Use the Workbook Now

Welcome to the Assembly Language Workbook, written by Kip R. Irvine to serve as a supplement to *Assembly Language for Intel-Based Computers* (Prentice-Hall). By combining my book with the workbook exercises, you should have an even greater chance of success in your Assembly Language course. Of course, there is still no substitute for having a knowledgeable, helpful instructor when you are learning a programming language. The lessons are placed in a more-or-less logical order from easy to difficult. For example, you should start with the following topics:

- Binary and Hexadecimal Numbers
- Signed Integers
- Floating-Point Binary
- Register and Immediate Operands
- Addition and Subtraction Instructions

Many of the topics begin with a tutorial and are followed by a set of related exercises. Each exercise page is accompanied by a corresponding page with all of the answers. Of course, you should try to do the exercises first, without looking at the answers!

In addition to the tutorials found here, you may want to look at the Supplemental Articles page on this Web site.

If you think you've found a mistake, verify it with your instructor, and if it needs correcting, post a message to the book's discussion group.

Download the workbook as an Adobe Acrobat (PDF) file (1/15/2003)
1. Binary and Hexadecimal Integers
2. Signed Integers (tutorial)
3. Signed Integers
4. Floating-Point Binary
5. Register and Immediate Operands
6. Addition and Subtraction Instructions
7. Direct Memory Operands
8. Indirect and Indexed Operands
9. Mapping Variables to Memory
10. MS-DOS Function Calls, Part 1
11. MS-DOS Function Calls, Part 2
12. Error-Correcting Codes
13. Boolean and Comparison Instructions
14. Decoding a 12-bit FAT (tutorial)
Binary and Hexadecimal Integers

Click here to view the answers.

1. Write each of the following decimal numbers in binary:
   a. 2   g. 15
   b. 7   h. 16
   c. 5   i. 20
   d. 8   j. 27
   e. 9   k. 32
   f. 12   l. 64

2. Write each of the following binary numbers in decimal:
   a. 00000101   g. 00110000
   b. 00001111   h. 00100111
   c. 00010000   i. 01000000
   d. 00010110   j. 01100011
   e. 00010101   k. 10100000
   f. 00011100   l. 10101010

3. Write each of the following binary numbers in hexadecimal:
   a. 00000101   g. 00110000
   b. 00001111   h. 00100111
   c. 00010000   i. 01000000
   d. 00010110   j. 01100011
   e. 00001011   k. 10100000
   f. 00011100   l. 10101011

4. Write each of the following hexadecimal numbers in binary:
   a. 0005h   g. 0030h
   b. 000Fh   h. 0027h
   c. 0010h   i. 0048h
   d. 0016h   j. 0063h
   e. 000Bh   k. A064h
   f. 001Ch   l. ABDEh

5. Write each of the following hexadecimal numbers in decimal:
   a. 00D5h   g. 0B30h
   b. 002Fh   h. 06DFh
   c. 0110h   i. 1AB6h
   d. 0216h   j. 0A63h
   e. 004Bh   k. 02A0h
   f. 041Ch   l. 1FABh
Tutorial: Signed Integers

In mathematics, the additive inverse of a number $n$ is the value, when added to $n$, produces zero. Here are a few examples, expressed in decimal:

$6 + -6 = 0$
$0 + 0 = 0$
$-1 + 1 = 0$

Programs often include both subtraction and addition operations, but internally, the CPU really only performs addition. To get around this restriction, the computer uses the additive inverse. When subtracting $A - B$, the CPU instead performs $A + (-B)$. For example, to simulate the subtraction of 4 from 6, the CPU adds $-4$ to 6:

$6 + -4 = 2$

**Binary Two’s Complement**

When working with binary numbers, we use the term two’s complement to refer to a number’s additive inverse. The two’s complement of a number $n$ is formed by reversing $n$’s bits and adding 1. Here, for example, $n$ equals the 4-bit number 0001:

\[
\begin{align*}
N: & \quad 0001 \\
Reverse\ N: & \quad 1110 \\
Add\ 1: & \quad 1111
\end{align*}
\]

The two’s complement of $n$, when added to $n$, produces zero:

\[
0001 + 1111 = 0000
\]

It doesn’t matter how many bits are used by $n$. The two’s complement is formed using the same method:

\[
\begin{align*}
N = 1 & \quad 00000001 \\
Reverse\ N: & \quad 11111110 \\
Add\ 1: & \quad 11111111
\end{align*}
\]

\[
\begin{align*}
N = 1 & \quad 0000000000000001 \\
Reverse\ N: & \quad 1111111111111110 \\
Add\ 1: & \quad 1111111111111111
\end{align*}
\]

Here are some examples of 8-bit two’s complements:

<table>
<thead>
<tr>
<th>n(decimal)</th>
<th>n(binary)</th>
<th>NEG(n)</th>
<th>(decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2</td>
<td>00000010</td>
<td>11111110</td>
<td>-2</td>
</tr>
<tr>
<td>+16</td>
<td>00010000</td>
<td>11110000</td>
<td>-16</td>
</tr>
<tr>
<td>+127</td>
<td>01111111</td>
<td>10000001</td>
<td>-127</td>
</tr>
</tbody>
</table>
Signed Integers

Click here to view a tutorial that helps to clarify the representation of signed integers using two's complement notation. Click here to view the answers.

1. Write each of the following signed decimal integers in 8-bit binary notation:
   If any number cannot be represented as a signed 8-bit binary number, indicate this in your answer.
   
   a. -2   e. +15
   b. -7   f. -1
   c. -128  g. -56
   d. -16  h. +127

2. Write each of the following 8-bit signed binary integers in decimal:

   a. 11111111  g. 00001111
   b. 11110000  h. 10101111
   c. 10000000  i. 11111100
   d. 10000001  j. 01010101

3. Which of the following integers are valid 16-bit signed decimal integers?
   (indicate V=valid, I=invalid)

   a. +32469  d. +32785
   b. +32767  e. 32785
   c. -32768  f. +65535

4. Indicate the sign of each of the following 16-bit hexadecimal integers:
   (indicate P=positive, N=negative)

   a. 7FB9h  c. 0D000h
   b. 8123h  d. 649Fh

5. Write each of the following signed decimal integers as a 16-bit hexadecimal value:

   a. -42  e. -32768
   b. -127  f. -1
   c. -4096  g. -8193
   d. -16  h. -256
Floating-Point Binary Representation

Updated 9/30/2002

Click here to view the answers

1. For each of the following binary floating-point numbers, supply the equivalent value as a base 10 fraction, and then as a base 10 decimal. The first problem has been done for you:

<table>
<thead>
<tr>
<th>Binary Floating-Point</th>
<th>Base 10 Fraction</th>
<th>Base 10 Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.101 (sample)</td>
<td>1 5/8</td>
<td>1.625</td>
</tr>
<tr>
<td>11.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101.001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101.0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110.00111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000.101011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111.0000011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.000101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. For each of the following exponent values, shown here in decimal, supply the actual binary bits that would be used for an 8-bit exponent in the IEEE Short Real format. The first answer has been supplied for you:

<table>
<thead>
<tr>
<th>Exponent (E)</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 (sample)</td>
<td>10000001</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

3. For each of the following floating-point binary numbers, supply the normalized value and the resulting exponent. The first answer has been supplied for you:

<table>
<thead>
<tr>
<th>Binary Value</th>
<th>Normalized As</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000.11 (sample)</td>
<td>1.000011</td>
<td>4</td>
</tr>
<tr>
<td>1101.101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000011.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.0000011001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. For each of the following floating-point binary examples, supply the complete binary representation of the number in IEEE Short Real format. The first answer has been supplied for you:

<table>
<thead>
<tr>
<th>Binary Value</th>
<th>Sign, Exponent, Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.11 (sample)</td>
<td>1 01111111 11000000000000000000000</td>
</tr>
<tr>
<td>+1101.101</td>
<td></td>
</tr>
<tr>
<td>-.00101</td>
<td></td>
</tr>
<tr>
<td>+100111.0</td>
<td></td>
</tr>
<tr>
<td>+.00000011101011</td>
<td></td>
</tr>
</tbody>
</table>
Register and Immediate Operands

This topic covers the MOV instruction, applied to register and immediate operands. Click here to view the answers.

