CHAPTER 3: ASSEMBLY LANGUAGE FUNDAMENTALS

from: ***Assembly Language for Intel-Based Computers***, by Kip R. Irvine.
(c) 1999 by Prentice-Hall. Inc. Simon and Schuster, a Viacom Company. All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

ISBN 0-13-660390-4

From this point on, we will use the Microsoft Assembler (called *MASM*) and Turbo Assembler (called *TASM*). Although many examples in this chapter could be assembled and tested using Debug, it has many limitations. For example, Debug does not let you create symbolic names, or insert and delete individual source code lines. If you need to review the commands for assembling, linking, and debugging programs, refer to Section 3.3 for the specific commands applying to various assemblers.

3.1 BASIC ELEMENTS OF ASSEMBLY LANGUAGE

In this section, we elaborate on the basic elements of Intel assembly language. Compared to other computer languages, assembly language has a very simple syntax. Assembly language statements are made up of constants, literals, names, mnemonics, operands, and comments.

3.1.1 Constants and Expressions

Numeric Literal. A numeric literal is a combination of digits and other optional parts: a sign, a decimal point, and an exponent. Here are some examples:

```
5
5.5
–5.5
26.E+05
```

Integer constants can end with a radix symbol that identifies the numeric base. The bases are: h = hexadecimal, q (or o) = octal, d = decimal, b = binary. If no radix is specified, decimal is the default. Uppercase/lowercase differences are ignored. Here are examples:

```
26              decimal
1Ah             hexadecimal
1101b           binary
36q             octal
2BH             hexadecimal
42Q             octal
36D             decimal
47d             decimal
0F6h            hexadecimal
```

When a hexadecimal constant begins with a letter, it must contain a leading zero. Although the radix can be uppercase, we recommend that lowercase be used consistently for a more uniform appearance.

A *constant expression* consists of combinations of numeric literals, operators, and defined symbolic constants. The expression value must be able to be determined at assembly time, and its value cannot change at runtime. Here are some examples of expressions involving only numeric literals:

```
5
26.5
4 * 20
–3 * 4 / 6
–2.301E+04
```

A *symbolic constant* is created by assigning a constant expression to a name. For example,

```
rows = 5
columns = 10
tablePos = rows * columns
```

Although these declarations look like runtime statements written in a high-level language, it is important to realize that they can *only* be evaluated at assembly time.

Character or String Constant. A constant may also represent a string of characters enclosed in either single or double quotation marks. Embedded quotes are permitted, as the following examples show:

```
‘ABC’
‘X’
“This is a test”
‘4096’
“This isn’t a test”
‘Say “hello” to Bill.’
```

Notice that the string constant containing “4096” is four bytes long, each containing the ASCII code for a single character.

3.1.2 Statements

An assembly language *statement* consists of a name, an instruction mnemonic, operands, and a comment. Statements generally fall into two classes, instructions and directives. *Instructions* are executable statements, and *directives* are statements that provide information to tell the assembler how to generate executable code. The general format of a statement is:

```
[name] [mnemonic] [operands] [;comment]
```

Statements are free-form, meaning that they can be written in any column with any number of spaces between each operand. Blank lines are permitted between statements. A statement must be written on a single line and cannot pass column 128. You can continue a line onto the next line, if the last character in the first line is \ (*backslash*):

```
longArrayDefinition dw 1000h, 1020h, 1030h  \
        1040h, 1050h, 1060h, 1070h, 1080h
```

An *instruction* is a statement that is executed by the processor at runtime. Instructions fall into general types: transfer of control, data transfer, arithmetic, logical, and input/output. Instructions are translated directly into machine code by the assembler. Here are examples of instructions, shown by category:

```
call  MySub     ; transfer of control
mov   ax,5      ; data transfer
add   ax,20     ; arithmetic
jz    next1     ; logical (jump if Zero flag was set)
in    al,20     ; input/output (reads from hardware port)
```

A *directive* is a statement that affects either the program listing or the way machine code is generated. For example, the **db** directive tells the assembler to create storage for a byte variable named **count** and initialize it to 50:

```
count  db   50
```

The following .**stack** directive tells the assembler to reserve 4096 bytes of stack space:

```
.stack 4096
```

3.1.3 Names

A *name* identifies a label, variable, symbol, or keyword. It may contain any of the following characters:

Character	Description
A . . . Z, a . . . z	Letters
0 . . . 9	Digits
?	Question mark
_	Underscore
@	@ Sign
$	Dollar sign
.	Period

Names have the following restrictions:

- A maximum of 247 characters (in MASM).

- There is no distinction between uppercase and lowercase letters.
- The first character can be a letter, '@', '_', or '$'. Subsequent characters can be the same, or they can also be decimal digits. Avoid using '@' as the first character, because many predefined symbol names start with it.
- A programmer-chosen name cannot be the same as an assembler reserved word.

Variable. A *variable* is a location in a program's data area that has been assigned a name. For example:

```
count1 db 50      ; a variable (memory allocation)
```

Label. If a name appears in the code area of a program, it is called a *label*. Labels serve as place markers when a program needs to jump or loop from one location to another. A label can be on a blank line by itself, or it can share a line with an instruction. In the following example, **Label1** and **Label2** are labels identifying locations in a program:

```
Label1:  mov  ax,0
         mov  bx,0
         .
         .
Label2:
         jmp  Label1     ; jump to Label1
```

Keyword. A *keyword* always has some predefined meaning to the assembler. It can be an instruction, or it can be an assembler directive. Examples are MOV, PROC, TITLE, ADD, AX, and END. Keywords cannot be used out of context or as identifiers. In the following, the use of **add** as a label is a syntax error:

```
add:  mov  ax,10
```

3.2 SAMPLE HELLO PROGRAM

Example 1 shows a program that displays the traditional "Hello, world!" message on the screen. It contains the essential ingredients of an assembly language application. Line 1 contains the **Title** directive; all remaining characters on the line are treated as comments, as are all characters on line 3. The source code for this program was written in assembly language and must be assembled into machine language before it can run. This program is compatible with both the Microsoft and Borland assemblers.

