

Assembly Language Fundamentals

Outline

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

Integer Literals

- **Literals:** also known as **constant**
- Syntax: **[{+|-}] digits [radix]**
- Optional leading + or – sign
- Digits can be 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 must have a leading zero, e.g., 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 Literals (Skip!)

- Decimal Real
 - [sign] integer.[integer [exponent]]
 - E.g., 2., +3.0, -44.2E+05, 26.E5
 - Note, a real number constant must have a **digit** and a **decimal point**
- Encoded Real
 - Specify by IEEE standard

Character and String Literals

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

Practice: I don't "listen" to radio

Reserved Words

- Reserved words cannot be used as identifiers
 - Instruction mnemonics (ADD, MOV ...)
 - Directives (tell MASM how to assemble programs)
 - Type attributes (BYTE, WORD ...)
 - Operators (+, - ...)
 - Predefined symbols (@data ...)

```
section .data
    msg db 'Hello, world!', 0

section .text
global _start

_start:
```

Identifiers

- **Identifiers:** a programmer chosen name
 - 1-247 characters, including digits
 - Case insensitive (assembler dependent)
 - First character must be a letter, _, @, or \$

Directives

- **Directives:** commands that are recognized and acted upon by the **assembler**
 - Command understood only by the assembler
 - Not part of Intel instruction set
 - Do not execute at run time (since not instruction)
 - Case insensitive
- Different assemblers have different directives
 - NASM != MASM

```
.stack 100h
.data
message db 'Hello, world!$'

.code
start:
    ; 程序代码...
    mov ah, 09h
    lea dx, message
    int 21h

    mov ax, 4C00h
    int 21h
```

Instructions

- Assembled into **machine code** by assembler
- Executed at runtime by the CPU
- Parts:
 - Label (optional)
 - Instruction Mnemonic (required)
 - Operand (depends on the instruction)
 - Comment (optional)

Label: Mnemonic Operand(s) ; Comment

Assembly Language v.s. Machine Code

Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C 'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	

Labels

- Act as place markers
 - Marks **the address (offset)** of code and data
 - See the previous slide
- Follow identifier rules
- **Data label**
 - Example: first BYTE 10
- **Code label**
 - Target of jump and loop instructions
 - Example:

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

Mnemonics and Operands

- **Instruction Mnemonics**

- A short word that identifies an instruction
- Examples: MOV, ADD, SUB, MUL, INC, DEC

- **Operands:** an instruction can have 0~3 operands

- **constant** (immediate value)
 - `mov ax, 96`
- **constant expression** (immediate value)
 - `mov ax, 5+4`
- **register**
 - `mov ax, bx`
- **memory** (data label)
 - `mov count, bx`

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** (see the following slide) and a programmer-chosen character
 - End with the same programmer-chosen character

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

The NOP (No Operation) Instruction

- NOP: No Operation
- Take up 1 byte of storage and does not do any work
- Used to **align** code to **even-address boundaries**
- Example
 - 00000000 mov ax, bx
 - 00000003 nop
 - 00000004 mov edx, ecx
 - Address of the third instruction is aligned to doubleword boundary, i.e., **alignment**
- IA-32 are designed to load code/data more quickly from even doubleword address

Exercise

- Using the value **-35**, write it as an integer literal in **decimal**, **hexadecimal**, and **binary** formats that are consistent with the MASM syntax
- Is **A5h** a valid hexadecimal literal?

