CSC 3210 Computer Organization and Programming

CHAPTER3: ASSEMBLY LANGUAGE FUNDAMENTALS

Outline

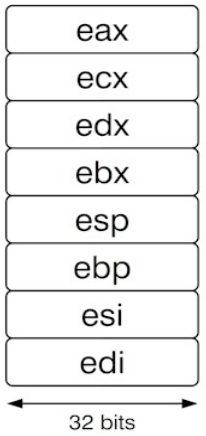
- **Basic Elements of Assembly Language**
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Basic Language Elements

- **Parts of assembly program:**
  - Main procedure (entry point)
  - eax register
  - **mov & add** instruction (eax=11)
  - INVOKE (call, unix) ExitProcess, 0
  - Comments begin with **semicolon ;**

AddTwo program:

```
1: main PROC
2:    mov eax,5                ; move 5 to the eax register
3:    add eax,6                ; add 6 to the eax register
4:
5:    INVOKE ExitProcess,0     ; end the program
6: main ENDP
```

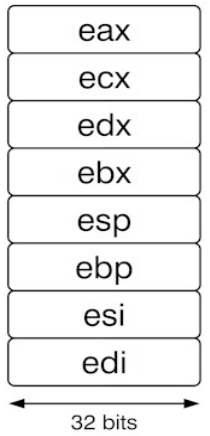


Basic Language Elements

- No output?
- **Debugger**!
  - See what is in CPU registers and flags
    - Need to manage data representation and instruction formats
- Can NOT run this program

AddTwo program:

```
1: main PROC
2:    mov eax,5                ; move 5 to the eax register
3:    add eax,6                ; add 6 to the eax register
4:
5:    INVOKE ExitProcess,0     ; end the program
6: main ENDP
```

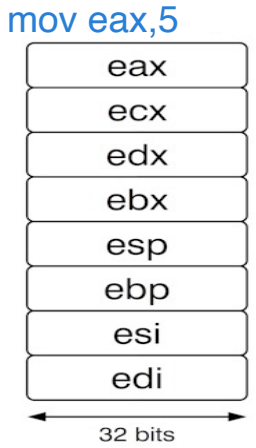


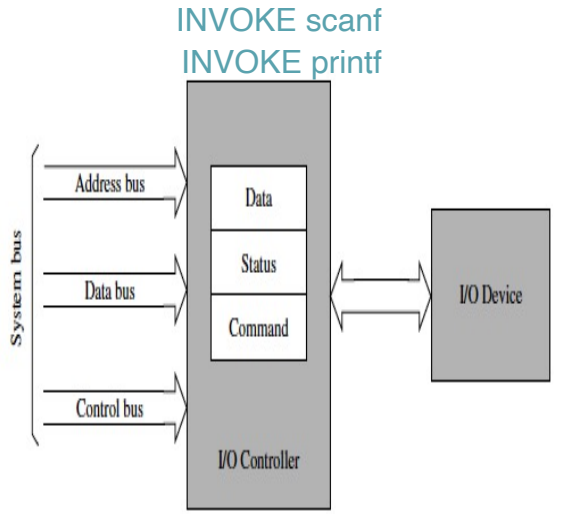
AddTwo program:

```
1: main PROC
2:    mov eax,5                    ; move 5 to the eax register
3:    add eax,6                    ; add 6 to the eax register
4:
5:    INVOKE ExitProcess,0         ; end the program
6: main ENDP
```

- Instructions operand can be
- register,
- memory location or
- immediate

  - **Immediate operand:**
    - Even though the data are in memory, it is **located in the code segment**, **not** in the data segment
    - **Signed** integer
    - Has a limited **range**



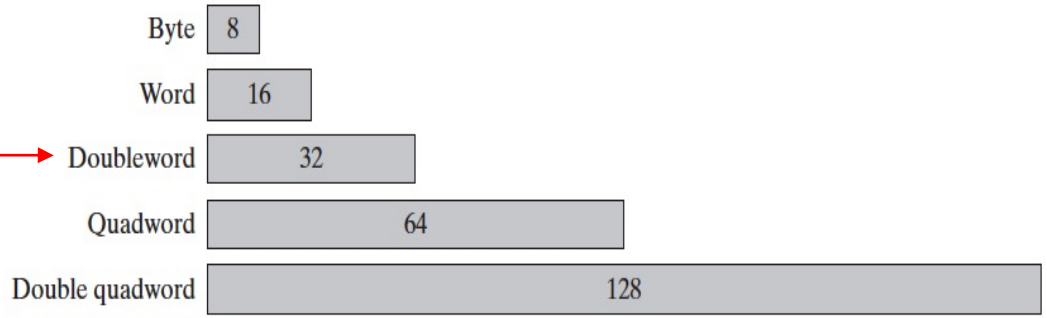


Basic Language Elements: Adding a Variable

- **Declarations**
  - Identify Code and Data areas (directives)
    - **.data**
    - **.code**
  - Segments (how about .stack)
- Variable declaration (sum)
  - DWORD (keyword): size of 32 bits
  - No checking in what gets into inside the variable (not like C/Java)
  - Must **defined** and **initialized** to **zero**

- Syntax rules imposed by MS-**MASM**

```
1: .data                        ; this is the data area
2: sum DWORD 0                  ; create a variable named sum
3:
4: .code                        ; this is the code area
5: main PROC
6:    mov eax,5                 ; move 5 to the eax register
7:    add eax,6                 ; add 6 to the eax register
8:    mov sum,eax
9:
10:   INVOKE ExitProcess,0      ; end the program
11: main
```



Basic Elements of Assembly Language

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

Basic Elements of Assembly Language: Integer Constants

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

➢ **Examples**:

```
26                  ; decimal
26d                 ; decimal
11010011b           ; binary
42q                 ; octal
42o                 ; octal
1Ah                 ; hexadecimal
0A3h                ; hexadecimal
```

Basic Elements of Assembly Language: 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

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

Basic Elements of Assembly Language: Integer Expressions

- **Examples**:

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

Suggestion: Use parentheses in expressions **to clarify the order** of **operations** so you don't have to remember precedence rules.

Basic Elements of Assembly Language: 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'

Basic Elements of Assembly Language: Real Numbers

How about REAL numbers?

Following are examples of valid decimal reals:

```
2.
+3.0
-44.2E+05
26.E5
```

2r

Basic Elements of Assembly Language: Reserved Words

- Reserved words **cannot be used as identifiers**
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
  - See MASM reference in Appendix A

- Instruction mnemonics, such as MOV, ADD, and MUL
- Register names
- Directives, which tell the assembler how to assemble programs
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD
- Operators, used in constant expressions
- Predefined symbols, such as @data, which return constant integer values at assembly time

Basic Elements of Assembly Language: Identifiers

- They may contain between 1 and 247 characters.
- They are not case sensitive.
- The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.
- An identifier cannot be the same as an assembler reserved word.

