CHAPTER 3

ARITHMETIC AND LOGIC INSTRUCTIONS AND PROGRAMS

OBJECTIVES

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

- **Demonstrate how 8-bit and 16-bit unsigned numbers are added in the 80x86**
- **Convert data to any of the forms: ASCII, packed BCD, or unpacked BCD**
- **Explain the effect of unsigned arithmetic instructions on the flag register**
- **Code the following Assembly language unsigned arithmetic instructions:**
  - **Addition instructions ADD and ADC**
  - **Subtraction instructions SUB and SBB**
  - **Multiplication and division instructions MUL and DIV**
- **Code BCD and ASCII arithmetic instructions:**
  - **DAA, DAS, AAA, AAS, AAM, and AAD**
- **Code the following Assembly language logic instructions:**
  - **AND, OR, and XOR**
  - **Logical shift instructions SHR and SHL**
  - **The compare instruction CMP**
- **Code BCD and ASCII arithmetic instructions**
- **Code bitwise rotation instructions ROR, ROL, RCR, and RCL**
- **Demonstrate an ability to use all of the instructions above in Assembly language programs**
- **Perform bitwise manipulation using the C language**

In this chapter, most of the arithmetic and logic instructions are discussed and program examples are given to illustrate the application of these instructions. Unsigned numbers are used in this discussion of arithmetic and logic instructions. Signed numbers are discussed separately in Chapter 6. Unsigned numbers are defined as data in which all the bits are used to represent data and no bits are set aside for the positive or negative sign. This means that the operand can be between 00 and FFH (0 to 255 decimal) for 8-bit data and between 0000 and FFFFH (0 to 65535 decimal) for 16-bit data. The last section of the chapter describes bitwise operations in the C language.

SECTION 3.1: UNSIGNED ADDITION AND SUBTRACTION

Addition of unsigned numbers

The form of the ADD instruction is

```
ADD  destination,source  ;dest. operand = dest. operand + source operand
```

The instructions ADD and ADC are used to add two operands. The destination operand can be a register or in memory. The source operand can be a register, in memory, or immediate. Remember that memory-to-memory operations are never allowed in 80x86 Assembly language. The instruction could change any of the ZF, SF, AF, CF, or PF bits of the flag register, depending on the operands involved. The effect of the ADD instruction on the overflow flag is discussed in Chapter 6 since it is used in signed number operations. Look at Example 3-1.

Example 3-1

Show how the flag register is affected by

```
MOV     AL,0F5H
ADD     AL,0BH
```

Solution:

```
  F5H        1111 0101
+ 0BH      + 0000 1011
 100H        0000 0000
```

After the addition, the AL register (destination) contains 00 and the flags are as follows:
CF = 1 since there is a carry out from D7
SF = 0 the status of D7 of the result
PF = 1 the number of 1s is zero (zero is an even number)
AF = 1 there is a carry from D3 to D4
ZF = 1 the result of the action is zero (for the 8 bits)

In discussing addition, the following two cases will be examined:

1. Addition of individual byte and word data
2. Addition of multibyte data

CASE 1: Addition of individual byte and word data

In Chapter 2 there was a program that added 5 bytes of data. The total sum was purposely kept less than FFH, the maximum value an 8-bit register can hold. To calculate the total sum of any number of operands, the carry flag should be checked after the addition of each operand. Program 3-1a uses AH to accumulate carries as the operands are added to AL.

Write a program to calculate the total sum of 5 bytes of data. Each byte represents the daily wages of a worker. This person does not make more than $255 (FFH) a day. The decimal data is as follows: 125, 235, 197, 91, and 48.

```
TITLE     PROG3-1A (EXE)  ADDING 5 BYTES
PAGE      60,132
.MODEL SMALL
.STACK 64
;-------------------------------
          .DATA
COUNT     EQU     05
DATA      DB      125,235,197,91,48
          ORG     0008H
SUM       DW      ?
;-------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          MOV     CX,COUNT          ;CX is the loop counter
          MOV     SI,OFFSET DATA;SI is the data pointer
          MOV     AX,00             ;AX will hold the sum
BACK:     ADD     AL,[SI]           ;add the next byte to AL
          JNC     OVER              ;If no carry, continue
          INC     AH                ;else accumulate carry in AH
OVER:     INC     SI                ;increment data pointer
          DEC     CX                ;decrement loop counter
          JNZ     BACK              ;if not finished, go add next byte
          MOV     SUM,AX            ;store sum
          MOV     AH,4CH
          INT     21H               ;go back to DOS
MAIN      ENDP
          END     MAIN
```

Program 3-1a

Analysis of Program 3-1a

These numbers are converted to hex by the assembler as follows: 125 = 7DH, 235 = 0EBH, 197 = 0C5H, 91 = 5BH, 48 = 30H. Three iterations of the loop are shown below. The tracing of the program is left to the reader as an exercise.

1. In the first iteration of the loop, 7DH is added to AL with CF = 0 and AH = 00. CX = 04 and ZF = 0.
2. In the second iteration of the loop, EBH is added to AL, which results in AL = 68H and CF = 1. Since a carry occurred, AH is incremented. CX = 03 and ZF = 0.
3. In the third iteration, C5H is added to AL, which makes AL = 2DH. Again a carry occurred, so AH is incremented again. CX = 02 and ZF = 0.

This process continues until CX = 00 and the zero flag becomes 1, which will cause JNZ to fall through. Then the result will be saved in the word-sized memory set aside in the data segment. Although this program works correctly, due to pipelining it is strongly recommended that the following lines of the program be replaced:

Replace these lines

```
BACK:  ADD     AL,[SI]
       JNC     OVER
       INC     AH
OVER:  INC     SI
```

With these lines

```
BACK:  ADD     AL,[SI]
       ADC     AH,00   ;add 1 to AH if CF=1
       INC     SI
```

The "ADC AH,00" instruction in reality means add 00 + AH + CF and place the result in AH. This is much more efficient since the instruction "JNC OVER" has to empty the queue of pipelined instructions and fetch the instructions from the OVER target every time the carry is zero (CF = 0).

The addition of many word operands works the same way. Register AX (or CX or DX or BX) could be used as the accumulator and BX (or any general-purpose 16-bit register) for keeping the carries. Program 3-1b is the same as Program 3-1a, rewritten for word addition.

Write a program to calculate the total sum of five words of data. Each data value represents the yearly wages of a worker. This person does not make more than $65,555 (FFFFH) a year. The decimal data is as follows: 27345, 28521, 29533, 30105, and 32375.

```
TITLE     PROG3-1B (EXE)  ADDING 5 WORDS
PAGE      60,132
.MODEL SMALL
.STACK 64
;-------------------------------------
          .DATA
COUNT     EQU     05
DATA      DW      27345,28521,29533,30105,32375
          ORG     0010H
SUM       DW      2 DUP(?)
;-------------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          MOV     CX,COUNT        ;CX is the loop counter
          MOV     SI,OFFSET DATA;SI is the data pointer
          MOV     AX,00           ;AX will hold the sum
          MOV     BX,AX           ;BX will hold the carries
BACK:     ADD     AX,[SI]         ;add the next word to AX
          ADC     BX,0            ;add carry to BX
          INC     SI              ;increment data pointer twice
          INC     SI              ;to point to next word
          DEC     CX              ;decrement loop counter
          JNZ     BACK            ;if not finished, continue adding
          MOV     SUM,AX          ;store the sum
          MOV     SUM+2,BX        ;store the carries
          MOV     AH,4CH
          INT     21H             ;go back to DOS
MAIN      ENDP
          END     MAIN
```

Program 3-1b

CASE 2: Addition of multiword numbers

Assume a program is needed that will add the total U. S. budget for the last 100 years or the mass of all the planets in the solar system. In cases like this, the numbers being added could be up to 8 bytes wide or more. Since registers are only 16 bits wide (2 bytes), it is the job of the programmer to write the code to break down these large numbers into smaller chunks to be processed by the CPU. If a 16-bit register is used and the operand is 8 bytes wide, that would take a total of four iterations. However, if an 8-bit register is used, the same operands would require eight iterations. This obviously takes more time for the CPU. This is one reason to have wide registers in the design of the CPU. Large and powerful computers such as the CRAY have registers of 64 bits wide and larger.

Write a program that adds the following two multiword numbers and saves the result:
DATA1 = 548FB9963CE7H and DATA2 = 3FCD4FA23B8DH.

```
TITLE     PROG3-2 (EXE)  MULTIWORD ADDITION
PAGE      60,132
.MODEL SMALL
.STACK 64
;-------------------------------
          .DATA
DATA1     DQ      548FB9963CE7H
          ORG     0010H
DATA2     DQ      3FCD4FA23B8DH
          ORG     0020H
DATA3     DQ      ?
;-------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          CLC                             ;clear carry before first addition
          MOV     SI,OFFSET DATA1         ;SI is pointer for operand1
          MOV     DI,OFFSET DATA2         ;DI is pointer for operand2
          MOV     BX,OFFSET DATA3         ;BX is pointer for the sum
          MOV     CX,04                   ;CX is the loop counter
BACK:     MOV     AX,[SI]                 ;move the first operand to AX
          ADC     AX,[DI]                 ;add the second operand to AX
          MOV     [BX],AX                 ;store the sum
          INC     SI                      ;point to next word of operand1
          INC     SI
          INC     DI                      ;point to next word of operand2
          INC     DI
          INC     BX                      ;point to next word of sum
          INC     BX
          LOOP    BACK                    ;if not finished, continue adding
          MOV     AH,4CH
          INT     21H                     ;go back to DOS
MAIN      ENDP
          END     MAIN
```

Program 3-2

Analysis of Program 3-2

In writing this program, the first thing to be decided was the directive used for coding the data in the data segment. DQ was chosen since it can represent data as large as 8 bytes wide. The question is: Which add instruction should be used? In the addition of multibyte (or multiword) numbers, the ADC instruction is always used since the carry must be added to the next-higher byte (or word) in the next iteration. Before executing ADC, the carry flag must be cleared (CF = 0) so that in the first iteration, the carry would not be added. Clearing the carry flag is achieved by the CLC (clear carry) instruction. Three pointers have been used: SI for DATA1, DI for DATA2, and BX for DATA3 where the result is saved. There is a new instruction in that program, "LOOP XXXX", which replaces the often used "DEC CX" and "JNZ XXXX". In other words:

```
LOOP  xxxx  ;is equivalent to the following two instructions

DEC   CX
JNZ   xxxx
```

When the "LOOP xxxx" is executed, CX is decremented automatically, and if CX is not 0, the microprocessor will jump to target address xxxx. If CX is 0, the next instruction (the one below "LOOP xxxx") is executed.

Subtraction of unsigned numbers

```
SUB  dest,source              ;dest = dest – source
```

In subtraction, the 80x86 microprocessors (indeed, almost all modern CPUs) use the 2's complement method. Although every CPU contains adder circuitry, it would be too cumbersome (and take too many transistors) to design separate subtractor circuitry. For this reason, the 80x86 uses internal adder circuitry to perform the subtraction command. Assuming that the 80x86 is executing simple subtract instructions, one can summarize the steps of the hardware of the CPU in executing the SUB instruction for unsigned numbers, as follows.

1. Take the 2's complement of the subtrahend (source operand).
2. Add it to the minuend (destination operand).
3. Invert the carry.

These three steps are performed for every SUB instruction by the internal hardware of the 80x86 CPU regardless of the source and destination of the operands as long as the addressing mode is supported. It is after these three steps that the result is obtained and the flags are set. Example 3-2 illustrates the three steps.

Example 3-2

Show the steps involved in the following:

```
    MOV    AL,3FH          ;load AL=3FH
    MOV    BH,23H          ;load BH=23H
    SUB    AL,BH           ;subtract BH from AL.  Place result in AL.
```

Solution:

```
  AL     3F   0011 1111          0011 1111
- BH   - 23   - 0010 0011        +1101 1101     (2's complement)
        1C                       1 0001 1100    CF=0 (step 3)
```

The flags would be set as follows: CF = 0, ZF = 0, AF = 0, PF = 0, and SF = 0. The programmer must look at the carry flag (not the sign flag) to determine if the result is positive or negative.

