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

Example Program

```
main PROC
mov eax, 5      ;  move 5 to the EAX register
add eax, 6      ;  add 6 to the EAX register
call WriteInt   ;  display value in EAX
exit      ;  quit
main ENDP
```

Add two numbers and displays the result

Integer Constants

- [{+ | -}] digits [radix]
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - q | o – octal
  - d – decimal
  - b – binary
  - r – encoded real
- If no radix given, assumed to be decimal

26	Decimal	42o	Octal
26d	Decimal	1Ah	Hexadecimal
11010011b	Binary	0A3h	Hexadecimal
42q	Octal		

Integer Expressions

integer values and arithmetic operators

- Operators and precedence levels:

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

Must evaluate to an integer that can be stored in 32 bits

These can be evaluated at assembly time – they are not runtime expressions

- Examples:

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

Precedence Examples:

4 + 5 * 2	Multiply, add
12 – 1 MOD 5	Modulus, subtract
-5 + 2	Unary minus, add
(4 + 2) * 6	Add, multiply

Real Number Constants

- Represented as decimal real.
- Decimal real contains optional sign followed by integer, decimal point, and optional integer that expresses a fractional and an optional exponent
  - [sign] integer.[integer] [exponent]
  - Sign {+, -}
  - Exponent E[{+, -}] integer
- Examples
  - 2.
  - +3.0
  - -44.2E+05
  - 26.E5

Character and String Constants

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

Reserved Words

- Reserved words cannot be used as identifiers
- Instruction mnemonics
  - MOV, ADD, MUL,, …
- Register names
- Directives – tells MASM how to assemble programs to reserve space in the program (for variable for example).
- type attributes – provides size and usage information for variables and operands.
  - BYTE, WORD
- Operators – used in constant expressions
- predefined symbols – @data, which return constant integer values

Identifiers

- Identifiers
  - Programmer-chosen name to identify a variable, constant, procedure, or code label
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a letter (A..Z, a..z), _, @, ?, or $
    - Subsequent characters may also be digits
  - Cannot be the same as a reserved word
  - @ is used by assembler as a prefix for predefined symbols, so avoid it in your own identifiers.
- Examples
  - Var1, Count, $first, _main, MAX, open_file, myFile, xVal, _12345

Directives

- Directives can define variables, macros, and procedures.
- They can assign names to memory segments and perform many other tasks related to the assembler.
- It is 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

```
myVar  DWORD 26          ; DWORD directive, reserve
                         ; enough space for double word
```

- One important function of assembler directives is to define program sections, or *segments*

.data
The .DATA directive identifies the area of a program containing variables

. code
The .CODE directive identifies the area of a program containing executable instructions

. stack 100h
The .STACK directive identifies the area of a program holding the runtime stack, setting its size

Instructions

- An instruction is a statement that becomes executable when a program is assembled.
- 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)
- Basic syntax
  - [label:] mnemonic [operands] [; comment]

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifier rules
- Data label : identifies the location of a variable to reference the variable in code
  - 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 and Operands

- Instruction Mnemonics
  - is short word that identifies an instruction.
  - examples: MOV, ADD, SUB, MUL, INC, DEC
- Operands
  - constant — 96
  - constant expression — 2 + 4
  - register — eax
  - memory (data label) — count

Constants and constant expressions are often called immediate values

Mnemonics and Operands
Examples

```
STC instruction
        stc                     ; set Carry flag

INC instruction
        inc     eax             ; add 1 to EAX

MOV instruction
        mov     count, ebx      ; move EBX to count
                                ; first operation is destination
                                ; second is the source
IMUL instruction (three operands)
        imul    eax, ebx, 5     ; ebx multiplied by 5, product in EAX
```

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 (;) or TITLE directive
- Multi-line comments
  - begin with COMMENT directive and a programmer-chosen character
  - end with the same programmer-chosen character

Comments

- Single line comment
  - inc eax ; single line at end of instruction
  - ; single line at beginning of line
- Multiline comment

```
COMMENT !
        This line is a comment
        This line is also a comment
!
COMMENT &
        This is a comment
        This is also a 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, constant-expression

NOP instruction

- Doesn’t do anything
- Takes up one byte
- An operation that does nothing is sometimes useful for timing or debugging, or as a placeholder for future code

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
```

Example: Adding and Subtracting Integers

```
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.

Example: Adding and Subtracting Integers

```
.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.

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

The MOV instruction moves (copies) the integer 10000h to the EAX register.

Example: Adding and Subtracting Integers

```
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.

Example: Adding and Subtracting Integers

```
Exit
```

The exit statement (indirectly) calls a predefined MS-Windows function that halts the program. it's a macro command defined in the Irvine32.inc.

```
main ENDP
```

The ENDP directive marks the end of the main procedure.

