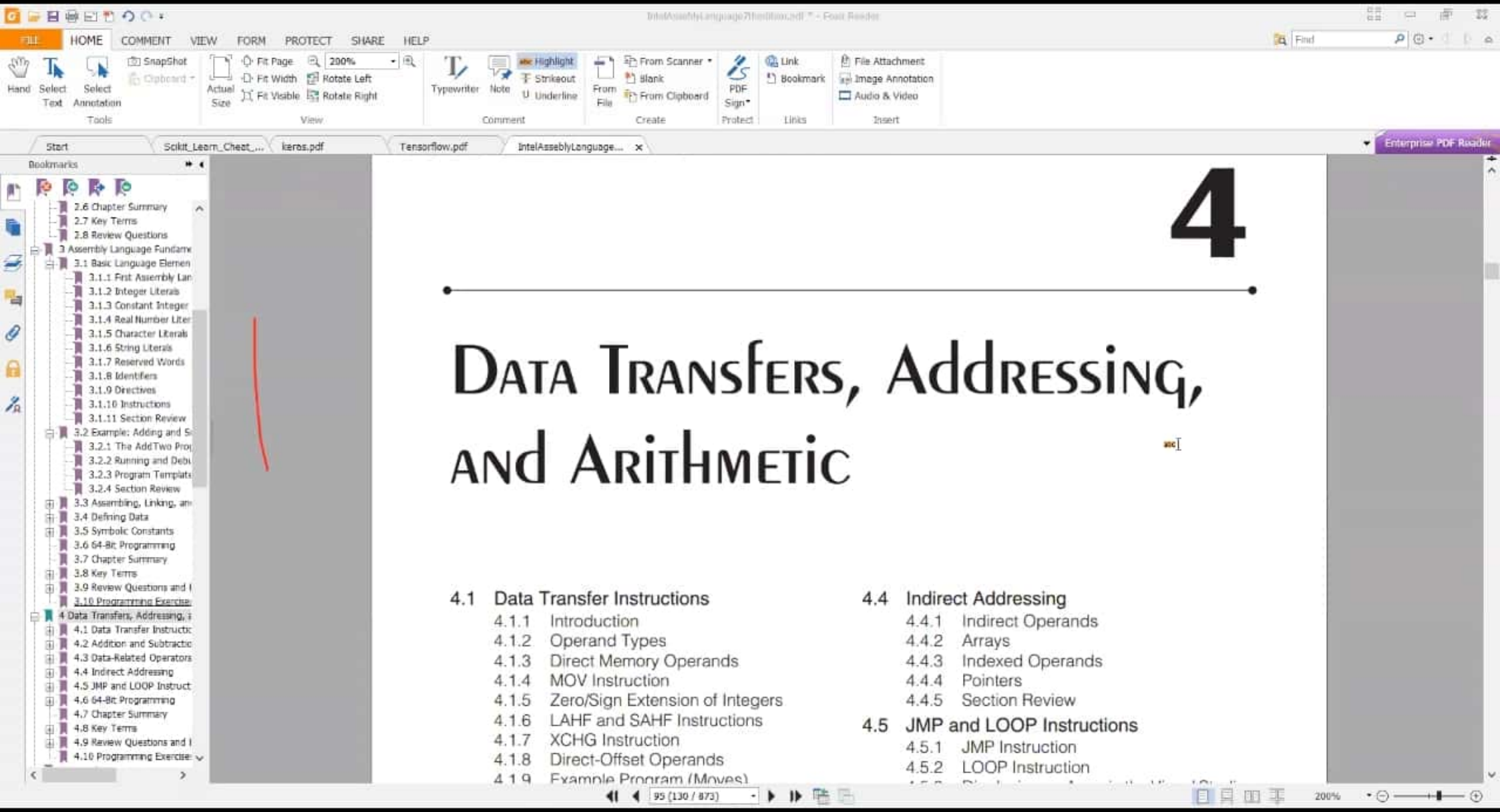


If you take the time to thoroughly learn the material presented in this chapter, the rest of this book will read a lot more smoothly. As the example programs become more complicated, you will rely on mastery of fundamental tools presented in this chapter.

4.1.2 Operand Types

Chapter 3 introduced x86 instruction formats:

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

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

```
mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]
```

There are three basic types of operands:

- Immediate—uses a numeric literal expression
- Register—uses a named register in the CPU
- Memory—references a memory location

Table 4-1 describes the standard operand types. It uses a simple notation for operands (in 32-bit mode) freely adapted from the Intel manuals. We will use it from this point on to describe the syntax of individual instructions.

4.1.3 Direct Memory Operands

Table 4-1 Instruction Operand Notation, 32-Bit Mode.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

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

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

```
mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]
```

There are three basic types of operands:

- Immediate—uses a numeric literal expression
- Register—uses a named register in the CPU
- Memory—references a memory location

Table 4-1 describes the standard operand types. It uses a simple notation for operands (in 32-bit mode) freely adapted from the Intel manuals. We will use it from this point on to describe the syntax of individual instructions.

