased, the click will become longer; if shortened, the click will become shorter.

EXAMPLE 4–7

```
                     .MODEL TINY                       ;select TINY model
0000                 .CODE                             ;indicate start of code segment
                     .STARTUP                          ;indicate start of program
0100  E4 61                 IN     AL,61H              ;read port 61H
0102  0C 03                 OR     AL,3                ;set rightmost two bits
0104  E6 61                 OUT    61H,AL              ;speaker is on

0106  B9 1000               MOV    CX,1000H            ;delay count
0109                 L1:
0109  E2 FE                 LOOP   L1                  ;time delay

010B  E4 61                 IN     AL,61H              ;read port 61H
010D  24 FC                 AND    AL,0FCH             ;clear rightmost two bits
010F  E6 61                 OUT    61H,AL              ;speaker is off
                     .EXIT                             ;exit to DOS
                     END                               ;end of file
```

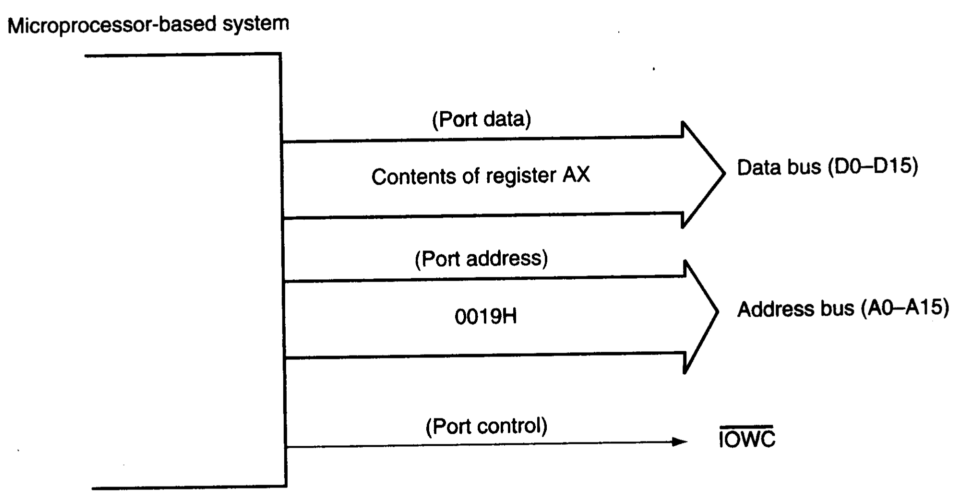


FIGURE 4–18 The signals found in the microprocessor-based system for an OUT 19H, AX instruction.

TABLE 4–16 Instructions that include segment override prefixes.

Assembly Language	*Segment Accessed*	*Default Segment*
MOV AX,DS:[BP]	Data	Stack
MOV AX,ES:[BP]	Extra	Stack
MOV AX,SS:[DI]	Stack	Data
MOV AX,CS:LIST	Code	Data
MOV AX,ES:[SI]	Extra	Data
LODS ES:DATA1	Data	Extra

4–6 SEGMENT OVERRIDE PREFIX

The **segment override prefix,** which may be added to almost any instruction in any memory addressing mode, allows the programmer to deviate from the default segment. The segment override prefix is an additional byte that appends the front of an instruction to select an alternate segment register. About the only instructions that cannot be prefixed are the jump and call instructions that must use the code segment register for address generation.

For example, the MOV AX, [DI] instruction accesses data within the data segment by default. If required by a program, this can be changed by prefixing the instruction. Suppose that the data are in the extra segment instead of in the data segment. This instruction addresses the extra segment if changed to MOV AX, ES:[DI].

Table 4–16 shows some altered instructions that address different memory segments that are different from normal. Each time an instruction is prefixed with a segment override prefix, the instruction becomes one byte longer. Although this is not a serious change to the length of the instruction, it does add to the instruction's execution time. It is usually customary to limit the use of the segment override prefix and remain in the default segments so that shorter and more efficient software can be written.

4–7 ASSEMBLER DETAIL

The assembler[1] for the microprocessor can be used in two ways: (1) with models that are unique to a particular assembler, and (2) with full segment definitions that allow complete control over the assembly process and are universal to all assemblers. This section of the text presents both methods, and explains how to organize a program's memory space by using the assembler. It also explains the purpose and use of some of the more important directives used with this assembler. Appendix A provides additional detail about the assembler.

Directives

Before the format of an assembly language program is discussed, some details about the directives (**pseudo-operations**) that control the assembly process must be learned. Some common assembly language directives appear in Table 4–17. **Directives** indicate how an operand or section of a program is to be processed by the assembler. Some directives generate and store information in the memory, while others do not. The DB (**define byte**) directive stores bytes of data in the memory, while the BYTE PTR directive never stores data. The **BYTE PTR** directive indicates the size of the data referenced by a pointer or index register.

Note that the assembler by default accepts only 8086/8088 instructions, unless a program is preceded by the .386 or .386P directive or one of the other microprocessor selection switches. The .386 directive tells the assembler to use the 80386 instruction set in the real mode, while the .386P directive tells the assembler to use the 80386

[1]The assembler used throughout this text is the Microsoft MACRO assembler MASM, version 6.X

TABLE 4–17 Common assembler directives.

Directive	*Function*
.286	Selects the 80286 instruction set; default is 8086
.386	Selects the 80386 instruction set
.486	Selects the 80486 instruction set
.586	Selects the Pentium instruction set
.287	Selects the 80287 math coprocessor
.387	Selects the 80387 math coprocessor
.EXIT	Exits to DOS
.MODEL	Selects the programming model
.STARTUP	Indicates the start of the program when using program models
ASSUME	Informs the assembler of the name of each segment for full segment definitions
BYTE	Indicates byte-sized, as in BYTE PTR
DB	Defines byte(s) (8-bits)
DD	Defines doubleword(s) (32-bits)
DQ	Defines quadword(s) (64-bits)
DT	Defines ten byte(s) (80-bits)
DUP	Generates duplicates
DW	Defines word(s) (16-bits)
DWORD	Indicates doubleword-sized, as in DWORD PTR
END	Ends a program file
ENDM	Ends a macro sequence
ENDP	Ends a procedure
ENDS	Ends a segment or data structure
EQU	Equates data to a label
FAR	Defines a far pointer
MACRO	Designates the start of a macro sequence
NEAR	Defines a near pointer
OFFSET	Specifies an offset address
ORG	Sets the origin within a segment
PROC	Starts a procedure
PTR	Designates a pointer
SEGMENT	Starts a segment
STACK	Starts a stack segment
STRUC	Defines the start of a data structure
WORD	Indicates word-sized, as in WORD PTR

protected mode instruction set. Most software is written assuming that the microprocessor is an 80386 or newer, so the .386 switch is often used. Windows 95 was the first major operating system to use a 32-bit architecture that conforms to the 80386.

Storing Data in a Memory Segment. The DB **(define byte)**, DW **(define word)**, and DD **(define doubleword)** directives, first presented in Chapter 1, are most often used with the microprocessor to define and store memory data. If a numeric coprocessor executes software in the system, the DQ **(define quadword)** and DT **(define ten bytes)** directives are also common. These directives label a memory location with a symbolic name and indicate its size.

Example 4–8 shows a memory segment that contains various forms of data definition directives. It also shows the full segment definition with the first SEGMENT statement to indicate the start of the segment and its symbolic name. Alternately, as in past examples in this and prior chapters, the SMALL model can be used with the .DATA statement. The last statement in this example contains the ENDS directive, which indicates the end of the segment. The name of the segment (LIST_SEG) can be anything that the programmer desires to call it. This allows a program to contain as many segments as required.

