Assembly Language Workbook

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.

Many of the lessons involve basic skills that should be practiced as soon as you begin learning assembly 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
- Register Names and Usage
- Using Debug to Write Programs
- Using the Link Library

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!

A number of tutorials were added to the workbook for topics not covered in the book. I found, by corresponding with other college professors, that they were providing their own handouts for selected topics. Here is a partial list of tutorials for these new topics:

- Floating-Point Binary
- The Precision Problem
- Error-Correcting Codes
- Cache Memory
- Pipelining
- Superscalar Architecture
- Branch Prediction
- Using Programmers Workbench

No doubt, additional topics will appear throughout the year, such as 32-bit flat model programming. This is a workbook in progress.

If you think you've found a mistake, verify it with your instructor and if it needs correcting, please let me know right away. Your information could help thousands of other people.

Use the Workbook Now

Workbook Topics

1.

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.00001011	**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.01001000

d.00010110 j.01100011

e.00001011 k.10100000

f.00011100 l.10101011

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

a. 0005hg. 0030h

b. 000Fhh. 0027h

c. 0010hi. 0048h

d. 0016hj. 0063h

e. 000Bhk. A064h

f. 001Chl. ABDEh

5. Write each of the following hexadecimal numbers in decimal:

a. 00D5hg. 0B30h

b. 002Fhh. 06DFh

c. 0110hi. 1AB6h

d. 0216hj. 0A63h

e. 004Bhk. 02A0h

f. 041Chl. 1FABh

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.

a. 2 = 00000010	**g. 15 = 00001111**
b. 7 = 00000111	**h. 16 = 00010000**
c. 5 = 00000101	**i. 20 = 00010100**
d. 8 = 00001000	**j. 27 = 00011011**
e. 9 = 00001001	**k. 32 = 00100000**
f. 12 = 00001100	**l. 64 = 01000000**

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 $(1 * 2^3) + (1 * 2^0)$.

a.00000101 = 5	**g.00110000 = 48**
b.00001111 = 15	**h.00100111 = 39**
c.00010000 = 16	**i.01000000 = 64**
d.00010110 = 22	**j.01100011 = 99**
e.00001011 = 11	**k.10100000 = 160**
f.00011100 = 28	**l.10101010 = 170**

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.

a.00000101 = 05hg.00110000 = 30h

b.00001111 = 0Fhh.00100111 = 27h

c.00010000 = 10hi.01001000 = 48h

d.00010110 = 16hj.01100011 = 63h

e.00001011 = 0Bhk.10100000 = A0h

f.00011100 = 1Chl.10101011 = ABh

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.

a.0005h = 00000101g.0030h = 00110000

b.000Fh = 00001111h.0027h = 00100111

c.0010h = 00010000i.0048h = 01001000

d.0016h = 00010110j.0063h = 01100011

e.000Bh = 00001011k.A064h = 10100000 01100100

f.001Ch = 00011100l.ABDEh = 10101011 11011110

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:

Extended Hexadecimal Digits	
A = 10	B = 11
C = 12	D = 13
E = 14	F = 15

Answers:

a.00D5h = 213 g.0B30h = 2864

b.002Fh = 47 h.06DFh = 1759

c.0110h = 272 i.1AB6h = 6838

d.0216h = 534 j.0A63h = 2659

e.004Bh = 75 k.02A0h = 672

f.041Ch = 1052l.1FABh = 8107

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:

N:	0001
Reverse N:	1110
Add 1:	1111

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:

N = 1	00000001
Reverse N:	11111110
Add 1:	11111111
N = 1	0000000000000001
Reverse N:	1111111111111110
Add 1:	1111111111111111

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

n(decimal)	n(binary)	NEG(n)	(decimal)

+2	00000010	11111110	–2
+16	00010000	11110000	–16
+127	01111111	10000001	–127

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

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 e.+15 = 00001111

b.-7 = 11111001 f.-1 = 11111111

c.-128 = 10000000g.-56 = 11001000

d.-16 = 11110000 h.+127 = 01111111

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 g.00001111 = +15

b.11110000 = -16 h.10101111 = -81

c.10000000 = -128i.11111100 = -4

d.10000001 = -127j.01010101 = +85

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

a.+32469 = Vd.+32785 = I

b.+32767 = Ve.-32785 = I

c.-32768 = Vf.+65535 = I

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

a.7FB9h = Pc.0D000h = N

b.8123h = Nd.649Fh = P

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

a.-42 = FFD6h e.-32768 = 8000h

b.-127 = FF81h f.-1 = FFFFh

c.-4096 = F000hg.-8193 = DFFFh

d.-16 = FFF0h h.-256 = FF00h

Tutorial: Floating-Point Binary

The two most common floating-point binary storage formats used by Intel processors were created for Intel and later standardized by the IEEE organization:

IEEE Short Real: 32 bits	1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. Also called *single precision*.
IEEE Long Real: 64 bits	1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. Also called *double precision*.