- The following names **are legal**, but **not as desirable**
  - _ @ ? $

_lineCount $first @myFile

Basic Elements of Assembly Language: 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**, **Ex:**

.data,
.DATA,
.Data
are equivalent.

Basic Elements of Assembly Language: Directives

- **Directive .vs Instruction:**

```
myVar DWORD 26

mov eax,myVar
```

- Directive:
DWORD directive tells the assembler to reserve space in the program for a doubleword variable.

- Instruction:
MOV instruction, executes at runtime, copying myVar to the EAX register

- Different assemblers have different directives
  - NASM not the same as MASM, for example

All **assemblers** for **Intel processors** share the **same instruction set**, **BUT** they usually have different sets of directives.
Ex: Microsoft assembler's **REPT** directive **is not recognized** by some other **assemblers**.

Basic Elements of Assembly Language: Directives

See Appendix A (p-591, section A5) for more directives types

- label **PROC**

  Marks start of a procedure block called label.

- name **ENDP**

  Marks the end of procedure name previously begun with PROC.

- **INVOKE** expression [[, arguments]]

  Calls the procedure at the address given by expression, passing the arguments on the stack or in registers according to the standard calling conventions of the language type.

- **.DATA**

  indicates the start of initialized data

- **.CODE**

  indicates the start of a code segment

- **.STACK** indicates the start of a stack segment

```
1:  .data                       ; this is the data area
2:  sum DWORD 0                 ; create a variable named sum
3:
4:  .code                       ; this is the code area
5:  main PROC
6:     mov eax,5                ; move 5 to the eax register
7:     add eax,6                ; add 6 to the eax register
8:     mov sum,eax
9:
10:    INVOKE ExitProcess,0     ; end the program
11: main ENDP
```

Basic Elements of Assembly Language: Directives

Appendix A contains a useful reference for directives and operators.

Basic Elements of Assembly Language: 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)

```
        [label:] mnemonic [operands] [;comment]
repeat:   inc      result   ;increment result by 1
```

The **label** repeat can be used to **refer to this particular statement.**

Instructions: Labels

- **Act as place markers**
  - Marks the address (offset) of **code** and **data**
- Follow identifier rules

1. Code label

- target of **jump** and **loop** instructions
- **ex1**: **L1:** (followed by colon)

- **Ex2** :

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

A **code label** can **share the same line with an instruction**, or it can be **on a line by itself**:

```
L1: mov ax,bx
```

Instructions: Labels

2. Data label

- must be unique
- **Ex**: one data → **count DWORD 100** (not followed by colon)
- **Ex**: multiple data →

```
array  DWORD 1024, 2048
       DWORD 4096, 8192
```

or

```
array  DWORD 1024, 2048, 4096, 8192
```

Instructions: Mnemonics

- Instruction **Mnemonics**
  - examples: MOV, ADD, SUB, MUL, JMP, CALL

Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

Instructions: Operands

- Operands
  - constant
  - constant **expression**
  - register
  - memory (data label)

Example	Operand Type
96	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

- Constants and constant expressions are often called **immediate** values

Instructions: Operands

- **Ordering of operands**:

  When instructions have multiple operands,

  - **The first operand** is called the **destination** operand.
  - **The second operand** is called the **source** operand.

  - The contents of the **destination** operand are modified by the instruction

```
mov     destination,source
```

Operands: Instruction Format

- **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
- **Three operands**
  - imul eax,ebx,5 ; register, register, constant **(imul: signed** multiply, **preserves the sign** of the product**)**

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**
- **Multi-line comment**

---→ see next for Ex

Comments: Examples

- **Single-line comments**

```
mov eax,5                    ; move 5 to the eax register
```

- **Multi-line comments**
  - begin with **COMMENT directive** and a **programmer-chosen character**
  - end with the same **programmer-chosen character**

```
COMMENT !
     This line is a comment.
     This line is also a comment.
!
```

```
COMMENT &
     This line is a comment.
     This line is also a comment.
&
```

Outline

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

Example: Running the AddTwo Program

```
; AddTwo.asm - adds two 32-bit integers.
; Chapter 3 example

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.code
main proc
        mov     eax,5
        add     eax,6

        invoke ExitProcess,0
main endp
end main
```

Program Template

```
; Program template (Template.asm)

.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
END main
```

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
```

.386 directive, identifies this as a 32-bit program that can access 32-bit registers and addresses.

.model selects the program's memory **model (flat),** and identifies the **calling convention (named stdcall)** for procedures. Ex. Windows API

.stack aside **4096** bytes of storage for the **runtime stack**, which every program must have. Size of a memory page.

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
```

ExitProcess, declares **a prototype** for the ExitProcess function.
- A prototype consists of the function name, the PROTO keyword, a comma, and a list of input parameters.
- The input parameter for ExitProcess is named dwExitCode.

.**CODE** directive marks the beginning of the code area of a program, the area that contains executable instructions

The label main identifies the **program entry point** (main) and marks the address at which the program will begin to execute.

