# Variables & Data Structures

# Lab Manual, Chapter Five

To write the shortest and fastest possible assembly language programs you need to understand how the CPU references data in memory. The Intel 80x86 processor family provides a wide variety of memory addressing modes that allow efficient access to memory. Unless you master these addressing modes, you will not be able to write the most efficient programs.

The 80386 and later processors support an extended set of memory addressing modes. For those working on '386 (or greater) processors, the task is both simpler and more complex. On the one hand, the greater variety of addressing modes opens even more opportunity for optimization. On the other hand, the additional modes complicate the task of selecting the most appropriate mode.

In this laboratory you will experiment with the PC's memory and study the various 80x86 addressing modes. You will also explore various high level language data types and their implementation in assembly language. To support these experiments, you will learn how to use the Microsoft CodeView™debugger, the 80x86 version of SIM886. Finally, you will begin writing real assembly language programs, assembling and linking them with the MASM 6.x assembler.

## 5.1 The LEA, LES, ADD, and MUL Instructions

You'll need to use the LEA, LES, ADD, and MUL instructions in this chapter's lab exercises, so it's worthwhile to briefly covering these instructions. A few examples may help demystify their use.

The LEA (load effective address) instruction loads a 16-bit register with the address of some specified memory location. This instruction takes the form:

```
LEA     reg16, memory
```

$Reg_{16}$ is one of the 8086's 16-bit general purpose registers and memory represents a memory addressing mode (any **mod-r/m** value where **mod** is not 11). This instruction computes the effective address of the memory operand (that is, the offset into the given segment) and loads that effective address into the specified register.

Suppose BX=100h, BP=200h, SI=10h, and DI=20h. The following examples demonstrate how LEA works:

| | |
|---|---|
| `LEA AX, DS:[105h]` | Loads AX with 105h. |
| `LEA AX, 5[BX][si]` | Loads AX with BX+SI+5 or 115h. |
| `LEA BX, 5[BX]` | Loads BX with BX+5 or 105h. |
| `LEA BX, [BX]` | Simply copies BX back into itself. |
| `LEA AX, [BP][DI]` | Loads AX with BP+DI or 220h. |
| `LEA AX, 10h[SI]` | Loads AX with SI+10h or 20h. |

**Given these values for BX, BP, SI, and DI, describe what the following LEA instructions will do:**

**5.1** `LEA BX, [BX][SI]`

**5.2** `LEA SP, [BP]`

**5.3** `LEA DX, DS:[1027H]`

**5.4 The opcode for the LEA instruction is 8Dh followed by a** mod-reg-r/m **byte and any necessary displacement bytes. Given this encoding, what are the instruction bytes for** `LEA AX, 5[BX][SI]` **?**

___

The LES instruction, although it looks like the LEA mnemonic, is a completely different instruction. It loads a 32-bit pointer into the ES register and some other 16-bit register. The syntax for this instruction is

$$\texttt{LES reg}_{16}\texttt{, memory}_{32}$$

The $memory_{32}$ operand means that the LES instruction loads 32 bits from four consecutive bytes in memory (L.O. byte first). The L.O. word goes into the 16-bit register, the H.O. word goes into the ES register. The main use of this instruction is to load memory pointers into ES and some other register (typically BX, SI, or DI) to gain access to the object referenced by the pointer.

As an example, suppose memory locations ds:0 through DS:7 contain 0, 2, 5, 6, 1, 2, 6, 4, respectively. The "`LES BX, DS:[0]`" would load BX with 200h (the word starting at location 0) and ES with 605h (the value of the word starting at location two). Likewise, "`LES SI, DS:[4]`" would load SI with 201h and ES with 406h.

**5.5 Given the above values for DS:0…DS:7, what would "`LES DI, DS:[2]`" do?**

___

The ADD instruction, as its name implies, adds two values together. It's syntax is almost identical to that of the MOV instruction's. The only major difference (other than using ADD rather than MOV) is that you cannot add a value to a segment register. This instruction adds the source and destination operands together and stores the sum into the destination operand. For example, the "ADD AX, 2" instruction adds two to the value in the AX register and leaves the result in the AX register.

**5.6 If the opcode for the ADD instruction is 000000dw (where "d" and "w" have the same meanings as for the MOV instruction), followed by a mod-reg-r/m byte and any necessary displacement bytes, what is the instruction encoding for "`ADD AX, BX`"?**

___

**5.7 If AX contains five and BX contains two, what will the instruction "`ADD AX, BX`" do?**

___

The 80x86 MUL instruction uses a slightly different syntax than the instructions you've seen thus far. Rather than having two operands, a source and a destination, the MUL instruction has only a single operand – the source operand. The destination for the 80x86 MUL instruction is always the AX register (if the source operand is an eight-bit register or memory location) or the DX:AX register pair (if the operand is 16 bits). These instructions do the following:

| | |
|---|---|
| MUL $reg_8/mem_8$ | Multiplies AL by operand, stores the result into AX. |
| MUL $reg_{16}/mem_{16}$ | Multiplies AX by operand, stores the result into DX:AX |

The following questions assume AX = 2, BX=3, CX=4, and DX=5 *at the beginning of each question.* Please specify the exact and complete results for each of the following:

**5.8 MUL AX**

___

**5.9 MUL BX**

___

5.1 BX := BX + SI

## 5.2 Variables in an Assembly Language Program

MASM provides many *assembler directives* specifically for declaring scalar variables. These directives are

5.2 SP := BP

- DB, BYTE, and SBYTE for declaring byte variables,
- DW, WORD, and SWORD for declaring word variables,
- DD, DWORD, and SDWORD for declaring double word variables,
- REAL4, REAL8, and REAL10 for declaring floating point variables, and
- DF/FWORD, DQ/QSORD, and DT/TBYTE for other data types.

The syntax for each of these directives is identical. Using the BYTE directive as an example:

```
variable_name            byte          ?
```

If you want a word variable rather than a byte variable, you would substitute *word* for *byte* above. Likewise, you would substitute *dword* for *byte* if you wanted a double word variable. For example, to declare a 16-bit signed variable named *CurValue* you could use the declaration:

```
CurValue                 sword         ?
```

The DB, DW, DD, DF, DQ, and DT mnemonics are older, obsolete versions of BYTE, WORD, DWORD, etc. In general, you should use the new, easier to read versions rather than these. BYTE, WORD, and DWORD are for declaring *unsigned* variables. SBYTE, SWORD, and SDWORD let you declare *signed* variables. MASM ignores the signed/unsigned specification, but CodeView uses this information to properly display values during debugging.