Both formats use essentially the same method for storing floating-point binary numbers, so we will use the Short Real as an example in this tutorial. The bits in an IEEE Short Real are arranged as follows, with the most significant bit (MSB) on the left:

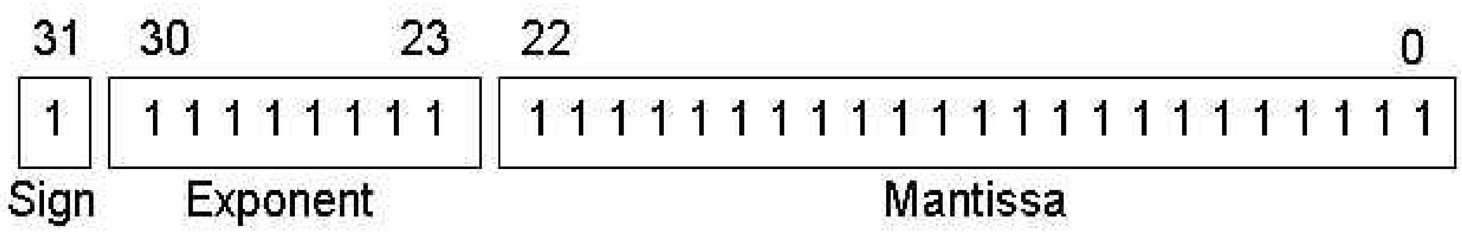


Fig. 1

The Sign

The sign of a binary floating-point number is represented by a single bit. A 1 bit indicates a negative number, and a 0 bit indicates a positive number.

The Mantissa

It is useful to consider the way decimal floating-point numbers represent their mantissa. Using -3.154 x 10^5 as an example, the **sign** is negative, the **mantissa** is 3.154, and the **exponent** is 5. The fractional portion of the mantissa is the sum of each digit multiplied by a power of 10:

.154 = 1/10 + 5/100 + 4/1000

A binary floating-point number is similar. For example, in the number +11.1011 x 2^3**,** the sign is positive, the mantissa is 11.1011, and the exponent is 3. The fractional portion of the mantissa is the sum of successive powers of 2. In our example, it is expressed as:

.1011 = 1/2 + 0/4 + 1/8 + 1/16

Or, you can calculate this value as 1011 divided by 2^4. In decimal terms, this is eleven divided by sixteen, or 0.6875. Combined with the left-hand side of 11.1011, the decimal value of the number is 3.6875. Here are additional examples:

Binary Floating-Point	Base 10 Fraction	Base 10 Decimal
11.11	3 3/4	3.75
0.00000000000000000000001	1/8388608	0.00000011920928955078125

The last entry in this table shows the smallest fraction that can be stored in a 23-bit mantissa. The following table shows a few simple examples of binary floating-point numbers alongside their equivalent decimal fractions and decimal values:

Binary	Decimal Fraction	Decimal Value

.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.0625
.00001	1/32	.03125

The Exponent

IEEE Short Real exponents are stored as 8-bit unsigned integers with a bias of 127. Let's use the number 1.101 x 2^5 as an example. The exponent (5) is added to 127 and the sum (132) is binary 10100010. Here are some examples of exponents, first shown in decimal, then adjusted, and finally in unsigned binary:

Exponent (E)	Adjusted (E + 127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+128	255	11111111
-127	0	00000000
-1	126	01111110

The binary exponent is unsigned, and therefore cannot be negative. The largest possible exponent is 128-- when added to 127, it produces 255, the largest unsigned value represented by 8 bits. The approximate range is from 1.0 x 2^{-127} to 1.0 x 2^{+128}.

