3

Assembly Language Fundamentals

3.1 Basic Elements of Assembly Language

Chapter 2 gave you some essential basics of computer hardware as well as specific knowledge of the IA-32 architecture. Now it's time to get practical and apply that knowledge. If you were a cook, I would now be showing you around the kitchen, explaining how to use mixers, grinders, knives, stoves, and saucepans. We're going to take the ingredients of assembly language, mix them together, and come up with working programs.

Assembly language programmers absolutely must first know their data backwards and forwards before writing executable code. Part of that goal was accomplished in Chapter 1, where you learned about various number systems and the binary storage of integers and characters. In this chapter, you will learn how to define and declare variables and constants, using Microsoft Assembler (MASM) syntax. Then you will get to see a complete program, which we dissect line by line. You can expand and modify the programs in this chapter as much as you wish, using the new knowledge you've gained.

3.1.1 Integer Constants

An *integer constant* (or integer literal) is made up of an optional leading sign, one or more digits, and an optional suffix character (called a *radix*) indicating the number's base:

```
[{+|−}] digits [radix]
```

Microsoft syntax notation is used throughout this chapter. Elements within square brackets [..] are optional, and elements within braces {..} require a choice of one of the enclosed elements (separated by the | character). Elements in *italics* denote items which have known definitions or descriptions.

The *radix* may be one of the following (uppercase or lowercase):

h	hexadecimal	r	encoded real
q/o	octal	t	decimal *(alternate)*
d	decimal	y	binary (*alternate*)
b	binary		

If no radix is given, the integer constant is assumed to be decimal. Here are some examples using different radixes:

26	decimal	42o	octal
26d	decimal	1Ah	hexadecimal
11010011b	binary	0A3h	hexadecimal
42q	octal		

A hexadecimal constant beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.

3.1.2 Integer Expressions

An *integer expression* is a mathematical expression involving integer values and arithmetic operators. The expression must evaluate to an integer which can be stored in 32 bits (0 – FFFFFFFFh). The arithmetic operators are listed in Table 3–1 according to their precedence order, from highest (1) to lowest (4).

Table 3–1 Arithmetic Operators.

Operator	Name	Precedence Level
( )	parentheses	1
+,-	unary plus, minus	2
***,/**	multiply, divide	3
MOD	modulus	3
+,-	add, subtract	4

Precedence refers to the implied order of operations when an expression contains two or more operators. The order of operations is shown for the following expressions:

```
4 + 5 * 2          multiply, add
12 - 1 MOD 5       modulus, subtract
-5 + 2             unary minus, add
(4 + 2) * 6        add, multiply
```

The following are examples of valid expressions and their values:

Expression	Value
`16 / 5`	`3`
`-(3 + 4) * (6 - 1)`	`-35`
`-3 + 4 * 6 - 1`	`20`
`25 mod 3`	`1`

It's a good idea to use parentheses in expressions to clarify the order of operations. Then you don't have to remember the precedence rules.

3.1.3 Real Number Constants

There are two types of real number constants: decimal reals and encoded (hexadecimal) reals. A *decimal real* constant contains a sign followed by an integer, a decimal point, an integer that expresses a fraction, and an exponent:

```
[sign]integer.[integer][exponent]
```

This is how we describe the sign and exponent:

```
sign        {+,-}
exponent  E[{+,-}]integer
```

The sign is optional, and the choices are + or –. Following are examples of valid real constants:

```
2.
+3.0
-44.2E+05
26.E5
```

At the very least, there must be a digit and a decimal point. Without the decimal point, it would just be an integer constant.

Encoded Reals You can specify a real constant in hexadecimal as an *encoded real* if you know the exact binary representation of the number. The following, for example, is the encoded 4-byte real representation of decimal +1.0:

```
3F800000r
```

(We will delay the discussion of IEEE real number formats until Chapter 17, stored on the book's CD-ROM.)

3.1.4 Character Constants

A *character constant* is a single character enclosed in either single or double quotes. The assembler converts it to the binary ASCII code matching the character. Examples are:

```
'A'
"d"
```

A complete list of ASCII codes is printed on the inside back cover of this book.

3.1.5 String Constants

A *string constant* is a string of characters enclosed in either single or double quotes:

```
'ABC'
'X'
"Goodnight, Gracie"
'4096'
```

Embedded quotes are permitted when used in the manner shown by the following examples:

```
"This isn't a test"
'Say "Goodnight," Gracie'
```

3.1.6 Reserved Words

Assembly language has a list of words called *reserved words.* These have special meaning and can only be used in their correct context. Reserved words can be any of the following:

- Instruction mnemonics, such as MOV, ADD, or MUL, which correspond to built-in operations performed by Intel processors.
- Directives, which tell MASM how to assemble programs.
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD.
- Operators, used in constant expressions.
- Predefined symbols, such as @data, which return constant integer values at assembly time.

A complete list of MASM reserved words can be found in Appendix D.

3.1.7 Identifiers

An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. Keep the following in mind when creating identifiers:

- They may contain between 1 and 247 characters.
- They are not case-sensitive.

- The first character must be either a letter (A..Z, a..z), underscore (_), @ , or $. Subsequent characters may also be digits.
- An identifier cannot be the same as an assembler reserved word.

> You can make all keywords and identifiers case-sensitive by adding the –Cp command line switch when running the assembler.

Avoid using a single @ sign as the first character, because it is used extensively by the assembler for predefined symbols. Here are some valid identifiers:

```
var1                Count               $first
_main               MAX                 open_file
@@myfile            xVal                _12345
```

Common sense suggests that you should make identifier names descriptive and easy to understand.

3.1.8 Directives

A *directive* is a command that is recognized and acted upon by the assembler as the program's source code is being assembled. Directives are used for defining logical segments, choosing a memory model, defining variables, creating procedures, and so on.

Directives are part of the assembler's syntax, but are not related to the Intel instruction set. Various assemblers may generate identical machine code for the Intel processor, but their sets of directives need not be the same.

Different capitalizations of the same directive are assumed to be equivalent. For example, the assembler does not recognize any difference between **.data**, **.DATA**, and **.Data**.

Examples The .DATA directive identifies the area of a program that contains variables:

```
.data
```

The .CODE directive identifies the area of a program that contains instructions:

```
.code
```

The PROC directive identifes the beginning of a procedure. *Name* may be any identifier:

```
   name PROC
```

It would take a very long time to learn all the directives in MASM, so we will necessarily concentrate on the the few that are most essential. Appendix D contains a complete reference to all MASM directives and operators.