Segments are the building blocks of programs: The *code* segment is where program instructions are stored; the *data* segment contains all variables, and the *stack* segment contains the program's runtime stack. The stack is a special area of memory that the program uses when calling and returning from subroutines.

Example 1. The Hello World Program.

```
title Hello World Program        (hello.asm)

; This program displays "Hello, world!"
.model small
.stack 100h
.data
message db "Hello, world!",0dh,0ah,'$'

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

    mov  ah,9
    mov  dx,offset message
    int  21h

    mov  ax,4C00h
    int  21h
main endp
end main
```

Here is a brief description of the important lines in the program:

- The **.model small** directive indicates that the program uses a type of structure in which the program uses no more than 64K of memory for code, and 64K for data. The **.stack** directive sets aside 100h (256) bytes of stack space for the program. The **.data** directive marks the beginning of the data segment where variables are stored. In the declaration of **message**, the assembler allocates a block of memory to hold the string containing "Hello, world!," along with two bytes containing a *newline* character sequence (0dh,0ah). The **$** is a required string terminator character for the particular MS-DOS output subroutine being used.
- The **.code** directive marks the beginning of the code segment, where the executable instructions are located. The **PROC** directive declares the beginning of a procedure. In this program, we have a procedure called **main**.
- The first two statements in the main procedure copy the address of the data segment (@data) into the DS register. The **mov** instruction always has two operands: first the destination, then the source.

- Next in the main procedure, we write a character string on the screen. This is done by calling an MS-DOS function that displays a string whose address is in the DX register. The function number is placed in the AH register.
- The last two statements in the main procedure (mov ax,4C00h / int 21h) halt the program and return control to the operating system.
- The statement **main endp** uses the ENDP directive. ENDP marks the end of the current procedure. Procedures, by the way, are not allowed to overlap.
- The end of the program contains the **end** directive, which is the last line to be assembled. The label **main** next to it identifies the location of the program entry point, that is, the point at which the CPU starts to execute the program's instructions.

At this point you may be fondly remembering the first "Hello, world" program you wrote in C, C++, Java, or Pascal. How can the assembly language version be so complicated? Actually, if you looked at the machine code generated by a high-level language compiler, you would see a lot of extra code that was automatically added to the program. In contrast, an assembler only inserts machine code for instructions that you have written. It is interesting to compare the executable size of the same program written and compiled in C++ (8772 bytes) to that of the program in Example 1 (562 bytes). Now who's doing more work?

Table 1 contains a list of the most commonly used assembler directives.

Table 1. Standard Assembler Directives.

Directive	Description
end	End of program assembly
endp	End of procedure
page	Set a page format for the listing file
proc	Begin procedure
title	Title of the listing file
.code	Mark the beginning of the code segment
.data	Mark the beginning of the data segment
.model	Specify the program's memory model
.stack	Set the size of the stack segment

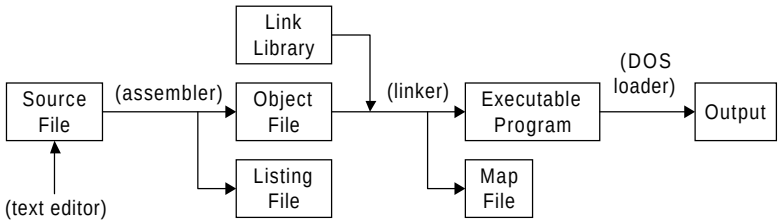
3.3 ASSEMBLING, LINKING, AND DEBUGGING

By now, we have seen that it's fairly easy to assemble and run short programs with Debug. You will soon be using the *assembler,* a utility program that converts a source program into an object file, and a *linker,* a program that converts object files into executable programs.

One major advantage to using an assembler is that source programs can be more easily modified with a text editor than with Debug. Another is that you can use symbolic names for variables, rather than hard-coded numeric addresses. The linker has one primary advantage—programs can take advantage of existing libraries full of useful subroutines; the subroutines are "attached" to our programs by the linker.

A Two-Staged Process. Figure 1 shows the stages a program goes through before it can be executed. A programmer uses a *text editor* to create the source file of ASCII text. The *assembler* program reads the source file and produces an *object file,* a machine-language translation of the program. The object file may contain calls to subroutines in an external *link library*. The linker then copies the needed subroutines from the link library into the object file, creates a special header record at the beginning of the program, and produces an *executable program.* When we are ready to run the program, we type its name on the DOS command line, and the *DOS loader* decodes the header record of the executable program and loads it into memory. The CPU begins executing the program.

Figure 1. The Assemble-Link-Execute Cycle.



The assembler produces an optional *listing file,* which is a copy of the program's source file (suitable for printing) with line numbers and translated machine code. The linker produces an optional *map file,* which contains information about the program's code, data, and stack segments.

A *link library* is a file containing subroutines that are already compiled into machine language. Any procedures called by your program are copied into the executable program during the link step. (If you've ever programmed in Pascal or C, you have made extensive use of link libraries, perhaps without realizing it). Table 2 displays a list of filenames that would be created if we assembled and linked the Hello World program.

Table 2. Files Created by the Assembler and Linker.

Filename	Description	Step When Created
hello.asm	Source program	Edit
hello.obj	Object program	Assembly
hello.lst	Listing file	Assembly
hello.exe	Executable program	Link
hello.map	Map file	Link

3.3.1 Borland Turbo Assembler (TASM)

Once the Hello World program has been created using a text editor and saved to disk as sample.asm, it is ready to be assembled. The command to assemble **hello.asm** with the Borland assembler is:

```
C:\> tasm/l/n/z hello
```

We have shown the *MS-DOS* command prompt here as C:\> for illustrative purposes, but your command prompt may be different. This command can be typed at the command-line prompt in *MS-DOS*, or in an MS-DOS shell running under Windows. The **/l/n** options tell the assembler to produce a listing file, with no symbol table, and **/z** indicates that source lines with errors are to be displayed. The following is the screen output produced by assembling the program with the Borland assembler:

```
Turbo Assembler  Version 4.1
Copyright (c) 1988, 1996 Borland International

Assembling file:   hello.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  418k
```

The primary file produced by the assembly step is **hello.obj**. The assembler distinguishes between *error messages* and *warning messages*. A program with warning messages will still assemble, but the object file may have errors. In general, it is best to fix such errors before linking the program.