**5.10 How would you declare an eight-byte floating point variable "R"?**

**5.11 How would you declare an unsigned 32-bit variable "U"?**

The question mark in the operand field tells MASM that you don't want the variable initialized when the program loads into memory. If you want to give the variable an initial value, specify that value in the operand field. As an example, the following initializes the CHR variable to the character 'A':

```
CHR                     byte          'A'
```

**5.12 How would you declare a signed 16-bit variable ("S") and initialize it to the value -129?**

**5.13 How would you declare a four byte floating point variable "PI" and initialize it to 3.14159?**

Generally, you will place all variables in the data segment of your program. When using the SHELL.ASM program you will almost always put your variables in the DSEG segment, e.g.,

```
dseg            segment         para public 'data'
bytevar         byte            ?
wordvar         word            ?
dwordvar        dword           ?
byte2           sbyte           ?
word2           sword           ?
dseg            ends
```

## 5.3 Declaring Your Own Types with TYPEDEF

The TYPEDEF directive lets you create your own data type directives. The TYPEDEF directive is especially useful for declaring pointer types (see the next section), you can also use this directive to create your own names for common types. For example, if you prefer *integer* to *sword* when declaring integer variables, you could create your own type as follows:

```
integer         typedef         sword
```

To declare a variable, I, of type integer, you would use the declaration:

```
I               integer         ?
```

You could even initialize I by specifying a value in the operand field:

```
I               integer         -13
```

Likewise, if you prefer "float" to REAL4 or DOUBLE to "REAL8" (i.e., you're a "C" programmer) you can create such types using the declarations:

```
FLOAT           typedef         real4
DOUBLE          typedef         real8
```

You can declare FLOAT and DOUBLE variables using statements like:

```
F               FLOAT           ?
D               DOUBLE          3.19
```

**5.14 How would you declare a "char" type which reserves storage for a one-byte character variable?**

____________________________________________________________

**5.15 Give an example of how to declare a variable "chr" initialized with the character "A" using the above declaration.**

____________________________________________________________

## 5.4 Pointers

A pointer is a memory location (generally 16 or 32 bits) which contains the address of some other object in memory. This text will typically use 32-bit (far) pointers since they mesh well with the UCR Standard Library. Keep in mind, though, that 16-bit (near) pointers are more efficient if you are able to use them..

To declare a far (32-bit) pointer in your program, just use the **dword** directive (or some typedef'd equivalent) and declare the pointer as you would any other variable:

```
pointer         typedef         far ptr
fptr1           dword           ?
fptr2           pointer         ?
```

You can initialize a pointer with the address of an object by placing that object's name in the operand field of the declaration:

```
I               word            10
Ptr2I           pointer         I
Ptr2Ptr2I       pointer         Ptr2I
```

The 80x86 family does not let you access objects through memory pointers directly[8]. To access an object referenced by a pointer you must use one of the 80x86's indirect or indexed addressing modes. When dealing with far pointers, you would typically use the ES:[BX], ES:[SI], or ES:[DI] addressing modes to access objects referenced by far pointers.

**5.16 Suppose you have an integer variable J and you want to create a pointer, PJ, to this variable. How could you declare such a pointer so that it will be properly initialized when loaded into memory**

______________________________________________

**5.17 What 80x86 instruction would you use to load the 32-bit pointer PJ into ES and BX?**

______________________________________________

## 5.5 Arrays in Assembly Language Programs

Assembly language programmers generally implement arrays as a contiguous set of memory locations. In general, this provides the most efficient mechanism for accessing elements of an array. Therefore, this is the technique we will use to implement them.

There are a couple of ways to declare an array in your assembly language program.. The first and easiest way is to use the MASM *dup* operator:

```
IntArray        word            16 dup (?)
```

The example above reserves 16 words of storage. The name "IntArray" is a word variable whose address just happens to be the very first word of the array (this is the *base address* of the array). Because each element of the array is two bytes long, you must multiply the index by two when attempting to access elements of this array. IntArray[0] refers to the first element of the array, IntArray[2] is the second element, IntArray[4] is the third element, IntArray[6] is the fourth element, and so on. A very common mistake beginners make when writing assembly language programs is that they forget to multiply the index by the size of an array element when computing an index into an array. Because the notation "IntArray[2]" looks just like the notation high level languages use, it's very easy to forget the real computation which must take place. The complete formula for this computation is

```
Element_Address = Base_Address + Index * Element_Size
```

*Element_Size* is the size in bytes of one item in the array. Since the element size of a word array is two bytes, Element_Size is two, hence you must multiply your index by two before adding it to the base address[9].

Even more confusing is the fact that this multiplicative factor changes with the size of the array elements. If you have an array of bytes, the multiplication factor is one and you can ignore it. For words, the factor is two. The factor is four when accessing double word arrays. If you are manipulating several different data types, keeping the multiplication factors straight can be a problem.

Consider the following array declaration:

8. Or should this be *indirectly*?

9. Remember, the notation XYZ[ABC] in assembly language means XYZ+ABC. Hence the notation IntArray[2] means "add two to the base address of IntArray."

5.3 DX := 1027h

5.4 8Dh, 40h, 05h

5.5 Load 605h into DI and 201h into ES.

5.6 03h, C3h

5.7 Set AX to seven.

5.8 Set DX = 0 & AX = 4

5.9 Set AX = 6 & DX = 0

5.10 R real8 ?

5.11 U DWORD ?

5.12 S SWORD -129

5.13 PI real4 3.14159

```
IntArray       word   16 dup (?)
```

You can access elements of this array using 80x86 instructions like the following:

```
mov    IntArray[4], 0
mov    IntArray[8], ax
add    ax, IntArray[2]
mul    IntArray[0]
lea    bx, IntArray[8]
```

(the last instruction loads BX with the offset of the fifth element from IntArray.)

**5.18 What instruction would you use to load the last element of IntArray into BX?**

______________________________________________________________________

**5.19 How would you declare an array, RArray, containing 64 single precision (32-bit) floating point variables?**

______________________________________________________________________

**5.20 What instruction would you use to load the offset of RArray[10] into BX?**

______________________________________________________________________

More often than not, you won't need to access a fixed element of an array as in the above examples. Instead, you'll probably have a value in a variable or in a register which you'll use to specify which element to operate on. You cannot use an operand of the form "IntArray[i]" to select the $i^{th}$ element of the array. Instead, you will have to compute the index into the array using the formula given above. For example, to access the $i^{th}$ element of IntArray, you will need to multiply the index by two (the element size is two). This product plus the base address of the array provides the address of the $i^{th}$ element. Some 80x86 code to accomplish this is

```
mov    bx, i                 ;Get the index into the array
add    bx, bx                ;Multiplies the index by two.
mov    ax, IntArray[bx]      ;Fetch the specified element.
```

There are two important things to note above. First, this example used the ADD instruction to multiply i's value by two before using it. While it could have used the multiply instruction, the ADD instruction is simpler, faster, and easier to use. You can use sequences of the ADD instruction to easily multiply a value in a register by two, four, eight, or any other power of two. The other thing to notice above is that to access an array element, this code uses the indexed addressing mode. The 80x86 does not provide a memory addressing mode which lets you use an integer variable directly as an index into an array. Instead, you must first load the index value into an appropriate register and use one of the indexed addressing modes.

