Chapter 3

Assembly Language Fundamentals

Chapter 3

Assembly Language Fundamentals

Objectives

After reading this Chapter, you should be able to understand or do each of the following:

- Know how to represent integer constants, expressions, real number constants, character constants, and string constants in assembly language
- Know how to formulate assembly language instructions, using valid syntax
- Understand the difference between instructions and directives
- Be able to code, assemble, and execute a program that adds and subtracts integers
- Be able to create variables using all standard assembly language data types
- Be able to define symbolic constants
- Be able to calculate the size of arrays at assembly time

- Syntax:

  [{+ | -}] *digits* [*radix*]

- Microsoft syntax notation is used throughout this chapter
  - Elements within square brackets [ ] are **optional**
  - Elements within { …| …|…} requires **a choice** of the enclosed elements
  - Elements in italics denote items which have known definitions or descriptions
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - d – decimal
  - b – binary
  - r – encoded real
    - Examples:

      30**d**, 6A**h**, 42, 1101**b**

    - Hexadecimal beginning with letter must have **leading 0**: 0A5h
    - If no radix is given, the integer constant is **assumed** to be decimal

3.1.2 Integer Expressions 52

- An integer expression is a mathematical expression involving integer value and arithmetic operators.
- Operators and **precedence** levels:

TABLE 3-1 Arithmetic Operators (Precedence).

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

- Examples:

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

3.1.3 Real Number Constants 53

- Syntax:

  [*sign*] integer.[integer][*exponent*]
  sign {+ | -}
  exponent E[{+ | -}]integer

- Examples:

  2., +3.0, -44.26E+05, 26.E-5

3.1.4 Character Constants 54

- Enclose *character* in **single or double** quotes
  - ASCII character = 1 byte
- Examples:

  'A', "x"

3.1.5 String Constants 54

- Enclose strings in **single or double** quotes
  - Each character occupies a single byte
- Examples:

  'xyz', "ABC"

- Embedded quotes: 'Say "Goodnight," Gracie'

3.1.6 Reserved Words 54

- Reserved words have special meaning in MASM and can only be used in their context.
- There are different types of **reserved words**:
  - **Instruction mnemonics**: such as MOV, ADD, and MUL
  - **Directives**: Tell MSAM how assemble programs, such as .DATA and .CODE
  - **Attributes**: Provide size and usage information for variables and operands, such as BYTE and WORD
  - **Operators**: used in constant expressions, such as 10 * 10
  - **Predefined symbols**: such as @data, which return constant integer values at assembly time.
- *Reserved words* **cannot** be used as identifiers
- See MASM reference in Appendix A (**Page 600**)

3.1.7 Identifiers 54

- Identifiers – a programmer-choice name
  - 1-**247** characters, including digits
  - **not** case sensitive
  - The first character must be a letter (A..Z, a..z), underscore (_), @, ?, or $. Subsequent character may also be digits.
  - An identifier cannot be the same as an assembler reserved word.
- Examples:

  var1, Count, $first, _main, MAX, open_file, xVal

3.1.8 Directives 55

- Commands that are recognized and acted upon by the assembler
  - **Not** part of the Intel instruction set
    - Directives do not execute at run time, whereas instructions do.
    - Example

```
myVar   DWORD   26          ; DWORD directive
move    ax,     myVar       ; MOV instruction
```

  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - **not** case sensitive: It recognizes .data, .DATA, and .Data as equivalent.
- Defining Segments:
  - One important function of assembler directives is to define program section, or segments.
  - The .DATA directive identifies the area of a program containing variables:
    .data
  - The .CODE directive identifies the area of a program containing instructions:
    .code
  - The .STACK directive identifies the area of a program holding the runtime stack, setting its size:
    .stack 1000h
- Different assemblers have different directives
  - NASM not the same as MASM
  - See MASM Directives in Appendix A.5 (**Page 604**)

3.1.9 Instructions 55

- An instruction is a statement that becomes executable when a program is assembled.
- Instructions are translated by the assembler into machine language bytes, which are loaded and **executed by the CPU at run time**.
- We use the Intel IA-32 instruction set
- Syntax:

  [*label*] *mnemonic operand(s)* [;*comment*]

label	optional
instruction mnemonic	required: such as MOV, ADD, SUB, MUL
operands	usually required
comment	optional

- An instruction contains:
  - Labels (optional)
    - Act as place markers
    - marks the address (offset) of code and data
    - Follow identifer rules
    - Data label
      - must be unique
      - example: **count** (**not followed by colon**)

```
count   DWORD   100
```

    - Code label
      - target of jump and loop instructions
      - example: **target:** (**followed by colon**)

```
target:
   MOV   ax,   bx
   …
   JMP   target
```

  - Mnemonics (required)
    - Instruction Mnemonics
    - memory aid
    - examples: MOV, ADD, SUB, MUL, 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

- Operands (depends on the instruction)
  - Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, constant expression, or I/O port.
    - constant (immediate value): ex. **96**
    - constant expression: ex. **10 * 10**
    - register: ex. **eax**
    - memory (data label): ex. **count**
  - Examples of assembly language instructions having varying numbers of operands
    - No operands

```
stc                 ; set Carry flag
```

    - One operand

```
inc eax             ; register
inc myByte          ; memory
```

    - Two operands

```
add ebx, ecx        ; register, register
sub myByte,  25     ; memory, constant
add eax,36 * 25     ; register, constant-expression
```

- Comments (optional)
  - Comments can be specified in two ways: single-line and block comments
  - Single-line comments
    - Begin with semicolon (;)
  - Multi-line comments
    - Begin with COMMENT directive and a programmer-chosen character
    - End with the same programmer-chosen character
    - Example:

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

We can also use any other symbol:

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

3.1.10 The NOP (No Operations) Instruction 57

- The safest instruction you can write is called NOP (no operation).
- It takes up **1 byte** of program storage and does not do any work.
- I t is sometimes used by compilers and assemblers to align code to even-address boundaries.
- Example:
  - In the following example, the NOP instruction aligns the address of third instruction to a double word boundary (even multiple of 4).

```
0000 0000    66    8B    C3    mov  ax,   bx
0000 0003    90                nop             ; align next instruction
0000 0004    8B    D1          mov  edx,  ecx
```

  - IA-32 processors are designed to load code and data **more quickly** from even double word address.

3.2 Example: Adding Three Integers 58

- Program listing

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

; This program adds and subtracts 32-bit integers.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

.code
main PROC

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

   exit
main ENDP
END main
```

- Program Output: showing registers and flags

```
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
```

- Program Description
  - The **TITLE** directive marks the entire line as a comment
  - The **INCLUDE** directive copies necessary definitions and setup information from a test file (**Irvine32.inc**) located in assembler's INCLUDE directory
  - The **.code** directive marks the beginning of the *code segment*
  - The **PROC** directive identifies the **beginning** of a procedure
  - The **MOVE** instruction moves (copies) the second operand (*source operand*) to the first operand (*destination operator*)
  - The **ADD** instruction add second operand to the first operand
  - The **SUB** instruction subtracts second operand from the from operand
  - The **CALL** statement calls a procedure. **DumpRegs**: Irvine32 procedure
  - The **exit** statement calls a predefined **MS-Window function** that halts the program
  - The **ENDP** directive marks the **end** of the procedure
  - 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.) Procedure **main** is the **startup** procedure.
- Segments – organize the program
  - The code segment **(.code**) contains all of the program's executable **instruction**
  - The data segment (**.data**) holds **variable**
  - The stack (**.stack**) holds procedure **parameters** and **local variables**

- Suggested Coding Standards
  - This approach is used in **this book**, except that lowercase is used for the .code, .stack, .mode, and .data directives.
    - Capitalize only directives and operators
    - Use mixed case for identifiers
    - Lower case everything else

3.2.1 Alternative Version of AddSub 60

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

; This program adds and subtracts 32-bit integers.
; 32-bit Protected mode version
; Last update: 06/01/2006

.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
```

- The .386 directive identifies the minimum CPU required for this program (**Intel386**).
- The .MODEL directive instructs the assembler to generate code for a **protected mode** program, and STDCALL enables the calling of **MS-Windows functions**.
- Two **PROTO** directives declare prototypes for 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 is an assembler directive that calls a procedure or function.
  - This program ends by calling the ExitProcess function, passing it a return code of **zero**.