Syntax Errors. We were fortunate to have not made any mistakes when writing this program. But if we had, the assembler would have displayed the line with the mistake, along with an explanation. For example, **value1** was incorrectly spelled as **vlue1**:

```
mov   al,vlue1      ; load the AL register
**Error** sample.asm(13) Undefined symbol: VLUE1
```

Linking the Program. In the LINK step, the linker reads the object file, called hello.obj, as input and creates the executable file, called hello.exe. Here is the command:

```
C:\> tlink/3/m/v hello
```

The /3 option allows the use of 32-bit registers, the **/m** option tells the linker to create a *map* file, and the **/v** option includes debugging information in the executable program. Be sure to use these options when assembling and linking programs shown in this book.

Running the Program. You can run an assembly language program from the MS-DOS command prompt by just typing its name:

```
C:\> hello
```

You will probably want use a debugger to test a newly written program. For example, we can run the sample.exe program in Borland's Turbo Debugger with the following command:

```
C:\> td hello
```

3.3.2 Microsoft Assembler (MASM)

The Microsoft Assembler package contains the ML.EXE program, which assembles and links one or more assembly language source files, producing an object file (extension .obj) and an executable file (.exe). The basic syntax is:

```
ML options filename.ASM
```

For example, the following command assembles and links hello.asm (with CodeView debugging information), producing hello.obj and hello.exe:

```
ml/Zi hello.asm
```

The following assembles and links hello.asm, producing a listing file called hello.lst, as well as hello.obj and hello.exe:

```
ml/Fl hello.asm
```

The following assembles, links, and produces a *map* file called hello.map during the link step. It still produces hello.obj and hello.exe:

```
ml/Fm hello.asm
```

The following produces an object file, listing file, map file, executable file, and includes debugging information. The line following it runs the CodeView debugger:

```
ml/Zi/Fm/Fl hello.asm
cv hello
```

Microsoft also supplies the **masm.exe** program to remain compatible with programs written under older versions of the assembler. The MASM commands to assemble and link a program called sample are the following:

```
masm/l/n/z sample;
link/m/co sample;
```

3.4 DATA ALLOCATION DIRECTIVES

A *variable* is a symbolic name for a location in memory where some data are stored. A variable's *offset* is the distance from the beginning of the segment to the variable. A variable's name is automatically associated with its offset. For example, if we declare an array containing four characters, the name **aList** identifies only the offset of the first character (A):

```
.data
aList  db "ABCD"
```

Offset	Contents
0000	'A'
0001	'B'
0002	'C'
0003	'D'

If the first letter is at offset 0, the next one is at offset 1, the next at 2, and so on. The offset of **aList** is equal to 0, the offset of the first letter.

We use *data allocation directives* to allocate storage, based on several following predefined types. (In this chapter we will discuss the DB, DW, and DD directives and leave the others for later chapters):

Mnemonic	Description	Bytes	Attribute
DB	Define byte	1	Byte
DW	Define word	2	Word
DD	Define doubleword	4	Doubleword
DF, DP	Define far pointer	6	Far pointer
DQ	Define quadword	8	Quadword
DT	Define tenbytes	10	Tenbyte

3.4.1 Define Byte (DB)

The DB (*define byte*) directive allocates storage for one or more 8-bit (byte) values. The following syntax diagram shows that *name* is optional and at least one initializer is required. If more are supplied, they must be separated by commas:

```
[name] DB initval [,initval] . . .
```

Each initializer can be a constant expression containing numeric literals, defined symbols, and quoted characters and strings. If the value is signed, it has a range of –128 to +127; if unsigned, the range is 0 to 255. A list of values can be grouped under a single label with the values separated by commas. For example:

```
char1       db 'A'              ; ASCII character
char2       db 'A'–10           ; expression
signed1     db –128             ; smallest signed value
signed2     db +127             ; largest signed value
unsigned1   db 255              ; largest unsigned value
```

A variable's initial contents may be left undefined by using a question mark for the initializer:

```
myval db ?
```

Multiple Initializers. Sometimes the name of a variable identifies the beginning of a sequence of bytes. In that case, multiple initializers can be used in the same declaration. In the following example, assume that **list** is stored at offset 0000. This means that 10 is stored at offset 0000, 20 at offset 0001, 30 at offset 0002, and 40 at offset 0003:

```
list  db  10,20,30,40
```

Characters and integers are one and the same as far as the assembler is concerned. The following variables contain exactly the same value and can be processed the same way:

```
char    db  'A'             ; a character (ASCII 41h)
hex     db  41h             ; hexadecimal
dec     db  65              ; decimal
bin     db  01000001b       ; binary
oct     db  101q            ; octal
```

Each initializer can use a different radix when a list of items is defined, and numeric, character, and string constants can be freely mixed. When a hexadecimal number begins with a letter (A-F), a leading zero is added to prevent the assembler from interpreting it as a label. In this example, **list1** and **list2** have the same contents:

```
list1   db   10,  32,  41h,  00100010b
list2   db   0Ah, 20h, 'A',  22h
```

Representing Strings. A string can be identified by a variable, which marks the offset of the beginning of the string. There is no universal storage format for strings, although null-terminated strings used by the C language are used when calling Microsoft Windows functions. The following shows a null-terminated string called CString, and another called PString that has its length encoded in the first byte:

```
Cstring  db  "Good afternoon",0
Pstring  db  14, "Good afternoon"
```

The DB directive is ideal for allocating strings of any length. The string can continue on multiple lines without the necessity of supplying a label for each line. The following is a null-terminated string:

```
LongString  db "This is a long string, that "
            db "clearly is going to take "
            db "several lines to store",0
```

The assembler can automatically calculate the storage used by any variable by subtracting its starting offset from the next offset following the variable. The **$** operator returns the current location counter, so we can use it an expression such as the following:

```
(offset)
  0000  mystring  db "This is a string"
  0010  mystring_len = ($ – mystring)
```

In this example, **mystring_len** is equal to 10h.