Note that if you're using an 80386 or later processor and your array element size is two, four, or eight you can use the 80386 *scaled indexed addressing modes* to automatically perform this multiplication for you. Since these are the most common array element sizes, the scaled indexed addressing mode can be very useful. The previous example, using the scaled indexed addressing mode takes only two instructions:

```
mov    ebx, i                      ;"i" must be a dword variable!
mov    ax, IntArray[ebx*2]
```

**5.21 What statement would you use to declare an array, DArray, of 128 double precision (64-bit) floating point values?**

______________________________________________________________________

**5.22 What sequence of instructions would you use to access DArray[j] (8086 instructions only)?**

______________________________________________________________________

**5.23 What (shorter) sequence of instructions could you use on the 80386 to access DArray[j]?**

______________________________________________________________________

5.14 typedef byte

The MASM *dup* operator simply tells the assembler to duplicate the operand inside the parentheses. You may also specify multiple operands in the operand field and define an array in this fashion. This allows you to *initialize* the elements of the array before the program runs:

```
IntArray          word        0,1,2,3,4,5,6,7
                  word        8,9,10,11,12,13,14,15
```

The declaration above sets aside 16 words in memory and initializes them with the values zero through fifteen. You would use the same 80x86 code to access these array elements as before.

5.15 chr char 'A'

Note that there is nothing magic about an array declaration. All you're doing is reserving some storage. When you access an element of an array, the 80x86 simply accesses a memory location at a given offset from the base address you supply. Consider the following variable declarations in the data segment:

```
I                 word        0
J                 word        1
K                 word        2
L                 word        3
M                 word        4
N                 word        5
O                 word        6
P                 word        7
```

5.16 PJ dword J

Now consider the following sequence of instructions:

```
                  mov         bx, Index
                  add         bx, bx
                  mov         I[bx], ax
```

The sequence above is a very typical set of instructions you'd normally use to index into a word array whose base address is "I". However, as you can probably tell, "I" isn't really an array. It's just a word variable. The 80x86 doesn't care though, it will happily index off "I" and return the word at offset "Index*2" beyond "I". It *is* quite possible that "I" really is an array and the programmer wanted to access elements of "I" using names like "J", "K", etc. More likely than not, however, if you see code like the above in a program, it's an indication that there are problems with that program.

5.17 LES bx, PJ

**5.24 Suppose INDEX = 4, what variable would the above code access?**

## 5.6 Multidimensional Arrays

The 80x86 hardware is set up to handle one-dimensional arrays with ease. Handling two or more dimensions is a bit more work. While there are a wide variety of ways to map multidimensional structures to the one-dimensional structure of memory, there are two techniques in common use: row-major ordering and column major ordering. We'll use row major ordering most of the time, though column major ordering is useful on occasion. Here we will concentrate on two-dimensional arrays, for a more general discussion, please consult the text.

The row major function that maps two values (indices) to a linear offset is

```
ElementAdrs = BaseAdrs + (ColIndex * RowSize + RowIndex) * ElementSize
```

For column major ordering the formula is

```
Element_Adrs = BaseAdrs + (RowIndex * ColSize + ColIndex) * ElementSize
```

To declare a multidimensional array use multiple *dup* operators as follows:

```
TwoDArray     word   4 dup (4 dup (?))
```

The dup operator duplicates everything inside the parentheses. "4 dup (0,1,2,3)" duplicates the four values 0, 1, 2, and 3 four times for a total of 16 values (0, 1, 2, 3, 0, 1, 2, 3, ..., 2, 3). Likewise "4 dup (4 dup (0))" says to duplicate "4 dup (0)" four times, to produce a total of 16 zeros. The array declaration above reserves storage for 16 words. Of course, you could also declare a 4x4 array using the declaration

```
TwoDArray      word   16 dup (0)
```

However, the former declaration is a little clearer as to its intent.

To access element TwoDArray[i][j] (row major order) you would use 80x86 code like the following

```
mov     bx, i
add     bx, bx          ;Multiply by row size
add     bx, bx          ;*4
add     bx, j           ;+ row index
add     bx, bx          ;* Element Size (2)
mov     ax, TwoDArray[bx]
```

**5.25 How would you declare the array equivalent to "a:array [0..15][0..15] of integer;" in assembly language?**

________________________________________________________________________

**5.26 What is the code to load "a[i][j]" into ax (assume column major ordering)?**

__________________________ _____________________________________

__________________________ _____________________________________

__________________________ _____________________________________

## 5.7 Structures

An array is a contiguous homogeneous collection of objects[10]. A structure is a contiguous heterogeneous collection of objects in memory. Structures let you easily associate values which are logically related, yet of differing types, by placing these values in contiguous memory locations.

Structures are mainly an assembly language convention. When viewing structures in memory there is really no difference between a structure and a sequence of independent variables, other than that the elements of a structure always occupy contiguous locations. However, structures provide considerable value in an assembly language program where you may refer to the fields of the structure using a high level syntax.

To declare a structure *type* with MASM you use the STRUCT and ENDS directives. The following template provides the basic format:

```
StructureName             STRUCT

                    <Field Definitions>

StructureName             ENDS
```

The <Field Definitions> section contains standard MASM variable declarations (using BYTE, WORD, DWORD, etc.). The following example demonstrates a structure for a complex number:

```
complex                   struct
Real                      real8          ?
Imaginary                 real8          ?
complex                   ends
```

To declare a variable of type complex, you could use a declaration like the following:

10. That is, all elements of the array are the same type.

```
Vector                    complex        {}
```

Any initial values you want to supply for the variable "Vector" must appear inside the braces. For example, if you want the assembler to preinitialize "Vector" to (1.0, -1.0) you would use the declaration:

```
Vector                    complex        {1.0, -1.0}
```

If, by default, you want to initialize *all* variables of type complex to the same value, you could define "complex" as follows:

```
complex                   struct
Real                      real8          1.0
Imaginary                 real8          -1.0
complex                   ends
```

You can still override this default initialization by specifying the initial values in the braces as above.

**5.27 Create a structure for a type *string* which contains two fields: *length* which is a single byte and *chars* which is an array of 80 bytes.**

______________ ______________________

______________ ______________________

______________ ______________________

______________ ______________________

______________ ______________________

It is important to note that a structure definition (using the STRUCT and ENDS directives) does *not* create a structure variable for you. All it does is create a type for declaring variables. To actually create a structure variable, you must use the structure's data as a data definition directive as with the "Vector" example above.

