Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Chapter 3: Assembly Language Fundamentals

Chapter Overview

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- Real-Address Mode Programming

Basic Elements of Assembly Language

- Integer constants
- Integer expressions
- Real-number constants
- Character and string constants
- Reserved words and identifiers
- Directives
- Instructions
  - Labels
  - Mnemonics
  - Operands
  - Comments
- Examples

Integer Constants

- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - d – decimal
  - b – binary
  - r- encoded real

Examples: 30d, 6Ah, 42, 1101b

Hexadecimal beginning with letter: 0A5h

- To prevent the assembler from interpreting it as an identifier

Integer Expressions

- Operators and precedence levels:

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

Real Number Constants

- Decimal Real
  - [sign] integer.[integer [exponent]]
  - E.g., 2., +3.0, -44.2E+05, 26.E5
- Encoded Real
  - Specify a real constant in hexadecimal as an encoded real if you know the exact binary representation of the number
  - E.g., 3F800000r -> +1.0

Character and String Constants

- Enclose character in single or double quotes
  - 'A', "x"
  - ASCII character = 1 byte
- Enclose strings in single or double quotes
  - "ABC"
  - 'xyz'
  - Each character occupies a single byte
- Embedded quotes:
  - 'Say "Goodnight," Gracie'

Reserved Words and Identifiers

- Reserved words (Appendix D) cannot be used as identifiers
  - Instruction mnemonics (ADD, MOV …), directives (tell MASM how to assemble programs), type attributes (BYTE, WORD …), operators (+, - …), predefined symbols (@data …)
- Identifiers
  - 1-247 characters, including digits
  - case insensitive (by default)
  - first character must be a letter, _, @, or $

Directives

- Directives
  - not part of Intel instruction set
  - part of the assembler's syntax
  - case insensitive
  - .data, .code, proc
- Commands that are recognized and acted upon by the assembler as the program's source code is being assembled
  - Used to declare code, data areas, select memory model, declare procedures, etc.
- Different assemblers have different directives.

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Member of the Intel IA-32 instruction set
- Parts
  - Label
  - Mnemonic
  - Operand
  - Comment

```
Label:     Mnemonic   Operand(s)    ; Comment
```

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifier rules
- Data label
  - example: first BYTE 10
- Code label
  - target of jump and loop instructions
  - example: L1:

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

Mnemonics and Operands

- Instruction Mnemonics
  - "reminder"
  - examples: MOV, ADD, SUB, MUL, INC, DEC
- Operands
  - constant (immediate value)
  - constant expression
  - register
  - memory (data label)

Comments

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- 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

```
COMMENT  !
        comment.
        also comment.
!
```

Instruction Format Examples

- 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, expression

Example: Adding and Subtracting Integers

```
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                        ; call a predefined MS-Windows
                                ; function that halts the program
                                ; in Irvine32.inc

main ENDP
END main
```

Example Output

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
```

Suggested Coding Standards [1/2]

- Some approaches to capitalization
  - capitalize nothing
  - capitalize everything
  - capitalize all reserved words, including instruction mnemonics and register names
  - capitalize only directives and operators
- Other suggestions
  - descriptive identifier names
  - spaces surrounding arithmetic operators
  - blank lines between procedures

Suggested Coding Standards [2/2]

- Indentation and spacing
  - code and data labels – no indentation
  - executable instructions – indent 4-5 spaces
  - comments: begin at column 40-45, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: mov ax,bx
  - 1-2 blank lines between procedures

Alternative Version of AddSub