3.1.9 Instructions

An *instruction* is a statement that is executed by the processor at runtime after the program has been loaded into memory and started. An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

Aome source code lines may consist only of labels or comments. The following diagram shows the standard format for instructions:

Label :	Mnemonic	Operand(s)	; Comment

Let's explore each part separately, starting with the *label* field, which is optional.

3.1.9.1 Label

A *label* is an identifier that acts as a place marker for either instructions or data. In the process of scanning a source program, the assembler assigns a numeric address to each program statement. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

Why use labels at all? We could directly reference numeric addresses in our program code. For example, the following instruction moves a 16-bit word from memory location 0020 to the AX register:[1]

```
mov ax,[0020]
```

But when new variables are inserted in programs, the addresses of all subsequent variables automatically change. A reference such as [0020] would have to be modified manually. Clearly, this creates a headache for programmers, and is not worth the effort. Instead, if location 0020h is assigned a label, the assembler automatically matches the label to the address. Now the same MOV instruction could be written as:

```
mov ax,myVariable
```

Of course, we're getting ahead a bit ahead of ourselves. Variable definitions will be explained in Section 3.4.2, and the MOV instruction will be explained in Section 3.2.3.

Code Labels A label in the code area of a program (where instructions are located) must end with a colon (:) character. In this context, labels are often used as targets of jumping and looping

1. Don't try to assemble this instruction. It is only here for illustrative purposes.

instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

```
target:
   mov ax,bx
   ...
   jmp target
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
target: mov ax,bx
target:
```

Data Labels If a label is used in the data area of a program (where variables are defined) , it cannot end with a colon. Here is an example that defines a variable named **first**:

```
first BYTE 10
```

Label names are created using the rules for identifiers already shown in Section 3.1.7. Data label names must be unique within the same source file. If, for example, you have a label named **first**, then you cannot have another label named **first** anywhere in the same source code file.

3.1.9.2 Instruction Mnemonic

An *instruction mnenonic* is a short word that identifies the operation carried out by an instruction. In the English dictionary, a mnemonic is generally described as a *device that assists memory.* That is why instruction mnemonics have useful names such as mov, add, sub, mul, jmp, and call:

mov	move (assign) one value to another
add	add two values
sub	subtract one value from another
mul	multiply two values
jmp	jump to a new location
call	call a procedure

3.1.9.3 Operands

An assembly language instruction can have between zero and three operands, each of which can be a register, memory operand, constant expression, or I/O port. We discussed register names in Chapter 2, and we discussed constant expressions in Section 3.1.2. (We will leave the discussion of I/O ports for a later chapter.) A memory operand is specified either by the name of a variable or by a register that contains the address of a variable. A variable name implies the address of the variable, and instructs the computer to reference the contents of memory at the given address, as shown in the following table:

Example	Operand Type
`96`	constant (*immediate value*)
`2 + 4`	constant expression
`eax`	register
`count`	variable name

Following are some examples of assembly language instructions with various numbers of operands. The STC instruction, for example, has no operands:

```
stc                    ; set Carry flag
```

The INC instruction has one operand:

```
inc  ax                ; add 1 to AX
```

The MOV instruction has two operands:

```
mov count,bx           ; move BX to count
```

3.1.9.4 Comments

Comments, as you probably know, are an important way for the writer of a program to communicate information about how the program works to a person reading the source code. The following information is typically included at the top of a program listing:

- A short description of the program's overall purpose.
- The name of the programmer(s) who has written and/or revised the program.
- The date the program was written, along with revision dates.

Comments can be specified in two ways:

- Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler and may be used to comment the program.
- Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. For example:

```
COMMENT  !
   This line is a comment.
   This line is also a comment.
!
```

We can also use any other symbol:

```
COMMENT  &
    This line is a comment.
    This line is also a comment.
&
```

3.1.10 Section Review

1. List the valid suffix characters that may be used in integer constants.
2. *(Yes/No):* Is A5h a valid hexadecimal constant?
3. *(Yes/No):* Does the multiply sign (*) have a higher precedence than the divide sign (/) in integer expressions?
4. Write a constant expression that divides 10 by 3 and returns the integer remainder.
5. Show an example of a valid real number constant with an exponent.
6. *(Yes/No):* Must string constants be enclosed in single quotes?
7. Reserved words can be instruction mnemonics. attributes, operators, predefined symbols, and __________.
8. What is the maximum length of an identifier?
9. *(True/False):* An identifier cannot begin with a numeric digit.
10. *(True/False):* Assembly language identifiers are (by default) case-insensitive.
11. *(True/False):* Assembler directives execute at run time.
12. *(True/False):* Assembler directives can be written in any combination of uppercase and lowercase letters.
13. Name the four basic parts of an assembly language instruction.
14. *(True/False):* MOV is an example of an instruction mnemonic.
15. *(True/False):* A code label is followed by a colon (:), but a data label does not have a colon.
16. Show an example of a block comment.
17. Why would it not be a good idea to use numeric addresses when writing instructions that access variables?

3.2 Example: Adding Three Integers

3.2.1 Program Listing

Now it's time to look at that first working program we promised you in the chapter introduction. It's really trivial—it just adds and subtracts three integers, using CPU registers. At the end, the registers are displayed on the screen:

```
TITLE Add and Subtract           (AddSub.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
.code
main PROC

   mov eax,10000h                ; EAX = 10000h
   add eax,40000h                ; EAX = 50000h
   sub eax,20000h                ; EAX = 30000h
   call DumpRegs                 ; display registers

   exit
main ENDP
END main
```

3.2.2 Program Output

The following is a snapshot of the the program's output, generated by the call to the **DumpRegs** procedure:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

The first two rows show the hexadecimal values of the 32-bit general-purpose registers. Notice that EAX equals 00030000h, the value produced by the ADD and SUB instructions in the program. The third row shows the values of the EIP (extended instruction pointer) and EFL (extended flags) registers, as well as the values of the Carry, Sign, Zero, and Overflow flags.

