

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

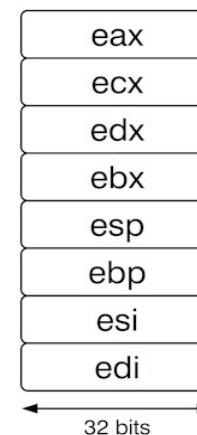
# Basic Language Elements

- **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
```

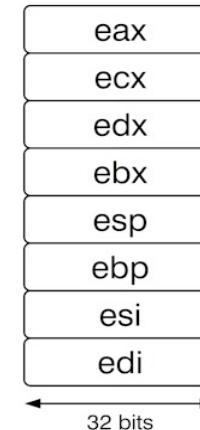


# Basic Language Elements

- 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
```



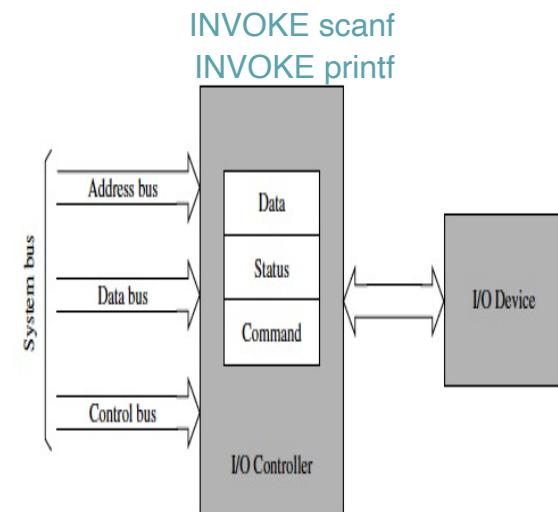
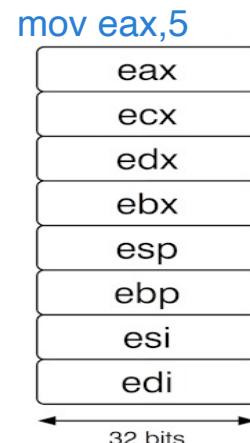
### AddTwo program:

```
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
```

Byte	8
Word	16
Doubleword	32
Quadword	64
Double quadword	128

# 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 <b>Integer division?</b>	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)
- Constants and constant expressions are often called **immediate values**

Example	Operand Type
96	<i>Integer literal</i>
$2 + 4$	Integer expression
eax	Register
count	Memory

# 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
```

Processor  
version 88  
mode).

Main Memory

Operating System function call

stack size. (bytes)