Example: Adding and Subtracting Integers

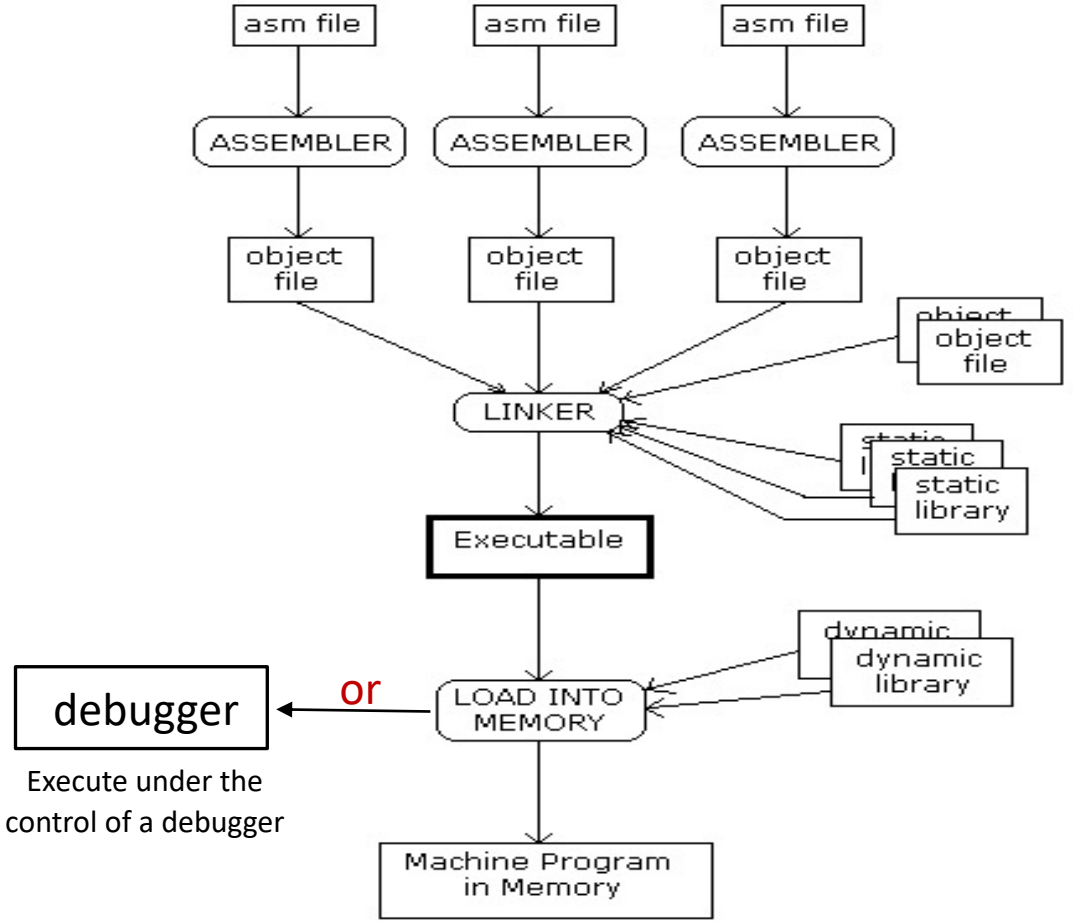
```
; AddTwo.asm - adds two 32-bit integers

.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
```

INVOKE Calls on, the procedure ExitProcess, passing the arguments on the stack or in registers.

ENDP directive marks the end of a procedure. Our program had a procedure named main,

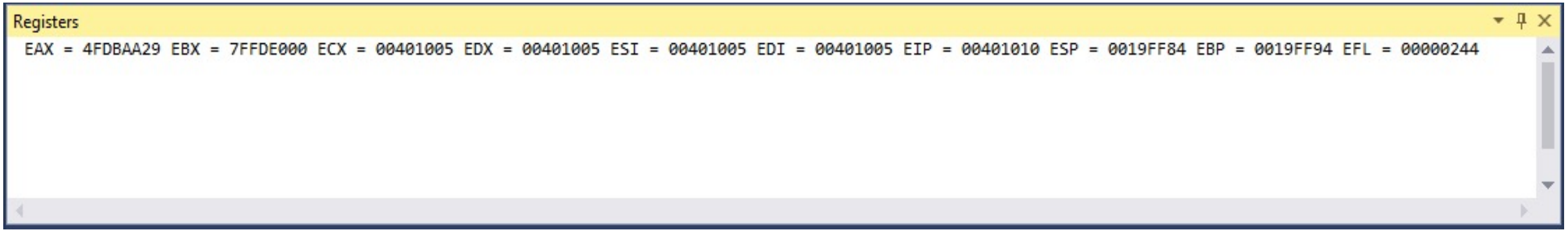
end directive marks the last line to be assembled, and it identifies the program entry point (main).

Example: Debugging the AddTwo Program



Example: Running and Debugging the AddTwo Program

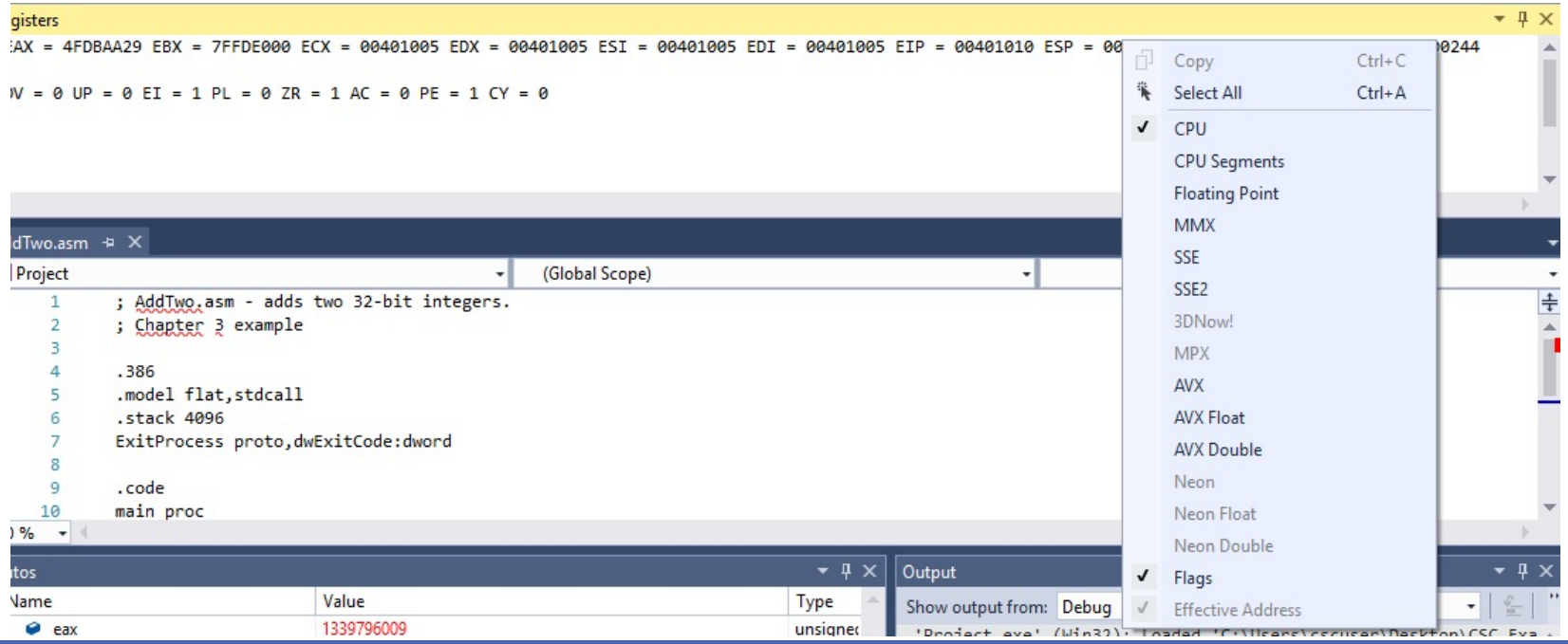
- Showing **registers** and **flags** in the debugger:
  - Go to the Debug Menu
  - Select Windows
  - Then select Registers



You Must be in the Debugging Mode to see the register window

Example: Running and Debugging the AddTwo Program

- Showing **registers** and **flags** in the debugger:
  - Right click on the Registers Window
  - Select Flags from the drop menu



- OV (overflow flag),
- UP (direction flag),
- EI (interrupt flag),
- PL (sign flag),
- ZR (zero flag),
- AC (auxiliary carry),
- PE (parity flag),
- and CY
- (carry flag).