After the execution of SUB, if CF = 0, the result is positive; if CF = 1, the result is negative and the destination has the 2's complement of the result. Normally, the result is left in 2's complement, but the NOT and INC instructions can be used to change it. The NOT instruction performs the 1's complement of the operand; then the operand is incremented to get the 2's complement. See Example 3-3.

Example 3-3

Analyze the following program:

```
;from the data segment:
 DATA1    DB  4CH
 DATA2    DB  6EH
 DATA3    DB  ?
;from the code segment:
          MOV    DH,DATA1      ;load DH with DATA1 value (4CH)
          SUB    DH,DATA2      ;subtract DATA2 (6E) from DH (4CH)
          JNC    NEXT          ;if CF=0 jump to NEXT target
          NOT    DH            ;if CF=1 then take 1's complement
          INC    DH            ;and increment to get 2's complement
NEXT:     MOV    DATA3,DH      ;save DH in DATA3
```

Solution:

Following the three steps for "SUB DH,DATA2":

```
  4C   0100 1100                0100 1100
- 6E   0110 1110   2's comp     +1001 0010
- 22                           0 1101 1110    CF=1 (step 3) the result is negative
```

SBB (subtract with borrow)

This instruction is used for multibyte (multiword) numbers and will take care of the borrow of the lower operand. If the carry flag is 0, SBB works like SUB. If the carry flag is 1, SBB subtracts 1 from the result. Notice the "PTR" operand in Example 3-4. The PTR (pointer) data specifier directive is widely used to specify the size of the operand when it differs from the defined size. In Example 3-4, "WORD PTR" tells the assembler to use a word operand, even though the data is defined as a doubleword.

Example 3-4

Analyze the following program:

```
DATA_A    DD 62562FAH
DATA_B    DD 412963BH
RESULT    DD ?
...       ... ...
          MOV   AX,WORD PTR DATA_A        ;AX=62FA
          SUB   AX,WORD PTR DATA_B        ;SUB 963B from AX
          MOV   WORD PTR RESULT,AX        ;save the result
          MOV   AX,WORD PTR DATA_A +2     ;AX=0625
          SBB   AX,WORD PTR DATA_B +2     ;SUB 0412 with borrow
          MOV   WORD PTR RESULT+2,AX      ;save the result
```

Solution:

After the SUB, AX = 62FA – 963B = CCBF and the carry flag is set. Since CF = 1, when SBB is executed, AX = 625 – 412 – 1 = 212. Therefore, the value stored in RESULT is 0212CCBF.

Review Questions

1. The ADD instruction that has the syntax "ADD destination, source" replaces the _____ operand with the sum of the two operands.
2. Why is the following ADD instruction illegal?
   ADD DATA_1,DATA_2
3. Rewrite the instruction above in a correct form.
4. The ADC instruction that has the syntax "ADC destination, source" replaces the _____ operand with the sum of _____.
5. The execution of part (a) below results in ZF = 1, whereas the execution of part (b) results in ZF = 0. Explain why.
   (a) MOV BL,04FH ADD BL,0B1H (b) MOV BX,04FH ADD BX,0B1H
6. The instruction "LOOP ADD_LOOP" is equivalent to what two instructions?
7. Show how the CPU would subtract 05H from 43H.
8. If CF = 1, AL = 95, and BL = 4F prior to the execution of "SBB AL,BL", what will be the contents of AL after the subtraction?

SECTION 3.2: UNSIGNED MULTIPLICATION AND DIVISION

One of the major changes from the 8080/85 microprocessor to the 8086 was inclusion of instructions for multiplication and division. In this section we cover each one with examples. This is multiplication and division of unsigned numbers. Signed numbers are treated in Chapter 6.

In multiplying or dividing two numbers in the 80x86 microprocessor, the use of registers AX, AL, AH, and DX is necessary since these functions assume the use of those registers.

Multiplication of unsigned numbers

In discussing multiplication, the following cases will be examined: (1) byte times byte, (2) word times word, and (3) byte times word.

byte × byte: In byte by byte multiplication, one of the operands must be in the AL register and the second operand can be either in a register or in memory as addressed by one of the addressing modes discussed in Chapter 1. After the multiplication, the result is in AX. See the following example:

```
RESULT  DW   ?              ;result is defined in the data segment
        ...
        MOV  AL,25H         ;a byte is moved to AL
        MOV  BL,65H         ;immediate data must be in a register
        MUL  BL             ;AL = 25 × 65H
        MOV  RESULT,AX      ;the result is saved
```

In the program above, 25H is multiplied by 65H and the result is saved in word-sized memory named RESULT. In that example, the register addressing mode was used. The next three examples show the register, direct, and register indirect addressing modes.

```
;from the data segment:
DATA1     DB   25H
DATA2     DB   65H
RESULT    DW   ?
;from the code segment:
          MOV  AL,DATA1
          MOV  BL,DATA2
          MUL  BL                    ;register addressing mode
          MOV  RESULT,AX
or
          MOV  AL,DATA1
          MUL  DATA2                 ;direct addressing mode
          MOV  RESULT,AX
or
          MOV  AL,DATA1
          MOV  SI,OFFSET DATA2
          MUL  BYTE PTR [SI]         ;register indirect addressing mode
          MOV  RESULT,AX
```

In the register addressing mode example, any 8-bit register could have been used in place of BL. Similarly, in the register indirect example, BX or DI could have been used as pointers. If the register indirect addressing mode is used, the operand size must be specified with the help of the PTR pseudo-instruction. In the absence of the "BYTE PTR" directive in the example above, the assembler could not figure out if it should use a byte or word operand pointed at by SI. This confusion would cause an error.

word × word: In word by word multiplication, one operand must be in AX and the second operand can be in a register or memory. After the multiplication, registers AX and DX will contain the result. Since word × word multiplication can produce a 32-bit result, AX will hold the lower word and DX the higher word. Example:

```
DATA3     DW   2378H
DATA4     DW   2F79H
RESULT1   DW   2 DUP(?)
          .....
          MOV  AX,DATA3          ;load first operand into AX
          MUL  DATA4             ;multiply it by the second operand
          MOV  RESULT1,AX        ;store the lower word result
          MOV  RESULT1+2,DX      ;store the higher word result
```

word × byte: This is similar to word by word multiplication except that AL contains the byte operand and AH must be set to zero. Example:

```
;from the data segment:
DATA5       DB     6BH
DATA6       DW     12C3H
RESULT3     DW     2 DUP(?)

;from the code segment:
... ...
MOV    AL,DATA5              ;AL holds byte operand
SUB    AH,AH                 ;AH must be cleared
MUL    DATA6                 ;byte in AL multiplied by word operand
MOV    BX,OFFSET RESULT3     ;BX points to storage for product
MOV    [BX],AX               ;AX holds lower word
MOV    [BX]+2,DX             ;DX holds higher word
```

Table 3-1 gives a summary of multiplication of unsigned numbers. Using the 80x86 microprocessor to perform multiplication of operands larger than 16-bit size takes some manipulation, although in such cases the 8087 coprocessor is normally used.

Table 3-1: Unsigned Multiplication Summary

Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX
word × word	AX	register or memory	DX AX
word × byte	AL = byte, AH = 0	register or memory	DX AX

Division of unsigned numbers

In the division of unsigned numbers, the following cases are discussed:

1. Byte over byte
2. Word over word
3. Word over byte
4. Doubleword over word

In divide, there could be cases where the CPU cannot perform the division. In these cases an interrupt is activated. In recent years this is referred to as an *exception*. In what situation can the microprocessor not handle the division and must call an interrupt? They are

1. if the denominator is zero (dividing any number by 00), and
2. if the quotient is too large for the assigned register.

In the IBM PC and compatibles, if either of these cases happens, the PC will display the "divide error" message.

byte/byte: In dividing a byte by a byte, the numerator must be in the AL register and AH must be set to zero. The denominator cannot be immediate but can be in a register or memory as supported by the addressing modes. After the DIV instruction is performed, the quotient is in AL and the remainder is in AH. The following shows the various addressing modes that the denominator can take.

```
DATA7    DB      95
DATA8    DB      10
QOUT1    DB      ?
REMAIN1  DB      ?

;using immediate addressing mode will give an error
         MOV     AL,DATA7      ;move data into AL
         SUB     AH,AH         ;clear AH
         DIV     10            ;immed. mode not allowed!!

;allowable modes include:

;using direct mode
         MOV     AL,DATA7      ;AL holds numerator
         SUB     AH,AH         ;AH must be cleared
         DIV     DATA8         ;divide AX by DATA8
         MOV     QOUT1,AL      ;quotient = AL = 09
         MOV     REMAIN1,AH    ;remainder = AH = 05

;using register addressing mode
         MOV     AL,DATA7      ;AL holds numerator
         SUB     AH,AH         ;AH must be cleared
         MOV     BH,DATA8      ;move denom. to register
         DIV     BH            ;divide AX by BH
         MOV     QOUT1,AL      ;quotient = AL = 09
         MOV     REMAIN1,AH    ;remainder = AH = 05

;using register indirect addressing mode
         MOV     AL,DATA7              ;AL holds numerator
         SUB     AH,AH                 ;AH must be cleared
         MOV     BX,OFFSET DATA8       ;BX holds offset of DATA8
         DIV     BYTE PTR [BX]         ;divide AX by DATA8
         MOV     QOUT2,AX
         MOV     REMAIND2,DX
```

word/word: In this case the numerator is in AX and DX must be cleared. The denominator can be in a register or memory. After the DIV, AX will have the quotient and the remainder will be in DX.

```
         MOV     AX,10050      ;AX holds numerator
         SUB     DX,DX         ;DX must be cleared
         MOV     BX,100        ;BX used for denominator
         DIV     BX
         MOV     QOUT2,AX      ;quotient = AX = 64H = 100
         MOV     REMAIND2,DX   ;remainder = DX = 32H = 50
```

word/byte: Again, the numerator is in AX and the denominator can be in a register or memory. After the DIV instruction, AL will contain the quotient, and AH will contain the remainder. The maximum quotient is FFH. The following program divides AX = 2055 by CL=100. Then AL = 14H (20 decimal) is the quotient and AH = 37H (55 decimal) is the remainder.

```
         MOV     AX,2055       ;AX holds numerator
         MOV     CL,100        ;CL used for denominator
         DIV     CL
         MOV     QUO,AL        ;AL holds quotient
         MOV     REMI,AH       ;AH holds remainder
```

doubleword/word: The numerator is in AX and DX, with the most significant word in DX and the least significant word in AX. The denominator can be in a register or in memory. After the DIV instruction, the quotient will be in AX, the remainder in DX. The maximum quotient is FFFFH.

```
;from the data segment:
DATA1    DD      105432
DATA2    DW      10000
QUOT     DW      ?
REMAIN   DW      ?
;from the code segment:
         MOV     AX,WORD PTR DATA1       ;AX holds lower word
         MOV     DX,WORD PTR DATA1+2     ;DX higher word of numerator
         DIV     DATA2
         MOV     QUOT,AX                 ;AX holds quotient
         MOV     REMAIN,DX               ;DX holds remainder
```

In the program above, the contents of DX:AX are divided by a word-sized data value, 10000. Now one might ask: How does the CPU know that it must use the doubleword in DX:AX for the numerator? The 8086/88 automatically uses DX:AX as the numerator anytime the denominator is a word in size, as was seen earlier in the case of a word divided by a word. This explains why DX had to be cleared in that case. Notice in the example above that DATA1 is defined as DD but fetched into a word-size register with the help of WORD PTR. In the absence of WORD PTR, the assembler will generate an error. A summary of the results of division of unsigned numbers is given in Table 3-2.

Table 3-2: Unsigned Division Summary

Division	Numerator	Denominator	Quotient	Rem
byte/byte	AL = byte, AH = 0	register or memory	AL [1]	AH
word/word	AX = word, DX = 0	register or memory	AX [2]	DX
word/byte	AX = word	register or memory	AL [1]	AH
doubleword/word	DXAX = doubleword	register or memory	AX [2]	DX

Notes:
1. Divide error interrupt if AL > FFH.
2. Divide error interrupt if AX >FFFFH.