To access a field of a structure variable you use a syntax similar to that of high-level languages like "C" or Pascal. A name of the form *variable.field* selects the specified field in the structure. "Vector.Imaginary" selects the "Imaginary" field from the "Vector" variable.

Since MASM stores successive fields in contiguous memory locations, you can think of the field names as *offsets* from the base address of a structure. The *base address* of a structure is the address of the first element of that structure, which corresponds to the name of the structure variable. In many respects, the *variable.field* syntax is comparable to *variable[field]* since both mechanisms compute the address of the specified object by adding the address of variable with field. However, MASM will not allow the brackets operator on structure names.

**5.28 How would you declare a variable of type "string" (see the previous question) named "Identifier" in your data segment?**

____________________________________

**5.29 How would you declare the "Identifier" variable above, initializing the "Length" field to zero?**

____________________________________

5.18 mov bx, IntArray[30]

5.19
RArray real4 64 dup (?)

5.20 lea bx, RArray[40]

5.21
DArray real8 128 dup (?)

5.22
mov bx, J
add bx, bx
add bx, bx
add bx, bx
mov ax, DArray [bx]

or

mov bx, J
mov ax, 8
mul bx
mov bx, ax
mov ax, DArray [bx]

5.23 Assume J is 32 bits.
mov ebx, J
mov eax, DArray[ebx*8]

5.24 M

**5.30 What instruction could you use to load the value of the "length" field above into the AL register?**

_______________________________________________________________________

## 5.8 Memory Organization Laboratory Exercises

In this laboratory you will examine how the 80x86 family organizes values in memory. You will also create several data structures in memory and examine them with the CodeView debugger. Finally, you will also assemble and link some very simple assembly language programs and load them into memory with the CodeView debugger.

### 5.8.1 Before Coming to the Laboratory

Your pre-lab report should contain the following:

- A copy of this lab guide chapter with all the questions answered and corrected.
- A write-up on the CodeView debugger explaining, in your own words, how the following commands work in CodeView: A, D, E (Enter), F, G, I (input), M (Move), O (Output), Q, R, T, and U.
- A write-up explaining how the MOV, ADD, LEA, LES, and MUL instructions work.

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Teaching Assistant or Lab Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

### 5.8.2 Laboratory Exercises

In this laboratory you will perform the following activities:

- Demonstrate the use of the CodeView debugger and many of the commands in the debugger
- Demonstrate the operation of the 8086 MOV, LEA, LES, ADD, and MUL instructions and addressing modes.
- Enter several 8086 machine language programs into the CodeView and single step through the programs to execute them.
- Use the debugger to modify the operation of the programs.
- Examine memory locations using CodeView and explore the memory organization of the 8086
- Create various data structures (i.e., arrays and structs) and explore their memory organization.
- Create simple 8086 programs to access the above data structures.
- Explore the various encodings of 8086 instructions.

❏ Exercise 1: Create an array of the form "A:array [0..3, 0..4] of word;" starting at location 8000:0. Initialize the array elements as you did for exercise 5 with the values 0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22 , 23, 24, 30, 31, 32, 33, and 34.. Use row major ordering when creating the array. Dump the memory using the CodeView Dump command to the printer or a file. Include this printout in your lab report and mark each row of the array. Enter the following short machine language program using CodeView's Assemble command. It uses the word at location 8000:80 as the first index (the row number) and location 8000:82 as the second index (the column number) and loads the word at that address into AX. Don't forget that DS must contain 8000h before running this code.

```
mov     ax, ds:[80h]         ;Get row number (column index)
mov     bx, 5                ;Multiply by the size of a row
mul     bx
add     ax, ds:[82h]         ;Add in the column number (row index)
mov     bx, ax
```

```
                add     bx, bx                  ;Element size is two
                mov     ax, ds:0[bx]            ;Fetch desired array
element.
```

5.25 A word 16 dup (16 dup (?))

**For your lab report:** Dump the array and annotate each element of the array. Run the code above with a reasonable pair of values in locations 8000:80 and 8000:82. Capture the results and include them in your lab report. Explain the results.

**For additional credit**: Run the code above with several different values in locations 8000:80 and 8000:82. Capture the output and describe the results in your lab report.

- ❏ Exercise 2: Repeat exercise 7 using *column major ordering*. Use the following code for this example:

```
        mov     bx, ds:[80]             ;Fetch first index
        add     bx, bx                  ;Multiply by column size (4)
        add     bx, bx
        add     bx, ds:[82]             ;Add in second index
        add     bx, bx                  ;Multiply by element size (2)
        mov     ax, ds:0[bx]    ;Fetch array element
        int     3               ;Stop inside CodeView.
```

5.26
mov bx, j
add bx, bx
add bx, bx
add bx, i
add bx, bx
mov ax, a[bx]

- ❏ Exercise 3: Make a copy of the LAB4.ASM file on the disk accompanying this lab manual. Name the file "LAB4_1.ASM". In the data segment (DSEG) create an array named "A1" which is a single dimensional array of 128 bytes. Create a second array, named "A2" which is a 4x4 array of words. Create a third array, named "A3" which is a 3x3 array of words initialized to the value 00, 01, 02, 10, 11, 12, 20, 21, 22. Use row major ordering for these arrays. Finally, declare two word variables, "I" and "J", and initialize them to zero in the data segment. Now, enter the code into the main program (after the comment which states "Enter your main program here.") which copies A1[i] to A2[i][j] and then copies A2[i][j] to A3[j][i]. Terminate your program with the INT 3 instruction. Use row major ordering for A2 and A3. Next, assemble your code using the following DOS command to run MASM:

5.27
string struct
length byte ?
chars byte 80 dup (?)
string ends

ml /Zi lab4_1.asm

Note that the '/Zi" must be an uppercase "Z" and a lowercase "i". the "/Zi" command line parameter tells MASM to include *source information* in the .EXE file. This will produce a file named "LAB4_1.EXE" which you can load into CodeView using the DOS command:

CV LAB4_1

5.28
Identifier string {}

Single step through the code and verify that it works correctly.

**For your lab report:** Include a printout of the Lab4_1.asm file. Heavily comment each instruction you add to this program to describe its purpose.

**For additional credit:** MASM lets you create an *assembly listing* of your source code. Use the DOS command "ml /?" to get a list of legal MASM command line options. Determine which one lets you produce and assembly listing. Include the assembly listing with your lab report.

- ❏ Exercise 4: Add a structure to the Lab4_1.ASM program and create a structure variable in the data segment (DSEG). Modify your main program to load various fields of this structure into the 80x86 registers. Use the CodeView Unassemble command to look at the raw machine code produced by the assembler.