EXAMPLE 4–8

```
                                ;Using the DB, DW, and DD directives
                                ;
0000                            LIST_SEG     SEGMENT

0000  01 02 03                  DATA1   DB     1,2,3              ;define bytes
0003  45                                DB     45H                ;hexadecimal
0004  41                                DB     'A'                ;ASCII
0005  F0                                DB     11110000B          ;binary
0006  000C 000D                 DATA2   DW     12,13              ;define words
000A  0200                              DW     LIST1              ;symbolic
000C  2345                              DW     2345H              ;hexadecimal
000E  00000300                  DATA3   DD     300H               ;hexadecimal
0012  4007DF3B                          DD     2.123              ;real
0016  544269E1                          DD     3.34E+12           ;real
001A  00                        LISTA   DB     ?                  ;reserve 1 byte
001B  000A[                     LISTB   DB     10 DUP (?)         ;reserve 10 bytes
                           ??
                                    ]
0025  00                                ALIGN  2                  ;set word boundary

0026  0100[                     LISTC   DW     100H DUP (0)       ;word array
                           0000
                                      ]
0226  0016[                     LIST_9  DD     22 DUP (?)         ;doubleword array
                           ????????
                                        ]
027E  0064[                     SIXES   DB     100 DUP (6)        ;byte array
                           06
                                    ]
02E2                            LIST_SEG ENDS
```

Example 4–8 shows various forms of data storage for bytes at DATA1. More than one byte can be defined on a line in binary, hexadecimal, decimal, or ASCII code. The DATA2 label shows how to store various forms of word data. Doublewords are stored at DATA3; they include floating-point, single-precision real numbers.

Memory is **reserved** for use in the future by using a ? as an operand for a DB, DW, or DD directive. When a ? is used in place of a numeric or ASCII value, the assembler sets aside a location and does not initialize it to any specific value. (Actually, the assembler usually stores a zero into locations specified with a ?). The DUP **(duplicate)** directive creates an array, as shown in several ways in Example 4–8. A 10 DUP (?) reserves 10 locations of memory, but stores no specific value in any of the 10 locations. If a number appears within the ( ) part of the DUP statement, the assembler initializes the reserved section of memory with the data indicated. For example, the DATA1 DB 10 DUP (2) instruction reserves 10 bytes of memory for array DATA1 and initializes each location with a 02H.

The ALIGN directive, used in this example, makes sure that the memory arrays are stored on word boundaries. It is important that word-sized data are placed at word boundaries and doubleword-sized data are placed at doubleword boundaries. If not, the microprocessor spends additional time accessing these data types. A word stored at an odd-numbered memory location takes twice as long to access as a word stored on an even-numbered memory location. Note that the ALIGN directive cannot be used with memory models because the size of the model determines the data alignment. If all doubleword data are defined first, followed by word and then byte-sized data, the ALIGN statement is not necessary to align data correctly.

ASSUME, EQU, and ORG. The equate directive (EQU) equates a numeric, ASCII, or label to another label. Equates make a program clearer and simplify debugging. Example 4–9 shows several equate statements and a few instructions that show how they function in a program.

EXAMPLE 4–9

```
     ;Using equate directive
     ;
= 000A TEN EQU 10
= 0009 NINE EQU 9

0000 B0 0A MOV AL,TEN
0002 04 09 ADD AL,NINE
```

The THIS directive always appears as THIS BYTE, THIS WORD, or THIS DWORD. In certain cases, data must be referred to as both a byte and a word. The assembler can only assign either a byte or a word address to a label. To assign a byte label to a word, use the software listed in Example 4–10.

EXAMPLE 4–10

```
                        ;Using the THIS and ORG directives
                        ;
0000                    DATA_SEG     SEGMENT

0100                          ORG    100H

= 0100                  DATA1  EQU   THIS BYTE
0100  0000              DATA2  DW    ?

0102                    DATA_SEG     ENDS

0000                    CODE_SEG     SEGMENT 'CODE'

                              ASSUME CS:CODE_SEG,DS:DATA_SEG

0000  8A 1E 0100 R            MOV BL,DATA1
0004  A1 0100 R               MOV AX,DATA2
0007  8A 3E 0101 R            MOV BH,DATA1+1

000B                    CODE_SEG ENDS
```

This example also illustrates how the ORG **(origin)** statement changes the starting offset address of the data in the data segment to location 100H. At times, the origin of data or the code must be assigned to an absolute offset address with the ORG statement. The **ASSUME statement** tells the assembler what names have been chosen for the code, data, extra, and stack segments. Without the ASSUME statement, the assembler assumes nothing and automatically uses a segment override prefix on all instructions that address memory data. The ASSUME statement is only used with full-segment definitions, as described later in this section of the text.

PROC and ENDP. The PROC and ENDP directives indicate the start and end of a procedure **(subroutine).** These directives *force structure* because the procedure is clearly defined. Note that if structure is to be violated for whatever reason, use the CALLF, CALLN, RETF, and RETN instructions. Both the PROC and ENDP directives require a label to indicate the name of the procedure. The PROC directive, which indicates the start of a procedure, must also be followed with a NEAR or FAR. A NEAR procedure is one that resides in the same code segment as the program. A FAR procedure may reside at any location in the memory system. Often the call NEAR procedure is considered to be *local,* and the call FAR procedure is considered to be *global.* The term global denotes a procedure that can be used by any program, while local defines a procedure that is only used by the current program. Any labels that are defined within the procedure block are also defined as either local (NEAR) or global (FAR).

Example 4–12 shows a procedure that adds BX, CX, and DX and stores the sum in register AX. Although this procedure is short and may not be particularly useful, it does illustrate how to use the PROC and ENDP directives to delineate the procedure. Note that information about the operation of the procedure should appear as a grouping of comments that show the registers changed by the procedure and the result of the procedure.

EXAMPLE 4–11

```
                      ;A procedure that adds BX, CX, and DX with the sum
                      ;stored in AX
                      ;
0000                  ADDEM    PROC    FAR                 ;start procedure

0000   03 D9                   ADD     BX,CX
0002   03 DA                   ADD     BX,DX
0004   8B C3                   MOV     AX,BX
0006   CB                      RET

0007                  ADDEM ENDP                           ;end procedure
```

Memory Organization

The assembler uses two basic formats for developing software: one method uses models and the other uses full-segment definitions. Memory models, as presented in this section and briefly in Chapters 2 and 3, are unique to the MASM assembler program. The TASM assembler also uses memory models, but they differ somewhat from the MASM models. The full-segment definitions are common to most assemblers, including the Intel assembler, and are often used for software development. The models are easier to use for simple tasks. The full-segment definitions offer better control over the assembly language task and are recommended for complex programs. The model was used in early chapters because it is easier to understand for the beginning programmer. Models are also used with assembly language procedures that are used by high-level languages such as C/C++. Although this text fully develops and uses the memory model definitions for its programming examples, realize that full-segment definitions offer some advantages over memory models, as discussed later in this section.

Models. There are many models available to the MASM assembler, ranging from tiny to huge. Appendix A contains a table that lists all the models available for use with the assembler. To designate a model, use the .MODEL statement followed by the size of the memory system. The **TINY model** requires that all software and data fit into one 64K-byte memory segment; it is useful for many small programs. The **SMALL model** requires that only one data segment be used with one code segment, for a total of 128K bytes of memory. Other models are available, up to the HUGE model.

Example 4–12 illustrates how the .MODEL statement defines the parameters of a short program that copies the contents of a 100-byte block of memory (LISTA) into a second 100-byte block of memory (LISTB). It also shows how to define the stack, data, and code segments. The .EXIT 0 directive returns to DOS with an error code of 0 (no error). If no parameter is added to .EXIT, it still returns to DOS, but the error code is not defined. Also note that special directives such as @DATA (see Appendix A) are used to identify various segments. If the .STARTUP directive is used (MASM version 6.X), the MOV AX,@DATA followed by MOV DS,AX statements can be eliminated. The .STARTUP directive also eliminates the need to store the starting address next to the END label. Models are important with both Microsoft C/C++ and Borland C/C++ development systems if assembly language is included with C/C++ programs. Both development systems use in-line assembly programming for adding assembly language instructions and require an understanding of programming models. Refer to the respective C/C++ language reference for each system to determine the model protocols.