3.2.2 Program Template 61

- ***Program Template***

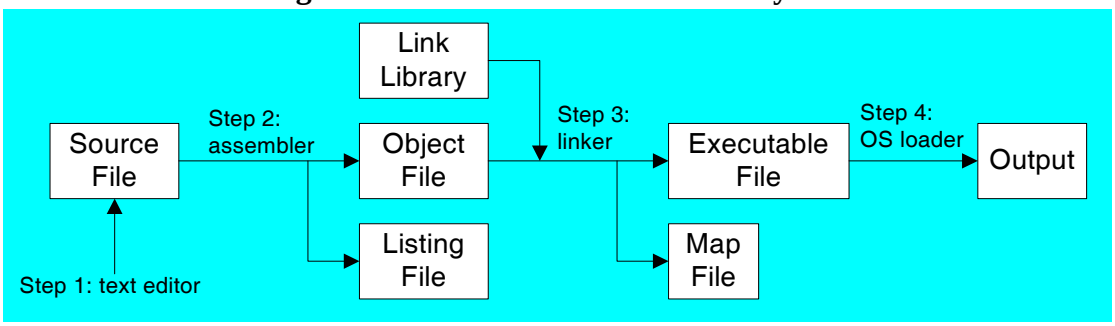
```
TITLE Program Template       (template.asm)

; Program Description:
; Author:
; Date Created:
; Last Modification Date:

INCLUDE Irvine32.inc

; (insert symbol definitions here)

.data

; (insert variables here)

.code
main PROC

; (insert executable instructions here)

  exit   ; exit to operating system
main ENDP

; (insert additional procedures here)

END main
```

3.3 Assembling, Linking, and Running Programs 62

3.3.1 The Assemble-Link-Execute Cycle 62

- Assemble-Link Execute Cycle
  - The following diagram describes the steps from creating a source program through executing the compiled program.
  - If the source code is modified, Steps 2 through 4 must be repeated.

Figure 3-1 Assemble-Link-Execute Cycle



- Listing File
  - Use it to see how your program is compiled
  - Contains
    - source code
    - addresses
    - object code (machine language)
    - segment names
    - symbols (variables, procedures, and constants)
  - Example: **addSub.lst**
- Map File
  - Information about each program segment:
    - starting address
    - ending address
    - size
    - segment type
  - Example: addSub.map (16-bit version, not generated in 32-bit version)

3.4 Defining Data 64

3.4.1 Intrinsic Data Types 64

- Intrinsic Data Types
  - BYTE, SBYTE
    - 8-bit unsigned integer; 8-bit signed integer
  - WORD, SWORD
    16-bit unsigned & signed integer
  - DWORD, SDWORD
    - 32-bit unsigned & signed integer
  - QWORD
    - 64-bit integer
  - TBYTE
    - 80-bit integer
  - REAL4
    - 4-byte IEEE short real
  - REAL8
    - 8-byte IEEE long real
  - REAL10
    - 10-byte IEEE extended real

3.4.2 Data Definition Statement 64

- Data Definition Statement
  - A data definition statement sets aside storage in memory for a variable.
  - May optionally assign a name (label) to the data
  - Syntax:

    [*name*] *directive initializer* [,*initializer*] . . .

