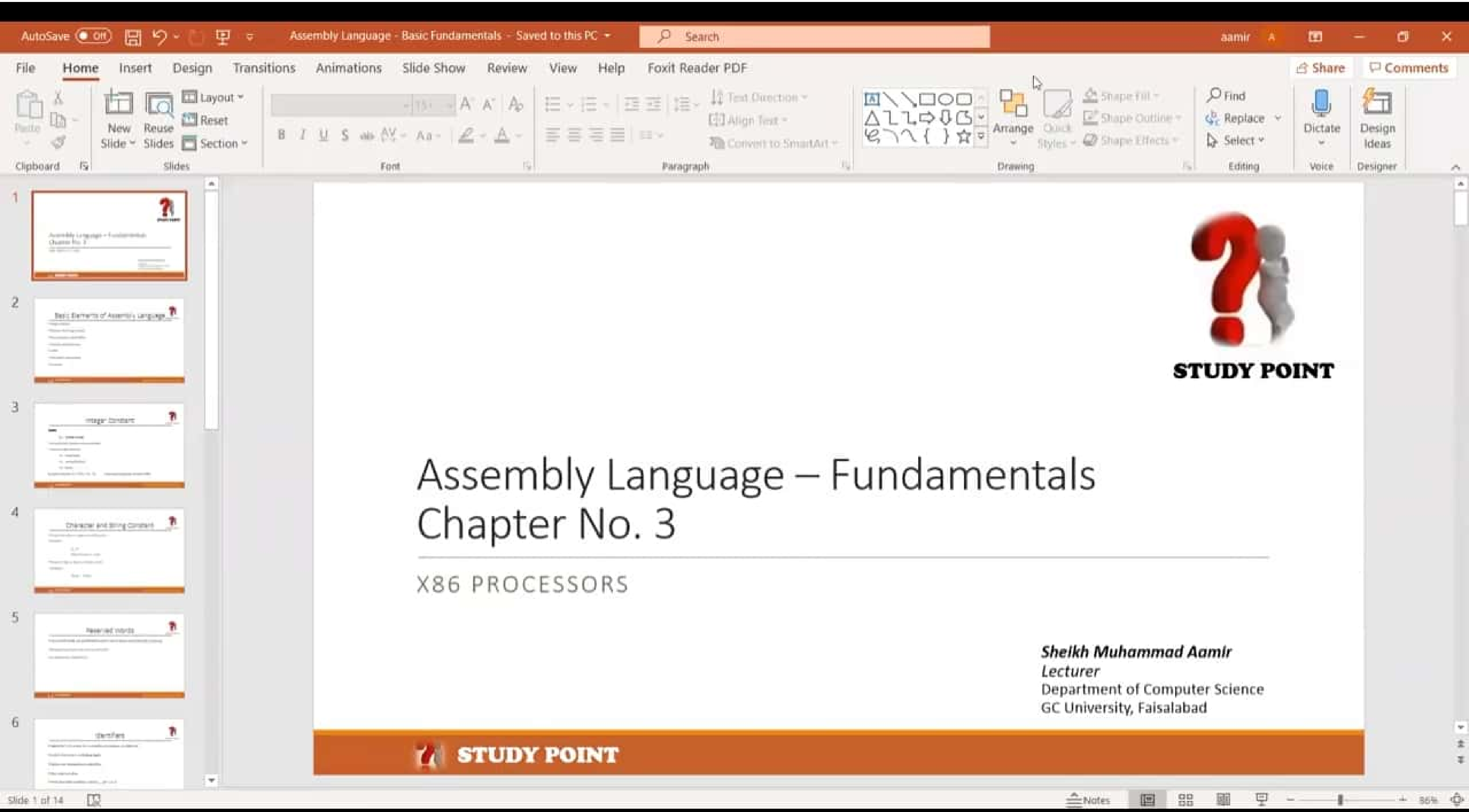


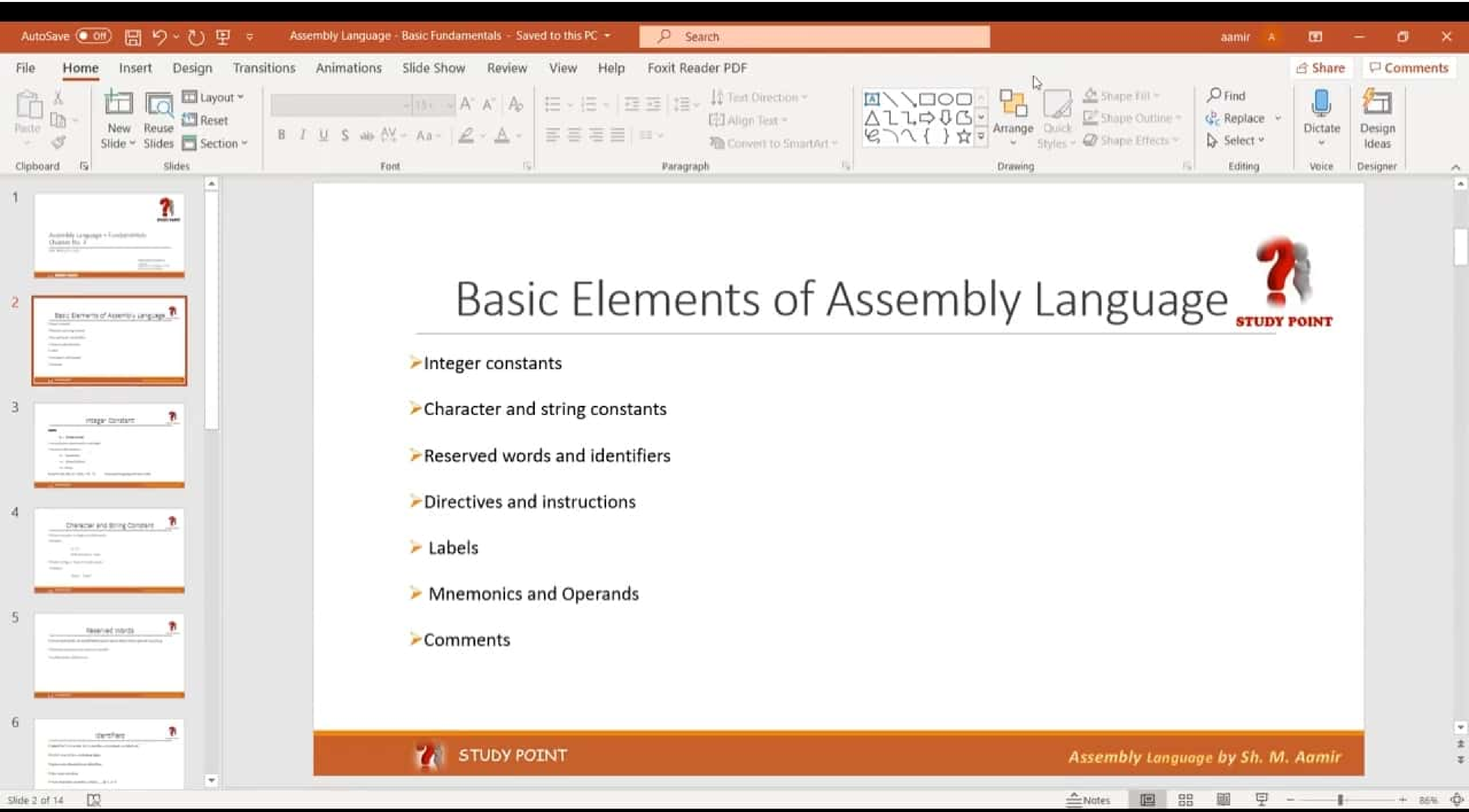
STUDY POINT

Assembly Language – Fundamentals Chapter No. 3

X86 PROCESSORS

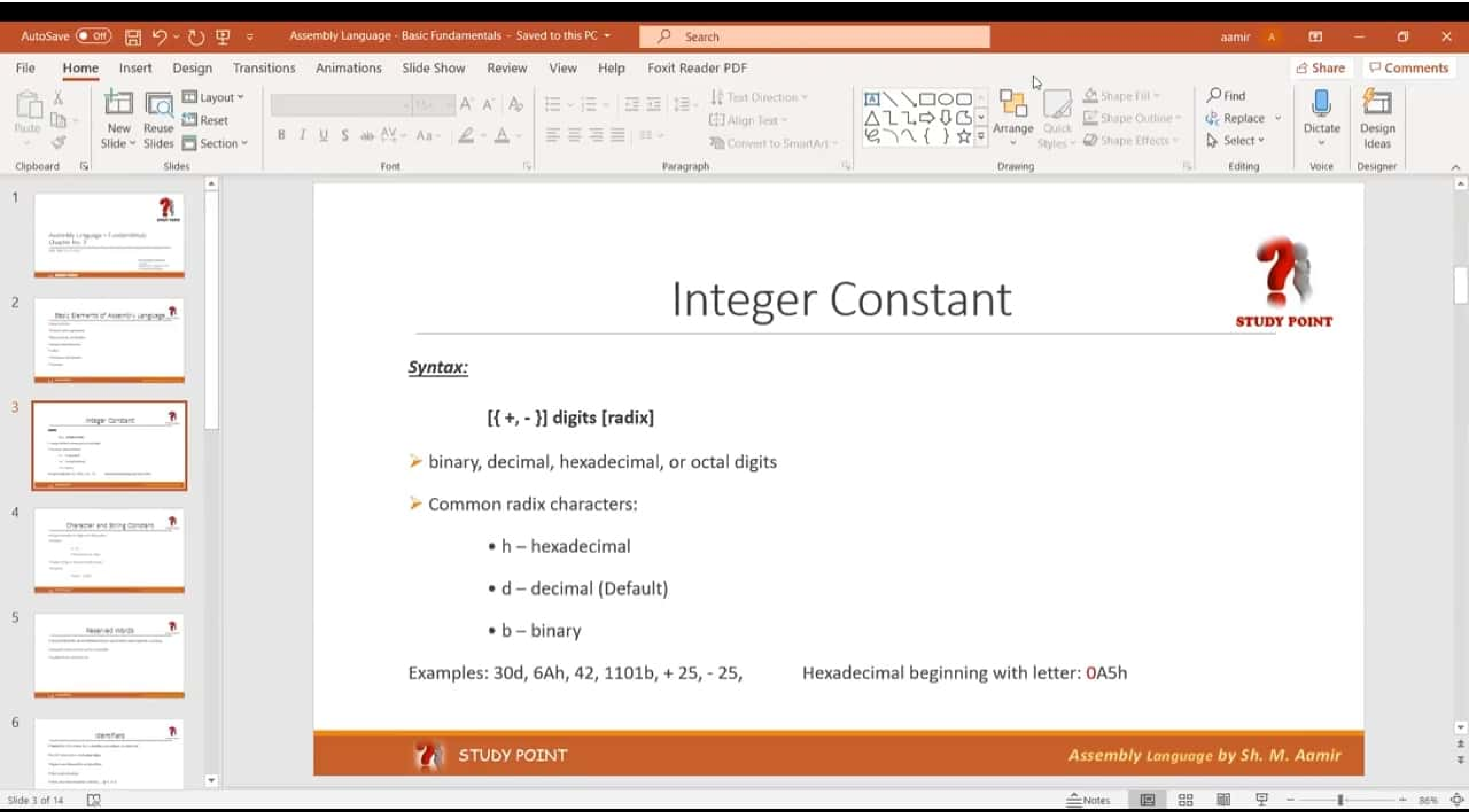
Sheikh Muhammad Aamir
Lecturer
Department of Computer Science
GC University, Faisalabad

STUDY POINT



Basic Elements of Assembly Language

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



Integer Constant

STUDY POINT

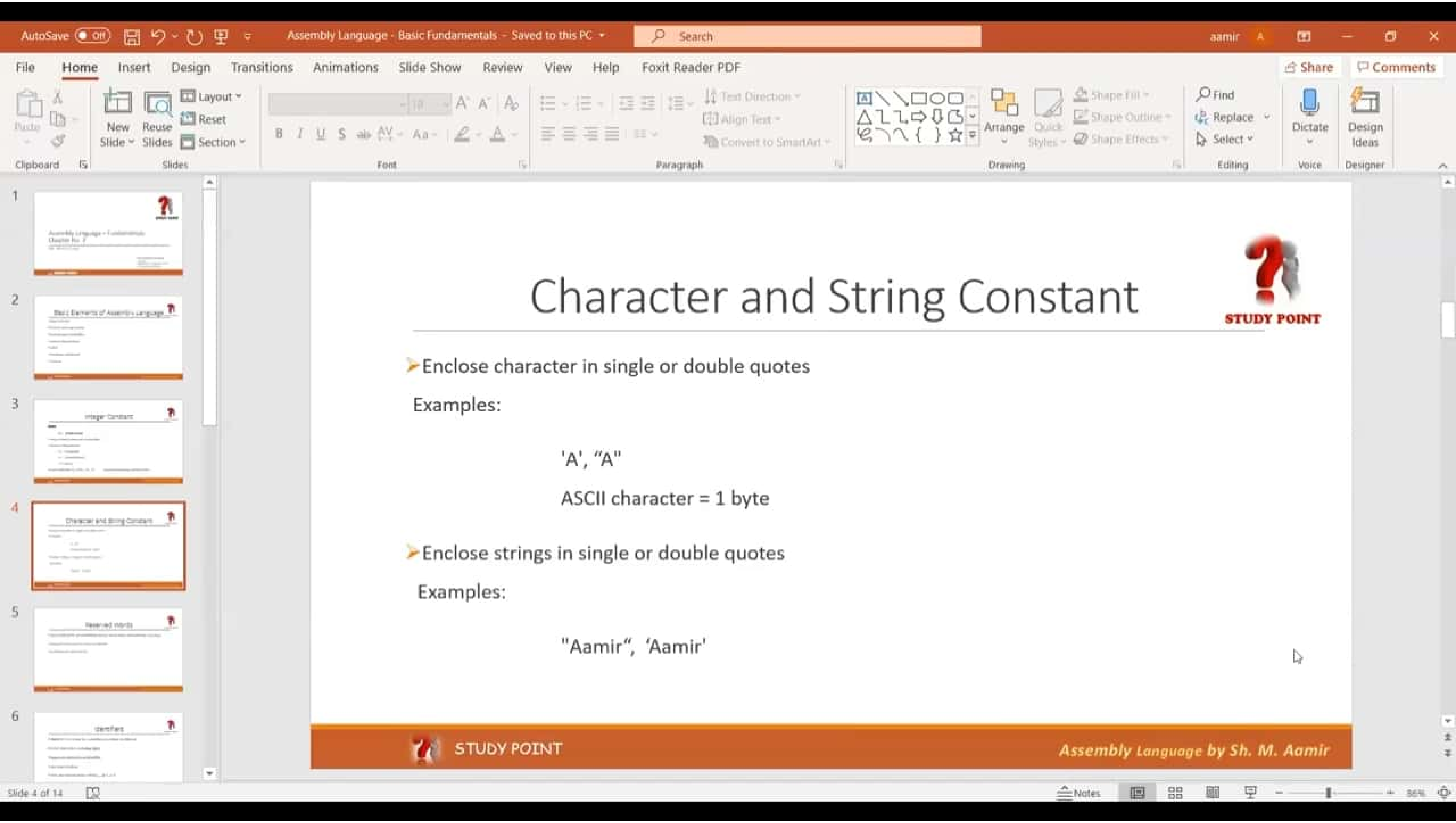
Syntax:

[{ +, - }] digits [radix]

- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - d – decimal (Default)
  - b – binary

Examples: 30d, 6Ah, 42, 1101b, + 25, - 25,   Hexadecimal beginning with letter: 0A5h

STUDY POINT   *Assembly Language by Sh. M. Aamir*



Character and String Constant

- Enclose character in single or double quotes

Examples:

'A', "A"

ASCII character = 1 byte

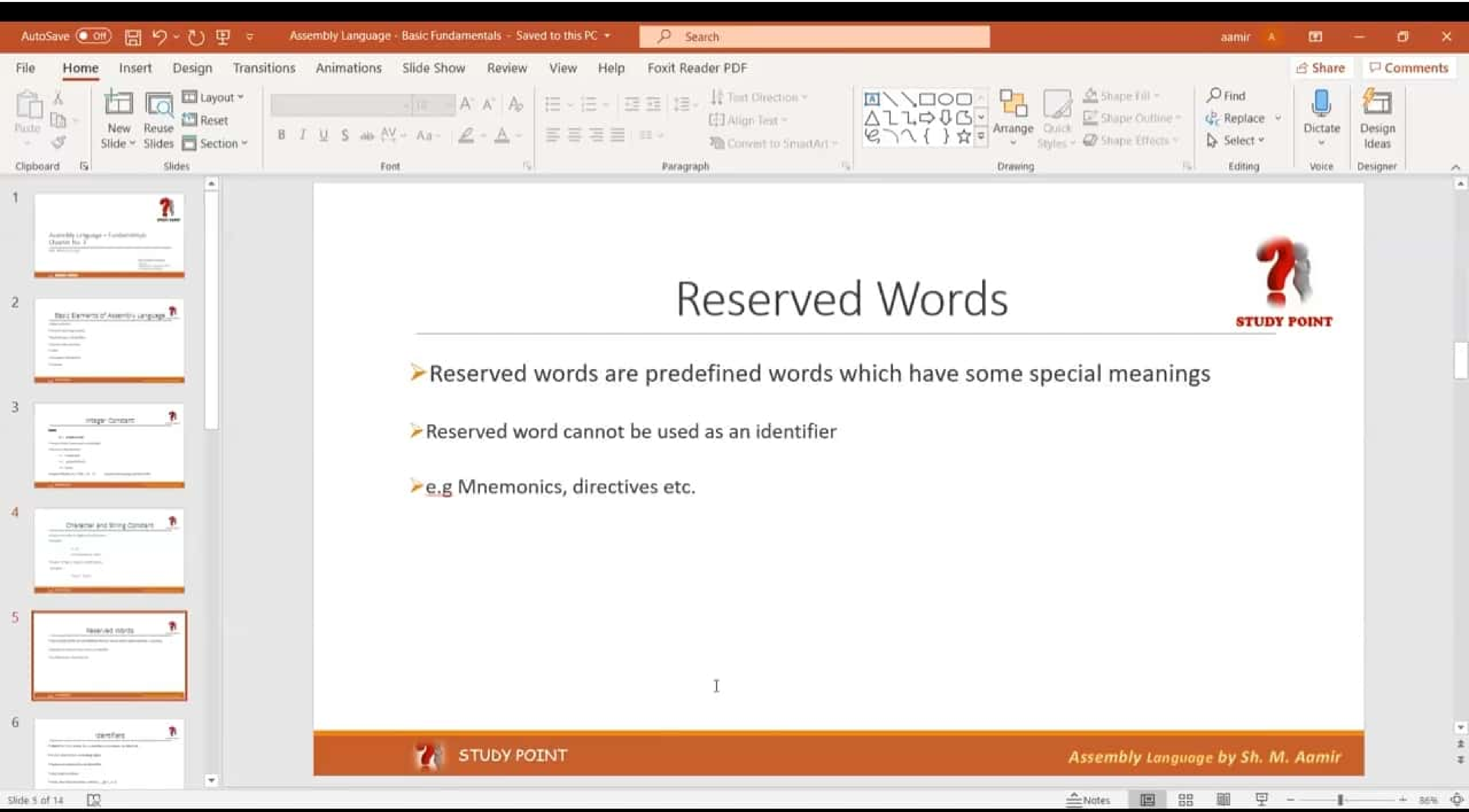
- Enclose strings in single or double quotes