DUP Operator. The DUP operator only appears after a storage allocation directive, such as DB or DW. With DUP, you can repeat one or more values when allocating storage. It is especially useful when allocating space for a string or array. Notice that many of the following examples initialize storage to default values:

```
db 20 dup(0)              ; 20 bytes, all equal to zero
db 20 dup(?)              ; 20 bytes, uninitialized
db  4 dup("ABC")          ; 12 bytes: "ABCABCABCABC"
db 4096 dup(0)            ; 4096–byte buffer, all zeros
```

The DUP operator can also be nested. The first example that follows creates storage containing 000XX000XX000XX000XX. The second example creates a two-dimensional word table of 3 rows and 4 columns:

```
aTable   db  4 dup( 3 dup(0), 2 dup('X') )
aMatrix  dw  3 dup( 4 dup(0) )
```

3.4.2 Define Word (DW)

The DW (*define word*) directive creates storage for one or more 16-bit values. The syntax is:

```
[name] DW initval [,initval] . . .
```

Each initializer is equivalent to an unsigned integer between 0 and 65,535 (FFFFh). If *initval* is signed, the acceptable range is –32,768 (8000h) to +32,767 (7FFFh). A character constant can be stored in the lower half of a word. One can also leave the variable uninitialized by using the **?** operator. Here are some examples:

```
dw  0,65535                 ; smallest/largest unsigned values
dw  -32768,+32767           ; smallest/largest signed values
dw  256 * 2                 ; calculated expression (512)
dw  1000h,4096,'AB',0       ; mutiple initializers
dw  ?                       ; uninitialized
dw  5  dup(1000h)           ; 5 words, each equal to 1000h
dw  5  dup(?)               ; 5 words, uninitialized
```

Pointer. The offset of a variable or subroutine can be stored in another variable, called a *pointer*. In the next example, the assembler initializes **P** to the offset of **list**:

```
list dw 256,257,258,259     ; define 4 words
P    dw list                ; P points to list
```

Reversed Storage Format. The assembler reverses the bytes in a word value when storing it in memory; the lowest byte occurs at the lowest address. When the variable is moved to a 16-bit register, the CPU re-reverses the bytes. For example, the value 1234h would be stored in memory as follows:

```
Offset:  00  01
 Value:  34  12
```

3.4.3 Define Doubleword (DD)

The DD (*define doubleword*) directive allocates storage for one or more 32-bit doublewords. The syntax is

```
[name] DD initval [,initval] . . .
```

Each initializer is equivalent to an integer between 0 and 0FFFFFFFFh. For example:

```
signed_val  dd  0, 0BCDA1234h, –2147483648
            dd  100h dup(?)     ; 256 doublewords (1024 bytes)
```

The bytes in a doubleword are stored in reverse order, so the least significant digits are stored at the lowest offset. For instance, the value 12345678h would be stored as:

```
Offset:  00  01  02  03
 Value:  78  56  34  12
```

A doubleword can hold the 32-bit segment-offset address of a variable or procedure. In the following example, the assembler automatically initializes **pointer1** to the address of **subroutine1**:

```
pointer1  dd  subroutine1
```

3.5 SYMBOLIC CONSTANTS

Equate directives allow constants and literals to be given symbolic names. A constant can be defined at the start of a program and, in some cases, redefined later on.

3.5.1 Equal-Sign Directive

Known as a *redefinable equate,* the equal-sign directive creates an absolute symbol by assigning the value of a numeric expression to a name. The syntax is:

```
name = expression
```

In contrast to the DB and DW directives, the equal-sign directive allocates no storage. As the program is assembled, all occurrences of *name* are replaced by *expression*. The expression must be able to be expressed by a 32-bit signed or unsigned integer (32-bit integers require that you use the .386 or higher directive). Examples are as follows:

```
prod      = 10 * 5              ; Evaluates an expression
maxInt   = 7FFFh                ; Maximum 16-bit signed value
minInt   = 8000h                ; Minimum 16-bit signed value
maxUInt  = 0FFFFh               ; Maximum 16-bit unsigned value
string   = 'XY'                 ; Up to two characters allowed
count    = 500
endvalue = count + 1            ; Can use a predefined symbol

.386
maxLong  = 7FFFFFFFh            ; Maximum 32-bit signed value
minLong  = 80000000h            ; Minimum 32-bit signed value
maxULong = 0FFFFFFFFh           ; Maximum 32-bit unsigned value
```

A symbol defined with the equal-sign directive can be redefined any number of times. In Example 2, **count** changes value several times. On the right side of the example, we see how the assembler evaluates the constant:

Example 2. Using the Equal-Sign Directive.

Statement	Assembled As
`count = 5`	
`mov al,count`	`mov al,5`
`mov dl,al`	`mov dl,al`
`count = 10`	
`mov cx,count`	`mov cx,10`
`mov dx,count`	`mov dx,10`
`count = 2000`	
`mov ax,count`	`mov ax,2000`

3.5.2 EQU Directive

The EQU directive assigns a symbolic name to a string or numeric constant. This increases the readability of a program and makes it possible to change multiple occurrences of a constant from a single place in a program. There is an important limitation imposed on EQU: A symbol defined with EQU *cannot* be redefined later in the program.

Expressions containing integers evaluate to numeric values, but floating point values evaluate to strings. Also, string equates may be enclosed in angle brackets (< . . . >) to ensure their interpretation as string expressions. This eliminates ambiguity on the part of the assembler when assigning the correct value to a name:

Example	Type of Value
`maxint equ 32767`	Numeric
`maxuint equ 0FFFFh`	Numeric
`count equ 10 * 20`	Numeric
`float1 equ <2.345>`	String

3.5.3 TEXTEQU Directive

The TEXTEQU directive creates what is called a text macro. You can assign a sequence of characters to a symbolic name, and then use the name later in the program. The syntax is:

```
name TEXTEQU <text>
name TEXTEQU textmacro
name TEXTEQU %constExpr
```

In this syntax, *text* is any sequence of characters enclosed in angle brackets <...>, *textmacro* is a previously defined text macro, and *constExpr* is an expression that evaluates to text. Text macros can appear anywhere in a program's source code. A symbol defined with TEXTEQU can be redefined later in the program.