```
END main
```

The END directive marks the last line of the program to be assembled.

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 of 2)

- Some approaches to capitalization
  - Use lowercase for keywords, mixed case for identifiers, and all capitals for constants.
  - capitalize all reserved words, including instruction mnemonics and register names
  - Capitalize directives and operators except that lowercase is used for the .code, .stack, .model, and .data directives.

Suggested Coding Standards (2 of 2)

- Other suggestions
  - descriptive identifier names
  - spaces surrounding arithmetic operators
  - blank lines between procedures
- 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

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

`.386`

The .386 directive identifies the minimum CPU required for this program (Intel386, the first x86 processor).

`.MODEL flat,stdcall`

The .MODEL directive is used to identify the segmentation model used by the program and it identifies the convention used for passing parameters to procedures. The **flat** keyword tells the assembler to generate code for a protected mode program. The **stdcall** keyword enables the calling of MS-Windows functions.

`.STACK 4096`

The .STACK directive identifies the area of a program holding the runtime stack, and setting its size 4096.

Alternative Version of AddSub

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

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

```
INVOKE ExitProcess,0
```

INVOKE is directive that calls a procedure or function.
ExitProcess function ends the program.

Program Template

```
TITLE Program Template                (Template.asm)

;/////////////////////////////////////////////////////
; Program Name:
; Program Description:
; Author:
; Version:
; Date:
; Other Information:
;/////////////////////////////////////////////////////

INCLUDE Irvine32.inc
.data
    ; (insert variables here)
.code
main PROC
    ; (insert executable instructions here)
    exit
main ENDP
    ; (insert additional procedures here)
END main
```

What's Next

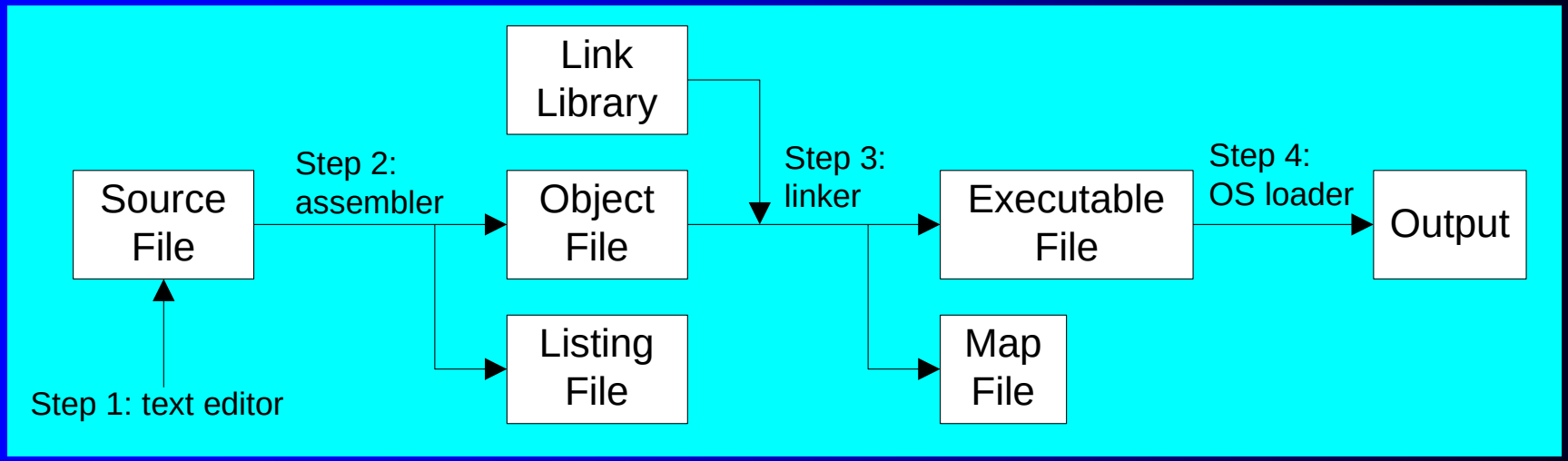
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants
- Real-Address Mode Programming

Assembling, Linking, and Running Programs

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

Assemble-Link Execute Cycle

- Assembly language program must be translated to machine language for the target processor.
- 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.



Assemble-Link-Execute

Step 1: A programmer uses a text editor to create the source file.

Step 2: The assembler reads the source file and produces an object file, and listing file.

Step 3: The linker reads the object file and checks to see if the program contains any calls to procedures in a link library, combines them with the object file, and produces the executable file.

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

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)

Map File

- Information about each program segment:
  - starting address
  - ending address
  - size
  - segment type

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

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 short real
- REAL8
  - 8-byte long real
- REAL10
  - 10-byte extended real

Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- It creates variables based on intrinsic data types

- Syntax:

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

```
value1 BYTE 10
```

- All initializers become binary data in memory

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

