Assembly Language
組合語言-資訊工程二年級
Lecture slides(2018 – 2019)

Fu Jen Catholic University, Dept of Computer Science and Information Engineering –CSIE

資訊工程系-輔仁大學

教授: 周賜福

107學年度第一學期

Assembly Language for x86 Processors
7th Edition

Kip Irvine

Chapter 3: Assembly Language Fundamentals

Slides prepared by the author

Revision date: 2/15/2015

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
- 64 Bit Programming (on the 7th Edition)

Basic Elements of Assembly Language

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

Basic Elements of Assembly Language

It has been shown on the Second week how to write a simple assembly language program to add two integers.

3.1.3 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

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

3.1.5 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'**

3.1.7 Reserved Words and Identifiers

- Reserved words cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
  - See MASM reference in Appendix A
  - Ex: MOV, ADD, and MUL
- Identifiers
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a letter, _, @, ?, or $
  - Ex: var1 _main $first open_file
  - xVa1 _12345
  - Try to avoid @ and _ (underscore) as leading characters, since they are used in the High level lang.

3.1.9 Directives

- **Commands that are recognized and acted upon by the assembler**
  - **Not part of the Intel instruction set**
  - **Used to declare code, data areas, select memory model, declare procedures, etc.**
  - **not case sensitive**
- **Different assemblers have different directives**
  - **NASM not the same as MASM, for example**

Directives

MASM directives are as follows:

For example:

.data .DATA .Data as equivalent

3.1.10 Instructions

- **Assembled into machine code by assembler**
- **Executed at runtime by the CPU**
- **We use the Intel IA-32 instruction set**
- **An instruction contains:**
  - **Label (optional)**
  - **Mnemonic (required)**
  - **Operand (depends on the instruction)**
  - **Comment (optional)**

Ex:

```
Label1:   Mov  eax, 5  ; Move 5 to register eax
```

Example Assembly program

```
TITLE Chapter 4 Exercise 4              (ch03_04.asm)

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

** For best appearance, set your editor's Tab indent size to
5 **
!

INCLUDE Irvine32.inc
```

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifier rules
- Data label
  - must be unique
  - example: myArray (not followed by colon)
- Code label
  - target of jump and loop instructions
  - example: L1: (followed by colon)

Mnemonics and Operands

- **Instruction Mnemonics**
  - **memory aid**
  - **examples: MOV, ADD, SUB, MUL, INC, DEC**
- **Operands**
  - **constant**
  - **constant expression**
  - **register**
  - **memory (data label)**

Constants and constant expressions are often called immediate values

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

Comments as follows

- **Single line comment**
- **; I am writing my first program**
- **Block Comment or multiple line comment**
  - **COMMENT !**
    - **This is my second program and so I want to get 100**
    - **This is an excellent program that I tried to practice.**

    !
- **We can also use any other symbol:**

  COMMENT &

  I am glad that I am learning Assembly language program to understand the computer Hardware better.

  &

Instruction Format Examples

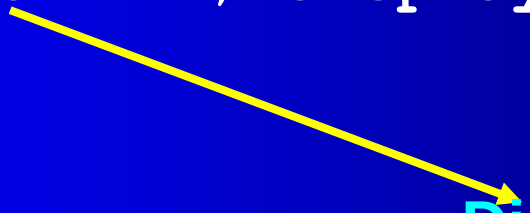
- **No operands**
  - **stc ; set Carry flag**
- **One operand**
  - **inc eax ; register**
  - **inc myByte ; memory**
- **Two operands**
  - **add ebx,ecx ; register, register**
  - **sub myByte,25 ; memory, constant**
  - **add eax,36 * 25 ; register, constant-expression**