```
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
```

Program Template

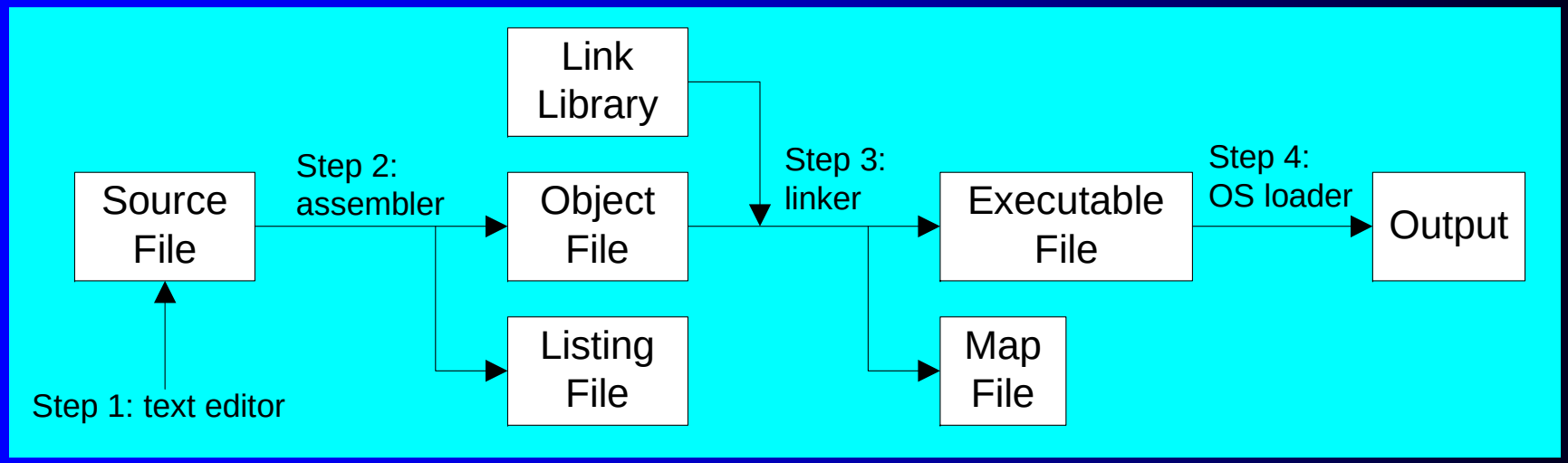
```
TITLE Program Template                    (Template.asm)

; 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
```

Assembling, Linking, and Running Programs

- Assemble-Link-Execute Cycle
- make32.bat
- Listing File
- Map File

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.



make32.bat

- Called a batch file
- Run it to assemble and link programs
- Contains a command that executes ML.EXE (the Microsoft Assembler)
- Contains a command that executes LINK32.EXE (the 32-bit Microsoft Linker)
- Command-Line syntax:

  make32 *progName*

  (*progName* includes the .asm extension)

(use make16.bat to assemble and link Real-mode programs)

Listing and Map Files

- Listing File
  - Use it to see how your program is compiled
  - Contains
    - source code
    - offset addresses
    - object code (machine language)
    - segment names
    - symbols (variables, procedures, and constants)
- Map File
  - Information about each program segment:
    - starting address
    - ending address
    - size
    - segment type

Data Types [1/2]

- 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

Data Types [2/2]

- REAL4
  - 4-byte IEEE short real
- REAL8
  - 8-byte IEEE long real
- REAL10
  - 10-byte IEEE extended real

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*] . . .

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

```
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
```

A variable name is a data label that implies an offset (an address).

Defining Bytes

Examples that use multiple initializers:

```
list1 BYTE 10,20,30,40
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 [1/2]

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It usually has a null byte at the end
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting1 BYTE "Welcome to the Encryption Demo program "
          BYTE "created by Kip Irvine.",0
greeting2 \
    BYTE "Welcome to the Encryption Demo program "
    BYTE "created by Kip Irvine.",0
```

Defining Strings [2/2]

- 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.
- Counter must be constants or constant expressions

```
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"
var4 BYTE 10,3 DUP(0),20
```

Defining WORD and SWORD Data

- Define storage for 16-bit integers
  - or double characters
  - single value or multiple values

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

```
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
```

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
```

Defining QWORD, TBYTE, Real Data

Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  1234567812345678h
val1  TBYTE  1000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

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
```

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

Adding Variables to AddSub

```
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
```

Mixing Code and Data

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

Symbolic Constants

- Associating an identifier (a symbol) and either an integer expression or some text
- A symbolic constant does not use any storage (can not change at runtime)
- Equal-Sign Directive
- Calculating the Sizes of Arrays and Strings
- EQU Directive
- TEXTEQU Directive

Equal-Sign Directive

- *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
```

Calculating the Size of a Byte Array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes

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

Calculating the Size of a Word Array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes
  - divide by 2 (the size of a word)

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

Calculating the Size of a Doubleword Array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes
  - divide by 4 (the size of a doubleword)

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

EQU Directive

- Define a symbol as either an integer or text expression
- Cannot be redefined

```
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.data
prompt BYTE pressKey
```

```
matrix1  EQU  10*10
matrix2  EQU  <10*10>
.data
M1 WORD  matrix1
M2 WORD  matrix2
```

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
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
```

||

setupAL TEXTEQU <mov al, 10>

Real-Address Mode Programming

- Generate 16-bit MS-DOS Programs
- Advantages
  - enables calling of MS-DOS
  - 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
- Requirements
  - INCLUDE Irvine16.inc
  - Initialize DS to the data segment:

```
mov ax,@data
mov ds,ax            ; why not “mov ds,$data”?
```

Add and Subtract, 16-Bit Version

```
TITLE Add and Subtract, Version 2      (AddSub2.asm)
INCLUDE Irvine16.inc
.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
```