A symbolic name can be assigned to a string, allowing the name to be replaced by the string wherever it is found. For example, the **prompt1** variable references the **continueMsg** text macro:

```
continueMsg textequ <"Do you wish to continue (Y/N)?">
.data
prompt1 db continueMsg
```

An alias, which is a name representing another predefined symbol, can be created. For example:

```
;Symbol declarations:
move     textequ <mov>
address  textequ <offset>

;Original code:
move bx,address value1
move al,20

;Assembled as:
mov bx,offset value1
mov al,20
```

In the following example, TEXTEQU is used to define a pointer (**p1**) to a string. Later, p1 is assigned a literal containing "0":

```
.data
myString db "A string",0

.code
p1 textequ <offset myString>
mov  bx,p1        ; bx = offset myString

p1 textequ <0>
mov  si,p1        ; si = 0
```

3.6 DATA TRANSFER INSTRUCTIONS

3.6.1 MOV Instruction

The MOV (*move data*) instruction copies data from one operand to another, so it is called a *data transfer* instruction. The following basic forms of MOV can be used, where the first operand is the *target* of the move, and the second operand is the source:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,immed
MOV reg,immed
```

In these formats, *reg* can be any non-segment register, except that IP cannot be a target operand. The sizes of both operands must be the same. A 16-bit register must be moved to a 16-bit memory location, for example.

Where segment registers are concerned, the following types of moves are possible, with the exception that CS cannot be a target operand:

```
mov segreg,reg16
mov segreg,mem16
mov reg16,segreg
mov mem16,segreg
```

Notably missing from the MOV instruction is the ability to use two memory operands[1]. Instead, you must use a register when copying a byte, word, or doubleword from one memory location to another. The following instructions, for example, copy a word from var1 to var2:

```
mov  ax,var1
mov  var2,ax
```

Examples of MOV with all three types of operands are shown here:

```
.data
count  db  10
total  dw  4126h
bigVal dd  12345678h

.code
mov  al,bl              ; 8-bit register to register
mov  bl,count           ; 8-bit memory to register
mov  count,26           ; 8-bit immediate to memory
mov  bl,1               ; 8-bit immediate to register
```

```
mov  dx,cx              ; 16–bit register to register
mov  bx,8FE2h           ; 16–bit immediate to register
mov  total,1000h        ; 16–bit immediate to memory
mov  eax,ebx            ; 32–bit register to register
mov  edx,bigVal         ; 32–bit memory to register
```

Type Checking. When a variable is created using DB, DW, DD, or any of the other data definition directives, the assembler gives it a default attribute (byte, word, doubleword) based on its size. This type is checked when you refer to the variable, and an error results if the types do not match. For example, the following MOV instruction is invalid because **count** has a *word* attribute and AL is a *byte:*

```
.data
count  dw  20h

.code
mov al,count   ; error: operand sizes must match
```

Type checking, while sometimes inconvenient, helps you avoid logic errors. Even when a smaller value fits into a larger one, a type mismatch error is flagged by the assembler:

```
.data
byteval  db  1

.code
mov  ax,byteval       ; errror
```

If necessary, you can use the LABEL directive to create a new name with a different attribute at the same offset. The same data can now be accessed using either name:

```
.data
countB label byte       ; byte attribute
countW dw  20h          ; word attribute

.code
mov  al,countB          ; retrieve low byte of count
mov  cx,countW          ; retrieve all of count
```

3.6.2 Operands with Displacements

You can add a displacement to the name of a memory operand, using a method called *direct-offset* addressing. This lets you access memory values that do not have their own labels. For example, the following are arrays of bytes, words, and doublewords:

```
arrayB  db 10h,20h
arrayW  dw 100h,200h
arrayD  dd 10000h,20000h
```

The notation **arrayB+1** refers to the location one byte beyond the beginning of **arrayB**, and **arrayW+2** refers to the location two bytes from the beginning of **arrayW**. You can code operands in MOV instructions that use this notation to move data to and from memory. In the following example, we show the value of AL after each move has taken place:

```
mov  al,arrayB          ; AL = 10h
mov  al,arrayB+1        ; AL = 20h
```

When dealing with an array of 16-bit values, the offset of each array member is two bytes beyond the previous one:

```
mov  ax,arrayW          ; AX = 100h
mov  ax,arrayW+2        ; AX = 200h
```

The members of a doubleword array are four bytes apart:

```
mov  eax,arrayD         ; EAX = 10000h
mov  eax,arrayD+4       ; EAX = 20000h
```

3.6.3 XCHG Instruction

The XCHG (*exchange data*) instruction exchanges the contents of two registers, or the contents of a register and a variable. The syntax is:

```
XCHG reg,reg
XCHG reg,mem
XCHG mem,reg
```

XCHG is the most efficient way to exchange two operands, because you don't need a third register or variable to hold a temporary value. Particularly in sorting applications, this instruction provides a speed advantage. One or both operands can be registers, or a register can be combined with a memory operand, but two memory operands cannot be used togther. For example:

```
xchg  ax,bx      ; exchange two 16-bit registers
xchg  ah,al      ; exchange two 8-bit registers
xchg  var1,bx    ; exchange 16-bit memory operand with BX
xchg  eax,ebx    ; exchange two 32-bit registers
```

If you do want to exchange two variables, a register must be used as a temporary operand. The program in Example 3 exchanges the contents of two variables.

Example 3. Exchanging Two Variables.

```
title Exchange Two Variables       (Exchange.asm)

.model small
.stack 100h
.data
value1 db 0Ah
value2 db 14h

.code
main proc
    mov  ax,@data     ; initialize DS register
    mov  ds,ax

    mov  al,value1    ; load the AL register
    xchg value2,al    ; exchange AL and value2
    mov  value1,al    ; store AL back into value1

    mov  ax,4C00h     ; exit program
    int  21h
main endp
end main
```

3.7 ARITHMETIC INSTRUCTIONS

Hardly any computer program can avoid performing arithmetic. The Intel instruction set has instructions for integer arithmetic, using 8, 16, and 32-bit operands. Floating-point operations are handled in one of three ways: (1) a special math coprocessor, (2) software that emulates the math coprocessor, and (3) software that converts floating-point values to integers, calculates, and then converts the numbers back to floating-point.

In this chapter, we focus on a few simple arithmetic instructions. The INC and DEC instructions add 1 to or subtract 1 from an operand, and the ADD and SUB instructions perform the addition and subtraction of integers.

3.7.1 INC and DEC Instructions

The INC (*increment*) and DEC (*decrement*) instructions add 1 or subtract 1 from a single operand, respectively. Their syntax is

```
INC destination
DEC destination
```

Destination can be a register or memory operand. All status flags are affected except the Carry flag. Examples are shown here.