EXAMPLE 4–12

```
                          .MODEL SMALL
                          .STACK 100H                          ;define stack
                          .DATA                                ;define data segment

0000  0064[               LISTA    DB      100 DUP (?)
                   ??
                            ]
0064  0064[               LISTB    DB      100 DUP (?)
                   ??
                            ]

                          .CODE                                ;define code segment

0000  B8 ---- R           HERE:   MOV     AX,@DATA             ;load ES, DS
0003  8E C0                       MOV     ES,AX
0005  8E D8                       MOV     DS,AX

0007  FC                          CLD                          ;move data
0008  BE 0000 R                   MOV     SI,OFFSET LISTA
000B  BF 0064 R                   MOV     DI,OFFSET LISTB
000E  B9 0064                     MOV     CX,100
0011  F3/A4                       REP     MOVSB

0013                      .EXIT 0                              ;exit to DOS
                          END HERE
```

Full-segment Definitions. Example 4–13 illustrates the same program, using full segment definitions. Full-segment definitions are also used with the Borland and Microsoft C/C++ environments for procedures developed in assembly language. The program in Example 4–13 appears longer than the one pictured in Example 4–12, but it is more structured than the model method of setting up a program. The first segment defined is the STACK_SEG that is clearly delineated with the SEGMENT and ENDS directives. Within these directives, a DW 100 DUP (?) sets aside 100H words for the stack segment. Because the word STACK appears next to SEGMENT, the assembler and linker automatically load both the stack segment register (SS) and stack pointer (SP).

EXAMPLE 4–13

```
0000                      STACK_SEG   SEGMENT STACK

0000  0100[                     DW    100H DUP (?)
                   ????
                      ]

0200                      STACK_SEG   ENDS

0000                      DATA_SEG   SEGMENT 'DATA'

0000  0064[               LISTA   DB     100 DUP (?)
                   ??
                            ]
0064  0064[               LISTB   DB     100 DUP (?)
                   ??
                            ]

00C8                      DATA_SEG   ENDS

0000                      CODE_SEG   SEGMENT 'CODE'

                                  ASSUME CS:CODE_SEG,DS:DATA_SEG
                                  ASSUME SS:STACK_SEG
```

```
0000                    MAIN    PROC    FAR

0000  B8 ---- R                 MOV     AX,DATA_SEG       ;load DS and ES
0003  8E C0                     MOV     ES,AX
0005  8E D8                     MOV     DS,AX

0007  FC                        CLD                       ;move data
0008  BE 0000 R                 MOV     SI,OFFSET LISTA
000B  BF 0064 R                 MOV     DI,OFFSET LISTB
000E  B9 0064                   MOV     CX,100
0011 F3/A4                     REP MOVSB

0013  B4 4C                     MOV     AH,4CH            ;exit to DOS
0015  CD 21                     INT     21H

0017                    MAIN    ENDP

0017                    CODE_SEG   ENDS

                                END     MAIN
```

Next, the data are defined in the DATA_SEG. Here, two arrays of data appear as LISTA and LISTB. Each array contains 100 bytes of space for the program. The names of the segments in this program can be changed to any name. Always include the group name 'DATA', so that the Microsoft program CodeView can be effectively used to symbolically debug this software. CodeView is a part of the MASM package used to debug software. To access CodeView, type CV, followed by the file name at the DOS command line; if operating from Programmer's WorkBench, select Debug under the Run menu. If the group name is not placed in a program, CodeView can still be used to debug a program, but the program will not be debugged in symbolic form. Other group names such as 'STACK', 'CODE', and so forth are listed in Appendix A. You must at least place the word 'CODE' next to the code segment SEGMENT statement if you want to view the program symbolically in CodeView.

The CODE_SEG is organized as a far procedure because most software is procedure oriented. Before the program begins, the code segment contains the ASSUME statement. The ASSUME statement tells the assembler and linker that the name used for the code segment (CS) is CODE_SEG; it also tells the assembler and linker that the data segment is DATA_SEG and the stack segment is STACK_SEG. Notice that the group name 'CODE' is used for the code segment for use by CodeView. Other group names appear in Appendix A with the models.

After the program loads both the extra segment register and data segment register with the location of the data segment, it transfers 100 bytes from LISTA to LISTB. Following this is a sequence of two instructions that return control back to DOS (the disk operating system). Note that the program loader does not automatically initialize DS and ES. These registers must be loaded with the desired segment addresses in the program.

The last statement in the program is END MAIN. The END statement indicates the end of the program and the location of the first instruction executed. Here, we want the machine to execute the main procedure so that a label follows the END directive.

A Sample Program

Example 4–14 provides a sample program, using full-segment definitions, that reads a character from the keyboard and displays it on the CRT screen. Although this program is trivial, it illustrates a complete workable program that functions on any personal computer using DOS, from the earliest 8088-based system to the latest Pentium 4-based system. This program also illustrates the use of a few DOS function calls. (Appendix A lists the DOS function calls with their parameters.) The BIOS function calls allow the use of the keyboard, printer, disk drives, and everything else that is available in your computer system.

This example program uses only a code segment because there is no data. A stack segment should appear, but it has been left out because DOS automatically allocates a 128-byte stack for all programs. The only time that the stack is used in this example is for the INT 21H instructions that call a procedure in DOS. Note that when this program is linked, the linker signals that no stack segment is present. This warning may be ignored in this example because the stack is fewer than 128 bytes.

Notice that the entire program is placed into a far procedure called MAIN. It is good programming practice to write all software in procedural form, which allows the program to be used as a procedure at some future time if necessary. It is also fairly important to document register use and any parameters required for the program in the program header, which is a section of comments that appear at the start of the program.

The program uses DOS functions 06H and 4CH. The function number is placed in AH before the INT 21H instruction executes. The 06H function reads the keyboard if DL = 0FFH, or displays the ASCII contents of DL if it is not 0FFH. Upon close examination, the first section of the program moves a 06H into AH and a 0FFH into DL, so that a key is read from the keyboard. The INT 21H tests the keyboard; if no key is typed, it returns equal. The JE instruction tests the equal condition and jumps to MAIN if no key is typed.

When a key is typed, the program continues to the next step, which compares the contents of AL with an @ symbol. Upon return from the INT 21H, the ASCII character of the typed key is found in AL. In this program, if an @ symbol is typed, the program ends. If the @ symbol is not typed, the program continues by displaying the character typed on the keyboard with the next INT 21H instruction.

The second INT 21H instruction moves the ASCII character into DL so it can be displayed on the CRT screen. After displaying the character, a JMP executes. This causes the program to continue at MAIN, where it repeats reading a key.

If the @ symbol is typed, the program continues at MAIN1, where it executes the DOS function code number 4CH. This causes the program to return to the DOS prompt (A>), so that the computer can be used for other tasks.

More information about the assembler and its application appears in Appendix A and in the next several chapters. Appendix A provides a complete overview of the assembler, linker, and DOS functions. It also provides a list of the BIOS (basic I/O system) functions. The information provided in the following chapters clarifies how to use the assembler for certain tasks at different levels of the text.

EXAMPLE 4–14

```
            ;An example program that reads a key and displays it.
            ;Note that an @ key ends the program.
            ;
0000        CODE_SEG   SEGMENT 'CODE'

                     ASSUME CS:CODE_SEG

0000        MAIN     PROC   FAR

0000 B4 06           MOV    AH,6            ;read key
0002 B2 FF           MOV    DL,0FFH
0004 CD 21           INT    21H
0006 74 F8           JE     MAIN            ;if no key

0008 3C 40           CMP    AL,'@'          ;test for @
000A 74 08           JE     MAIN1           ;if @

000C B4 06           MOV    AH,6            ;display key
000E 8A D0           MOV    DL,AL
0010 CD 21           INT    21H
0012 EB EC           JMP    MAIN            ;repeat
0014        MAIN1:
0014 B4 4C           MOV    AH,4CH          ;exit to DOS
0016 CD 21           INT    21H

0018        MAIN     ENDP

0018        CODE_SEG    ENDS

                     END    MAIN
```

Example 4–15 shows the program listed in Example 4–16, except models are used instead of full-segment ssdescriptions. Please compare the two programs to determine the differences. Notice how much shorter the models can make a program.