1. Indicate whether or not each of the following MOV instructions is valid:
   (notate: V = valid, I = invalid)

   a. mov ax, bx  g. mov al, dh
   b. mov dx, bl  h. mov ax, dh
   c. mov ecx, edx  i. mov ip, ax
   d. mov si, di  j. mov si, cl
   e. mov ds, ax  k. mov edx, ax
   f. mov ds, es  l. mov ax, es

2. Indicate whether or not each of the following MOV instructions is valid:
   (notate: V = valid, I = invalid)

   a. mov ax, 16  g. mov 123, dh
   b. mov dx, 7F65h  h. mov ss, ds
   c. mov ecx, 6F23458h  i. mov 0FABh, ax
   d. mov si, -1  j. mov si, cl
   e. mov ds, 1000h  k. mov edx, esi
   f. mov al, 100h  l. mov edx, -2
Addition and Subtraction Instructions

This topic covers the ADD, SUB, INC, and DEC instructions, applied to register and immediate operands. Click here to view the answers.

1. Indicate whether or not each of the following instructions is valid. (note: V = valid, I = invalid) Assume that all operations are unsigned.

   a. add ax,bx
   b. add dx,bl
   c. add ecx,dx
   d. sub si,di
   e. add bx,90000
   f. sub ds,1
   g. dec ip
   h. dec edx
   i. add edx,1000h
   j. sub ah,126h
   k. sub al,256
   l. inc ax,1

2. What will be the value of the Carry flag after each of the following instruction sequences has executed? (note: CY = carry, NC = no carry)

   a. mov ax,0FFFFh
      add ax,1
   b. mov bh,2
      sub bh,2
   c. mov dx,0
      dec dx
   d. mov al,0DFh
      add al,32h
   e. mov si,0B9F6h
      sub si,9874h
   f. mov cx,695Fh
      sub cx,A218h

3. What will be the value of the Zero flag after each of the following instruction sequences has executed? (note: ZR = zero, NZ = not zero)

   a. mov ax,0FFFFh
      add ax,1
   b. mov bh,2
      sub bh,2
   c. mov dx,0
      dec dx
   d. mov al,0DFh
      add al,32h
   e. mov
4. What will be the value of the Sign flag after each of the following instruction sequences has executed?
   (note: PL = positive, NG = negative)
   
   a. mov
     ax,0FFFFh
     sub ax,1
   b. mov
     bh,2
     sub bh,3
   c. mov
     dx,0
     dec dx
   d. mov
     ax,7FFEh
     add ax,22h
   e. mov
     si,0B9F6h
     sub si,9874h
   f. mov
     cx,8000h
     add cx,A69Fh

5. What will be the values of the Carry, Sign, and Zero flags after the following instructions have executed?
   (note: CY/NC, PL/NG, ZR/NZ)
   
   mov
   ax,620h
   sub
   ah,0F6h

6. What will be the values of the Carry, Sign, and Zero flags after the following instructions have executed?
   (note: CY/NC, PL/NG, ZR/NZ)
   
   mov
   ax,720h
   sub
   ax,0E6h

7. What will be the values of the Carry, Sign, and Zero flags after the following instructions have executed?
   (note: CY/NC, PL/NG, ZR/NZ)
   
   mov
   ax,0B6D4h
   add
   al,0B3h

8. What will be the values of the Overflow, Sign, and Zero flags after the following instructions have executed?
   (note: OV/NV, PL/NG, ZR/NZ)
   
   mov
9. What will be the values of the Carry, Overflow, Sign, and Zero flags after the following instructions have executed?

(notate: CY/NC, OV/NV, PL/NG, ZR/NZ)

```
bl,-
127
dec
bl
```

10. What will be the values of the Carry, Overflow, Sign, and Zero flags after the following instructions have executed?

(notate: CY/NC, OV/NV, PL/NG, ZR/NZ)

```
mov
  cx,-
4097
add
cx,1001h
```
Direct Memory Operands

Updated 9/30/2002

This topic covers the MOV instruction, applied to direct memory operands and operands with displacements. Click here to view the answers.

Use the following data declarations for Questions 1-4. Assume that the offset of byteVal is 00000000h, and that all code runs in Protected mode.

.data
byteVal  BYTE 1,2,3,4
wordVal  WORD 1000h,2000h,3000h,4000h
dwordVal DWORD 12345678h,34567890h
aString  BYTE "ABCDEFG",0

1. Indicate whether or not each of the following MOV instructions is valid:
   (notate: V = valid, I = invalid)
   
   a. mov ax,byteVal
   b. mov dx,wordVal
   c. mov ecx,dwordVal
   d. mov si,aString
   e. mov esi,offset aString
   f. mov al,byteVal

2. Indicate whether or not each of the following MOV instructions is valid:
   (notate: V = valid, I = invalid)
   
   mov
   a. mov eax,offset byteVal
   b. mov dx,wordVal+2
   c. mov ecx,offset dwordVal
   d. mov si,dwordVal
   e. mov esi,offset aString+2
   f. mov al,offset byteVal+1

3. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
   (If any instruction is invalid, indicate "I" as the answer.)
   
   mov
   a. mov eax,offset byteVal
   b. mov dx,wordVal
4. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal+2
b. dx, wordVal+4
c. ecx, dwordVal+4
d. esi, offset wordVal+4
e. esi, offset aString-1
```

5. Use the following data declarations for Questions 5-6. Assume that the offset of byteVal is 0000:

```
.data
byteVal   BYTE 3 DUP(0FFh),2,"XY"
wordVal   WORD 2 DUP(6),2
dwordVal  DWORD 8,7,6,5
dwordValSiz WORD ($ - dwordVal)
ptrByte   DWORD byteVal
ptrWord   DWORD wordVal
```

5. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
    a. mov eax, offset wordVal
    b. mov dx, wordVal+4
    c. mov ecx, dwordVal+4
    d. mov si, dwordValSiz
    e. mov al, byteVal+4
```

6. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ax, dwordVal+2
b. dx, wordVal-2
c. eax, ptrByte
    mov
d. esi, ptrWord
```

7. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

8. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

9. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

10. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

11. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

12. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

13. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

14. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

15. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

16. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

17. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

18. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

19. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ebx, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```

20. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset byteVal
b. edx, offset wordVal
    mov
c. ecx, offset aString
    mov
d. esi, offset dwordVal
```
e. edi, offset
dwordVal + 2
Indirect and Indexed Operands

This topic covers the MOV instruction, applied to indirect, based, and indexed memory operands. Click here to view the answers.

Use the following data declarations. Assume that the offset of byteVal is 0000:

```
data
byteVal  db 1,2,3,4
wordVal dw 1000h,2000h,3000h,4000h
dwordVal dd 12345678h,34567890h
aString db "ABCDEFG",0
pntr   dw wordVal
```

1. Indicate whether or not each of the following instructions is valid:

   (notate: V = valid, I = invalid)

   a. mov
      
      ax,byteVal[si]
   b. add
      
      dx,[cx+wordVal]
   c. mov
      
      ecx,[edi+dwordVal]
   d. xchg al,[bx]
   e. mov ax,[bx+4]
   f. mov [bx],[si]
   g. xchg
      
      al,byteVal[dx]

2. Indicate the hexadecimal value of the final destination operand after each of the following code fragments has executed:

   (If any instruction is invalid, indicate "I" as the answer.)

   a. mov si,offset
      
      byteVal
      mov al,[si+1]
   b. mov di,6
      
      mov
      
      dx,wordVal[di]
   c. mov bx,4
      
      mov
      
      ecx,[bx+dwordVal]
   d. mov si,offset
      
      aString
      mov al,byteVal+1
      mov [si],al
   e. mov si,offset
      
      aString+2
      inc byte ptr
      
      [si]
   f. mov bx,pntr
      
      add word ptr
      
      [bx],2
   g. mov di,offset
      
      pntr
      mov si,[di]
      mov ax,[si+2]

3. Indicate the hexadecimal value of the final destination operand after each of the following code fragments has executed:

   (If any instruction is invalid, indicate "I" as the answer.)
a. xchg
    si, pntr
    xchg
    [si], wordVal
b. mov
    ax, pntr
    xchg ax, si
    mov
    dx, [si+4]
c. mov edi, 0
    mov di, pntr
    add edi, 8
    mov
    eax, [edi]
d. mov
    esi, offset
    aString
    xchg
    esi, pntr
    mov
    dl, [esi]
e. mov
    esi, offset
    aString
    mov
    dl, [esi+2]
Mapping Variables to Memory

When you're trying to learn how to address memory, the first challenge is to have a clear mental picture of the storage (the mapping) of variables to memory locations.