Example: Running and Debugging the AddTwo Program

Use Visual Studio and run the program

Suggested Coding Standards

- 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

- 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

Outline

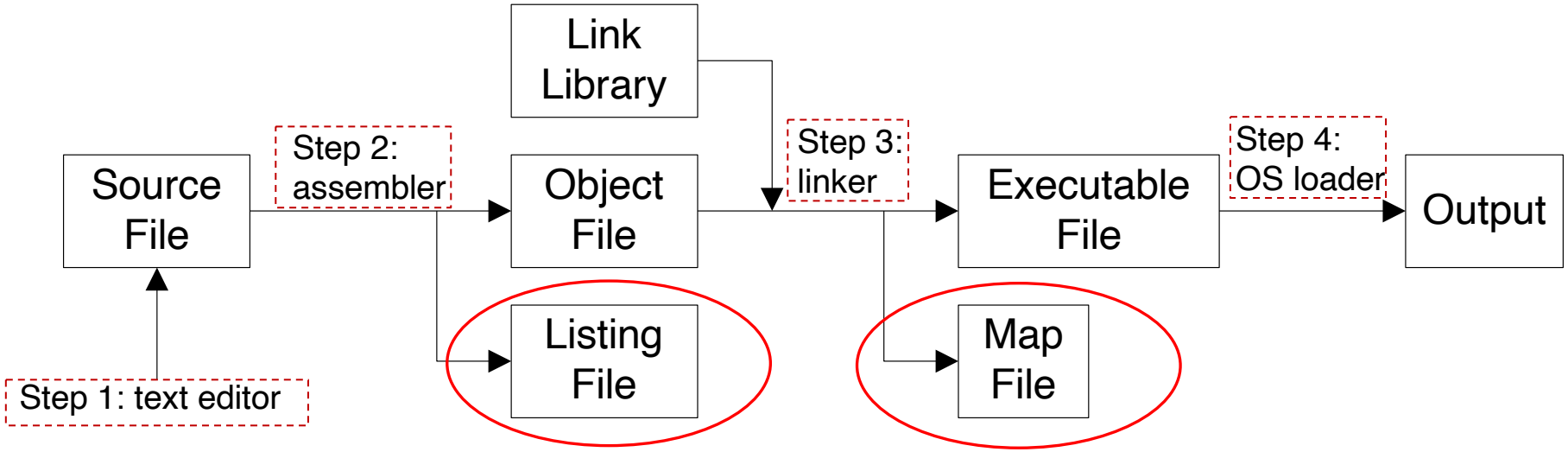
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants
- 64-Bit Programming

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.



- **How to Configure Visual Studio to generate a listing file?** 🤔

```
line#  offset  machine-code  nesting-level  source-line
```

FIGURE 3–8 Excerpt from the AddTwo source listing file.

```
 1:      ; AddTwo.asm - adds two 32-bit integers.
 2:      ; Chapter 3 example
 3:
 4:      .386
 5:      .model flat,stdcall
 6:      .stack 4096
 7:      ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:      00000000
10:      00000000
11:      00000000  B8 00000005
12:      00000005  83 C0 06
13:
14:
15:      00000008  6A 00
16:      0000000A  E8 00000000 E
17:      0000000F
18:
```

segment

addresses

object code

source code

```
.code
main PROC
    mov   eax,5
    add   eax,6

    invoke ExitProcess,0
    push   +000000000h
    call   ExitProcess
main ENDP
END main
```

Listing File

- Use it to see how your program is compiled
- Contains
  - source code
  - addresses
  - object code (machine language)
  - segment names
  - symbols (names of :variables, procedures, and constants)
- The **format** of the list file lines is

```
line#  offset   machine-code   nesting-level   source-line

11:      00000000   B8 00000005                    mov   eax,5
```

Note: Source lines such as **comments** do not generate any offset (and **what else?**) .

Listing File

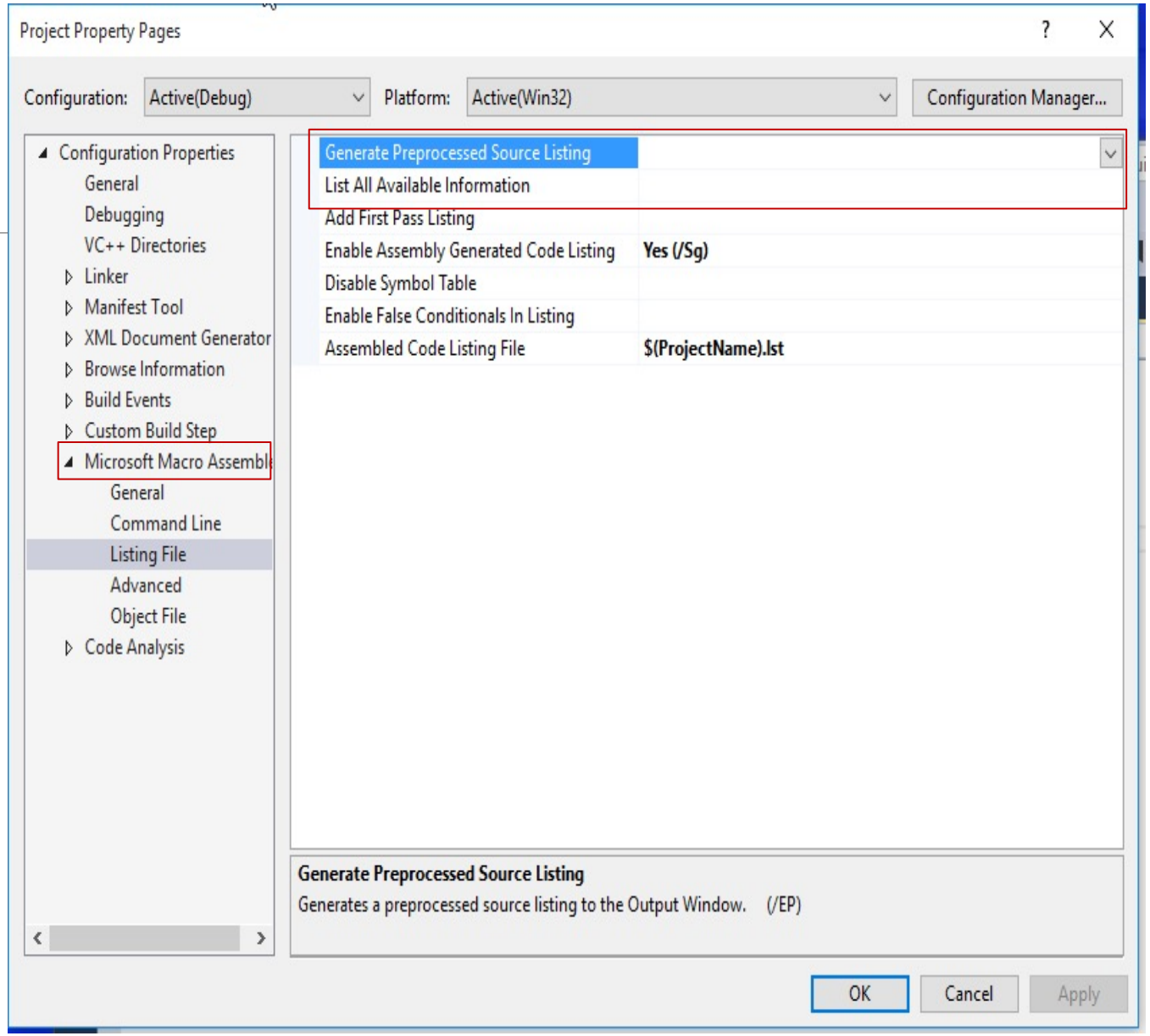
- **Line 12** also contains an executable instruction, starting at **offset** 00000005.
- **The offset** is a distance of 5 bytes from the beginning of the program.
- Can you guess **how that offset was calculated**.