5.29
Identifier string {0}

**For your lab report:** Comment on what you see. Include appropriate screen dumps/ captures in your lab report.

- ❏ Exercise 5: Connect the circuit you built for labs two and three to the parallel printer port on your computer. Use the following output command to write a value to the LEDs on your circuit:

O *port value*

*Port* is the base address of the parallel port to which you've connected your circuit. To determine the appropriate port address, dump memory locations 40:8 through 40:d. The first two locations (40:8 and 40:9) contain the base I/O address for LPT1:. The second two locations (40:a and 40:b) contain the base I/O address for LPT2:. The last pair of locations contain the base address for LPT3:. Typical addresses are 378h, 278h, and 3BCh. If a zero appears in one of these words, then the system did not recognize the associated device. Be sure to select the appropriate port value for your connection.

The *value* operand is the value to write to the printer port. For example, if you've connected your circuit to LPT1: and it is at I/O address 378h (i.e., the word at location 40:8 contains 378h), you can turn off all the LEDs with the command **O 378 0.** Likewise, you can turn on all the LEDs using the command **O 378 ff**. Try turning on each of the LEDS individually with a set of eight "O" commands.

**For your lab report:** Describe the use of this command to turn on and off various LEDs on your circuit.

## 5.9 Sample Programs

This section contains several sample programs that demonstrate the concepts in Chapter Four of the textbook. Each of these short programs can be found on the diskette accompanying this lab manual. These programs all assemble and run, although you should run them from the CodeView debugger since they do not produce any output.

### 5.9.1 Sample Program #1: Simple Variable Declarations

This short program demonstrates how to declare byte, word, and double word global variables. It also demonstrates how to use the typedef directive to create your own variable types. The program also declares some simple pointer variables and the main program accesses data indirectly using those pointers. Finally, this short sample program also demonstrates how to initialize variables you declare in the data segment.

```
; Sample variable declarations
; This sample file demonstrates how to declare and access some simple
; variables in an assembly language program.
;
; Randall Hyde
;
;
; Note: global variable declarations should go in the "dseg" segment:

dseg            segment    para public 'data'

; Some simple variable declarations:

character       byte       ?                 ;"?" means uninitialized.
UnsignedIntVar  word       ?
DblUnsignedVar  dword      ?

;You can use the typedef statement to declare more meaningful type names:

integer         typedef    sword
char            typedef    byte
FarPtr          typedef    dword

; Sample variable declarations using the above types:

J               integer    ?
c1              char       ?
PtrVar          FarPtr     ?
```

5.30
mov al, Identifier.Length

```
; You can tell MASM & DOS to initialize a variable when DOS loads the
; program into memory by specifying the initial value in the operand
; field of the variable's declaration:

K               integer    4
c2              char       'A'
PtrVar2         FarPtr     L                  ;Initializes PtrVar2
with the
                                              ; address of K.


; You can also set aside more than one byte, word, or double word of
; storage using these directives.  If you place several values in the
; operand field, separated by commas, the assembler will emit one
byte,
; word, or dword for each operand:

L               integer    0, 1, 2, 3
c3              char       'A', 0dh, 0ah, 0
PtrTbl          FarPtr     J, K, L

; The BYTE directive lets you specify a string of characters byte
enclosing
; the string in quotes or apostrophes.  The directive emits one byte
of data
; for every character in the string (not including the quotes or
apostrophes
; that delimit the string):

string          byte       "Hello world",0dh,0ah,0


dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

Main            proc
                mov        ax, dseg           ;These statements are
provided by
                mov        ds, ax             ; shell.asm to initialize
the
                mov        es, ax             ; segment register.


; Some simple instructions that demonstrate how to access memory:

                lea        bx, L              ;Point bx at first word
in L.
                mov        ax, [bx]           ;Fetch word at L.
                add        ax, 2[bx]          ;Add in word at L+2 (the
"1").
                add        ax, 4[bx]          ;Add in word at L+4 (the
"2").
                add        ax, 6[bx]          ;Add in word at L+6 (the
"3").
```

```
                mul     K                   ;Compute (0+1+2+3)*123.
                mov     J, ax               ;Save away result in J.

                les     bx, PtrVar2         ;Loads es:di with address of L.
                mov     di, K               ;Loads 4 into di
                mov     ax, es:[bx][di]     ;Fetch value of L+4.


; Examples of some byte accesses:

                mov     c1, ' '             ;Put a space into the c1 var.
                mov     al, c2              ;c3 := c2
                mov     c3, al

Quit:           mov     ah, 4ch             ;Magic number for DOS
                int     21h                 ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

## 5.9.2 Sample Program #2: Using Pointers

This brief program demonstrates how to declare and use near and far pointers in an assembly language program.

```
; Using Pointer Variables in an Assembly Language Program
;
; This short sample program demonstrates the use of pointers in
; an assembly language program.
;
; Randall Hyde

dseg            segment para public 'data'


; Some variables we will access indirectly (using pointers):

J               word    0, 0, 0, 0
K               word    1, 2, 3, 4
L               word    5, 6, 7, 8

; Near pointers are 16-bits wide and hold an offset into the current data
; segment (dseg in this program).  Far pointers are 32-bits wide and hold
; a complete segment:offset address.  The following type definitions let
; us easily create near and far pointers

nWrdPtr         typedef near ptr word
fWrdPtr         typedef far ptr word


; Now for the actual pointer variables:
```

```
Ptr1            nWrdPtr    ?
Ptr2            nWrdPtr    K                  ;Initialize with K's
address.
Ptr3            fWrdPtr    L                  ;Initialize with L's
segmented adrs.

dseg            ends



cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

Main            proc
                mov        ax, dseg           ;These statements are
provided by
                mov        ds, ax             ; shell.asm to initialize
the
                mov        es, ax             ; segment register.


; Initialize Ptr1 (a near pointer) with the address of the J
variable.

                lea        ax, J
                mov        Ptr1, ax

; Add the four words in variables J, K, and L together using pointers
to
; these variables:

                mov        bx, Ptr1           ;Get near ptr to J's 1st
word.
                mov        si, Ptr2           ;Get near ptr to K's 1st
word.
                les        di, Ptr3           ;Get far ptr to L's 1st
word.

                mov        ax, ds:[si]        ;Get data at K+0.
                add        ax, es:[di]        ;Add in data at L+0.
                mov        ds:[bx], ax        ;Store result to J+0.

                add        bx, 2              ;Move to J+2.
                add        si, 2              ;Move to K+2.
                add        di, 2              ;Move to L+2.

                mov        ax, ds:[si]        ;Get data at K+2.
                add        ax, es:[di]        ;Add in data at L+2.
                mov        ds:[bx], ax        ;Store result to J+2.

                add        bx, 2              ;Move to J+4.
                add        si, 2              ;Move to K+4.
                add        di, 2              ;Move to L+4.

                mov        ax, ds:[si]        ;Get data at K+4.
                add        ax, es:[di]        ;Add in data at L+4.
                mov        ds:[bx], ax        ;Store result to J+4.

                add        bx, 2              ;Move to J+6.
                add        si, 2              ;Move to K+6.
                add        di, 2              ;Move to L+6.

                mov        ax, ds:[si]        ;Get data at K+6.
                add        ax, es:[di]        ;Add in data at L+6.
```

```
                mov       ds:[bx], ax       ;Store result to J+6.