What's Next

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

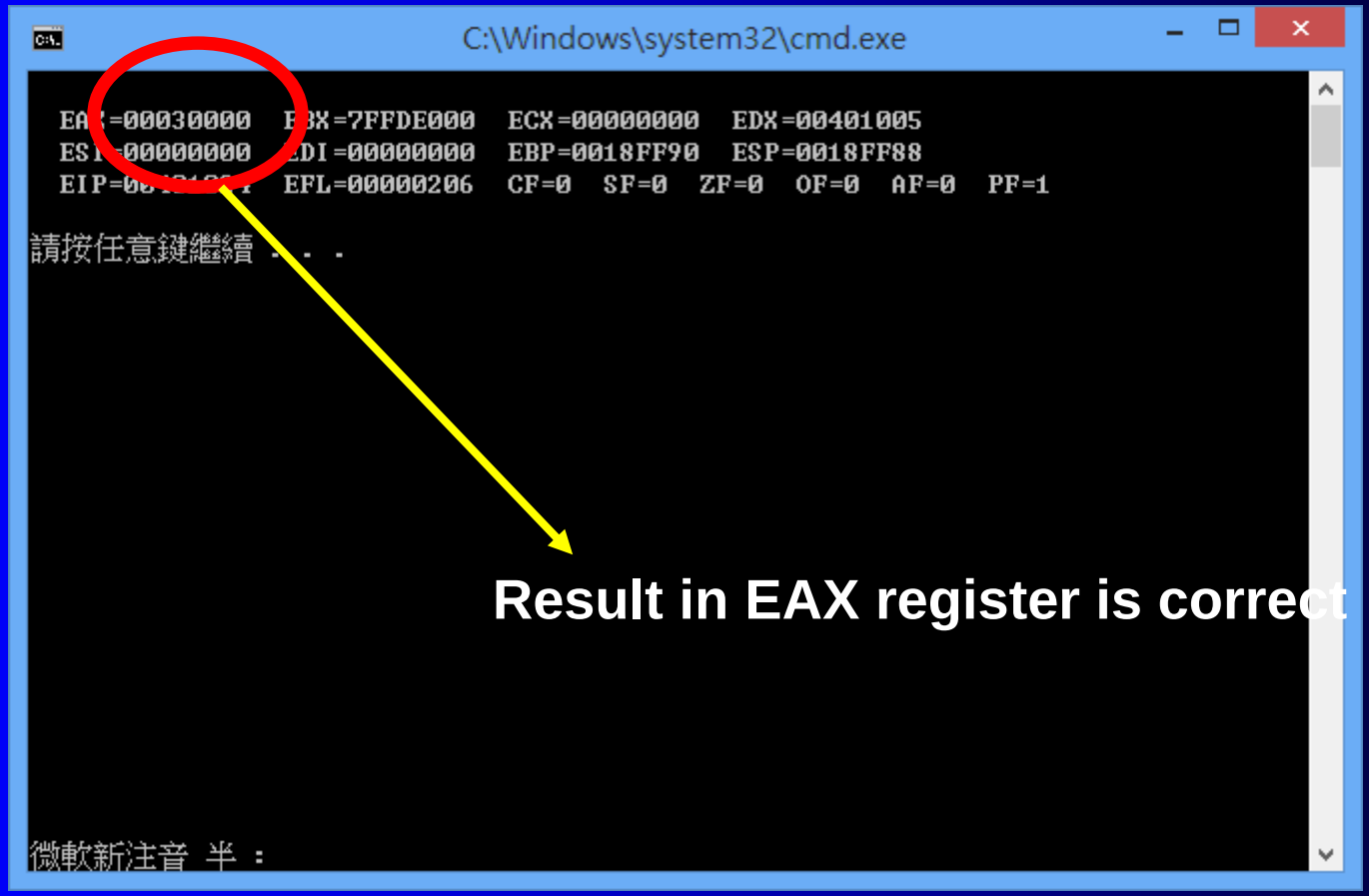
Example: Adding and Subtracting Integers

```
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
main ENDP
END main
```



Display what is in the Register?

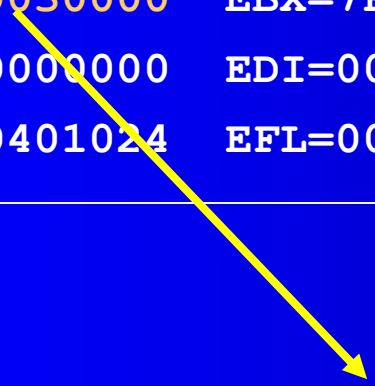
Run of the program on the previous slide