```
inc  al    ; increment 8-bit register
dec  bx    ; decrement 16-bit register
inc  eax   ; increment 32-bit register
inc  membyte               ; increment memory operand
dec  byte ptr membyte      ; increment 8-bit memory operand
dec  memword               ; decrement memory operand
inc  word ptr memword      ; increment 16-bit memory operand
```

In these examples, the BYTE PTR operator identifies an 8-bit operand, and WORD PTR identifies a 16-bit operand.

3.7.2 ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. The syntax is:

```
ADD  destination,source
```

Source is unchanged by the operation, and *destination* is assigned the sum. The sizes of the operands must match, and no more than one operand can be a memory operand. A segment register cannot be the destination. All status flags are affected. Examples are as follows:

```
add  cl,al       ; add 8-bit register to register
add  eax,edx     ; add 32-bit registers
add  bx,1000h    ; add immediate value to 16-bit register
add  var1,ax     ; add 16-bit register to memory
add  dx,var1     ; add 16-bit memory to register
add  var1,10     ; add immediate value to memory
add  dword ptr memVal, ecx
```

The DWORD PTR operator identifies a 32-bit memory operand.

3.7.3 SUB Instruction

The SUB instruction subtracts a source operand from a destination operand. The syntax is:

```
SUB destination,source
```

The sizes of the two operands must match, and only one can be a memory operand. Inside the CPU, the source operand is first negated and then added to the destination. For

example, 4 – 1 is really 4 + (–1). Recall that twos complement notation is used for negative numbers, so –1 is stored as 11111111:

```
  0 0 0 0 0 1 0 0   ( 4 )
+ 1 1 1 1 1 1 1 1   (–1 )
-----------------
  0 0 0 0 0 0 1 1   ( 3 )
```

Examples of SUB used with various types of operands are shown here:

```
sub  eax,12345h     ; subtract 32–bit immediate from register
sub  cl,al          ; subtract 8–bit register from register
sub  edx,eax        ; subtract 32–bit register from register
sub  bx,1000h       ; subtract immediate value from 16–bit register
sub  var1,ax        ; subtract 16–bit register from memory
sub  dx,var1        ; subtract 16–bit memory from register
sub  var1,10        ; subtract immediate value from memory
```

3.7.4 Flags Affected by ADD and SUB

If either ADD or SUB generates a result of zero, the Zero flag is set; if the result is negative, the Sign flag is set. In the following example, line 1 generates a result of zero, and line 4 generates a result of –1 (FFFFh).

```
mov  ax,10
sub  ax,10          ; AX = 0, ZF = 1
mov  bx,1
sub  bx,2           ; BX = FFFF, SF = 1
```

The *Zero flag* is set when the result of an arithmetic operation equals 0. Note that INC and DEC affect the Zero flag, but *not* the Carry flag:

```
mov   bl,4Fh
add   bl,0B1h       ; BL = 00, ZF = 1, CF = 1
mov   ax,0FFFFh
inc   ax            ; ZF = 1 (CF not affected)
```

The identification of an operand as either signed or unsigned is completely up to the programmer. The CPU updates the Carry and Overflow flags to cover both possibilities. For this reason, we need to discuss the two types of operations separately.

Unsigned Operations. When performing unsigned arithmetic, the Carry flag is useful. If the result of an addition operation is too large for the destination operand, the Carry flag is set. For example, the sum of 0FFh + 1 should equal 100h, but only the two lowest digits (00) fit into AL. The addition sets the Carry flag:

```
mov  ax,0FFh
add  al,1          ; AL = 00, CF = 1
```

This is an 8-bit operation because AL is used. If we want to get the right answer, we must add 1 to AX, making it a 16-bit operation:

```
mov  ax,0FFh
add  ax,1          ; AX = 0100, CF = 0
```

A similar situation occurs when subtracting a larger operand from a smaller one. In the next example, the Carry flag tells us the result in AL is invalid:

```
mov   al,1
sub   al,2         ; AL = FF, CF = 1
```

Signed Operations. The *Overflow flag* is set when an addition or subtraction operation generates a out-of-range signed value. You may recall that signed 8-bit numbers can range from –128 to +127, and 16-bit numbers can range from –32,768 to +32,767. If we exceed these ranges (as in the following examples), the Overflow flag is set and the signed results are invalid:

Example 1:

```
mov  al,+126      01111110
add  al,2        + 00000010
                  ________
                  10000000  AL = 80h, OF = 1
```

Example 2:

```
mov  al,–128      10000000
sub  al,2        – 00000010
                  ________
                  01111110  AL = 7Eh, OF = 1
```

3.8 BASIC OPERAND TYPES

There are three basic types of operands: *immediate*, *register*, and *memory*. An immediate operand is a constant. A register operand is one of the CPU registers. A memory operand is a reference to a location in memory.

The Intel instruction set provides a wide variety of ways of representing memory operands, to make it easier to handle arrays and other more complex data structures. There are six types of memory operands, shown in Table 3. We will use the first three types, *direct, direct-offset,* and *register indirect,* in the current chapter and defer the others to Chapter 4.

Some terms used in the table must be explained: A *displacement* is either a number or the offset of a variable. The *effective address* of an operand refers to the offset (distance) of the data from the beginning of its segment. Each operand type in Table 3 refers to the contents of memory at an effective address. The *addressing mode* used by an instruction refers to the type of memory operand in use. For example, the following instruction uses the *register indirect* addressing mode:

```
mov ax,[si]
```

To use a real-life analogy, each house in a neighborhood is assigned a unique address. Suppose a house is located at 121 Maple Street. **Maple Street** could be considered the house's base location, and **121** could be considered the house's *offset* from the beginning of the street. We could also use relative references to houses by using phrases such as "the second house after the house at 121 Maple Street." In a computer program, we might refer to a particular element in an array as: `intArray + 2`.

3.8.1 Register Operands

A register operand can be any register. In general, the register addressing mode is the most efficient because registers are part of the CPU and no memory access is required. Some examples using the MOV instruction with register operands are shown here:

```
mov  eax,ebx
mov  cl,20h
mov  si,offset var1
```

Table 3. Memory Operand Types.

Operand Type	Examples	Description
direct	`op1 bytelist`	EA is the offset of a variable.
direct-offset	`bytelist + 2`	EA is the sum of a variable's offset and a displacement.
register-indirect	`[si]` `[bx]`	EA is the contents of a base or index register.

indexed	`list[bx]`
	`[list + bx]`
	`list[di]`
	`[si+2]`
		EA is the sum of a base or index register and a displacement.

base-indexed	`[bx+di]`
	`[bx][di]`
	`[bp-di]`
		EA is the sum of a base register and an index register.

base-indexed with displacement	`[bx+si+2]`
	`list[bx+si]`
	`list[bx][si]`
		EA is the sum of a base register, an index register, and a displacement.

3.8.2 Immediate Operands

An immediate operand is a constant expression, such as a number, character constant, arithmetic expression, or symbolic constant. The assembler must be able to determine the value of an immediate operand at assembly time. Its value is inserted directly into the machine instruction. Examples of immediate operands are shown here:

```
mov  al,10
mov  eax,12345678h
mov  dl,'X'
mov  ax,(40 * 50)
```

3.8.3 Direct Operands

A direct operand refers to the contents of memory at a location identified by a label in the data segment. At runtime, the CPU assumes that the offset of any variable is from the beginning of the segment addressed by the DS (data segment) register. Here are examples of direct addressing using byte, word, and doubleword operands:

```
.data
count      db  20
wordList   dw  1000h,2000h
longVal    dd  0F63B948Ch
.code
mov  al,count
mov  bx,wordList + 2
mov  edx,longVal
```

OFFSET Operator. The OFFSET operator returns the 16-bit offset of a variable. The assembler automatically calculates every variable's offset as a program is being assembled. In the following example, if the variable **aWord** is located at offset 0000, the MOV statement moves 0 to BX:

```
.data
aWord dw 1234h
.code
mov  bx,offset aWord       ; BX = 0000
```

3.8.4 Direct-Offset Operands

A particularly good use of the addition and subtraction operators (+, –) is to access a list of values. The + operator adds to the offset of a variable. In the following series of instructions, the first byte of **array** is moved to AL, the second byte to BL, the third byte to CL, and the fourth byte to DL. The value of each register after a move is shown at the right:

```
.data
array db 0Ah,0Bh,0Ch,0Dh
.code
mov  al,array             ; AL = 0Ah
mov  bl,array+1           ; BL = 0Bh
mov  cl,array+2           ; CL = 0Ch
mov  dl,array+3           ; DL = 0Dh
```

You can also subtract from a label's offset. In the following example, the label **endlist** is one byte beyond the last byte in **list**. To move the last byte in **list** to AL, we write:

```
.data
list    db  1,2,3,4,5
endlist label byte
.code
mov   al,endlist-1        ; move 5 to AL
```

In a list of 16-bit numbers, add 2 to a number's offset to get the offset of the next element. This is done in the following example:

```
.data
wvals  dw  1000h,2000h,3000h,4000h
.code
mov   ax,wvals            ; AX = 1000h
mov   bx,wvals+2          ; BX = 2000h
mov   cx,wvals+4          ; CX = 3000h
mov   dx,wvals+6          ; DX = 4000h
```

3.9 REVIEW QUESTIONS

1. Show three examples of assembly language instructions: one with no operands, one with a single operand, and one with two operands.
2. Write an example of an assembly language mnemonic.
3. Explain what a *disassembler* utility does.
4. Can a comment be placed on the same line as an instruction? (y/n)
5. Name three types of objects that can be represented by operands.
6. Name at least two popular debuggers used with assemblers sold today.
7. Write a series of instructions that move the values 1, 2 and 3 to the AX, BX, and CX registers.
8. Write an instruction that adds the number in the BX register to the CX register.

9. Identify each of the following Debug commands:

```
A 100
T
R
G
Q
```

10. Addresses are shown in Debug as a combination of two numbers, called the segment and the _____.
11. Show the storage of memory bytes for the 16-bit value 0A6Bh.
12. Show several examples of integer constants.
13. Can a symbolic constant contain an arithmetic expression? (y/n)
14. Show an example of assigning a numeric constant to a symbol, using the = operator.
15. Which radix character is used for hexadecimal constants?
16. Create several examples of string constants, including one that contains embedded quotes.
17. Name the four basic parts of assembly language statements.
18. Show how a source program statement can be divided between two lines.
19. Can multiple statements appear on the same line? (y/n)
20. In assembly language, how is a *directive* different from an *instruction*?
21. Name the six general types of memory operands.
22. Which special characters can appear in identifier names?
23. Would the following be a valid identifier name? first$try
24. Show several examples of labels.
25. How are labels used in programs?
26. What are segments in an assembly language program?
27. What is the diffference between the .stack and .code directives?
28. Write a statement that copies the location of the data segment into the DS register.
29. If any of the following MOV statements are illegal, explain why:

```
a. mov ax,bx
b. mov var2,al
c. mov ax,bl
d. mov bh,4A6Fh
e. mov dx,3
```

```
f.  mov var1,bx
g.  mov  al,var3
h.  mov  cs,0
i.  mov  ip,ax
j.  mov  var1,var2
k.  mov  ds,1000h
l.  mov  ds,es

.data
var1   dw    0
var2   dw    6
var3   db    5
```

30. The three basic types of operands are *register, memory,* and _____.

31. Is the address of each operand relative to the start of the program calculated at *assembly* time or at *link* time?

32. Which directive marks the end of a procedure?

33. What is the significance of the label used with the END directive?

34. Identify the types of operands (register, immediate, direct, or indirect) used in each of the following instructions:

```
a.  mov   al,20
b.  add   cx,wordval
c.  mov   bx,offset count
d.  add   dl,[bx]
```

35. Mark and correct any syntax errors in the following listing:

```
.data
blist  db 1,2,3,4,5
wlist  dw 6,7,8,9,0Ah

.code
mov  al,blist
add  al,wlist+1
mov  bx,offset blist
mov  cx,wlist
mov  dx,cx
inc  word ptr dx
dec  ax
```

36. Write a data definition for the following string:

    "MYFILE.DTA"

37. Write a data definition statement for a list of 8-bit memory operands containing the following values:

    3, 15h, 0F6h, 11010000b

38. Write a data declaration directive for a sequence of 500 16-bit words, each containing the value 1000h.

39. An operand in an instruction can be a memory variable, a register, or _____ .

40. Which of the following registers cannot be used as destination operands?

    AX, CL, IP, DX, CS, BH, SS, SP, BP

41. What will be the hexadecimal value of the destination operand after each of the statements in Table 4 has executed? (If any instruction is illegal, write the word ILLEGAL as the answer.) **var1** and **var2** are 16-bit operands, and **count** is 8 bits long. All numbers are in hexadecimal.

42. Write a data definition for the variable **arrayptr** that contains the offset address of the variable **intarray**.

Table 4. Examples for Question 41.

Instruction	Before	After
a. mov ax,bx	AX = 0023, BX = 00A5	AX =
b. mov ah,3	AX = 06AF	AX =
c. mov dl,count	DX = 8F23, count = 1A	DL =
d. mov bl,ax	BX = 00A5, AX = 4000	BL =
e. mov di,100h	DI = 06E9	DI =
f. mov ds,cx	DS = 0FB2, CX = 0020	DS =
g. mov var1,bx	var1 = 0025, BX = A000	var1 =
h. mov count,ax	count = 25, AX = 4000	count =
i. mov var1,var2	var1 = 0400, var2 = 0500	var1 =

43. What will be the hexadecimal value of the destination operand after each of the statements in Table 5 has executed? You may assume that **var1** is a word variable and that **count** and **var2** are byte variables. If any instruction is illegal, write the word ILLEGAL as the answer:

44. What will AX equal after the following instructions have executed?

```
.code
  mov   ax,array1
  inc   ax
  add   ah,1
  sub   ax,array1
.data
array1  dw  10h,20h
array2  dw  30h,40h
```

45. As each of the following instructions is executed, fill in the hexadecimal value of the operand listed on the right side:

```
.code
mov   ax,array1          ; AX =
xchg  array2,ax          ; AX =
dec   ax
```

Table 5. Examples for Question 43.

Instruction	Before	After
a. mov ah,bl	AX = 0023 BX = 00A5	AX =
b. add ah,3	AX = 06AF	AX =
c. sub dl,count	DX = 8F23, count = 1A	DX =
d. inc bl	BX = FFFF	BX =
e. add di,100h	DI = 06E9	DI =
f. dec cx	CX = 0000	CX =
g. add var1,bx	var1 = 0025, BX = A000	var1 =
h. xchg var2,al	AL = 41, var2 = 25	var2 =
i. sub var1,var2	var1 = 15A6, var2 = B8	var1 =
j. dec var2	var2 = 01	var2 =

```
sub    array2,2           ; array2 =
mov    bx,array2
add    ah,bl        ; AX =

.data
array1  dw  20h,10h
array2  dw  30h,40h
```

3.10 PROGRAMMING EXERCISES

The following exercises must be completed by creating a source file and assembling it with an assembler (MASM or TASM). Be sure to trace the execution of the program with a debugger.

1. Program Trace

Code a program containing the following list of instructions. Where marked, write down the anticipated values of the Carry, Sign, Zero, and Overflow flags before you actually run the program:

```
mov  ax,1234h
mov  bx,ax
mov  cx,ax
add  ch,al       ; CF =    , SF =     ,ZF =    , OF =
add  bl,ah       ; CF =    , SF =     ,ZF =    , OF =
add  ax,0FFFFh    ; CF =    , SF =     ,ZF =    , OF =
dec  bx          ; CF =    , SF =     ,ZF =    , OF =
inc  ax          ; CF =    , SF =     ,ZF =    , OF =
```

Also, before running the program, write down what you think AX, BX, CX, and DX will contain at the end of the program. Finally, run and trace the program with a debugger. Verify the register and flag values that you wrote down before running the program.

2. Define and Display 8-Bit Numbers

Write, assemble, and test a program to do the following:

Use the DB directive to define the following list of numbers and name it **array:**

```
31h, 32h, 33h, 34h
```

Write instructions to load each number into DL and display it on the console. (The following instructions display the byte in DL on the console:)

```
mov   ah,2
int   21h
```

Explain why the output on the screen is "1234".

3. Arithmetic Sums

Write a program that finds the sum of three 8-bit values and places the sum in another variable. Use the following data definitions. Use direct-offset addressing:

```
ThreeBytes   db  10h, 20h, 30h
TheSum       db  ?
```

4. Uppercase Conversion

Write a program that converts a string containing up to 256 lowercase characters to uppercase. (A lowercase character can be converted to uppercase by subtracting 32 from its ASCII code.)

5. Extended Registers (80386)

Write a program that moves various 32-bit memory operands to the EAX, EBX, ECX, and EDX registers. Experiment with each of the addressing modes introduced in this chapter. (This exercise requires an 80386 processor or higher.)

6. Simple Number Sequence

Write a program that generates a sequence of numbers in which each number is equal to double the previous number. The range is 1 – 1000h, shown here in hexadecimal:

```
1 2 4 8 10 20 40 80 100 200 400 800 1000
```

Use a debugger to dump the area of memory containing the numbers.

7. Fibonacci Numbers

The well-known *Fibonacci* number series, reputedly discovered by Leonardo of Pisa around the year 1200, has been valued for centuries for its universal qualities by artists, mathematicians, and composers. Each number in the series after the number 1 is the sum of the two previous numbers:

```
1, 1, 2, 3, 5, 8, 13 ,21, 34, 55 . . .
```

Write a program that generates and displays the first 24 numbers in the Fibonacci series, beginning with 1 and ending with 46,368.

End Notes:

[1] There is a specialized type of memory-to-memory move instruction called MOVS, which is often used when moving large blocks of data. See Chapter 10 for details.