Quit:           mov       ah, 4ch           ;Magic number for DOS
                int       21h               ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment   para stack 'stack'
stk             byte      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end       Main
```

## 5.9.3 Sample Program #3:Single Dimension Arrays

This short program demonstrates how to declare, initialize, and access elements of single dimensional arrays.

```
; Sample array declarations
; This sample file demonstrates how to declare and access some single
; dimension array variables in an assembly language program.
;
; Randall Hyde


                .386                                ;Need to use some 80386 addressing
                option    segment:use16     ; modes.

dseg            segment   para public 'data'

J               word      ?                 ;We will use these variables as the
K               word      ?                 ; indexes into the arrays.
L               word      ?
M               word      ?

JD              dword     0
KD              dword     1
LD              dword     2
MD              dword     3

; Some simple uninitialized array declarations:

ByteAry         byte      4 dup (?)
WordAry         word      4 dup (?)
DwordAry        dword     4 dup (?)
RealAry         real8     4 dup (?)


; Some arrays with initialized values:

BArray          byte      0, 1, 2, 3
WArray          word      0, 1, 2, 3
DWArray         dword     0, 1, 2, 3
RArray          real8     0.0, 1.0, 2.0, 3.0
```

```
; An array of pointers:

PtrArray        dword       ByteAry, WordAry, DwordAry, RealAry

dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg            ;These statements are
provided by
                mov         ds, ax              ; shell.asm to initialize
the
                mov         es, ax              ; segment register.


; Initialize the index variables.  Note that these variables provide
; logical indices into the arrays.  Don't forget that we've got to
; multiply these values by the element size when accessing elements
of
; an array.

                mov         J, 0
                mov         K, 1
                mov         L, 2
                mov         M, 3

; The following code shows how to access elements of the arrays using
; simple 80x86 addressing modes:

                mov         bx, J               ;AL := ByteAry[J]
                mov         al, ByteAry[bx]

                mov         bx, K               ;AX := WordAry[K]
                add         bx, bx              ;Index*2 since this is a
word array.
                mov         ax, WordAry[bx]

                mov         bx, L               ;EAX := DwordAry[L]
                add         bx, bx              ;Index*4 since this is a
double
                add         bx, bx              ; word array.
                mov         eax, DwordAry[bx]

                mov         bx, M               ;BX :=
address(RealAry[M])
                add         bx, bx              ;Index*8 since this is a
quad
                add         bx, bx              ; word array.
                add         bx, bx
                lea         bx, RealAry[bx]     ;Base address + index*8.


; If you have an 80386 or later CPU, you can use the 386's scaled
indexed
; addressing modes to simplify array access.
```

```
                mov         ebx, JD
                mov         al, ByteAry[ebx]

                mov         ebx, KD
                mov         ax, WordAry[ebx*2]

                mov         ebx, LD
                mov         eax, DwordAry[ebx*4]

                mov         ebx, MD
                lea         bx, RealAry[ebx*8]


Quit:           mov         ah, 4ch             ;Magic number for DOS
                int         21h                 ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment     para stack 'stack'
stk             byte        1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

### 5.9.4 Sample Program #4: Multidimensional Array Declaration and Access

The following sample program demonstrates how to declare, initialize, and access elements of multidimensional arrays.

```
; Multidimensional Array declaration and access
;
; Randall Hyde


                .386                            ;Need these two statements to use
                option      segment:use16       ; 80386 register set.


dseg            segment     para public 'data'


; Indices we will use for the arrays.

J               word        1
K               word        2
L               word        3

; Some two-dimensional arrays.
; Note how this code uses the "dup" operator to suggest the size
; of each dimension.

B2Ary           byte        3 dup (4 dup (?))
```

```
W2Ary           word        4 dup (3 dup (?))
D2Ary           dword       2 dup (6 dup (?))



; 2D arrays with initialization.
; Note the use of data layout to suggest the sizes of each array.

B2Ary2          byte        0, 1, 2, 3
                byte        4, 5, 6, 7
                byte        8, 9, 10, 11

W2Ary2          word        0,  1,  2
                word        3,  4,  5
                word        6,  7,  8
                word        9, 10, 11

D2Ary2          dword       0,  1,  2,  3,  4,  5
                dword       6,  7,  8,  9, 10, 11


; A sample three dimensional array.

W3Ary           word        2 dup (3 dup (4 dup (?)))

dseg            ends



cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg            ;These statements are
provided by
                mov         ds, ax              ; shell.asm to initialize
the
                mov         es, ax              ; segment register.


; AL := B2Ary2[j,k]

                mov         bx, J               ;index := (j*4+k)
                add         bx, bx              ;j*2
                add         bx, bx              ;j*4
                add         bx, K               ;j*4+k
                mov         al, B2Ary2[bx]


; AX := W2Ary2[j,k]

                mov         ax, J               ;index := (j*3 + k)*2
                mov         bx, 3
                mul         bx                  ;(j*3)-- This destroys
DX!
                add         ax, k               ;(j*3+k)
                add         ax, ax              ;(j*3+k)*2
                mov         bx, ax
                mov         ax, W2Ary2[bx]


; EAX := D2Ary[i,j]

                mov         ax, J               ;index := (j*6 + k)*4
```

```
                mov     bx, 6
                mul     bx                  ;DX:AX := j*6, ignore overflow in DX.
                add     ax, k               ;j*6 + k
                add     ax, ax              ;(j*6 + k)*2
                add     ax, ax              ;(j*6 + k)*4
                mov     bx, ax
                mov     eax, D2Ary[bx]


; Sample access of a three dimensional array.
;
; AX := W3Ary[J,K,L]

                mov     ax, J               ;index := ((j*3 + k)*4 + l)*2
                mov     bx, 3
                mul     bx                  ;j*3
                add     ax, K               ;j*3 + k
                add     ax, ax              ;(j*3 + k)*2
                add     ax, ax              ;(j*3 + k)*4
                add     ax, l               ;(j*3 + k)*4 + l
                add     ax, ax              ;((j*3 + k)*4 + l)*2
                mov     bx, ax
                mov     ax, W3Ary[bx]


Quit:           mov     ah, 4ch             ;Magic number for DOS
                int     21h                 ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

### 5.9.5 Sample Program #5: Structures

This sample program demonstrates how to declare structure types and variables. It also shows how to initialize the fields of a structure at assembly time. Finally, it demonstrates how to access fields of a structure from within an assembly language program and how to deal with pointers to structures.