```
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,12h,22h
```

	Offset	Value
list1	0000	10
	0001	20
	0002	30
	0003	40
list2	0004	10
	0005	20
	0006	30
	0007	40
	0008	50
	0009	60
	000A	70
	000B	80
	000C	81
	000D	82
	000E	83
	000F	84
list3	0010	

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 (ending with ,0)
- 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

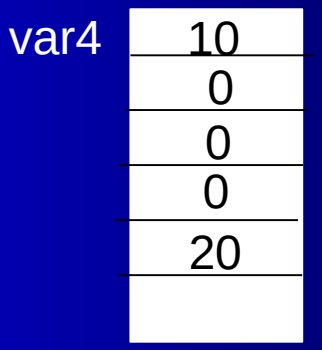
```
str1 BYTE "Enter your name:    ",0Dh,0Ah
     BYTE "Enter your address: ",0

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

- The DUP operator allocates storage for multiple data items, using a constant expression as a counter.
- 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
var4 BYTE 10,3 DUP(0),20     ; 5 bytes
```



Using the DUP Operator

- Str1 BYTE 4 DUP("STACK")

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
List    WORD   1,2,3,4,5      ; array of words
array   WORD   5 DUP(?)       ; uninitialized array (5
                                word elements)
```

- The legacy DW directive can also be used:

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

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
```

The legacy DD directive can also be used to define doubleword data.

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

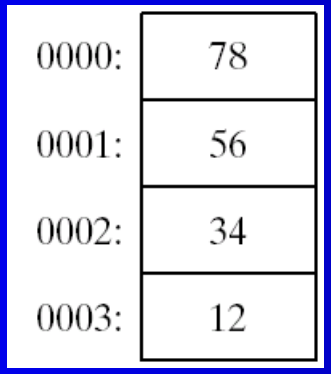
Defining QWORD, TBYTE, Real Data

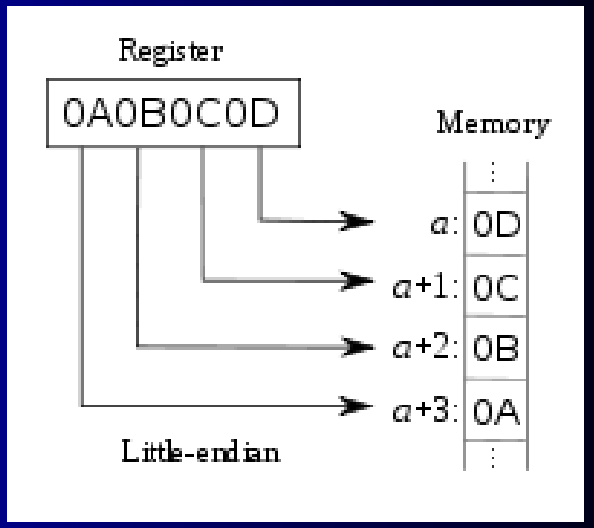
Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  1234567812345678h
val1  TBYTE  10000000001234567890Ah
rVal1 REAL4  -2.1
ShortArray REAL4 20 DUP(0.0)
```

Little Endian Order

- The terms endian and endianness, refers to how bytes of a data word are ordered within memory.
- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first memory address (low to high).
- Example: `val1 DWORD 12345678h`





Some other computer systems use big endian order (high to low).

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
```

Declaring Unitialized Data

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

```
.data?
  bigArray DWORD 5000 DUP(?) ; 20,000 bytes, not initialized
```

- 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

- Symbolic constants are used to assign names (identifiers) to constant values.
- Using symbolic constants instead of numbers makes your code more readable and easier to maintain.
- *name = expression*
  - expression is a 32-bit integer (expression or constant)
  - Can not redefine
  - *name* is called a symbolic constant

	Symbol	Variable
Uses storage?	No	Yes
Value changes at runtime?	No	Yes

- good programming style to use symbols

```
COUNT = 500

mov eax,COUNT
```

Calculating the Size of a Byte Array

- *current location counter*: $
  - return the offset associated with the current program statement.
  - subtract address of list
  - difference is the number of bytes

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

- subtracting the offset of list from the current location counter ($) to get the number of bytes as size of array.

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 10000000h,20000000h
ListSize = ($ - list) / 4
```

EQU Directive

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

name EQU expression
name EQU symbol
name EQU <text>

```
pressKey EQU <"Press any key to continue...",0>
matrix1 EQU 10 * 10
matrix2 EQU <10 * 10>
.data
prompt BYTE pressKey
M1 WORD matrix1       ; matrix1 100
M2 WORD matrix2       ; matrix2 10 * 10
```

TEXTEQU Directive

- Define a symbol as either an integer, text expression or text macro.
- Can be redefined

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

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?"> ; text macro
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)      ; evaluates the expression
move TEXTEQU <mov>
setupAL TEXTEQU <move 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 Address
- Advantages
  - enables calling of MS-DOS and BIOS functions
  - no memory access restrictions
- Disadvantages
  - 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  ; starting location of the data segment
mov ds,ax
```

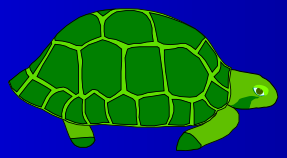
- MOV instruction does not permit a constant to be moved directly to data segment register (ds).

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
```

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



4C 61 46 69 6E