Digital System-CNSCC131
Course Overview

- Hardware
- Assembly coding
- Operating System(new part)
- Note: New assessment policy
  - -The weight of coursework 30%
  - -The weight of exam 70%

Learning outcome for assembly language

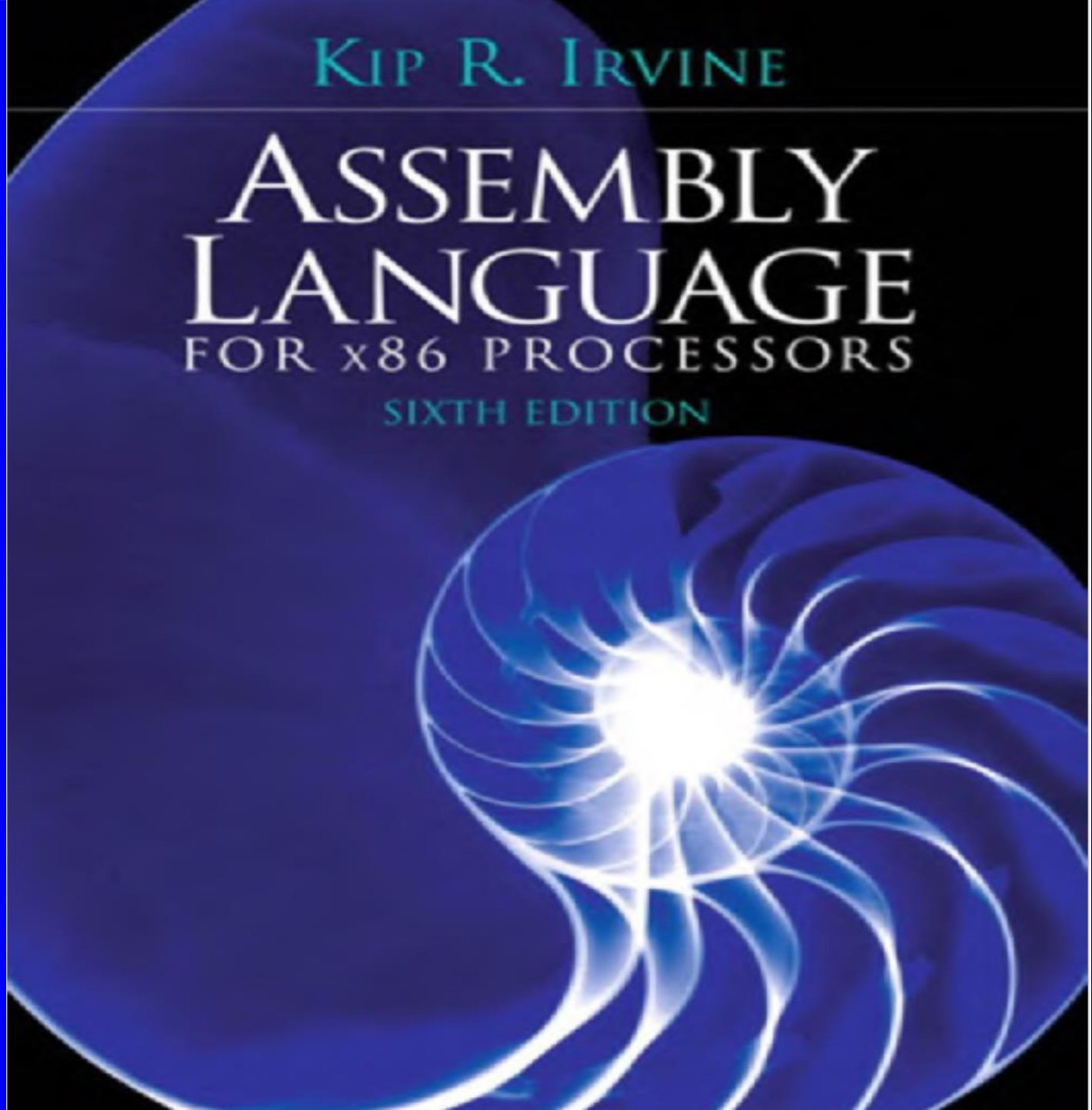
- ✓ Master basic knowledge
- ✓ Should be able to do simple programming
- ✓ Have a deep understanding about computer system

Assembly Language for x86 Processors

7th Edition

Kip Irvine

Chapter 3: Assembly Language Fundamentals


Kip R. Irvine
Assembly Language
for x86 Processors
Sixth Edition

Chapter3 Overview

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

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

Integer Constants

- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal (suffix 后缀)
  - d – decimal（can be leave out）
  - b – binary