4.1.3 Direct Memory Operands

Variable names are references to offsets within the data segment. For example, the following declaration for a variable named **var1** says that its size attribute is **byte** and it contains the value 10 hexadecimal:

mem	An 8-, 16-, or 32-bit memory operand

```
.data
var1 BYTE 10h
```

We can write instructions that dereference (look up) memory operands using their addresses. Suppose **var1** were located at offset 10400h. The following instruction copies its value into the AL register:

```
mov al var1
```

It would be assembled into the following machine instruction:

```
A0 00010400
```

The first byte in the machine instruction is the operation code (known as the *opcode*). The remaining part is the 32-bit hexadecimal address of **var1**. Although it might be possible to write programs using only numeric addresses, symbolic names such as **var1** make it easier to reference memory.

Alternative Notation. Some programmers prefer to use the following notation with direct operands because the brackets imply a dereference operation:

```
mov  al,[var1]
```

MASM permits this notation, so you can use it in your own programs if you want. Because so many programs (including those from Microsoft) are printed without the brackets, we will only

```
.data
var1 BYTE 10h
```

We can write instructions that dereference (look up) memory operands using their addresses. Suppose **var1** were located at offset 10400h. The following instruction copies its value into the AL register:

```
mov al var1
```

It would be assembled into the following machine instruction:

```
A0 00010400
```

The first byte in the machine instruction is the operation code (known as the *opcode*). The remaining part is the 32-bit hexadecimal address of **var1**. Although it might be possible to write programs using only numeric addresses, symbolic names such as **var1** make it easier to reference memory.

Alternative Notation. Some programmers prefer to use the following notation with direct operands because the brackets imply a dereference operation:

```
mov  al,[var1]
```

MASM permits this notation, so you can use it in your own programs if you want. Because so many programs (including those from Microsoft) are printed without the brackets, we will only use them in this book when an arithmetic expression is involved:

```
mov  al,[var1 + 5]
```

4.1.4 MOV Instruction

The MOV instruction copies data from a source operand to a destination operand. Known as a *data transfer* instruction, it is used in virtually every program. Its basic format shows that the first operand is the destination and the second operand is the source:

```
MOV destination,source
```

The destination operand's contents change, but the source operand is unchanged. The right to left movement of data is similar to the assignment statement in C++ or Java:

```
dest = source;
```

In nearly all assembly language instructions, the left-hand operand is the destination and the right-hand operand is the source. MOV is very flexible in its use of operands, as long as the following rules are observed:

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer register (IP, EIP, or RIP) cannot be a destination operand.

Here is a list of the standard MOV instruction formats:

```
MOV reg,reg
```

Here is a list of the standard MOV instruction formats:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
```

Memory to Memory A single MOV instruction cannot be used to move data directly from one memory location to another. Instead, you must move the source operand's value to a register before assigning its value to a memory operand:

```
.data
var1 WORD ?
var2 WORD ?
.code
mov  ax,var1
mov  var2,ax
```

You must consider the minimum number of bytes required by an integer constant when copying it to a variable or register. For unsigned integer constant sizes, refer to Table 1-4 in Chapter 1. For signed integer constants, refer to Table 1-7.

Overlapping Values

The following code example shows how the same 32-bit register can be modified using differently sized data. When **oneWord** is moved to AX, it overwrites the existing value of AL. When **oneDword** is moved to EAX, it overwrites AX. Finally, when 0 is moved to AX, it overwrites the lower half of EAX.

Memory to Memory – A single MOV instruction cannot be used to move data directly from one memory location to another. Instead, you must move the source operand's value to a register before assigning its value to a memory operand:

```
.data
var1 WORD ?
var2 WORD ?
.code
mov  ax,var1
mov  var2,ax
```

You must consider the minimum number of bytes required by an integer constant when copying it to a variable or register. For unsigned integer constant sizes, refer to Table 1-4 in Chapter 1. For signed integer constants, refer to Table 1-7.

Overlapping Values

The following code example shows how the same 32-bit register can be modified using differently sized data. When **oneWord** is moved to AX, it overwrites the existing value of AL. When **oneDword** is moved to EAX, it overwrites AX. Finally, when 0 is moved to AX, it overwrites the lower half of EAX.

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
```

```
.code
mov  eax,0            ; EAX = 00000000h
mov  al,oneByte       ; EAX = 00000078h
mov  ax,oneWord       ; EAX = 00001234h
mov  eax,oneDword     ; EAX = 12345678h
mov  ax,0             ; EAX = 12340000h
```

4.1.5 Zero/Sign Extension of Integers

Copying Smaller Values to Larger Ones

Although MOV cannot directly copy data from a smaller operand to a larger one, programmers can create workarounds. Suppose **count** (unsigned, 16 bits) must be moved to ECX (32 bits). We can set ECX to zero and move **count** to CX:

```
.data
count WORD 1
.code
mov ecx,0
mov cx,count
```

What happens if we try the same approach with a signed integer equal to −16?