Outline

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

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
```

Example: Adding and Subtracting Integers

- **.386** directive (see the following slide)
 - This is a 32-bit program
- **.model** directive (shown later!)
 - Select the program's memory model (**flat**)
 - Identify the calling convention (**stdcall**) for procedures
- **.stack** directive
 - A **stack segment** with size 4096 bytes
- Line 7
 - Declare a **function prototype** for the **ExitProcess** function (a Windows function)

Example: Adding and Subtracting Integers

- **.code**: directive
 - Mark the beginning of the **code segment** (see the following slide)
- **PROC**: directive
 - Identify the beginning of a procedure
- **mov eax, 10000h**
 - First operand: destination operand
 - Second operand: source operand
- **INVOKE ExitProcess, 0**
 - Invoke a predefined MS-Windows function (ExitProcess) and return 0
- **ENDP**: directive
 - Mark the end of the main procedure
- **END**: directive
 - Also identify the name of the program's startup procedure
 - See the following slide

Another Example: XYZ is the Startup's Procedure

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

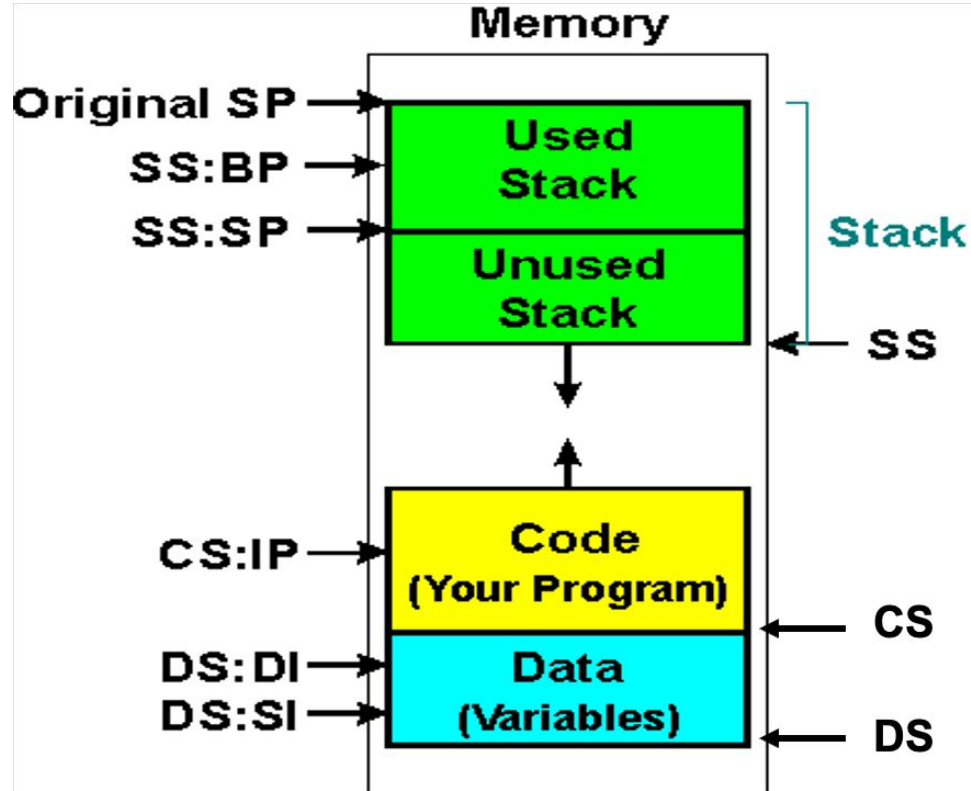
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.code
xyz PROC
    mov  eax,5      ; move 5 to the EAX register
    add  eax,6      ; add 6 to the EAX register

    INVOKE ExitProcess,0
xyz ENDP
END xyz
```

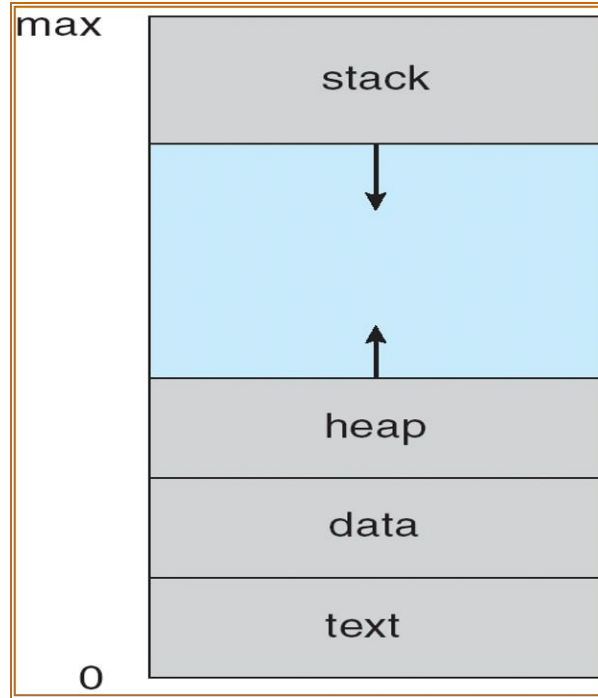
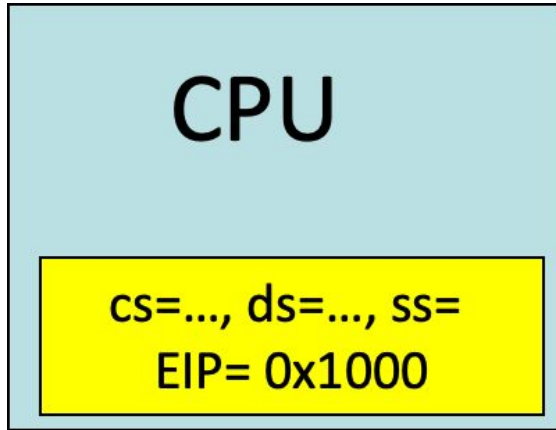
Code Segment, Data Segment, and Stack Segment