Use the following data declarations, and assume that the offset of arrayW is 0000:

```assembly
.data
arrayW  WORD  1234h,5678h,9ABCh
ptr1    WORD  offset arrayD
arrayB  BYTE  10h,20h,30h,40h
arrayD  DWORD  40302010h
```

Click here to view a memory mapping table (GIF). Right-click here to download the same table as an Adobe Acrobat file. Print this table, and fill in the hexadecimal contents of every memory location with the correct 32-bit, 16-bit, and 8-bit values.
Required reading: Chapter 13

1. Write a program that inputs a single character and redisplay (echoes) it back to the screen. *Hint:* Use INT 21h for the character input. [Solution program]

2. Write a program that inputs a string of characters (using a loop) and stores each character in an array. Using CodeView, display a memory window containing the array. [Solution program]

(Contents of memory window after the loop executes:)

```
000A 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D ABCDEFGHIJKLM
0017 4E 4F 50 51 52 53 54 00 4E 4E 20 38 NOPQRST.NN08
```

3. Using the array created in the previous question, redisplay the array on the screen. [Solution program]

4. Write a program that reads a series of ten lowercase letters from input (without displaying it), converts each character to uppercase, and then displays the converted character. [Solution program]

5. Write a program that displays a string using INT 21h function 9. [Solution program]
Required reading: Chapter 13

1. Write a program that inputs a string using DOS function 0Ah. Limit the input to ten characters. Redisplay the string backwards. Solution program.

2. Write a program that inputs a string of up to 80 characters using DOS function 3Fh. After the input, display a count on the screen of the actual number of characters typed by the user. Solution program.

3. Write a program that inputs the month, day, and year from the user. Use the values to set the system date with DOS function 2Bh. Hint: Use the Readint function from the book's link library to input the integer values. (Under Windows NT/2000, you must have administrator privileges to run this program.) Solution program.

4. Write a program that uses DOS function 2Ah to get and display the system date. Use the following display format: yyyy-m-d. Solution program.
Error Correcting Codes

Even and Odd Parity

If a binary number contains an even number of 1 bits, we say that it has even parity. If the number contains an odd number of 1 bits, it has odd parity.

When data must be transmitted from one device to another, there is always the possibility that an error might occur. Detection of a single incorrect bit in a data word can be detected simply by adding an additional parity bit to the end of the word. If both the sender and receiver agree to use even parity, for example, the sender can set the parity bit to either 1 or zero so as to make the total number of 1 bits in the word an even number:

8-bit data value: 1 0 1 1 0 1 0 1
added parity bit: 1
transmitted data: 1 0 1 1 0 1 0 1 1

Or, if the data value already had an even number of 1 bits, the parity bit would be set to 0:

8-bit data value: 1 0 1 1 0 1 0 0
added parity bit: 0
transmitted data: 1 0 1 1 0 1 0 0 0

The receiver of a transmission also counts the 1 bits in the received value, and if the count is not even, an error condition is signalled and the sender is usually instructed to re-send the data. For small, non-critical data transmissions, this method is a reasonable tradeoff between reliability and efficiency. But it presents problems in cases where highly reliable data must be transmitted.

The primary problem with using a single parity bit is that it cannot detect the presence of more than one transmission error. If two bits are incorrect, the parity can still be even and no error can be detected. In the next section we will look at an encoding method that can both detect multiple errors and can correct single errors.

Hamming Code

In 1950, Richard Hamming developed an innovative way of adding bits to a number in such a way that transmission errors involving no more than a single bit could be detected and corrected.

The number of parity bits depends on the number of data bits:

<table>
<thead>
<tr>
<th>Data Bits</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity Bits:</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Codeword :</td>
<td>7</td>
<td>12</td>
<td>21</td>
<td>38</td>
<td>71</td>
<td>136</td>
</tr>
</tbody>
</table>

We can say that for N data bits, \( \log_2 N + 1 \) parity bits are required. In other words, for a data of size \( 2^n \) bits, \( n+1 \) parity bits are embedded to form the codeword. It's interesting to note that doubling the number of data bits results in the addition of only 1 more data bit. Of course, the longer the codeword, the greater the chance that more than error might occur.

Placing the Parity Bits

(From this point onward we will number the bits from left to right, beginning with 1. In other words, bit 1 is the most significant bit.)

The parity bit positions are powers of 2: \( \{1,2,4,8,16,32\ldots\} \). All remaining positions hold data bits. Here is a table representing a 21-bit code word:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|

The 16-bit data value 100011100100101 would be stored as follows:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| P | P | P | 0 | 0 | 0 | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |

Calculating Parity

For any data bit located in position N in the code word, the bit is checked by parity bits in positions \( P_1, P_2, P_3, \ldots, P_k \) if \( N \) is equal to the sum of \( P_1, P_2, P_3, \ldots, P_k \). For example, bit 11 is checked by parity bits 1, 2 and 8 (11 = 1 + 2 + 8). Here is a table covering code words up to 21 bits.
Data Bit ...is checked by parity bits

<table>
<thead>
<tr>
<th>Data Bit</th>
<th>Checks the following Data Bits</th>
<th>Hint*</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1, 4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2, 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1, 2, 4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1, 8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2, 8</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4, 8</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1, 4, 8</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2, 4, 8</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1, 2, 4, 8</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1, 16</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2, 16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1, 2, 16</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4, 16</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1, 4, 16</td>
<td></td>
</tr>
</tbody>
</table>

Turning this data around in a more useful way, the following table shows exactly which data bits are checked by each parity bit in a 21-bit code word:

<table>
<thead>
<tr>
<th>Parity Bit</th>
<th>Checks the following Data Bits</th>
<th>Hint*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21</td>
<td>use 1, skip 1, use 1, skip 1, ...</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 6, 7, 10, 11, 14, 15, 18, 19</td>
<td>use 2, skip 2, use 2, skip 2, ...</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6, 7, 12, 13, 14, 15, 20, 21</td>
<td>use 4, skip 4, use 4, ...</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10, 11, 12, 13, 14, 15</td>
<td>use 8, skip 8, use 8, ...</td>
</tr>
<tr>
<td>16</td>
<td>16, 17, 18, 19, 20, 21</td>
<td>use 16, skip 16, ...</td>
</tr>
</tbody>
</table>

It is useful to view each row in this table as a bit group. As we will see later, error correcting using the Hamming encoding method is based on the intersections between these groups, or sets, of bits.

* Some of the hints (3rd column) only make sense for larger code words.

## Encoding a Data Value

Now it's time to put all of this information together and create a code word. We will use even parity for each bit group, which is an arbitrary decision. We might just as easily have decided to use odd parity. For the first example, let's use the 8-bit data value 1 1 0 0 1 1 1 1, which will produce a 12-bit code word. Let's start by filling in the data bits:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, we begin calculating and inserting each of the parity bits.