Examples:

"Aamir", 'Aamir'

STUDY POINT

Assembly Language by Sh. M. Aamir



Reserved Words

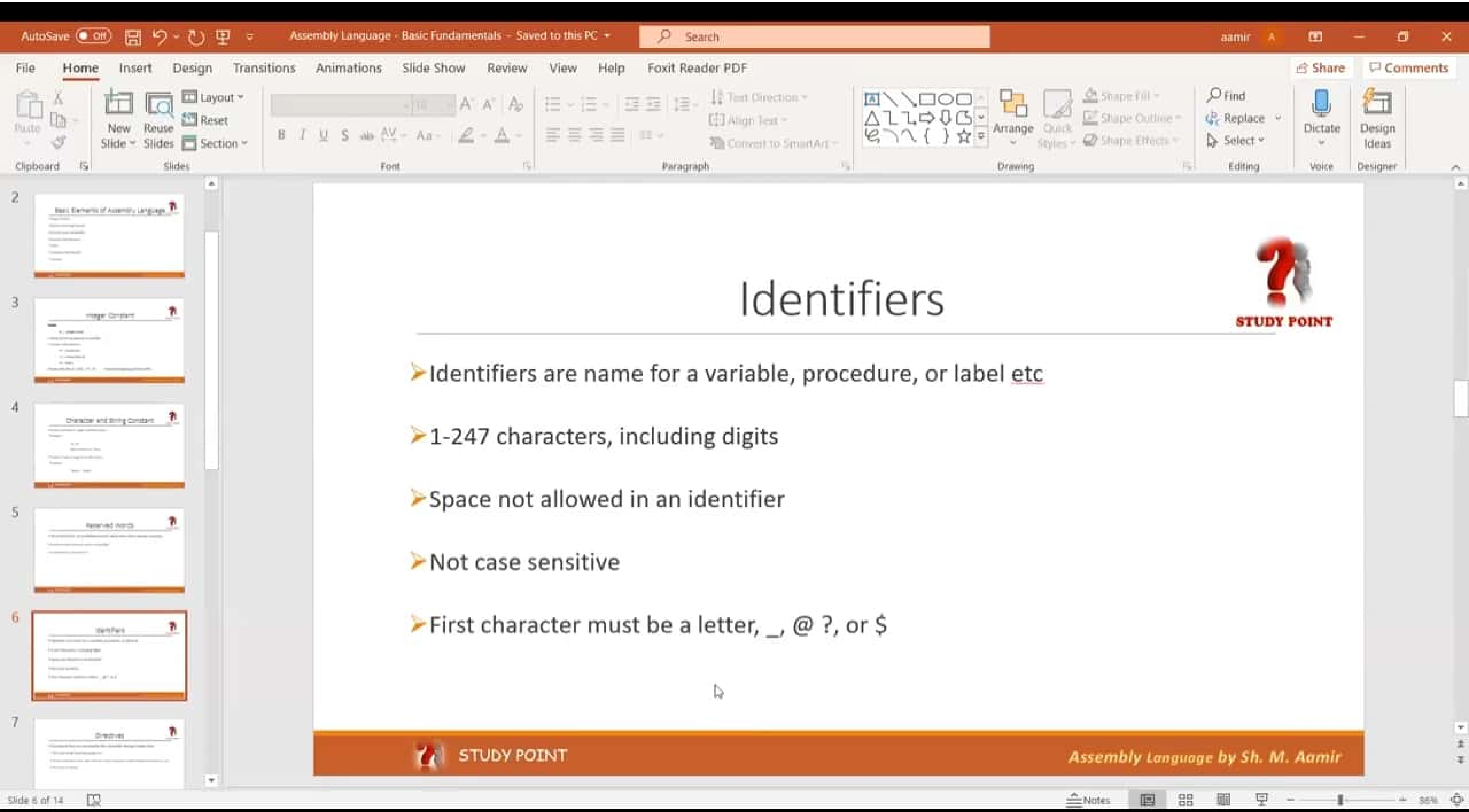
STUDY POINT

- Reserved words are predefined words which have some special meanings
- Reserved word cannot be used as an identifier
- e.g Mnemonics, directives etc.

STUDY POINT

Assembly Language by Sh. M. Aamir

Scanned with CamScanner

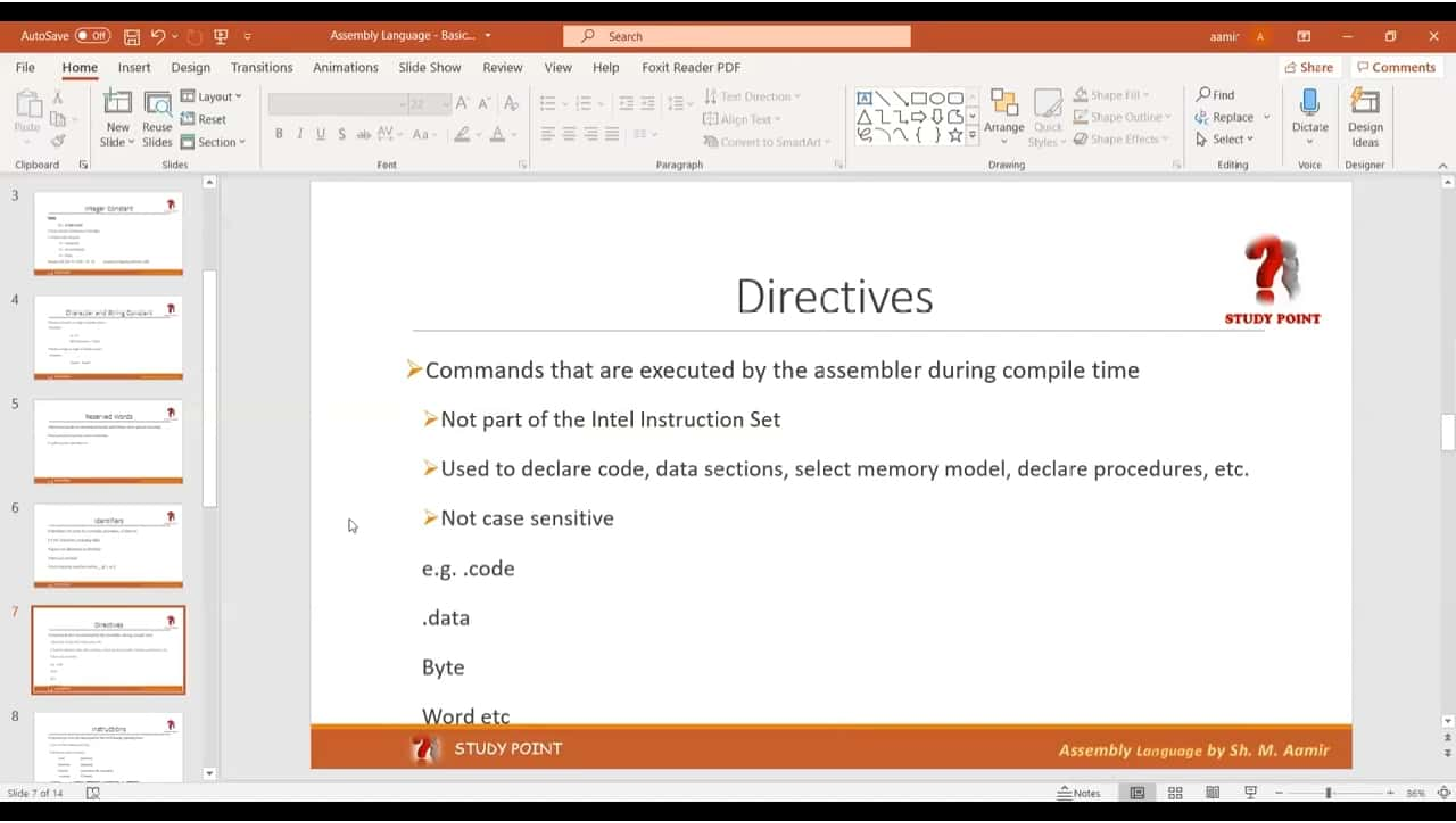


Identifiers

- Identifiers are name for a variable, procedure, or label etc
- 1-247 characters, including digits
- Space not allowed in an identifier
- Not case sensitive
- First character must be a letter, _, @ ?, or $

STUDY POINT

Assembly Language by Sh. M. Aamir



Directives

STUDY POINT

- Commands that are executed by the assembler during compile time
  - Not part of the Intel Instruction Set
  - Used to declare code, data sections, select memory model, declare procedures, etc.
  - Not case sensitive

e.g. .code

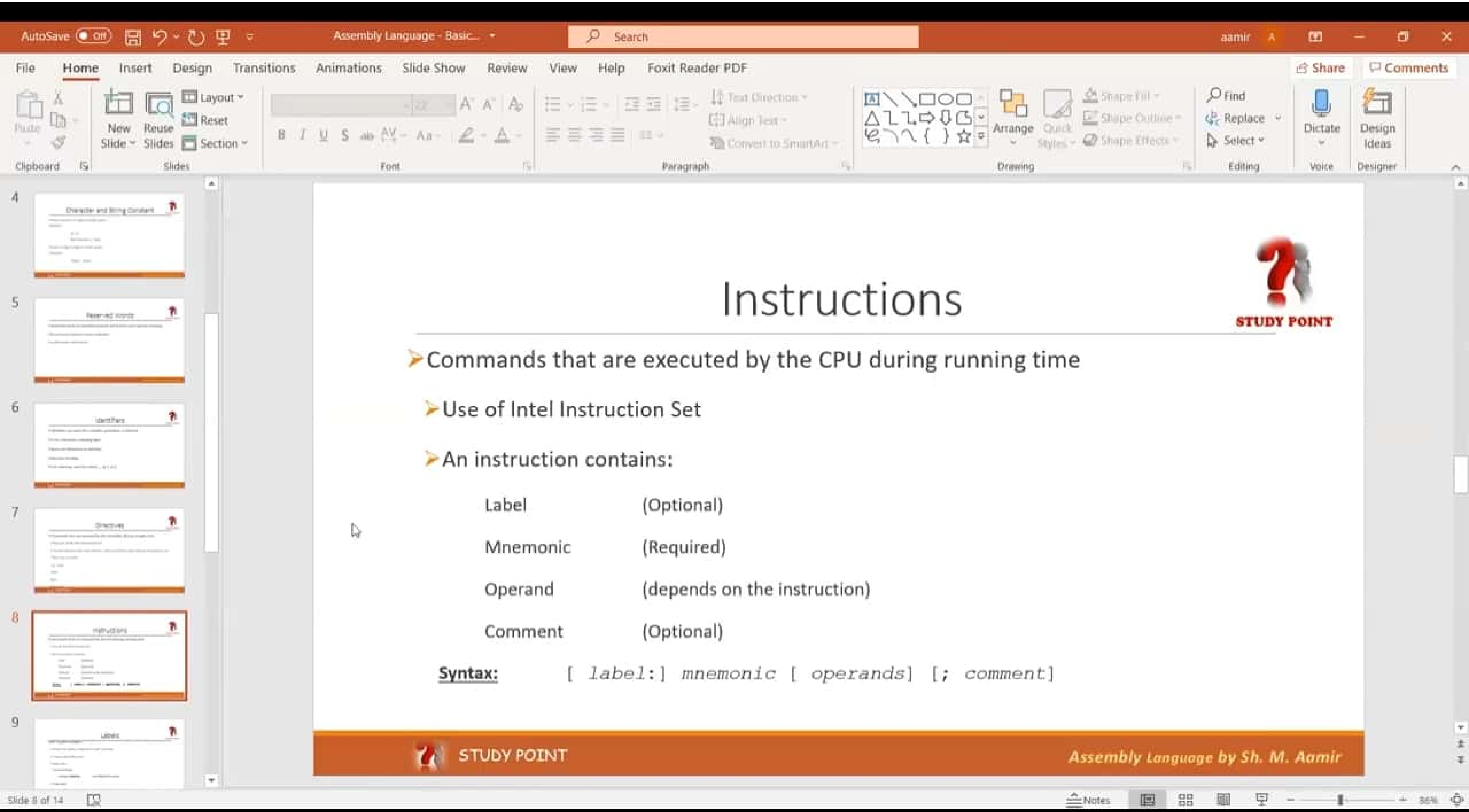
.data

Byte

Word etc

STUDY POINT

Assembly Language by Sh. M. Aamir


AutoSave Off
Assembly Language - Basic...
Search
aamir
File Home Insert Design Transitions Animations Slide Show Review View Help Foxit Reader PDF
Share
Comments
Clipboard Slides Font Paragraph Drawing Editing Voice Designer
Slide 8 of 14
Notes

Instructions

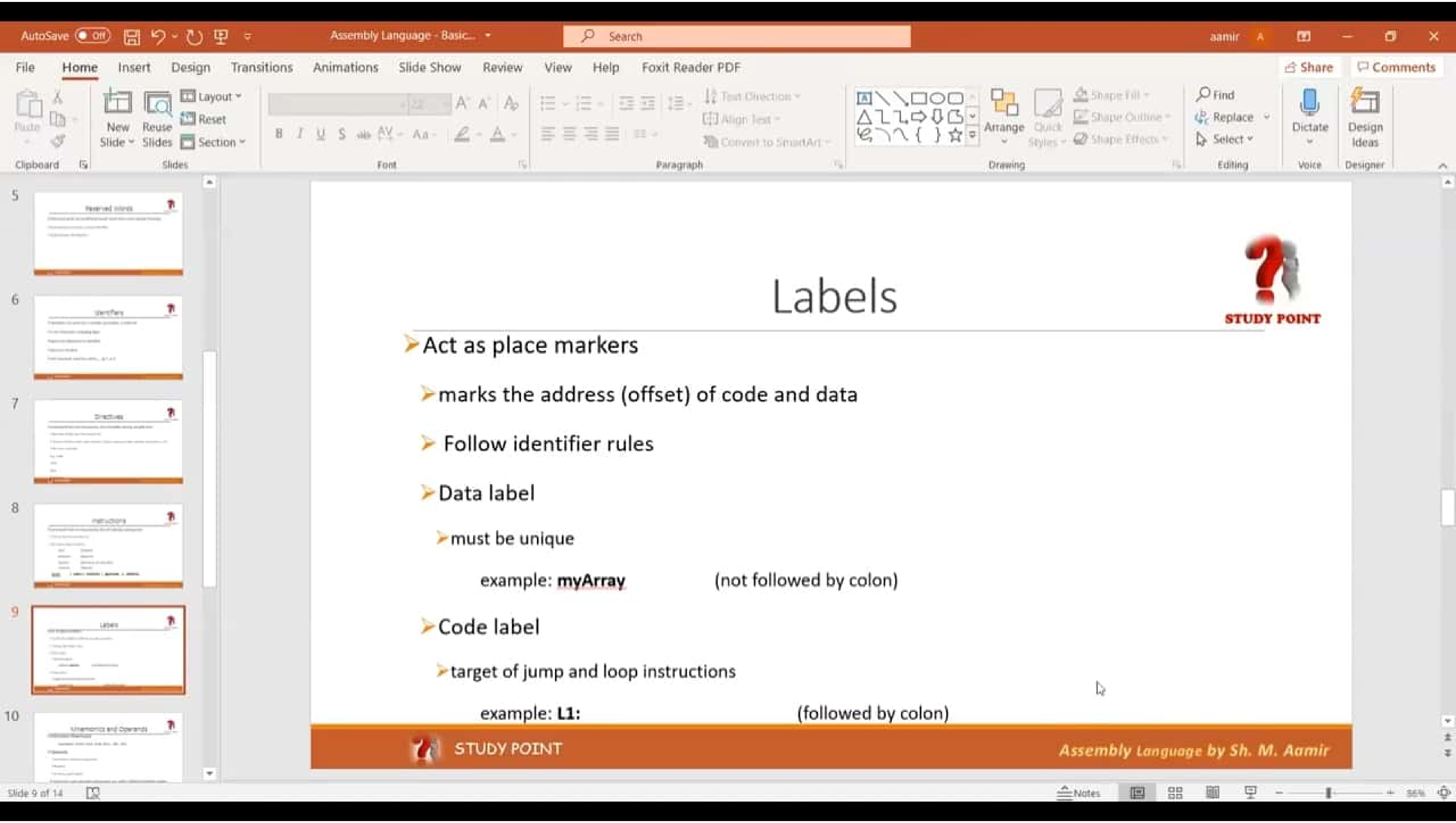
STUDY POINT

- Commands that are executed by the CPU during running time
  - Use of Intel Instruction Set
  - An instruction contains:

Label	(Optional)
Mnemonic	(Required)
Operand	(depends on the instruction)
Comment	(Optional)