```

Source.asm* 3 → 32-bit
1 .386
2 .MODEL flat, stdcall
3 .STACK 4096
4 ExitProcess Proto, dwExitCode:DWORD ×86
5
6 .DATA
7 array1 DWORD 12, 13, 14, 15
8
9 .CODE
10 main PROC
11     mov eax, 10 ; this is a comment
12     inc eax
13
14     INVOKE ExitProcess, 0
15 main ENDP
16 END main
17

```

Processor  
version 88  
mode).

Main Memory

Operating System function call

stack size. (bytes)

# 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.

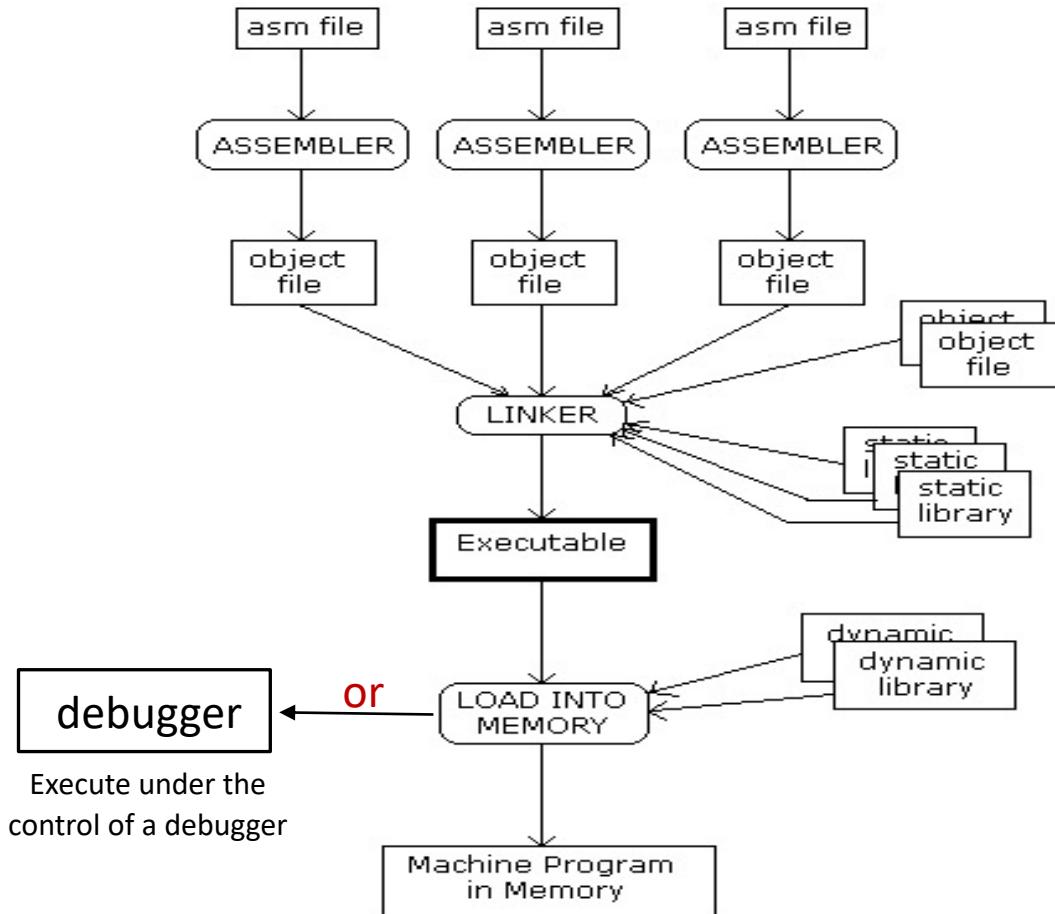
**ENDP** directive marks the end of a procedure. Our program had a procedure named main,

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

# Attendance!

---

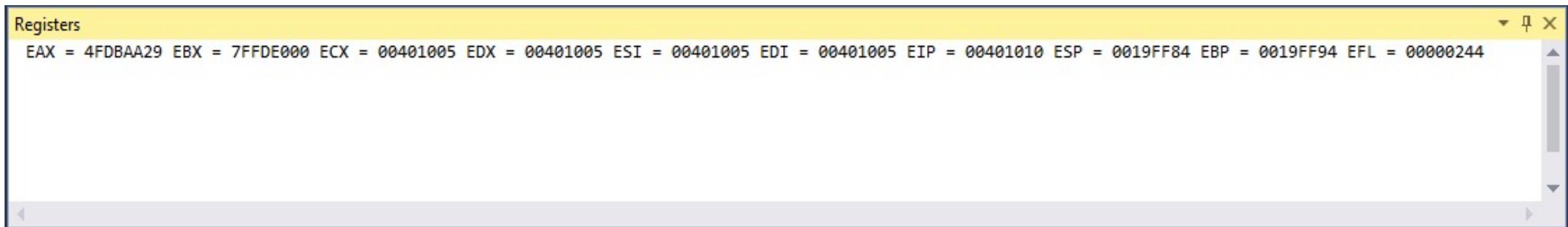