**P1:** To calculate the parity bit in position 1, we sum the bits in positions 3, 5, 7, 9, and 11: (1+1+0+1+1 = 4). This sum is even (indicating even parity), so parity bit 1 should be assigned a value of 0. By doing this, we allow the parity to remain even:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>P</td>
<td></td>
<td>P</td>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>1 1</td>
</tr>
</tbody>
</table>

**P2:** To generate the parity bit in position 2, we sum the bits in positions 3, 6, 7, 10, and 11: (1+0+0+1+1 = 3). The sum is odd, so we assign a value of 1 to parity bit 2. This produces even parity for the combined group of bits 2, 3, 6, 7, 10, and 11:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P</td>
<td>P</td>
<td>1</td>
<td>P</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

**P4:** To generate the parity bit in position 4, we sum the bits in positions 5, 6, 7, and 12: (1+0+0+1 = 2). This results in even parity, so we set parity bit 4 to zero, leaving the parity even:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

**P8:** To generate the parity bit in position 8, we sum the bits in positions 9, 10, 11 and 12: (1+1+1+1 = 4). This results in even parity, so we set parity bit 8 to zero, leaving the parity even:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>
Detecting a Single Error

When a code word is received, the receiver must verify the correctness of the data. This is accomplished by counting the 1 bits in each bit group (mentioned earlier) and verifying that each has even parity. Recall that we arbitrarily decided to use even parity when creating code words. Here are the bit groups for a 12-bit code value:

<table>
<thead>
<tr>
<th>Parity Bit</th>
<th>Bit Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 5, 7, 9, 11</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 6, 7, 10, 11</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6, 7, 12</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10, 11, 12</td>
</tr>
</tbody>
</table>

If one of these groups produces an odd number of bits, the receiver knows that a transmission error occurred. As long as only a single bit was altered, it can be corrected. The method can be best shown using concrete examples.

Example 1: Suppose that the bit in position 4 was reversed, producing 01110001111. The receiver would detect an odd parity in the bit group associated with parity bit 4. After eliminating all bits from this group that also appear in other groups, the only remaining bit is bit 4. The receiver would toggle this bit, thus correcting the transmission error.

Example 2: Suppose that bit 7 was reversed, producing 011010101111. The bit groups based on parity bits 1, 2, and 4 would have odd parity. The only bit that is shared by all three groups (the intersection of the three sets of bits) is bit 7, so again the error bit is identified:

<table>
<thead>
<tr>
<th>Parity Bit</th>
<th>Bit Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 5, 7, 9, 11</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 6, 7, 10, 11</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6, 7, 12</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10, 11, 12</td>
</tr>
</tbody>
</table>

Example 3: Suppose that bit 6 was reversed, producing 011011001111. The groups based on parity bits 2 and 4 would have odd parity. Notice that two bits are shared by these two groups (their intersection): 6 and 7:

<table>
<thead>
<tr>
<th>Parity Bit</th>
<th>Bit Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 5, 7, 9, 11</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 6, 7, 10, 11</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6, 7, 12</td>
</tr>
<tr>
<td>8</td>
<td>8, 9, 10, 11, 12</td>
</tr>
</tbody>
</table>

But then, but 7 occurs in group 1, which has even parity. This leaves bit 6 as the only choice as the incorrect bit.

Multiple Errors

If two errors were to occur, we could detect the presence of an error, but it would not be possible to correct the error. Consider, for example, that both bits 5 and 7 were incorrect. The bit groups based on parity bit 2 would have odd parity. Groups 1 and 4, on the other hand, would have even parity because bits 5 and 7 would counteract each other:

<table>
<thead>
<tr>
<th>Parity Bit</th>
<th>Bit Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 5, 7, 9, 11</td>
</tr>
<tr>
<td>2</td>
<td>2, 3, 6, 7, 10, 11</td>
</tr>
<tr>
<td>4</td>
<td>4, 5, 6, 7, 12</td>
</tr>
</tbody>
</table>

This would incorrectly lead us to the conclusion that bit 2 is the culprit, as it is the only bit that does not occur in groups 1 and 4. But toggling bit 2 would not to fix the error--it would simply make it worse.


If you would like to learn how to construct your own error-correcting codes, here is a good explanation of the mathematics: Laufer, Henry B. Discrete Mathematics and Applied Modern Algebra. Chapter 1: Group Codes. Prindle, Weber & Schmidt, 1984.
Boolean and Comparison Instructions

AND and OR Instructions

1. Write instructions that jump to a label named Target if bits 0, 1, and 2 in the AL register are all set (the remaining bits are unimportant).
2. Write instructions that will jump to a label named Target if either bit 0, 1, or 2 is set in the AL register (the remaining bits are unimportant).
3. Clear bits 4-6 in the BL register without affecting any other bits.
4. Set bits 3-4 in the CL register without affecting any other bits.
Decoding a 12-bit File Allocation Table

In this section we present a simple program that loads the file allocation table and root directory from a diskette (in drive A), and displays the list of clusters owned by each file. Let's look at part of a sample 12-bit FAT in raw form (shown by Debug) so we can decode its structure:

```
F0 FF FF FF 4F 00 05 60-00 07 80 00 09 A0 00 0B
C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60
```

A decoded form of entries 2 through 9 is shown here:

Entry: 2 3 4 5 6 7 8 9 ...  
Value: <FFF> <004> <005> <006> <007> <008> <009> ...

You can track down all clusters allocated to a particular file by following what is called a cluster chain. Let's follow the cluster chain starting with cluster 3. Here is how we find its matching entry in the FAT, using three steps:

1. Divide the cluster number by 2, resulting in an integer quotient. Add the same cluster number to this quotient, producing the offset of the cluster's entry in the FAT. Using cluster 3 as a sample, this results in Int(3 /2) + 3 = 4, so we look at offset 4 in the FAT.
2. The 16-bit word at offset 4 contains 004Fh (0000 0000 0100 1111). We need to examine this entry to determine the next cluster number allocated to the file.
3. If the current cluster number is even, keep the lowest 12 bits of the 16-bit word. If the current cluster number is odd, keep the highest 12 bits of the 16-bit word. For example, our cluster number (3) is odd, so we keep the highest 12 bits (0000 0000 0100), and this indicates that cluster 4 is the next cluster.

We return to step 1 and calculate the offset of cluster 4 in the FAT table: The current cluster number is 4, so we calculate Int(4 /2) + 4 = 6. The word at offset 6 is 6005h (0110 0000 0000 0101). The value 6 is even, so we take the lowest 12 bits of 6005h, producing a new cluster number of 5. Therefore, FAT entry 4 contains the number 5.

Fortunately, a 16-bit FAT is easier to decode, because entries do not cross byte boundaries. In a 16-bit FAT, cluster n is represented by the entry at offset n * 2 in the table.

Finding the Starting Sector

Given a cluster number, we need to know how to calculate its starting sector number:

1. Subtract 2 from the cluster number and multiply the result by the disk's sectors per cluster. A 1.44MB disk has one sector per cluster, so we multiply by 1.
2. Add the starting sector number of the data area. On a 1.44MB disk, this is sector 33. For example, cluster number 3 is located at sector 34: 

\[(3 - 2) \times 1 + 33 = 34\]

Cluster Display Program

In this section, we will demonstrate a program that reads a 1.44MB diskette in drive A, loads its file allocation table and root directory into a buffer, and displays each filename along with a list of all clusters allocated to the file. The following is a sample of the program's output:

```
C:\WINDOWS\System32\cmd.exe
Cluster Display Program (CLUSTER.EXE)

The following clusters are allocated to each file:

SECTOR16ASM  3 4 5 6 7 8 9 10 11  
DISKS INC  12 13  
DRIVED~1TXT  14  
IRVINE16INC  15  
MAKE16 BAT  16 17  
DEVICE OBJ 18 19 20 21 22 23 24  
Press any key to continue ....
```

The main procedure displays a greeting, loads the directory and FAT into memory, and loops through each directory entry. The most important task here is to check the first character of each directory entry to see if it refers to a filename. If it does, we check the file's attribute byte at offset 08h
to make sure the entry is not a volume label or directory name. We screen out directory entries with attributes of 00h, E5h, 2Eh, and 18h.