3.2.3 Program Description

Let's go through the program line by line. In each case, the program code appears before its explanation:

```
TITLE Add and Subtract           (AddSub.asm)
```

The TITLE directive marks the entire line as a comment. You can put anything you want on this line.

```
; This program adds and subtracts 32-bit integers.
```

All text to the right of a semicolon is ignored by the assembler, so we use it for comments.

```
INCLUDE Irvine32.inc
```

The INCLUDE directive copies necessary definitions and setup information from a text file named *Irvine32.inc*, located in the assembler's INCLUDE directory. (The file is described in Chapter 5.)

```
.code
```

The **.code** directive marks the beginning of the *code segment*, where all executable statements in a program are located.

```
main PROC
```

The PROC directive identifies the beginning of a procedure. The name chosen for the only procedure in our program is **main**.

```
    mov eax,10000h                ; EAX = 10000h
```

The MOV instruction moves (copies) the integer 10000h to the EAX register. The first operand (EAX) is called the *destination operand*, and the second operand is called the *source operand*.

```
    add eax,40000h                ; EAX = 50000h
```

The ADD instruction adds 40000h to the EAX register.

```
    sub eax,20000h                ; EAX = 30000h
```

The SUB instruction subtracts 20000h from the EAX register.

```
    call DumpRegs                 ; display registers
```

The CALL statement calls a procedure that displays the current values of the CPU registers. This can be a useful way to verify that a program is working correctly.

```
    exit
main ENDP
```

The **exit** statement (indirectly) calls a predefined MS-Windows function that halts the program. The ENDP directive marks the end of the **main** procedure. Note that **exit** is not a MASM keyword; instead, it's a command defined in *Irvine32.inc* that provides a simple way to end a program.

```
END main
```

The END directive marks the last line of the program to be assembled. It identifies the name of the program's *startup* procedure (the procedure that starts the program execution).

Segments Programs are organized around segments, which are usually named code, data, and stack. The *code* segment contains all of a program's executable instructions. Ordinarily, the code segment contains one or more procedures, with one designated as the *startup* procedure. In the **AddSub** program, the startup procedure is **main**. Another segment, the *stack* segment, holds procedure parameters and local variables. The *data* segment holds variables.

Coding Styles You may be wondering at this point whether you should capitalize any particular keywords in assembly language programs. Because assembly language is case-insensitive, you are free to decide how to capitalize your programs, unless your instructor has specific requirements. Here are some varied approaches to capitalization that you can try:

- Capitalize nothing, except perhaps the initial letters of identifiers. C++ programmers often feel comfortable with this approach, since all their keywords are in lowercase. This approach makes the typing of source code lines fairly rapid.
- Capitalize everything: This approach was used in pre-1970 mainframe assembler programs, when computer terminals often did not support lowercase letters. It has the advantage of overcoming the effects of poor-quality printers and less-than-perfect eyesight.
- Use capital letters for all assembler reserved words, including instruction mnemonics and register names. This makes it easy to distinguish between user-defined names and assembler reserved words.
- Capitalize assembly language directives and operators. Leave everything else in lowercase. This is the approach used in the example programs throughout this book, except that .code and .data are lowercase.

3.2.3.1 Alternative Version of AddSub

You may have looked at the **AddSub** program and wondered exactly what was inside the *Irvine32.inc* file. To make your coding more convenient, we hide quite a few details that will be covered later in the book. Understandably, you (or your instructor) may prefer to create programs that do not depend on include files. The following version of AddSub hides nothing:

```
TITLE Add and Subtract              (AddSubAlt.asm)

; This program adds and subtracts 32-bit integers.

.386
.MODEL flat,stdcall
.STACK 4096
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC

    mov eax,10000h              ; EAX = 10000h
```

```
	add eax,40000h                ; EAX = 50000h
	sub eax,20000h                ; EAX = 30000h
	call DumpRegs

	INVOKE ExitProcess,0
main ENDP
END main
```

Several lines in the program are different from the original version. As before, we show each line of code followed by its explanation:

```
.386
```

The .386 directive identifies the minimum CPU required for this program (Intel386).

```
.MODEL flat,stdcall
```

The .MODEL directive instructs the assembler to generate code for a Protected mode program, and STDCALL enables the calling of MS-Windows functions.

```
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO
```

Two PROTO directives declare procedures used by this program: **ExitProcess** is an MS-Windows function that halts the current program (called a *process*), and **DumpRegs** is a procedure from the Irvine32 link library that displays registers.

```
INVOKE ExitProcess,0
```

The program ends by calling the **ExitProcess** function, passing it a return code of zero. INVOKE is an assembler directive that calls a procedure or function.

3.2.4 Program Template

Assembly language programs have a simple structure, with some small variations. When you begin to write and assemble your own programs, it helps to start with an empty shell program that has all the basic elements in place. You can avoid redundant typing by filling in the missing parts and saving the file under a new name. The following Protected-mode program (*Template.asm*) can easily be customized. Note that comments have been inserted, marking the points where your own code should be added:

```
TITLE Program Template           (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:              Modified by:
```

```
INCLUDE Irvine32.inc
.data
   ; (insert variables here)

.code
main PROC
   ; (insert executable instructions here)

   exit
main ENDP

   ; (insert additional procedures here)
END main
```

Use Comments Several comment fields have been inserted at the beginning of the program. It's a very good idea to include a program description, the name of the program's author, creation date, and information about subsequent modifications.

Documentation of this kind is useful to anyone who reads the program listing (including you, months or years from now). Many programmers have discovered, years after writing a program, that they must become reacquainted with their own code before they can modify it. If you're taking a programming course, your instructor may insist on additional information.

3.2.5 Section Review

1. In the AddSub program (Section 3.2), what is the meaning of the INCLUDE directive?
2. In the AddSub program, what does the .CODE directive identify?
3. What are the names of the segments in the AddSub program?
4. In the AddSub program, how are the CPU registers displayed?
5. In the AddSub program, which statement halts the program?
6. Which directive begins a procedure?
7. Which directive ends a procedure?
8. What is the purpose of the identifier in the END statement?
9. What does the PROTO directive do?

3.3 Assembling, Linking, and Running Programs

In earlier chapters we saw examples of simple machine-language programs, so it is clear that a source program written in assembly language cannot be executed directly on its target computer. It must be translated, or *assembled* into executable code. In fact, an assembler is very similar to a *compiler*, the type of program you would use to translate a C++ or Java program into executable code.

The assembler produces a file containing machine language called an *object file*. This file isn't quite ready to execute. It must be passed to another program called a *linker*, which in turn produces an *executable file*. This file is ready to execute from the MS-DOS/Windows command prompt.

3.3.1 The Assemble-Link-Execute Cycle

The process of editing, assembling, linking, and executing assembly language programs is summarized in Figure 3–1. Following is a detailed description of each step:

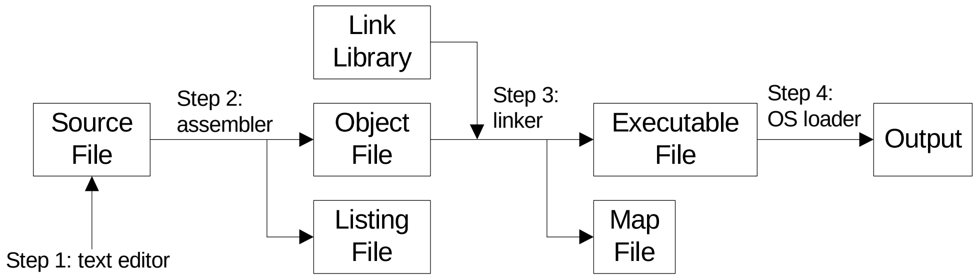
Step 1: A programmer uses a **text editor** to create an ASCII text file named the *source file*.