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

Hexadecimal beginning with letter: 0A5h

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 cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
  - See MASM reference in Appendix A
- Identifiers(标识符)
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a letter, _, @, ?, or $

Directives

- Commands that are recognized and acted upon by the assembler(汇编器)
  - 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 伪指令

Instructions
（指令）

- 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：助记符 (required)
  - Operand (depends on the instruction)
  - Comment (optional)

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifer rules

- Code label
  - target of jump and loop instructions
  - example: **L1:** (followed by colon)

Mnemonics and Operands

- Instruction Mnemonics
  - memory aid 助记符
  - examples: MOV, ADD, SUB, MUL, INC, DEC
- Operands(操作数)
  - constant
  - constant expression
  - register
  - memory

Constants and constant expressions are often called immediate values 立即数

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

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
- 64-Bit Programming

Example: Adding and Subtracting Integers

```
; AddTwo.asm – adds two 32-bit integers

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
        mov    eax,5          ; move 5 to the EAX register
        add    eax,6          ; add 6 to the EAX register

        INVOKE ExitProcess,0
main ENDP
END main
```

Note： only one main proc in one project
Exclude other main proc from project

.386 means 32 bits program

In the current .model directive, the flat keyword tells the assembler to generate code for a protected mode program

the stdcall keyword enables the calling of MS-Windows functions.

Program Template

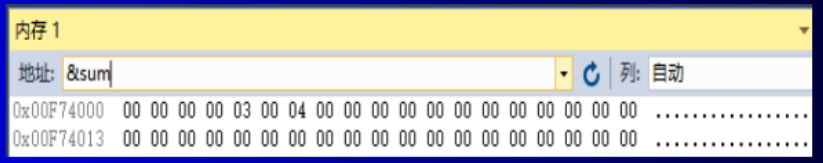
```
; Program Description:          (Template.asm)
; Author:
; Creation Date:
; Revisions:
; Date:                 Modified by:

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
; declare variables here

.code
main PROC
    ; write your code here
    INVOKE ExitProcess,0
main ENDP
; (insert additional procedures here)
END main
```

```
; Template 2

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
```

Note: code must be at the procedure, can not be in data

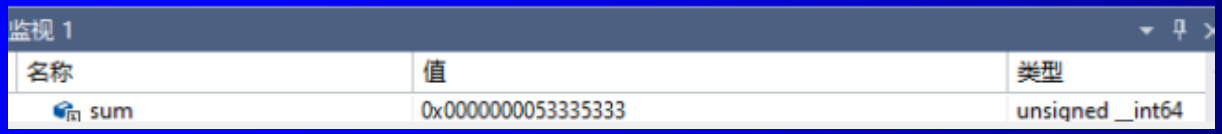
Adding Variables to AddSub

```
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
```



#use memory window to check variable address

#use watch window to check variable



Example Output
How to check register

Showing registers and flags in the debugger:

```
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
```

If no CPU status flag，right-click，select flag,

NV: no overflow OV: overflow

UP: up DN:down

DI: disable interrupt EI: enable interrupt

PL: plus NG: negative

NZ: no zero ZR: zero

NA: no assistant carry AC: assistant carry

PO: parity odd PE: parity even

NC: no carry CY: carry

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, data and 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
    - add eax, 5
  - 1-2 blank lines between procedures

What's Next

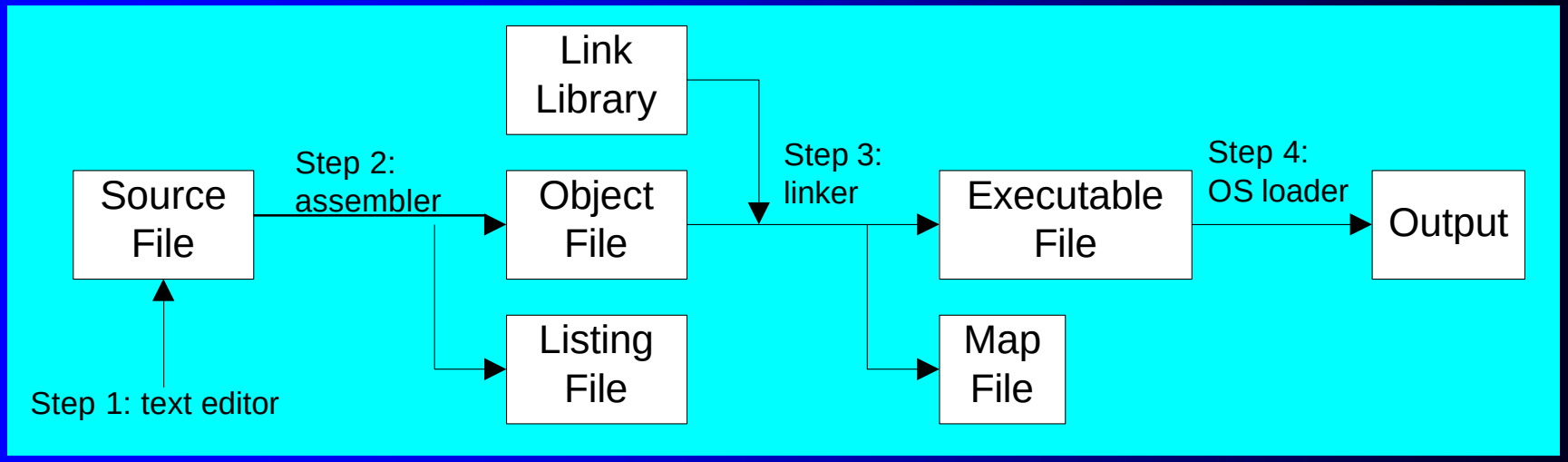
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Assembling, Linking, and Running Programs