Assembly Language - MOV-MOVZX-MOVSX - PowerPoint

File Home Insert Design Transitions Animations Slide Show Review View Foxit Reader PDF Tell me what you want to do... Sign in Share

Cut Copy Format Painter Paste Clipboard

New Slide Layout Reset Section Slides

Font

Text Direction Align Text Convert to SmartArt Paragraph

Arrange Quick Styles Shape Fill Shape Outline Shape Effects Drawing

Find Replace Select Editing

1 Assembly Language

2 MOV instruction

3 List of standard MOV instructions

4 MOV instruction Examples

5 MOV with Zero Extension instruction (MOVZX)

6 MOV with Zero Extension instruction (MOVZX)

STUDY POINT

Assembly Language

FOR

X86 PROCESSORS

Sheikh Muhammad Aamir

Department of Computer Science

GC University, Faisalabad

STUDY POINT

MOV Instruction

- MOV instruction is used to copy data from source operand to destination operand
- Syntax:

MOV Destination_Operand, Source_Operand

Example:

MOV EAX, 12345678H

Rules:

- Both operands must be of same sizes
- Both operands must not be memory variables
- EIP/IP/Immediate value can't be used as a destination operand

STUDY POINT

1 Assembly Language
2 MOV instruction
3 List of standard MOV instructions
4 MOV instruction Examples
5 MOV with Zero Extension instruction (MOVZX)
6 MOV with Zero Extension instruction (MOVZX)

List of standard MOV Instructions

- MOV REG, REG (register to register)
- MOV REG, MEM (memory variable to register)
- MOV REG, IMM (immediate value to register)
- MOV MEM, REG (register to memory variable)
- MOV MEM, IMM (immediate value to memory variable)

STUDY POINT

MOV Instruction Examples

Some Valid MOV Instructions	Some In-Valid MOV Instruction
MOV X, 75	MOV EBX, AL
MOV AL, 75	MOV AL, 256
MOV AL, -75	MOV X, Y
MOV AX, 1234H	MOV EIP, 12345678H
MOV EBX, ECX	MOV AL, 100H
MOV EAX, 12345678H	MOV 50, AL

STUDY POINT

1 Assembly Language
2 MOV instruction
3 List of standard MOV instructions
4 MOV instruction Examples
5 MOV with Zero Extension instruction (MOVZX)
6 MOV with Zero Extension instruction (MOVZX)

MOV with Zero Extension Instruction (MOVZX)

- MOVZX instruction copies the source operand value (with zero extension) into destination operand if source operand size is smaller than destination operand size.
- MOVZX is only used with un-signed data

Syntax:

MOVZX Destination_Operand, Source_Operand

Like

MOVZX *reg32, reg/mem8*

MOVZX *reg32, reg/mem16*

MOVZX *reg16, reg/mem8*

STUDY POINT

MOV with Zero Extension Instruction (MOVZX)

Examples:

```
.DATA
VAL Byte 55H
TEMP Byte 11011011B          ;8 Bits Binary value

.CODE
MOVZX EAX, VAL               ; EAX = 00000055H

MOVZX BX, TEMP               ; BX = 0000000011011011
```

STUDY POINT

3
4
5
6
7
8

MOV with Sign Bit Extension Instruction (MOVSX)

- MOVSX instruction copies the source operand value (with sign bit extension) into destination operand if source operand size is smaller than destination operand size.

- MOVSX is only used with Signed data
- MSB is 0 if the value is +ve
- MSB is 1 if the value is –ve
- Negative number is represented in memory as two's compliment of original positive number

Syntax:

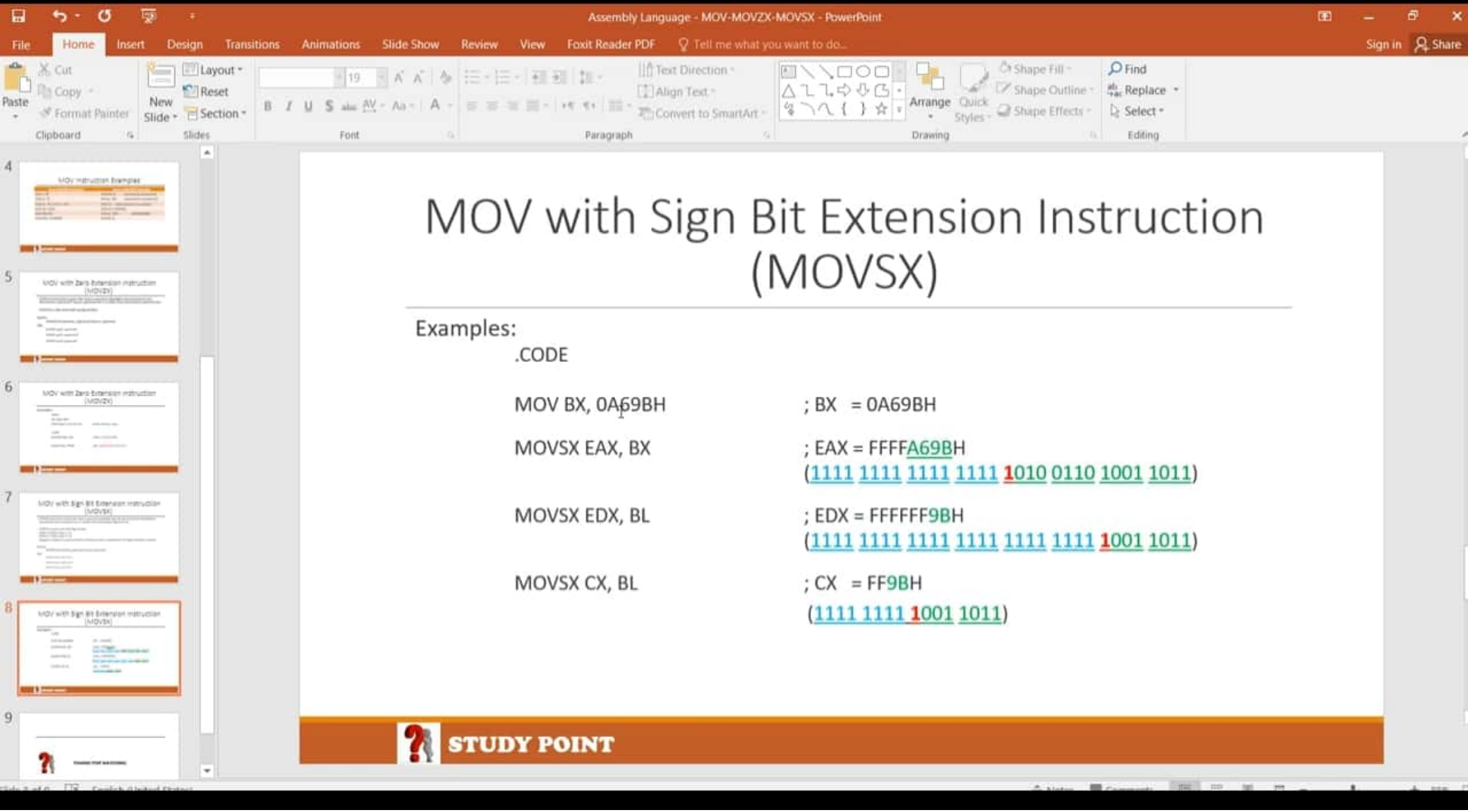
MOVSX Destination_Operand, Source_Operand