Regarding the attribute byte: Bit 3 is set if the entry is a volume name, and bit 4 is set if it is a directory name. The TEST instruction used here sets the Zero flag only if both bits are clear.

LoadFATandDir loads the disk directory into dirbuf, and it loads the FAT into fattable. DisplayClusters contains a loop that displays all cluster numbers allocated to a single file. The disk directory has already been read into dirbuf, and we assume that SI points to the current directory entry.

The Next_FAT_Entry procedure uses the current cluster number (passed in AX) to calculate the next cluster number, which it returns in AX. The SHR instruction in this procedure checks to see if the cluster number is even by shifting its lowest bit into the Carry flag. If it is, we retain the low 12 bits of DX; otherwise, we keep the high 12 bits. The new cluster number is returned in AX.

Here is the complete program listing:

TITLE Cluster Display Program (Cluster.asm)

; This program reads the directory of drive A, decodes
; the file allocation table, and displays the list of
; clusters allocated to each file.

INCLUDE Irvine16.inc

; Attributes specific to 1.44MB diskettes:
; num sectors, first copy of FAT
FATSectors = 9

; num sectors, root directory
DIRSectors = 14

; starting directory sector num
DIR_START = 19

SECTOR_SIZE = 512
DRIVE_A = 0
FAT_START = 1

EOLN equ <0dh,0ah>

Directory STRUCT
fileName BYTE 8 dup(?)
extension BYTE 3 dup(?)
attribute BYTE ?
reserved BYTE 10 dup(?)
time WORD ?
date WORD ?
startingCluster WORD ?
fileSize DWORD ?
Directory ENDS

ENTRIES_PER_SECTOR = SECTOR_SIZE / (size Directory)

.data
heading LABEL byte
BYTE 'Cluster Display Program (CLUSTER.EXE)'
BYTE EOLN,EOLN,'The following clusters are allocated '
BYTE 'to each file:',EOLN,EOLN,0

fattable WORD ((FATSectors * SECTOR_SIZE) / 2) DUP(?)
dirbuf Directory (DIRSectors * ENTRIES_PER_SECTOR) DUP(<>)
driveNumber BYTE ?

.code
main PROC
  call Initialize
  mov ax,OFFSET dirbuf
  mov ax,OFFSET driveNumber
  call LoadFATandDir
  jc A3 ; quit if we failed
  mov si,OFFSET dirbuf ; index into the directory
  A1: cmp (Directory PTR [si]).filename,0 ; entry never used?
      je A3 ; yes: must be the end
      cmp (Directory PTR [si]).filename,0E5h ; entry deleted?
      je A2 ; yes: skip to next entry
      cmp (Directory PTR [si]).filename,2Eh ; parent directory?
      je A2 ; yes: skip to next entry

A3: mov ax,OFFSET dirbuf

cmp (Directory PTR [si]).attribute, 0Fh ; extended filename?
je A2

test (Directory PTR [si]).attribute, 18h ; vol or directory name?
jnz A2 ; yes: skip to next entry
call displayClusters ; must be a valid entry

A2: add si, 32 ; point to next entry
jmp A1
A3: exit

main ENDP

;----------------------------------------------------------
LoadFATandDir PROC
; Load FAT and root directory sectors.
; Receives: nothing
; Returns: nothing
;----------------------------------------------------------
pusha
; Load the FAT
mov al, DRIVE_A
mov cx, FATsectors
mov dx, FAT_START
mov bx, OFFSET fattable
int 25h ; read sectors
add sp, 2 ; pop old flags off stack
; Load the Directory
mov cx, DIRsectors
mov dx, DIR_START
mov bx, OFFSET dirbuf
int 25h
add sp, 2
popa
ret
LoadFATandDir ENDP

;----------------------------------------------------------
DisplayClusters PROC
; Display all clusters allocated to a single file.
; Receives: SI contains the offset of the directory entry.
;----------------------------------------------------------
push ax
call displayFilename ; display the filename
mov ax, [si+1Ah] ; get first cluster
C1: cmp ax, 0FFFh ; last cluster?
je C2 ; yes: quit
mov bx, 10
mov ah, read
call WriteDec ; display the number
call writeSpace ; display a space
call next_FAT_entry ; returns cluster # in AX
jmp C1 ; find next cluster
C2: call Crlf
pop ax
ret
DisplayClusters ENDP

;----------------------------------------------------------
WriteSpace PROC
; Write a single space to standard output.
;----------------------------------------------------------
push ax
mov ah, 2 ; function: display character
mov dl, 20h ; 20h = space
int 21h
pop ax
ret
WriteSpace ENDP

;----------------------------------------------------------
Next_FAT_entry PROC
; Find the next cluster in the FAT.
; Receives: AX = current cluster number
; Returns: AX = new cluster number
;----------------------------------------------------------
push bx ; save regs
push cx
mov bx,ax ; copy the number
shr bx,1 ; divide by 2
add bx,ax ; new cluster OFFSET
mov dx,fattable[bx] ; DX = new cluster value
shr ax,1 ; old cluster even?
jc E1 ; no: keep high 12 bits
and dx,0FFFh ; yes: keep low 12 bits
jmp E2
E1: shr dx,4 ; shift 4 bits to the right
E2: mov ax,dx ; return new cluster number
pop cx ; restore regs
pop bx
ret
Next_FAT_entry ENDP

DisplayFilename PROC
; Display the file name.
;----------------------------------------------------------
mov byte ptr [si+11],0 ; SI points to filename
mov dx,si
call Writestring
mov ah,2 ; display a space
mov dl,20h
int 21h
ret
DisplayFilename ENDP

Initialize PROC
; Set up DS, clear screen, display a heading.
;-----------------------------------------------
mov ax,@data
mov ds,ax
call ClrScr
mov dx,OFFSET heading ; display program heading
call Writestring
ret
Initialize ENDP
END main
Answers: Binary and Hexadecimal Numbers

1. Write each of the following decimal numbers in binary.

**Hint:** To convert a binary number to its decimal equivalent, evaluate each digit position as a power of 2. The decimal value of \(2^0\) is 1, \(2^1\) is 2, \(2^2\) is 4, and so on. For example, the binary number 1111 is equal to 15 decimal.

\[
\begin{align*}
a. \quad 2 &= 00000010 \\
b. \quad 7 &= 00000111 \\
c. \quad 5 &= 00000101 \\
d. \quad 8 &= 00001000 \\
e. \quad 9 &= 00001001 \\
f. \quad 12 &= 00001100 \\
g. \quad 15 &= 00001111 \\
h. \quad 16 &= 00010000 \\
i. \quad 20 &= 00010100 \\
j. \quad 27 &= 00011011 \\
k. \quad 32 &= 00100000 \\
l. \quad 64 &= 01000000 \\
\end{align*}
\]

2. Write each of the following binary numbers in decimal:

**Hint:** To calculate the decimal value of a binary number, add the value of each bit position containing a 1 to the number’s total value. For example, the binary number 0 0 0 0 1 0 0 1 may be interpreted in decimal as \((1 \times 2^3) + (1 \times 2^0)\).

\[
\begin{align*}
a. \quad 00000101 &= 5 \\
b. \quad 00001111 &= 15 \\
c. \quad 00010000 &= 16 \\
d. \quad 00010110 &= 22 \\
e. \quad 00001011 &= 11 \\
f. \quad 00011100 &= 28 \\
g. \quad 00110000 &= 48 \\
h. \quad 00100111 &= 39 \\
i. \quad 01001000 &= 64 \\
j. \quad 01100011 &= 39 \\
k. \quad 10100000 &= 160 \\
l. \quad 10101010 &= 170 \\
\end{align*}
\]

3. Write each of the following binary numbers in hexadecimal:

**Hint:** To calculate the hexadecimal value of a binary number, translate each group of four bits to its equivalent hexadecimal digit. For example, 1100 = C, and 1011 = B.