- Assemble-Link-Execute Cycle
- 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.

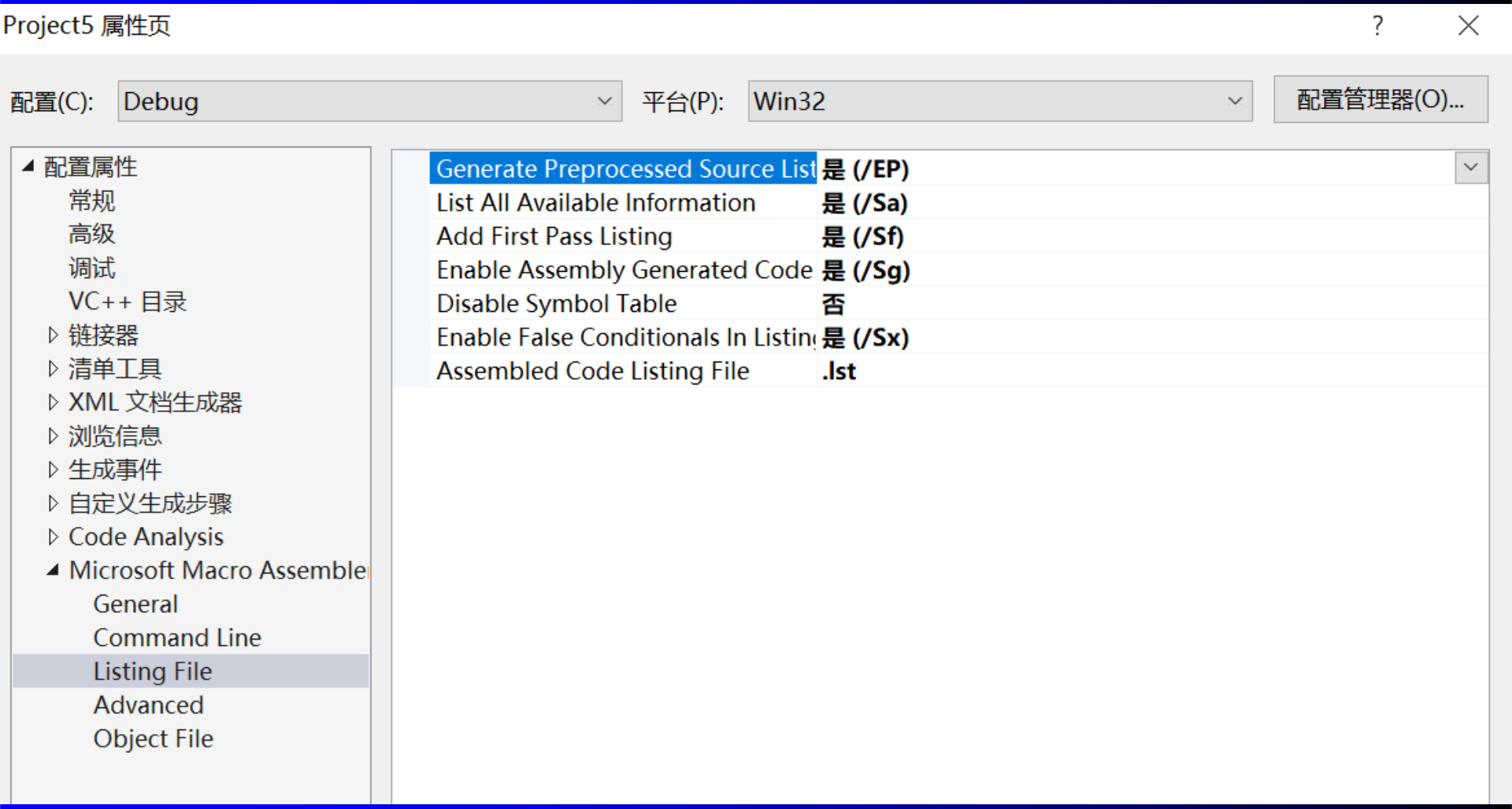


Listing File

- Use it to see how your program is compiled
- Contains
  - source code
  - addresses
  - object code (machine language)
  - segment names
  - symbols (variables, procedures, and constants)

```
00000000                .code
00000000                main proc
00000000  A1 00000000 R          mov eax,firstval
00000005  B0 80                  mov al,-128
00000007  F6 D8                  neg al
00000009  03 05 00000004 R       add eax,secondval
0000000F  03 05 00000008 R       add eax,thirdval
00000015  A3 0000000C R          mov sum,eax
                        ;call dumpregs
                        invoke ExitProcess,0
0000001A  6A 00           *      push   +000000000h
0000001C  E8 00000000 E   *          call   ExitProcess
00000021                main endp
                    end main
```

- Set up the assembler property to enable .lst



In anti-assembly mode

```
    mov eax,firstval
00141010 A1 00 40 14 00       mov         eax,dword ptr [firstval (0144000h)]
    mov al,-128
00141015 B0 80                mov         al,80h
    neg al
00141017 F6 D8                neg         al
    add eax,secondval
00141019 03 05 04 40 14 00    add         eax,dword ptr [secondval (0144004h)]
    add eax,thirdval
0014101F 03 05 08 40 14 00    add         eax,dword ptr [thirdval (0144008h)]
    mov sum,eax
00141025 A3 0C 40 14 00       mov         dword ptr [sum (014400Ch)],eax
```

What's Next

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

Defining Data

- Intrinsic Data Types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- 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
  - 64-bit integer
- TBYTE
  - 80-bit integer

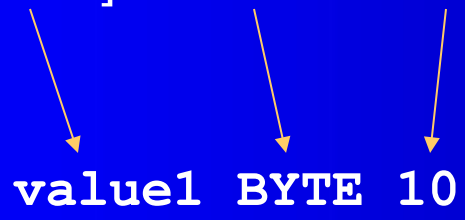
Intrinsic Data Types (2 of 2)

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

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*] . . .



- All initializers become binary data in memory

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(空白符),
- 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

          Note: 0 should be at the end for string
```

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 回车
  - 0Ah = line feed 换行

```