Review Questions

1. In unsigned multiplication of a byte in DATA1 with a byte in AL, the product will be placed in register(s) _____.
2. In unsigned multiplication of AX with BX, the product is placed in register(s) _____.
3. In unsigned multiplication of CX with a byte in AL, the product is placed in register(s) _____.
4. In unsigned division of a byte in AL by a byte in DH, the quotient will be placed in _____ and the remainder in _____.
5. In unsigned division of a word in AX by a word in DATA1, the quotient will be placed in _____ and the remainder in _____.
6. In unsigned division of a word in AX by a byte in DATA2, the quotient will be placed in _____ and the remainder in _____.
7. In unsigned division of a doubleword in DXAX by a word in CX, the quotient will be placed in _____ and the remainder in _____.

SECTION 3.3: LOGIC INSTRUCTIONS AND SAMPLE PROGRAMS

In this section we discuss the logic instructions AND, OR, XOR, SHIFT, and COMPARE. Instructions are given in the context of examples.

AND

AND destination, source

This instruction will perform a logical AND on the operands and place the result in the destination. The destination operand can be a register or in memory. The source operand can be a register, in memory, or immediate.

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Example 3-5

Show the results of the following:

```
MOV   BL,35H
AND   BL,0FH          ;AND BL with 0FH. Place the result in BL.
```

Solution:

```
35H   00110101
0FH   00001111
05H   00000101    Flag settings will be: SF = 0, ZF = 0, PF = 1, CF = OF = 0.
```

AND will automatically change the CF and OF to zero and PF, ZF and SF are set according to the result. The rest of the flags are either undecided or unaffected. As seen in Example 3-5, AND can be used to mask certain bits of the operand. It can also be used to test for a zero operand:

```
      AND   DH,DH
      JZ    XXXX
      ...
XXXX: ...
```

The above will AND DH with itself and set ZF = 1 if the result is zero, making the CPU fetch from the target address XXXX. Otherwise, the instruction below JZ is executed. AND can thus be used to test if a register contains zero.

OR

OR destination,source

The destination and source operands are ORed and the result is placed in the destination. OR can be used to set certain bits of an operand to 1. The destination operand can be a register or in memory. The source operand can be a register, in memory, or immediate.

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

The flags will be set the same as for the AND instruction. CF and OF will be reset to zero and SF, ZF, and PF will be set according to the result. All other flags are not affected. See Example 3-6.

The OR instruction can also be used to test for a zero operand. For example, "OR BL,0" will OR the register BL with 0 and make ZF = 1 if BL is zero. "OR BL,BL" will achieve the same result.

Example 3-6

Show the results of the following:

```
MOV AX,0504          ;AX = 0504
OR  AX,0DA68H        ;AX = DF6C
```

Solution:

```
0504H   0000 0101 0000 0100.
DA68H   1101 1010 0110 1000
DF6C    1101 1111 0110 1100
```

Flags will be: SF =1 , ZF = 0, PF = 1, CF = OF = 0.
Notice that parity is checked for the lower 8 bits only.

XOR

XOR dest,src

The XOR instruction will eXclusive-OR the operands and place the result in the destination. XOR sets the result bits to 1 if they are not equal; otherwise, they are reset to 0. The flags are set the same as for the AND instruction. CF = 0 and OF = 0 are set internally and the rest are changed according to the result of the operation. The rules for the operands are the same as in the AND and OR instructions. See Examples 3-7 and 3-8.

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Example 3-7

Show the results of the following:

```
MOV   DH,54H
XOR   DH,78H
```

Solution:

```
54H   0 1 0 1 0 1 0 0
78H   0 1 1 1 1 0 0 0
2C    0 0 1 0 1 1 0 0
```

Flag settings will be: SF = 0, ZF = 0, PF = 0, CF = OF = 0.

Example 3-8

The XOR instruction can be used to clear the contents of a register by XORing it with itself. Show how "XOR AH,AH" clears AH, assuming that AH = 45H.

Solution:

```
45H   01000101
45H   01000101
00    00000000
```

Flag settings will be: SF = 0, ZF = 1, PF =1 , CF = OF = 0.

XOR can also be used to see if two registers have the same value. "XOR BX,CX" will make ZF = 1 if both registers have the same value, and if they do, the result (0000) is saved in BX, the destination.

Another widely used application of XOR is to toggle bits of an operand. For example, to toggle bit 2 of register AL:

```
XOR   AL,04H          ;XOR AL with 0000 0100
```

This would cause bit 2 of AL to change to the opposite value; all other bits would remain unchanged.

SHIFT

There are two kinds of shift: logical and arithmetic. The logical shift is for unsigned operands, and the arithmetic shift is for signed operands. Logical shift will be discussed in this section and the discussion of arithmetic shift is postponed to Chapter 6. Using shift instructions shifts the contents of a register or memory location right or left. The number of times (or bits) that the operand is shifted can be specified directly if it is once only, or through the CL register if it is more than once.

SHR: This is the logical shift right. The operand is shifted right bit by bit, and for every shift the LSB (least significant bit) will go to the carry flag (CF) and the MSB (most significant bit) is filled with 0. Examples 3-9 and 3-10 should help to clarify SHR.

Example 3-9

Show the result of SHR in the following:

```
MOV   AL,9AH
MOV   CL,3      ;set number of times to shift
SHR   AL,CL
```

Solution:

```
9AH = 10011010
      01001101      CF=0   (shifted once)
      00100110      CF=1   (shifted twice)
      00010011      CF=0   (shifted three times)
After three times of shifting right, AL = 13H and CF = 0.
```

If the operand is to be shifted once only, this is specified in the SHR instruction itself rather than placing 1 in the CL. This saves coding of one instruction:

```
MOV   BX,0FFFFH          ;BX=FFFFH
SHR   BX,1               ;shift right BX once only
```

After the shift above, BX = 7FFFH and CF = 1. Although SHR does affect the OF, SF, PF, and ZF flags, they are not important in this case. The operand to be shifted can be in a register or in memory, but immediate addressing mode is not allowed for shift instructions. For example, "SHR 25,CL" will cause the assembler to give an error.

Example 3-10

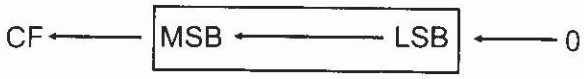
Show the results of SHR in the following:

```
;from the data segment:
DATA1  DW 7777H
;from the code segment:
TIMES  EQU   4
MOV    CL,TIMES       ;CL=04
SHR    DATA1,CL       ;shift DATA1 CL times
```

Solution:

After the four shifts, the word at memory location DATA1 will contain 0777. The four LSBs are lost through the carry, one by one, and 0s fill the four MSBs.

SHL: Shift left is also a logical shift. It is the reverse of SHR. After every shift, the LSB is filled with 0 and the MSB goes to CF. All the rules are the same as SHR.



Example 3-11

Show the effects of SHL in the following:

```
MOV   DH,6
MOV   CL,4
SHL   DH,CL
```

Solution:

```
              00000110
CF=0          00001100          (shifted left once)
CF=0          00011000
CF=0          00110000
CF=0          01100000          (shifted four times)
```

After the four shifts left, the DH register has 60H and CF = 0.

Example 3-11 could have been coded as

```
MOV   DH,6
SHL   DH,1
SHL   DH,1
SHL   DH,1
SHL   DH,1
```

COMPARE of unsigned numbers

```
CMP   destination,source  ;compare dest and src
```

The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged. The destination operand can be in a register or in memory and the source operand can be in a register, in memory, or immediate. Although all the CF, AF, SF, PF, ZF, and OF flags reflect the result of the comparison, only the CF and ZF are used, as outlined in Table 3-3.

Table 3-3: Flag Settings for Compare Instruction

Compare operands	CF	ZF
destination > source	0	0
destination = source	0	1
destination < source	1	0

The following demonstrates how the CMP instruction is used:

```
DATA1   DW      235FH
        ....
        MOV     AX,0CCCCH
        CMP     AX,DATA1      ;compare CCCC with 235F
        JNC     OVER          ;jump if CF=0
        SUB     AX,AX
OVER:   INC     DATA1
```

In the program above, AX is greater than the contents of memory location DATA1 (0CCCCH >235FH); therefore, CF = 0 and JNC (jump no carry) will go to target OVER. In contrast, look at the following:

```
        MOV     BX,7888H
        MOV     CX,9FFFH
        CMP     BX,CX          ;compare 7888 with 9FFF
        JNC     NEXT
        ADD     BX,4000H
NEXT:   ADD     CX,250H
```

In the above, BX is smaller than CX (7888H < 9FFFH), which sets CF = 1, making "JNC NEXT" fall through so that "ADD BX,4000H" is executed. In the example above, CX and BX still have their original values (CX = 9FFFH and BX = 7888H) after the execution of "CMP BX,CX". Notice that CF is always checked for cases of greater or smaller than, but for equal, ZF must be used. The next program sample has a variable named TEMP, which is being checked to see if it has reached 99:

```
TEMP    DB ?
        ...
        MOV     AL,TEMP        ;move the TEMP variable into AL
        CMP     AL,99          ;compare AL with 99
        JZ      HOT_HOT        ;if ZF=1 (TEMP = 99) jump to HOT_HOT
        INC     BX             ;otherwise (ZF=0) increment BX
        ...
HOT_HOT: HLT                   ;halt the system
```

The compare instruction is really a SUBtraction except that the values of the operands do not change. The flags are changed according to the execution of SUB. Although all the flags are affected, the only ones of interest are ZF and CF. It must be emphasized that in CMP instructions, the operands are unaffected regardless of the result of the comparison. Only the flags are affected. This is despite the fact that CMP uses the SUB operation to set or reset the flags. Program 3-3 uses the CMP instruction to search for the highest byte in a series of 5 bytes defined in the data segment. The instruction "CMP AL,[BX]" works as follows, where [BX] is the contents of the memory location pointed at by register BX.

If AL < [BX], then CF = 1 and [BX] becomes the basis of the new comparison.
If AL > [BX], then CF = 0 and AL is the larger of the two values and remains the basis of comparison.

Although JC (jump carry) and JNC (jump no carry) check the carry flag and can be used after a compare instruction, it is recommended that JA (jump above) and JB (jump below) be used for two reasons. One reason is that DEBUG will unassemble JC as JB, and JNC as JA, which may be confusing to beginning programmers. Another reason is that "jump above" and "jump below" are easier to understand than "jump carry" and "jump no carry," since it is more immediately apparent that one number is larger than another, than whether a carry would be generated if the two numbers were subtracted.

Program 3-3 uses the CMP instruction to search through 5 bytes of data to find the highest grade. The program uses register AL to hold the highest grade found so far. AL is given the initial value of 0. A loop is used to compare each of the 5 bytes with the value in AL. If AL contains a higher value, the loop continues to check the next byte. If AL is smaller than the byte being checked, the contents of AL are replaced by that byte and the loop continues.