# Example: Debugging the AddTwo Program



## Example: Running and Debugging the AddTwo Program

---

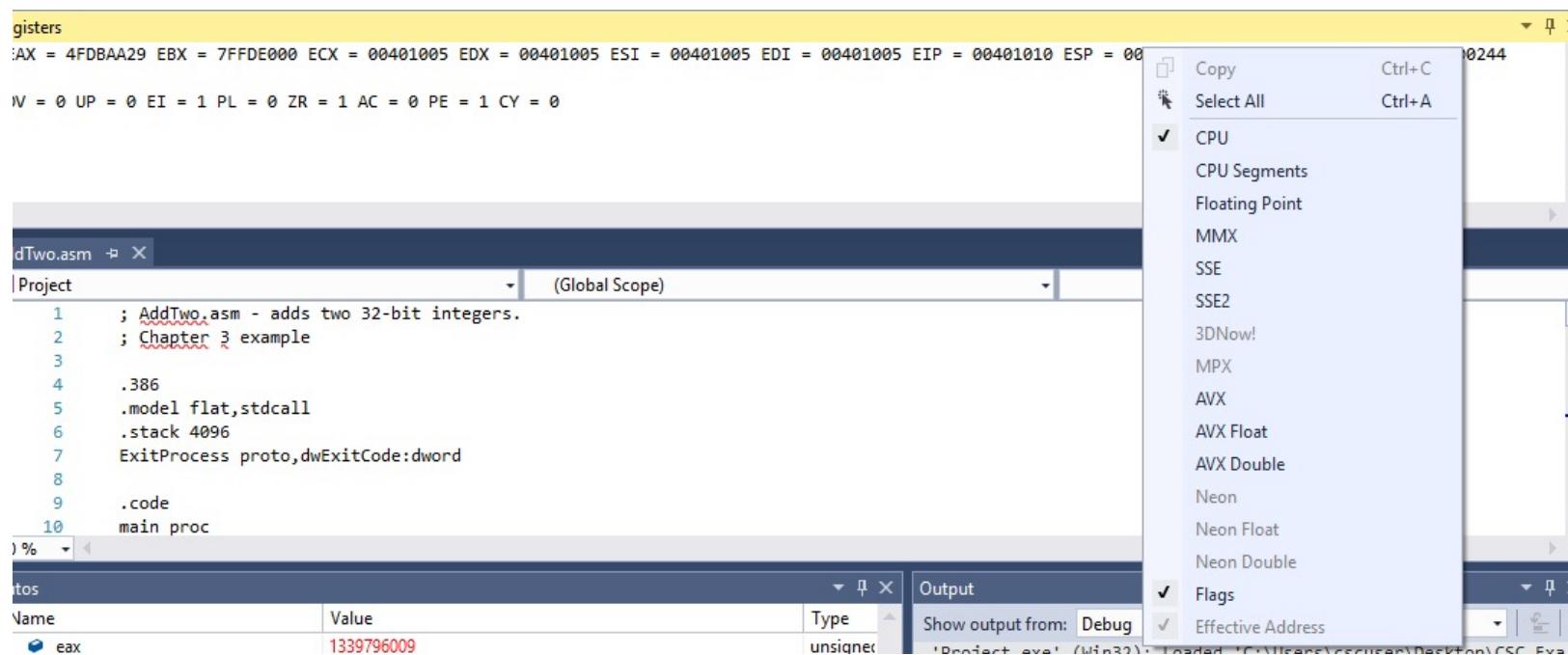
- Showing **registers** and **flags** in the debugger:
  - Go to the Debug Menu
  - Select Windows
  - Then select Registers



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

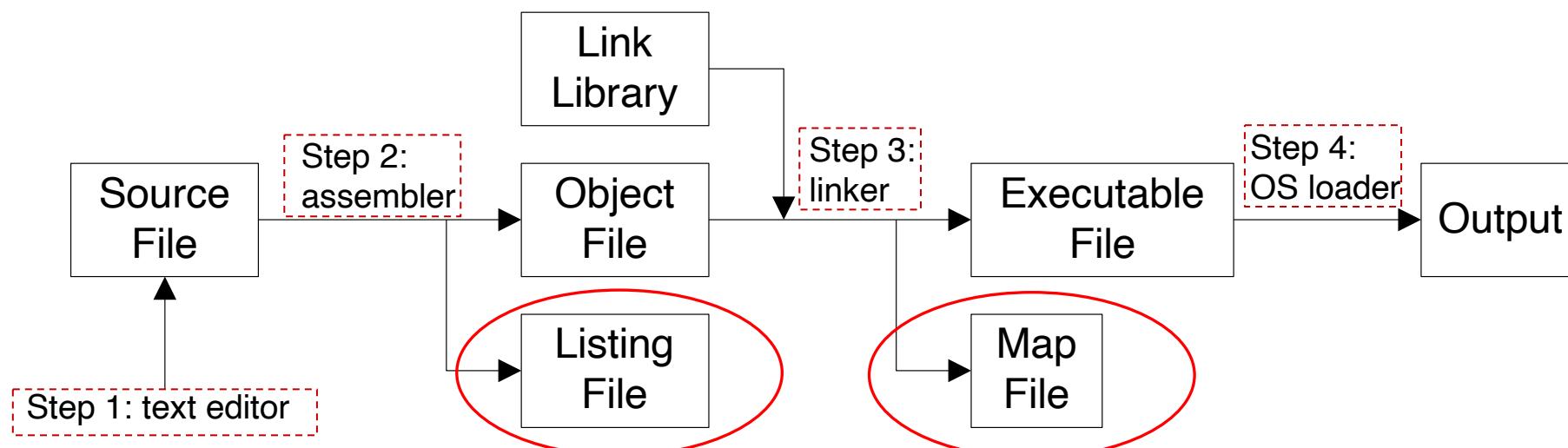
# Assembling, Linking, and Running Programs

---

- 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    segment  
7:      ExitProcess PROTO,dwExitCode:DWORD  
8:      addresses  
9:      00000000  
10:     00000000  
11:     00000000  
12:     00000005  
13:  
14:  
15:     00000008  
16:     0000000A  
17:     0000000F  
18:
```

