NEUB CSE 322 Lab Manual 1: Introduction to Assembly Language

Advantages of Assembly Language

- Low-level access to the computer
- Higher speed
- Total control over CPU
- (Must know what you are doing in order to make these advantages work)

Disadvantages of assembly language

- Increased risk of bugs
  - subtle mistakes can be more costly
- Reduced portability
  - programs run on only one type of CPU
- Absence of library routines
  - must write your own

Structure of an assembly language program

- Assembly language programs divide roughly into five sections
  - header
  - equates
  - data
  - body
  - closing

The Header

- The header contains various directives which do not produce machine code
- Sample header:

```
%TITLE "Sample Header"
IDEAL
MODEL small
STACK 256
```

Equates

- Constant values are known as *equates*
- Sample equate section:

```
Count EQU 10
Element EQU 5
Size = Count * Element
MyString EQU "Maze of twisty passages"
Size = 0
```

- = is used for numeric values only
- Cannot change value of EQU symbol
- EQUated symbols are not variables
- EQU expressions are evaluated where used; = expressions are evaluated where defined

The Data Segment

- Begins with the DATASEG directive
- Two kinds of variables, *initialized* and *uninitialized*.
- Initialized variables take up space in the program's code file
- Declare uninitialized variables after initialized ones so they do not take up space in the program's code file

Reserving space for variables

- Sample DATASEG

```
DATASEG
numRows    DB   25
numColumns DB   80
videoBase  DW   0800h
```

- DB and DW are common directives (define byte) and (define word)
- The symbols associated with variables are called *labels*
- Strings may be declared using the DB directive:

aTOm DB "ABCDEFGHIJKLM"

The Program Body

- Also known as the *code segment*
- Divided into four columns: labels, mnemonics, operands, and comments
- Labels refer to the positions of variables and instructions, represented by the *mnemonics*
- Operands are required by most assembly language instructions
- Comments aid in remembering the purpose of various instructions

An example

```
Label   Mnemonic Operand     Comment
-------------------------------------------------------
       DATASEG
exCode  DB      0           ;A byte variable
myWord  DW      ?           ;Uninitialized word var.
       CODESEG
Start:  mov     ax,@data     ;Initialize DS to address
       mov     ds, ax       ; of data segment
       jmp     Exit         ;Jump to Exit label
       mov     cx, 10       ;This line skipped!
Exit:   mov     ah, 04Ch     ;DOS function: Exit prog
       mov     al, [exCode]  ;Return exit code value
       int     21h          ;Call DOS. Terminate prog
       END     Start        ;End program and specify
                          ; entry point
```

The Label Field

- Labels mark places in a program which other instructions and directives reference
- Labels in the code segment always end with a colon
- Labels in the data segment never end with a colon
- Labels can be from 1 to 31 characters long and may consist of letters, digits, and the special characters ? . @ _ $ %
- If a period is used, it must be the first character
- Labels must not begin with a digit
- The assembler is *case insensitive*

Legal and Illegal Labels

- Examples of legal names
    - COUNTER1
    - @character
    - SUM_OF_DIGITS
    - $1000
    - DONE?
    - .TEST
- Examples of illegal names
    - TWO WORDS contains a blank
    - 2abc begins with a digit
    - A45.28 . not first character
    - YOU&ME contains an illegal character

The Mnemonic Field

- For an instruction, the operation field contains a symbolic operation code (*opcode*)
- The assembler translates a symbolic opcode into a machine language opcode
- Examples are: ADD, MOV, SUB
- In an assembler directive, the operation field contains a directive (*pseudo-op*)
- Pseudo-ops are not translated into machine code; they tell the assembler to do something

The Operand Field

- For an instruction, the operand field specifies the data that are to be acted on by the instruction. May have zero, one, or two operands

```
NOP        ;no operands -- does nothing
INC AX     ;one operand -- adds 1 to the contents of AX
ADD WORD1,2 ;two operands -- adds 2 to the contents
           ; of memory word WORD1
```

- In a two-operand instruction, the first operand is the *destination operand*. The second operand is the *source operand*.
- For an assembler directive, the operand field usually contains more information about the directive.