Figure 3–8 Excerpt from the AddTwo source listing file.

```
 1:     ; AddTwo.asm - adds two 32-bit integers.
 2:     ; Chapter 3 example
 3:
 4:     .386
 5:     .model flat,stdcall
 6:     .stack 4096
 7:     ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:     00000000                          .code
10:     00000000                        main PROC
11:     00000000  B8 00000005              mov  eax,5
12:     00000005  83 C0 06                 add  eax,6
13:
14:                                        invoke ExitProcess,0
15:     00000008  6A 00                    push   +000000000h
16:     0000000A  E8 00000000 E            call   ExitProcess
17:     0000000F                        main ENDP
18:                                     END main
```

Listing File

- **Configuring Visual Studio to generate a listing file.**
  - Select Properties from the Project menu.
  - Under Configuration Properties, select Microsoft Macro Assembler.
    - Then select Listing File.
    - In the dialog window,
      - set Generate Preprocessed Source Listing to Yes,
      - set List All Available Information to Yes.



Map File

- **MAP File**: lists all segments in the program. The following
- **MAP File** shows code segment, data segment, and stack segment:

```
Start  Stop   Length Name     Class
00000H 0001BH 0001CH CSEG     CODE
0001CH 00035H 0001AH DSEG     DATA
00040H 0013FH 00100H SSEG     STACK

Program entry point at 0000:0000
```

Outline

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

Defining Data: Outline

- **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

Defining Data: Intrinsic Data Types

Table 3-2 Intrinsic Data Types.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Table 3-3 Legacy Data Directives.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

Defining Data: Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- Assign a name (label) to the data

Syntax:

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

Note:

- All initializers become **binary data** in memory
- At least **one initializer is required** in a data definition, even if it is zero.
- Additional initializers, if any, **are separated by commas**.
- If you prefer to leave the variable uninitialized (assigned **a random value),** the **?** symbol can be used as the initializer.

Defining Data: Adding a Variable to the AddTwo Program

```
 1: ; AddTwoSum.asm - Chapter 3 example
 2:
 3:     .386
 4:     .model flat,stdcall
 5:     .stack 4096
 6:     ExitProcess PROTO, dwExitCode:DWORD
 7:
 8:     .data
 9:     sum DWORD 0
10:
11:     .code
12:     main PROC
13:         mov eax,5
14:         add eax,6
15:         mov sum,eax
16:
17:         INVOKE ExitProcess,0
18:     main ENDP
19:     END main
```

Use Visual Studio to run this program

Midterm: 2/24/21 (No Lecture) ~~Tues~~ (Thursday)

9:00 AM to 11:59 PM

→ 8 questions — parts (a), (b), (c)

→ closed book.

→ Notes ~~for~~ your own. ⇒

→ Calculator.

→ Answer in a paper ⇒ (optional)

→ Study Guide ⇒

⇒ Mock Midterm → (Question) optional

Defining Data: 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 Data: Defining Byte Arrays

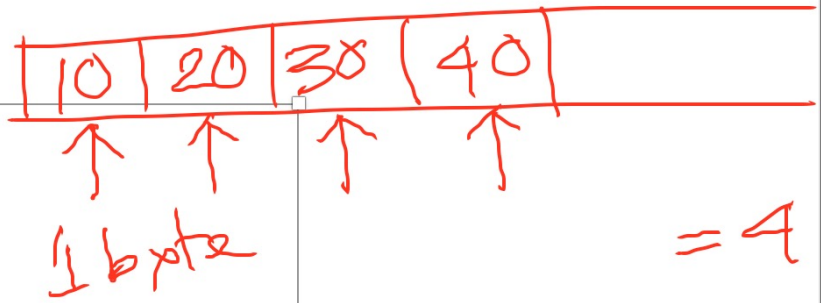
- Examples that use multiple initializers:

```
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 Data: 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
```



size of each item in the array.

Java: float a[4] = {1.2, 1.3, 1.4, 1.5}

Defining Data: 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

Examples:

```
str1 BYTE "Enter your name"
str2 BYTE 'Error: halting program',0
```

```
str3 BYTE 'A','E','I','O','U'
```