Syntax: `[ label:] mnemonic [ operands] [; comment]`

STUDY POINT   *Assembly Language by Sh. M. Aamir*



Labels

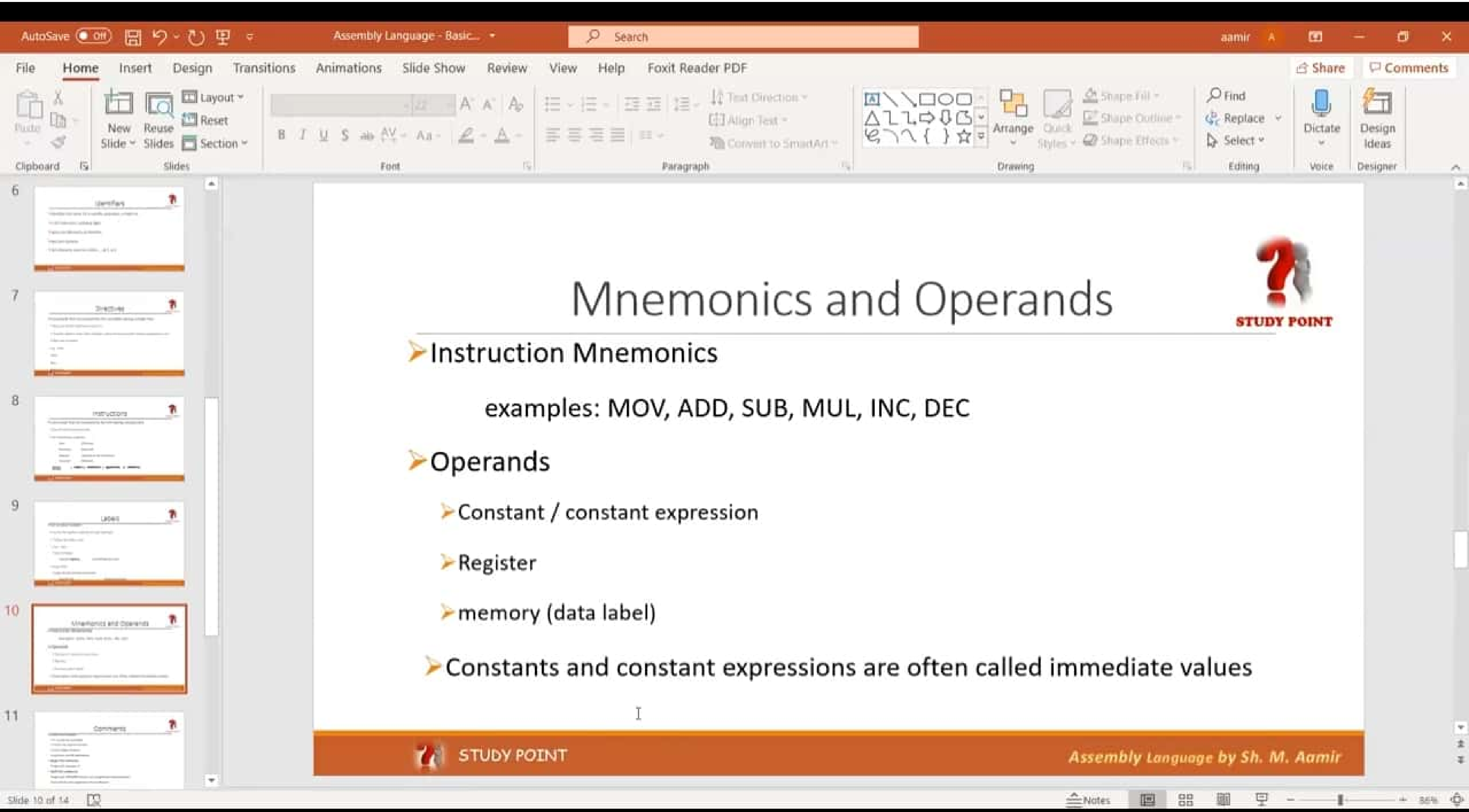
STUDY POINT

- Act as place markers
  - marks the address (offset) of code and data
  - Follow identifier rules
  - Data label
    - must be unique

      example: **myArray** (not followed by colon)
  - Code label
    - target of jump and loop instructions

      example: **L1:** (followed by colon)

STUDY POINT *Assembly Language by Sh. M. Aamir*



Mnemonics and Operands

STUDY POINT

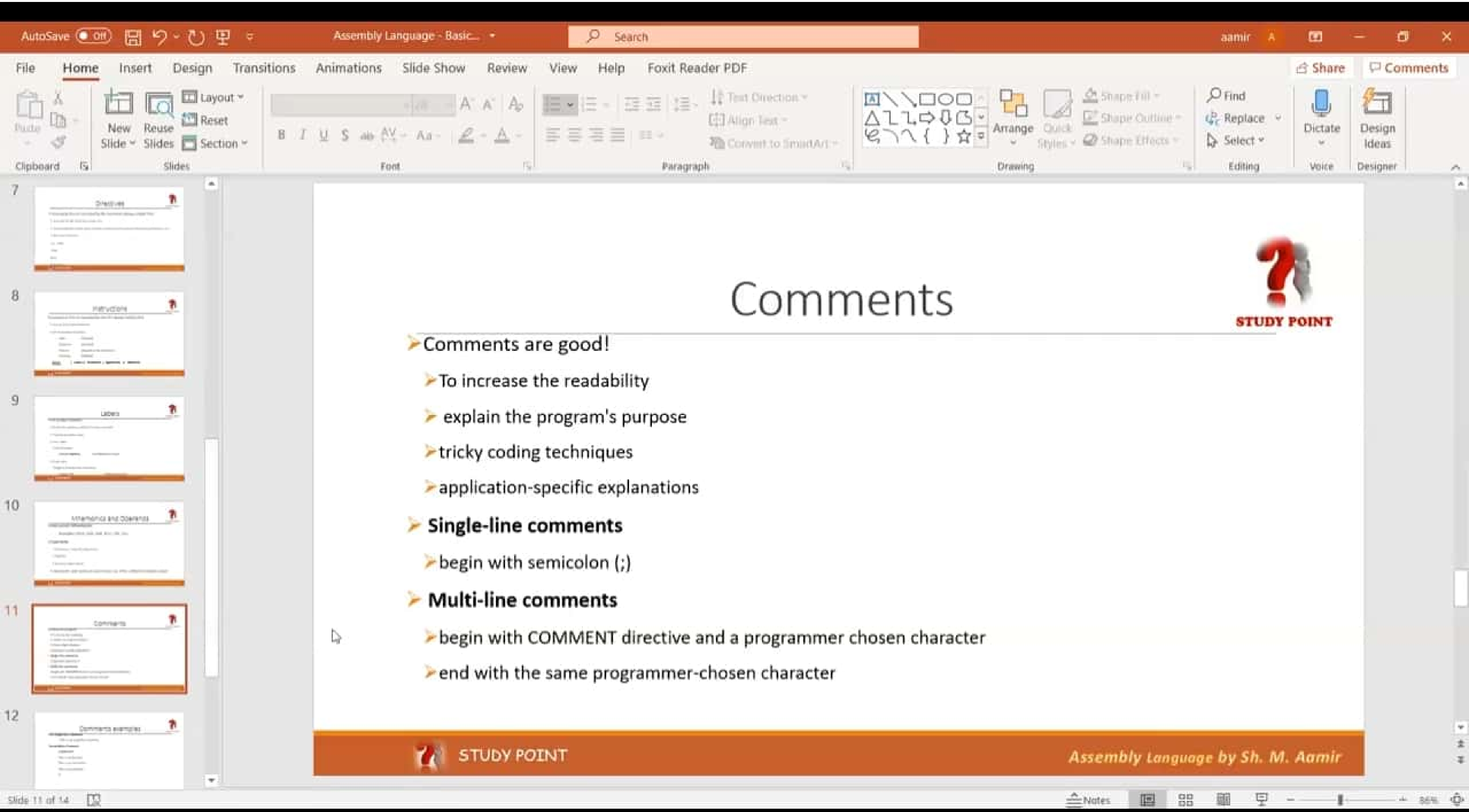
- Instruction Mnemonics

  examples: MOV, ADD, SUB, MUL, INC, DEC

- Operands
  - Constant / constant expression
  - Register
  - memory (data label)
- Constants and constant expressions are often called immediate values

STUDY POINT

Assembly Language by Sh. M. Aamir

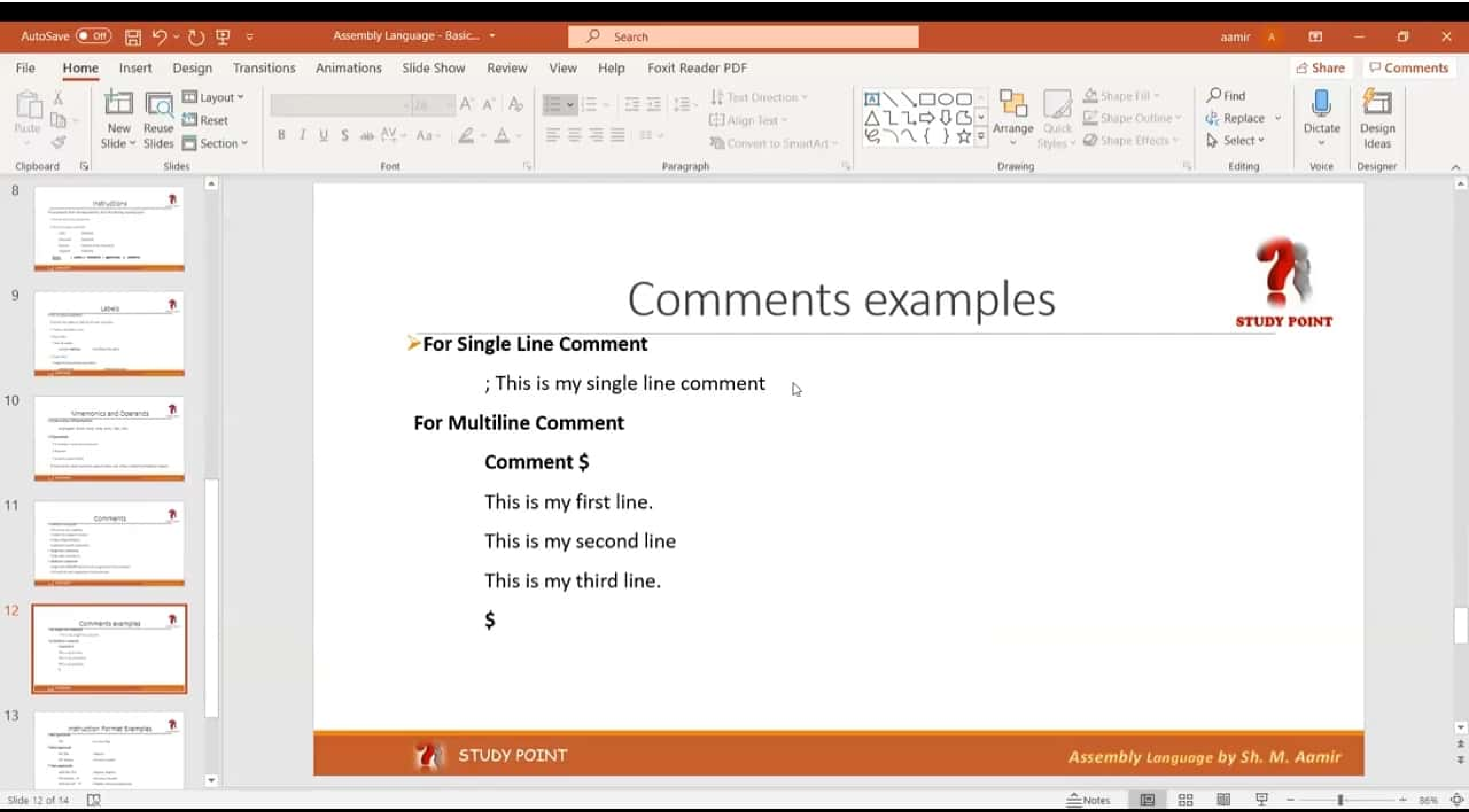


Comments

STUDY POINT

- Comments are good!
  - To increase the readability
  - explain the program's purpose
  - tricky coding techniques
  - application-specific explanations
- **Single-line comments**
  - begin with semicolon (;)
- **Multi-line comments**
  - begin with COMMENT directive and a programmer chosen character
  - end with the same programmer-chosen character

STUDY POINT *Assembly Language by Sh. M. Aamir*



Comments examples

STUDY POINT

- **For Single Line Comment**

  ; This is my single line comment

For Multiline Comment

Comment $

This is my first line.

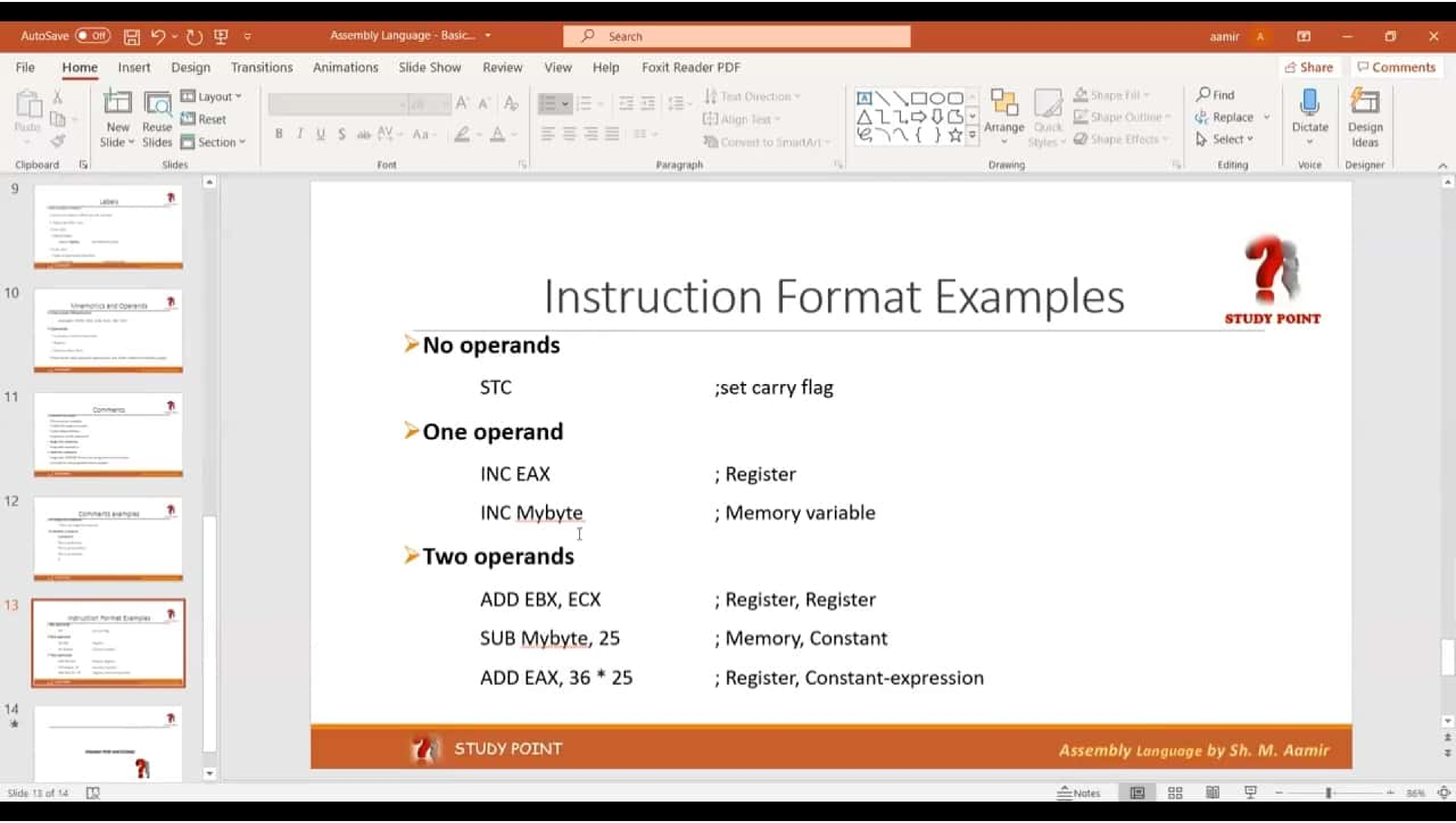
This is my second line

This is my third line.

$

STUDY POINT

Assembly Language by Sh. M. Aamir



Instruction Format Examples

STUDY POINT

- **No operands**

```
STC                 ;set carry flag
```

- **One operand**

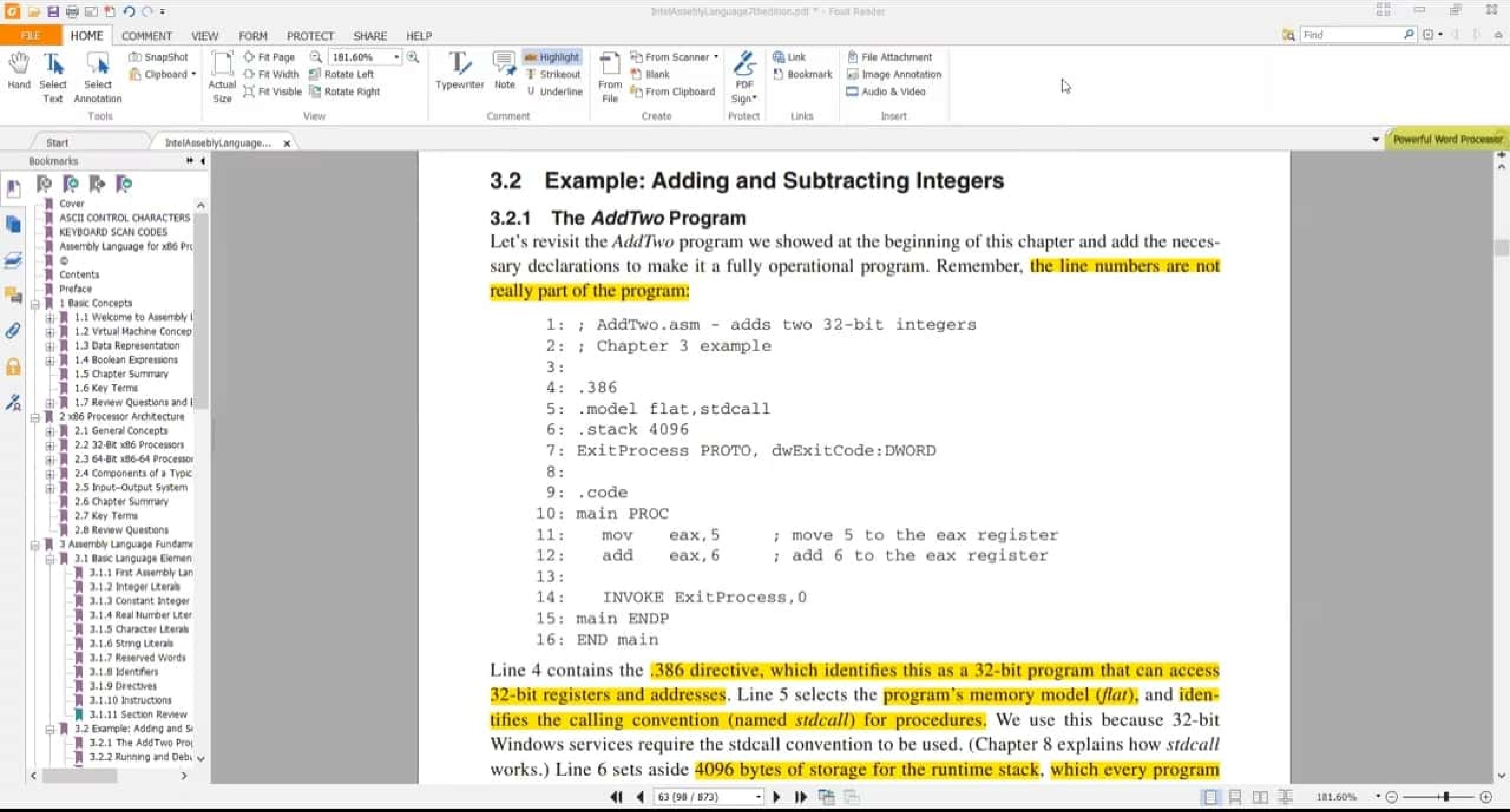
```
INC EAX             ; Register
INC Mybyte          ; Memory variable
```