The Comment Field

- A semicolon marks the beginning of a comment field
- The assembler ignores anything typed after the semicolon on that line
- It is almost impossible to understand an assembly language program without good comments
- Good programming practice dictates a comment on almost every line

The Closing

- The last line of an assembly language program is the closing
- Indicates to assembler that it has reached the end of the program

```
END Start ;End of program / entry point
```

- END is a pseudo-op; the single "operand" is the label specifying the beginning of execution, usually the first instruction after the CODESEG pseudo-op

Data Transfer Instructions

- MOV *destination,source*
    - reg, reg
    - mem, reg
    - reg, mem
    - mem, immed
    - reg, immed
- Sizes of both operands must be the same
- reg can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

Examples

- **mov ax, word1**
    - "Move **word1** to **ax**"
    - Contents of register **ax** are replaced by the contents of the memory location **word1**
- **xchg ah, bl**
    - Swaps the contents of **ah** and **bl**
- **Illegal: mov word1, word2**
    - can't have both operands be memory locations

Sample MOV Instructions

```
b   db  4Fh
w   dw  2048
        mov bl,dh
        mov ax,w
        mov ch,b
        mov al,255
        mov w,-100
        mov b,0
```

- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)
- You can assign a size attribute using LABEL

```
        LoByte LABEL BYTE
        aWord  DW    97F2h
```

Addresses with Displacements

```
b   db  4Fh, 20h, 3Ch
w   dw  2048, -100, 0
        mov bx, w+2
        mov b+1, ah
        mov ah, b+5
        mov dx, w-3
```

- Type checking is still in effect
- The assembler computes an address based on the expression
- *NOTE: These are address computations done at assembly time*
  MOV ax,b-1
  will not subtract 1 from the value stored at b

eXCHanGe

- XCHG *destination,source*
  - reg, reg
  - reg, mem
  - mem, reg
- MOV and XCHG cannot perform memory to memory moves
- This provides an efficient means to swap the operands
  - No temporary storage is needed
  - Sorting often requires this type of operation
  - This works only with the general registers

Arithmetic Instructions

ADD *dest, source*
SUB *dest, source*
INC *dest*
DEC *dest*
NEG *dest*

- *Operands must be of the same size*
- *source* can be a general register, memory location, or constant
- *dest* can be a register or memory location
  - except operands cannot both be memory

ADD and INC

- ADD is used to add the contents of
  - two registers
  - a register and a memory location
  - a register and a constant

- INC is used to add 1 to the contents of a register or memory location

Examples

- **add ax, word1**
  - "Add **word1** to **ax**"
  - Contents of register **ax** and memory location **word1** are added, and the sum is stored in **ax**
- **inc ah**
  - Adds one to the contents of **ah**
- **Illegal: add word1, word2**
  - can't have both operands be memory locations

SUB, DEC, and NEG

- SUB is used to subtract the contents of
  - one register from another register
  - a register from a memory location, or vice versa
  - a constant from a register
- DEC is used to subtract 1 from the contents of a register or memory location
- NEG is used to negate the contents of a register or memory location

Examples

- **sub ax, word1**
  - "Subtract **word1** from **ax**"
  - Contents of memory location **word1** is subtracted from the contents of register **ax**, and the sum is stored in **ax**
- **dec bx**
  - Subtracts one from the contents of **bx**
- **Illegal: sub byte1, byte2**
  - can't have both operands be memory locations

Type Agreement of Operands

- The operands of two-operand instructions must be of the same type (byte or word)
  - mov ax, bh   ;illegal
  - mov ax, byte1 ;illegal
  - mov ah,'A'   ;legal -- moves 41h into ah
  - mov ax,'A'   ;legal -- moves 0041h into ax