Step 2: The **assembler** reads the source file and produces an *object file,* a machine-language translation of the program. Optionally, it produces a *listing file*. If any errors occur, the programmer must return to Step 1 and fix the program.

Step 3: The **linker** reads the object file and checks to see if the program contains any calls to procedures in a link library. The **linker** copies any required procedures from the link library, combines them with the object file, and produces the *executable file*. Optionally, the linker can produce a *map file*.

Step 4: The operating system **loader** utility reads the executable file into memory, branches the CPU to the program's starting address, and the program begins to execute.

Figure 3–1 Assemble-Link-Execute Cycle.



Assembling and Linking 32-Bit Programs To assemble and link a Protected mode assembly language program, execute the following command at the MS-DOS prompt:

```
make32 progname
```

Progname is the base name of your assembly language source file, with no extension. For example, the *AddSub.asm* program would be assembled and linked with the following command:

```
make32 AddSub
```

The make32.bat file must be located either in the same directory as your ASM file or on the system path. Consult your operating system's documentation to find out how to add a directory to the system path. Refer to Appendix A for instructions on setting up your computer.

Assembling and Linking 16-Bit Programs If you are programming in Real-address mode, use the **make16** command to assemble and link. Using the *AddSub* program as an example, the command would be the following:

```
make16 AddSub
```

3.3.1.1 Listing File

A *listing* file contains a copy of the program's source code, suitable for printing, with line numbers, offset addresses, translated machine code, and a symbol table. Let's look at the listing file for the **AddSub** program we created in Section 3.2:

```
Microsoft (R) Macro Assembler Version 6.15.8803    10/26/01 13:50:21
Add and Subtract              (AddSub.asm)                 Page 1 - 1


                      TITLE Add and Subtract              (AddSub.asm)

                      ; This program adds and subtracts 32-bit integers.

                      INCLUDE Irvine32.inc
               C      ; Include file for Irvine32.lib (Irvine32.inc)
               C      INCLUDE SmallWin.inc

00000000              .code
00000000              main PROC

00000000  B8 00010000     mov eax,10000h     ; EAX = 10000h
00000005  05 00040000     add eax,40000h     ; EAX = 50000h
0000000A  2D 00020000     sub eax,20000h     ; EAX = 30000h
0000000F  E8 00000000E    call DumpRegs

                          exit
0000001B              main ENDP
                      END main

Structures and Unions: (omitted)

Segments and Groups:

N a m e                          Size    Length    Align  Combine Class
FLAT . . . . . . . . . . . . . . GROUP
STACK  . . . . . . . . . . . . . 32 Bit  00001000  DWord  Stack   'STACK'
```

```
_DATA  . . . . . . . . . . . . . 32 Bit  00000000  DWord  Public 'DATA'
_TEXT  . . . . . . . . . . . . . 32 Bit  0000001B  DWord  Public 'CODE'


Procedures,  parameters and locals (list abbreviated):

N a m e                   Type  Value     Attr
CloseHandle  . . . . . . P Near  00000000  FLAT  Length=00000000 External STDCALL
ClrScr . . . . . . . . . P Near  00000000  FLAT  Length=00000000 External STDCALL
.
.
main . . . . . . . . . . . P Near  00000000 _TEXT  Length=0000001B Public STDCALL


Symbols (list abbreviated):

N a m e                           Type   Value      Attr
@CodeSize  . . . . . . . . . . . Number 00000000h
@DataSize  . . . . . . . . . . . Number 00000000h
@Interface . . . . . . . . . . . Number 00000003h
@Model . . . . . . . . . . . . . Number 00000007h
@code  . . . . . . . . . . . . . Text             _TEXT
@data  . . . . . . . . . . . . . Text              FLAT
@fardata?  . . . . . . . . . . . Text              FLAT
@fardata . . . . . . . . . . . . Text              FLAT
@stack . . . . . . . . . . . . . Text              FLAT
.
.
exit . . . . . . . . . . . . . . Text   INVOKE ExitProcess,0
          0 Warnings
          0 Errors
```

3.3.1.2 Files Created or Updated by the Linker

Map File A *map* file is a text file (extension MAP) that contains information about the segments contained in a program being linked. It contains the following information:

- The EXE module name, which is the base name of the file.
- The timestamp from the program file header (not from the file system).
- A list of segment groups in the program, with each group's start address, length, group name, and class.
- A list of public symbols, with each address, symbol name, flat address, and module where the symbol is defined.
- The address of the program's entry point.

Program Database File When you assemble a program with the –Zi option (debugging), MASM creates a *program database file* (extension PDB). During the link step, the linker reads the PDB file and updates it. Then, when you run your program using a debugger, the latter is able to display the program’s source code and provide supplemental information about the program.

3.3.2 Section Review

1. What types of files are produced by the assembler?
2. *(True/False):* The linker extracts copies of compiled procedures from the link library file.
3. *(True/False):* When a program’s source code is modified, it must be assembled and linked again before it can be executed with the changes.
4. What is the name of the part of the operating system that reads the executable file and starts its execution?
5. What types of files are produced by the linker?

Read Appendix A before answering the following questions:

6. What command line option tells the assembler to produce a listing file?
7. What command line option tells the assembler to add debugging information?
8. What does the linker’s /SUBSYSTEM:CONSOLE option signify?
9. *Challenge:* List the names of at least four functions in the kernel32.lib file.
10. *Challenge:* Which linker option lets you specify the program’s entry point?

3.4 Defining Data

3.4.1 Intrinsic Data Types

MASM defines various intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type. A DWORD variable, for example, can hold any 32-bit integer value. Some types are slightly more restrictive, such as REAL4, which can only be initialized by a real number constant. In Table 3–2, all data types pertain to integers except the last three. In those, the notation "IEEE” refers to standard real number formats published by the IEEE Computer Society.

Table 3–2 Intrinsic Data Types.

Type	Usage
BYTE	8-bit unsigned integer
SBYTE	8-bit signed integer

Table 3–2 Intrinsic Data Types.