Normalizing the Mantissa

Before a floating-point binary number can be stored correctly, its mantissa must be normalized. The process is basically the same as when normalizing a floating-point decimal number. For example, decimal 1234.567 is normalized as 1.234567 x 10^3 by moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent).

Similarly, the floating-point binary value 1101.101 is normalized as 1.101101 x 2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. Here are some examples of normalizations:

Binary Value	Normalized As	Exponent
1101.101	1.101101	3
.00101	1.01	-3
1.0001	1.0001	0
10000011.0	1.0000011	7

You may have noticed that in a normalized mantissa, the digit 1 always appears to the left of the decimal point. In fact, the leading 1 is omitted from the mantissa in the IEEE storage format because it is redundant.

Creating the IEEE Bit Representation

We can now combine the sign, exponent, and normalized mantissa into the binary IEEE short real representation. Using Figure 1 as a reference, the value 1.101 x 2^0 is stored as sign = 0 (positive), mantissa = 101, and exponent = 01111111 (the exponent value is added to 127). The "1" to the left of the decimal point is dropped from the mantissa. Here are more examples:

Binary Value	Biased Exponent	Sign, Exponent, Mantissa
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

Converting Decimal Fractions to Binary Reals

If a decimal fraction can be easily represented as a sum of fractions in the form (1/2 + 1/4 + 1/8 + ...), it is fairly easy to discover the corresponding binary real. Here are a few simple examples

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

Of course, the real world is never so simple. A fraction such as 1/5 (0.2) must be represented by a sum of fractions whose denominators are powers of 2. Here is the output from a program that subtracts each succesive fraction from 0.2 and shows each remainder. In fact, an exact value is not found after creating the 23 mantissa bits. The result, however, is accurate to 7 digits. The blank lines are for fractions that were too large to be subtracted from the remaining value of the number. Bit 1, for example, was equal to .5 (1/2), which could not be subtracted from 0.2.

```
     starting:   0.200000000000

 1
 2
 3   subtracting 0.125000000000
     remainder = 0.075000000000
 4   subtracting 0.062500000000
     remainder = 0.012500000000
 5
 6
 7   subtracting 0.007812500000
```

```
     remainder = 0.004687500000
 8   subtracting 0.003906250000
     remainder = 0.000781250000
 9
10
11   subtracting 0.000488281250
     remainder = 0.000292968750
12   subtracting 0.000244140625
     remainder = 0.000048828125
13
14
15   subtracting 0.000030517578
     remainder = 0.000018310547

16   subtracting 0.000015258789
     remainder = 0.000003051758
17
18
19   subtracting 0.000001907349
     remainder = 0.000001144409
20   subtracting 0.000000953674
     remainder = 0.000000190735
21
22
23   subtracting 0.000000119209
     remainder = 0.000000071526


Mantissa: .00110011001100110011001
```

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:

Binary Floating-Point	Base 10 Fraction	Base 10 Decimal
1.101 *(sample)*	1 5/8	1.625
11.11		
1.1		
101.001		
1101.0101		
1110.00111		
10000.101011		
111.0000011		
11.000101		

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:

Exponent (E)	Binary Representation
2 *(sample)*	10000001
5	
0	
-10	
128	
-1	

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:

Binary Value	Normalized As	Exponent
10000.11 *(sample)*	1.000011	4
1101.101		

.00101		
1.0001		
10000011.0		
.0000011001		

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:

Binary Value	Sign, Exponent, Mantissa
-1.11 *(sample)*	1 01111111 11000000000000000000000
+1101.101	
-.00101	
+100111.0	
+.0000001101011	

Answers: Floating-Point Binary

Updated 9/30/2002

There is no section of the book covering this topic, so click here to view a tutorial.

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:

Binary Floating-Point	Base 10 Fraction	Base 10 Decimal
1.101	1 5/8	1.625
11.11	3 3/4	3.75
1.1	1 1/2	1.5
101.001	5 1/8	5.125
1101.0101	13 5/16	13.3125
1110.00111	14 7/32	14.21875
10000.101011	16 43/64	16.671875
111.0000011	7 3/128	7.0234375
11.000101	3 5/64	3.078125

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:

Exponent (E)	Binary Representation
2	10000001
5	10000100
0	01111111
-10	01110101
128	11111111
-1	01111110

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:

Binary Value	Normalized As	Exponent
10000.11	1.000011	4
1101.101	1.101101	3
.00101	1.01	-3

1.0001	1.0001	0
10000011.0	1.0000011	7
.0000011001	1.1001	-6

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:

Binary Value	Sign, Exponent, Mantissa
-1.11	1 01111111 11000000000000000000000
+1101.101	0 10000010 10110100000000000000000
-.00101	1 01111100 01000000000000000000000
+100111.0	0 10000100 00111000000000000000000
+.0000001101011	0 01111000 10101100000000000000000

Tutorial: The Precision Problem

You need to read the previous topic, Floating-Point Binary Numbers before reading this topic.