object code

```
B8 00000005  
83 C0 06  
  
6A 00  
E8 00000000 E
```

source code

```
.code  
main PROC  
    mov eax, 5  
    add eax, 6  
  
    invoke ExitProcess, 0  
    push +00000000h  
    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

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

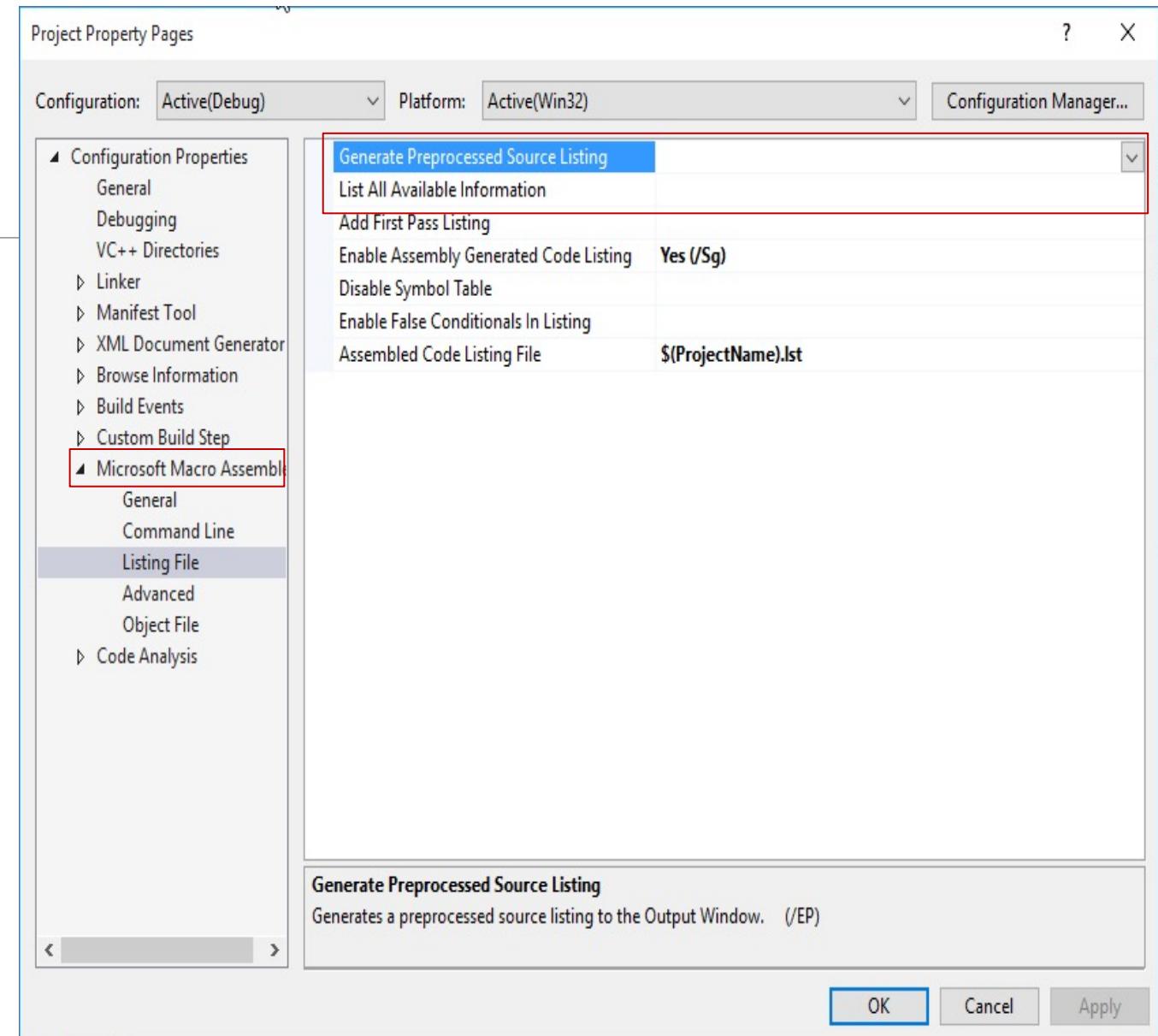
- Line 12 also contains an executable instruction, starting at **offset** 00000005.
- The offset is a distance of 5 bytes from the beginning of the program.
- Can you guess **how that offset was calculated**.

```
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  +00000000h
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:

<b>Start</b>	<b>Stop</b>	<b>Length</b>	<b>Name</b>	<b>Class</b>
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] ...*

↓      ↓      ↓  
**value1** **BYTE** **10**

## 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

# 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 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",0
```

```
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: 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: Defining QWORD, Real Data, TBYTE

---

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

quad1 **QWORD** 1234567812345678h → (12 34 56 78 12 34 56 78 h)

rVal1 **REAL4** -2.1

rVal2 **REAL8** 3.2E-260

rVal3 **REAL10** 4.6E+4096

ShortArray **REAL4** 20 DUP(0.0)