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
```



Result in EAX register is correct

Suggested Coding Standards (1 of 2)

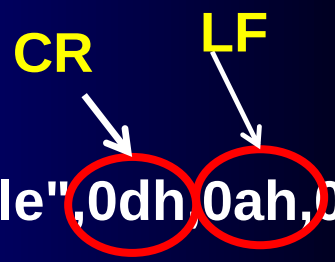
- Some approaches to capitalization
  - capitalize nothing
  - capitalize everything
  - capitalize all reserved words, including instruction mnemonics and register names
  - capitalize only directives and operators
- Other suggestions
  - descriptive identifier names
  - spaces surrounding arithmetic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

Suggested Coding Standards (2 of 2)

- **Indentation and spacing**
  - **code and data labels – no indentation**
  - **executable instructions – indent 4-5 spaces**
  - **comments: right side of page, aligned vertically**
  - **1-3 spaces between instruction and its operands**
    - **ex: mov ax, bx**
  - **1-2 blank lines between procedures**

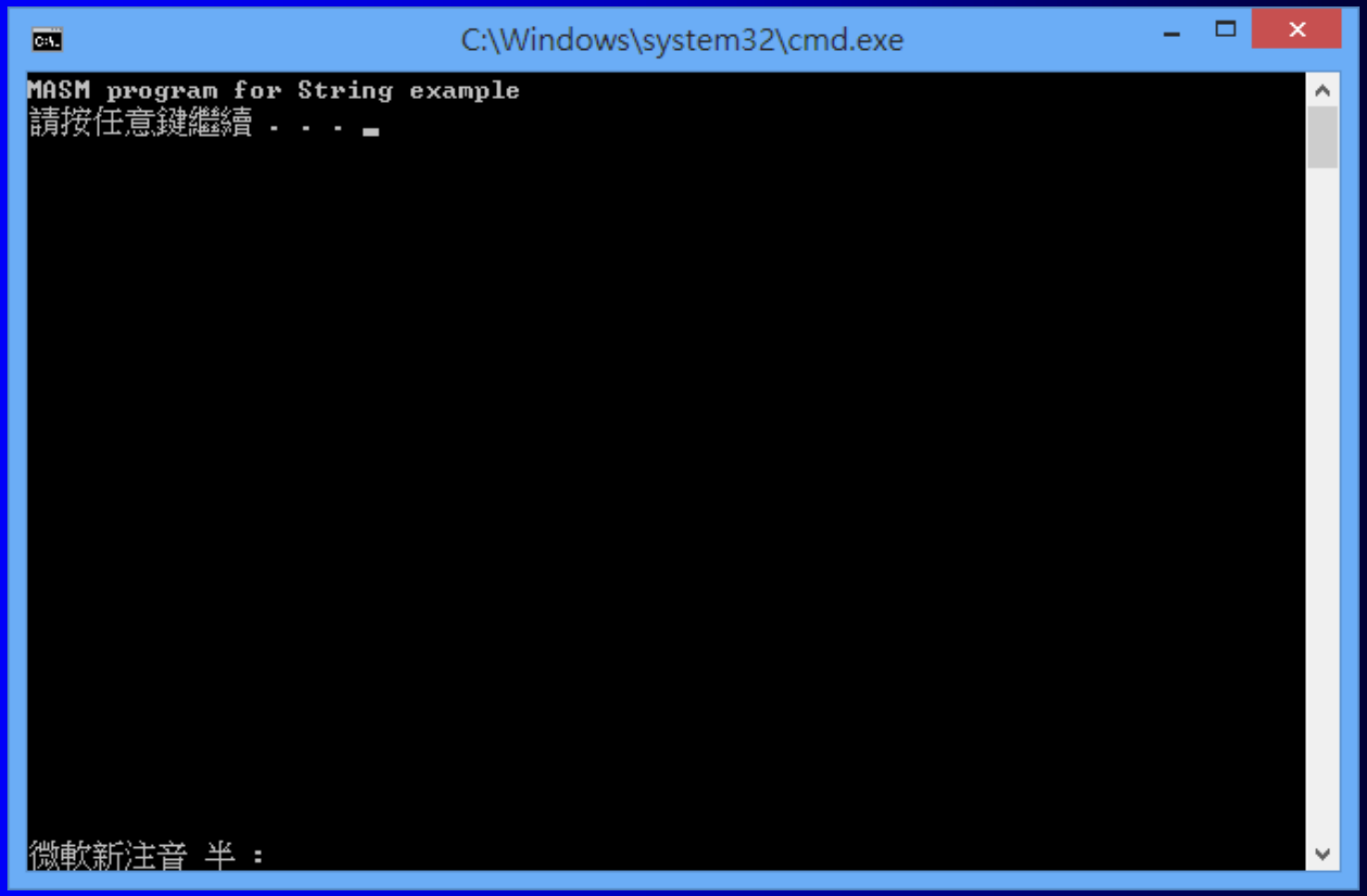
Required Coding Standards