EXAMPLE 4–15

```
                    ;An example program that reads a key and displays it.
                    ;Note that an @ key ends the program.
                    ;
                    .MODEL TINY
0000                .CODE
                    .STARTUP

0100                MAIN:
0100 B4 06                MOV AH,6      ;read key
0102 B2 FF                MOV DL,0FFH
0104 CD 21                INT 21H
0106 74 F8                JE  MAIN      ;if no key

0108 3C 40                CMP AL, '@'   ;text for @
010A 74 08                JE  MAIN1     ;if no @

010C B4 06                MOV AH,6      ;display key
010E 8A D0                MOV DL,AL
0110 CD 21                INT 21 H
0112 EB EC                JMP MAIN      ;repeat

0114                MAIN1:
                    .EXIT               ;exit to DOS
                    END
```

4–8 SUMMARY

1. Data movement instructions transfer data between registers, a register and memory, a register and the stack, memory and the stack, the accumulator and I/O, and the flags and the stack. Memory-to-memory transfers are allowed only with the MOVS instruction.
2. Data movement instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LAHF, SAHF; and the following string instructions: LODS, STOS, and MOVS.
3. The first byte of an instruction contains the opcode. The opcode specifies the operation performed by the microprocessor. The opcode may be preceded by one or more override prefixes in some forms of instructions.
4. The D bit, located in many instructions, selects the direction of data flow. If D = 0, the data flow from the REG field to the R/M field of the instruction. If D = 1, the data flow from the R/M field to the REG field.
5. The W bit, found in most instructions, selects the size of the data transfer. If W = 0, the data are byte-sized; if W = 1, the data are word-sized. In the 80386 and above, W = 1 specifies either a word or doubleword register.
6. MOD selects the addressing mode of operation for a machine language instruction's R/M field. If MOD = 00, there is no displacement; if MOD-01, an 8-bit sign-extended displacement appears; if MOD-10, a 16-bit displacement occurs; and if MOD-11, a register is used instead of a memory location. In the 80386 and above, the MOD bits also specify a 32-bit displacement.
7. A 3-bit binary register code specifies the REG and R/M fields when the MOD = 11. The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL. The 16-bit registers are AX, BX, CX, DX, SP, BP, DI, and SI.
8. When the R/M field depicts a memory mode, a 3-bit code selects one of the following modes: [BX+DI], [BX+SI], [BP+DI], [BP+SI], [BX], [BP], [DI], or [SI] for 16-bit instructions.

9. All memory-addressing modes, by default, address data in the data segment unless BP addresses memory. The BP register addresses data in the stack segment.
10. The segment registers are addressed only by the MOV, PUSH, or POP instructions. The MOV instruction may transfer a segment register to a 16-bit register, or vice versa. MOV CS,reg or POP CS instructions are not allowed because they change only part of the address. The 80386 through the Pentium 4 include two additional segment registers, FS and GS .
11. Data are transferred between a register or a memory location and the stack by the PUSH and POP instructions. Variations of these instructions allow immediate data to be pushed onto the stack, the flags to be transferred between the stack, and all 16-bit registers can be transferred between the stack and the registers. When data are transferred to the stack, two bytes (8086–80286) always move. The most-significant byte is placed at the location addressed by SP – 1, and the least-significant byte is placed at the location addressed by SP – 2. After placing the data on the stack, SP decrements by 2. In the 80386–Pentium 4, four bytes of data from a memory location or register may also be transferred to the stack.
12. Opcodes that transfer data between the stack and the flags are PUSHF and POPF. Opcodes that transfer all the 16-bit registers between the stack and the registers are PUSHA and POPA.
13. LEA, LDS, and LES instructions load a register or registers with an effective address. The LEA instruction loads any 16-bit register with an effective address; LDS and LES load any 16-bit register, and either DS or ES, with the effective address.
14. String data transfer instructions use either or both DI and SI to address memory. The DI offset address is located in the extra segment, and the SI offset address is located in the data segment.
15. The direction flag (D) chooses the auto-increment or auto-decrement mode of operation for DI and SI for string instructions. To clear D to 0, use the CLD instruction to select the auto-increment mode; to set D to 1, use the STD instruction to select the auto-decrement mode. Either or both DI and SI increment/decrement by 1 for a byte operation, by 2 for a word operation, and by 4 for a doubleword operation.
16. LODS loads AL or AX with data from the memory location addressed by SI; STOS stores AL or AX in the memory location addressed by DI; and MOVS transfers a byte or a word from the memory location addressed by SI into the location addressed by DI.
17. The REP prefix may be attached to any string instruction to repeat it. The REP prefix repeats the string instruction the number of times found in register CX.
18. Arithmetic and logic operators can be used in assembly language. An example is MOV AX, 34*3, which loads AX with 102.
19. Translate (XLAT) converts the data in AL into a number stored at the memory location address by BX plus AL.
20. IN and OUT transfer data between AL or AX and an external I/O device. The address of the I/O device is either stored with the instruction (fixed port) or in register DX (variable port).
21. The segment override prefix selects a different segment register for a memory location than the default segment. For example, the MOV AX,[BX] instruction uses the data segment, but the MOV AX,ES:[BX] instruction uses the extra segment because of the ES: prefix.
22. Assembler directives DB, (define byte), DW (define word), DD (define doubleword), and DUP (duplicate) store data in the memory system.
23. The EQU (equate) directive allows data or labels to be equated to labels.
24. The SEGMENT directive identifies the start of a memory segment and ENDS identifies the end of a segment when full-segment definitions are in use.
25. The ASSUME directive tells the assembler what segment names you have assigned to CS, DS, ES, and SS when full-segment definitions are in effect.
26. The PROC and ENDP directives indicate the start and end of a procedure.

27. The assembler assumes that software is being developed for the 8086/8088 microprocessors unless the .286, .386, .486, or .586 directive is used to select one of these other microprocessors.
28. Memory models can be used to shorten the program slightly, but they can cause problems for larger programs. Also be aware that memory models are not compatible with all assembler programs.

4–9 QUESTIONS AND PROBLEMS

1. The first byte of an instruction is the ___________, unless it contains one of the override prefixes.
2. Describe the purpose of the D- and W-bits found in some machine language instructions.
3. In a machine language instruction, what information is specified by the MOD field?
4. If the register field (REG) of an instruction contains a 010 and W = 0, what register is selected, assuming that the instruction is a 16-bit mode instruction?
5. What memory-addressing mode is specified by R/M = 001 with MOD = 00 for a 16-bit instruction?
6. Identify the default segment registers assigned to the following:
   (a) SP
   (b) BX
   (c) DI
   (d) BP
   (e) SI
7. Convert an 8B07H from machine language to assembly language.
8. Convert an 8B1E004CH from machine language to assembly language.
9. If a MOV SI, [BX+2] instruction appears in a program, what is its machine language equivalent?
10. What is wrong with a MOV CS, AX instruction?
11. Form a short sequence of instructions that load the data segment register with a 1000H.
12. The PUSH and POP instructions always transfer a(n) ____-bit number between the stack and a register or memory location in the 8086–microprocessors.
13. What segment register may not be popped from the stack?
14. Which registers move onto the stack with the PUSHA instruction?
15. Describe the operation of each of the following instructions:
    (a) PUSH AX
    (b) POP SI
    (c) PUSH [BX]
    (d) PUSHF
    (e) POP DS
    (f) PUSH 4
16. Explain what happens when the PUSH BX instruction executes. Make sure to show where BH and BL are stored. (Assume that SP = 0100H and SS = 0200H.)
17. Repeat question 16 for the PUSH AX instruction.
18. The 16-bit POP instruction (except for POPA) increments SP by _______.
19. What values appear in SP and SS if the stack is addressed at memory location 02200H?
20. Compare the operation of a MOV DI, NUMB instruction with an LEA DI, NUMB instruction.
21. What is the difference between an LEA SI, NUMB instruction and a MOV SI, OFFSET NUMB instruction?
22. Which is more efficient, a MOV with an OFFSET or an LEA instruction?
23. Describe how the LDS BX, NUMB instruction operates.