The accuracy of a decimal fraction is limited by how closely it can be represented by a sum of fractions based on the powers of two. For example, the decimal value of 1/128 is .0078125. In the 23-bit mantissa of an IEEE Short Real storage format, the next largest number that can be stored is the sum of 1/128 and $1/2^{23}$:

1/128	0.00781250000000000000000
$1/2^{23}$	plus: 0.00000011920928955078125
$1/128 + 1/2^{23}$	equals: 0.00781261920928955078125

What if you want to store a number such as 0.00781251? In fact, it is impossible to do so with a 23 bit mantissa. There is no available fraction than can be added to 1/128 that will give us this number. The difference, .00000001, is smaller than $1/2^{23}$ (approximately 0.00000011920928955078).

A more accurate binary representation of 0.00781251, using 52 bits, would be: .0000001000000000000000000010101011110011000111011100.

There is clearly no way that we can accurately represent this number with only 23 bits.

When the Short Real data type is implemented by high-level language compilers, the guaranteed range of precision is 7 digits. This is because no number more precise that range can be trusted.

Unless you're an engineer, you probably won't lose much sleep over problems with binary precision. But be careful: serious errors occur in any program that does a successive series of floating-point calculations in which each new calculation is based on the results of a previous calculation. The small errors in each calculation are compounded to the point where the program's output bears no relation to reality.

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,edxi.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,6F23458hi.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**

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`	`g. mov al,dh V`
`b.mov dx,bl I`	`h. mov ax,dh I`
`c.mov ecx,edxV`	`i. mov ip,ax I`
`d.mov si,di V`	`j. mov si,cl I`
`e.mov ds,ax V`	`k. mov edx,axI`
`f.mov ds,es 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`	`g.mov 123,dh I`
`b.mov dx,7F65h V`	`h.mov ss,ds I`
`c.mov ecx,6F23458hV`	`i.mov 0FABh,ax I`
`d.mov si,-1 V`	`j.mov si,cl I`
`e.mov ds,1000h I`	`k.mov edx,esi V`
`f.mov al,100h I`	`l.mov edx,-2 V`

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.

(notate: 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?

(notate: 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?

(notate: 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 si,0B9F6h
sub si,9874h
```

```
f. mov cx,695Fh
   add cx,96A1h
```

4. What will be the value of the Sign flag after each of the following instruction sequences has executed?

(notate: 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?

(notate: 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?

(notate: 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?

(notate: 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?

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

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

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)

```
mov cx,-4097
add cx,1001h
```

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 ah,-56
add ah,-60
```

Answers: Addition and Subtraction Instructions

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

```
a.add ax,bx     V

b.add dx,bl     Ioperand size mismatch

c.add ecx,dx    I

d.sub si,di     V

e.add bx,90000 Isource too large

f.sub ds,1      Icannot use segment reg

g.dec ip        Icannot modify IP

h.dec edx       V

i.add edx,1000hV

j.sub ah,126h   Isource too large

k.sub al,256    Isource too large

l.inc ax,1      Iextraneous 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            NC
  sub bh,2

c.mov dx,0            ?? (Carry not affected by INC and DEC)
  dec dx

d.mov al,0DFh         CY
  add al,32h

e.mov si,0B9F6h       NC
  sub si,9874h

f.mov cx,695Fh        CY
  sub cx,A218h
```

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)

```
a.mov ax,0FFFFh       ZR
  add ax,1

b.mov bh,2            ZR
  sub bh,2

c.mov dx,0            NZ
  dec dx

d.mov al,0DFh         NZ
  add al,32h

e.mov si,0B9F6h       NZ
  sub si,9874h

f.mov cx,695Fh        ZR
  add cx,96A1h
```

4. What will be the value of the Sign flag after each of the following instruction sequences has executed?

(notate: PL = positive, NG = negative)

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,7FFEh
add ax,22h
```
NG

e.
```
mov si,0B9F6h
sub si,9874h
```
PL

f.
```
mov cx,8000h
add cx,A69Fh
```
PL

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

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

```
mov ax,620h
sub ah,0F6h  CY,PL,NZ
```

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

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

```
mov ax,720h
sub ax,0E6h  NC,PL,NZ
```

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

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

```
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?