- Each program contains
 - **Code segment:** executed code
 - **Data segment:** global variables
 - **Stack segment:** local variable, parameters, return address.....
 - Pointed by CS, DS, SS segment registers

Code Segment, Data Segment, and Stack Segment



An Executed Program



Program Template

```
; Program template (Template.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.code
main PROC
    ; write your code here

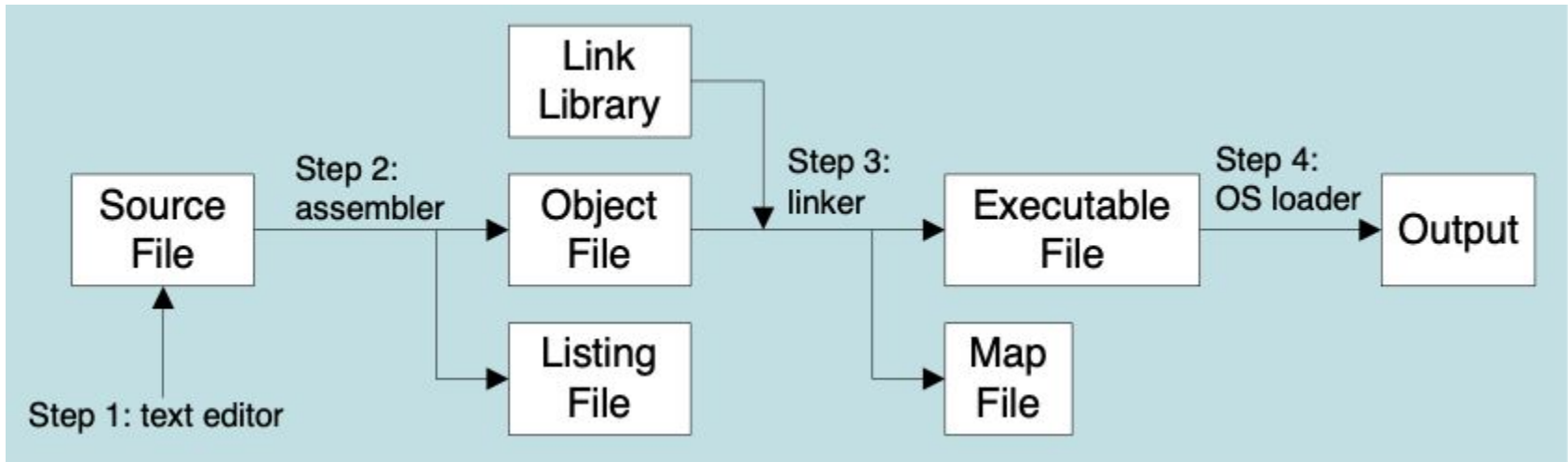
    INVOKE ExitProcess,0
main ENDP
END main
```

Outline

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

Assemble-Link Execute Cycle

- Steps from [creating a source program](#) through [executing the program](#).
 - If the source code is modified, Steps 2 through 4 must be repeated.



Listing File (Skip)

- 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 (Skip)

- Information about each program segment:
 - Starting address
 - Ending address
 - Size
 - Segment type

Assembling, Linking, and Running

- <http://www.asmirvine.com>
 - Getting started with MASM and Visual Studio

Outline

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

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

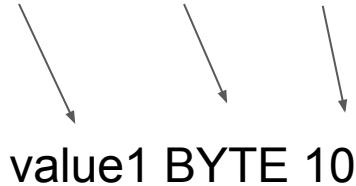
Intrinsic Data Types

- 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] . . .



value1 BYTE 10

- All initializers become **binary data** by assembler

Example: Adding and Subtracting Integers

```
;AddVariables.asm - Chapter 3 example
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
sum          DWORD 0
.code
main PROC
    mov     eax,5
    add     eax,6
    mov     sum,eax
    INVOKE  ExitProcess,0
main ENDP
END main
```