24. Develop a sequence of instructions that move the contents of data segment memory locations NUMB and NUMB+1 into BX, DX, and SI.
25. What is the purpose of the direction flag?
26. Which instructions set and clear the direction flag?
27. The string instructions use DI and SI to address memory data in which memory segments?
28. Explain the operation of the LODSB instruction.
29. Explain the operation of the STOSW instruction.
30. Explain the operation of the OUTSB instruction.
31. What does the REP prefix accomplish and what type of instruction is it used with?
32. Develop a sequence of instructions that copy 12 bytes of data from an area of memory addressed by SOURCE into an area of memory addressed by DEST.
33. Would the LAHF and SAHF instructions normally appear in software?
34. Explain how the XLAT instruction transforms the contents of the AL register.
35. Write a short program that uses the XLAT instruction to convert the BCD numbers 0–9 into ASCII-coded numbers 30H–39H. Store the ASCII-coded data in a TABLE located within the data segment.
36. Explain what the IN AL,12H instruction accomplishes.
37. Explain how the OUT DX, AX instruction operates.
38. What is a segment override prefix?
39. Select an instruction that moves a byte of data from the memory location addressed by the BX register, in the extra segment, into the AH register.
40. Develop a sequence of instructions that exchange the contents of AX with BX, CX with DX, and SI with DI.
41. What is an assembly language directive?
42. Describe the purpose of the following assembly language directives: DB, DW, and DD.
43. Select an assembly language directive that reserves 30 bytes of memory for array LIST1.
44. Describe the purpose of the EQU directive.
45. What is the purpose of the .386 directive?
46. What is the purpose of the .MODEL directive?
47. If the start of a segment is identified with .DATA, what type of memory organization is in effect?
48. If the SEGMENT directive identifies the start of a segment, what type of memory organization is in effect?
49. What does the INT 21H accomplish if AH contains a 4CH?
50. What directives indicate the start and end of a procedure?
51. Develop a near procedure that stores AL in four consecutive memory locations, within the data segment, as addressed by the DI register.
52. Develop a far procedure that copies contents of the word-sized memory location CS:DATA1 into AX, BX, CX, DX, and SI.

CHAPTER 5

Arithmetic and Logic Instructions

INTRODUCTION

In this chapter, the arithmetic and logic instructions are examined. 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). 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 tasks perform 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 Pentium 4 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 Pentium 4 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.

TABLE 5–1 Addition instructions.

Assembly Language	*Operation*
ADD AL, BL	AL = AL + BL
ADD CX, DI	CX = CX + DI
ADD CL, 44H	CL = CL + 44H
ADD BX, 245FH	BX = BX + 245FH
ADD [BX], AL	AL adds to the 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 BX, [S1+2]	The word contents of the data segment memory location addressed by the sum of S1 plus 2 add to BX with the sum stored in BX
ADD CL, TEMP	The byte contents of the 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 plus DI add to BX with the sum stored in BX
ADD [BX + DI], DL	DL adds to the contents of the data segment memory location addressed by BX plus 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

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 a one to a number. In Section 5–3, other forms of addition are examined, such as BCD and ASCII.

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 over 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.

Register Addition. Example 5–1 shows a simple procedure 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. Here, a procedure is used because assembly language is procedure-oriented, as are most languages.

EXAMPLE 5–1

```
                      ;A procedure that sums AX, BX, CD, and DX;
                      ;the result is returned in AX.
                      ;
0000                  ADDS    PROC    NEAR

0000  03 C3                   ADD     AX,BX
0002  03 C1                   ADD     AX,CX
0004  03 C2                   ADD     AX,DX
0006  C3                      RET

0007                  ADDS    ENDP
```

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 eight 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, load DL is first loaded with a 12H by using an immediate move instruction. Next, a 33H is added to the 12H in DL by using 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

```
0006  B2 12      MOV   DL,12H
0008  80 C2 33   ADD   DL,33H
```

Memory-to-Register Addition. Suppose that an application requires that memory data are 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

```
                   ;A procedure that sums data in locations NUMB and NUMB+1;
                   ;the result is returned in AX.
                   ;
0000               SUMS   PROC   NEAR

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
000A  C3                  RET

000B               SUMS   ENDP
```

Procedure SUMS first 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. In most cases, loading the address inside of a procedure is poor programming practice. It is usually better to load the address outside of the procedure, and then CALL the procedure with the address in place. Next, the ADD AL,[DI] instruction adds the contents of memory location NUMB to AL. Note that AL is initialized to zero, which occurs because DI addresses memory location NUMB, and then the ADD instruction adds its contents to AL. Finally, the ADD AL,[DI+1] instruction adds the contents of memory location NUMB plus one 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 a procedure that adds the contents of array elements 3, 5, and 7. (The procedure and the array elements it adds are chosen to demonstrate the use of some of the addressing modes for the microprocessor.)

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

```
                        ;A procedure that sums ARRAY elements 3, 5, and 7;
                        ;the result is returned in AL.
                        ;
                        ;Note this procedure destroys the contents of SI.
                        ;
0000                    SUM     PROC    NEAR

0000  B0 00                     MOV     AL,0                ;clear sum
0002  BE 0003                   MOV     SI,3                ;address element 3
0005  02 84 0002 R              ADD     AL,ARRAY[SI]        ;add element 3
0009  02 84 0004 R              ADD     AL,ARRAY[SI+2]      ;add element 5
000D  02 84 0006 R              ADD     AL,ARRAY[SI+4]      ;add element 7
0011  C3                        RET

0012                    SUM     ENDP
```

Increment Addition. Increment addition (INC) adds 1 to a register or a memory location. The INC instruction can add 1 to any register or memory location, except a segment register. Table 5–2 illustrates some of the possible forms of the increment instruction available to the 8086–Pentium 4 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, or DWORD PTR directives. The reason is that the 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.

Example 5–5 shows how to modify the procedure of 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 procedures shown in Examples 5–3 and 5–6 add the contents of

TABLE 5–2 Increment instructions.

Assembly Language	*Operation*
INC BL	BL = BL + 1
INC SP	SP = SP + 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 DATA1	Increments the contents of data segment memory location DATA1

NUMB and NUMB+1. The difference between these programs is the way that this data's address is formed through the contents of the DI register using the increment instruction.

EXAMPLE 5–5

```
                      ;A procedure that sums NUMB and NUMB+1;
                      ;the result is returned in AL.
                      ;
                      ;Note that the contents of DI are destroyed.
                      ;
0000                  SUMS   PROC   NEAR

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                ;address NUMB+1
0008  02 05                  ADD    AL,[DI]           ;add NUMB+1
000A  C3                     RET

000B                  SUMS    ENDP
```

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–Pentium 4.

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.

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 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 5–6 shows how the addition occurs with a procedure. Here, 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.

TABLE 5–3 Add-with-carry instructions.