Type	Usage
WORD	16-bit unsigned integer (can also be a Near pointer in Real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in Protected mode)
SDWORD	32-bit signed integer
FWORD	48-bit integer (Far pointer in Protected mode)
QWORD	64-bit integer
TBYTE	80-bit (10 byte) integer
REAL4	32-bit (4 byte) IEEE short real
REAL8	64-bit (8 byte) IEEE long real
REAL10	80-bit (10 byte) IEEE extended real

3.4.2 Data Definition Statement

A *data definition statement* sets aside storage in memory for a variable and may optionally assign a name to the variable. We use data definition statements to create variables based on the assembler's intrinsic types (Table 3–2). Each data definition has the same syntax:

```
[name] directive initializer [,initializer]...
```

Initializers At least one *initializer* is required in a data definition, even if it is the **?** expression, which does not assign a specific value to the data. Additional initializers, if any, are separated by commas. For integer data types, *initializer* is an integer constant or expression that matches the size implied by the type (BYTE,WORD, etc.) Integer constants and expressions were explained in Section 3.1.1, and integer expressions were discussed in Section 3.1.2.

All initializers, regardless of their number format, are converted to binary data by the assembler. That is why initializers such as 00110010b, 32h, and 50d all produce the same binary value.

3.4.3 Defining BYTE and SBYTE Data

The BYTE (define byte) and SBYTE (define signed byte) directives, used in data definition statements, allocate storage for one or more unsigned or signed values. Each initializer must be an 8-bit integer expression or character constant. For example:

```
value1 BYTE  'A'          ; character constant
value2 BYTE   0           ; smallest unsigned byte
```

```
value3 BYTE  255           ; largest unsigned byte
value4 SBYTE –128          ; smallest signed byte
value5 SBYTE +127          ; largest signed byte
```

(We're capitalizing the BYTE and SBYTE keywords here for emphasis, but you can just as easily code them in lowercase letters.)

A variable can be left uninitialized by using a question mark for the initializer. This implies that the variable will be assigned a value at runtime by executable instructions:

```
value6 BYTE ?
```

Variable Name A *variable name* is a label that marks the offset of a variable from the beginning of its enclosing segment. For example, suppose that **value1** was located at offset 0 in the data segment and consumed one byte of storage. Then **value2** would be located at offset 1:

```
.data
value1 BYTE 10h
value2 BYTE 20h
```

DB Directive Earlier versions of the assembler used the DB directive to define byte data. You can still use DB, but it permits no distinction between signed and unsigned data:

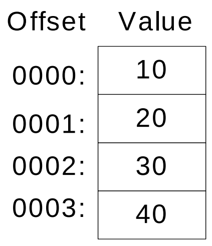
```
val1 DB 255                ; unsigned byte
val2 DB -128               ; signed byte
```

3.4.3.1 Multiple Initializers

If multiple initializers are used in the same data definition, its label refers only to the offset of the first byte. In the following example, assume that the label **list** is at offset 0. If so, the value 10 is at offset 0, 20 is at offset 1, 30 is at offset 2, and 40 is at offset 3:

```
.data
list BYTE 10,20,30,40
```

The following illustration shows **list** as a sequence of bytes, each with its own offset:



Not all data definitions require labels. If we wanted to continue the array of bytes begun with **list**, for example, we could define additional bytes on the next lines:

```
list BYTE 10,20,30,40
```

```
          BYTE 50,60,70,80
          BYTE 81,82,83,84
```

Within a single data definition, its initializers can use different radixes. Also, character and string constants can be freely mixed. In the following example, **list1** and **list2** have the same contents:

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

3.4.3.2 Defining Strings

To create a string data definition, enclose a sequence of characters in quotation marks. The most common type of string ends with a null byte, a byte containing the value 0. This type of string is used by C/C++, by Java, and by Microsoft Windows functions:

```
greeting1 BYTE "Good afternoon",0
```

Each character uses a byte of storage. Strings are an exception to the rule that byte values must be separated by commas. Without that exception, **greeting1** would have to be defined as

```
greeting1 BYTE 'G','o','o','d'....etc.
```

which would be exceedingly tedious.

A string can be spread across multiple lines without the necessity of supplying a label for each line, as the next example shows:

```
greeting1 BYTE "Welcome to the Encryption Demo program "
  BYTE "created by Kip Irvine.",0dh,0ah,
  BYTE "If you wish to modify this program, please "
  BYTE "send me a copy.",0dh,0ah,0
```

As a reminder, the hexadecimal bytes 0Dh and 0Ah are called either CR/LF (explained in Chapter 1) or *end-of-line characters*. When written to standard output, they move the cursor to the left column of the line following the current line.

MASM's line continuation character (\) may be used to concatenate two lines into a single program statement. The \ symbol may only be placed at the end of a line. In other words, the following statements are equivalent:

```
greeting1 BYTE "Welcome to the Encryption Demo program "
```

and

```
greeting1 \
BYTE "Welcome to the Encryption Demo program "
```

3.4.3.3 Using the DUP Operator

The DUP operator generates a repeated storage allocation, using a constant expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with both initialized and uninitialized data definitions:

```
BYTE 20 DUP(0)              ; 20 bytes, all equal to zero
BYTE 20 DUP(?)              ; 20 bytes, uninitialized
BYTE  4 DUP("STACK")        ; 20 bytes: "STACKSTACKSTACKSTACK"
```

3.4.4 Defining WORD and SWORD Data

The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit integers. Here are some examples:

```
word1  WORD   65535         ; largest unsigned value
word2  SWORD  -32768        ; smallest signed value
word3  WORD   ?             ; uninitialized, unsigned
```

Older versions of the assembler used the DW directive to define both signed and unsigned words. You can still use DW:

```
val1  DW 65535              ; unsigned
val2  DW -32768             ; signed
```

Array of Words You can create an array of word values either by explicitly initializing each element or by using the DUP operator. Here is an array containing specific values:

```
myList  WORD 1,2,3,4,5
```

Following is a diagram of the array in memory, if we assume that **myList** starts at offset 0. Notice that the addresses increment by 2:

Offset	Value
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

The DUP operator provides a convenient way to initialize multiple words:

```
array WORD 5 DUP(?)         ; 5 values, uninitialized
```

3.4.5 Defining DWORD and SDWORD Data

The DWORD (define doubleword) and SDWORD (define signed doubleword) directives allocate storage for one or more 32-bit integers. For example:

```
val1 DWORD   12345678h            ; unsigned
val2 SDWORD –2147483648           ; signed
val3 DWORD   20 DUP(?)            ; unsigned array
```

Older versions of the assembler used the DD directive to define both unsigned and signed doublewords. You can still use DD:

```
val1 DD 12345678h                 ; unsigned
val2 DD –2147483648               ; signed
```

Array of Doublewords You can create an array of doubleword values either by explicitly initializing each element or by using the DUP operator. Here is an array containing specific unsigned values:

```
myList DWORD 1,2,3,4,5
```