Like

MOVSX *reg32, reg/mem8*

MOVSX *reg32, reg/mem16*

MOVSX *reg16, reg/mem8*



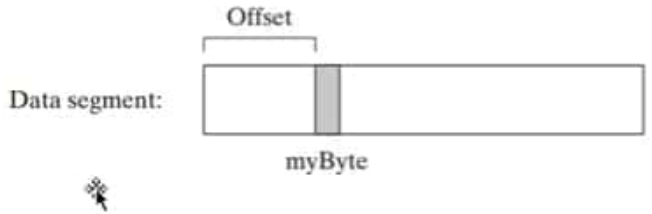
Data related Operators and Directives

- Operators and directives are not executable instructions but they are interpreted by Assembler
- Some Operators are :
  - OFFSET
  - PTR
  - TYPE
  - LENGTHOF
  - SIZEOF

OFFSET OPERATOR

- OFFSET operator returns the **offset address** of data label
- OFFSET presents the distance of data label from the beginning of data segment

A variable named myByte.



OFFSET OPERATOR

Examples:

```
.data
X BYTE ?
Y WORD ?
Z DWORD ?
MESSAGE BYTE “HELLO WORLD”, 0
.CODE
MOV ESI, OFFSET X          ; ESI = 00404000
MOV ESI, OFFSET Y          ; ESI = 00404001
MOV ESI, OFFSET            ; ESI = 00404003
MOV ESI, OFFSET MESSAGE    ; ESI = 00404007
```

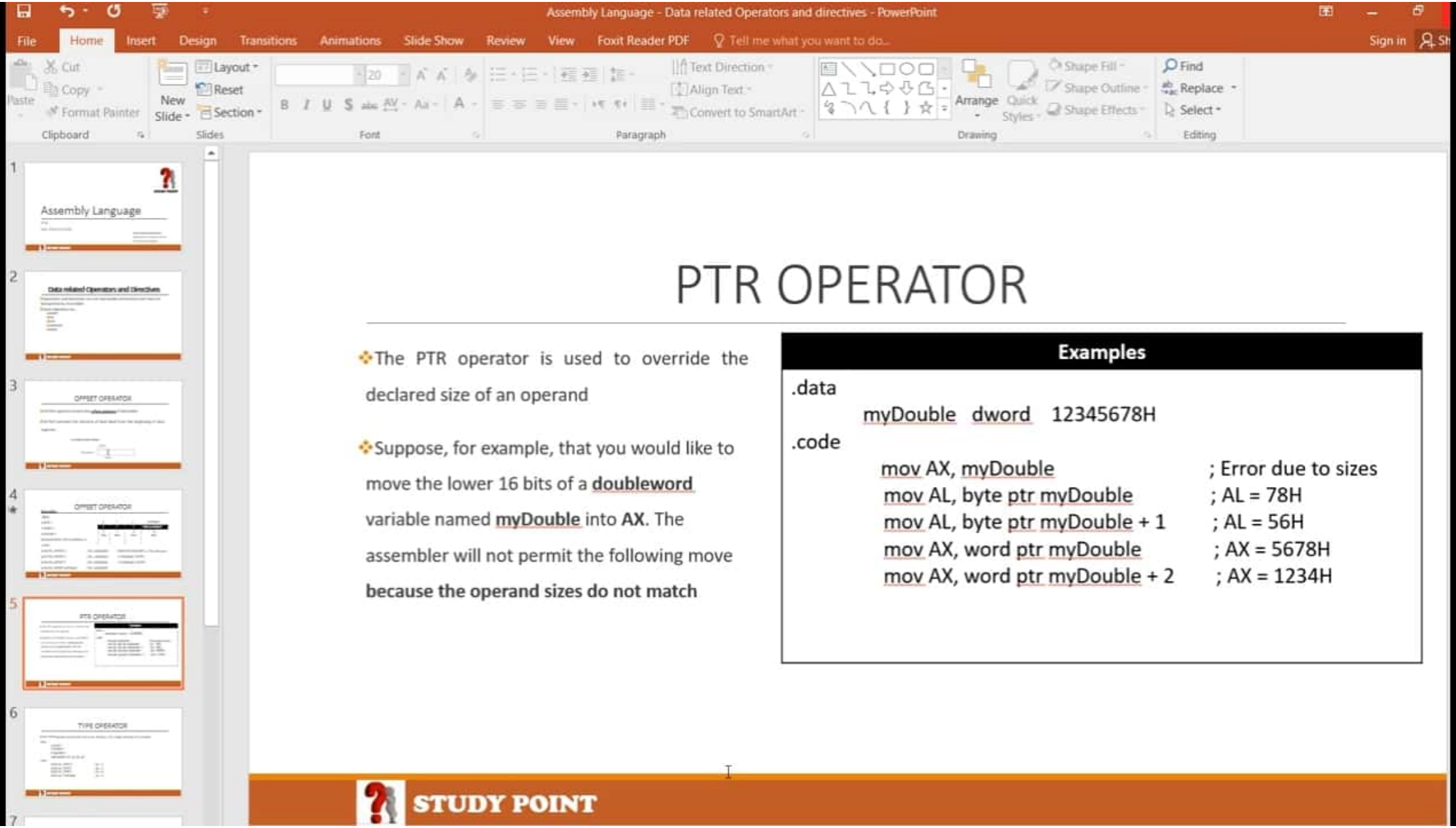
X	Y	Z	MESSAGE
?	?	?	“HELLO WORLD”
00404000	00404001	00404003	00404007
1 byte	2 byte	4 byte	11 byte