\[
\begin{align*}
a. \quad 00000101 &= 05h \\
b. \quad 00001111 &= 0Fh \\
c. \quad 00010000 &= 10h \\
d. \quad 00010110 &= 16h \\
e. \quad 00001011 &= 0Bh \\
f. \quad 00011100 &= 1Ch \\
g. \quad 00110000 &= 30h \\
h. \quad 00100111 &= 27h \\
i. \quad 01001000 &= 48h \\
j. \quad 01100011 &= 63h \\
k. \quad 10100000 &= A0h \\
l. \quad 10101010 &= ABh \\
\end{align*}
\]

4. Write each of the following hexadecimal numbers in binary:

**Hint:** To calculate the binary value of a hexadecimal number, translate each hexadecimal digit into its corresponding four-bit binary pattern. (You can also translate the digit to decimal, and then convert it to its equivalent binary bit pattern.) For example, hex C = 1100, and hex B = 1011.

\[
\begin{align*}
a. \quad 0005h &= 00000101 \\
b. \quad 000Fh &= 00001111 \\
g. \quad 0030h &= 0000111000 \\
h. \quad 0027h &= 000100111 \\
\end{align*}
\]
5. Write each of the following hexadecimal numbers in decimal:

*Hint:* To calculate the decimal value of a hexadecimal number, multiply each hexadecimal digit by its corresponding power of 16. The sum of these products is the decimal value of the number. For example, hexadecimal 12A = (1 * 256) + (2 * 16) + (10 * 1) = 298. *Hint:* $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, and $16^3 = 4096$. Also, you can use the following Hexadecimal digit table as an aid:

<table>
<thead>
<tr>
<th>Extended Hexadecimal Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  = 10</td>
</tr>
<tr>
<td>B  = 11</td>
</tr>
<tr>
<td>C  = 12</td>
</tr>
<tr>
<td>D  = 13</td>
</tr>
<tr>
<td>E  = 14</td>
</tr>
<tr>
<td>F  = 15</td>
</tr>
</tbody>
</table>

Answers:

- **a.** 00D5h = 0B30h = 213
- **b.** 002Fh = 06DFh = 47
- **c.** 0110h = 1AB6h = 272
- **d.** 0216h = 0A63h = 534
- **e.** 004Bh = 02A0h = 75
- **f.** 041Ch = 1FABh = 1052
Answers: Signed Integers

1. Write each of the following signed decimal integers in 8-bit binary notation:

   **Hint:** Remove the sign, create the binary representation of the number, and then convert it to its two's complement.

   a. \(-2\) = 11111110
   b. \(-7\) = 11111001
   c. \(-128\) = 10000000
   d. \(-16\) = 11110000
   e. \(+15\) = 00001111
   f. \(-1\) = 11111111
   g. \(+127\) = 01111111
   h. \(-56\) = 11001000

2. Write each of the following 8-bit signed binary integers in decimal:

   **Hint:** If the highest bit is set, convert the number to its two's complement, create the decimal representation of the number, and then prepend a negative sign to the answer.

   a. 11111111 = -1
   b. 11110000 = -16
   c. 10000000 = -128
   d. 10000001 = -127
   e. 00001111 = +15
   f. 10101111 = -81
   g. 11111111 = +127
   h. 01010101 = +85

3. Which of the following integers are valid 16-bit signed decimal integers?

   a. \(+32469\) = V
   b. \(+32767\) = V
   c. \(-32768\) = V
   d. \(+65535\) = I
   e. \(-32768\) = V
   f. \(+32785\) = I
   g. \(+32767\) = I
   h. \(+65535\) = I

4. Indicate the sign of each of the following 16-bit hexadecimal integers:

   a. \(7FB9h\) = \(P\)
   b. \(8123h\) = \(N\)
   c. \(0D000h\) = \(N\)
   d. \(649Fh\) = \(P\)

5. Write each of the following signed decimal integers as a 16-bit hexadecimal value:

   a. \(-42\) = \(FFD6h\)
   b. \(-127\) = \(FF81h\)
   c. \(-4096\) = \(F000h\)
   d. \(-16\) = \(FFF0h\)
   e. \(+32768\) = \(8000h\)
   f. \(-1\) = \(FFFFh\)
   g. \(-1\) = \(FFFFh\)
   h. \(-256\) = \(FF00h\)
1. For each of the following binary floating-point numbers, supply the equivalent value as a base 10 fraction, and then as a base 10 decimal. The first problem has been done for you:

<table>
<thead>
<tr>
<th>Binary Floating-Point</th>
<th>Base 10 Fraction</th>
<th>Base 10 Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.101</td>
<td>1 5/8</td>
<td>1.625</td>
</tr>
<tr>
<td>11.1</td>
<td>3 3/4</td>
<td>3.75</td>
</tr>
<tr>
<td>1.1</td>
<td>1 1/2</td>
<td>1.5</td>
</tr>
<tr>
<td>101.001</td>
<td>5 1/8</td>
<td>5.125</td>
</tr>
<tr>
<td>1101.0101</td>
<td>13 5/16</td>
<td>13.3125</td>
</tr>
<tr>
<td>1110.00111</td>
<td>14 7/32</td>
<td>14.21875</td>
</tr>
<tr>
<td>10010.10101</td>
<td>16 43/64</td>
<td>16.671875</td>
</tr>
<tr>
<td>111.0000011</td>
<td>7 3/128</td>
<td>7.0234375</td>
</tr>
<tr>
<td>11000101</td>
<td>3 5/64</td>
<td>3.078125</td>
</tr>
</tbody>
</table>

2. For each of the following exponent values, shown here in decimal, supply the actual binary bits that would be used for an 8-bit exponent in the IEEE Short Real format. The first answer has been supplied for you:

<table>
<thead>
<tr>
<th>Exponent (E)</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10000001</td>
</tr>
<tr>
<td>5</td>
<td>10000100</td>
</tr>
<tr>
<td>0</td>
<td>01111111</td>
</tr>
<tr>
<td>-10</td>
<td>01110101</td>
</tr>
<tr>
<td>128</td>
<td>11111111</td>
</tr>
<tr>
<td>-1</td>
<td>01111110</td>
</tr>
</tbody>
</table>

3. For each of the following floating-point binary numbers, supply the normalized value and the resulting exponent. The first answer has been supplied for you:

<table>
<thead>
<tr>
<th>Binary Value</th>
<th>Normalized As</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000.11</td>
<td>1.000011</td>
<td>4</td>
</tr>
<tr>
<td>1101.101</td>
<td>1.101101</td>
<td>3</td>
</tr>
<tr>
<td>.00101</td>
<td>1.01</td>
<td>-3</td>
</tr>
<tr>
<td>1.0001</td>
<td>1.0001</td>
<td>0</td>
</tr>
<tr>
<td>10000011.0</td>
<td>1.0000011</td>
<td>7</td>
</tr>
<tr>
<td>.00000011001</td>
<td>1.1001</td>
<td>-6</td>
</tr>
</tbody>
</table>

4. For each of the following floating-point binary examples, supply the complete binary representation of the number in IEEE Short Real format. The first answer has been supplied for you:

<table>
<thead>
<tr>
<th>Binary Value</th>
<th>Sign, Exponent, Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.11</td>
<td>1 01111111 1100000000000000000000000</td>
</tr>
<tr>
<td>+1101.101</td>
<td>0 10000010 1011010000000000000000000</td>
</tr>
<tr>
<td>-.00101</td>
<td>1 01111100 0100000000000000000000000</td>
</tr>
<tr>
<td>+100111.0</td>
<td>0 10000100 0011100000000000000000000</td>
</tr>
<tr>
<td>+.0000001101011</td>
<td>0 01111000 1010110000000000000000000</td>
</tr>
</tbody>
</table>
Answers: Register and Immediate Operands