Translation of HLL Instructions

- B = A

  mov ax,A
  mov B,ax
  - memory-memory moves are illegal
- A = B - 2*A

  mov ax,B
  sub ax,A
  sub ax,A
  mov A,ax

Program Segment Structure

- Data Segments
  - Storage for variables
  - Variable addresses are computed as offsets from start of this segment
- Code Segment
  - contains executable instructions
- Stack Segment
  - used to set aside storage for the stack
  - Stack addresses are computed as offsets into this segment

- Segment directives

  .DATA
  .CODE
  .STACK *size*

Memory Models

- .Model *memory_model*
  - tiny: code+data <= 64K (.com program)
  - small: code<=64K, data<=64K, one of each
  - medium: data<=64K, one data segment
  - compact: code<=64K, one code segment
  - large: multiple code and data segments
  - huge: allows individual arrays to exceed 64K
  - flat: no segments, 32-bit addresses, protected mode only (80386 and higher)

Program Skeleton

```
.MODEL small
.STACK 100h
.DATA
        ;declarations
.CODE
MAIN PROC
        ;main proc code
        ;return to DOS
ENDP MAIN
;other procs (if any) go here
end MAIN
```

- Select a memory model
- Define the stack size
- Declare variables
- Write code
  - organize into procedures
- Mark the end of the source file
  - define the entry point

Input and Output Using 8086 Assembly Language

- Most input and output is not done directly via the I/O ports, because
  - port addresses vary among computer models
  - it's much easier to program I/O with the service routines provided by the manufacturer
- There are BIOS routines (which we'll look at later) and DOS routines for handling I/O (using interrupt number 21h)

Interrupts

- The interrupt instruction is used to cause a software interrupt (system call)
  - An interrupt interrupts the current program and executes a subroutine, eventually returning control to the original program
  - Interrupts may be caused by hardware or software
- **int *interrupt_number* ;software interrupt**

Output to Monitor

- DOS Interrupts : interrupt 21h
  - This interrupt invokes one of many support routines provided by DOS
  - The DOS function is selected via AH
  - Other registers may serve as arguments

- AH = 2, DL = ASCII of character to output
    - Character is displayed at the current cursor position, the cursor is advanced, AL = DL

Output a String

- Interrupt 21h, function 09h
    - DX = offset to the string (in data segment)
    - The string is terminated with the '$' character
- To place the address of a variable in DX, use one of the following
    - **lea DX,theString** ***;load effective address***
    - **mov DX, offset theString** ***;immediate data***

Print String Example

```
%TITLE "First Program -- HELLO.ASM"
     .8086
     .MODEL   small
     .STACK   256
     .DATA
msg    DB     "Hello, World!$"
     .CODE
MAIN   PROC
     mov    ax,@data     ;Initialize DS to address
     mov    ds,ax       ; of data segment
     lea    dx,msg      ;get message
     mov    ah,09h      ;display string function
     int    21h         ;display message
Exit:  mov    ah,4Ch      ;DOS function: Exit program
     mov    al,0        ;Return exit code value
     int    21h         ;Call DOS. Terminate program
MAIN   ENDP              ;End of program
     END    MAIN        ; entry point
```

Input a Character

- Interrupt 21h, function 01h
- Filtered input with echo
    - This function returns the next character in the keyboard buffer (waiting if necessary)
    - The character is echoed to the screen
    - AL will contain the ASCII code of the non-control character
        - AL=0 if a control character was entered

Sample Assembly Language Program

```
%TITLE "SAMPLE PROGRAM"
.MODEL SMALL  ←——————————————————————  Directives
.STACK 100H
.DATA
A     DW    2
B     DW    5
SUM   DW    ?
.CODE
MAIN PROC
;initialize DS
     MOV   AX,@DATA
     MOV   DS,AX  ←——————————————————————  Instruction
;add the numbers
```

```
          MOV    AX,A            ;AX has A
          ADD    AX,B            ;AX has A+B
          MOV    SUM,AX          ;SUM = A+B
     ;exit to DOS
          MOV    AX,4C00h
          INT    21h
     MAIN ENDP
          END    MAIN
```

//Additional contents for better understanding

SYNTAX OF 8086/8088 ASSEMBLY LANGUAGE

- The language is not case sensitive.
- There may be only one statement per line. A statement may start in any column.
- A statement is either an instruction, which the assembler translates into machine code, or an assembler directive (pseudo-op), which instructs the assembler to perform some specific task.