```
TITLE MASM Template(main.asm)
; Description: This is my first program
; Name: 鄭玉鎮
; Student no:   098473743
; Original Date:  Oct 1st, 2013, Fu Jen Catholic University
; Modified Date:  Sept 2016
INCLUDE Irvine32.inc
.data
myMessage BYTE "MASM program for String example",0dh,0ah,0
.code
        main PROC
        call Clrscr
        mov edx,OFFSET myMessage
        call WriteString
        exit
main ENDP
END main
```



EDX register is used to store the pointer of the string before we start printing the string.

Run of the program in the previous page



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
```

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
```

Two PROTO directives declare prototypes for procedures used by this program: ExitProcess and DumpRegs

Model directive is used in this example:

1. In order to identify the segmentation model used by the program and it identifies the convention used for passing parameters to procedures.
2. Flat keyword tells the assembler to generate code for a protected mode program, and stdcall keyword enables the calling of MS-windows functions.

Identifies the minimum CPU required for this program x86

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
```

An instruction contains:

- Label (optional)
- Mnemonic (required)
- Operand (depends on the instruction)
- Comment (optional)

We have completed the basic elements of an Assembly Language

What's Next

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

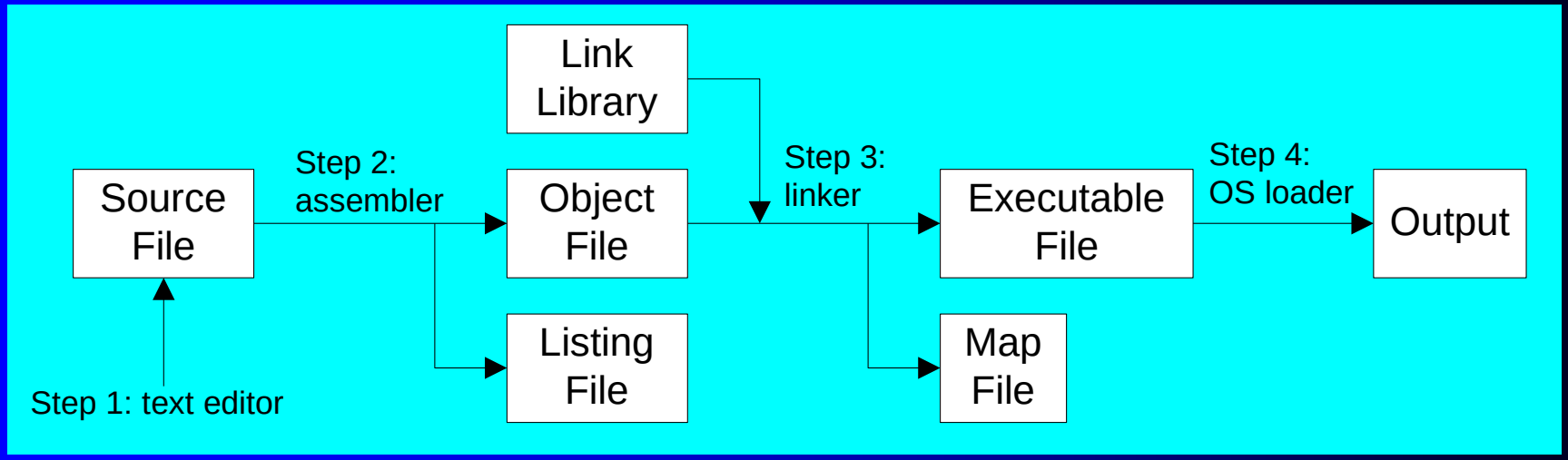
3.3 Assembling, Linking, and Running Programs

- **Assemble-Link-Execute Cycle**
- **Listing File**
- **Map File**

If you have an assembler file called first.asm then after using an assembler, you will see a file created first.lst.
We shall see what does list file means.

3.3 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.**



3.3.2 Listing File filename.lst

- **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**

List file generation

- **Generating a Source Listing File**
- Open the project. From the menu, select **Project**, select **Project Properties**. In the list box, select **Microsoft Macro Assembler**, then select **Listing File**. Set the **Assembled Code Listing file** option to **$(InputName).lst**.