- **Two operands**

```
ADD EBX, ECX        ; Register, Register
SUB Mybyte, 25      ; Memory, Constant
ADD EAX, 36 * 25    ; Register, Constant-expression
```

STUDY POINT

Assembly Language by Sh. M. Aamir



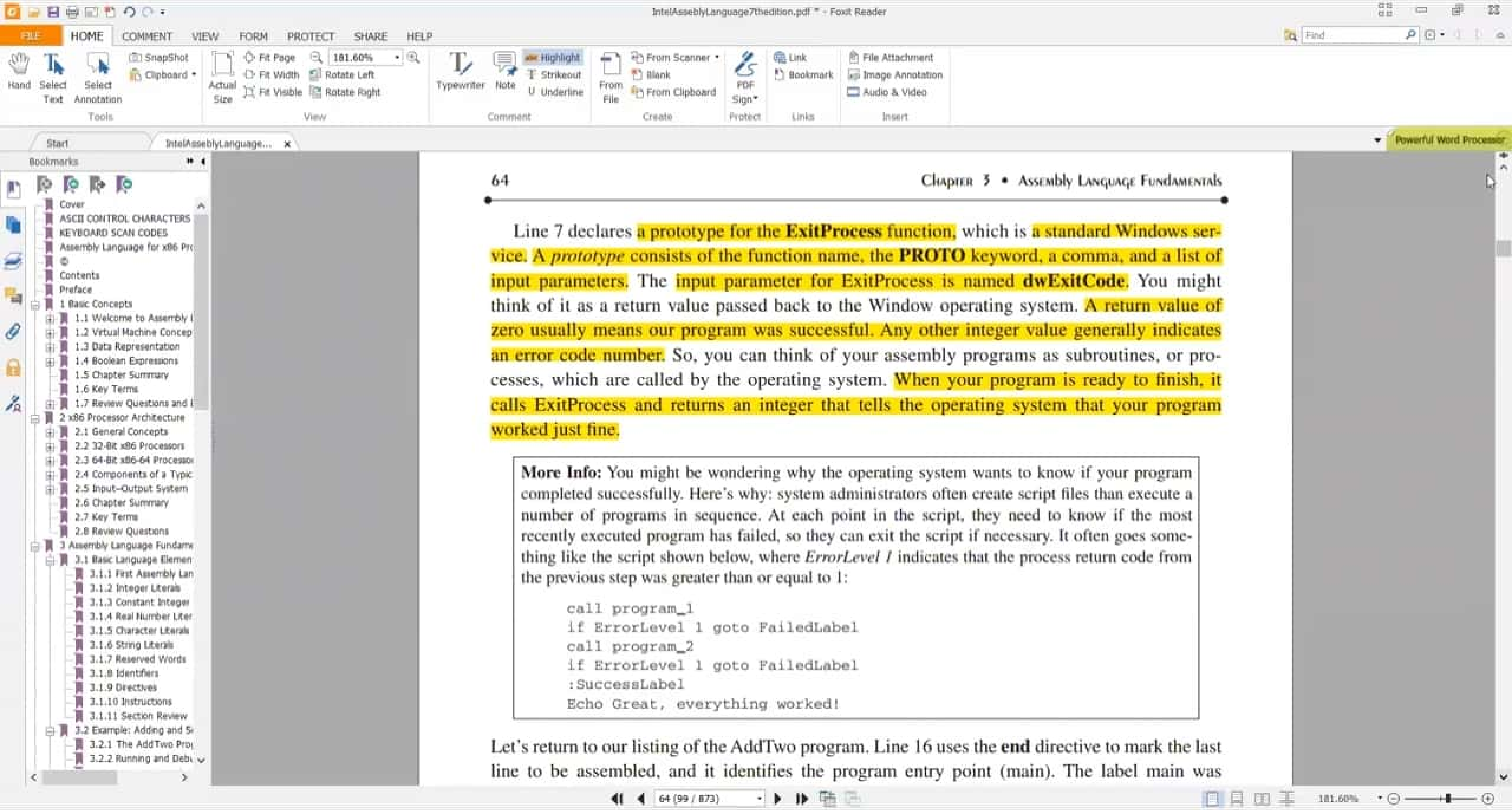
3.2 Example: Adding and Subtracting Integers

3.2.1 The *AddTwo* Program

Let's revisit the *AddTwo* program we showed at the beginning of this chapter and add the necessary declarations to make it a fully operational program. Remember, the line numbers are not really part of the program:

```
 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: .code
10: main PROC
11:    mov    eax,5     ; move 5 to the eax register
12:    add    eax,6     ; add 6 to the eax register
13:
14:    INVOKE ExitProcess,0
15: main ENDP
16: END main
```

Line 4 contains the .386 directive, which identifies this as a 32-bit program that can access 32-bit registers and addresses. Line 5 selects the program's memory model (*flat*), and identifies the calling convention (named *stdcall*) for procedures. We use this because 32-bit Windows services require the stdcall convention to be used. (Chapter 8 explains how *stdcall* works.) Line 6 sets aside 4096 bytes of storage for the runtime stack, which every program

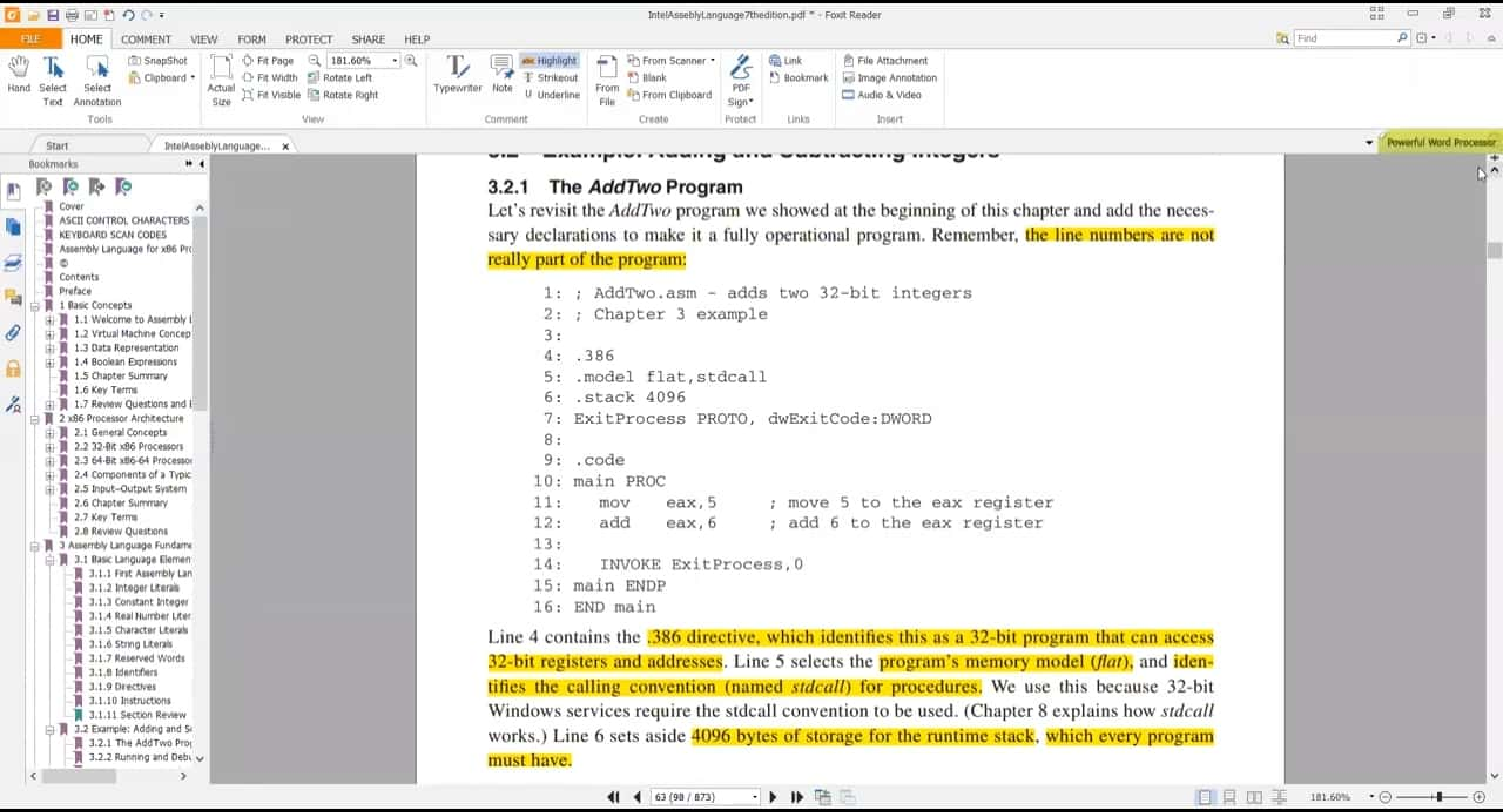


Line 7 declares a prototype for the **ExitProcess** function, which is a standard Windows service. A *prototype* consists of the function name, the **PROTO** keyword, a comma, and a list of input parameters. The input parameter for ExitProcess is named **dwExitCode**. You might think of it as a return value passed back to the Window operating system. A return value of zero usually means our program was successful. Any other integer value generally indicates an error code number. So, you can think of your assembly programs as subroutines, or processes, which are called by the operating system. When your program is ready to finish, it calls ExitProcess and returns an integer that tells the operating system that your program worked just fine.

More Info: You might be wondering why the operating system wants to know if your program completed successfully. Here's why: system administrators often create script files than execute a number of programs in sequence. At each point in the script, they need to know if the most recently executed program has failed, so they can exit the script if necessary. It often goes something like the script shown below, where *ErrorLevel 1* indicates that the process return code from the previous step was greater than or equal to 1:

```
call program_1
if ErrorLevel 1 goto FailedLabel
call program_2
if ErrorLevel 1 goto FailedLabel
:SuccessLabel
Echo Great, everything worked!
```

Let's return to our listing of the AddTwo program. Line 16 uses the **end** directive to mark the last line to be assembled, and it identifies the program entry point (main). The label main was



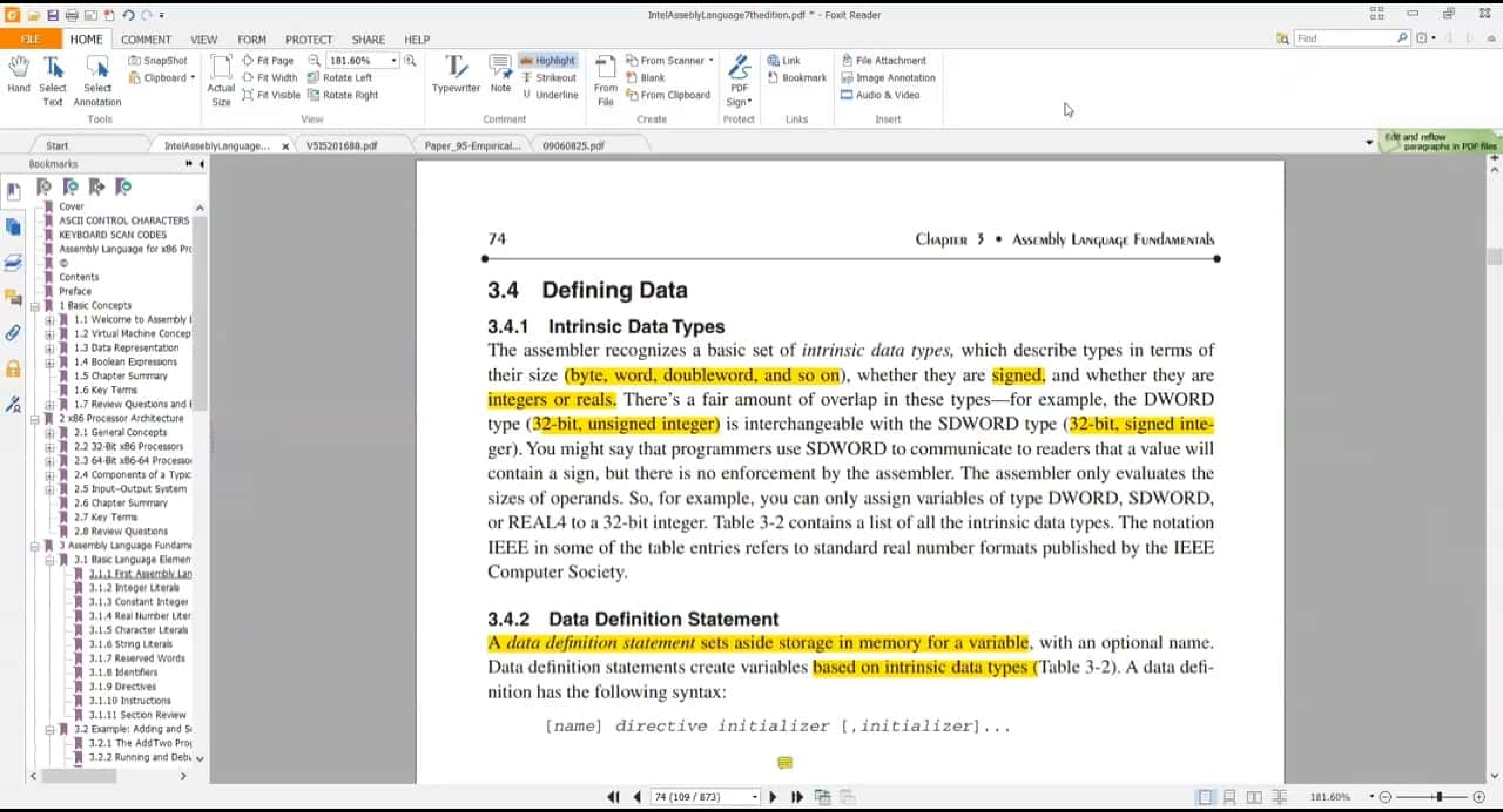
3.2 Example: Adding and Subtracting Integers

3.2.1 The *AddTwo* Program

Let's revisit the *AddTwo* program we showed at the beginning of this chapter and add the necessary declarations to make it a fully operational program. Remember, the line numbers are not really part of the program:

```
 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: .code
10: main PROC
11:    mov    eax,5      ; move 5 to the eax register
12:    add    eax,6      ; add 6 to the eax register
13:
14:    INVOKE ExitProcess,0
15: main ENDP
16: END main
```

Line 4 contains the .386 directive, which identifies this as a 32-bit program that can access 32-bit registers and addresses. Line 5 selects the program's memory model (*flat*), and identifies the calling convention (named *stdcall*) for procedures. We use this because 32-bit Windows services require the stdcall convention to be used. (Chapter 8 explains how *stdcall* works.) Line 6 sets aside 4096 bytes of storage for the runtime stack, which every program must have.



3.4 Defining Data

3.4.1 Intrinsic Data Types

The assembler recognizes a basic set of *intrinsic data types*, which describe types in terms of their size (byte, word, doubleword, and so on), whether they are signed, and whether they are integers or reals. There's a fair amount of overlap in these types—for example, the DWORD type (32-bit, unsigned integer) is interchangeable with the SDWORD type (32-bit, signed integer). You might say that programmers use SDWORD to communicate to readers that a value will contain a sign, but there is no enforcement by the assembler. The assembler only evaluates the sizes of operands. So, for example, you can only assign variables of type DWORD, SDWORD, or REAL4 to a 32-bit integer. Table 3-2 contains a list of all the intrinsic data types. The notation IEEE in some of the table entries refers to standard real number formats published by the IEEE Computer Society.

3.4.2 Data Definition Statement

A *data definition statement* sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types (Table 3-2). A data definition has the following syntax:

```
[name] directive initializer [,initializer]...
```