- Syntax of a statement:

  {name} mnemonic {operand(s)} {; comment}

  (a) The curly brackets indicate those items that are not present or are optional in some statements.
  (b) The name field is used for instruction labels, procedure names, segment names, macro names, names of variables, and names of constants.
  (c) MASM 6.1 accepts identifier names up to 247 characters long. All characters are significant, whereas under MASM 5.1, names are significant to 31 characters only. Names may consist of letters, digits, and the following 6 special characters: **? . @ _ $ %** .If a period is used; it must be the first character. Names may not begin with a digit.
  (d) Instruction mnemonics, directive mnemonics, register names, operator names and other words are reserved.

- Syntax of an instruction:
  {label:} mnemonic {operand { , operand} } {; comment}

  The curly brackets indicate those items that are not present or are optional in some instructions. Thus an instruction may have zero, one, or two operands.

- Operators:
  The 8086/8088 Assembly language has a number of operators. An operator acts on an operand or operands to produce a value at assembly time. Examples are: + , - , *, / , DUP, and OFFSET

- Comments:
  A semicolon starts a comment. A comment may follow a statement or it may be on a separate line. Multiple-line comments can be written by using the COMMENT directive. The syntax is:

  COMMENT **delimiter** {comment}
  comment
  . . .
  delimiter { comment }

where delimiter is any non-blank character not appearing in comment. The curly brackets indicate an item that is optional.

e.g.,

```
COMMENT  *
        This program finds
        the maximum element in a byte array
*
```

- Numbers:
  (a) A binary number is suffixed by b or B.
  e.g., 11010111B
  (b) A decimal number is suffixed by an optional d or D.
  e.g., 42d -22D 3578
  (c) A hexadecimal number must begin with a decimal digit and it is suffixed by h or H
  e.g., 20H 0bF2Ah

- Characters:
  A character is enclosed in a pair of single quotes or in a pair of double quotes.
  e.g., 'x' "B"

- Strings:
  A string is enclosed in a pair of single quotes or in a pair of double quotes.
  e.g., 'ENTER YOUR NAME: '
  "THE MAXIMUM VALUE IS "
  'Omar shouted, "help !" '
  "say, 'hello' "
  "Omar's books"
  For a string delimited by single quotes, a pair of consecutive single quotes stands for a single quote.
  e.g., 'Omar' 's books'

- Data definition
  Each variable has a data type and is assigned a memory address by the program. The data-defining directives are:

Directive	Description of Initializers
BYTE, **DB** (byte)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed byte)	Allocates signed numbers from –128 to +127.
WORD, DW (word = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed word)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doubleword = 4 bytes),	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doubleword)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.

e.g.,

```
ALPHA   DB  4
VAR1    DB  ?
ARRAY1  DB  40H, 35H, 60H, 30H
VAR2    DW  3AB4h
ARRAY2  DW  500, 456, 700, 400, 600
PROMPT  DB  'ENTER YOUR  NAME $'
```

```
        POINTER1  DD   6BA7000AH
```

A **?** in place of an initializer indicates you do not require the assembler to initialize the variable. The assembler allocates the space but does not write in it. Use **?** for buffer areas or variables your program will initialize at run time.

```
integer        BYTE    16
negint          SBYTE   -16
expression    WORD    4*3
signedexp    SWORD   4*3
empty          QWORD   ?         ; Allocate uninitialized long int
                  BYTE    1,2,3,4,5,6 ; Initialize six unnamed bytes
long            DWORD   4294967295
 longnum        SDWORD  -2147433648
```