Assembly Language	*Operation*
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with carry 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 carry with the sum stored in BX

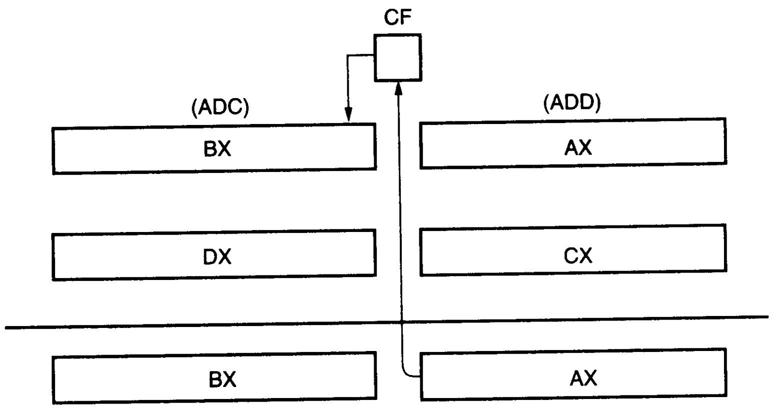


FIGURE 5–1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.

EXAMPLE 5–6

```
                 ;A procedure that sums BX-AX and DX-CX;
                 ;the result is returned in BX-AX.
                 ;
0000             SUM32   PROC   NEAR

0000  03 C1              ADD    AX,CX
0002  13 DA              ADC    BX,DX
0004  C3                 RET

0005             SUM32   ENDP
```

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 a 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.

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 5–7 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 5–7

```
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 5–8 presents a short sequence of instructions that subtract a 44H from a 22H. Here, we

TABLE 5–4 Subtraction instructions.

Assembly Language	*Operation*
SUB CL, BL	CL = CL – BL
SUB AX, SP	AX = AX – SP
SUB DH, 6FH	DH = DH – 6FH
SUB AX, 0CCCCH	AX = AX – CCCCH
SUB [DI], CH	Subtracts the contents of CH from the contents of the data segment memory location addressed by DI
SUB CH, [BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH
SUB AH, TEMP	Subtracts the byte contents of the data segment memory location TEMP from AH

first load the 22H into CH using an immediate move instruction. Next, the SUB instruction, using immediate data 44H, subtracts a 44H from the 22H. After the subtraction, the difference (DEH) 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 5–8

```
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 a 44H (+68) from a 22H (+34), resulting in a DEH (–34). Because the correct 8-bit signed result is a –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 a 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, or DWORD 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 the size of the data to the assembler.

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–Pentium 4. Wide subtractions require that borrows propagate through the subtraction, just as wide additions propagate the carry.

TABLE 5–5 Decrement instructions.

Assembly Language	*Operation*
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
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 NUMB	Subtracts 1 from the contents of the data segment memory location NUMB

TABLE 5–6 Subtraction-with-borrow instructions.

Assembly Language	*Operation*
SBB AH, AL	AH = AH – AL – carry
SBB AX, BX	AX = AX – BX – carry
SBB CL, 2	CL = CL – 2 – carry
SBB BYTE PTR [DI], 3	Both a 3 and carry subtract from the contents of 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 the sum of BP and 2 subtract from DI

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, or DWORD 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 5–9 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 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 5–9

```
0004  2B C7            SUB  AX,DI
0006  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.

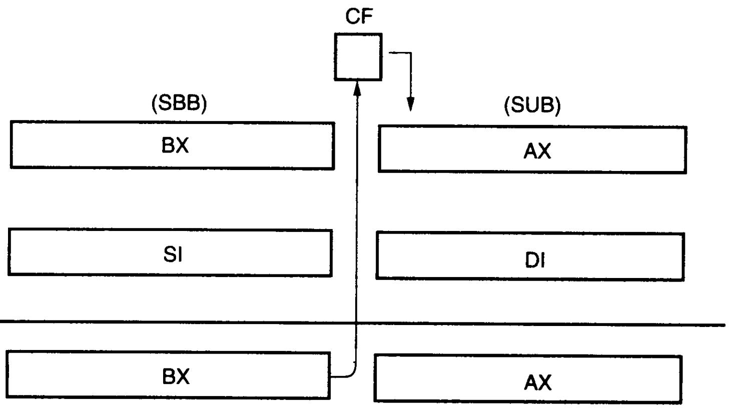


FIGURE 5–2 Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.

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 5–10 shows a comparison followed by a conditional jump instruction. In this example, the contents of AL are compared with a 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. Chapter 6 provides more detail on the comparison and conditional jump instructions.

TABLE 5–7 Comparison instructions.

Assembly Language	*Operation*
CMP CL, BL	CL – BL
CMP AX, SP	AX – SP
CMP AX, 2000H	AX – 2000H
CMP [DI], CH	CH subtracts from the 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 subtract from CL
CMP AH, TEMP	The byte contents of the data segment memory location TEMP subtract from AH
CMP DI, TEMP [BX]	The word contents of the data segment memory location addressed by the sum of TEMP plus BX subtract from DI

EXAMPLE 5–10

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

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–Pentium 4 processors contain special circuitry that performs a multiplication in as little as one clocking period, while 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 Pentium 4 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.

Some flag bits (O and C) 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 0, both C and O flag bits equal 0. 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 the product is 0, both C and O clear to 0. 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 5–11 shows a short program that

TABLE 5–8 8-bit multiplication instructions.

Assembly Language	*Operation*
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 the data segment memory location addressed by TEMP; the unsigned product is in AX

TABLE 5–9 16-bit multiplication instructions.

Assembly Language	*Operation*
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

generates DX = BL × 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 5–11

```
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 true 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 5–11 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 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.

Division

As with multiplication, **division** occurs on 8- or 16-bit numbers in the 8086 processors. 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.

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 divided 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 = FDH (–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 a +3, the result will be a quotient of –5 (AL) with a remainder of –1 (AH). Table 5–10 lists some of the 8-bit division instructions.

TABLE 5–10 8-bit division instructions.

Assembly Language	*Operation*
DIV CL	AX is divided by CL; the unsigned quotient is in AL and the remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the 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 remainder is in AH

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**). 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.

Example 5–12 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.

EXAMPLE 5–12

```
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 5–13 shows the same basic program except that the numbers are signed numbers. This means that instead of zero-extending AL into AH, it is sign-extended with the CBW instruction.

EXAMPLE 5–13

```
0000  A0 0000 R          MOV    AL,NUMB          ;get NUMB
0003  98                 CBW                     ;sign-extend
0004  F6 3E 0002 R       IDIV   NUMB1            ;divide by NUMB1
0008  A2 0003 R          MOV    ANSQ,AL          ;save quotient
000B  88 26 0004 R       MOV    ANSR,AH          ;save remainder
```

16-bit Division. 16-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 5–11 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 0. If AX is a 16-bit signed number, the CWD (**convert word to doubleword**) instruction sign-extends it into a signed 32-bit number.

Example 5–14 shows the division of two 16-bit signed numbers. Here, a –100 in AX is divided by a +9 in CX. The CWD instruction converts the –100 in AX to a –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.

TABLE 5–11 16-bit division instructions.

Assembly Language	*Operation*
DIV CX	DX–AX is divided by CX; the unsigned quotient is in AX and the remainder is in DX
IDIV SI	DX–AX is divided by SI; the signed quotient is in AX and the remainder is in DX
DIV NUMB	AX is divided by the contents of the data segment memory location NUMB; the unsigned quotient is in AX and the remainder is in DX

EXAMPLE 5–14

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

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 result or just dropped to **truncate** the result. 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 5–15 shows a sequence of instructions that divide AX by BL, and round the 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.

EXAMPLE 5–15

```
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
0008  FE C0               INC   AL               ;round
000A            NEXT:
```

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 5–16 shows how a 13 is divided by a 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 an 80H. This is a 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.

EXAMPLE 5–16

```
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
```

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 arithmetic. The ASCII operations are performed on ASCII data used by many programs. In many cases, BCD or ASCII arithmetic is rarely used today.

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 5–17 provides a short sample program that adds the BCD numbers in DX and BX, and stores the result in CX.

EXAMPLE 5–17