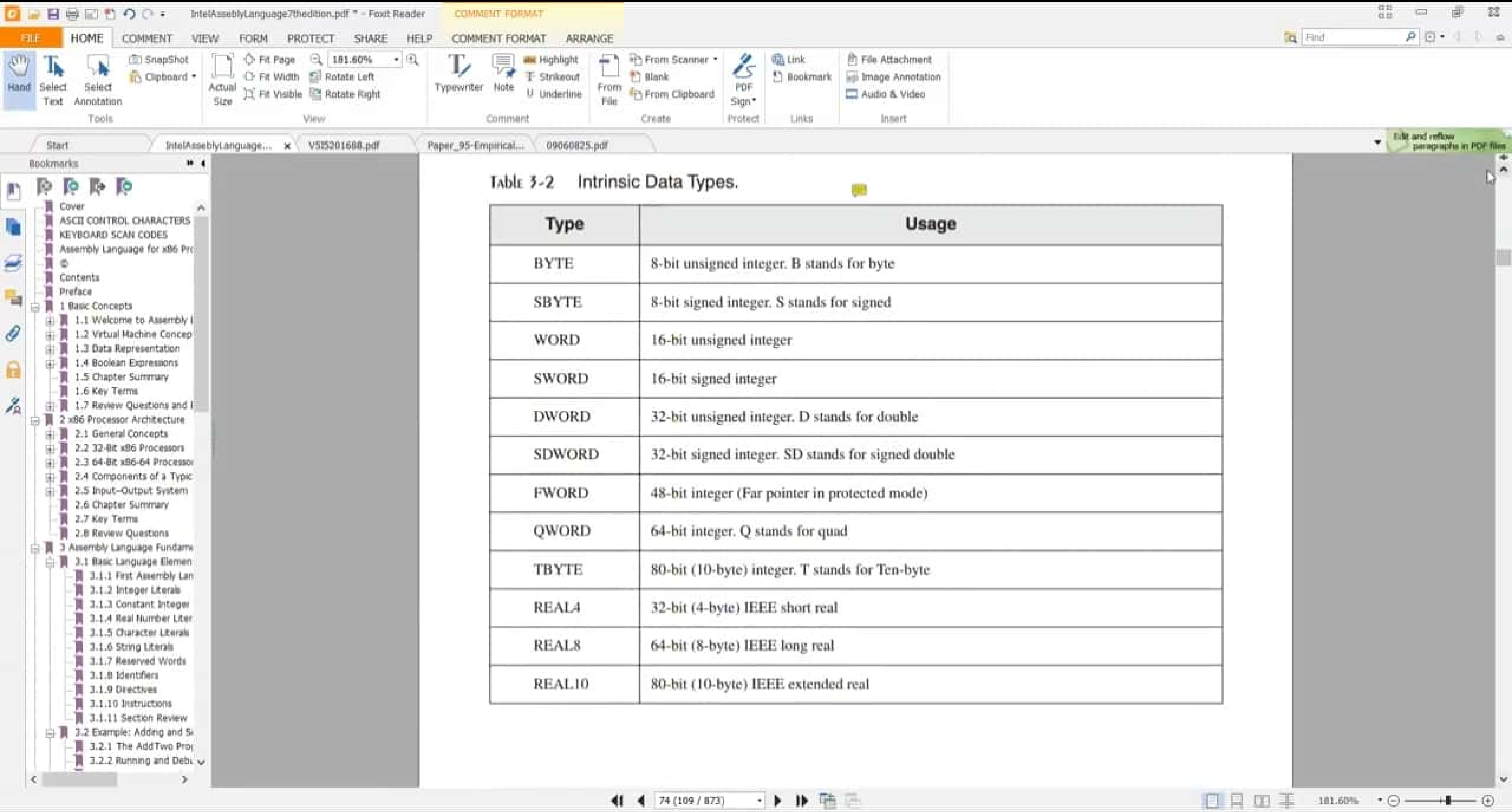
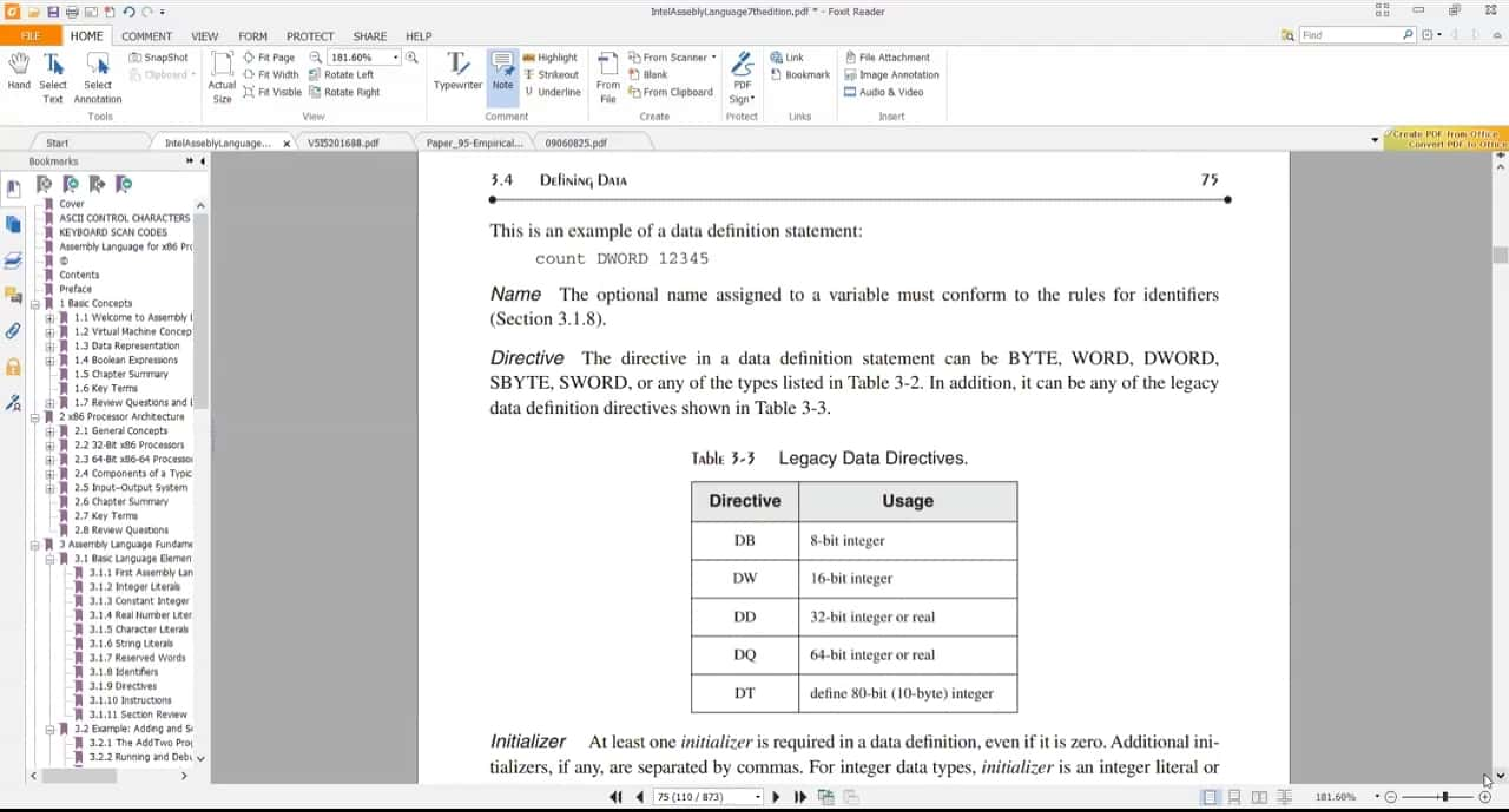


Table 3-2 Intrinsic Data Types.

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



This is an example of a data definition statement:

```
count DWORD 12345
```

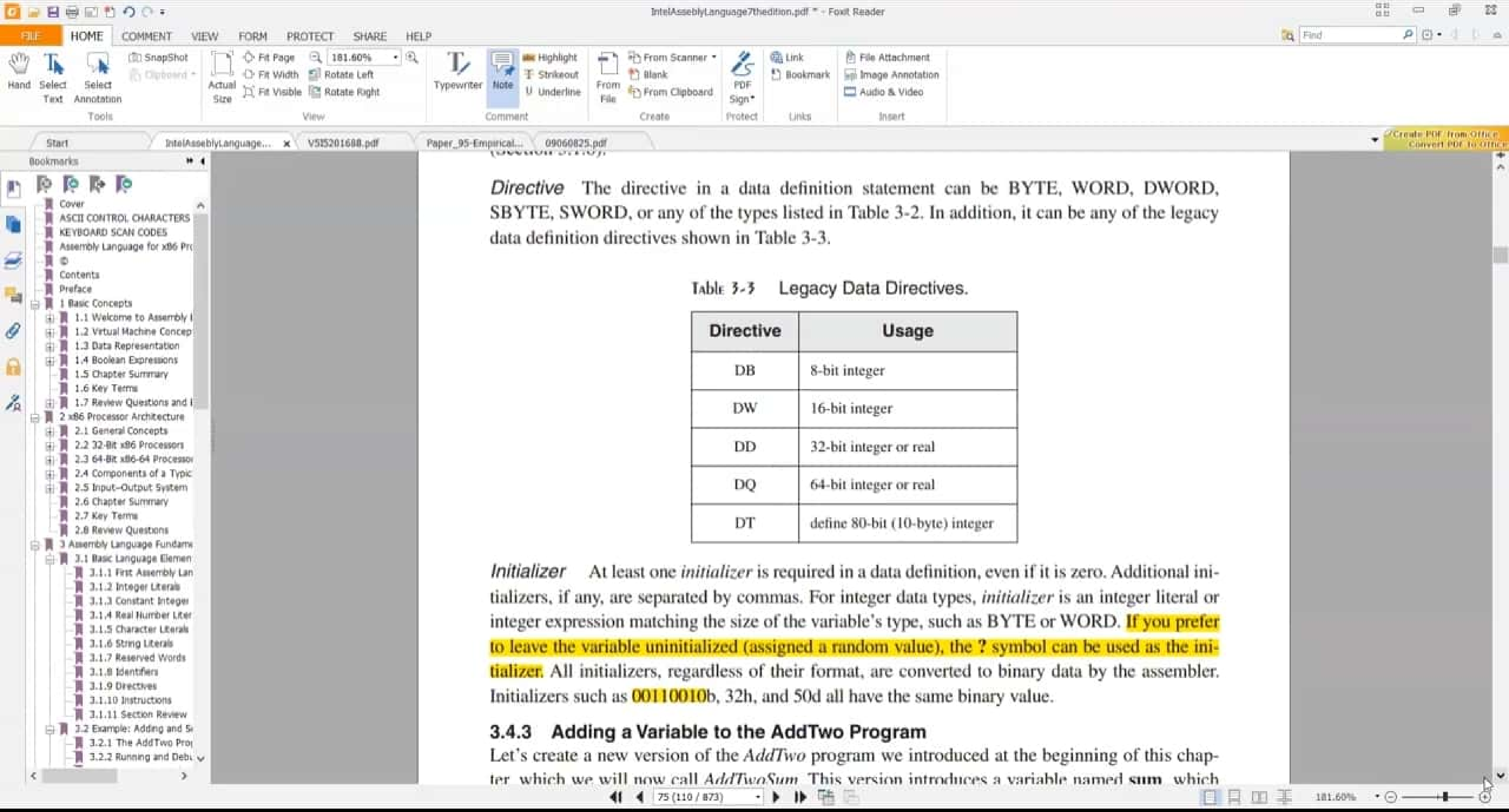
Name The optional name assigned to a variable must conform to the rules for identifiers (Section 3.1.8).

Directive The directive in a data definition statement can be BYTE, WORD, DWORD, SBYTE, SWORD, or any of the types listed in Table 3-2. In addition, it can be any of the legacy data definition directives shown in Table 3-3.

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

Initializer At least one *initializer* is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas. For integer data types, *initializer* is an integer literal or



Directive The directive in a data definition statement can be BYTE, WORD, DWORD, SBYTE, SWORD, or any of the types listed in Table 3-2. In addition, it can be any of the legacy data definition directives shown in Table 3-3.

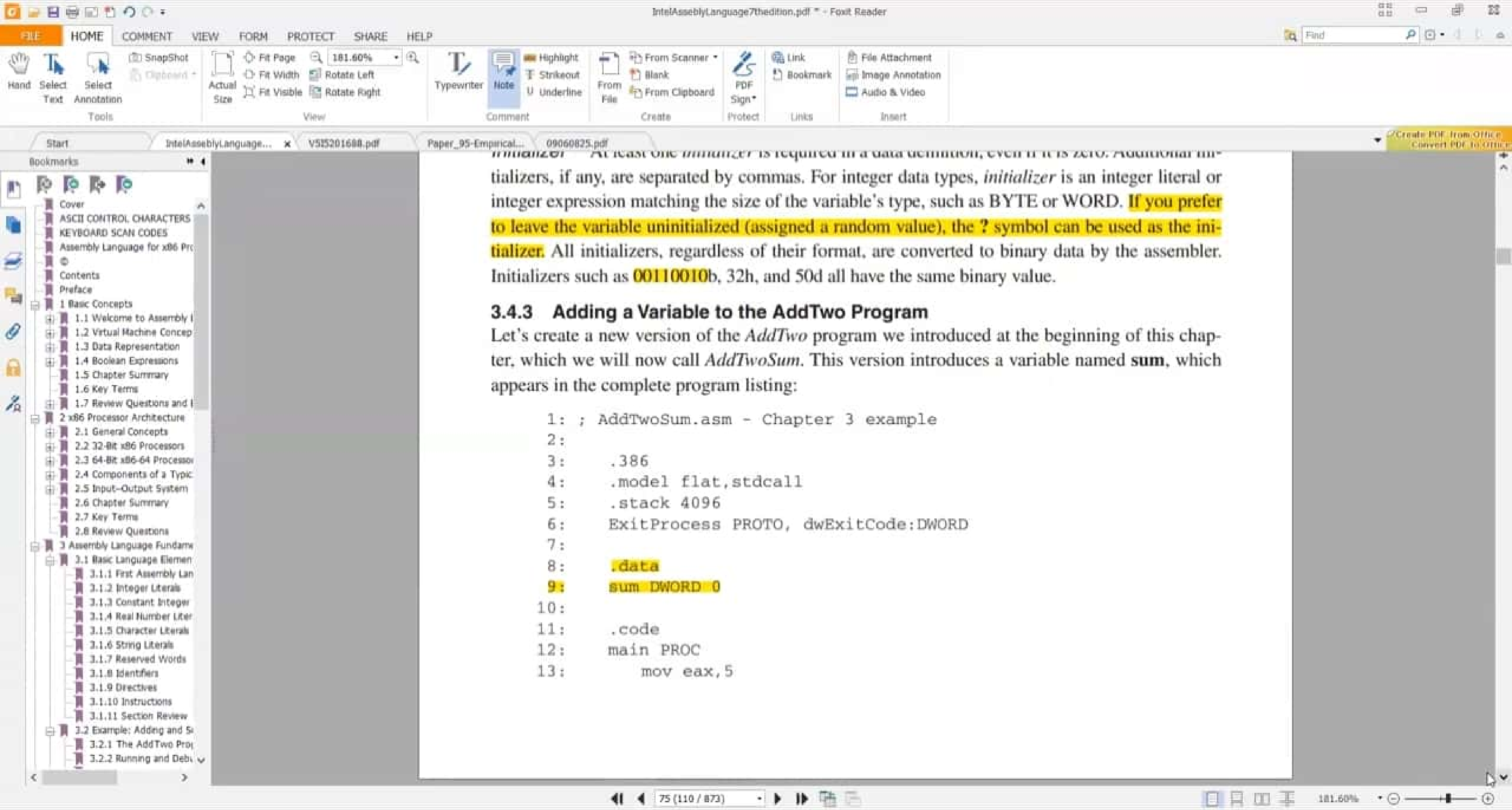
Table 3-3 Legacy Data Directives.

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

Initializer At least one *initializer* is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas. For integer data types, *initializer* is an integer literal or integer expression matching the size of the variable's type, such as BYTE or WORD. If you prefer to leave the variable uninitialized (assigned a random value), the ? symbol can be used as the initializer. All initializers, regardless of their format, are converted to binary data by the assembler. Initializers such as 00110010b, 32h, and 50d all have the same binary value.

3.4.3 Adding a Variable to the AddTwo Program

Let's create a new version of the *AddTwo* program we introduced at the beginning of this chapter, which we will now call *AddTwoSum*. This version introduces a variable named **sum**, which

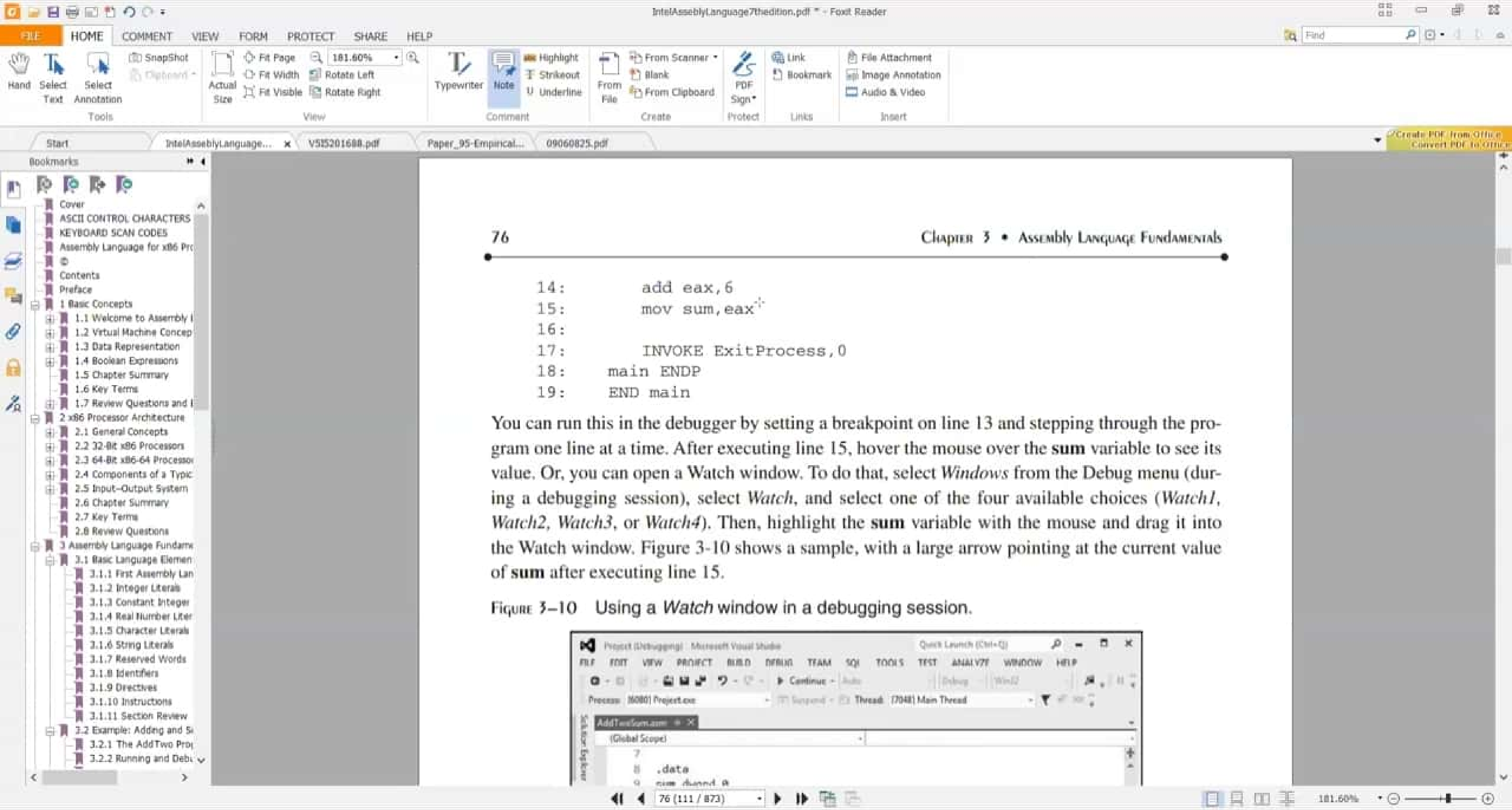


initializer At least one *initializer* is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas. For integer data types, *initializer* is an integer literal or integer expression matching the size of the variable's type, such as BYTE or WORD. If you prefer to leave the variable uninitialized (assigned a random value), the ? symbol can be used as the initializer. All initializers, regardless of their format, are converted to binary data by the assembler. Initializers such as 00110010b, 32h, and 50d all have the same binary value.