Shown below is a diagram of the array in memory, assuming that **myList** starts at offset 0. Notice that the offsets increment by 4:

Offset	Value
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

3.4.6 Defining QWORD Data

The QWORD (define quadword) directive allocates storage for 64-bit (8 byte) values. Here is an example:

```
quad1 QWORD 1234567812345678h
```

You can also use DQ, for compatibility with older assemblers:

```
quad1 DQ 1234567812345678h
```

3.4.7 Defining TBYTE Data

The TBYTE (define tenbyte) directive creates storage for 80-bit integers. This data type is primarily for the storage of binary-coded decimal numbers. Manipulating these values requires special instructions in the floating-point instruction set:

```
val1 TBYTE 1000000000123456789Ah
```

You can also use DT, for compatibility with older assemblers:

```
val1 DT 1000000000123456789Ah
```

3.4.8 Defining Real Number Data

REAL4 defines a 4-byte single-precision real variable. REAL8 defines an 8-byte double-precision real, and REAL10 defines a 10-byte double extended-precision real. Each requires one or more real constant initializers that can fit into the assigned storage:

```
rVal1      REAL4 -1.2
rVal2      REAL8  3.2E-260
rVal3      REAL10 4.6E+4096
ShortArray REAL4  20 DUP(0.0)
```

The following table describes each of the standard real types in terms of their minimum number of significant digits and approximate range:

Data Type	Significant Digits	Approximate Range
Short real	6	1.18×10^{-38} to 3.40×10^{38}
Long real	15	2.23×10^{-308} to 1.79×10^{308}
Extended-precision real	19	3.37×10^{-4932} to 1.18×10^{4932}

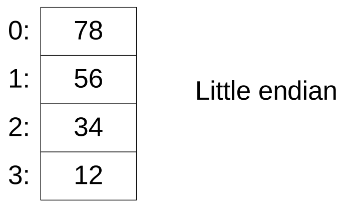
Programs written under earlier versions of the assembler used DD, DQ, and DT to define real numbers; these directives can still be used:

```
rVal1 DD -1.2
rVal2 DQ  3.2E-260
rVal3 DT  4.6E+4096
```

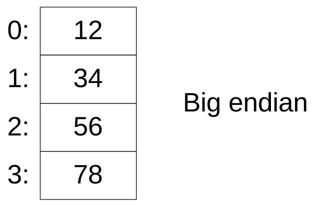
3.4.9 Little Endian Order

Intel processors store and retrieve data from memory using what is referred to as *little endian* order. This means that the least significant byte of a variable is stored at the lowest address. The remaining bytes are stored in the next consecutive memory positions.

Consider the doubleword 12345678h. If placed in memory at offset 0, 78h would be stored in the first byte. 56h would be stored in the second byte, and the remaining bytes would be at offsets 3 and 4, as the following diagram shows:



Some other computer systems use *big endian* order (high to low). The following figure shows an example of 12345678h stored in big endian order at offset 0:



3.4.10 Adding Variables to the AddSub Program

Let's return for a moment to the **AddSub** program we wrote in Section 3.2. Using the information we've developed regarding data definition directives, we can easily add a data segment containing several variables. The revised program is named **AddSub2**:

```
TITLE Add and Subtract, Version 2         (AddSub2.asm)

; This program adds and subtracts 32-bit unsigned
; integers and stores the sum in a variable.

INCLUDE Irvine32.inc
.data
val1  DWORD 10000h
val2  DWORD 40000h
val3  DWORD 20000h
finalVal DWORD ?

.code
main PROC
   mov  eax,val1          ; start with 10000h
   add  eax,val2          ; add 40000h
   sub  eax,val3          ; subtract 20000h
   mov  finalVal,eax      ; store the result (30000h)
```

```
    call DumpRegs          ; display the registers
    exit
main ENDP
END main
```

How does it work? Firs,t the integer inside the variable **val1** is moved to EAX:

```
mov eax,val1               ; start with 10000h
```

Next, the integer inside **val2** is added to EAX:

```
add eax,val2               ; add 40000h
```

Next, the integer inside **val3** is subtracted from EAX:

```
sub eax,val3               ; subtract 20000h
```

Finally, the integer in EAX is copied into the variable **finalVal**:

```
mov finalVal,eax           ; store the result (30000h)
```

3.4.11 Declaring Uninitialized Data

The .DATA? directive can be used to declare uninitialized data. It is particularly useful for large blocks of uninitialized data because it reduces the size of a compiled program. For example, the following code is declared efficiently:

```
.data
smallArray DWORD 10 DUP(0)       ; 40 bytes
.data?
bigArray DWORD 5000 DUP(?)       ; 20000 bytes
```

The following code, on the other hand, produces a compiled program that is 20,000 bytes larger:

```
.data
smallArray DWORD 10 DUP(0)       ; 40 bytes
bigArray DWORD 5000 DUP(?)       ; 20000 bytes
```

Mixing Code and Data The assembler lets you switch back and forth between code and data in your programs. This can be convenient when you want to declare a variable that will be used only within a localized area of your program. In the following example, we create a variable named **temp** by inserting it directly within our code:

```
.code
mov eax,ebx
.data
temp DWORD ?
.code
mov temp,eax
. . .
```

Although it appears as if **temp** would interrupt the flow of executable instructions in this example, it turns out that the assembler places **temp** in the data segment along with all the other variables. The variable **temp** has *file scope,* making it visible to every statement within the same source code file.

3.4.12 Section Review

1. Create an uninitialized data declaration for a 16-bit signed integer.
2. Create an uninitialized data declaration for an 8-bit unsigned integer.
3. Create an uninitialized data declaration for an 8-bit signed integer.
4. Create an uninitialized data declaration for an 64-bit integer.
5. Which data type can hold a 32-bit signed integer?
6. Declare a 32-bit signed integer variable and initialize it with the smallest possible negative decimal value. (*Hint:* refer to integer ranges in Chapter 1.)
7. Declare an unsigned 16-bit integer variable named **wArray** that uses three initializers.
8. Declare a string variable containing the name of your favorite color. Initialize it as a null-terminated string.
9. Declare an uninitialized array of 50 unsigned doublewords named **dArray**.
10. Declare a string variable containing the word "TEST" repeated 500 times.
11. Declare an array of 20 unsigned bytes named **bArray** and initialize all elements to zero.
12. Show the order of individual bytes in memory (lowest to highest) for the following doubleword variable:

    ```
    val1 DWORD 87654321h
    ```

3.5 Symbolic Constants

A *symbolic constant* (or *symbol definition*) is created by associating an identifier (a symbol) and either an integer expression or some text. Unlike a variable definition, which reserves storage, a symbolic constant does not use any storage. Symbols are used only during the assembly of a program, so they cannot change at runtime. The following table summarizes their differences:

	Symbol	Variable
Uses storage?	no	yes
Value changes at run time?	no	yes

In the next section, we will show how to use the equal-sign directive (=) to create symbols that represent integer constants. After that, we will use the EQU and TEXTEQU directives to create symbols that represent arbitrary text.