```
0000  BA 1234            MOV   DX,1234H          ;load 1,234
0003  BB 3099            MOV   BX,3099H          ;load 3,099
0006  8A C3              MOV   AL,BL             ;sum BL with DL
0008  02 C2              ADD   AL,DL
000A  27                 DAA                     ;adjust
000B  8A C8              MOV   CL,AL             ;answer to CL
000D  8A C7              MOV   AL,BH             ;sum BH, DH, and carry
000F  12 C6              ADC   AL,DH
0011  27                 DAA                     ;adjust
0012  8A E8              MOV   CH,AL             ;answer to CH
```

Because the DAA instruction functions only with the AL register, this addition must occur eight 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 adds to a 3099 to generate a sum of 4333 that 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 5–18 is the same as Example 5–17, 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 5–18

```
0000  BA 1234            MOV   DX,1234H          ;load 1,234
0003  BB 3099            MOV   BX,3099H          ;load 3,099
0006  8A C3              MOV   AL,BL             ;subtract DL from BL
0008  2A C2              SUB   AL,DL
000A  2F                 DAS                     ;adjust
000B  8A C8              MOV   CL,AL             ;answer to CL
000D  8A C7              MOV   AL,BH             ;subtract DH
000F  1A C6              SBB   AL,DH
0011  2F                 DAS                     ;adjust
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, which generates 3130H. The AAA instruction clears AH if the result is less than 10, and adds a 1 to AH if the result is greater than 10.

Example 5–19 shows the way ASCII addition functions in the microprocessor. Please note that AH is cleared before the addition by using the MOV AX,31H instruction. The operand of 0031H places a 00H in AH and a 31H into AL.

EXAMPLE 5–19

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

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 5–20 illustrates how a 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 5–20

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

AAM Instruction. The AAM instruction follows the multiplication instruction after multiplying two one-digit unpacked BCD numbers. Example 5–21 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 a 0205H. This is an unpacked BCD result of 25. If 3030H is added to 0205H, it has an ASCII result of 3235H.

EXAMPLE 5–21

```
0000  B0 05             MOV   AL,5            ;load multiplicand
0002  B1 05             MOV   CL,5            ;load multiplier
0004  F6 E1             MUL   CL
0006  D4 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. It has been noted that the second byte of the instruction contains a 0AH. If the 0AH is changed to another value, AAM divides by the new value. For example, if the second byte is changed to a 0BH, the AAM instruction divides by an 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 a 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 5–22 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 a 3030H is added to convert to ASCII code.

EXAMPLE 5–22

```
0000  33 D2          XOR   DX,DX         ;clear DX register
0002  B9 0064        MOV   CX,100        ;divide DX-AX by 100
0005  F7 F1          DIV   CX
0007  D4 0A          AAM                 ;convert quotient 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 5–23 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.

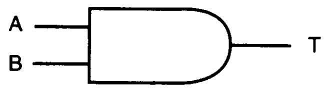
EXAMPLE 5–23

```
                   ;A program that displays the number loaded into AL,
                   ;with the first instruction (48H), as a decimal number.
                   ;
                             .MODEL TINY              ;select TINY model
0000                         .CODE                    ;start of CODE segment
                             .STARTUP                 ;indicate start of program
0100  B0 48                        MOV   AL,48H       ;load AL with 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           ;save least-significant digit
010E  CD 21                        INT   21H
0110  58                           POP   AX           ;restore AL
0111  8A D0                        MOV   DL,AL        ;display least-significant digit
0113  CD 21                        INT   21H
                             .EXIT                    ;exit to DOS
                             END                      ;end of file
```

AAS Instruction. Like other ASCII adjust instructions, AAS adjusts the AX register after an ASCII subtraction. For example, suppose that a 35H subtracts from a 39H. The result will be a 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.

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

(a)



(b)

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

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.

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

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.

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. With the 8086 microprocessor, the AND instruction often executes in about a microsecond. With newer versions, the execution speed is greatly increased. 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 (7408) 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 four 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.

```
  xxxx xxxx   Unknown number
• 0000 1111   Mask
  ---------
  0000 xxxx   Result
```

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

The AND instruction uses any addressing mode except memory-to-memory and segment register addressing. Table 5–12 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.

TABLE 5–12 AND instructions.

Assembly Language	*Operation*
AND AL, BL	AL = AL AND BL
AND CX, DX	CX = CX AND DX
AND CL, 33H	CL = CL AND 33H
AND DI, 4FFFH	DI = DI AND 4FFFH
AND AX, [DI]	AX is ANDed with the word contents of the data segment memory location addressed by DI
AND ARRAY [SI], AL	The byte contents of the data segment memory location addressed by the sum of ARRAY plus SI is ANDed with AL; the result moves to memory

Example 5–24 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 5–24

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

OR

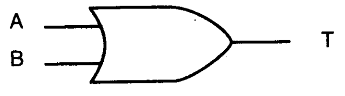
The **OR operation** performs logical addition and is often called the *Inclusive-OR* function. The OR function generates a logic 1 output if any inputs are 1. A 0 appears at the output only when all inputs are 0. The truth table for the OR function appears in Figure 5–5. Here, the inputs A and B OR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

In embedded controller applications, the OR instruction can also replace discrete OR gates. This results in considerable savings because a quad, 2-input OR gate (7432) 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 four bits set, while the leftmost four bits remain unchanged. The OR operation sets any bit; the AND operation clears any bit.

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

(a)



(b)

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

```
  x x x x  x x x x   Unknown number
+ 0 0 0 0  1 1 1 1   Mask
  x x x x  1 1 1 1   Result
```

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

TABLE 5–13 OR instructions.

Assembly Language	*Operation*
OR AH, BL	AH = AH OR BL
OR SI, DX	SI = SI OR DX
OR DH,0A3H	DH = DH OR A3H
OR SP, 990DH	SP = SP OR 990DH
OR DX, [BX]	DX is ORed with the word contents of the data segment memory location addressed by BX
OR DATES [DI + 2], AL	The byte contents of the data segment memory location addressed by the sum of DATES, DI, and 2 are ORed with AL

The OR instruction uses any of the addressing modes allowed to any other instruction except segment register addressing. Table 5–13 illustrates several example OR instructions with comments about their operation.

Suppose that two BCD numbers are multiplied and adjusted with the AAM instruction. The result appears in AX as a two-digit unpacked BCD number. Example 5–25 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 5–25

```
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        ;to ASCII
```

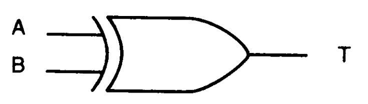
Exclusive-OR

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, while the Inclusive-OR includes it.

Figure 5–7 shows the truth table of the Exclusive-OR function. (Compare this with Figure 5–5 to appreciate the difference between these two OR functions.) If the inputs of the Exclusive-OR function are both 0 or both 1,

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

(a)



(b)

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

TABLE 5–14 Exclusive-OR instructions.

Assembly Language	*Operation*
XOR CH, DL	CH = CH XOR DL
XOR SI, BX	SI = SI XOR BX
XOR AH, 0EEH	AH = AH XOR EEH
XOR DI, 0DDH	DI = DI XOR 00DDH
XOR DX, [SI]	DX is Exclusive-ORed with the word contents of the data segment memory location addressed by SI
XOR DATES [DI + 2], AL	AL is Exclusive-ORed with the byte contents of the data segment memory location addressed by the sum of DATES, DI, and 2

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 5–14 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 7486 quad, 2-input Exclusive-OR gate is replaced by one XOR instruction. The 7486 costs about 40¢, while the instruction costs less than 1/100¢ to store in the memory. Replacing just one 7486 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.

x x x x x x x x	Unknown number
⊕0 0 0 0 1 1 1 1	Mask
x x x x $\overline{x}\,\overline{x}\,\overline{x}\,\overline{x}$	Result

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

Suppose that the leftmost 10 bits of the BX register must be inverted without changing the rightmost six 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 two bytes of memory to store the instruction. Likewise, the MOV CH,00H instruction also clears CH to 00H, but requires three bytes of memory. Because of this saving, the XOR instruction is used to clear a register in place of a move immediate.