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

```
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?

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

```
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?

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

```
mov ah,-56
add ah,-60 CY,NV,NG,NZ
```

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)

```
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.)

a.mov eax,offset byteVal

b.mov dx,wordVal

c.mov ecx,dwordVal

d.mov esi,offset wordVal

e.mov esi,offset aString

f.mov al,aString+2

g. mov edi,offset dwordVal

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.)

a.mov eax,offset byteVal+2

b.mov dx,wordVal+4

c.mov ecx,dwordVal+4

d.mov esi,offset wordVal+4

e.mov esi,offset aString-1

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.)

```
a.mov ax,dwordVal+2

b.mov dx,wordVal-2

c.mov eax,ptrByte

d.mov esi,ptrWord

e.mov edi,offset dwordVal+2
```

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:

(notate: V = valid, I = invalid)

```
a.mov ax,byteVal         I

b.mov dx,wordVal         V

c.mov ecx,dwordVal       V

d.mov si,aString         I

e.mov esi,offset aString V

f.mov al,byteVal         V
```

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

(notate: V = valid, I = invalid)

```
a.mov eax,offset byteVal   V

b.mov dx,wordVal+2         V
```

c.mov ecx,offset dwordVal V

d.mov si,dwordVal I

e.mov esi,offset aString+2 V

f.mov al,offset byteVal+1 I

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.)

a.mov ax,offset byteVal 00000000h

b.mov dx,wordVal 1000h

c.mov ecx,dwordVal 12345678h

d.mov esi,offset wordVal 00000004h

e.mov esi,offset aString 00000014h

f.mov al,aString+2 43h ('C')

g. mov edi,offset dwordVal 0000000Ch

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.)

a.mov eax,offset byteVal+2 00000002h

b.mov dx,wordVal+4 3000h

c.mov ecx,dwordVal+4 34567890h

d.mov esi,offset wordVal+4 00000008h

e.mov esi,offset aString-1 00000013h

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 00000006h

b.mov dx,wordVal+4 0002h

c.mov ecx,dwordVal+4 00000007h

d.mov si,dwordValSiz 0010h

e.mov 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.)

a.mov ax,dwordVal+2 I

b.mov dx,wordVal-2 5958h ("YX")*

c.mov eax,ptrByte 00000000h

d.mov esi,ptrWord 00000006h

e.mov edi,offset dwordVal+2 0000000Eh

** The two character bytes are automatically reversed when loaded into a 16-bit register.*

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]
```

Answers: Indirect and Indexed Operands

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]     I (operand size mismatch)

b.add  dx,[cx+wordVal]    I (CX is not a base
                          or index register)

c.mov  ecx,[edi+dwordVal]V

d.xchg al,[bx]            V

e.mov  ax,[bx+4]          V

f.mov  [bx],[si]          I (memory to memory
                             not permitted)

g.xchg al,byteVal[dx]     I (DX is not a base
                          or index register)
```

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]     4000h

c.mov bx,4
  mov ecx,[bx+dwordVal]  34567890h

d.mov si,offset aString
  mov al,byteVal+1
  mov [si],al            2

e.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          I (memory to memory
  xchg [si],wordVal     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 aStringI (esi and pntr
  xchg esi,pntr         have different
  mov  dl,[esi]         sizes)

e.mov esi,offset aString
  mov dl,[esi+2]        43h ('C')
```

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:

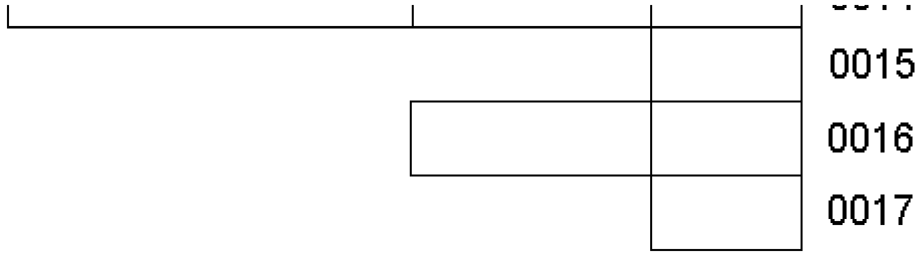
```
.data
arrayW   WORD 1234h,5678h,9ABCh
ptr1     WORD offset arrayD
arrayB   BYTE 10h,20h,30h,40h
arrayD   DWORD 40302010h
```

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 hexacecimal contents of every memory location with the correct 32-bit, 16-bit, and 8-bit values.

MEMORY MAP

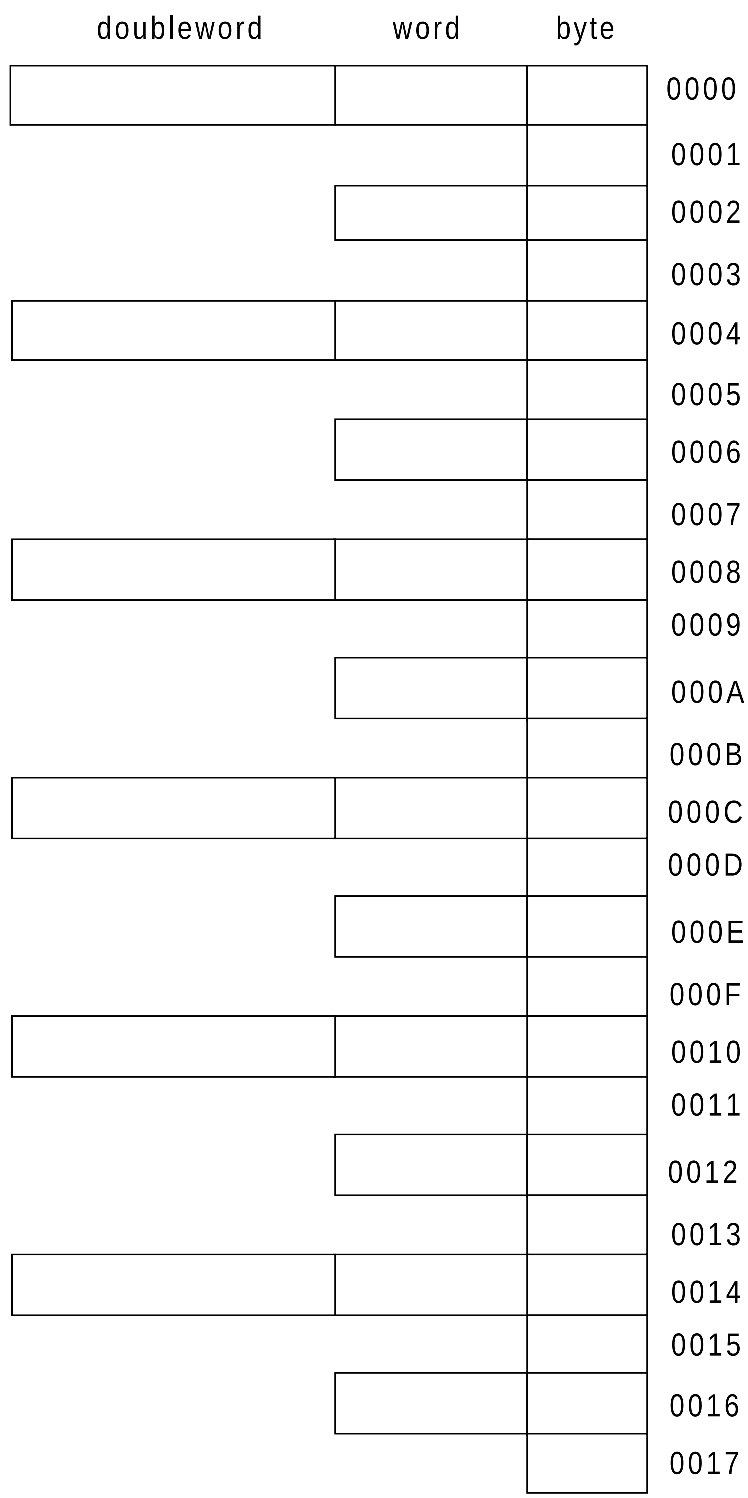
Write the names of variables next to their corresponding memory locations

doubleword	word	byte	
			0000
			0001
			0002
			0003
			0004
			0005
			0006
			0007
			0008
			0009
			000A
			000B
			000C
			000D
			000E
			000F
			0010
			0011
			0012
			0013
			0014


0015
0016
0017

MEMORY MAP

Write the names of variables next to their corresponding memory locations



MS-DOS Function Calls - 1

Required reading: Chapter 13

1. Write a program that inputs a single character and redisplays (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 42 30 38 NOPQRST.NNB08
```

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.