Map File

- **Information about each program segment:**
  - **starting address**
  - **ending address**
  - **size**
  - **segment type**
- **Example: addSub.map (16-bit version)**

What's Next

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

3.4 Defining Data

- **Intrinsic Data Types**
- **Data Definition Statement**
- **Defining BYTE and SBYTE Data (s-stands for signed)**
- **Defining WORD and SWORD Data**
- **Defining DWORD and SDWORD Data (d for double)**
- **Defining QWORD Data**
- **Defining TBYTE Data**
- **Defining Real Number Data**
- **Little Endian Order**
- **Adding Variables to the AddSub Program**
- **Declaring Uninitialized Data**

Intrinsic Data Types (1 of 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 (Q- Quad Word)**
  - **64-bit integer**
- **TBYTE (Ten bytes)**
  - **80-bit integer**

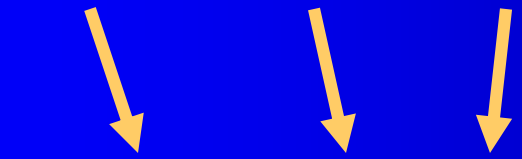
Intrinsic Data Types (2 of 2)

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

3.4.2 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*] . . .



```
value1 BYTE 10
```

- **All initializers become binary data in memory**

3.4.4 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
```

- MASM does not prevent you from initializing a BYTE with a negative value, but it's considered poor style.
- If you declare a SBYTE variable, the Microsoft debugger will automatically display its value in decimal with a leading sign.

Defining Byte Arrays

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 of 3)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It often will be null-terminated
- 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
```

Defining Strings (2 of 3)

- **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
```

Defining Strings (3 of 3)

- **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
```

Idea: Define all strings used by your program in the same area of the data segment.

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**

```
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 ; 15 bytes
```

3.4.6 Defining WORD and SWORD Data

- Define storage for 16-bit integers
  - or double characters
  - 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
```

3.4.6 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:

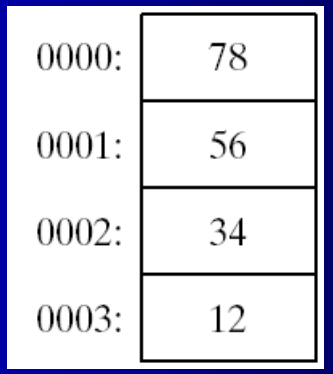
```
quad1 QWORD   1234567812345678h
val1  TBYTE   10000000001234567890h
rVal1 REAL4   -2.1
rVal2 REAL8   3.2E-260
rVal3 REAL10  4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