START OF SEGMENT i.e. Zero Distance

Y CONTAINS 2 BYTES

Z CONTAINS 4 BYTES

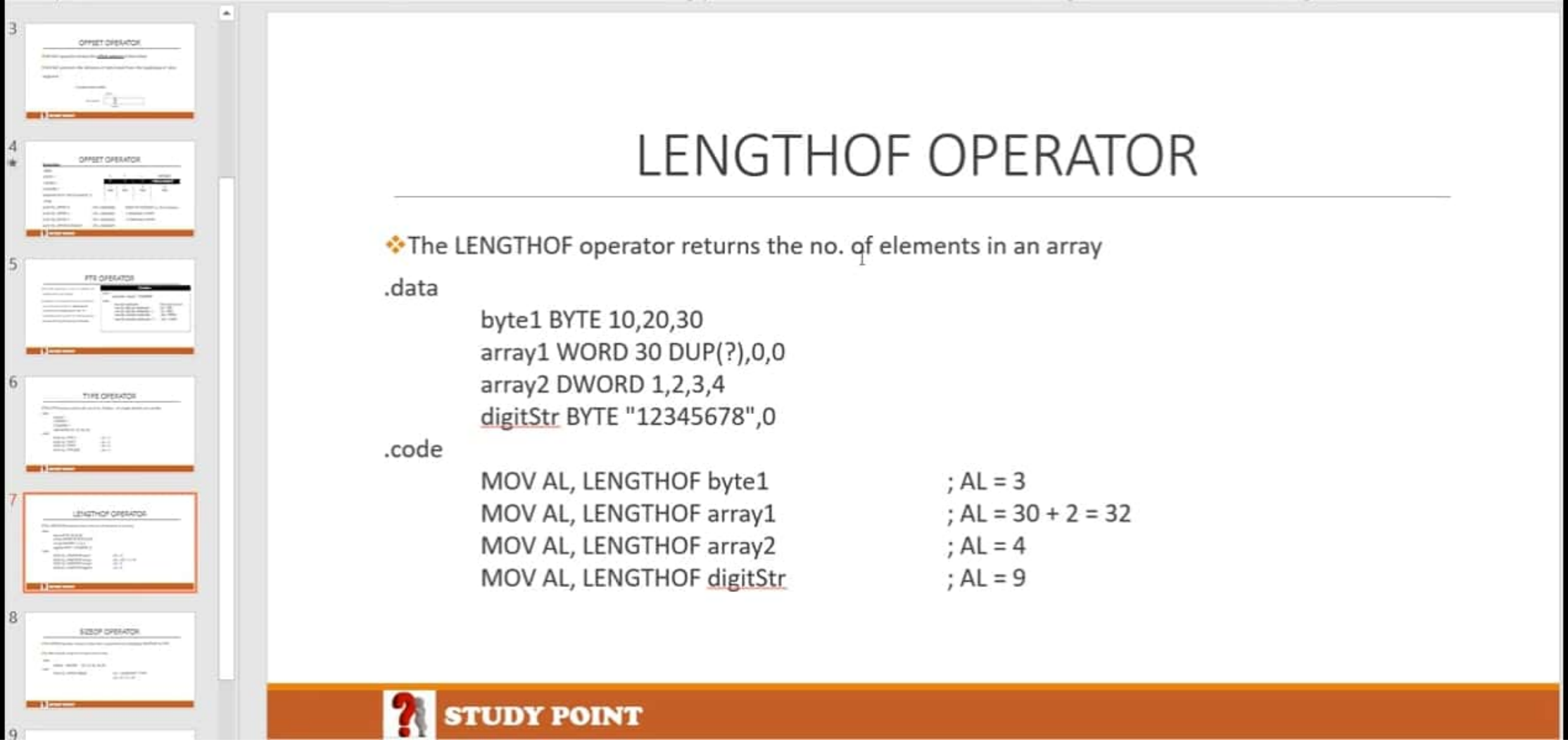
STUDY POINT



TYPE OPERATOR

- The TYPE operator returns the size in no. of bytes, of a single element of a variable.

```
.data
        X BYTE ?
        Y WORD ?
        Z DWORD ?
        ARR WORD 10, 20, 30, 40
.code
        MOV AL, TYPE X          ; AL = 1
        MOV AL, TYPE Y          ; AL = 2
        MOV AL, TYPE Z          ; AL = 4
        MOV AL, TYPE ARR        ; AL = 2
```