```
Title MS-DOS Example                                   (DOS1-1.ASM)

;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
```

```
Title MS-DOS Example                      (DOS1-3.ASM)

; 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
```

```
Title MS-DOS Example                  (DOS1-4.ASM)

;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 upper case
    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
```

```
Title MS-DOS Example 1            (DOS1-5.ASM)

;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
```

MS-DOS Function Calls - 2

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/200, 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 .

```
title MS-DOS Function Calls - 2      (DOS2-1.ASM)

;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
```

```
title MS-DOS Function Calls - 2          (DOS2-2.ASM)

;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

end main
```

```
title MS-DOS Function Calls - 2        (DOS2-4.ASM)

;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
```

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:

Data Bits :	**4**	**8**	**16**	**32**	**64**	**128**
Parity Bits:	**3**	**4**	**5**	**6**	**7**	**8**

```
Codeword :     7   12    21     38   71    136
```

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...}. 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
P	P		P				P								P					

The 16-bit data value 1000111100110101 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	1	P	0	0	0	P	1	1	1	1	0	0	1	P	1	0	1	0	1

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 long:

Data Bit	...is checked by parity bits
3	1, 2
5	1, 4
6	2, 4
7	1,2,4
9	1,8
10	2,8
11	1,2,8
12	4,8
13	1,4,8
14	2,4,8
15	1,2,4,8
17	1,16
18	2,16
19	1,2,16
20	4,16

21	1,4,16

(table 4)

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:

Parity Bit	Checks the following Data Bits	Hint*
1	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21	use 1, skip 1, use 1, skip 1, ...
2	2, 3, 6, 7, 10, 11, 14, 15, 18, 19	use 2, skip 2, use 2, skip 2, ...
4	4, 5, 6, 7, 12, 13, 14, 15, 20, 21	use 4, skip 4, use 4, ...
8	8, 9, 10, 11, 12, 13, 14, 15	use 8, skip 8, use 8, ...
16	16, 17, 18, 19, 20, 21	use 16, skip 16, ...

(table 5)

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:

1	2	3	4	5	6	7	8	9	10	11	12
P	P	**1**	P	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

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:

1	2	3	4	5	6	7	8	9	10	11	12
0	P	**1**	P	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

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:

1	2	3	4	5	6	7	8	9	10	11	12
0	**1**	**1**	P	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

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:

1	2	3	4	5	6	7	8	9	10	11	12
0	1	**1**	**0**	**1**	**0**	**0**	P	**1**	**1**	**1**	**1**

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:

1	2	3	4	5	6	7	8	9	10	11	12
0	1	**1**	0	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

All parity bits have been created, and the resulting code word is: 011010001111.

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:

Parity Bit	Bit Group
1	1, 3, 5, 7, 9, 11
2	2, 3, 6, 7, 10, 11
4	4, 5, 6, 7, 12
8	8, 9, 10, 11, 12

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 011110001111. 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:

Parity Bit	Bit Group
1	1, 3, 5, 7, 9, 11
2	2, 3, 6, 7, 10, 11
4	4, 5, 6, 7, 12
8	8, 9, 10, 11, 12

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:

Parity Bit	Bit Group
1	1, 3, 5, 7, 9, 11
2	2, 3, 6, 7, 10, 11
4	4, 5, 6, 7, 12
8	8, 9, 10, 11, 12

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:

Parity Bit	Bit Group
1	1, 3, 5, 7
2	2, 3, 6, 7
4	4, 5, 6, 7

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.

For an excellent introductory discussion of error-correcting codes, see Tanenbaum, Andrew. **Structured Computer Organization, Fourth Edition** (1999), pp. 61-64.

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 & Scmidt, 1984.

Boolean and Comparison Instructions

Click here to view the Answers

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.

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^b^c == ~(~a v ~b v ~c)

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

2.

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

3.