3.5.1 Equal-Sign Directive

The *equal-sign* directive associates a symbol name with an integer expression (see Section 3.1.2). The syntax is:

```
name = expression
```

Ordinarily, *expression* is a 32-bit integer value. When a program is assembled, all occurrences of *name* are replaced by *expression* during the assembler's preprocessor step. For example, if the assembler reads the following lines,

```
COUNT = 500
mov  al,COUNT
```

it generates and assembles the following statement:

```
mov  al,500
```

Why Use Symbols? We might have skipped the COUNT symbol entirely and simply coded the MOV instruction with the literal 500, but experience has shown that programs are easier to read and maintain if symbols are used. Suppose COUNT were used ten times throughout a program. At a later time, it could be increased to 600 by altering only a single line of code:

```
 COUNT = 600
```

When the program was assembled again, all instances of the symbol COUNT would automatically be replaced by 600. Without this symbol, the programmer would have to manually find and replace every 500 with 600 in the program's source code. What if one occurrence of 500 were not actually related to all of the others? Then a bug would be caused by changing it to 600.

Keyboard Definitions Programs often define symbols for important keyboard characters. For example, 27 is the ASCII code for the Esc key:

```
Esc_key = 27
```

Later in the same program, a statement is more self-describing if it uses the symbol rather than an immediate value. Use this,

```
mov  al,Esc_key            ; good style
```

rather than this:

```
mov  al,27                 ; poor style
```

Using the DUP Operator Section 3.4.3.3 showed how to use the DUP operator to create storage for arrays and strings. It is good coding style to combine a symbolic constant with DUP because it simplifies program maintenance. In the next example, if COUNT has already been defined, it can be used in the following data definition:

```
array COUNT DUP(0)
```

Redefinitions A symbol defined with = can be redefined any number of times. The following example shows how the assembler evaluates COUNT each time it changes value:

```
COUNT = 5
mov al,COUNT              ; AL = 5
COUNT = 10
mov al,COUNT              ; AL = 10
COUNT = 100
mov al,COUNT              ; AL = 100
```

The changing value of a symbol such as COUNT has nothing to do with the runtime execution order of statements. Instead, the symbol changes value according to the sequential processing of your source code by the assembler.

3.5.2 Calculating the Sizes of Arrays and Strings

Often when using an array, we would like to know its size. In the following example, we create a constant named **ListSize** and manually count the bytes in **list**:

```
list BYTE 10,20,30,40
ListSize = 4
```

But this is not good practice if this code must be later modified and maintained. If we were to add more bytes to **list**, **ListSize** would also have to be corrected or a program bug would result. A better way to handle this situation would be to let the assembler automatically calculate **ListSize** for us. MASM uses the $ operator (*current location counter*) to return the offset associated with the current program statement. In the following example, **ListSize** is calculated by subtracting the offset of **list** from the current location counter ($):

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

It is important for **ListSize** to follow immediately after **list**. The following, for example, would produce too large a value for **ListSize** because of the storage used by **var2**:

```
list BYTE 10,20,30,40
var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

String lengths are time consuming to calculate manually, so you can let the assembler do the job for you:

```
myString  BYTE "This is a long string, containing"
          BYTE "any number of characters"
```

```
myString_len = ($ – myString)
```

Arrays of Words and DoubleWords If each element in an array contains a 16-bit word, the array's total size in bytes must be divided by 2 to produce the number of array elements:

```
list  WORD  1000h,2000h,3000h,4000h
ListSize = ($ – list) / 2
```

Similarly, each element of an array of doublewords is 4 bytes long, so its overall length must be divided by 4 to produce the number of array elements:

```
list  DWORD  10000000h,20000000h,30000000h,40000000h
ListSize = ($ – list) / 4
```

3.5.3 EQU Directive

The EQU directive associates a symbolic name with either an integer expression or some arbitrary text. There are three formats:

```
name EQU expression
name EQU symbol
name EQU <text>
```

In the first format, *expression* must be a valid integer expression (see Section 3.1.2). In the second format, *symbol* is an existing symbol name, already defined with = or EQU. In the third format, any text may appear within the brackets <. . . >. When the assembler encounters *name* later in the program, it substitutes the integer value or text for the symbol.

EQU can be useful when defining any value that does not evaluate to an integer. A real number constant, for example, can be defined using EQU:

```
PI EQU <3.1416>
```

Example We can associate a symbol with a character string. Then a variable can be created using the symbol:

```
pressKey EQU <"Press any key to continue...",0>
.
.
.data
prompt  BYTE   pressKey
```

Example Suppose we would like to define a symbol that calculates the number of cells in a 10-by-10 integer matrix. We will define symbols two different ways, first as an integer expression, and second as a text expression. The two symbols are then used in data definitions:

```
matrix1  EQU   10 * 10
matrix2  EQU  <10 * 10>
.data
M1 WORD matrix1
```

```
M2 WORD matrix2
```

The assembler will produce different data definitions for **M1** and **M2**. The integer expression in **matrix1** is evaluated and assigned to **M1**. On the other hand, the text in **matrix2** is copied directly into the data definition for **M2**:

```
M1 WORD  100
M2 WORD  10 * 10
```

No Redefinition Unlike the = directive, a symbol defined with EQU cannot be redefined in the same source code file. This may be seen as a restriction, but it also prevents an existing symbol from being inadvertently assigned a new value.

3.5.4 TEXTEQU Directive

The TEXTEQU directive, introduced in MASM Version 6, is very similar to EQU. It creates what Microsoft calls a *text macro*. There are three different formats: the first assigns text; the second assigns the contents of an existing text macro, and the third assigns a constant integer expression:

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

For example, the **prompt1** variable uses the **continueMsg** text macro:

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

Text macros can easily build on each other. In the next example, **count** is set to the value of an integer expression involving **rowSize**. Next, the symbol **move** is defined as **mov**. Then **setupAL** incorporates the values of **move** and **count**:

```
rowSize = 5
count   TEXTEQU  %(rowSize * 2)    ; same as: count TEXTEQU <10>
move    TEXTEQU  <mov>
setupAL TEXTEQU  <move al,count>
; same as: setupAL TEXTEQU <mov al,10>
```

Unlike the EQU directive, a symbol defined with TEXTEQU can be redefined later in the program.

Compatibility Note: TEXTEQU was first introduced in MASM version 6. If you're writing assembler code that must be compatible with various assemblers including earlier versions of MASM, you should use EQU rather than TEXTEQU.

3.5.5 Section Review

1. Declare a symbolic constant using the equal-sign directive that contains the ASCII code (08h) for the Backspace key.
2. Declare a symbolic constant named **SecondsInDay** using the equal-sign directive and assign it an arithmetic expression that calculates the number of seconds in a 24-hour period.
3. Show how to calculate the number of bytes in the following array and assign the value to a symbolic constant named **ArraySize**:

   ```
   myArray WORD 20 DUP(?)
   ```
4. Show how to calculate the number of elements in the following array and assign the value to a symbolic constant named **ArraySize**:

   ```
   myArray DWORD 30 DUP(?)
   ```
5. Use a TEXTEQU expression to redefine "PROC" as "PROCEDURE."
6. Use TEXTEQU to create a symbol named **Sample** for a string constant, and then use the symbol when defining a string variable named **MyString**.
7. Use TEXTEQU to assign the symbol **SetupESI** to the following line of code:

   ```
   mov esi,OFFSET myArray
   ```

3.6 Real-Address Mode Programming (*Optional*)

If you are programming for MS-DOS or for Linux’s DOS Emulation feature, you can easily code your programs as 16-bit applications to run in Real-address mode. We will assume that you are using an Intel386 or later processor. When we call this a *16-bit* application, we refer to the use of 16-bit segments, also known as *Real-address mode* segments.

3.6.1 Basic Changes

There are only a few changes you must make to the 32-bit programs presented in this chapter to transform them into 16-bit programs:

- The INCLUDE directive references a different library:

  ```
  INCLUDE Irvine16.inc
  ```
- Two additional instructions must be inserted at the beginning of the startup procedure (main). They initialize the DS register to the starting location of the data segment, identified by the predefined MASM constant **@data**:

  ```
  mov ax,@data
  mov ds,ax
  ```
- The batch file that assembles and links your programs is named **make16.bat** (we will show an example later).

- Offsets (addresses) of data and code labels are 16 bits rather than 32 bits.

You cannot move @data directly into DS and ES because the Intel instruction set does not permit a constant to be moved directly to a segment register.

3.6.1.1 The AddSub2 Program

Here is a listing of the *AddSub2.asm* Program, revised to run in Real-address mode. New lines are marked by comments:

```
TITLE Add and Subtract, Version 2   (AddSub2.asm)

; This program adds and subtracts 32-bit integers
; and stores the sum in a variable.
; Target: Real-address mode.

INCLUDE Irvine16.inc      ; changed
.data
val1     DWORD 10000h
val2     DWORD 40000h
val3     DWORD 20000h
finalVal DWORD ?

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

   mov eax,val1           ; get first value
   add eax,val2           ; add second value
   sub eax,val3           ; subtract third value
   mov finalVal,eax       ; store the result
   call DumpRegs          ; display registers

   exit
main ENDP
END main
```

3.7 Chapter Summary

An integer expression is a mathematical expression involving integer constants, symbolic constants, and arithmetic operators. Precedence refers to the implied order of operations when an expression contains two or more operators.

A character constant is a single character enclosed in either single or double quotes. The assembler converts a character to the binary ASCII code matching the character. A string constant is a string of characters enclosed in either single or double quotation marks, possibly ending with a Null character.

Assembly language has a list of reserved words, shown in Appendix D, that have special meanings and can only be used in their correct contexts. An identifier is a programmer-chosen name that can identify a variable, a symbolic constant, a procedure, or a code label. It cannot be a reserved word.

A directive is a command that is recognized and acted upon by the assembler as the program's source code is assembled. An instruction is a statement that is executed by the processor at runtime. An instruction mnemonic is a short assembler keyword that identifies the operation carried out by an instruction. A label is an identifier that acts as a place-marker for either instructions or data.

An assembly language instruction can have between zero and three operands, each of which can be a register, memory operand, constant expression, or I/O port.

Programs contain logical segments named code, data, and stack. The code segment contains executable instructions. The stack segment holds procedure parameters, local variables, and return addresses. The data segment holds variables.

A source file is a text file containing assembly language statements. A listing file contains a copy of the program's source code, suitable for printing, with line numbers, offset addresses, translated machine code, and a symbol table. A map file contains information about a program’s segments. A source file is created with a text editor. The assembler (MASM) is a program that reads the source file, producing both object and listing files. The linker reads the object file and produces an executable file. The latter can be executed by the operating system.

MASM recognizes intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type:

- BYTE and SBYTE define 8-bit variables.
- WORD and SWORD define 16-bit variables.
- DWORD and SDWORD define32-bit variables.
- QWORD and TBYTE define 8-byte and 10-byte variables, respectively.
- REAL4, REAL8, and REAL10 define 4-byte, 8-byte, and 10-byte real number variables, respectively.

A data definition statement sets aside storage in memory for a variable and may optionally assign a name to the variable. If multiple initializers are used in the same data definition, its label refers only to the offset of the first byte. To create a string data definition, enclose a sequence of characters in quotation marks. The DUP operator generates a repeated storage allocation, using a constant expression as a counter. The current location counter operator ($) can be used in an expression that calculates the number of bytes in an array.

Intel processors store and retrieve data from memory using little endian order. This means that the least significant byte of a variable is stored at the lowest memory address.

A symbolic constant (or symbol definition) is created by associating an identifier (a symbol) with an integer or text expression. There are three directives that create symbolic constants:

- The equal-sign directive associates a symbol name with an integer expression.
- The EQU and TEXTEQU directives associate a symbolic name with either an integer expression or some arbitrary text.

It is easy to switch between writing 32-bit Protected mode and 16-bit Real mode programs, if you keep in mind a few differences. The book is supplied with two link libraries containing the same procedure names for both types of programs.

3.8 Programming Exercises

The following exercises can be done in either Protected mode or Real-address mode.

1. Subtracting Three Integers

Using the **AddSub** program from Section 3.2 as a reference, write a program that subtracts three 16-bit integers using only registers. Insert a **call DumpRegs** statement to display the register values.

2. Data Definitions

Write a program that contains a definition of each data type listed in Section 3.4. Initialize each variable to a value that is consistent with its data type.

3. Symbolic Integer Constants

Write a program that defines symbolic constants for all of the days of the week. Create an array variable that uses the symbols as initializers.

4. Symbolic Text Constants

Write a program that defines symbolic names for several string literals (characters between quotes). Use each symbolic name in a variable definition.