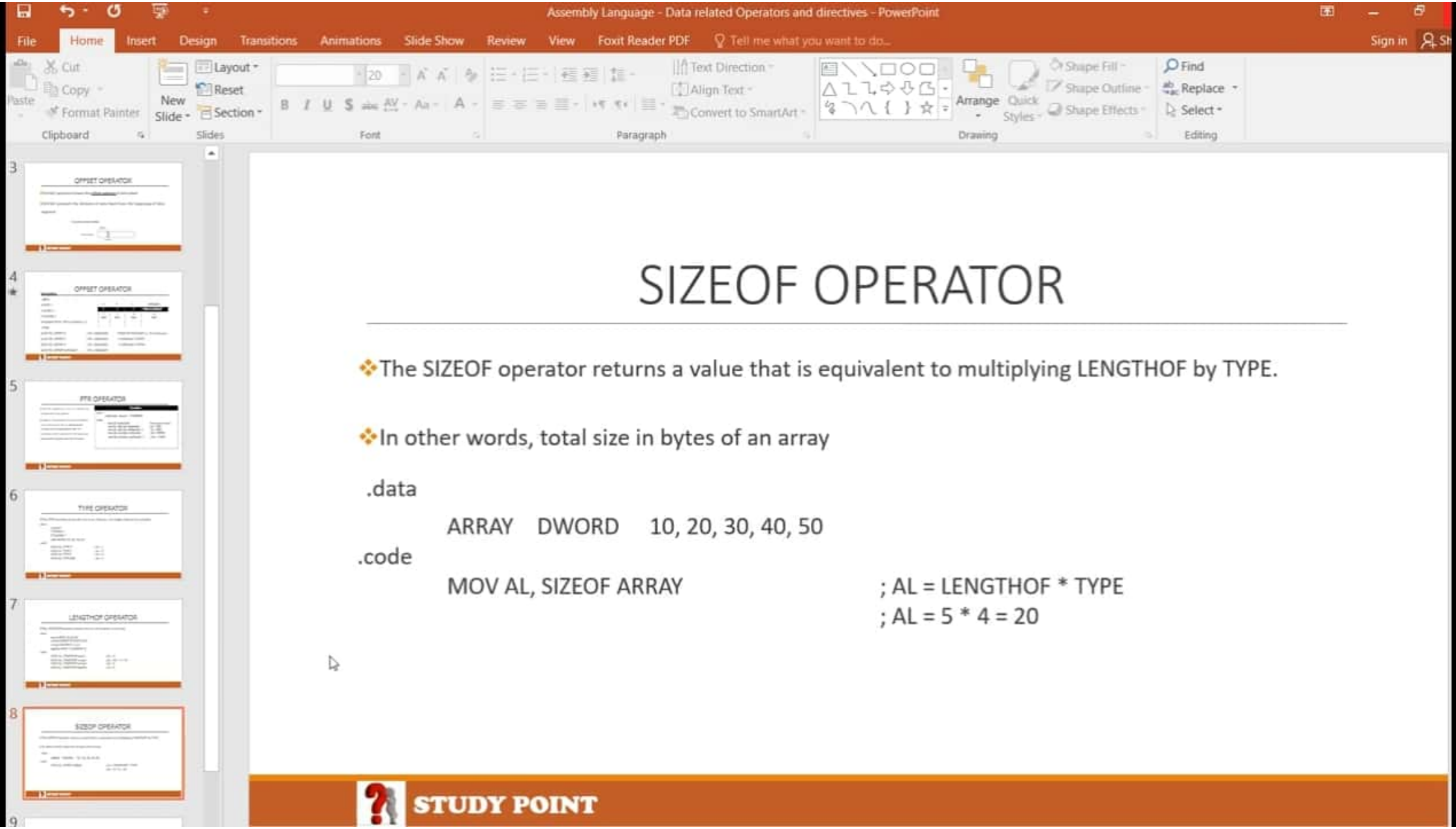


LENGTHOF OPERATOR

- The LENGTHOF operator returns the no. of elements in an array

```
.data
        byte1 BYTE 10,20,30
        array1 WORD 30 DUP(?),0,0
        array2 DWORD 1,2,3,4
        digitStr BYTE "12345678",0
.code
        MOV AL, LENGTHOF byte1          ; AL = 3
        MOV AL, LENGTHOF array1         ; AL = 30 + 2 = 32
        MOV AL, LENGTHOF array2         ; AL = 4
        MOV AL, LENGTHOF digitStr       ; AL = 9
```

STUDY POINT



SIZEOF OPERATOR

- The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.
- In other words, total size in bytes of an array

```
.data
        ARRAY   DWORD    10, 20, 30, 40, 50
.code
        MOV AL, SIZEOF ARRAY                ; AL = LENGTHOF * TYPE
                                            ; AL = 5 * 4 = 20
```

STUDY POINT

4. *(True/False):* The LENGTHOF operator returns the number of bytes in an operand.
5. *(True/False):* The SIZEOF operator returns the number of bytes in an operand.