3.4.3 Adding a Variable to the AddTwo Program

Let's create a new version of the *AddTwo* program we introduced at the beginning of this chapter, which we will now call *AddTwoSum*. This version introduces a variable named **sum**, which appears in the complete program listing:

```
; AddTwoSum.asm - Chapter 3 example

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

 .data
 sum DWORD 0

 .code
 main PROC
     mov eax,5
```



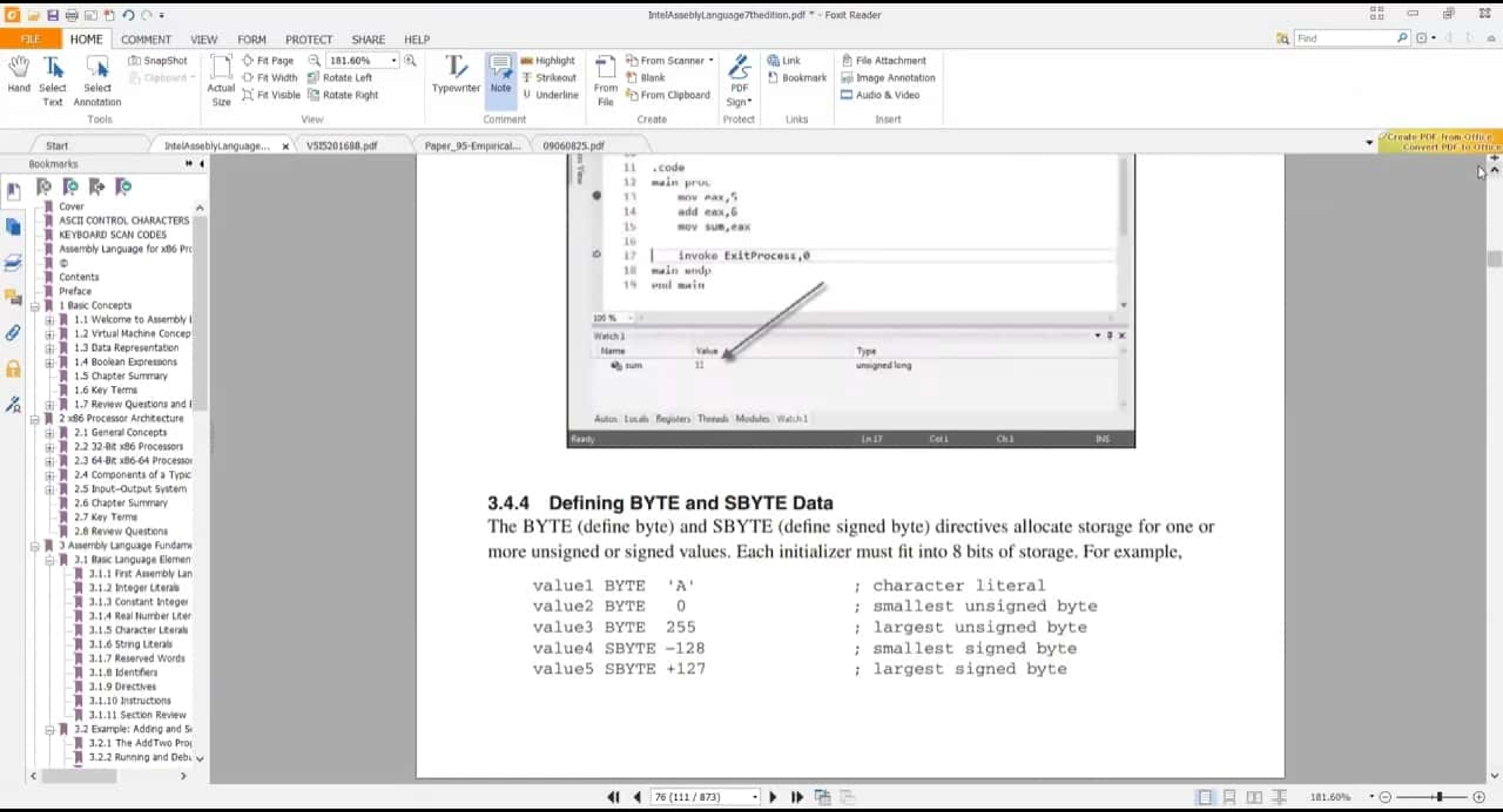
```
14:         add eax,6
15:         mov sum,eax
16:
17:         INVOKE ExitProcess,0
18:     main ENDP
19:     END main
```

You can run this in the debugger by setting a breakpoint on line 13 and stepping through the program one line at a time. After executing line 15, hover the mouse over the **sum** variable to see its value. Or, you can open a Watch window. To do that, select *Windows* from the Debug menu (during a debugging session), select *Watch*, and select one of the four available choices (*Watch1*, *Watch2*, *Watch3*, or *Watch4*). Then, highlight the **sum** variable with the mouse and drag it into the Watch window. Figure 3-10 shows a sample, with a large arrow pointing at the current value of **sum** after executing line 15.

Figure 3–10 Using a *Watch* window in a debugging session.

Project (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Continue
Debug
Win32
Process: [6080] Project.exe
Suspend
Thread: [7048] Main Thread
Solution Explorer
AddTwoSum.asm
(Global Scope)

```
.data
sum dword 0
```



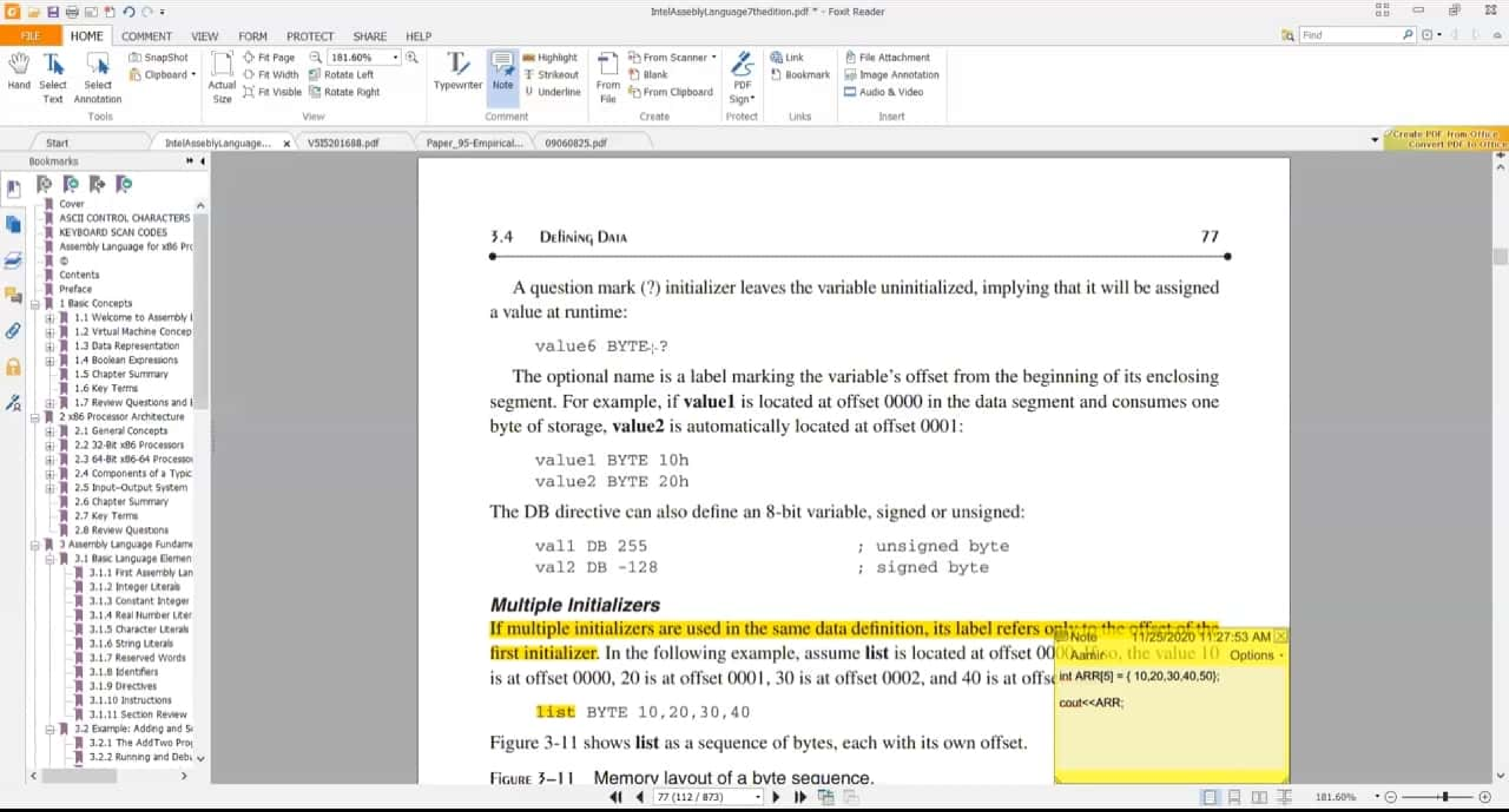
```
.code
main proc
    mov eax,5
    add eax,6
    mov sum,eax

    invoke ExitProcess,0
main endp
end main
```

Name	Value	Type
sum	11	unsigned long

3.4.4 Defining BYTE and SBYTE Data

The BYTE (define byte) and SBYTE (define signed byte) directives allocate storage for one or more unsigned or signed values. Each initializer must fit into 8 bits of storage. For example,

```
value1 BYTE  'A'               ; character literal
value2 BYTE   0                ; smallest unsigned byte
value3 BYTE  255               ; largest unsigned byte
value4 SBYTE -128              ; smallest signed byte
value5 SBYTE +127              ; largest signed byte
```



3.4 Defining Data 77

A question mark (?) initializer leaves the variable uninitialized, implying that it will be assigned a value at runtime:

```
value6 BYTE ?
```

The optional name is a label marking the variable's offset from the beginning of its enclosing segment. For example, if **value1** is located at offset 0000 in the data segment and consumes one byte of storage, **value2** is automatically located at offset 0001:

```
value1 BYTE 10h
value2 BYTE 20h
```

The DB directive can also define an 8-bit variable, signed or unsigned:

```
val1 DB 255                 ; unsigned byte
val2 DB -128                ; signed byte
```

Multiple Initializers

If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer. In the following example, assume **list** is located at offset 0000. Also, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offs

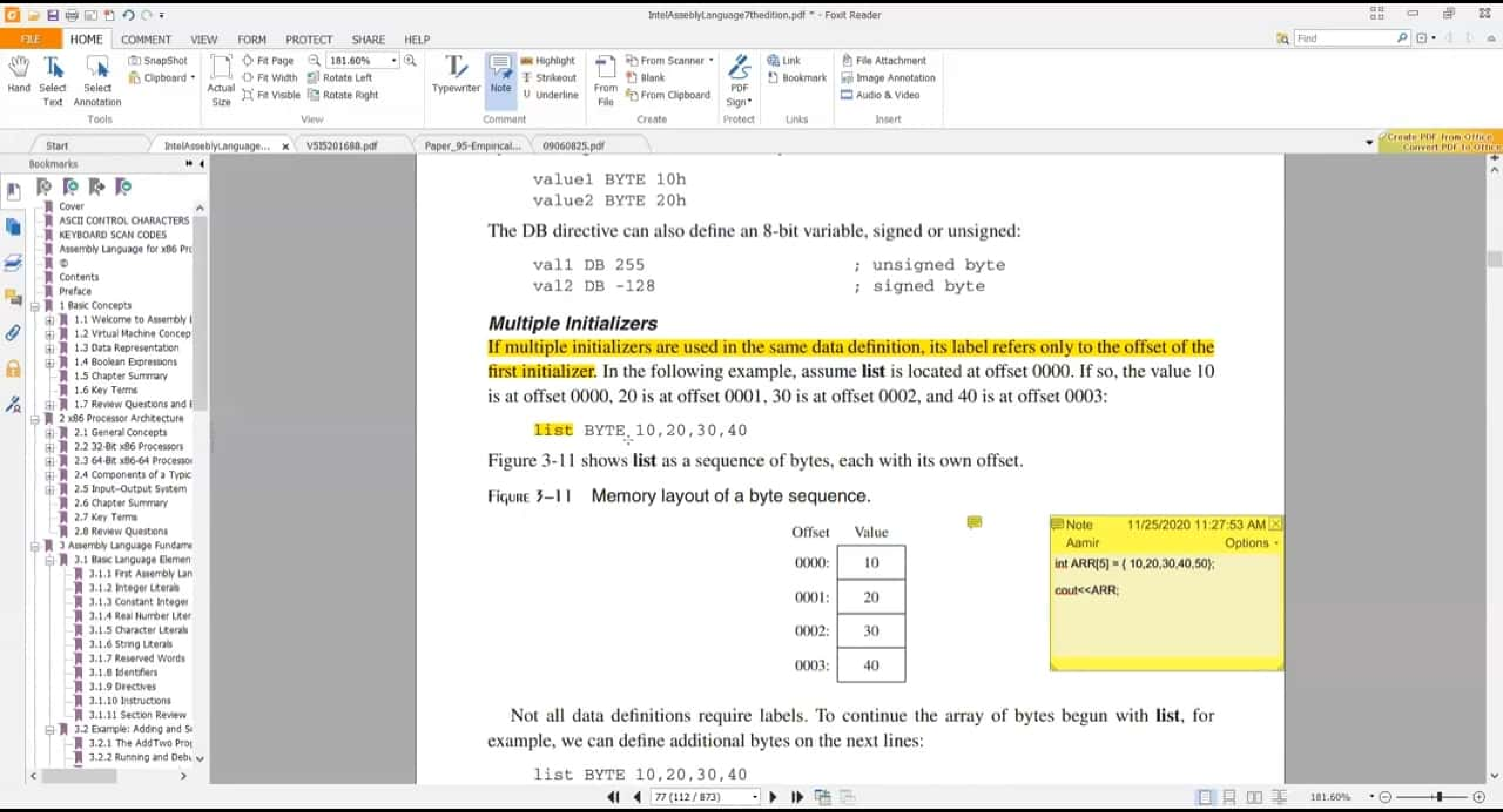
```
list BYTE 10,20,30,40
```

Figure 3-11 shows **list** as a sequence of bytes, each with its own offset.

FIGURE 3-11 Memory layout of a byte sequence.

Note 11/25/2020 11:27:53 AM
Aamir Options

```
int ARR[5] = { 10,20,30,40,50};

cout<<ARR;
```



```
value1 BYTE 10h
value2 BYTE 20h
```

The DB directive can also define an 8-bit variable, signed or unsigned:

```
val1 DB 255                    ; unsigned byte
val2 DB -128                   ; signed byte
```

Multiple Initializers

If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer. In the following example, assume **list** is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003:

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

Figure 3-11 shows **list** as a sequence of bytes, each with its own offset.

Figure 3–11 Memory layout of a byte sequence.

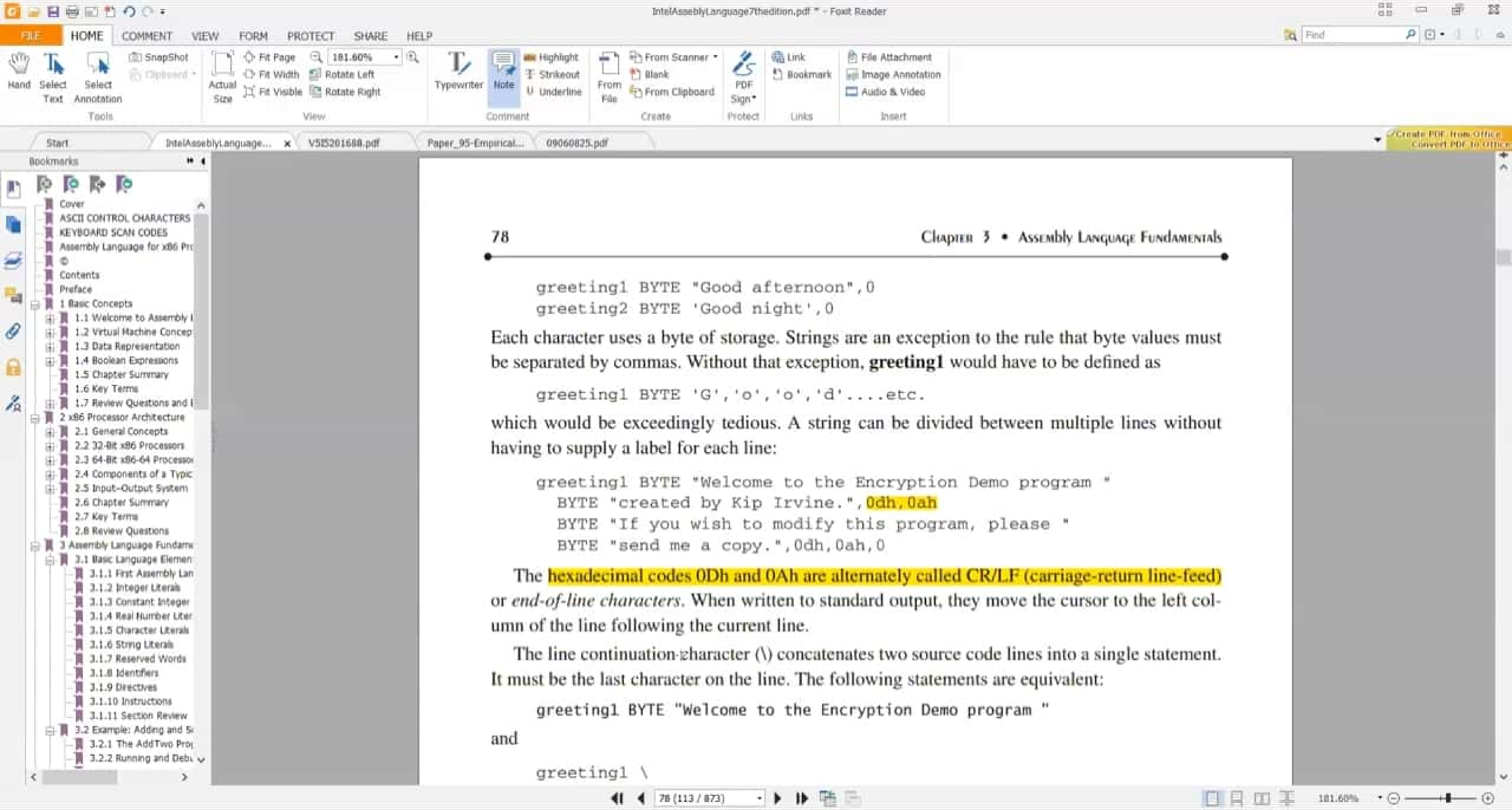
Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

Note 11/25/2020 11:27:53 AM
Aamir Options

```
int ARR[5] = { 10,20,30,40,50};

cout<<ARR;
```

Not all data definitions require labels. To continue the array of bytes begun with **list**, for example, we can define additional bytes on the next lines:

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



```
greeting1 BYTE "Good afternoon",0
greeting2 BYTE 'Good night',0
```

Each character uses a byte of storage. Strings are an exception to the rule that byte values must be separated by commas. Without that exception, **greeting1** would have to be defined as

```
greeting1 BYTE 'G','o','o','d'....etc.
```

which would be exceedingly tedious. A string can be divided between multiple lines without having to supply a label for each line:

```
greeting1 BYTE "Welcome to the Encryption Demo program "
  BYTE "created by Kip Irvine.",0dh,0ah
  BYTE "If you wish to modify this program, please "
  BYTE "send me a copy.",0dh,0ah,0
```