Assume that there is a class of five people with the following grades: 69, 87, 96, 45, and 75. Find the highest grade.

```
TITLE     PROG3-3 (EXE)  CMP EXAMPLE
PAGE      60,132
.MODEL SMALL
.STACK 64
;-----------------------------------
          .DATA
GRADES    DB      69,87,96,45,75
          ORG     0008
HIGHEST   DB      ?
;-----------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          MOV     CX,5                ;set up loop counter
          MOV     BX,OFFSET GRADES    ;BX points to GRADE data
          SUB     AL,AL               ;AL holds highest grade found so far
AGAIN:    CMP     AL,[BX]             ;compare next grade to highest
          JA      NEXT                ;jump if AL still highest
          MOV     AL,[BX]             ;else AL holds new highest
NEXT:     INC     BX                  ;point to next grade
          LOOP    AGAIN               ;continue search
          MOV     HIGHEST,AL          ;store highest grade
          MOV     AH,4CH
          INT     21H                 ;go back to DOS
MAIN      ENDP
          END     MAIN
```

Program 3-3

Program 3-4 uses the CMP instruction to determine if an ASCII character is uppercase or lowercase. Note that small and capital letters in ASCII have the following values:

Letter	Hex	Binary	Letter	Hex	Binary
A	41	01000001	a	61	01100001
B	42	01000010	b	62	01100010
C	43	01000011	c	63	01100011
...	...	...	...	...	...
Y	59	01011001	y	79	01111001
Z	5A	01011010	z	7A	01111010

As can be seen, there is a relationship between the pattern of lowercase and uppercase letters, as shown below for A and a:

```
A   0100 0001  41H
a   0110 0001  61H
```

The only bit that changes is d5. To change from lowercase to uppercase, d5 must be masked. Program 3-4 first detects if the letter is in lowercase, and if it is, it is ANDed with 1101 1111B = DFH. Otherwise, it is simply left alone. To determine if it is a lowercase letter, it is compared with 61H and 7AH to see if it is in the range a to z. Anything above or below this range should be left alone.

```
TITLE     PROG3-4 (EXE)  LOWERCASE TO UPPERCASE CONVERSION
PAGE      60,132
.MODEL SMALL
.STACK 64
;-------------------------------
          .DATA
DATA1     DB      'mY NAME is jOe'
          ORG     0020H
DATA2     DB      14 DUP(?)
;-------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          MOV     SI,OFFSET DATA1       ;SI points to original data
          MOV     BX,OFFSET DATA2       ;BX points to uppercase data
          MOV     CX,14                 ;CX is loop counter
BACK:     MOV     AL,[SI]               ;get next character
          CMP     AL,61H                ;if less than 'a'
          JB      OVER            ;then no need to convert
          CMP     AL,7AH          ;if greater than 'z'
          JA      OVER            ;then no need to convert
          AND     AL,11011111B    ;mask d5 to convert to uppercase
OVER:     MOV     [BX],AL         ;store uppercase character
          INC     SI              ;increment pointer to original
          INC     BX              ;increment pointer to uppercase data
          LOOP    BACK            ;continue looping if CX > 0
          MOV     AH,4CH
          INT     21H             ;go back to DOS
MAIN      ENDP
          END     MAIN
```

Program 3-4

In Program 3-4, 20H could have been subtracted from the lowercase letters instead of ANDing with 1101 1111B. That is what IBM does, as shown next.

IBM BIOS method of converting from lowercase to uppercase

```
                2357        ;---- CONVERT ANY LOWERCASE TO UPPERCASE
                2358
EBFB            2359        K60:                                ;LOWER TO UPPER
EBFB 3C61       2360                CMP     AL,'a'              ;FIND OUT IF ALPHABETIC
EBFD 7206       2361                JB      K61                 ;NOT_CAPS_STATE
EBFF 3C7A       2362                CMP     AL,'z'
EC01 7702       2363                JA      K61                 ;NOT_CAPS_STATE
EC03 2C20       2364                SUB     AL,'a'-'A'          ;CONVERT TO UPPERCASE
EC05            2365        K61:
EC05 8B1E1C00   2366                MOV     BX,BUFFER_TAIL      ;GET THE END POINTER
                                                                ;TO THE BUFFER
```


Review Questions

1. Use operands 4FCAH and C237H to perform:
   (a) AND (b) OR (c) XOR
2. ANDing a word operand with FFFFH will result in what value for the word operand? To set all bits of an operand to 0, it should be ANDed with _____.
3. To set all bits of an operand to 1, it could be ORed with _____.
4. XORing an operand with itself results in what value for the operand?
5. Show the steps if value A0F2H were shifted left three times. Then show the steps if A0F2H were shifted right three times.
6. The CMP instructions works by performing a(n) _____ operation on the operands and setting the flags accordingly.
7. True or false. The CMP instruction alters the contents of its operands.

BIOS examples of logic instructions

In this section we examine some real-life examples from IBM PC BIOS programs. The purpose is to see the instructions discussed so far in the context of real-life applications.

When the computer is turned on, the CPU starts to execute the programs stored in BIOS in order to set the computer up for DOS. If anything has happened to the BIOS programs, the computer can do nothing. The first subroutine of BIOS is to test the CPU. This involves checking the flag register bit by bit as well as checking all other registers. The BIOS program for testing the flags and registers is given followed by their explanation:

```
                306          ASSUME CS:CODE,DS:NOTHING,ES:NOTHING,SS:NOTHING
E05B            307          ORG    0E05BH
E05B                         ...    ...
E05B            309 START:
E05B FA         310          CLI                    ; DISABLE INTERRUPTS
E05C B4D5       311          MOV    AH,0D5H         ; SET SF, CF, ZF, AND AF FLAGS ON
E05E 9E         312          SAHF
E05F 734C       313          JNC    ERRO1           ; GO TO ERR ROUTINE IF CF NOT SET
E061 754A       314          JNZ    ERRO1           ; GO TO ERR ROUTINE IF ZF NOT SET
E063 7B48       315          JNP    ERRO1           ; GO TO ERR ROUTINE IF PF NOT SET
E065 7946       316          JNS    ERRO1           ; GO TO ERR ROUTINE IF SF NOT SET
E067 9F         317          LAHF                   ; LOAD FLAG IMAGE TO AH
E068 B105       318          MOV    CL,5            ; LOAD CNT REG WITH SHIFT CNT
E06A D2EC       319          SHR    AH,CL           ; SHIFT AF INTO CARRY BIT POS
E06C 733F       320          JNC    ERRO1           ; GO TO ERR ROUTINE IF AF NOT SET
E06E B040       321          MOV    AL,40H          ; SET THE OF FLAG ON
E070 D0E0       322          SHL    AL,1            ; SETUP FOR TESTING
E072 7139       323          JNO    ERRO1           ; GO TO ERR ROUTINE IF OF NOT SET
E074 32E4       324          XOR    AH,AH           ; SET AH = 0
E076 9E         325          SAHF                   ; CLEAR SF, CF, ZF, AND PF
E077 7634       326          JBE    ERRO1           ; GO TO ERR ROUTINE IF CF ON
                327                                 ; OR GO TO ERR ROUTINE IF ZF ON
E079 7832       328          JS     ERRO1           ; GO TO ERR ROUTINE IF SF ON
E07B 7A30       329          JP     ERRO1           ; GO TO ERR ROUTINE IF PF ON
E07D 9F         330          LAHF                   ; LOAD FLAG IMAGE TO AH
E07E B105       331          MOV    CL,5            ; LOAD CNT REG WITH SHIFT CNT
E080 D2EC       332          SHR    AH,CL           ; SHIFT 'AF' INTO CARRY BIT POS
E082 7229       333          JC     ERRO1           ; GO TO ERR ROUTINE IF ON
E084 D0E4       334          SHL    AH,1            ; CHECK THAT 'OF' IS CLEAR
E086 7025       335          JO     ERRO1           ; GO TO ERR ROUTINE IF ON
                336
                337 ;----- READ/WRITE THE 888 GENERAL AND SEGMENTATION REGISTERS
                338 ;          WITH ALL ONES AND ZEROES.
                339
E088 B8FFFF     340          MOV    AX,0FFFFH       ; SET UP ONE'S PATTERN IN AX
E08B F9         341          STC
E08C            342 C8:
E08C 8ED8       343          MOV    DS,AX           ; WRITE PATTERN TO ALL REGS
E08E 8CDB       344          MOV    BX,DS
E090 8EC3       345          MOV    ES,BX
E092 8CC1       346          MOV    CX,ES
E094 8ED1       347          MOV    SS,CX
E096 8CD2       348          MOV    DX,SS
E098 8BE2       349          MOV    SP,DX
E09A 8BEC       350          MOV    BP,SP
E09C 8BF5       351          MOV    SI,BP
E09E 8BFE       352          MOV    DI,SI
E0A0 7307       353          JNC    C9              ; TST1A
E0A2 33C7       354          XOR    AX,DI           ; PATTERN MAKE IT THRU ALL REGS
E0A4 7507       355          JNZ    ERRO1           ; NO - GO TO ERR ROUTINE
E0A6 F8         356          CLC
E0A7 EBE3       357          JMP    C8
E0A9            358 C9:
E0A9 0BC7       359          OR     AX,DI           ; ZERO PATTERN MAKE IT THRU?
E0AB 7401       360          JZ     C10             ; YES - GO TO NEXT TEST
E0AD F4         361 ERRO1: HLT                      ; HALT SYSTEM
```

Line-by-line explanation:

Line	Explanation
310	CLI ensures that no interrupt will occur while the test is being conducted.
311	MOV AH,0D5H: flag S Z - AC - P - C D5H 1 1 0 1 0 1 0 1
312	SAHF (store AH into lower byte of the flag register) is one way to move data to flags. Another is to use the stack MOV AX,00D5H PUSH AX POPF However, there is no RAM available yet to use for the stack because the CPU is tested before memory is tested.
313 - 316	Will make the CPU jump to HLT if any flag does not work.
317	LAHF (load AH with the lower byte of flag register) is the opposite of SAHF.
318	Loads CL for five shifts.
319	"SHR AH,CL". By shifting AH five times, AF (auxiliary carry) will be in the CF position.
320	If no AF, there is an error. Lines 317 to 320 are needed because there is no jump condition instruction for AF.
321 - 323	Checks the OF flag. This is discussed in Chapter 6 when signed numbers are discussed.
324 - 335	Checks the same flags for zero. Remember that JNZ is the same as JBE.
340	Loads AX with FFFFH.
341	STC (set the carry) makes CF = 1.
343 - 352	Moves the AX value (FFFFH) into every register and ends up with DI = FFFFH if the registers are good.
353	Since CF=1 (remember STC) it falls through.
354	Exclusive-ORing AX and DI with both having the same FFFFH value makes AX = 0000 and ZF = 1 if the registers are good (see lines 343 - 352). If ZF = 0, one of the registers must have corrupted the data FFFF, therefore the CPU is bad.
355	If ZF = 0, there is an error.
356	CLC clears the carry flag. This is the opposite of STC.
357	Jumps to C8 and repeats the same process, this time with value 0000. The contents of AX are moved around to every register until DI = 0000, and at 353 the JNC C9 will jump since CF = 0 by the CLC instruction before it went to the loop.
359	At C9, AX and DI are ORed. If 0000, the contents of AX are copied successfully to all registers, DI will be 0000; therefore, ORing will raise the ZF, making ZF = 1.
360	If ZF = 1, the CPU is good and the system can perform the next test. Otherwise, ZF = 0, meaning that the CPU is bad and the system should be halted.

SECTION 3.4: BCD AND ASCII OPERANDS AND INSTRUCTIONS

In 80x86 microprocessors, there are many instructions that handle ASCII and BCD numbers. This section covers these instructions with examples.

BCD number system

BCD stands for binary coded decimal. BCD is needed because human beings use the digits 0 to 9 for numbers. Binary representation of 0 to 9 is called BCD (see Figure 3-1). In computer literature one encounters two terms for BCD numbers:
(1) unpacked BCD (2) packed BCD

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figure 3-1. BCD Code

Unpacked BCD

In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0. Example: "0000 1001" and "0000 0101" are unpacked BCD for 9 and 5, respectively. In the case of unpacked BCD it takes 1 byte of memory location or a register of 8 bits to contain it.

Packed BCD

In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits. For example, "0101 1001" is packed BCD for 59. It takes only 1 byte of memory to store the packed BCD operands. This is one reason to use packed BCD since it is twice as efficient in storing data.

ASCII numbers

In ASCII keyboards, when key "0" is activated, for example, "011 0000" (30H) is provided to the computer. In the same way, 31H (011 0001) is provided for key "1", and so on, as shown in the following list.

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

It must be noted that although ASCII is standard in the United States (and many other countries), BCD numbers have universal application. Now since the keyboard and printers and monitors are all in ASCII, how does data get converted from ASCII to BCD, and vice versa? These are the subjects covered next.