```
and  BL,10001111b
```

4.

```
or  CL,00011000b
```

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> <00A> ...

You can 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.

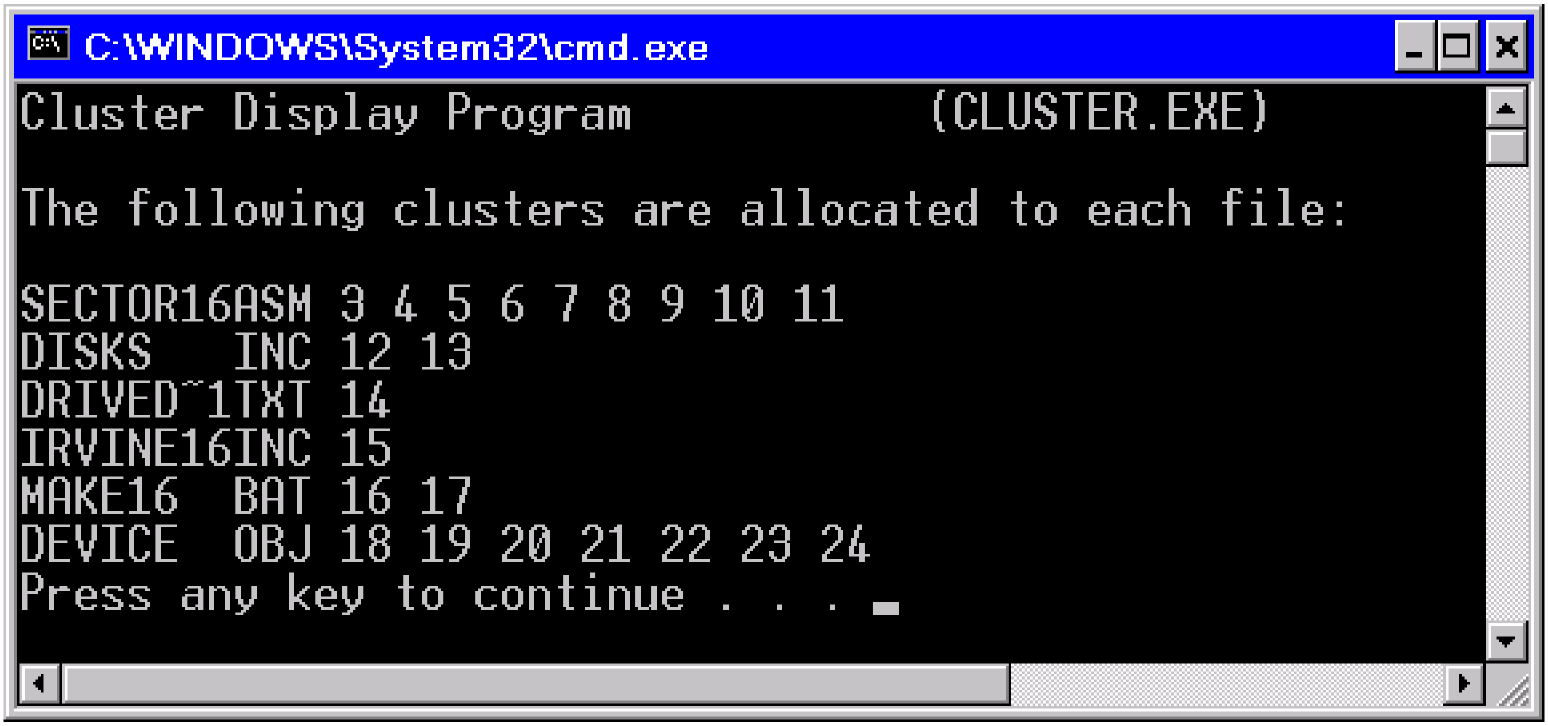
Finding the Starting Sector

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) * 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:



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 0Bh 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:
```

```
   FATSectors = 9              ; num sectors, first copy of FAT
   DIRSectors = 14             ; num sectors, root directory
   DIR_START = 19              ; starting directory sector num

SECTOR_SIZE = 512
   DRIVE_A = 0
   FAT_START = 1               ; starting sector of FAT
   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
   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                                    ; choose decimal radix
   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 upt 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
```