Defining Data: Using the DUP Operator

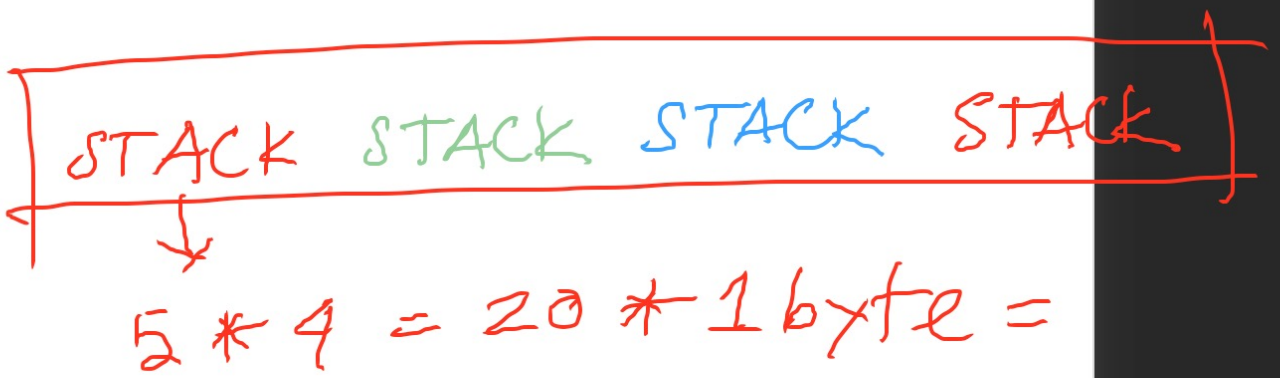
- Use **DUP** to allocate (create space for) an array or string.
- Syntax: *counter* **DUP (*argument*)**
- *Counter* and *argument* must be constants or constant expressions

```
var1 BYTE 20 DUP(0)
var2 BYTE 20 DUP(?)
var3 BYTE 4 DUP("STACK")
var4 BYTE 10,3 DUP(0),20
```

Defining Data: 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)
var2 BYTE 20 DUP(?)
var3 BYTE 4 DUP("STACK")
var4 BYTE 10,3 DUP(0),20
```



Defining Data: 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 Data: Defining DWORD and SDWORD Data

- Storage definitions for signed and unsigned 32-bit integers:

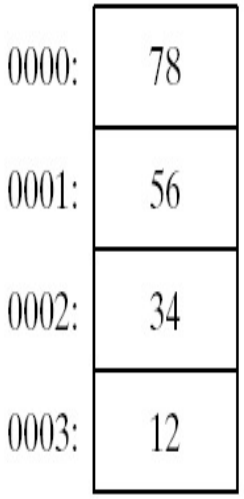
```
val1 DWORD  12345678h          ; unsigned
val2 SDWORD –2147483648        ; signed
val3 DWORD  20 DUP(?)          ; unsigned array
val4 SDWORD –3,–2,–1,0,1       ; signed array
```

Defining Data: Little Endian Order (X86 uses it)

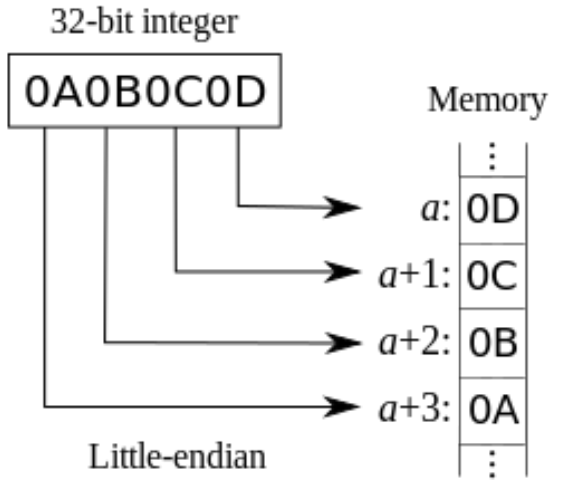
- All data types larger than a byte store their individual bytes in **reverse order**.
- The **least significant byte** occurs at **the first (lowest) memory address**.

Example1:

```
val1 DWORD 12345678h
```

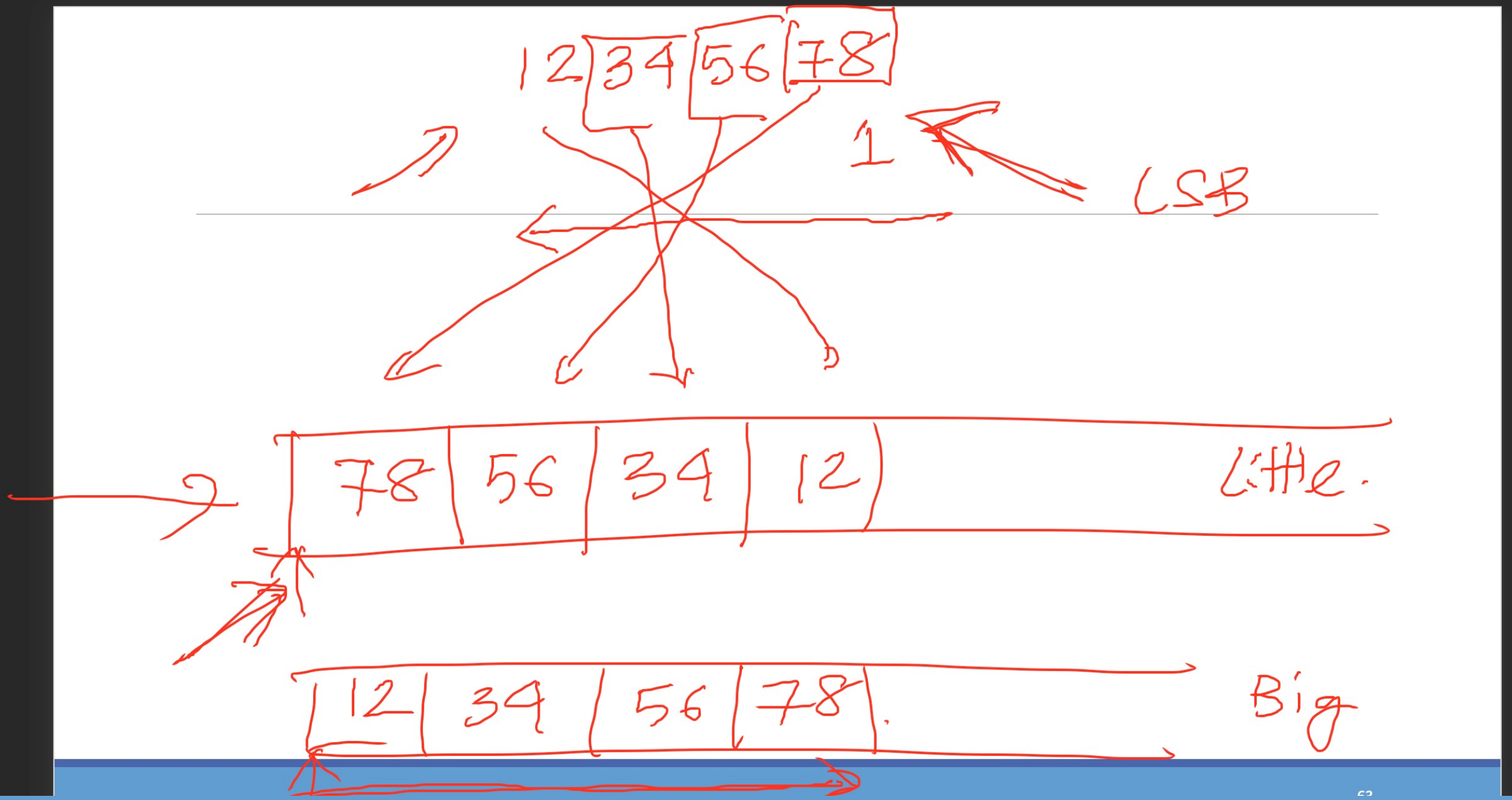


Example2:



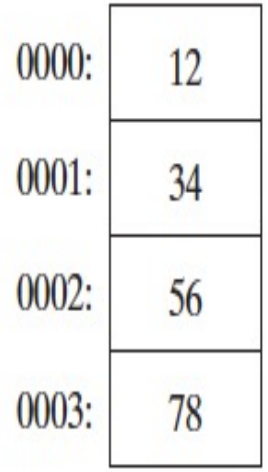
What is a?

Why plus 1?


12 34 56 78
1
LSB
78 56 34 12
Little.
12 34 56 78
Big

Defining Data: Big Endian Order

FIGURE 3–15 Big-endian representation of 12345678h.



Attendance!

Defining Data: Adding Variables to AddSub