4.4 Indirect Addressing

Direct addressing is rarely used for array processing because it is impractical to use constant offsets to address more than a few array elements. Instead, we use a register as a pointer (called *indirect addressing*) and manipulate the register's value. When an operand uses indirect addressing, it is called an *indirect operand*.

4.4.1 Indirect Operands

Protected Mode An indirect operand can be any 32-bit general-purpose register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) surrounded by brackets. The register is assumed to contain the address of some data. In the next example, ESI contains the offset of **byteVal**. The MOV instruction uses the indirect operand as the source, the offset in ESI is dereferenced, and a byte is moved to AL:

```
.data
byteVal BYTE 10h
.code
mov  esi,OFFSET byteVal
mov  al,[esi]                    ; AL = 10h
```

If the destination operand uses indirect addressing, a new value is placed in memory at the location pointed to by the register. In the following example, the contents of the BL register are copied to the memory location addressed by ESI.

```
mov [esi],bl
```

Using PTR with Indirect Operands The size of an operand may not be evident from the context of an instruction. The following instruction causes the assembler to generate an "operand must have size" error message:

```
inc [esi]                    ; error: operand must have size
```

The assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

```
inc BYTE PTR [esi]
```

4.4.2 Arrays

Indirect operands are ideal tools for stepping through arrays. In the next example, **arrayB** contains 3 bytes. As ESI is incremented, it points to each byte, in order:

```
.data
arrayB BYTE 10h,20h,30h
```

Using PTR with Indirect Operands The size of an operand may not be evident from the context of an instruction. The following instruction causes the assembler to generate an "operand must have size" error message:

```
inc [esi]                    ; error: operand must have size
```

The assembler does not know whether ESI points to a byte, word, doubleword, or some other size. The PTR operator confirms the operand size:

```
inc BYTE PTR [esi]
```

4.4.2 Arrays

Indirect operands are ideal tools for stepping through arrays. In the next example, **arrayB** contains 3 bytes. As ESI is incremented, it points to each byte, in order:

```
.data
arrayB  BYTE 10h,20h,30h
.code
mov  esi,OFFSET arrayB
mov  al,[esi]                ; AL = 10h
inc  esi
mov  al,[esi]                ; AL = 20h
inc  esi
mov  al,[esi]                ; AL = 30h
```

If we use an array of 16-bit integers, we add 2 to ESI to address each subsequent array element:

```
.data
```

```
mov  al,[esi]                    ; AL = 30h
```

If we use an array of 16-bit integers, we add 2 to ESI to address each subsequent array element:

```
.data
arrayW  WORD 1000h,2000h,3000h
.code
mov  esi,OFFSET arrayW
mov  ax,[esi]                    ; AX = 1000h
add  esi,2
mov  ax,[esi]                    ; AX = 2000h
add  esi,2
mov  ax,[esi]                    ; AX = 3000h
```

Suppose **arrayW** is located at offset 10200h. The following illustration shows the initial value of ESI in relation to the array data:

Offset	Value	
10200	1000h	←[esi]
10202	2000h	
10204	3000h	

Example: Adding 32-Bit Integers The following code example adds three doublewords. A displacement of 4 must be added to ESI as it points to each subsequent array value because doublewords are 4 bytes long:

```
.data
arrayD DWORD 10000h,20000h,30000h
.code
mov  esi,OFFSET arrayD
mov  eax,[esi]          ; first number
add  esi,4
add  eax,[esi]          ; second number
add  esi,4
add  eax,[esi]          ; third number
```

Suppose **arrayD** is located at offset 10200h. Then the following illustration shows the initial value of ESI in relation to the array data:

Offset	Value	
10200	10000h	← [esi]
10204	20000h	← [esi] + 4
10208	30000h	← [esi] + 8

Example: Adding 32-Bit Integers The following code example adds three doublewords. A displacement of 4 must be added to ESI as it points to each subsequent array value because doublewords are 4 bytes long:

```
.data
arrayD DWORD 10000h,20000h,30000h
.code
mov  esi,OFFSET arrayD
mov  eax,[esi]          ; first number
add  esi,4
add  eax,[esi]          ; second number
add  esi,4
add  eax,[esi]          ; third number
```

Suppose **arrayD** is located at offset 10200h. Then the following illustration shows the initial value of ESI in relation to the array data:

Offset	Value	
10200	10000h	← [esi]
10204	20000h	← [esi] + 4
10208	30000h	← [esi] + 8

4.4.3 Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the

Offset	Value	
10200	10000h	← [esi]
10204	20000h	← [esi] + 4
10208	30000h	← [esi] + 8

4.4.3 Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the 32-bit general-purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

```
constant[reg]
[constant + reg]
```

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable's offset. Here are examples that show both notational forms:

`arrayB[esi]`	`[arrayB + esi]`
`arrayD[ebx]`	`[arrayD + ebx]`

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

forms permitted by MASM (the brackets are part of the notation):

```
constant[reg]
[constant + reg]
```

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable's offset. Here are examples that show both notational forms:

`arrayB[esi]`	`[arrayB + esi]`
`arrayD[ebx]`	`[arrayD + ebx]`

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

```
.data
arrayB BYTE 10h,20h,30h
.code
mov  esi,0
mov  al,arrayB[esi]              ; AL = 10h
```

The last statement adds ESI to the offset of **arrayB**. The address generated by the expression **[arrayB + ESI]** is dereferenced and the byte in memory is copied to AL.