```
; Sample Structure Definitions and Accesses.
;
; Randall Hyde


dseg            segment para public 'data'


; The following structure holds the bit values for an 80x86 mod-reg-r/m byte.

mode            struct
modbits         byte    ?
reg             byte    ?
```

```
rm              byte        ?
mode            ends


Instr1Adrs      mode        {}                      ;All fields 
uninitialized.
Instr2Adrs      mode        {}


; Some structures with initialized fields.

axbx            mode        {11b, 000b, 000b}       ;"ax, ax" adrs 
mode.
axdisp          mode        {00b, 000b, 110b}       ;"ax, disp" adrs 
mode.
cxdispbxsi      mode        {01b, 001b, 000b}       ;"cx, 
disp8[bx][si]" mode.


; Near pointers to some structures:

sPtr1           word        axdisp
sPtr2           word        Instr2Adrs

dseg            ends


cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg        ;These statements are 
provided by
                mov         ds, ax          ; shell.asm to initialize 
the
                mov         es, ax          ; segment register.


; To access fields of a structure variable directly, just use the "."
; operator like you would in Pascal or C:

                mov         al, axbx.modbits
                mov         Instr1Adrs.modbits, al

                mov         al, axbx.reg
                mov         Instr1Adrs.reg, al

                mov         al, axbx.rm
                mov         Instr1Adrs.rm, al


; When accessing elements of a structure indirectly (that is, using a
; pointer) you must specify the structure type name as the first
; "field" so MASM doesn't get confused:

                mov         si, sPtr1
                mov         di, sPtr2

                mov         al, ds:[si].mode.modbits
                mov         ds:[di].mode.modbits, al

                mov         al, ds:[si].mode.reg
                mov         ds:[di].mode.reg, al
```

```
                mov     al, ds:[si].mode.rm
                mov     ds:[di].mode.rm, al


Quit:           mov     ah, 4ch             ;Magic number for DOS
                int     21h                 ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

### 5.9.6 Sample Program #6: Arrays of Structures

This short program shows you how to declare an array of structures and access elements of that array. It provides examples for one, two, and three dimensional arrays of structures.

```
; Arrays of Structures
;
; Randall Hyde


dseg            segment para public 'data'


; A structure that defines an (x,y) coordinate.
; Note that the Point data type requires four bytes.

Point           struct
X               word    ?
Y               word    ?
Point           ends



; An uninitialized point:

Pt1             Point   {}

; An initialized point:

Pt2             Point   {12,45}


; A one-dimensional array of uninitialized points:

PtAry1          Point   16 dup ({})         ;Note the "{}" inside the parens.


; A one-dimensional array of points, all initialized to the origin.

PtAry1i         Point   16 dup ({0,0})
```

```
; A two-dimensional array of points:

PtAry2          Point      4 dup (4 dup ({}))


; A three-dimensional array of points, all initialized to the origin.

PtAry3          Point      2 dup (3 dup (4 dup ({0,0})))



; A one-dimensional array of points, all initialized to different
values:

iPtAry          Point      {0,0}, {1,2}, {3,4}, {5,6}


; Some indices for the arrays:

J               word       1
K               word       2
L               word       3

dseg            ends





; The following program demonstrates how to access each of the above
; variables.

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

Main            proc
                mov        ax, dseg          ;These statements are
provided by
                mov        ds, ax            ; shell.asm to initialize
the
                mov        es, ax            ; segment register.

; PtAry1[J] := iPtAry[J]

                mov        bx, J             ;Index := J*4 since there
are four
                add        bx, bx            ; bytes per array element
(each
                add        bx, bx            ; element contains two
words).

                mov        ax, iPtAry[bx].X
                mov        PtAry1[bx].X, ax

                mov        ax, iPtAry[bx].Y
                mov        PtAry1[bx].Y, ax

; CX := PtAry2[K,L].X;  DX := PtAry2[K,L].Y

                mov        bx, K             ;Index := (K*4 + J)*4
                add        bx, bx            ;K*2
                add        bx, bx            ;K*4
```

```
                add     bx, J           ;K*4 + J
                add     bx, bx          ;(K*4 + J)*2
                add     bx, bx          ;(K*4 + J)*4

                mov     cx, PtAry2[bx].X
                mov     dx, PtAry2[bx].Y

; PtAry3[j,k,l].X := 0

                mov     ax, j           ;Index := ((j*3 +k)*4 + l)*4
                mov     bx, 3
                mul     bx              ;j*3
                add     ax, k           ;j*3 + k
                add     ax, ax          ;(j*3 + k)*2
                add     ax, ax          ;(j*3 + k)*4
                add     ax, l           ;(j*3 + k)*4 + l
                add     ax, ax          ;((j*3 + k)*4 + l)*2
                add     ax, ax          ;((j*3 + k)*4 + l)*4
                mov     bx, ax
                mov     PtAry3[bx].X, 0

Quit:           mov     ah, 4ch         ;Magic number for DOS
                int     21h             ; to tell this program to quit.
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

## 5.9.7 Sample Program #7: Structures and Arrays

This sample program demonstrates how to declare arrays of structures and how to include arrays and structures as fields within a structure. The 80x86 program code also demonstrates how to access the fields and elements of these data types.

```
; Structures Containing Structures as fields
; Structures Containing Arrays as fields
;
; Randall Hyde


dseg            segment para public 'data'

Point           struct
X               word    ?
Y               word    ?
Point           ends

; We can define a rectangle with only two points.
; The color field contains an eight-bit color value.
; Note: the size of a Rect is 9 bytes.

Rect            struct
```

```
UpperLeft       Point       {}
LowerRight      Point       {}
Color           byte        ?
Rect            ends

; Pentagons have five points, so use an array of points to
; define the pentagon.  Of course, we also need the color
; field.
; Note: the size of a pentagon is 11 bytes.

Pent            struct
Color           byte        ?
Pts             Point       5 dup ({})
Pent            ends


; Okay, here are some variable declarations:

Rect1           Rect        {}
Rect2           Rect        {{0,0}, {1,1}, 1}

Pentagon1       Pent        {}
Pentagons       ent         {}, {}, {}, {}

Index           word        2

dseg            ends


cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg           ;These statements are
provided by
                mov         ds, ax             ; shell.asm to initialize
the
                mov         es, ax             ; segment register.