  - Example:

    ```
    value1 BYTE 10
    ```

  - All initializers become binary data in memory

3.4.3 Defining BYTE and SBYTE Data 66

- Defining BYTE and SBYTE Data

```
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
value6 BYTE ?        ; uninitialized byte
```

- Defining Byte Arrays
  - Examples: use multiple initializers

```
list1  BYTE 10, 20, 30, 40
```

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

```
list2  BYTE 10, 20, 30, 40
       BYTE 50, 60, 70, 80
       BYTE 81, 82, 83, 84
list3  BYTE ?, 32, 41h, 00100010b
list4  BYTE 0Ah, 20h,'A', 22h
```

- Defining Strings
  - A string is implemented as **an array of characters**
  - For convenience, it is usually enclosed in quotation marks
  - It often will be **null-terminated** **(containing 0).** Strings of this type are used in **C, C++, and Java programs.**
  - Examples:

```
str1 BYTE "Enter your name", 0
str2 BYTE 'Error: halting program', 0
str3 BYTE 'A','E','I','O','U'
greeting   BYTE "Welcome to the Encryption Demo program "
           BYTE "created by Kip Irvine.", 0
```

- To continue a single string across multiple lines, end each line with a **comma**:

```
menu BYTE "Checking Account", 0dh, 0ah, 0dh, 0ah,
          "1. Create a new account", 0dh, 0ah,
          "2. Open an existing account", 0dh, 0ah,
          "3. Credit the account", 0dh, 0ah,
          "4. Debit the account", 0dh, 0ah,
          "5. Exit", 0ah, 0ah,
          "Choice> ", 0
```

  - End-of-line character sequence:
    - **0Dh** = carriage return
    - **0Ah** = line feed

```
str1     BYTE "Enter your name:    ", 0Dh, 0Ah
         BYTE "Enter your address: ", 0

newLine  BYTE 0Dh,0Ah, 0
```

- Using the DUP Operator
  - Use **DUP** to allocate (create space for) an array or string.
  - Syntax:

    counter DUP (*argument*)

  - Counter and argument must be constants or constant expressions
  - Examples:

```
var1 BYTE 20 DUP(0)        ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)        ; 20 bytes, uninitialized
var3 BYTE 4 DUP("STACK")   ; 20 bytes: "STACKSTACKSTACKSTACK"
```

3.4.4 Defining WORD and SWORD Data 67

- Defining WORD and SWORD Data
  - Define storage for 16-bit integers, single value or multiple values

```
word1  WORD  65535       ; largest unsigned value
word2  SWORD -32768      ; smallest signed value
word3  WORD  ?           ; uninitialized, unsigned
word4  WORD  "AB"        ; double characters
myList WORD  1,2,3,4,5   ; array of words
array  WORD  5 DUP(?)    ; uninitialized array
```

3.4.5 Defining DWORD and SDWORD Data 68

- Defining DWORD and SDWORD Data
  - Storage definitions for signed and unsigned 32-bit integers

```
val1 DWORD  12345678h  ; unsigned
val2 SDWORD -2147483648  ; signed
val3 DWORD  20 DUP(?)  ; unsigned array
val4 SDWORD -3,-2,-1,0,1 ; signed array
```

3.4.6-8 Defining QWORD, TBYTE, Real Number Data 69

- Defining QWORD, TBYTE, Real Data
  - Storage definitions for quadwords, tenbyte values, and real numbers

```
quad1 QWORD  1234567812345678h
val1  TBYTE  100000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

3.4.9 Little Endian Order 69

- Little Endian Order
  - All data types larger than a byte store their individual bytes in reverse order
  - The **least** significant byte occurs at the first (**lowest**) memory address
  - Example:

val1 DWORD 12345678h

Offset	Value
0000:	78
0001:	56
0002:	34
0003:	12

- Big Endian Order

val1 DWORD 12345678h

Offset	Value
0000:	12
0001:	34
0002:	56
0003:	78

3.4.10 Adding Variables to the AddSub Program 70

- Adding Variables to AddSub

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

; This program adds and subtracts 32-bit integers
; and stores the sum in a variable.
; Last update: 06/01/2006

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
```

3.5 Symbolic Constants 72

- Associate and identifier (a symbol) with an integer expression or some text
  - Symbols **do not reserve storage**
  - Used only by the assembler when scanning a program
  - **Cannot change at run time**

3.5.1 Equal-Sign Directive 72

- Equal-Sign Directive
  - *Syntax*

    name = *expression*

    - *expression* is a 32-bit integer (expression or constant)
    - may be redefined
    - *name* is called a **symbolic constant**