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:**

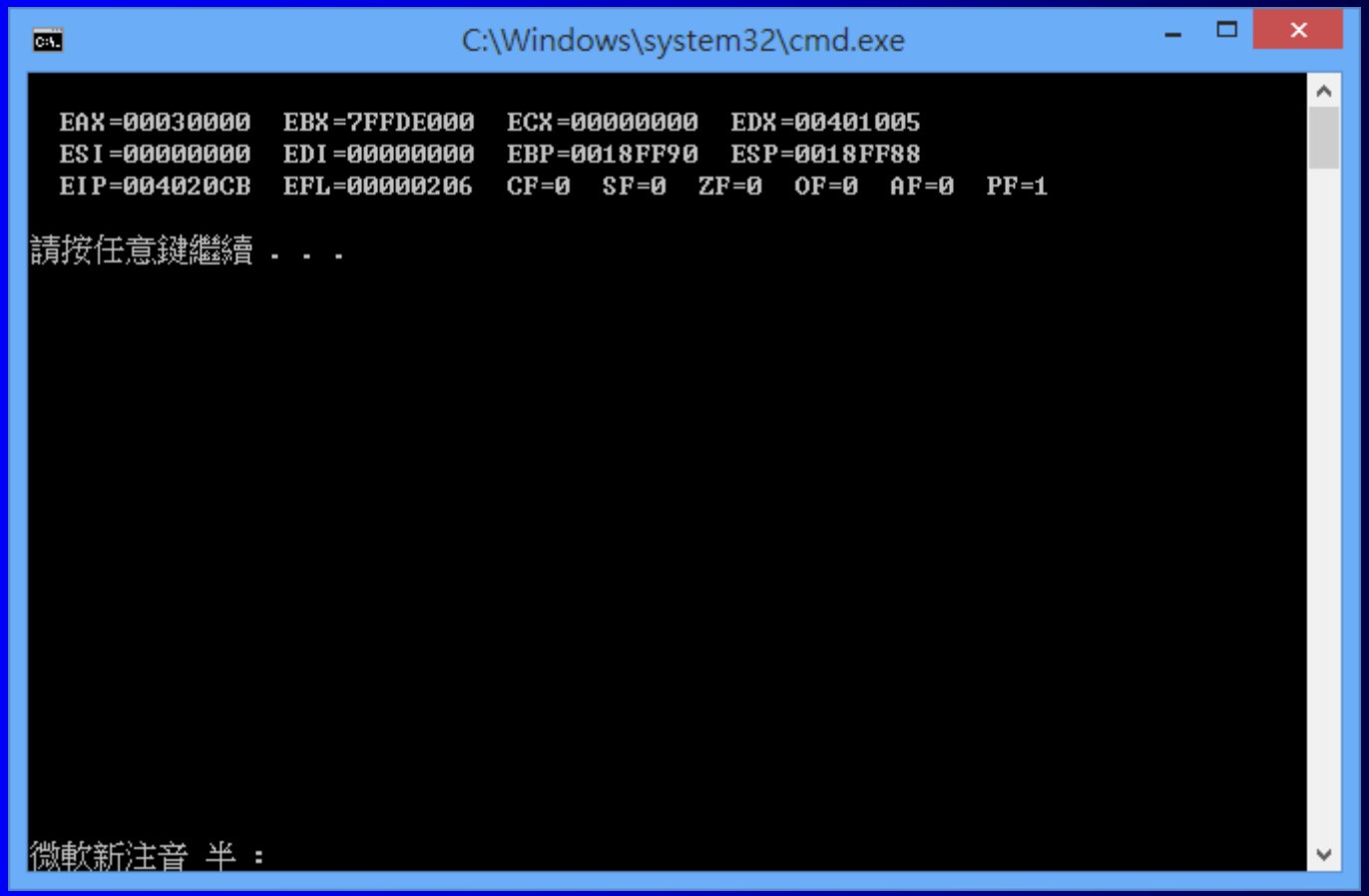
```
val1 DWORD 12345678h
```



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
```

Run of the program on the previous page.



3.4.12 Declaring Uninitialized Data

- Use the .data? directive to declare an unintialized data segment:

```
.data?
```

- Within the segment, declare variables with "?" initializers:

```
smallArray DWORD 10 DUP(?)
```

Advantage: the program's EXE file size is reduced.

What's Next

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

Symbolic Constants

- **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 ax,COUNT
```

3.5.2 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)
```

3.5.2 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
```

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
```

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"
```

What's Next

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

Real-Address Mode Programming (1 of 2)

- 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

Real-Address Mode Programming (2 of 2)

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

```
mov ax,@data
mov ds,ax
```

Refer to the author's website

- **Building 16-bit Applications (Chapters 14-17)**
- Only Chapters 14 through 17 require you to build 16-bit applications. Except for a few exceptions, which are noted in the book, your 16-bit applications will run under the 32-bit versions of Windows (XP, Vista, 7). But 16-bit applications will not run directly in any 64-bit version of Windows.
- If you plan to build 16-bit applications, you need to add two new commands to the Visual Studio Tools menu. To add a command, select **External Tools** from the Tools menu. The following dialog will appear, although many of the items in your list on the left side will be missing:

Add and Subtract, 16-Bit Version

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

64-Bit Programming

- MASM supports 64-bit programming, although the following directives are not permitted:
  - INVOKE, ADDR, .model, .386, .stack
  - (Other non-permitted directives will be introduced in later chapters)

32-Bit Version of AddTwoSum

```
1: ; AddTwoSum_64.asm - Chapter 3 example.
3: ExitProcess PROTO
5: .data
6: sum DWORD 0
8: .code
9: main  PROC
10:    mov  eax,5
11:    add  eax,6
12:    mov  sum,eax
13:
14:    mov  ecx,0
15:    call ExitProcess
16: main ENDP
17: END
```

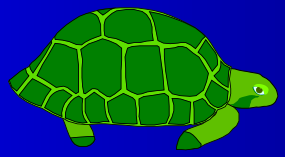
64-Bit Version of AddTwoSum use the register long enough for 64 bit

```
  .data
sum QWORD 0
.code
main  PROC
   mov  rax,5
   add  rax,6
   mov  sum,rax
```

The above 64 bit instruction can be run with 64-bit machine only. You can run on 32-bit machine.

Summary

- Integer expression, character constant
- directive – interpreted by the assembler
- instruction – executes at runtime
- code, data, and stack segments
- source, listing, object, map, executable files
- Data definition directives:
  - BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, QWORD, TBYTE, REAL4, REAL8, and REAL10
  - DUP operator, location counter ($)
- Symbolic constant
  - EQU and TEXTEQU



End of Chapter 3