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

<code>value1</code>	<code>BYTE</code>	<code>'A'</code>	<code>; character constant</code>
<code>value2</code>	<code>BYTE</code>	<code>0</code>	<code>; smallest unsigned byte</code>
<code>value3</code>	<code>BYTE</code>	<code>255</code>	<code>; largest unsigned byte</code>
<code>value4</code>	<code>SBYTE</code>	<code>-128</code>	<code>; smallest signed byte</code>
<code>value5</code>	<code>SBYTE</code>	<code>+127</code>	<code>; largest signed byte</code>
<code>value6</code>	<code>BYTE</code>	<code>?</code>	<code>; uninitialized byte</code>

Defining BYTE and SBYTE Data

- A variable name is a label that marks the **offset** of a variable from the beginning of its enclosing segment

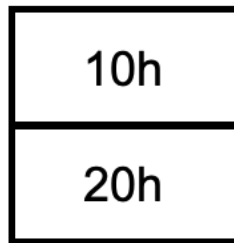
- Example

- .data

value1 BYTE 10h

value2 BYTE 20h

data segment



低位址

高位址

```
; 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
```

低位址

高位址

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
```

	value	offset
低位址	10	0000
	20	0001
	30	0002
高位址	40	0003

list1

Note, labels refer the offset of the first byte,
Example: list1 refer to the offset of the first byte, i.e., 10

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**
 - It usually has a null byte (a byte containing the value 0) 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

- 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

- 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

```
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

Assume `myList` starts from an offset 0
Note that the offsets increment by 2

Offset	Data
0000	1
0002	2
0004	3
0006	4
0008	5

`myList`

array of words

```
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
```

- Array of Doubleword:

```
myList    DWORD    1, 2, 3, 4, 5
```

Assume `myList` starts from an offset 0
Note that the offsets increment by 4

Offset	Data
0000	1
0004	2
0008	3
000C	4
0010	5

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 v.s. Big Endian Order

- How to store **each byte of a data** larger than one byte in memory?
- Example:

val1 DWORD 12345678h

Little Endian Order

- Example:

val1 DWORD 12345678h

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

- The least significant byte occurs at the first (lowest) memory address.

Big Endian Order

- Some other computers use big endian order
- Example: 12345678h

0000	12
0001	34
0002	56
0003	78

Adding Variables to AddSub

```
;AddVariables.asm - Chapter 3 example
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
firstval  DWORD 20002000h
Secondval DWORD 11111111h
thirdval  DWORD 22222222h
sum        DWORD 0
.code
main PROC
    mov  eax,firstval
    add  eax,secondval
    add  eax,thirdval
    mov  sum,eax
    INVOKE ExitProcess,0
main ENDP
END main
```

Declaring Uninitialized Data (Skip!)

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

```
.data
```

```
smallArray DWORD 10 DUP (5)    ;40 bytes
```

```
.data?
```

```
bigArray DWORD 5000 DUP(?)     ;20000 bytes
```

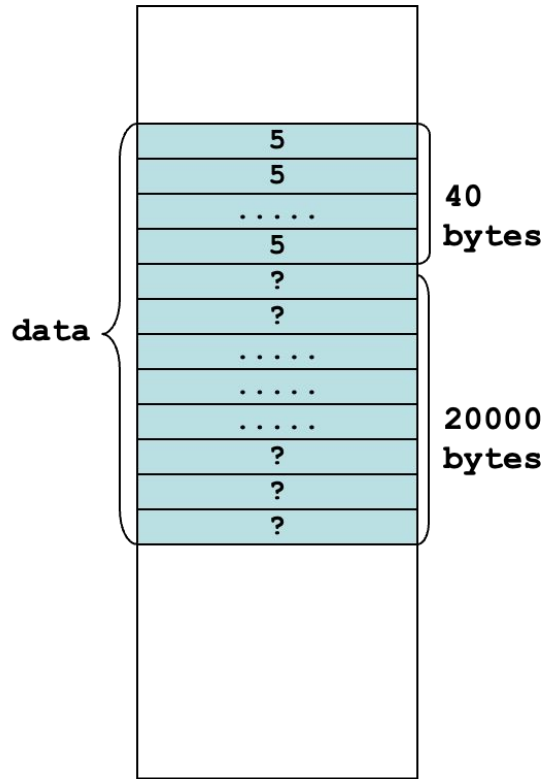
- On the other hand, the following code produces a compiled program that is 20000 bytes larger:

```
.data
```