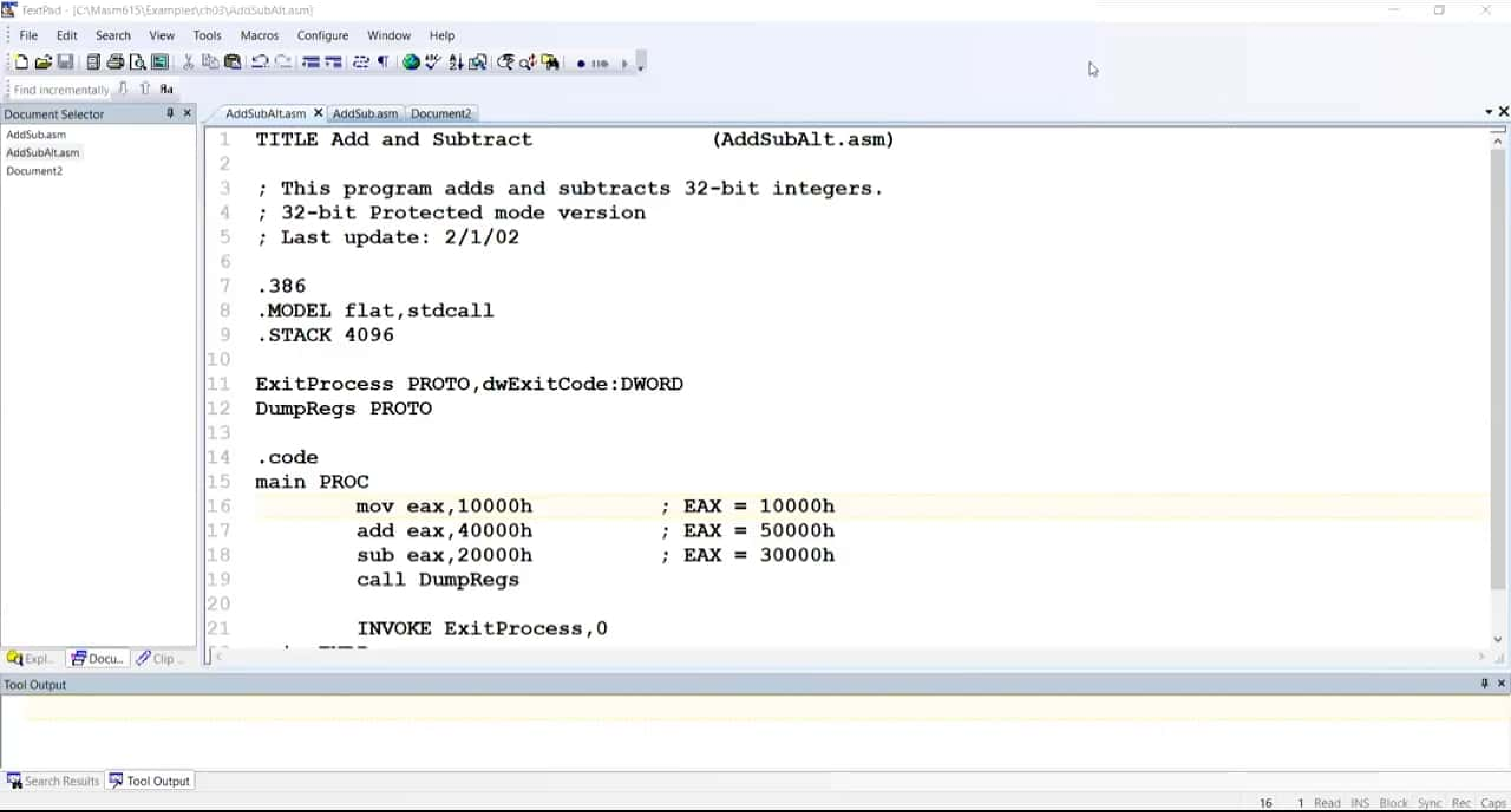
The hexadecimal codes 0Dh and 0Ah are alternately called CR/LF (carriage-return line-feed) or *end-of-line characters*. When written to standard output, they move the cursor to the left column of the line following the current line.

The line continuation character (\) concatenates two source code lines into a single statement. It must be the last character on the line. The following statements are equivalent:

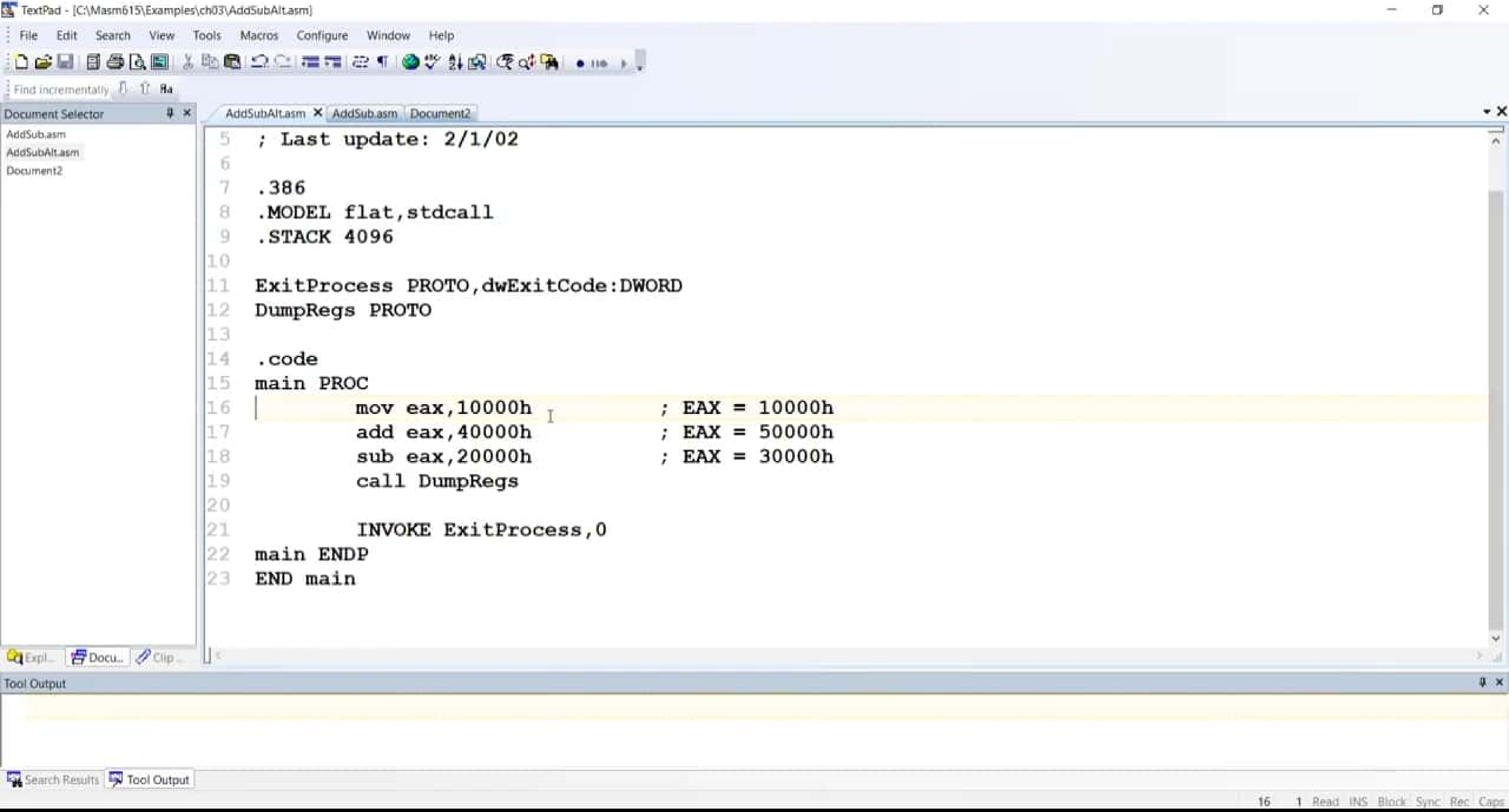
```
greeting1 BYTE "Welcome to the Encryption Demo program "
```

and

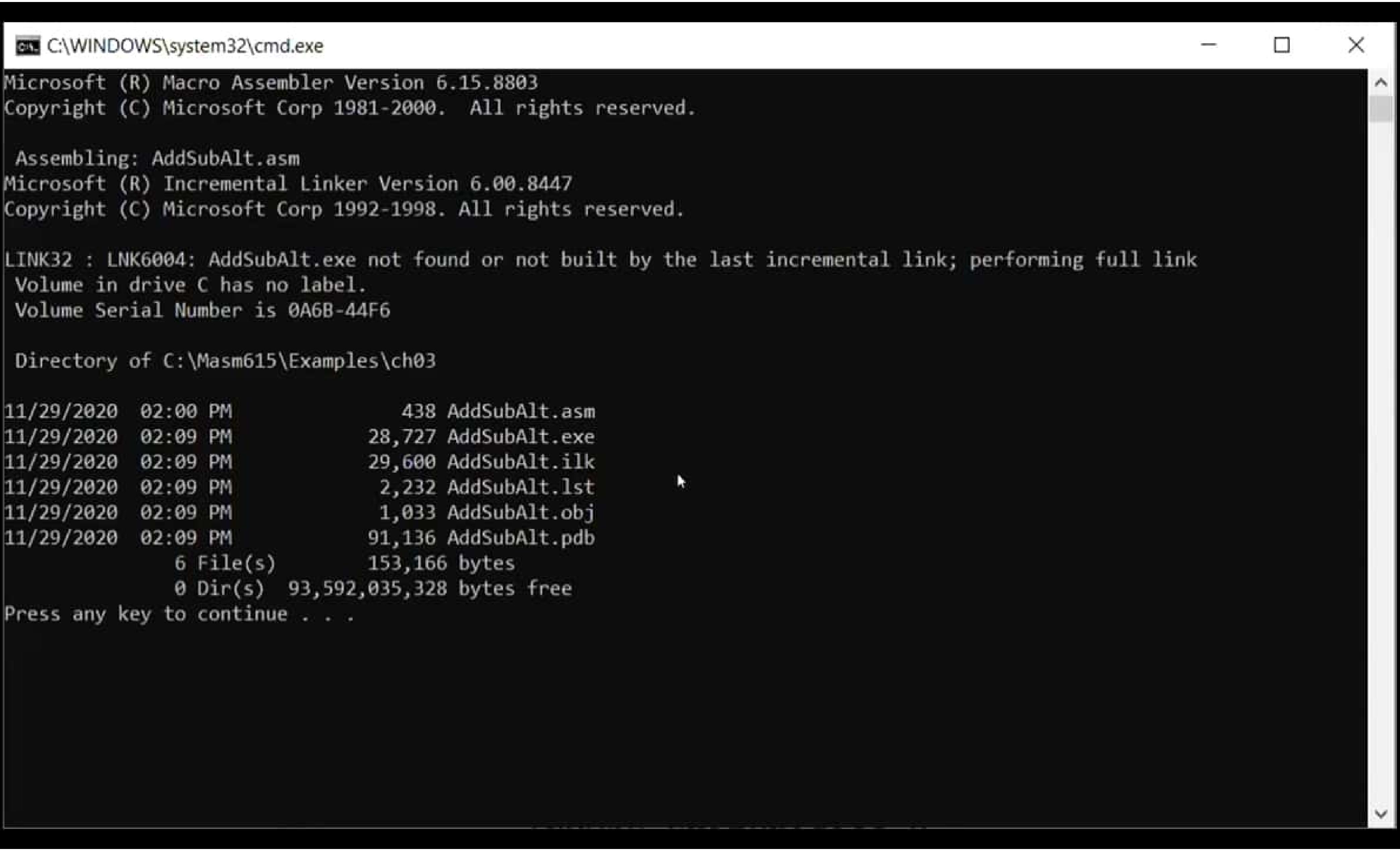
```
greeting1 \
```



```
TITLE Add and Subtract              (AddSubAlt.asm)

; This program adds and subtracts 32-bit integers.
; 32-bit Protected mode version
; Last update: 2/1/02

.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
```

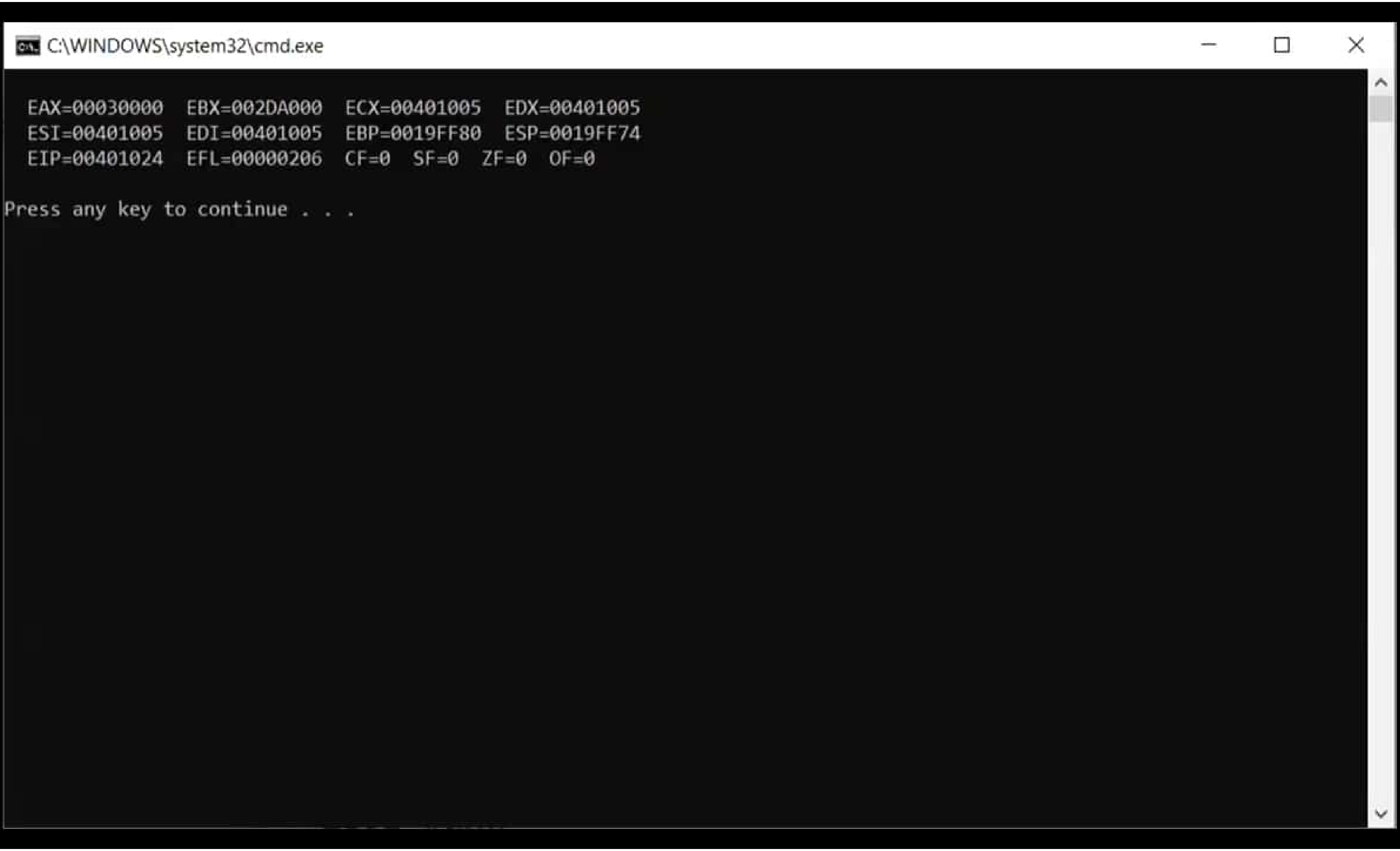


```
; Last update: 2/1/02

.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
```

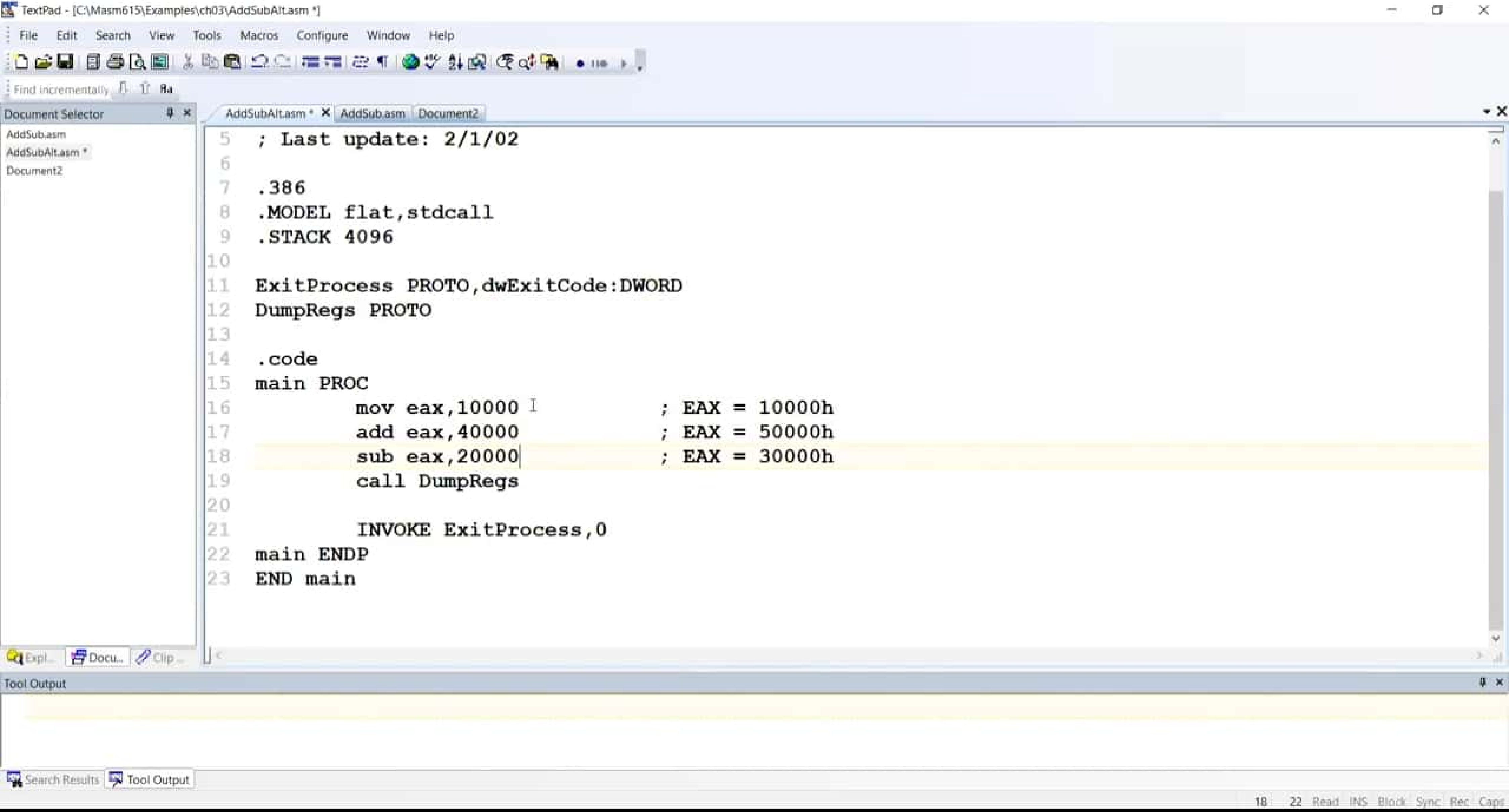


```
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000.  All rights reserved.

 Assembling: AddSubAlt.asm
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

LINK32 : LNK6004: AddSubAlt.exe not found or not built by the last incremental link; performing full link
 Volume in drive C has no label.
 Volume Serial Number is 0A6B-44F6

 Directory of C:\Masm615\Examples\ch03

11/29/2020  02:00 PM               438 AddSubAlt.asm
11/29/2020  02:09 PM            28,727 AddSubAlt.exe
11/29/2020  02:09 PM            29,600 AddSubAlt.ilk
11/29/2020  02:09 PM             2,232 AddSubAlt.lst
11/29/2020  02:09 PM             1,033 AddSubAlt.obj
11/29/2020  02:09 PM            91,136 AddSubAlt.pdb
               6 File(s)        153,166 bytes
               0 Dir(s)  93,592,035,328 bytes free
Press any key to continue . . .
```