```
 1: ; AddVariables.asm - Chapter 3 example
 2:
 3:    .386
 4:    .model flat,stdcall
 5:    .stack 4096
 6:    ExitProcess PROTO, dwExitCode:DWORD
 7:
 8:    .data
 9:    firstval  DWORD 20002000h
10:    secondval DWORD 11111111h
11:    thirdval  DWORD 22222222h
12:    sum       DWORD 0
13:
14:    .code
15:    main PROC
16:       mov eax,firstval
17:       add eax,secondval
18:       add eax,thirdval
19:       mov sum,eax
20:
21:       INVOKE ExitProcess,0
22:    main ENDP
23:    END main
```

Defining Data: Adding Variables to AddSub

Use Visual Studio and run the program

Lecture-10-Chapter3-Assembly-Language-fundamentals-with-annotations

Design Transitions Animations Slide Show Review View Acrobat

Layout Reset Section

Calibri (Body) 20

Defining Dat

```
1:  ; AddVariables.asm -
2:
3:      .386
4:      .model flat,stdca
5:      .stack 4096
6:      ExitProcess PROTO
7:
8:      .data
9:      firstval   DWORD 2
10:     secondval  DWORD 1
```

ExamplAddTwoNumbers program
on the desktop

You are screen sharing Stop Share

windows10_fall21 [Running]

Search in Presentation

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q)

Debug x86 Local Windows Debugger

Server Explorer Toolbox

Project4.lst Source.asm*

```
1   .386
2   .MODEL flat, stdcall
3   .STACK 4096
4   ExitProcess Proto, dwExitCode:DWORD
5
6   .DATA
7   var1 DWORD 10
8   var2 DWORD 11
9   var3 DWORD 12
10
11
12  .CODE
13  main PROC
14      mov eax, var2
15      sub eax, var3
16
17      add eax, var1
18
19      mov var1,eax
20
21      INVOKE ExitProcess, 0
22  main ENDP
```

var1 = var1 + (var2 - var3)

* Rule:

eax = (var2 - var3)

eax = var1 + (var2 - var3)

var1

ADD, MOV, SUB

CANNOT TAKE TWO variables AS INPUT.

Atleast one Register needed

110 % No issues found Ln: 19 Ch: 15

Error List Output

Entire Solution 3 Errors 0 Warnings 0 Messages Build + IntelliSense Search Error

Code	Description	Project	File	Line
A2070	invalid instruction operands	Project4	Source.asm	14
A2070	invalid instruction operands	Project4	Source.asm	15
MSB3721	The command "ml.exe /c /nologo /Zi /Fo"Debug\Source.obj" /W3 /errorReport:prompt /TaSource.asm" exited with code 1.	Project4	masm.targets	70

Ready

Defining Data: Declaring Uninitialized Data

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

.data?

```
.data
smallArray DWORD 10 DUP(0)      ; 40 bytes
bigArray DWORD 5000 DUP(?)      ; 20,000 bytes
```

.VS

```
.data
smallArray DWORD 10 DUP(0)      ; 40 bytes
.data?
bigArray DWORD 5000 DUP(?)      ; 20,000 bytes, not initialized
```

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

Outline

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

Symbolic **Constants** (General)

- A symbolic constant is name that substitute for a sequence of character that **cannot be changed during run time**.
- The character may represent
  - a **numeric** constant
  - a **character** constant
  - or a **string**

Pi = 3.1415926535

```
final double Pi = 3.1415926535;
final double E = 2.718281828;
```

Symbolic **Constants** (MSAM)

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

Good programming style to use **symbols**

Symbolic **Constants**

- **A symbolic constant** (or symbol definition) is created by associating an identifier (a symbol) with
  - an **integer** expression
  - or some **text**.

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

Why a symbolic constant does not reserve a storage? 🤔

```
COUNT = 500

mov eax, COUNT
```

When the file is assembled, MASM will scan the source file and produce the corresponding code lines:

```
mov eax, 500
```

Symbolic Constants: 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

Ex.