ASCII to BCD conversion

To process data in BCD, first the ASCII data provided by the keyboard must be converted to BCD. Whether it should be converted to packed or unpacked BCD depends on the instructions to be used. There are instructions that require that data be in unpacked BCD and there are others that must have packed BCD data to work properly. Each is covered separately.

ASCII to unpacked BCD conversion

To convert ASCII data to BCD, the programmer must get rid of the tagged "011" in the higher 4 bits of the ASCII. To do that, each ASCII number is ANDed with "0000 1111" (0FH), as shown in the next example. This example is written in three different ways using different addressing modes. The following three programs show three different methods for converting the 10 ASCII digits to unpacked BCD. All use the same data segment:

```
ASC       DB      '9562481273'
          ORG     0010H
UNPACK    DB      10 DUP(?)
```

In Program 3-5a, notice that although the data was defined as DB, a byte definition directive, it was accessed in word-sized chunks. This is a workable approach; however, using the PTR directive as shown in Program 3-5b makes the code more readable for programmers.

```
        MOV   CX,5
        MOV   BX,OFFSET ASC        ;BX points to ASCII data
        MOV   DI,OFFSET UNPACK     ;DI points to unpacked BCD data
AGAIN:  MOV   AX,[BX]              ;move next 2 ASCII numbers to AX
        AND   AX,0F0FH             ;remove ASCII 3s
        MOV   [DI],AX              ;store unpacked BCD
        ADD   DI,2                 ;point to next unpacked BCD data
        ADD   BX,2                 ;point to next ASCII data
        LOOP  AGAIN
```

Program 3-5a

```
        MOV   CX,5                 ;CX is loop counter
        MOV   BX,OFFSET ASC        ;BX points to ASCII data
        MOV   DI,OFFSET UNPACK     ;DI points to unpacked BCD data
AGAIN:  MOV   AX,WORD PTR [BX]     ;move next 2 ASCII numbers to AX
        AND   AX,0F0FH             ;remove ASCII 3s
        MOV   WORD PTR [DI],AX     ;store unpacked BCD
        ADD   DI,2                 ;point to next unpacked BCD data
        ADD   BX,2                 ;point to next ASCII data
        LOOP  AGAIN
```

Program 3-5b

In both of the solutions so far, registers BX and DI were used as pointers for an array of data. An array is simply a set of data located in consecutive memory locations. Now one might ask: What happens if there are four, five, or six arrays? How can they all be accessed with only three registers as pointers: BX, DI, and SI? Program 3-5c shows how this can be done with a single register used as a pointer to access two arrays. However, to do that, the arrays must be of the same size and defined similarly.

```
        MOV   CX,10                ;load the counter
        SUB   BX,BX                ;clear BX
AGAIN:  MOV   AL,ASC[BX]           ;move to AL content of mem [BX+ASC]
        AND   AL,0FH               ;mask the upper nibble
        MOV   UNPACK[BX],AL        ;move to mem [BX+UNPACK] the AL
        INC   BX                   ;make the pointer to point at next byte
        LOOP  AGAIN                ;loop until it is finished
```

Program 3-5c

Program 3-5c uses the based addressing mode since BX+ASC is used as a pointer. ASC is the displacement added to BX. Either DI or SI could have been used for this purpose. For word-sized operands, "WORD PTR" would be used since the data is defined as DB. This is shown below.

```
MOV     AX,WORD PTR ASC[BX]
AND     AX,0F0FH
MOV     WORD PTR UNPACKED[BX],AX
```

ASCII to packed BCD conversion

To convert ASCII to packed BCD, it is first converted to unpacked BCD (to get rid of the 3) and then combined to make packed BCD. For example, for 9 and 5 the keyboard gives 39 and 35, respectively. The goal is to produce 95H or "1001 0101", which is called packed BCD, as discussed earlier. This process is illustrated in detail below.

Key	ASCII	Unpacked BCD	Packed BCD
4	34	00000100	
7	37	00000111	01000111 or 47H

```
          ORG     0010H
VAL_ASC   DB      '47'
VAL_BCD   DB      ?
;reminder: the DB will put 34 in 0010H location and 37 in 0011H.
          MOV     AX,WORD PTR VAL_ASC     ;AH=37,AL=34
          AND     AX,0F0FH                ;mask 3 to get unpacked BCD
          XCHG    AH,AL                   ;swap AH and AL. :
          MOV     CL,4                    ;CL=04 to shift 4 times
          SHL     AH,CL                   ;shift left AH to get AH=40H
          OR      AL,AH                   ;OR them to get packed BCD
          MOV     VAL_BCD,AL              ;save the result
```

After this conversion, the packed BCD numbers are processed and the result will be in packed BCD format. As will be seen later in this section, there are special instructions, such as DAA and DAS, which require that the data be in packed BCD form and give the result in packed BCD. For the result to be displayed on the monitor or be printed by the printer, it must be in ASCII format. Conversion from packed BCD to ASCII is discussed next.

Packed BCD to ASCII conversion

To convert packed BCD to ASCII, it must first be converted to unpacked and then the unpacked BCD is tagged with 011 0000 (30H). The following shows the process of converting from packed BCD to ASCII.

Packed BCD	Unpacked BCD	ASCII
29H	02H & 09H	32H & 39H
0010 1001	0000 0010 & 0000 1001	011 0010 & 011 1001

```
VAL1_BCD DB 29H
VAL3-ASC DW?
          ...
          MOV     AL,VAL1_BCD
          MOV     AH,AL             ;copy AL to AH. now AH=29,AL=29H
          AND     AX,0F00FH         ;mask 9 from AH and 2 from AL
          MOV     CL,4              ;CL=04 for shift
          SHR     AH,CL             ;shift right AH to get unpacked BCD
          OR      AX,3030H          ;combine with 30 to get ASCII
          XCHG    AH,AL             ;swap for ASCII storage convention
          MOV     VAL3_ASC,AX       ;store the ASCII
```

BCD addition and subtraction

After learning how to convert ASCII to BCD, the application of BCD numbers is the next step. There are two instructions that deal specifically with BCD numbers: DAA and DAS. Each is discussed separately.

BCD addition and correction

There is a problem with adding BCD numbers, which must be corrected. The problem is that after adding packed BCD numbers, the result is no longer BCD. Look at this example:

```
MOV AL,17H
ADD AL,28H
```

Adding them gives 0011 1111B (3FH), which is not BCD! A BCD number can only have digits from 0000 to 1001 (or 0 to 9). In other words, adding two BCD numbers must give a BCD result. The result above should have been 17 + 28 = 45 (0100 0101). To correct this problem, the programmer must add 6 (0110) to the low digit:

3F + 06 = 45H. The same problem could have happened in the upper digit (for example, in 52H + 87H = D9H). Again to solve this problem, 6 must be added to the upper digit (D9H +60H =139H), to ensure that the result is BCD (52 + 87 = 139). This problem is so pervasive that all single-chip CISC microprocessors such as the Intel 80x86 and the Motorola 680x0 have an instruction to deal with it. The RISC processors have eliminated this instruction.

DAA

The DAA (decimal adjust for addition) instruction in 80x86 microprocessors is provided exactly for the purpose of correcting the problem associated with BCD addition. DAA will add 6 to the lower nibble or higher nibble if needed; otherwise, it will leave the result alone. The following example will clarify these points:

```
DATA1  DB    47H
DATA2  DB    25H
DATA3  DB    ?

       MOV   AL,DATA1      ;AL holds first BCD operand
       MOV   BL,DATA2      ;BL holds second BCD operand
       ADD   AL,BL         ;BCD addition
       DAA                 ;adjust for BCD addition
       MOV   DATA3,AL      ;store result in correct BCD form
```

After the program is executed, the DATA3 field will contain 72H (47 + 25 = 72). Note that DAA works only on AL. In other words, while the source can be an operand of any addressing mode, the destination must be AL in order for DAA to work. It needs to be emphasized that DAA must be used after the addition of BCD operands and that BCD operands can never have any digit greater than 9. In other words, no A - F digit is allowed. It is also important to note that DAA works only after an ADD instruction; it will not work after the INC instruction.

Summary of DAA action

1. If after an ADD or ADC instruction the lower nibble (4 bits) is greater than 9, or if AF = 1, add 0110 to the lower 4 bits.
2. If the upper nibble is greater than 9, or if CF = 1, add 0110 to the upper nibble.

In reality there is no other use for the AF (auxiliary flag) except for BCD addition and correction. For example, adding 29H and 18H will result in 41H, which is incorrect as far as BCD is concerned.

```
  Hex        BCD
   29        0010 1001
 + 18      + 0001 1000
   41        0100 0001   AF = 1
 +  6      +      0110   because AF = 1 DAA will add 6 to the lower nibble
   47        0100 0111   The final result is BCD.
```

Program 3-6 demonstrates the use of DAA after addition of multibyte packed BCD numbers.

BCD subtraction and correction

The problem associated with the addition of packed BCD numbers also shows up in subtraction. Again, there is an instruction (DAS) specifically designed to solve the problem. Therefore, when subtracting packed BCD (single-byte or multibyte) operands, the DAS instruction is put after the SUB or SBB instruction. AL must be used as the destination register to make DAS work.

Two sets of ASCII data have come in from the keyboard. Write and run a program to :

1. Convert from ASCII to packed BCD.
2. Add the multibyte packed BCD and save it.
3. Convert the packed BCD result into ASCII.

```
TITLE     PROG3-6 (EXE)  ASCII TO BCD CONVERSION AND ADDITION
PAGE      60,132
.MODE SMALL
.STACK 64
;-------------------------------
          .DATA
DATA1_ASC DB      '0649147816'
          ORG     0010H
DATA2_ASC DB      '0072687188'
          ORG     0020H
DATA3_BCD DB      5 DUP (?)
          ORG     0028H
DATA4_BCD DB      5 DUP (?)
          ORG     0030H
DATA5_ADD DB      5 DUP (?)
          ORG     0040H
DATA6_ASC DB      10 DUP (?)
;-------------------------------
          .CODE
MAIN      PROC    FAR
          MOV     AX,@DATA
          MOV     DS,AX
          MOV     BX,OFFSET DATA1_ASC     ;BX points to first ASCII data
          MOV     DI,OFFSET DATA3_BCD     ;DI points to first BCD data
          MOV     CX,10                   ;CX holds number bytes to convert
          CALL    CONV_BCD                ;convert ASCII to BCD
          MOV     BX,OFFSET DATA2_ASC     ;BX points to second ASCII data
          MOV     DI,OFFSET DATA4_BCD     ;DI points to second BCD data
          MOV     CX,10                   ;CX holds number bytes to convert
          CALL    CONV_BCD                ;convert ASCII to BCD
          CALL    BCD_ADD                 ;add the BCD operands
          MOV     SI,OFFSET DATA5_ADD     ;SI points to BCD result
          MOV     DI,OFFSET DATA6_ASC     ;DI points to ASCII result
          MOV     CX,05                   ;CX holds count for convert
          CALL    CONV_ASC                ;convert result to ASCII
          MOV     AH,4CH
          INT     21H                     ;go back to DOS
MAIN      ENDP
;-------------------------------
;THIS SUBROUTINE CONVERTS ASCII TO PACKED BCD
CONV_BCD  PROC
AGAIN:    MOV     AX,[BX] ;BX=pointer for ASCII data
          XCHG    AH,AL
          AND     AX,0F0FH        ;mask ASCII 3s
          PUSH    CX              ;save the counter
          MOV     CL,4            ;shift AH left 4 bits
          SHL     AH,CL           ;to get ready for packing
          OR      AL,AH           ;combine to make packed BCD
          MOV     [DI],AL         ;DI=pointer for BCD data
          ADD     BX,2            ;point to next 2 ASCII bytes
          INC     DI              ;point to next BCD data
          POP     CX              ;restore loop counter
          LOOP    AGAIN
          RET
CONV_BCD  ENDP
;-------------------------------
```