1. Indicate whether or not each of the following MOV instructions is valid:
   (notate: V = valid, I = invalid)
   - a. mov ax, bx  V
   - b. mov dx, bl  I
   - c. mov ecx, edx  V
   - d. mov si, di  V
   - e. mov ds, ax  V
   - f. mov ds, es  I
   - g. mov al, dh  V
   - h. mov ax, dh  I
   - i. mov ip, ax  I
   - j. mov si, cl  I
   - k. mov edx, ax  I
   - l. mov ax, es  V

2. Indicate whether or not each of the following MOV instructions is valid:
   (notate: V = valid, I = invalid)
   - a. mov ax, 16  V
   - b. mov dx, 7F65h  V
   - c. mov ecx, 6F23458h  V
   - d. mov si, -1  V
   - e. mov ds, 1000h  I
   - f. mov al, 100h  I
   - g. mov 123, dh  I
   - h. mov ss, ds  I
   - i. mov 0FABh, ax  I
   - j. mov si, cl  I
   - k. mov edx, esi  V
   - l. mov edx, -2  V
Answers: Addition and Subtraction Instructions

1. Indicate whether or not each of the following instructions is valid.

   a. add ax, bx  V
      operand
   b. add dx, bl  I
      mismatch
   c. add ecx, dx I
   d. sub si, di V
   e. add bx, 90000 I
      source too large
      cannot use
   f. sub ds, 1 I
      segment reg
   g. dec ip I
      cannot modify IP
   h. dec edx V
   i. add edx, 1000h V
   j. sub ah, 126h I
      source too large
   k. sub al, 256 I
      source too large
   l. inc ax, 1 I
      extraneous operand

2. What will be the value of the Carry flag after each of the following instruction sequences has executed?
   (notate: CY = carry, NC = no carry)

   a. mov ax, 0FFFFh
      add ax, 1  CY
   b. mov bh, 2
      sub bh, 2  NC
   c. mov dx, 0
      dec dx
      (Carry not affected by INC and DEC)
   d. mov al, 0DFh
      add al, 32h  CY
   e. mov si, 0B9F6h
      sub si, 9874h  NC
   f. mov cx, 695Fh
      sub cx, A218h  CY

3. What will be the value of the Zero flag after each of the following instruction sequences has executed?
   (notate: ZR = zero, NZ = not zero)
4. What will be the value of the Sign flag after each of the following instruction sequences has executed?

(a) mov ax,0FFFFh sub ax,1
    PL
(b) mov bh,2 sub bh,3 NG
(c) mov dx,0 dec dx NG
(d) mov ax,0DFh add ax,32h NG
(e) mov si,0B9F6h sub si,9874h PL
(f) mov cx,695Fh add cx,96A1h ZR

5. What will be the values of the Carry, Sign, and Zero flags after the following instructions have executed?

(a) mov ax,620h sub ah,0F6h CY,PL,NZ
(b) mov ax,720h sub ax,0E6h NC,PL,NZ
(c) mov
mov
ax, 0B6D4h
add
al, 0B3h CY, NG, NZ

8. What will be the values of the Overflow, Sign, and Zero flags after the following instructions have executed?

mov
bl, -127
dec
bl NV, NG, NZ

9. What will be the values of the Carry, Overflow, Sign, and Zero flags after the following instructions have executed?

mov
cx, -4097
add
cx, 1001h CY, NV, PL, ZR

10. What will be the values of the Carry, Overflow, Sign, and Zero flags after the following instructions have executed?

mov
ah, -56
add
ah, -60 CY, NV, NG, NZ
Answers: Direct Memory Operands

Updated 9/30/2002

Use the following data declarations for Questions 1-4. Assume that the offset of byteVal is 00000000h, and that all code runs in Protected mode.

.data
byteVal BYTE 1,2,3,4
wordVal WORD 1000h,2000h,3000h,4000h
dwordVal DWORD 12345678h,34567890h
aString BYTE "ABCDEFG",0

1. Indicate whether or not each of the following MOV instructions is valid:
   (note: V = valid, I = invalid)
   a. mov ax,byteVal    I
      mov byteVal v
   b. dx,wordVal     v
      mov wordVal v
   c. ecx,dwordVal   v
      mov dwordVal v
   d. si,aString    I
      mov aString v
   e. esi,offset v
      mov aString v
   f. al,byteVal v

2. Indicate whether or not each of the following MOV instructions is valid:
   (note: V = valid, I = invalid)
   mov
   a. eax,offset v
      byteVal
   mov
   b. dx,wordVal+2 v
      mov
   c. ecx,offset v
      dwordVal
   mov
   d. si,dwordVal I
      mov
   e. esi,offset v
      aString+2
      mov
   f. al,offset I
      byteVal+1

3. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
   (If any instruction is invalid, indicate "I" as the answer.)
   mov
   a. ax,offset 00000000h
      byteVal
   mov
   b. dx,wordVal 1000h
      mov
   c. ecx,dwordVal 12345678h
      mov
4. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset
byteVal+2 0000002h
b. dx, wordVal+4 3000h
c. ecx, dwordVal+4 34567890h
d. esi, offset
wordVal+4 0000008h
e. esi, offset
aString-1 0000013h
```

Use the following data declarations for Questions 5-6. Assume that the offset of byteVal is 0000:

```
.data
byteVal BYTE 3 DUP(0FFh),2,"XY"
wordVal WORD 2 DUP(6),2
dwordVal DWORD 8,7,6,5
dwordValSiz WORD ($ - dwordVal)
ptrByte DWORD byteVal
ptrWord DWORD wordVal
```

5. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. eax, offset
wordVal 0000006h
b. dx, wordVal+4 0002h
c. ecx, dwordVal+4 0000007h
d. esi, offset
dwordValSiz 0010h
e. al, byteVal+4 58h('X')
```

6. Indicate the hexadecimal value moved to the destination operand by each of the following MOV instructions:
(If any instruction is invalid, indicate "I" as the answer.)

```
mov
a. ax, dwordVal+2 I
b. dx, wordVal-2 5958h *
c. mov 00000000h
```
eax, ptrByte
mov
esi, ptrWord
mov
dwordVal+2
edi, offset

* The two character bytes are automatically reversed when loaded into a 16-bit register.
Answers: Indirect and Indexed Operands

Use the following data declarations. Assume that the offset of byteVal is 0000:

```assembly
.data
byteVal  db  1,2,3,4
wordVal  dw  1000h,2000h,3000h,4000h
dwordVal dd  12345678h,34567890h
aString db  "ABCDEF",0
pntr     dw  wordVal
```