```
COUNT = 500

mov eax, COUNT
```

Symbolic Constants: Equal-Sign Directive

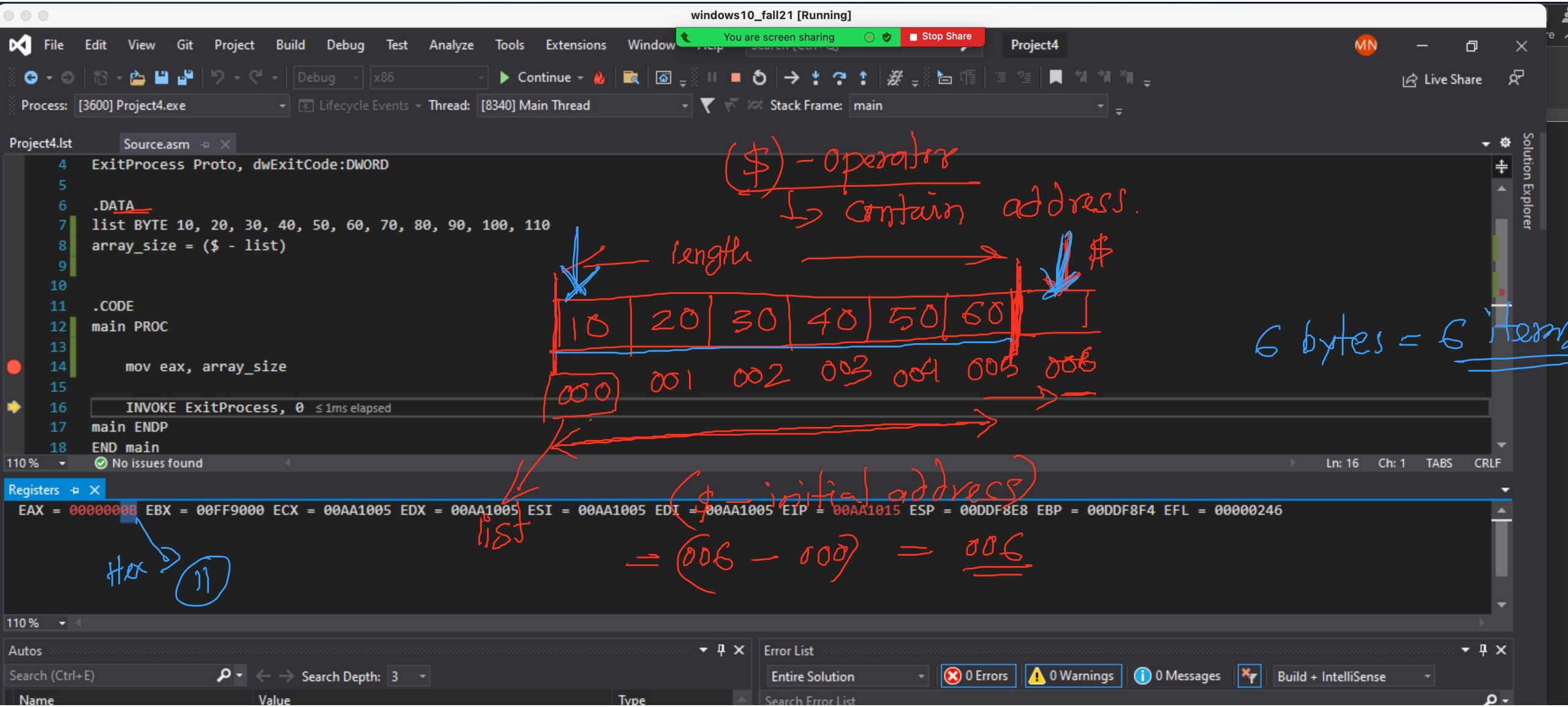
- *name* = *expression*
  - expression is a **32-bit integer** (expression or constant)
  - may be redefined
  - *name* is called a symbolic constant
- Good programming style to use symbols

Ex.

```
COUNT = 500

mov eax, COUNT
```

count = (4 * 3 / 6)

expression with all integers or Any number


windows10_fall21 [Running]
You are screen sharing
Stop Share
Project4
Live Share
Process: [3600] Project4.exe
Thread: [8340] Main Thread
Stack Frame: main
Project4.lst
Source.asm
4 ExitProcess Proto, dwExitCode:DWORD
5
6 .DATA
7 list BYTE 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110
8 array_size = ($ - list)
9
10
11 .CODE
12 main PROC
13
14 mov eax, array_size
15
16 INVOKE ExitProcess, 0 ≤ 1ms elapsed
17 main ENDP
18 END main
110 %
No issues found
Ln: 16 Ch: 1 TABS CRLF
($) - Operator
→ Contain address.
length
10 20 30 40 50 60
000 001 002 003 004 005 006
$
6 bytes = 6 items
list
($ - initial address)
= (006 - 000) = 006
Registers
EAX = 0000000B EBX = 00FF9000 ECX = 00AA1005 EDX = 00AA1005 ESI = 00AA1005 EDI = 00AA1005 EIP = 00AA1015 ESP = 00DDF8E8 EBP = 00DDF8F4 EFL = 00000246
Hex → 11
110 %
Autos
Search (Ctrl+E)
Search Depth: 3
Name
Value
Type
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Build + IntelliSense
Search Error List

Equal-Sign Directive: Calculating the Size of a Byte Array

- When using an array, we usually like to know its size.
- The following example uses a constant named ListSize to declare the size of list:

```
list BYTE 10,20,30,40

ListSize = 4
```

- An array's size can lead to a programming error,
- ex., if you should later **insert** or **remove** array elements.

- A better way to declare an array size is to **let the assembler calculate its value for you**.
- **How?**
- The $ operator (current location counter)

Equal-Sign Directive: Calculating the Size of a Byte Array

- **$** operator (**current location counter**) **returns the offset** associated with the current program statement.

Ex.

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

- Current location counter: **$**
  - Subtract address of list
  - Difference is the number of bytes

- How about this (when $ location changes)?

```
list BYTE 10,20,30,40
var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

- Produces too large value (**24**) for ListSize?
- Because the storage used by **var2** affects the **distance** between the **current location** counter and **the offset** of list

Equal-Sign Directive: Calculating the Size of a Byte Array

- How about **calculating the length of a string manually**

```
myString  BYTE "This is a long string, containing"
          BYTE "any number of characters"
myString_len = ???????
```

- Let the assembler do it:

```
myString  BYTE "This is a long string, containing"
          BYTE "any number of characters"
myString_len = ($ - myString)
```

Equal-Sign Directive: 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
```

- How about **DWORD**

```
list   DWORD   10000000h,20000000h,30000000h,40000000h
ListSize = ($ - list) / 4
```

Attendance!

Outline

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

64-Bit Programming

- MASM supports 64-bit programming,
- The following directives are not permitted:
  - .386, .model, .stack (what? No stack 😇), INVOKE

```
 1: ; AddTwoSum.asm - Chapter 3 example
 2:
 3:    .386
 4:    .model flat,stdcall
 5:    .stack 4096
 6:    ExitProcess PROTO, dwExitCode:DWORD
 7:
 8:    .data
 9:    sum DWORD 0
10:
11:    .code
12:    main PROC
13:       mov eax,5
14:       add eax,6
15:       mov sum,eax
16:
17:       INVOKE ExitProcess,0
18:    main ENDP
19:    END main
```

64-Bit Version of AddTwoSum

```
1: ; AddTwoSum_64.asm - Chapter 3 example.
2: ?????
3: ExitProcess PROTO ?????
4:
5: .data
6: sum DWORD 0
7:
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 Programming

- Use 64-bit registers when possible:

```
 6: sum QWORD 0
 7:
 8: .code
 9: main PROC
10:     mov  rax,5
11:     add  rax,6
12:     mov  sum,rax
```

```
 1: ; AddTwoSum.asm - Chapter 3 example
 2:
 3:     .386
 4:     .model flat,stdcall
 5:     .stack 4096
 6:     ExitProcess PROTO, dwExitCode:DWORD
 7:
 8:     .data
 9:     sum DWORD 0
10:
11:     .code
12:     main PROC
13:         mov eax,5
14:       add eax,6
15:       mov sum,eax
16:
17:         INVOKE ExitProcess,0
18:     main ENDP
19:     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