The DUP operator can be used to generate multiple bytes or words with known as well as un-initialized values.

```
e.g.,       table  dw    100  DUP(0)
            stars  db     50  dup('*')
            ARRAY3    DB   30   DUP(?)
            ARRAY4    DB   10   DUP(50), 45, 22, 20  DUP(60)
            STRINGS   DB    20H  DUP('Dhahran')
```

Note: If a variable name is missing in a data definition statement, memory is allocated; but no name is associated with that memory. For example:

```
            DB  50  DUP(?)
```

allocates 50 un-initialized bytes; but no name is associated with those 50 bytes.

In MASM 6.1 and obove, a comma at the end of a data definition line (except in the comment field) implies that the line continues. For example, the following code is legal in MASM 6.1:

```
longstring     BYTE    "This string ",
               "continues over two lines."
bitmasks       BYTE    80h, 40h, 20h, 10h,
               08h, 04h, 02h, 01h
```

- Named constants:
  The EQU (equate) directive, whose syntax is:
  ```
      name   EQU   constant_expression
  ```
  assigns a name to a constant expression. Example:
  ```
      MAX   EQU   32767
      MIN    EQU   MAX  -  10
      LF       EQU   0AH
      PROMPT   EQU   'TYPE YOUR NAME: $'
  ```
  Note: (i) No memory is allocated for EQU names
  (ii) A name defined by EQU may not be redefined later in a program.
- The LABEL directive, whose syntax is:
  ```
      name   LABEL   type
  ```
  where type (for MASM Version 5.1 and lower versions) is BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, or FAR provides a way to define or redefine the type associated with a variable or a label.
  Example1:
  ```
            ARRAY1   LABEL    WORD
            ARRAY2   DB   100   DUP(0)
  ```

Here ARRAY1 defines a 50-word array, and ARRAY2 defines a 100-byte array. The same memory locations are assigned to both arrays. Thus the array can be accessed as either the byte array ARRAY1 or the word array ARRAY2.

Example2:

```
VAR3  LABEL  DWORD
WORD1   LABEL   WORD
BYTE1  DB  ?
BYTE2  DB  ?
WORD2   LABEL   WORD
BYTE3  DB  50H
BYTE4  DB  66H
```

in this example, each of the words, and each of the bytes of the double word variable VAR3 can be accessed individually.

SEGMENT DEFINITION

- An 8086/8088 assembly language program file must have the extension **.asm**
- There are two types of 8086/8088 assembly language programs: exe-format and com-format.
- An exe-format program generates executable files with extension **.exe**. A com-format program generates executable files with extension **.com** .
- An exe-format program must contain a code segment and a stack segment. It may contain a data segment or an extra segment.
- A com-format program contains only the code segment (the stack segment is explicit).
- A programmer chooses an appropriate size for the stack segment, depending on the size of his program. Values in the range 100H to 400H are sufficient for most small programs.

Note: In a program, the data, code, and stack segments may appear in any order. However, to avoid forward references it is better to put the data segment before the code segment.

SIMPLIFIED SEGMENT DIRECTIVES

MASM version 5.0 and above, and TASM provide a simplified set of directives for declaring segments called simplified segment directives. To use these directives, you must initialize a memory model, using the **.**MODEL directive, before declaring any segment. The format of the **.**MODEL directive is:

```
.MODEL   memory-model
```

The memory-model may be TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE or FLAT :

memory-model	description
TINY	One segment. Thus code and data together may not be greater than 64K
SMALL	One code-segment. One data-segment. Thus neither code nor data may be greater than 64K
MEDIUM	More than one code-segment. One data-segment. Thus code may be greater than 64K
COMPACT	One code-segment. More than one data-segment. Thus data may be greater than 64K
LARGE	More than one code-segment. More than one data-segment. No array larger than 64K. Thus both code and data may be greater than 64K
HUGE	More than one code-segment. More than one data-segment. Arrays may be larger than 64K. Thus both code and data may be greater than 64K
FLAT	One segment up to 4GB. All data and code (including system resources) are in a single 32-bit segment.