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     ; 5 bytes
```

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
```

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:

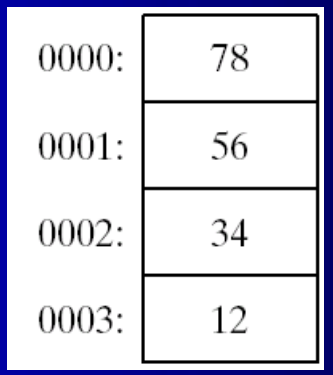
```
quad1 QWORD  1234567812345678h
val1  TBYTE  10000000001234567890Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

Little Endian Order
（小端模式）

- 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
```



Declaring Unitialized 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.

No room assign in compiling stage

Assign space in running time

What's Next

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

Symbolic Constants
符号常数

- Equal-Sign Directive (=)
- EQU Directive
- TEXTEQU Directive
- $: Calculating the Sizes of Arrays and Strings

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
```

No memory room

Generally, it is defined in front of .data

EQU Directive

- Define a symbol as either an integer or text expression.
- Cannot be redefined，不能重新定义

```
PI EQU <3.1416>   #constant
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 文本宏
- 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
```

Calculating the Size of a Byte Array

- current location counter:
- $ is constant symbolic for current address 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

Divide total number of bytes by 2 (the size of a word)

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

ListSize is the number of word

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
```

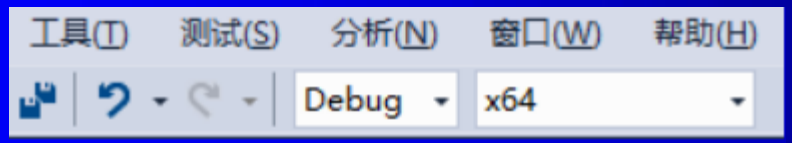
Here: ListSize is the number of dword

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- **64-Bit Programming(not required)**

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)
  - Select X64 platform under VS



64-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 register

```
10:    mov   rax,5
11:    add   rax,6
12:    mov   sum,rax
```

Things to Notice About the Previous Slide

- The following lines are not needed:

```
.386
.model flat,stdcall
.stack 4096
```

- INVOKE is not supported.
- CALL instruction cannot receive arguments
- Use 64-bit registers when possible

Summary

- Integer expression, character constant, string
- Elementary assembly code structure and example
  - 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