Program 3-6 *(continued on the following page)*

```
;THIS SUBROUTINE ADDS TWO MULTIBYTE PACKED BCD OPERANDS
BCD_ADD  PROC
         MOV   BX,OFFSET DATA3_BCD ;BX=pointer for operand 1
         MOV   DI,OFFSET DATA4_BCD ;DI=pointer for operand 2
         MOV   SI,OFFSET DATA5_ADD ;SI=pointer for sum
         MOV   CX,05
         CLC
BACK:    MOV   AL,[BX]+4           ;get next byte of operand 1
         ADC   AL,[DI]+4           ;add next byte of operand 2
         DAA                       ;correct for BCD addition
         MOV   [SI] +4,AL          ;save sum
         DEC   BX                  ;point to next byte of operand 1
         DEC   DI                  ;point to next byte of operand 2
         DEC   SI                  ;point to next byte of sum
         LOOP  BACK
         RET
BCD_ADD  ENDP
;-------------------------------------
;THIS SUBROUTINE CONVERTS FROM PACKED BCD TO ASCII
CONV_ASC PROC
AGAIN2:  MOV   AL,[SI]             ;SI=pointer for BCD data
         MOV   AH,AL               ;duplicate to unpack
         AND   AX,0F00FH           ;unpack
         PUSH  CX                  ;save counter
         MOV   CL,04               ;shift right 4 bits to unpack
         SHR   AH,CL               ;the upper nibble
         OR    AX,3030H            ;make it ASCII
         XCHG  AH,AL               ;swap for ASCII storage convention
         MOV   [DI],AX             ;store ASCII data
         INC   SI                  ;point to next BCD data
         ADD   DI,2                ;point to next ASCII data
         POP   CX                  ;restore loop counter
         LOOP  AGAIN2
         RET
CONV_ASC ENDP
         END   MAIN
```

Program 3-6 *(continued from preceding page)*

Summary of DAS action

1. If after a SUB or SBB instruction the lower nibble is greater than 9, or if AF = 1, subtract 0110 from the lower 4 bits.
2. If the upper nibble is greater than 9, or CF = 1, subtract 0110 from the upper nibble.

Due to the widespread use of BCD numbers, a specific data directive, DT, has been created. DT can be used to represent BCD numbers from 0 to $10^{20}-1$ (that is, twenty 9s). Assume that the following operands represent the budget, the expenses, and the balance, which is the budget minus the expenses.

```
BUDGET    DT    87965141012
EXPENSES  DT    31610640392
BALANCE   DT    ?                          ;balance = budget - expenses

          MOV   CX,10                      ;counter=10
          MOV   BX,00                      ;pointer=0
          CLC                              ;clear carry for the 1st iteration
BACK:     MOV   AL,BYTE PTR BUDGET[BX]     ;get a byte of the BUDGET
          SBB   AL,BYTE PTR EXPENSES[BX]   ;subtract a byte from it
          DAS                              ;correct the result for BCD
          MOV   BYTE PTR BALANCE[BX],AL    ;save it in BALANCE
          INC   BX                         ;increment for the next byte
          LOOP  BACK                       ;continue until CX=0
```

Notice in the code section above that (1) no H (hex) indicator is needed for BCD numbers when using the DT directive, and (2) the use of the based relative addressing mode (BX + displacement) allows access to all three arrays with a single register BX.

In Program 3-7 the DB directive is used to define ASCII values. This makes the LSD (least significant digit) be located at the highest memory location of the array. In VALUE1, 37, the ASCII for 7 is in memory location 0009; therefore, BX must be pointed to that and then decremented. Program 3-7 is repeated, rewritten with the full segment definition.

```
TITLE     PROG3-7 (EXE)  ADDING ASCII NUMBERS
PAGE      60,132
          .MODEL SMALL
          .STACK 64
;-----------------------
          .DATA
VALUE1    DB     '0659478127'
          ORG    0010H
VALUE2    DB     '0779563678'
          ORG    0020H
RESULT1   DB     10 DUP (?)
          ORG    0030H
RESULT2   DB     10 DUP (?)
;-----------------------
          .CODE
MAIN:     MOV    AX,@DATA
          MOV    DS,AX
          CALL   ASC_ADD          ;call ASCII addition subroutine
          CALL   CONVERT          ;call convert to ASCII subroutine
          MOV    AH,4CH
          INT    21H              ;go back to DOS
;-----------------------
;THIS SUBROUTINE ADDS THE ASCII NUMBERS AND MAKES THE RESULT UNPACKED.
ASC_ADD   PROC
          CLC                     ;clear the carry
          MOV    CX,10            ;set up loop counter
          MOV    BX,9             ;point to LSD
BACK:     MOV    AL,VALUE1[BX]    ;move next byte of operand 1
          ADC    AL,VALUE2[BX]    ;add next byte of operand 2
          AAA                     ;adjust to make it ASCII
          MOV    RESULT1[BX],AL   ;store ASCII sum
          DEC    BX               ;point to next byte
          LOOP   BACK
          RET
ASC_ADD   ENDP
;-----------------------
;THIS SUBROUTINE CONVERTS UNPACKED BCD TO ASCII
CONVERT   PROC
          MOV    BX,OFFSET RESULT1    ;BX points to BCD data
          MOV    SI,OFFSET RESULT2    ;SI points to ASCII data
          MOV    CX,05                ;CX is loop counter
BACK2:    MOV    AX,WORD PTR [BX]     ;get next 2 ASCII bytes
          OR     AX,3030H             ;insert ASCII 3s
          MOV    WORD PTR [SI],AX     ;store ASCII
          ADD    BX,2                 ;increment BCD pointer
          ADD    SI,2                 ;increment ASCII pointer
          LOOP   BACK2
          RET
CONVERT   ENDP
          END    MAIN
```

Program 3-7

```
TITLE   PROG3-7 (EXE) REWRITTEN WITH FULL SEGMENT DEFINITION
PAGE    60,132
STSEG   SEGMENT
        DB      64 DUP (?)
STSEG   ENDS
;-------------------------
DTSEG   SEGMENT
VALUE1  DB      '0659478127'
        ORG     0010H
VALUE2  DB      '0779563678'
        ORG     0020H
RESULT1 DB      10 DUP (?)
        ORG     0030H
RESULT2 DB      10 DUP (?)
DTSEG   ENDS
;-------------------------
CDSEG   SEGMENT
MAIN    PROC    FAR
        ASSUME CS:CDSEG,DS:DTSEG,SS:STSEG
        MOV     AX,DTSEG
        MOV     DS,AX
        CALL    ASC_ADD                 ;call ASCII addition subroutine
        CALL    CONVERT                 ;call convert to ASCII subroutine
        MOV     AH,4CH
        INT     21H                     ;go back to DOS
MAIN    ENDP
;-------------------------
;THIS SUBROUTINE ADDS THE ASCII NUMBERS AND MAKES THE RESULT UNPACKED.
ASC_ADD PROC
        CLC                     ;clear the carry
        MOV     CX,10           ;set up loop counter
        MOV     BX,9            ;point to LSD
BACK:   MOV     AL,VALUE1[BX]   ;move next byte of operand 1
        ADC     AL,VALUE2[BX]   ;add next byte of operand 2
        AAA                     ;adjust to make it unpacked BCD
        MOV     RESULT1[BX],AL  ;store BCD sum
        DEC     BX              ;point to next byte
        LOOP    BACK
        RET
ASC_ADD ENDP
;-------------------------
;THIS SUBROUTINE CONVERTS UNPACKED BCD TO ASCII
CONVERT PROC
        MOV     BX,OFFSET RESULT1   ;BX points to unpacked BCD data
        MOV     SI,OFFSET RESULT2   ;SI points to ASCII data
        MOV     CX,05               ;CX is loop counter
BACK2:  MOV     AX,WORD PTR [BX]    ;get next 2 ASCII bytes
        OR      AX,3030H            ;insert ASCII 3s
        MOV     WORD PTR [SI],AX    ;store ASCII
        ADD     BX,2                ;increment BCD pointer
        ADD     SI,2                ;increment ASCII pointer
        LOOP    BACK2
        RET
CONVERT ENDP
CDSEG   ENDS
        END     MAIN
```

Program 3-7, rewritten with full segment definition

ASCII addition and subtraction

ASCII numbers can be used as operands in add and subtract instructions the way they are, without masking the tagged 011, using instructions AAA and AAS.

```
MOV   AL,'5'      ;AL=35                          0011 0101
ADD   AL,'2'      ;add to AL 32 the ASCII for 2   0011 0010
AAA               ;changes 67H to 07H             0110 0111
OR    AL,30       ;OR AL with 30H to get ASCII
```

If the addition results in a value of more than 9, AAA will correct it and pass the extra bit to carry and add 1 to AH.

```
SUB     AH,AH       ;AH=00
MOV     AL,'7'      ;AL=37H
MOV     BL,'5'      ;BL=35H
ADD     AL,BL       ;37H+35H=6CH therefore AL=6C.
AAA                 ;changes 6CH to 02 in AL and AH=CF=1
OR      AX,3030H    ;AX=3132 which is the ASCII for 12H.
```

Two facts must be noted. First, AAA and AAS work only on the AL register, and second, the data added can be unpacked BCD rather than ASCII, and AAA and AAS will work fine. The following shows how AAS works on subtraction of unpacked BCD to correct the result into unpacked BCD:

```
MOV     AX,105H     ;AX=0105H unpacked BCD for 15
MOV     CL,06       ;CL=06H
SUB     AL,CL       ;5 - 6 = -1 (FFH)
AAS                 ;FFH in AL is adjusted to 09, and
                    ;AH is decremented, leaving AX = 0009
```

Unpacked BCD multiplication and division

There are two instructions designed specifically for multiplication and division of unpacked BCD operands. They convert the result of the multiplication and division to unpacked BCD.

AAM

The Intel manual says that this mnemonic stands for "ASCII adjust multiplication," but it really is unpacked multiplication correction. If two unpacked BCD numbers are multiplied, the result can be converted back to BCD by AAM.

```
MOV     AL,'7'      ;AL=37H
AND     AL,0FH      ;AL=07 unpacked BCD
MOV     DL,'6'      ;DL=36H
AND     DL,0FH      ;DL=06 unpacked BCD
MUL     DL          ;AX=ALxDL. =07x06=002AH=42
AAM                 ;AX=0402 (7x6=42 unpacked BCD)
OR      AX,3030H    ;AX=3432 result in ASCII
```

The multiplication above is byte by byte and the result is HEX. Using AAM converts it to unpacked BCD to prepare it for tagging with 30H to make it ASCII.

AAD

Again, the Intel manual says that AAD represents "ASCII adjust for division," but that can be misleading since the data must be unpacked BCD for this instruction to work. Before dividing the unpacked BCD by another unpacked BCD, AAD is used to convert it to HEX. By doing that the quotient and remainder are both in unpacked BCD.

```
MOV     AX,3539H    ;AX=3539. ASCII for 59
AND     AX,0F0FH    ;AH=05,AL=09 unpacked BCD data
AAD                 ;AX=003BH hex equivalent of 59
MOV     BH,08H      ;divide by 08
DIV     BH          ;3B / 08 gives AL=07 ,AH=03
OR      AX,3030H    ;AL=37H (quotient) AH=33H (rem)
```

As can be seen in the example above, dividing 59 by 8 gives a quotient of 7 and a remainder of 3. With the help of AAD, the result is converted to unpacked BCD, so it can be tagged with 30H to get the ASCII result. It must be noted that both AAM and AAD work only on AX.

Review Questions

1. For the following decimal numbers, give the packed BCD and unpacked BCD representations.
   (a) 15 (b) 99
2. Match the following instruction mnemonic with its function.

_____ DAA	(a) ASCII addition
_____ DAS	(b) ASCII subtraction
_____ AAS	(c) BCD subtraction
_____ AAA	(d) BCD addition

SECTION 3.5: ROTATE INSTRUCTIONS

In many applications there is a need to perform a bitwise rotation of an operand. The rotation instructions ROR, ROL and RCR, RCL are designed specifically for that purpose. They allow a program to rotate an operand right or left. In this section we explore the rotate instructions, which frequently have highly specialized applications. In rotate instructions, the operand can be in a register or memory. If the number of times an operand is to be rotated is more than 1, this is indicated by CL. This is similar to the shift instructions. There are two type of rotations. One is a simple rotation of the bits of the operand, and the other is a rotation through the carry. Each is explained below.