All of the program models except TINY result in the creation of exe-format programs. The TINY model creates com-format programs.

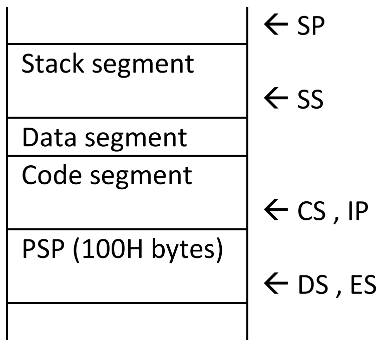
Memory Model	**Operating System**	Data and Code Combined
Tiny	MS-DOS	Yes
Small	MS-DOS, Windows	No
Medium	MS-DOS, Windows	No
Compact	MS-DOS, Windows	No
Large	MS-DOS, Windows	No
Huge	MS-DOS, Windows	No
Flat	Windows NT	Yes

- The simplified segment directives are: **.**CODE , **.**DATA , **.**STACK .
- The **.**CODE directive may be followed by the name of the code segment.
- The **.**STACK directive may be followed by the size of the stack segment, by default the size is 1K i.e., 1,024 bytes.

The definition of a segment extends from a simplified segment directive up to another simplified segment directive or up to the END directive if the defined segment is the last one.

THE GENERAL STRUCTURE OF AN EXE-FORMAT PROGRAM

The memory map of a typical exe-format program, with segments defined in the order code, data, and stack is:



The CS and IP registers are automatically initialized to point to the beginning of the code segment.
The SS register is initialized to point to the beginning of the stack segment.
The SP register is initialized to point one byte beyond the stack segment.
The DS and ES registers are initialized to point to the beginning of the PSP (Program Segment Prefix) segment.
This is a 100H (i.e., 256) byte segment that DOS automatically prefaces to a program when that program is loaded in memory.
The PSP contains important information about the program.
Thus if a program contains a data segment, the DS register must be initialized by the programmer to point to the beginning of that data segment.
Similarly if a program contains an extra segment, the ES register must be initialized by the programmer to point to the beginning of that extra segment.

Initialization of DS

Note: The instructions which initialize the DS register for an exe-format program with simplified segment directives are:

```
MOV  AX ,  @DATA
MOV  DS ,  AX
```

where AX may be replaced by any other 16-bit general purpose register.
At load time, @DATA is replaced with the 16-bit base address of the data segment.

Thus @DATA evaluates to a constant value; such an operand is usually called an immediate operand.

Since MOV instructions of the form:

```
MOV  SegmentRegister ,  ImmediateOperand
```

are invalid, an initialization of the form:

```
MOV  DS ,  @DATA
```

is invalid. Such an initialization is done indirectly using any 16-bit general-purpose register. Example:

```
MOV  AX ,  @DATA
MOV  DS , AX
```

Note: Every 8086 assembly language program must end with the END directive. This directive may be followed by an entry label, which informs the assembler the point where program execution is to begin. The entry label can have any valid name.

The general structure of an exe-format program is:

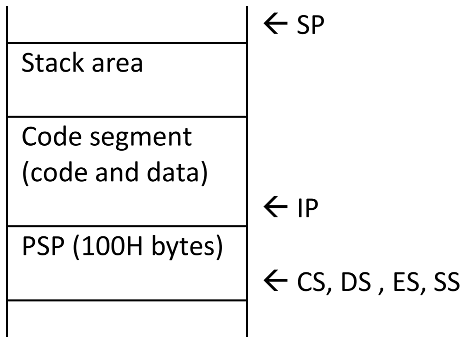
```
.MODEL  SMALL
.STACK  200
.DATA
    ; data definitions using DB, DW, DD, etc. come here
.CODE
START:   MOV  AX , @DATA              ; Initialize DS
          MOV  DS , AX                ;
               . . .
          ; Return to DOS
          MOV  AX , 4C00H
          INT  21H
END   START
```