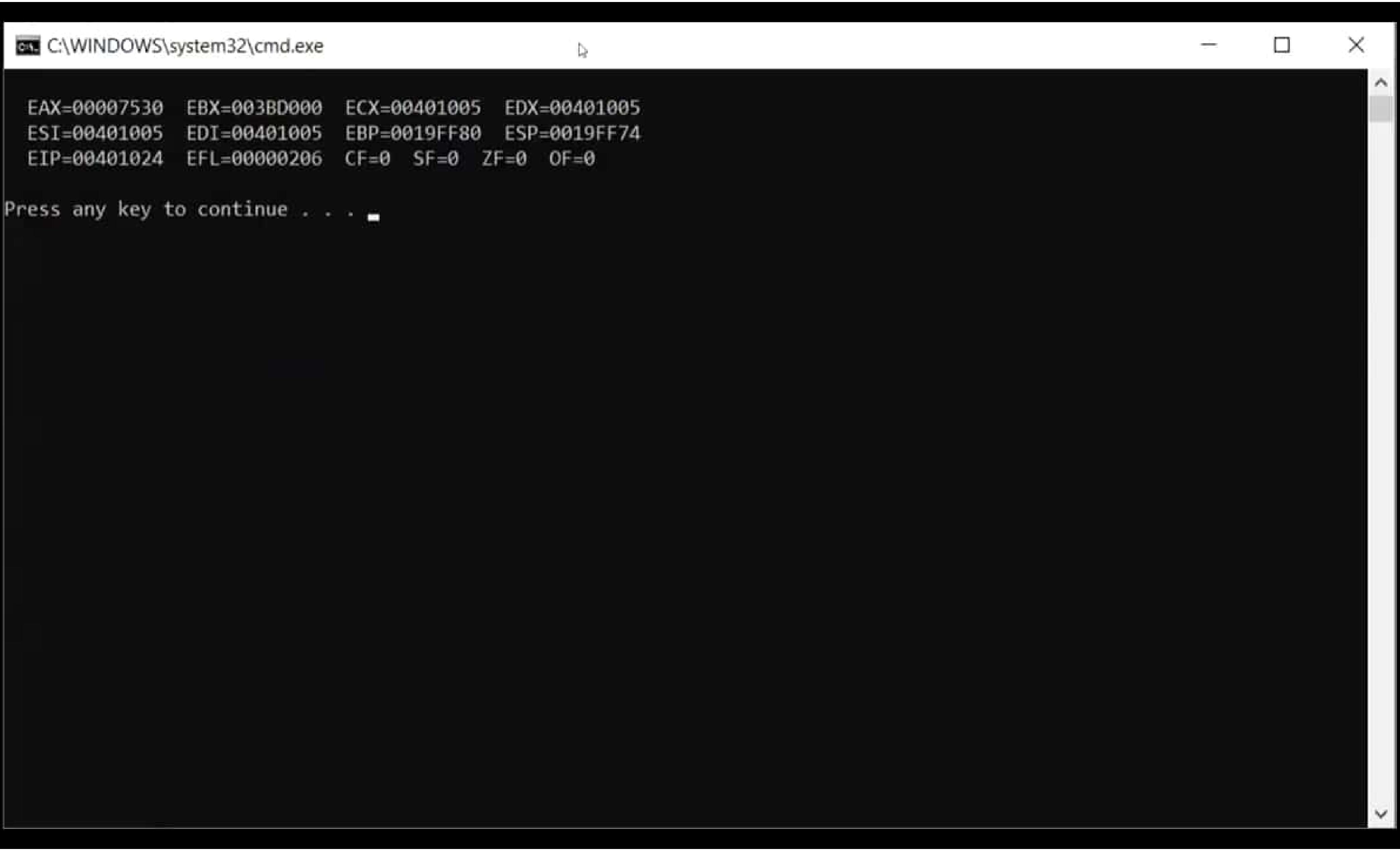
Scanned with CamScanner



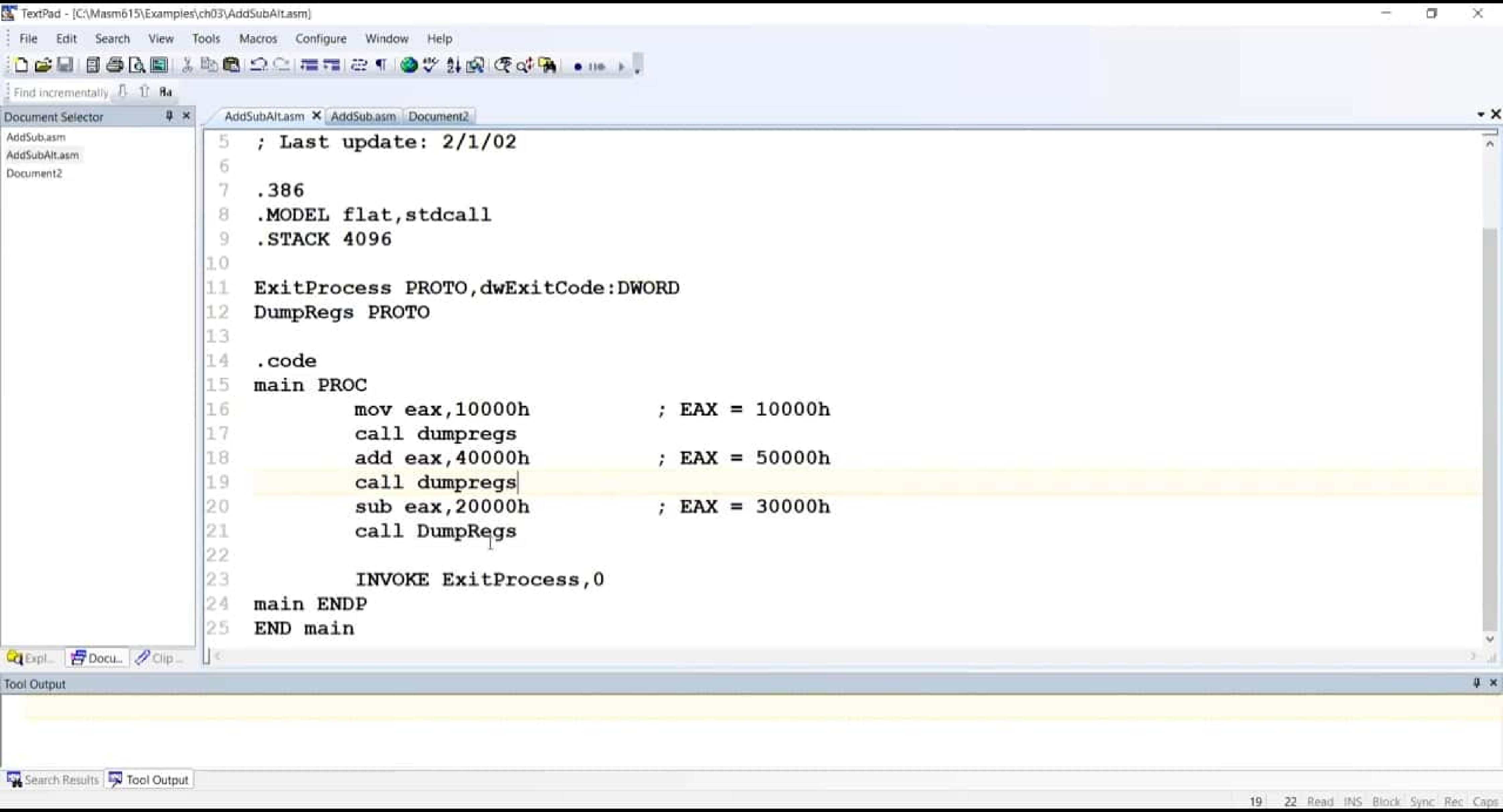
```
  EAX=00030000  EBX=002DA000  ECX=00401005  EDX=00401005
  ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
  EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0

Press any key to continue . . .
```



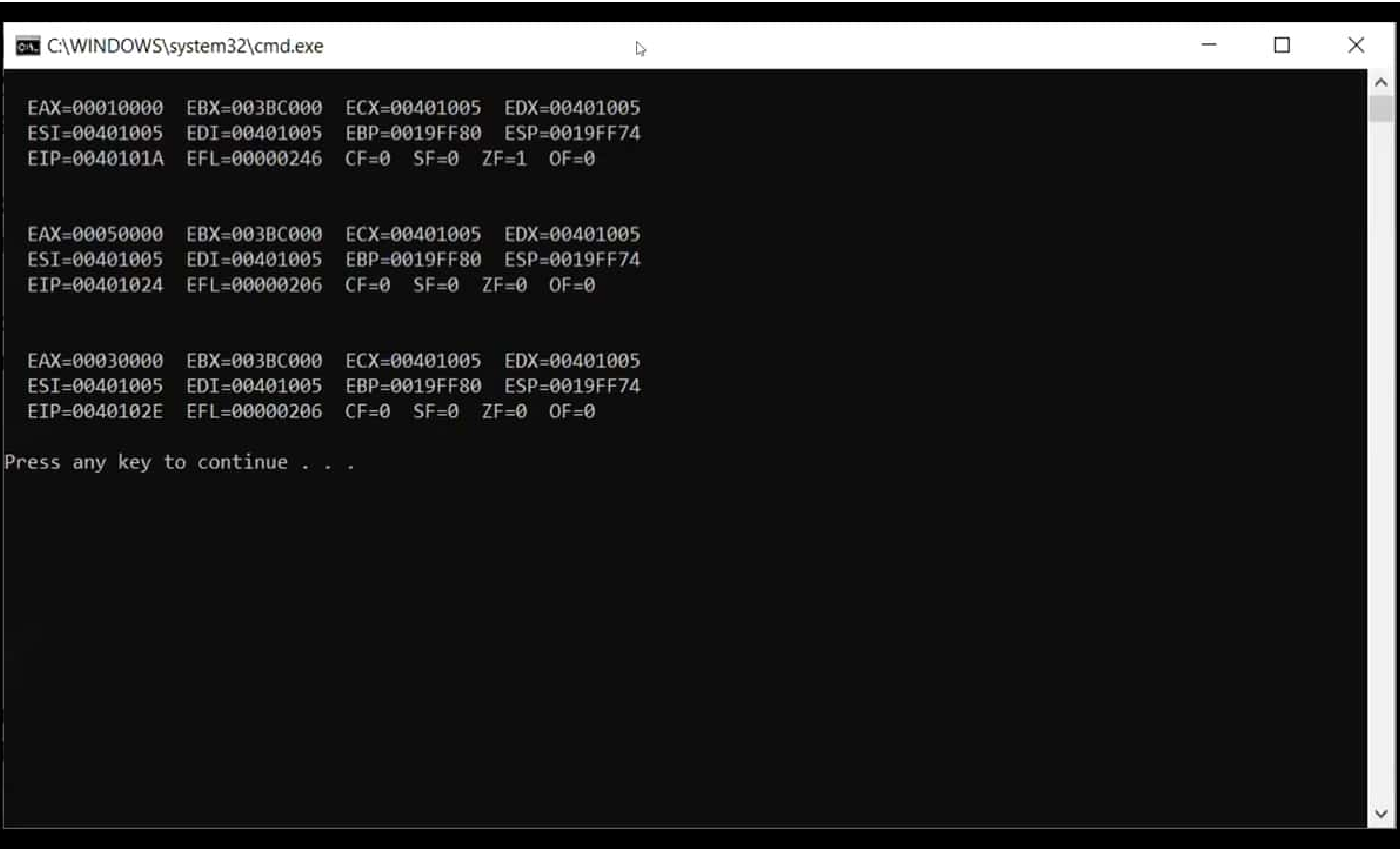
```
; Last update: 2/1/02

.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO,dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC
        mov eax,10000           ; EAX = 10000h
        add eax,40000           ; EAX = 50000h
        sub eax,20000           ; EAX = 30000h
        call DumpRegs

        INVOKE ExitProcess,0
main ENDP
END main
```



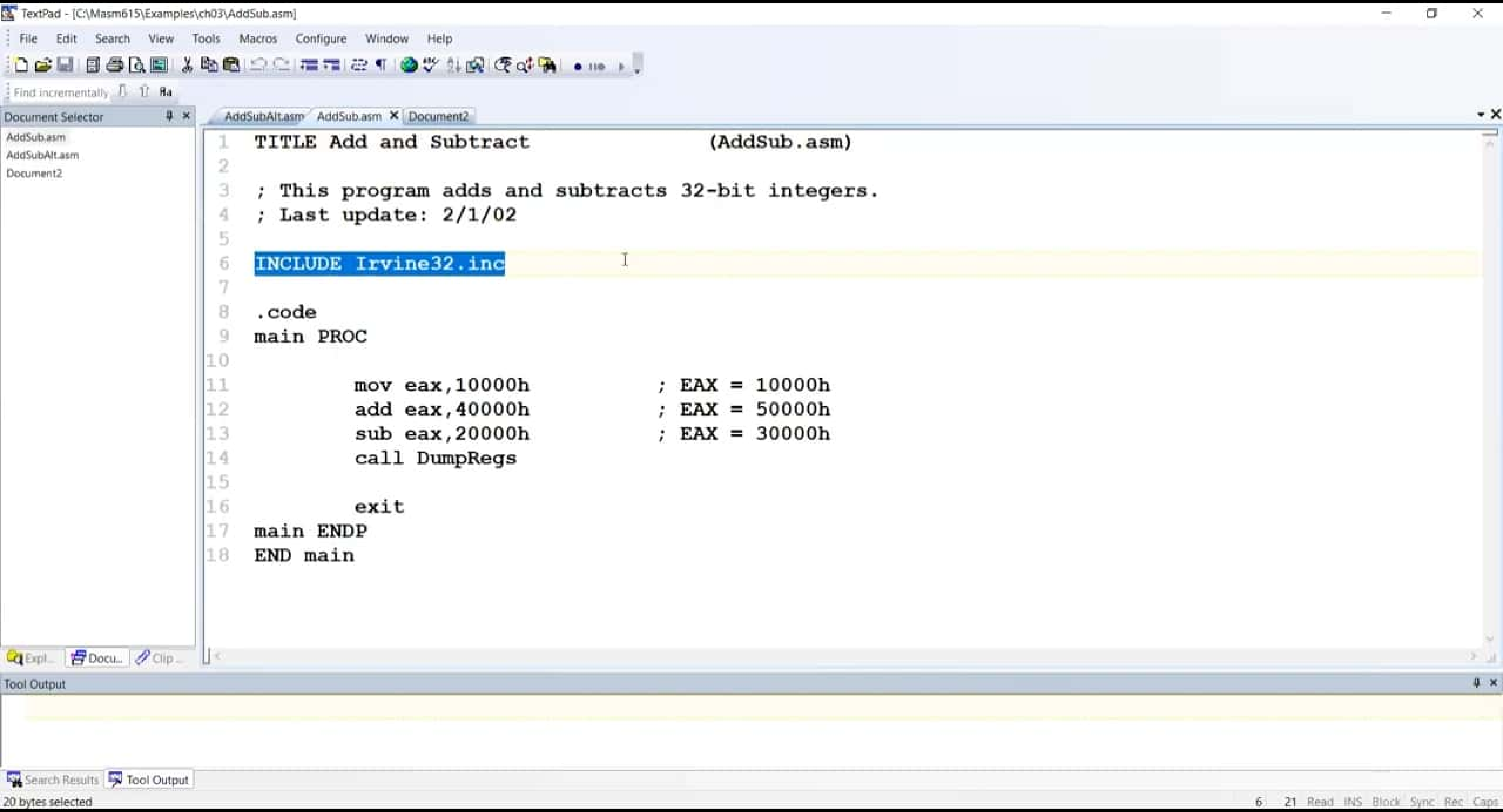
```
  EAX=00007530  EBX=003BD000  ECX=00401005  EDX=00401005
  ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
  EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0

Press any key to continue . . .
```



```
; Last update: 2/1/02

.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO,dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC
        mov eax,10000h          ; EAX = 10000h
        call dumpregs
        add eax,40000h          ; EAX = 50000h
        call dumpregs
        sub eax,20000h          ; EAX = 30000h
        call DumpRegs

        INVOKE ExitProcess,0
main ENDP
END main
```



```
  EAX=00010000  EBX=003BC000  ECX=00401005  EDX=00401005
  ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
  EIP=0040101A  EFL=00000246  CF=0  SF=0  ZF=1  OF=0


  EAX=00050000  EBX=003BC000  ECX=00401005  EDX=00401005
  ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
  EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0


  EAX=00030000  EBX=003BC000  ECX=00401005  EDX=00401005
  ESI=00401005  EDI=00401005  EBP=0019FF80  ESP=0019FF74
  EIP=0040102E  EFL=00000206  CF=0  SF=0  ZF=0  OF=0

Press any key to continue . . .
```



```
TITLE Add and Subtract              (AddSub.asm)

; This program adds and subtracts 32-bit integers.
; Last update: 2/1/02

INCLUDE Irvine32.inc

.code
main PROC

        mov eax,10000h          ; EAX = 10000h
        add eax,40000h          ; EAX = 50000h
        sub eax,20000h          ; EAX = 30000h
        call DumpRegs

        exit
main ENDP
END main
```