Rotating the bits of an operand right and left

ROR rotate right

In rotate right, as bits are shifted from left to right they exit from the right end (LSB) and enter the left end (MSB). In addition, as each bit exits the LSB, a copy of it is given to the carry flag. In other words, in ROR the LSB is moved to the MSB and is also copied to CF, as shown in the diagram. If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, register CL is used to hold the number of times it is to be rotated.

```
        MOV   AL,36H        ;AL=0011 0110
        ROR   AL,1          ;AL=0001 1011  CF=0
        ROR   AL,1          ;AL=1000 1101  CF=1
        ROR   AL,1          ;AL=1100 0110  CF=1
```

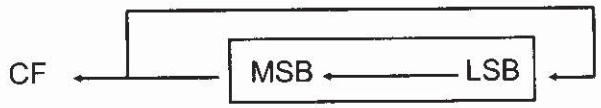
or:

```
        MOV   AL,36H        ;AL=0011 0110
        MOV   CL,3          ;CL=3 number of times to rotate
        ROR   AL,CL         ;AL=1100 0110 CF=1

;the operand can be a word:

        MOV   BX,0C7E5H     ;BX=1100 0111 1110 0101
        MOV   CL,6          ;CL=6 number of times to rotate
        ROR   BX,CL         ;BX=1001 0111 0001 1111 CF=1
```

ROL rotate left



In rotate left, as bits are shifted from right to left they exit the left end (MSB) and enter the right end (LSB). In addition, every bit that leaves the MSB is copied to the carry flag. In other words, in ROL the MSB is moved to the LSB and is also copied to CF, as shown in the diagram. If the operand is to be rotated once, the 1 is coded. Otherwise, the number of times it is to be rotated is in CL.

```
MOV   BH,72H      ;BH=0111 0010
ROL   BH,1        ;BH=1110 0100  CF=0
ROL   BH,1        ;BH=1100 1001  CF=1
ROL   BH,1        ;BH=1001 0011  CF=1
ROL   BH,1        ;BH=0010 0111  CF=1
```

or:

```
MOV   BH,72H      ;BH=0111 0010
MOV   CL,4        ;CL=4 number of times to rotate
ROL   BH,CL       ;BH=0010 0111  CF=1
```

The operand can be a word:

```
MOV   DX,672AH    ;DX=0110 0111 0010 1010
MOV   CL,3        ;CL=3 number of times to rotate
ROL   DX,CL       ;DX=0011 1001 0101 0011 CF=1
```

Write a program that finds the number of 1s in a byte.

```
From the data segment:
DATA1   DB    97H
COUNT   DB    ?
From the code segment:
        SUB   BL,BL       ;clear BL to keep the number of 1s
        MOV   DL,8        ;rotate total of 8 times
        MOV   AL,DATA1
AGAIN:  ROL   AL,1        ;rotate it once
        JNC   NEXT        ;check for 1
        INC   BL          ;if CF=1 then add one to count
NEXT:   DEC   DL          ;go through this 8 times
        JNZ   AGAIN       ;if not finished go back
        MOV   COUNT,BL    ;save the number of 1s
```

Program 3-8

Program 3-8 shows an application of the rotation instruction. The maximum count in Program 3-8 will be 8 since the program is counting the number of 1s in a byte of data. If the operand is a 16-bit word, the number of 1s can go as high as 16. Program 3-9 is Program 3-8, rewritten for a word-sized operand. It also provides the count in BCD format instead of hex. *Reminder:* AL is used to make a BCD counter because the DAA instruction works only on AL.

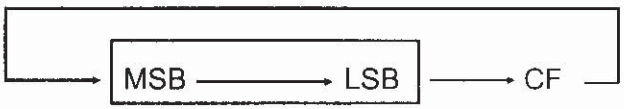
Write a program to count the number of 1s in a word. Provide the count in BCD.

```
DATAW1  DW    97F4H
COUNT2  DB    ?
        ...
        SUB   AL,AL       ;clear AL to keep the number of 1s in BCD
        MOV   DL,16       ;rotate total of 16 times
        MOV   BX,DATAW1   ;move the operand to BX
AGAIN:  ROL   BX,1        ;rotate it once
        JNC   NEXT        ;check for 1.  If CF=0 then jump
        ADD   AL,1        ;if CF=1 then add one to count
        DAA               ;adjust the count for BCD
NEXT:   DEC   DL          ;go through this 16 times
        JNZ   AGAIN       ;if not finished go back
        MOV   COUNT2,AL   ;save the number of 1s in COUNT2
```

Program 3-9

RCR rotate right through carry

In RCR, as bits are shifted from left to right, they exit the right end (LSB) to the carry flag, and the carry flag enters the left end (MSB). In other words, in RCR the LSB is moved to CF and CF is moved to the MSB. In reality, CF acts as if it is part of the operand. This is shown in the diagram. If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, the register CL holds the number of times.




```
        CLC             ;make CF=0
        MOV   AL,26H    ;AL=0010 0110
        RCR   AL,1      ;AL=0001 0011 CF=0
        RCR   AL,1      ;AL=0000 1001 CF=1
        RCR   AL,1      ;AL=1000 0100 CF=1
or:
        CLC             ;make CF=0
        MOV   AL,26H    ;AL=0010 0110
        MOV   CL,3      ;CL=3 number of times to rotate
        RCR   AL,CL     ;AL=1000 0100 CF=1

;the operand can be a word
        STC             ;make CF=1
        MOV   BX,37F1H  ;BX=0011 0111 1111 0001
        MOV   CL,5      ;CL=5 number of times to rotate
        RCR   BX,CL     ;BX=0001 1001 1011 1111 CF=0
```

RCL rotate left through carry

In RCL, as bits are shifted from right to left they exit the left end (MSB) and enter the carry flag, and the carry flag enters the right end (LSB). In other words, in RCL the MSB is moved to CF and CF is moved to the LSB. In reality, CF acts as if it is part of the operand. This is shown in the diagram. If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, register CL holds the number of times.


```
        STC             ;make CF=1
        MOV   BL,15H    ;BL=0001 0101
        RCL   BL,1      ;0010 1011 CF=0
        RCL   BL,1      ;0101 0110 CF=0
or:
        STC             ;make CF=1
        MOV   BL,15H    ;BL=0001 0101
        MOV   CL,2      ;CL=2 number of times for rotation
        RCL   BL,CL     ;BL=0101 0110 CF=0

;the operand can be a word:

        CLC             ;make CF=0
        MOV   AX,191CH  ;AX=0001 1001 0001 1100
        MOV   CL,5      ;CL=5 number of times to rotate
        RCL   AX,CL     ;AX=0010 0011 1000 0001 CF=1
```

Review Questions

1. What is the value of BL after the following?
   ```
   MOV  BL,25H
   MOV  CL,4
   ROR  BL,CL
   ```
2. What are the values of DX and CF after the following?
   ```
   MOV  DX,3FA2H
   MOV  CL,7
   ROL  DX,CL
   ```
3. What is the value of BH after the following?
   ```
   SUB   BH,BH
   STC
   RCR   BH,1
   STC
   RCR   BH,1
   ```
4. What is the value of BX after the following?
   ```
   MOV   BX,FFFFH
   MOV   CL,5
   CLC
   RCL   BX,CL
   ```
5. Why does "ROR BX,4" give an error in the 8086? How would you change the code to make it work?

SECTION 3.6: BITWISE OPERATION IN THE C LANGUAGE

One of the most important and powerful features of the C language is its ability to perform bit manipulation. Due to the fact that many books on C do not cover this important topic, it is appropriate to discuss it in this section. This section describes the action of operators and provides examples.

Table 3-4: Bitwise AND, OR, and EX-OR in C

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise operators in C

While every C programmer is familiar with the logical operators AND (&&), OR (||), and NOT (!), many C programmers are less familiar with the bitwise operators AND (&), OR (|), EX-OR (^), inverter (~), Shift Right (>>), and Shift Left (<<). These bitwise operators are widely used in software engineering and control; consequently, their understanding and mastery are critical in system design and interfacing. See Tables 3-4 and 3-5. The following code shows Examples 3-5 through 3-7 using the C logical operators. Recall that "0x" in the C language indicates that the data is in hex format.

Table 3-5: Bitwise Inverter in C

A	~A
0	1
1	0

```
0x35 & 0x0F = 0x05          /* ANDing: see Example 3-5 */
0x0504 | 0xDA68 = 0xDF6C    /* ORing:  see Example 3-6 */
0x54 ^ 0x78 = 0x2C          /* XORing: see Example 3-7 */
~0x37 = 0xC8                /* inverting 37H */
```

The last one is like the NOT instruction in x86 microprocessors:

```
MOV AL,37H    ;AL=37H
NOT AL        ;AFTER INVERTING 37, AL=C8H
```

Bitwise shift operators in C

There are two bitwise shift operators in C: Shift Right (> >) and Shift Left (<<). They perform exactly the same operation as SHR and SHL in Assembly language, as discussed in Section 3.3. Their format in C is as follows:

```
data >> number of bits to be shifted      /* shifting right */
data << number of bits to be shifted      /* shifting left */
```

The following shows Examples 3-9 through 3-11 using shift operators in C. Program 3-10 shows all of these examples with C syntax.

```
0x9A >> 3   = 0x13          /* shifting right 3 times: see Example 3-9 */
0x7777 >> 4     = 0x0777    /* shifting right 4 times: see Example 3-10 */
0x6 << 4        = 0x60      /* shifting left 4 times: see Example 3-11 */
```

```
/* Program 3-10 Repeats Examples 3-5 through 3-11 in C  */

#include     <stdio.h>
main()

{

 // Notice the way data is defined in C for Hex format using 0x

unsigned char data_1 = 0x35;
unsigned int data_2 = 0x504;
unsigned int data_3 = 0xDA66;
unsigned char data_4= 0x54;
unsigned char data_5=0x78;
unsigned char data_6=0x37;
unsigned char data_7=0x09A;
unsigned char temp;
unsigned int temp_2;

temp=data_1&0x0F;           //ANDing
printf("\nMasking the upper four bits of %X (hex) we get %X (hex)\n",data_1,temp);

temp_2=data_2|data_3;       //ORing
printf("The result of %X hex ORed with %X hex is %X hex\n",data_2,data_3,temp_2);

temp= data_4^data_5;        //EX-ORing
printf("The results of %X hex EX-ORed with %X hex is %X hex\n",data_4,data_5,temp);

temp=~data_6;                 //INVERTING
printf("The result of %X hex inverted is %X hex\n",data_6,temp);

temp=data_7>>3;                         //SHIFTING Right
printf("When %X hex is shifted right three times we get %X hex\n",data_7,temp);

printf("When %X hex is shifted right four times we get %X hex\n",0x7777,0x7777>>4);

temp=(0x6<<4);                          //SHIFTING Left
printf("When %X hex is shifted left %d times we get %X hex\n",0x6,4,temp);

}
```

Program 3-10

Program 3-10 demonstrates the syntax of bitwise operators in C. Next we show some real-world examples of their usage.

Packed BCD to ASCII conversion in C

Section 3.4 showed one way to convert a BCD number to ASCII. This conversion is widely used when dealing with a real-time clock chip. Many of the real-time clock chips provide very accurate time and date for up to ten years without the need for external power. There is a real-time clock in every x86 IBM PC or compatible computer. However, these chips provide the time and date in packed BCD. In order to display the data, it needs to be converted to ASCII. Program 3-11 is a C version of the packed BCD-to-ASCII conversion example discussed in Section 3.4. Program 3-11 converts a byte of packed BCD data into two ASCII characters and displays them using the C bitwise operators.

```
/* Program 3-11 shows packed BCD-to-ASCII conversion using logical bitwise operators in C */

#include    <stdio.h>
main()

{

unsigned char mybcd=0x29;          /* declare a BCD number in hex */
unsigned char asci_1;
unsigned char asci_2;
asci_1=mybcd&0x0f;                 /* mask the upper four bits */
asci_1=asci_1|0x30;                /* make it an ASCII character */
asci_2=mybcd&0xf0;                 /* mask the lower four bits */
asci_2=asci_2>>4;                  /* shift it right 4 times */
asci_2=asci_2|0x30;                /* make it an ASCII character */
printf("BCD data %X is %c , %c in ASCII\n",mybcd,asci_1,asci_2);
printf("My BCD data is %c if not converted to ASCII\n",mybcd);

}
```

Program 3-11

Notice in Program 3-11 that if the packed BCD data is displayed without conversion to ASCII, we get the parenthesis ")". See Appendix F.