Example:

```
.MODEL  SMALL
.STACK  200
.DATA
    MESSAGE  DB  'ICS 232' , '$'
.CODE
START:   MOV  AX , @DATA              ; Initialize DS
          MOV  DS , AX                ;
          ; Display the string
          MOV  AH , 09H
          MOV  DX , OFFSET  MESSAGE
          INT  21H
          ; Return to DOS
          MOV  AX , 4C00H
          INT  21H
END   START
```

THE GENERAL STRUCTURE OF A COM-FORMAT PROGRAM

The memory map of a typical com-format program is:



To work out the locations corresponding to symbols (labels and variables) in the source program, the assembler uses a variable called the location counter.
Before assembly of each segment begins the location counter is set to zero. As each statement in that segment is scanned, the location counter is incremented by the number of bytes required by that statement.
Since the CS register is initialized to point to the beginning of the PSP when a com-format program is loaded in memory, the location counter must be set to 100H instead of the usual zero, so that: (i) the assembler assigns offset addresses relative to the beginning of the code segment and not the PSP, and (ii) the IP register is set to 100H when the program is loaded.
The location counter is set to 100H by the directive:

```
        ORG   100H
```

Hence this directive must appear at the beginning of every com-format program before the program entry point.

Since a com-format program contains only one explicit segment i.e., the code segment, data, if any, must be defined within the code segment anywhere a data definition statement will not be treated as an executable statement.

This can be done at the beginning of the code segment by jumping across data definitions using a JMP instruction.

The general structure of a com-format program is:

```
.MODEL TINY
.CODE
        ORG  100H
ENTRY:   JMP  L1
        ; data definitions using DB, DW, DD, etc. come here
        .  .  .
L1:
        .  .  .
        ; Return to DOS
        MOV  AX ,  4C00H
        INT  21H
END  ENTRY
```

Example:

```
.MODEL   TINY
.CODE
          ORG  100H
ENTRY:  JMP  START
    MESSAGE  DB  'ICS 232' , '$'
START:
         ; Display the string
          MOV  AH , 09H
          MOV  DX , OFFSET  MESSAGE
          INT  21H
          ; Return to DOS
          MOV  AX , 4C00H
          INT  21H
END  ENTRY
```

Other Directives

.STARTUP
Generates program start-up code.

The **.EXIT** directive accepts a 1-byte exit code as its optional argument:

```
.EXIT   1           ; Return exit code 1
```

.EXIT generates the following code that returns control to MS-DOS, thus terminating the program. The return value, which can be a constant, memory reference, or 1-byte register, goes into AL:

```
mov    al, value
mov    ah, 04Ch
int    21h
```

If your program does not specify a return value, **.EXIT** returns whatever value happens to be in AL.

.586
Enables assembly of nonprivileged instructions for the Pentium processor.

.686
Enables assembly of nonprivileged instructions for the Pentium Pro processor.

The USE16, USE32, and FLAT Options
When working with an 80386 or later processor, MASM generates different code for 16 versus 32 bit segments. When writing code to execute in real mode under DOS, you must always use 16 bit segments. Thirty-two bit segments are only applicable to programs running in protected mode. Unfortunately, MASM often defaults to 32 bit mode whenever you select an 80386 or later processor using a directive like .386, .486, .586 or .686 in your program. If you want to use 32 bit instructions, you will have to explicitly tell MASM to use 16 bit segments. The use16, use32, and flat operands to the segment directive let you specify the segment size.
For most DOS programs, you will always want to use the use16 operand. This tells MASM that the segment is a 16 bit segment and it assembles the code accordingly. If you use one of the directives to activate the 80386 or later instruction sets, you should put use16 in all your code segments or MASM will generate bad code.
If you want to force use16 as the default in a program that allows 80386 or later instructions, there is one way to accomplish this. Place the following directive in your program before any segments:

```
option segment:use16
```

Example:

```
.586
option  segment:use16
```