; Rect1.UpperLeft.X := Rect2.UpperLeft.X

                mov         ax, Rect2.Upperleft.X
                mov         Rect1.Upperleft.X, ax

; Pentagon1 := Pentagons[Index]

                mov         ax, Index          ;Need Index*11
                mov         bx, 11
                mul         bx
                mov         bx, ax

; Copy the first point:

                mov         ax, Pentagons[bx].Pts[0].X
                mov         Pentagon1.Pts[0].X, ax

                mov         ax, Pentagons[bx].Pts[0].Y
                mov         Pentagon1.Pts[0].Y, ax

; Copy the second point:

                mov         ax, Pentagons[bx].Pts[2].X
                mov         Pentagon1.Pts[2].X, ax
```

```
                mov       ax, Pentagons[bx].Pts[2].Y
                mov       Pentagon1.Pts[2].Y, ax

; Copy the third point:

                mov       ax, Pentagons[bx].Pts[4].X
                mov       Pentagon1.Pts[4].X, ax

                mov       ax, Pentagons[bx].Pts[4].Y
                mov       Pentagon1.Pts[4].Y, ax

; Copy the fourth point:

                mov       ax, Pentagons[bx].Pts[6].X
                mov       Pentagon1.Pts[6].X, ax

                mov       ax, Pentagons[bx].Pts[6].Y
                mov       Pentagon1.Pts[6].Y, ax

; Copy the fifth point:

                mov       ax, Pentagons[bx].Pts[8].X
                mov       Pentagon1.Pts[8].X, ax

                mov       ax, Pentagons[bx].Pts[8].Y
                mov       Pentagon1.Pts[8].Y, ax

; Copy the Color:

                mov       al, Pentagons[bx].Color
                mov       Pentagon1.Color, al


Quit:           mov       ah, 4ch            ;Magic number for DOS
                int       21h                ; to tell this program to quit.
Main            endp
cseg            ends

sseg            segment   para stack 'stack'
stk             byte      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end       Main
```

## 5.9.8 Sample Program #8:Pointer to Structures

This sample program demonstrates how to work with pointers to structures and pointers to arrays of structures.

```
; Pointers to structures
; Pointers to arrays of structures
;
; Randall Hyde


                .386                         ;Need these two statements so we can
                option    segment:use16      ; use 80386 register set.
```

```
dseg            segment    para public 'data'

; Sample structure.
; Note: size is seven bytes.

Sample          struct
b               byte       ?
w               word       ?
d               dword      ?
Sample          ends


; Some variable declarations:

OneSampleSample{}
SampleArySample16 dup ({})

; Pointers to the above

OnePtr          word       OneSample         ;A near pointer.
AryPtr          dword      SampleAry


; Index into the array:

Index           word       8

dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

Main            proc
                mov        ax, dseg          ;These statements are
provided by
                mov        ds, ax            ; shell.asm to initialize
the
                mov        es, ax            ; segment register.

; AryPtr^[Index] := OnePtr^


                mov        si, OnePtr        ;Get pointer to OneSample
                les        bx, AryPtr        ;Get pointer to array of
samples
                mov        ax, Index         ;Need index*7
                mov        di, 7
                mul        di
                mov        di, ax

                mov        al, ds:[si].Sample.b
                mov        es:[bx][di].Sample.b, al

                mov        ax, ds:[si].Sample.w
                mov        es:[bx][di].Sample.w, ax
```

```
                mov     eax, ds:[si].Sample.d
                mov     es:[bx][di].Sample.d, eax


Quit:           mov     ah, 4ch          ;Magic number for DOS
                int     21h              ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

## 5.10 Programming Projects

- Program #1: Create a program with a single dimension array of structures. Place at least four fields (your choice) in the structure. Write a code segment to access element "i" ("i" being a word variable) in the array.
- Program #2: Write a program which copies the data from a 3x3 array and stores the data into a second 3x3 array. For the first 3x3 array, store the data in row major order. For the second 3x3 array, store the data in column major order. Use nine sequences of instructions which fetch the word at location (i,j) (i=0..2, j=0..2).
- Program #3: Rewrite the code sequence above just using MOV instructions. Read and write the array locations directly, do not perform the array address computations.
- Program #4: The PC's video display is a *memory mapped I/O device.* That is, the display adapter maps each character on the text display to a word in memory. The display is an 80x25 array of words declared as follows:

```
display:array[0..24,0..79] of word;
```

  Display[0,0] corresponds to the upper left hand corner of the screen, display[0,79] is the upper right hand corner, display[24,0] is the lower left hand corner, and display[24,79] is the lower right hand corner of the display.

  The L.O. byte of each word holds the ASCII code of the character to appear on the screen. The H.O. byte of each word contains the *attribute* byte (see "The PC Video Display" on page 1069 for more details on the attribute byte). The base address of this array is B000:0 for monochrome displays and B800:0 for color displays.

  The diskette accompanying this lab manual contains a sample program named "PROJ4_4.ASM" that is supposed to clear the screen. It contains a main program that uses several instructions you probably haven't seen yet. These instructions essentially execute a for loop as follows:

```
for i:= 0 to 79 do
        for j := 0 to 24 do
                putscreen(i,j,value);
```

  Inside this program you will find some comments that instruct you to supply the code that stores the value in AX to location display[i,j]. Modify this program as described in its comments and test the result.
- Program #5: Proj5_4.asm on the diskette accompanying this lab manual is a maze generation program. It is complete except for two routines that access the MAZE array (maze:array[0..26, 0..81] of word;) and the screen array (screen:array[0..24, 0..79] of word;). You need to supply the code in the two procedures MazeAdrs and ScrnAdrs to compute the indices into these arrays. On entry to these two routines, dl contains the y coordinate (first index) and dh contains the second coordinate (second index). You code must perform the necessary array index computation and leave the final index value in the AX register. See the comments in the code for further details. Note: this program will only run properly on a color display.

## 5.11 Answers to Selected Exercises

3) Use the BYTE directive.
Examples:

```
ByteVar1    BYTE  ?
CharVar     BYTE  ?
Boolean     BYTE  ?
```

Byte variables are useful for declaring small integer variables, boolean variables, character variables, and string variables.

8) A near pointer is 16 bits long and can only point at data within a specific segment. A far pointer is 32 bits long and can point at any location in memory.

11) The following code examples present one possible solution to these problems
a)

```
mov   ax, i        ;i*4
mov   bx, 4
mul   bx
add   ax, j        ;i*4 + j
mov   bx, ax
mov   al, TD[bx]         ;Fetch TD[i,j]
```

12) The following answers provide only one possible solution for each question, of many, to the questions.
a)

```
mov   ebx, 0             ;Initialize H.O. word to zero
mov   ax, i        ;i*4
mov   bx, 4
mul   bx
add   ax, j        ;i*4 + j
mov   bx, ax
mov   al, TD[ebx*1]      ;Fetch TD[i,j]
```

15)

```
Array word  0, 1, 2
      word  3, 4, 5
      word  6, 7, 8
```