  - good programming style to use symbols

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

3.5.2 Calculating the Sizes of Arrays and Strings 73

- Calculating the Size of a Byte Array
  - **Current location counter**: **$**
    - Subtract address of list
    - Difference is the number of bytes
    - Example:

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

    - Note: `ListSize` must follow immediately after `List`

- Calculating the Size of a Word Array
  - Divide total number of bytes by **2** (the size of a word)

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

- Calculating the Size of a Doubleword Array
  - Divide total number of bytes by 4 (the size of a doubleword)

```
list DWORD 1,2,3,4
ListSize = ($ - list) / 4
```

3.5.3 EQU Directive 74

- EQU Directive
  - Define a symbol as either an **integer** or **text** expression.
  - Cannot be redefined
  - *Syntax*

```
name EQU expression      ; integer expression
name EQU symbol          ; existing symbol name
name EQU <text>          ; any text
```

  - Example

```
matrix EQU 10 * 10
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.data
prompt    BYTE pressKey
MI        WORD matrix
```

3.5.4 TEXTEQU Directive 74

- TEXTEQU Directive
  - Define a symbol as either an integer or text expression.
  - Called a ***text macro***
  - Can be redefined

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)    ; evaluates the expression
setupAL TEXTEQU <mov al,count>

.code
setupAL             ; generates: "mov al,10"
```

3.6 Real-Address Mode Programming (Optional) 75

- Generate **16-bit** MS-DOS Programs
- Advantages
  - **enables calling of MS-DOS and BIOS functions**
  - **no memory access restrictions**
- Disadvantages
  - must be aware of both segments and offsets
  - cannot call Win32 functions (Windows 95 onward)
  - limited to 640K program memory

3.6.1 Basic Changes 75

- Requirements
  - INCLUDE **Irvine16.inc**
  - Initialize DS to the data segment:

    mov ax, @data
    mov ds, ax
    - Note: MOV instruction does not permit a constant to be moved directly to a segment register.
- Add and Subtract, **16-Bit** Version

```
TITLE Add and Subtract, Version 2         (AddSub2r.asm)

; This program adds and subtracts 32-bit integers
; and stores the sum in a variable. (From page 94.)
; Last update: 06/01/2006

INCLUDE Irvine16.inc    ; new

.data
val1     dword  10000h
val2     dword  40000h
val3     dword  20000h
finalVal dword  ?

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

   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
```

3.7 Chapter Summary 76

- Character and Strings
  - A **character** constant is a single character enclosed in **quotes**. The assembler converts a character to a byte containing the character's binary ASCII code.
  - A **string** constant is a sequence of characters enclosed in quotes, optionally ending with a null byte.
- An **identifier** is a programmer-chosen name identifying a variable, a symbolic constant, a procedure, or a code label.
- Assembly language has a set of **reserved words** with special meanings that may only be used in the correct context.
  - **Instruction mnemonics**: An **instruction** is a source code statement that is executed by the processor at run time. An **instruction mnemonic** is a short keyword that identifies the operation carried out by an instruction.
  - **Directives**: A **directive** is a command embedded in the source code and **interpreted** by the assembler.
  - **Attributes**: Provide size and usage information for variables and operands.
  - **Operators**: used in constant expressions, such as 10 * 10
  - **Predefined symbols**: such as @data, which return constant integer values at assembly time.
- 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.
- Assembler, Linker, and Loader
  - An **assembler** is a program that reads the source file, producing both object and listing files.
  - The **linker** is a program that reads one or more object files and produces an executable file.
  - The latter is executed by the operating system **loader**.
- Data definition directives:
  - BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, QWORD, TBYTE, REAL4, REAL8, and REAL10
  - The **DUP** operator generates a repeated storage allocation, using a constant expression as a counter.
  - The current location counter operator (**$**) is used in address-calculation expression.
- **Intel** processors store and retrieve data from memory using **little endian** order: The least significant by of a variable is stored at its starting address.
- Symbolic constant
  - The equal-sign directive (=) associates a symbol name with an integer expression.
  - The EQU and TEXTEQU directives associate a symbolic name with an integer expression or some arbitrary text.