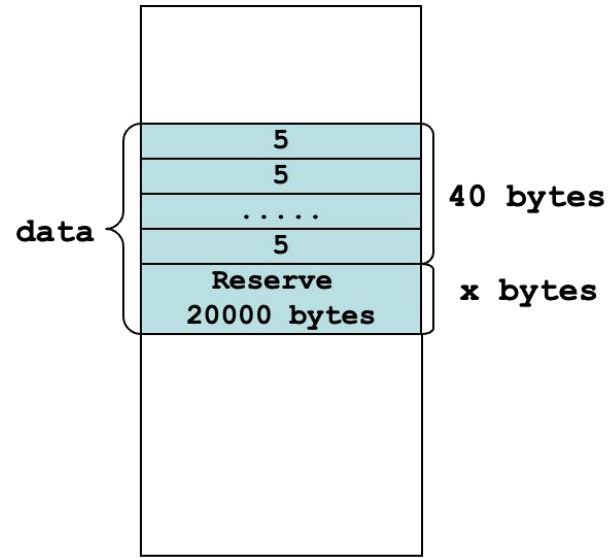
```
smallArray DWORD 10 DUP (5)    ;40 bytes
```

```
bigArray DWORD 5000 DUP(?)     ;20000 bytes
```

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



(a) The **EXE file** if not using .data?



(b) The **EXE file** if using .data?

However, when both .exe files are loaded into memory, the memory requirements of data are the same (20040 bytes)

Exercise

- Create an uninitialized data declaration for a 16-bit signed integer
- Create an uninitialized data declaration for a 8-bit unsigned integer
- Create an uninitialized data declaration for a 8-bit signed integer
- Which data type can hold a 32-bit signed integer?

Outline

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

Symbolic Constants

- Symbolic constant: associate an identifier (a symbol) with either an integer expression or some text
- A symbolic constant does not use any storage
- Used only during the assembly
 - Can not change at runtime
- Three methods
 - Equal-Sign Directive
 - Create symbols that represents integer constants
 - EQU Directive and TEXTEQU Directive
 - Create symbols that represents arbitrary text

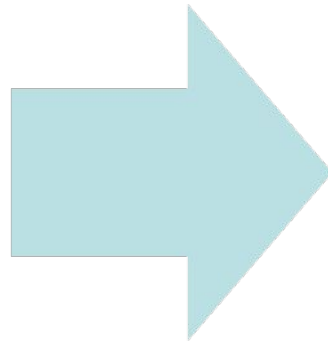
Equal-Sign Directive

- Associate a symbol with an integer expression
- name = expression
 - expression is a 32-bit integer (expression or constant)
 - name is called a **symbolic constant**
- When **assembled**, all occurrences of names are replaced by expression
 - See the following slide

```
COUNT = 500
```

```
...
```

```
mov ax, COUNT
```



```
mov ax, 500
```

Equal-Sign Directive

- Why use symbols?
 - Programs are easier to read and maintain if symbols are used
 - Good programming style to use symbols

Equal-Sign Directive

- Redefinition: a symbol defined with “=” can be redefined any number of times
- Example:

```
COUNT = 5  
mov  al, count      ;al = 5  
COUNT = 10  
mov  al, count      ;al = 10
```

Equal-Sign Directive

- **Current Location Counter**
 - An important symbol: \$
- Example:
 - Declare a variable named **selfPtr** and initializes it with its own location counter

selfPtr DWORD \$

Calculating the Size of a **Byte** Array

- Example

List BYTE 10, 20, 30, 40

ListSize = 4 ; bad programming style

- Use **current location counter**: \$
 - Return the **offset** associated with the current program statement

- Example
 - Subtract address of list
 - The difference is the number of bytes

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

Note: **ListSize** must follow immediately after **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
```

Note: because \$ returns offset in unit of bytes but List's element is 2 bytes long

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.
- Syntax
 - name EQU expression
 - name EQU symbol ; symbol must have to defined by = or EQU
 - name EQU <text>
- Cannot be redefined

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


EQU Directive

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

Integer expression is
evaluated by assembler



Text is copied directly

```
.data  
M1 WORD 100  
M2 WORD 10x10
```

TEXTEQU Directive

- Define a symbol as either an integer or text expression.
- Create a **text macro**
- Syntax
 - name TEXTEQU <text>
 - name TEXTEQU textmacro ; defined by TEXTEQU before
 - name TEXTEQU %constExpr
- Can be redefined

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
.data
prompt1 BYTE continueMsg
```

TEXTEQU Directive

```
rowSize = 5  
count TEXTEQU %(rowSize * 2)      ; evaluates the expression  
move TEXTEQU <mov>  
setupAL TEXTEQU <move al, count>
```

```
;The same as  
setupAL TEXTEQU <mov al, 10>
```

Outline

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