Testing bits in C

In many cases of system programming and hardware interfacing, it is necessary to test a given bit to see if it is high. For example, many devices send a high signal to state that they are ready for an action or to indicate that they have data. How can the bit (or bits) be tested? In such cases, the unused bits are masked and then the remaining data is tested. Program 3-12 asks the user for a byte and tests to see whether or not D0 of that byte is high.

```
/* Program 3-12 shows how to test bit D0 to see if it is high */
#include    <stdio.h>
main()

{
unsigned char status;
unsigned char temp;
printf("\nType in a Hex value\n");
scanf("%X",&status);               //get the data
temp=status&0x01;                  //mask all bits except D0
if (temp==0x01)                    //is it high?
  printf("D0 is high");            //if yes, say so
else printf("D0 is low");          //if no, say no
}
```

Program 3-12

The assembly language version of Program 3-12 is as follows:

```
;assume AL=value (in hex)
        AND AL,01          ;MASK ALL BITS EXCEPT D0
        CMP AL,01          ;IS D0 HIGH
        JNE BELOW          ;MAKE A DECISION
        ....               ;YES D0 IS HIGH
BELOW:  ....               ;D0 IS LOW
```

Review Questions

1. What is the result of 0x2F&0x27 ?
2. What is the result of 0x2F|0x27 ?
3. What is the result of 0x2F^0x27 ?
4. What is the result of ~0x2F ?
5. What is the result of 0x2F >> 3 ?
6. What is the result of 0x27 << 4 ?
7. In Program 3-11 if mybcd=0x32, what is displayed if it is not converted to BCD?
8. Modify Program 3-12 to test D3.

SUMMARY

The 8- or 16-bit data items in 80x86 computers can be treated as either signed or unsigned data. Unsigned data uses the entire 8 or 16 bits for data representation. Signed data uses the MSB as a sign bit and the remaining bits for data representation. This chapter covered the arithmetic and logic instructions that are used for unsigned data. The instructions ADD and SUB perform addition and subtraction on unsigned data. Instructions ADC and SBB do the same, but also take the carry flag into consideration. Instructions MUL and DIV perform multiplication and division on unsigned data. Logic instructions AND, OR, XOR, and CMP perform logic operations on all the bits of their operands and were therefore included in this chapter. Shift and rotate instructions for unsigned data include SHR, SHL, ROR, ROL, RCL, and RCR. ASCII and BCD data operations for addition and subtraction were also covered. Finally, bitwise logic instructions were demonstrated using the C language.

PROBLEMS

1. Find CF, ZF, and AF for each of the following. Also indicate the result of the addition and where the result is saved.

   (a)
   ```
   MOV BH,3FH
   ADD BH,45H
   ```
   (b)
   ```
   MOV DX,4599H
   MOV CX,3458H
   ADD CX,DX
   ```
   (c)
   ```
   MOV AX,255
   STC
   ADC AX,00
   ```
   (d)
   ```
   MOV BX,0FF01H
   ADD BL,BH
   ```
   (e)
   ```
   MOV CX,0FFFFH
   STC
   ADC CX,00
   ```
   (f)
   ```
   MOV AH,0FEH
   STC
   ADC AH,00
   ```
2. Write, run, and analyze a program that calculates the total sum paid to a salesperson for eight months. The following are the monthly paychecks for those months: $2300, $4300, $1200, $3700, $1298, $4323, $5673, $986.
3. Rewrite Program 3-2 (in Section 3.1) using byte addition.
4. Write a program that subtracts two multibytes and saves the result. Subtraction should be done a byte at a time. Use the data in Program 3-2.
5. State the three steps involved in a SUB and show the steps for the following data.
   (a) 23H-12H (b) 43H-51H (c) 99-99
6. Write, run, and analyze the result of a program that performs the following:
   (1)(a) byte1 × byte2 (b) byte1 × word1 (c) word1 × word2
   (2) (a) byte1 / byte2 (b) word1 / word2 (c) doubleword/byte1
   Assume byte1=230, byte2=100, word1=9998, word2=300 and doubleword =100000.

7. Assume that the following registers contain these HEX contents: AX = F000, BX = 3456, and DX = E390. Perform the following operations. Indicate the result and the register where it is stored. Give also ZF and CF in each case.
Note: the operations are independent of each other.

(a) AND DX,AX
(b) OR DH,BL
(c) XOR AL,76H
(d) AND DX,DX
(e) XOR AX,AX
(f) OR BX,DX
(g) AND AH,OFF
(h) OR AX,9999H
(i) XOR DX,0EEEEH
(j) XOR BX,BX
(k) MOV CL,04
SHL AL,CL
(l) SHR DX,1
(m) MOV CL,3
SHR DL,CL
(n) MOV CL,5
SHL BX,CL
(o) MOV CL,6
SHL DX,CL

8. Indicate the status of ZF and CF after CMP is executed in each of the following cases.

(a) MOV BX,2500
CMP BX,1400
(b) MOV AL,0FFH
CMP AL,6FH
(c) MOV DL,34
CMP DL,88
(d) SUB AX,AX
CMP AX,0000
(e) XOR DX,DX
CMP DX,0FFFFH
(f) SUB CX,CX
DEC CX
CMP CX,0FFFFH
(g) MOV BX,2378H
MOV DX,4000H
CMP DX,BX
(h) MOV AL,0AAH
AND AL,55H
CMP AL,00

9. Indicate whether or not the jump happens in each case.

(a) MOV CL,5
SUB AL,AL
SHL AL,CL
JNC TARGET
(b) MOV BH,65H
MOV AL,48H
OR AL,BH
SHL AL,1
JC TARGET
(c) MOV AH,55H
SUB DL,DL
OR DL,AH
MOV CL,AH
AND CL,0FH
SHR DL,CL
JNC TARGET

10. Rewrite Program 3-3 to find the lowest grade in that class.
11. Rewrite Program 3-4 to convert all uppercase letters to lowercase.
12. In the IBM BIOS program for testing flags and registers, verify every jump (conditional and unconditional) address calculation. Reminder: As mentioned in Chapter 2, in forward jumps the target address is calculated by adding the displacement value to IP of the instruction after the jump and by subtracting in backward jumps.
13. In Program 3-6 rewrite BCD_ADD to do subtraction of the multibyte BCD.
14. Rewrite Program 3-7 to subtract DATA2 from DATA1. Use the following data.

```
DATA1    DB  '0999999999'
DATA2    DB  '0077777775'
```

15. Using the DT directive, write a program to add two 10-byte BCD numbers.
16. We would like to make a counter that counts up from 0 to 99 in BCD. What instruction would you place in the dotted area?

```
    SUB  AL,AL
    ADD  AL,1
    .... ....
```

17. Write Problem 16 to count down (from 99 to 0).
18. An instructor named Mr. Mo Allem has the following grading policy: "Curving of grades is achieved by adding to every grade the difference between 99 and the highest grade in the class." If the following are the grades of the class, write a program to calculate the grades after they have been curved: 81, 65, 77, 82, 73, 55, 88, 78, 51, 91, 86, 76. Your program should work for any set of grades.
19. If we try to divide 1,000,000 by 2:
(a) What kind of problem is associated with this operation in 8086/286 CPUs?
(b) How does the CPU let us know that there is a problem?

20. Which of the following groups of code perform the same operation as LOOP XXX?

    (a) DEC CL
        JNZ XXX

    (b) DEC CH
        JNZ XXX

    (c) DEC BX
        JNZ

    (d) DEC CX
        JNZ XXX

21. Write a program that finds the number of zeros in a 16-bit word.
22. In Program 3-2, which demonstrated multiword addition, pointers were updated by two INC instructions instead of "ADD SI,2". Why?
23. Write a C program with the following components:
    (a) have two hex values: data1=55H and data2=AAH; both defined as unsigned char,
    (b) mask the upper 4 bits of data1 and display it in hex,
    (c) perform AND, OR, and EX-OR operations between the two data items and then display each result,
    (d) invert one and display it,
    (e) shift left data1 four times and shift right data2 two times, then display each result.
24. Repeat the above problem with two values input from the user. Use the scanf("%X") function to get the hex data.
25. In the same way that the real-time clock chip provides data in BCD, it also expects data in BCD when it is being initialized. However, data coming from the keyboard is in ASCII. Write a C program to convert two ASCII bytes of data to packed BCD.
26. Write a C program in which the user is prompted for a hex value. Then the data is tested to see if the two least significant bits are high. If so, a message states "D0 and D1 are both high"; otherwise, it states which bit is not high.
27. Repeat the above problem for bits D0 and D7.

ANSWERS TO REVIEW QUESTIONS

SECTION 3.1: UNSIGNED ADDITION AND SUBTRACTION

1. destination
2. in 80x86 Assembly language, there are no memory to memory operations
3. MOV AX,DATA_2
   ADD DATA_1,AX
4. destination, source + destination + CF
5. in (a), the byte addition results in a carry to CF, in (b), the word addition results in a carry to the high byte BH
6. DEC CX
   JNZ ADD_LOOP
7. 43H 0100 0011 0100 0011
   - 05H 0000 0101 2's complement=+1111 1011
   3EH 0011 1110
   CF=0; therefore, the result is positive
8. AL = 95 - 4F - 1 = 45

SECTION 3.2: UNSIGNED MULTIPLICATION AND DIVISION

1. AX 2. DX and AX 3. AX 4. AL, AH
5. AX, DX 6. AL, AH 7. AX, DX

SECTION 3.3: LOGIC INSTRUCTIONS AND SAMPLE PROGRAMS

1. (a) 4202 (b) CFFF (c) 8DFD
2. the operand will remain unchanged; all zeros 3. all ones 4. all zeros
5. A0F2 = 1010 0000 1111 0010
   shift left: 0100 0001 1110 0100 CF =1
   shift again: 1000 0011 1100 1000 CF =0
   shift again: 0000 0111 1001 0000 CF =1
   A0F2 shifted left three times = 0790.
   A0F2 = 1010 0000 1111 0010
   shift right: 0101 0000 0111 1001 CF = 0
   shift again: 0010 1000 0011 1100 CF = 1
   shift again: 0001 0100 0001 1110 CF = 0
   A0F2 shifted right three times = 141E
6. SUB 7. false

SECTION 3.4: BCD AND ASCII OPERANDS AND INSTRUCTIONS

1. (a) 15 = 0001 0101 packed BCD = 0000 0001 0000 0101 unpacked BCD
   (b) 99 = 1001 1001 packed BCD = 0000 1001 0000 1001 unpacked BCD
2. DAA -- BCD addition; DAS -- BCD subtraction; AAS -- ASCII subtraction; AAA -- ASCII addition

SECTION 3.5: ROTATE INSTRUCTIONS

1. BL = 52H, CF = 0
2. DX = D11FH, CF = 1
3. BH = C0H
4. BX = FFEFH
5. the source operand cannot be immediate; to fix it:

```
MOV  CL,4
ROR  BX,CL
```

SECTION 3.6: BITWISE OPERATION IN THE C LANGUAGE

1. 0x27
2. 0x2F
3. 0x08
4. 0xD0
5. 0x05
6. 0x70
7. 2
8.

```
/* This program shows how to test Bit D3 to see if it is high */
#include   <stdio.h>
main()
{
unsigned char status;
unsigned char temp;
printf("\nType in a Hex value\n");
scanf("%X",&status);
temp=status&0x04;
if (temp==0x04)
  printf("D3 is high");
else printf("D3 is low");
}
```