Example 5–26 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.

EXAMPLE 5–26

```
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, while 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 5–15 lists some TEST instructions and their operations.

TABLE 5–15 TEST instructions.

Assembly Language	*Operation*
TEST DL, DH	DL is ANDed with DH
TEST CX, BX	CX is ANDed with BX
TEST AH, 4	AH is ANDed with 4

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), while the CMP instruction tests the entire byte or word. 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 non-zero 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 5–27 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 5–27

```
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
```

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 5–16 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 5–16 NOT and NEG instructions.

Assembly Language	*Operation*
NOT CH	CH is one's complemented
NEG CH	CH is two's complemented
NEG AX	AX is two's complemented
NOT TEMP	The contents of the 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 is 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 set 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.

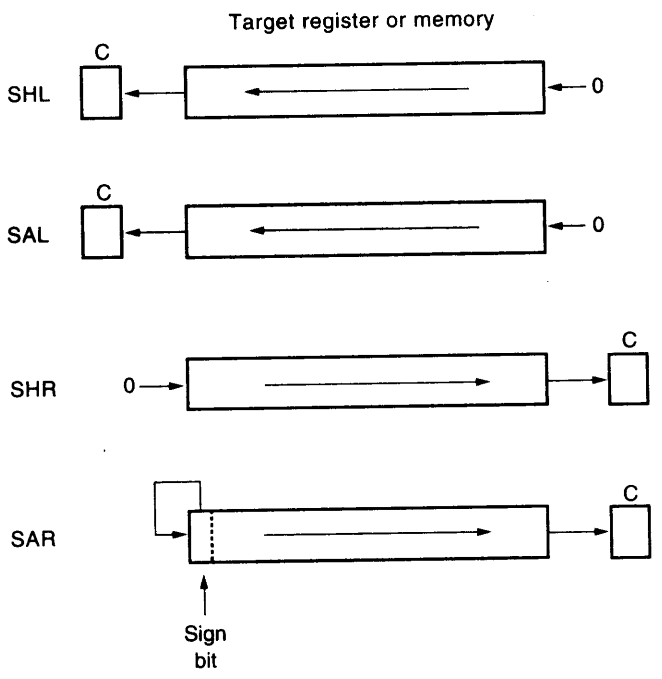


FIGURE 5–9 The shift instructions showing the operation and direction of the shift.

TABLE 5–17 Shift instructions.

Assembly Language	*Operation*
SHL AX, 1	AX is logically shifted left 1 place
SHR BX, 12	BX is logically shifted right 12 places
SAL DATA1, CL	The contents of the data segment memory location DATA1 is arithmetically shifted left the number of places specified by CL
SAR SI, 2	SI is arithmetically shifted right 2 places

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 and logical left shifts are identical. The arithmetic and logical right shifts are different because the arithmetic right shift copies the sign-bit through the number, while the logical right shift copies a 0 through the number.

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

Table 5–17 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 = remainder of 1).

Example 5–28 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 a 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 5–28

```
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 5–29. 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 5–29

```
                        ;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         ;10 times AX
                        ;
                        ;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         ;18 times AX
                        ;
                        ;Multiply AX by 5 (101)
                        ;
0012  8B D8                     MOV   BX,AX
0014  D1 E0                     SHL   AX,1          ;AX times 2
0016  D1 E0                     SHL   AX,1          ;AX times 4
0018  03 C3                     ADD   AX,BX         ;5 times AX
```

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-bit in the 80386 through the Pentium 4. The four available rotate instructions appear in Figure 5–10.

Numbers rotate through 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

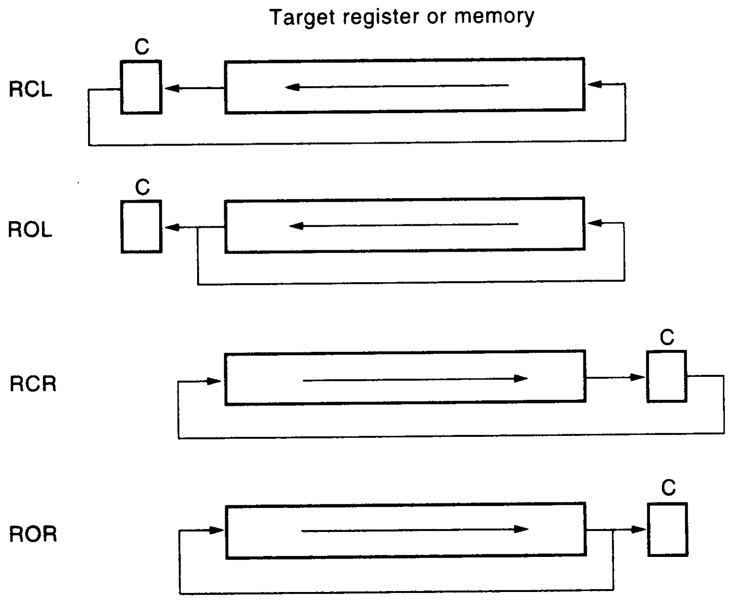


FIGURE 5–10 The rotate instructions showing the direction and operation of each rotate.

TABLE 5–18 Rotate instructions.

Assembly Language	*Operation*
ROL SI, 14	SI rotates left 14 places
RCL BL, 6	BL rotates left through carry 6 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

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 5–18 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.

Rotate instructions are often used to shift wide numbers to the left or right. The program listed in Example 5–30 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 5–30

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

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, and STOS.

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. The SCAS instruction subtracts memory from Al, AX, 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. In all cases, the contents of the extra segment memory location addressed by DI is compared with AL or AX. 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 a 00H. The program in Example 5–31 shows how to search this part of memory for a 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 5–31

```
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             ;search
```

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 5–32.) 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 **(repeat while equal)** 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 5–32

```
0000             SKIP    PROC   FAR

0000 FC                  CLD               ;auto-increment
0001 B9 0100             MOV    CX,256     ;counter
0004 B0 20               MOV    AL,20H     ;get space
0006 F3/AE               REPE SCASB        ;search
0008 CB                  RET

0009             SKIP    ENDP
```

CMPS

The CMPS (compare strings instruction) always compares two sections of memory data as bytes (CMPSB) or words (CMPSW). The contents of the data segment memory location addressed by SI is 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 5–33 illustrates a short procedure that compares two sections of memory searching for a match. The CMPSB instruction is prefixed with a 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 5–33

```
0000              MATCH    PROC   FAR

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              ;counter
000A F3/A6                 REPE   CMPSB              ;search
000C CB                    RET

000D              MATCH    ENDP
```

5–7 SUMMARY

1. Addition (ADD) can be 8-, 16-, or 32-bit. 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).
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, and DWORD 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, or doubleword 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, or DWORD 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.
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.
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.
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.
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.
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.
14. The scan string (SCAS) instruction compares AL or AX 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 22H to CX
   (d) add the data addressed by SI to AL
   (e) add CX to the data stored at memory location FROG
2. What is wrong with the ADD ECX, 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. Select an instruction that adds BX to DX, and also adds the contents of the carry flag (C) to the result.
8. Choose an instruction that adds a 1 to the contents of the SP register.
9. What is wrong with the INC [BX] instruction?
10. 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
11. If DL = 0F3H and BH = 72H, list the difference after BH subtract from DL, and show the contents of the flag register bits.
12. 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.
13. Choose an instruction that subtracts 1 from register EBX.
14. Explain what the SBB [DI–4], DX instruction accomplishes.
15. Explain the difference between the SUB and CMP instruction.
16. When two 8-bit numbers are multiplied, where is the product found?
17. 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.
18. When two numbers multiply, what happens to the O and C flag bits?
19. What is the difference between the IMUL and MUL instructions?
20. 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.
21. Describe the operation of the IMUL BX, DX, 100H instruction.
22. When 8-bit numbers are divided, in which register is the dividend found?