1. Indicate whether or not each of the following instructions is valid:
   (note: V = valid, I = invalid)

   a. mov
      ax,byteVal[si]  I (operand size mismatch)
      mov al,[si+1]  2
   b. add
dx,[cx+wordVal]  I (CX is not a base or index register)
   c. mov
      ecx,[edi+dwordVal]
      mov al,byteVal+1
      mov [si],al  2
   d. xchg
      al,[bx]  V
      xchg al,byteVal[dx]
      inc byte ptr [si]  44h('D')
   e. mov
      ax,[bx+4]
      mov bx,pntr
      add word ptr [bx],2  1002h
   f. mov
      [bx],[si]
      xchg
      [bx],[si]
```

2. Indicate the hexadecimal value of the final destination operand after each of the following code fragments has executed:
   (If any instruction is invalid, indicate "I" as the answer.)

   a. mov si,offset
      byteVal
      mov al,[si+1]  2
   b. mov di,6
      mov dx,wordVal[di]
      mov bx,4
      mov ecx,[bx+dwordVal]
      34567890h
   c. mov si,offset
      aString
      mov al,byteVal+1
      mov [si],al  2
   d. mov si,offset
      aString+2
      inc byte ptr [si]
      44h('D')
   f. mov bx,pntr
      add word ptr [bx],2  1002h
g. mov di, offset
   pntr
   mov si, [di]
   mov ax, [si + 2] 2000h

3. Indicate the hexadecimal value of the final destination operand after each of the following code fragments has executed:
(If any instruction is invalid, indicate "I" as the answer.)

(a) xchg si, pntr
    xchg [si], wordVal
    I (memory to memory not permitted)

(b) mov ax, pntr
    xchg ax, si
    mov dx, [si + 4]
    dx = 3000h

(c) mov edi, 0
    mov di, pntr
    add edi, 8
    mov eax, [edi]
    12345678h

(d) mov esi, offset aString
    xchg esi, pntr
    mov dl, [esi]
    I (esi and pntr have different sizes)

(e) mov esi, offset aString
    mov dl, [esi + 2]
    43h ('C')
Write the names of variables next to their corresponding memory locations.

<table>
<thead>
<tr>
<th>doubleword</th>
<th>word</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0017</td>
</tr>
</tbody>
</table>
;Problem statement:
;Write a program that inputs a single character and redisplays
;(echoes) it back to the screen. Hint: Use INT 21h for the
;character input.

INCLUDE Irvine16.inc

.code
main proc
    mov ax,@data
    mov ds,ax

    mov ah,1 ; input character with echo
    int 21h ; AL = character
    mov ah,2 ; character output
    mov dl,al
    int 21h

    exit
main endp
end main
Title MS-DOS Example                     (DOS1-2.ASM)

; Problem statement:
;Write a program that inputs a string of characters
;(using a loop) and stores each character in an array.
;Display a memory dump in CodeView showing the array.

INCLUDE Irvine16.inc

.data
COUNT = 20
charArray db COUNT dup(0),0

.code
main proc
    mov ax,@data
    mov ds,ax

    mov si,offset charArray
    mov cx,COUNT

L1:    mov ah,1      ; input character with echo
    int 21h       ; AL = character
    mov [si],al   ; save in array
    inc si        ; next array position
    Loop L1       ; repeat loop

    exit
main endp
end main
; Problem statement:
; Write a program that inputs a string of characters
; (using a loop) and stores each character in an array.
; Redisplay the array at the end of the program.

INCLUDE Irvine16.inc

.data
COUNT = 20
charArray db COUNT dup(0),0

.code
main proc
    mov ax,@data
    mov ds,ax

    mov si,offset charArray
    mov cx,COUNT

L1:    mov ah,1      ; input character with echo
    int 21h       ; AL = character
    mov [si],al   ; save in array
    inc si        ; next array position
Loop L1       ; repeat loop

; Redisplay the array on the screen

    call Crlf      ; start new line
    mov si,offset charArray
    mov cx,COUNT

L2:    mov ah,2      ; character output
    mov dl,[si]   ; get char from array
    int 21h       ; display the character
    inc si
Loop L2

    call Crlf

    exit
main endp
end main
Problem statement:
Write a program that reads a series of ten lowercase letters from input (without displaying it), converts each character to uppercase, and then displays the converted character.

```
INCLUDE Irvine16.inc

COUNT = 10

.code
main proc
    mov ax, @data
    mov ds, ax

    mov cx, COUNT ; loop counter

L1: mov ah, 7 ; input character, no echo
    int 21h ; AL = character
    sub al, 20h ; convert to uppercase
    mov ah, 2 ; character output function
    mov dl, al ; character must be in DL
    int 21h ; display the character
    Loop L1 ; repeat loop

    exit
main endp
```

end main
; Problem statement:
; Write a program that displays a string using
; INT 21h function 9.

INCLUDE Irvine16.inc

.data
message db "Displaying a string",0dh,0ah,"$"

.code
main proc
    mov ax,@data
    mov ds,ax

    mov ah,9 ; DOS function #9
    mov dx,offset message ; offset of the string
    int 21h ; display it

    exit
main endp

end main
Problem statement:
Write a program that inputs a string using DOS function 0Ah. Limit the input to ten characters.
Redisplay the string backwards

INCLUDE Irvine16.inc

.data
COUNT = 11
keyboardArea label byte
maxkeys   db  COUNT
charsInput db  ?
buffer    db  COUNT dup(0)

.code
main proc
    mov  ax,@data
    mov  ds,ax

    mov  ah,0Ah     ; buffered keyboard input
    mov  dx,offset keyboardArea
    int  21h
    call Crlf

    ; Redisplay the string backwards, using SI
    ; as an index into the string

    mov  ah,0
    mov  al,charsInput  ; get character count
    mov  cx,ax          ; put in loop counter
    mov  si,ax          ; point past end of string
    dec  si              ; back up one position
L1: mov  dl,buffer[si]  ; get char from buffer
    mov  ah,2           ; MS-DOS char output function
    int  21h
    dec  si              ; back up in buffer
    Loop L1             ; loop through the string
    call Crlf

    exit
main endp

end main
Problem statement:
Write a program that inputs a string of up to 80 characters using DOS function 3Fh. After the input, display a count on the screen of the actual number of characters typed by the user.

INCLUDE Irvine16.inc

.data
COUNT = 80

; create the input buffer, and allow for two extra characters (CR/LF)
buffer db (COUNT+2) dup(0)

.code
main proc
  mov ax,@data
  mov ds,ax

  mov ah,3Fh ; input from file or device
  mov bx,0 ; keyboard device handle
  mov cx,COUNT ; max input count
  mov dx,offset buffer
  int 21h ; call DOS to read the input

  ; Display the character count in AX that was returned by INT 21h function 3Fh
  ; (minus 2 for the CR/LF characters)
  sub ax,2
  call Writedec ; display AX
  call Crlf

  exit
main endp
end main
title MS-DOS Function Calls - 2        (DOS2-3.ASM)

;Problem statement:
;Write a program that inputs the month, day, and
;year from the user. Use the values to set the system
;date with DOS function 2Bh.

INCLUDE Irvine16.inc

.data
monthPrompt db "Enter the month: ",0
dayPrompt   db "Enter the day:   ",0
yearPrompt  db "Enter the year:  ",0
blankLine   db 30 dup(" "),0dh,0
month db ?
day db ?
year dw ?

.code
main proc
  mov  ax,@data
  mov  ds,ax

  mov  dx,offset monthPrompt
  call Writestring
  call Readint
  mov  month,al
  mov  dx,offset blankLine
  call Writestring

  mov  dx,offset dayPrompt
  call Writestring
  call Readint
  mov  day,al
  mov  dx,offset blankLine
  call Writestring

  mov  dx,offset yearPrompt
  call Writestring
  call Readint
  mov  year,ax

  mov  ah,2Bh         ; MS-DOS Set Date function
  mov  cx,year
  mov  dh,month
  mov  dl,day
  int  21h             ; set the date now

  ;(AL = FFh if the date could not be set)

  exit
main endp
;Problem statement:
;Write a program that uses DOS function 2Ah to
;get and display the system date. Use the
;following display format: yyyy-m-d.

INCLUDE Irvine16.inc

.data
month db ?
day db ?
year dw ?

.code
main proc
  mov  ax, @data
  mov  ds, ax

  mov  ah, 2Ah    ; MS-DOS Get Date function
  int  21h        ; get the date now
  mov  year, cx
  mov  month, dh
  mov  day, dl

  mov  ax, year
  call Writedec

  mov  ah, 2      ; display a hyphen
  mov  dl, "-"
  int  21h

  mov  al, month ; display the month
  mov  ah, 0
  call Writedec

  mov  ah, 2      ; display a hyphen
  mov  dl, "-"
  int  21h

  mov  al, day    ; display the day
  mov  ah, 0
  call Writedec
  call Crlf

  exit
main endp
end main
**Answers: Boolean and Comparison Instructions**

**AND and OR Instructions**

1. Method one: Clear all nonessential bits and compare the remaining ones with the mask value:

   ```
   and  AL, 00000111b
   cmp AL, 00000111b
   je   Target
   ```

   Method two: Use the boolean rule that $a \land b \land c = \sim (\sim a \lor \sim b \lor \sim c)$

   ```
   not  AL
   test AL, 00000111b
   jz   Target
   ```

2. 

   ```
   test AL, 00000111b
   jnz  Target
   ```

3.

   ```
   and  BL, 10001111b
   ```

4. 

   ```
   or   CL, 00011000b
   ```