Adding Displacements The second type of indexed addressing combines a register with a constant offset. The index register holds the base address of an array or structure, and the constant identifies offsets of various array elements. The following example shows how to do this with an array of 16-bit words:

```
.data
arrayW  WORD 1000h,2000h,3000h
.code
mov  esi,OFFSET arrayW
mov  ax,[esi]                ; AX = 1000h
mov  ax,[esi+2]              ; AX = 2000h
mov  ax,[esi+4]              ; AX = 3000h
```

Using 16-Bit Registers It is usual to use 16-bit registers as indexed operands in real-address mode. In that case, you are limited to using SI, DI, BX, or BP:

```
mov  al,arrayB[si]
mov  ax,arrayW[di]
mov  eax,arrayD[bx]
```

```
mov  esi,3                          ; subscript
mov  eax,arrayD[esi*TYPE arrayD]    ; EAX = 4
```

4.4.4 Pointers

A variable containing the address of another variable is called a *pointer*. Pointers are a great tool for manipulating arrays and data structures because the address they hold can be modified at runtime. You might use a system call to allocate (reserve) a block of memory, for example, and save the address of that block in a variable. A pointer's size is affected by the processor's current mode (32-bit or 64-bit). In the following 32-bit code example, **ptrB** contains the offset of arrayB:

```
.data
arrayB byte 10h,20h,30h,40h
ptrB dword arrayB
```

Optionally, you can declare **ptrB** with the OFFSET operator to make the relationship clearer:

```
ptrB dword OFFSET arrayB
```

The 32-bit mode programs in this book use near pointers, so they are stored in doubleword variables. Here are two examples: **ptrB** contains the offset of **arrayB**, and **ptrW** contains the offset of **arrayW**:

```
arrayB   BYTE   10h,20h,30h,40h
arrayW   WORD   1000h,2000h,3000h
ptrB     DWORD  arrayB
ptrW     DWORD  arrayW
```

Optionally, you can use the OFFSET operator to make the relationship clearer:

manipulating arrays and data structures because the address they hold can be modified at runtime. You might use a system call to allocate (reserve) a block of memory, for example, and save the address of that block in a variable. A pointer's size is affected by the processor's current mode (32-bit or 64-bit). In the following 32-bit code example, **ptrB** contains the offset of arrayB:

```
.data
arrayB byte 10h,20h,30h,40h
ptrB dword arrayB
```

Optionally, you can declare **ptrB** with the OFFSET operator to make the relationship clearer:

```
ptrB dword OFFSET arrayB
```

The 32-bit mode programs in this book use near pointers, so they are stored in doubleword variables. Here are two examples: **ptrB** contains the offset of **arrayB**, and **ptrW** contains the offset of **arrayW**:

```
arrayB   BYTE   10h,20h,30h,40h
arrayW   WORD   1000h,2000h,3000h
ptrB     DWORD  arrayB
ptrW     DWORD  arrayW
```

Optionally, you can use the OFFSET operator to make the relationship clearer:

```
ptrB   DWORD OFFSET arrayB
ptrW   DWORD OFFSET arrayW
```

High-level languages purposely hide physical details about pointers because their implementa-

1 Assembly Language
2 JMP & LOOP Instructions
3 JMP & LOOP Instructions
4 JMP & LOOP Instructions
5
6
7

JMP & LOOP Instructions

JMP instruction

- The JMP instruction causes an **unconditional transfer to a destination label**, identified by a code label that is translated by the assembler into an offset.
- Address of destination label is placed in EIP register

Syntax:

JMP LABEL_NAME

Example:

```
disp:
    mov eax, 100
    call writedec
    jmp disp
```

JMP & LOOP Instructions

LOOP instruction

- LOOP instruction is used to execute a statement or set of statements for a specific number of times (no. of iteration value is placed in ECX register before using LOOP instruction).
- ECX is automatically used as loop counter
- **Syntax:**

  LOOP LABEL_NAME

- **Working:**
  - ECX is decremented by 1 automatically when control approaches to LOOP instruction
  - Compare the value of ECX with 0 (zero)
    - If the value of **ECX ≠ 0** then control is transferred to destination label and iteration is performed again
    - Otherwise loop will be terminated and control is transferred to the next statement of LOOP instruction

1 2 3 4 5 6 7

JMP & LOOP Instructions

LOOP instruction

Example: To display the all odd numbers from 1 to 99

```
include irvine32.inc
.code
main proc
        mov ecx , 50
        mov eax, 1
    Disp:
        call writedec
        add eax, 2
    Loop Disp
        exit
main endp
end main
```






LOOP Instruction

Example: Assembly Program to calculate the factorial of a given number.

```
include irvine32.inc
.data
message1 byte "Enter any value to find factorial : ", 0
Message2 byte "The factorial of given number is : ",0
.code
main proc
        mov edx, offset message1
        call writestring
        call readint            ; just like  cin >> eax;

        ; n! = n * (n-1)!
        ; 5 ! = 5 * 4 * 3 * 2 * 1 = 120

        mov ecx , eax
        mov eax, 1
        factorial:
                mul ecx
        Loop factorial

mov edx, offset message2
call writestring
call writedec
Call crlf

exit
main endp
end main
```

STUDY POINT