# Addressing modes

The different ways in which the location of the operand is specified in an instruction are referred to as addressing modes.

## 1. Implicit addressing mode

- the operand is not written in the instruction.

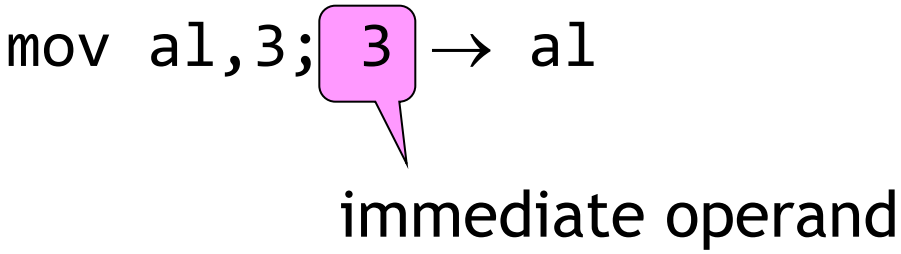
```
mul bx; bx * ax → dx:ax
```

implicit operands

# 2. Explicit addressing modes

## Immediate operand

- the number on which the instruction is to operate is written in the instruction.

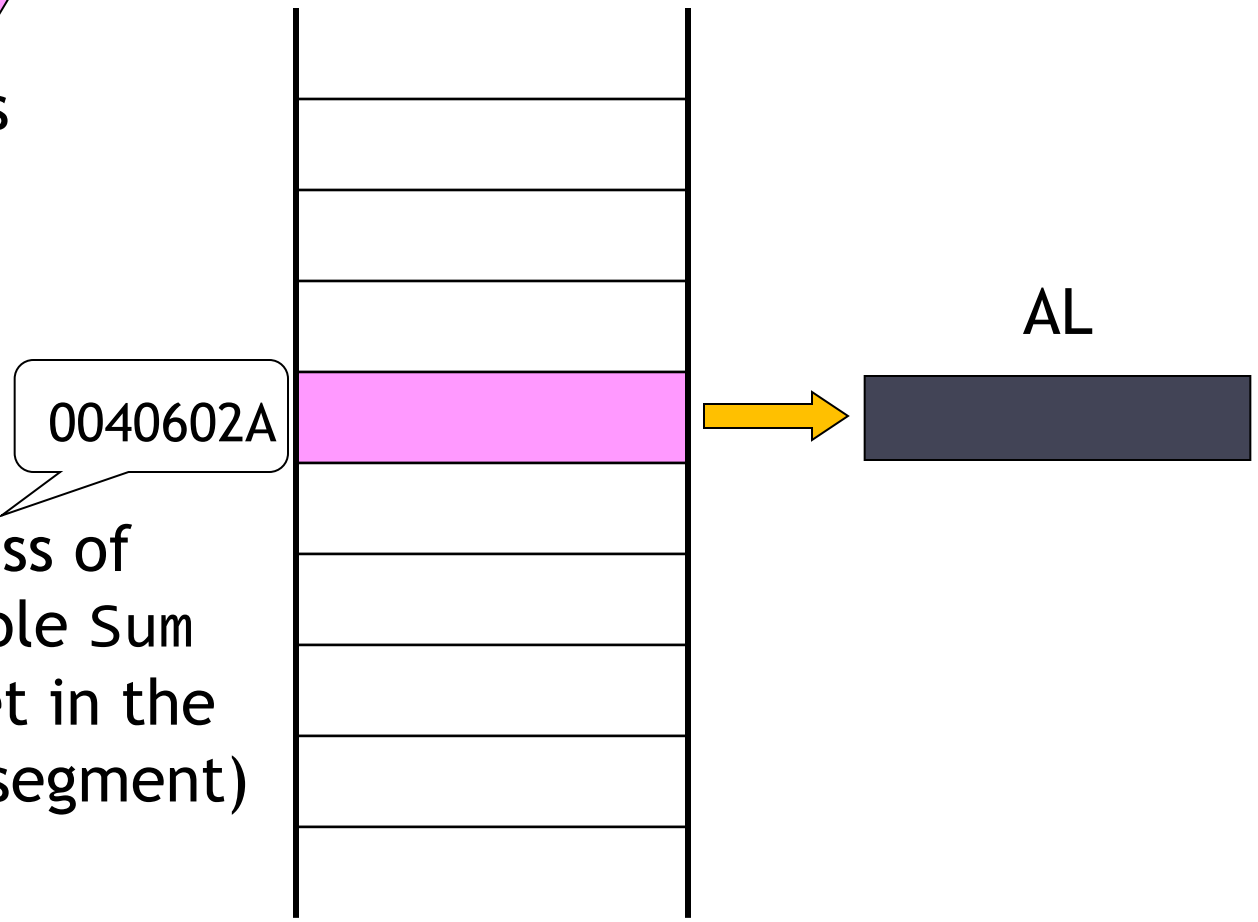


# Direct addressing mode

- the name of a register or the name of a variable.

```
mov al,Sum;   [Sum] → al
```

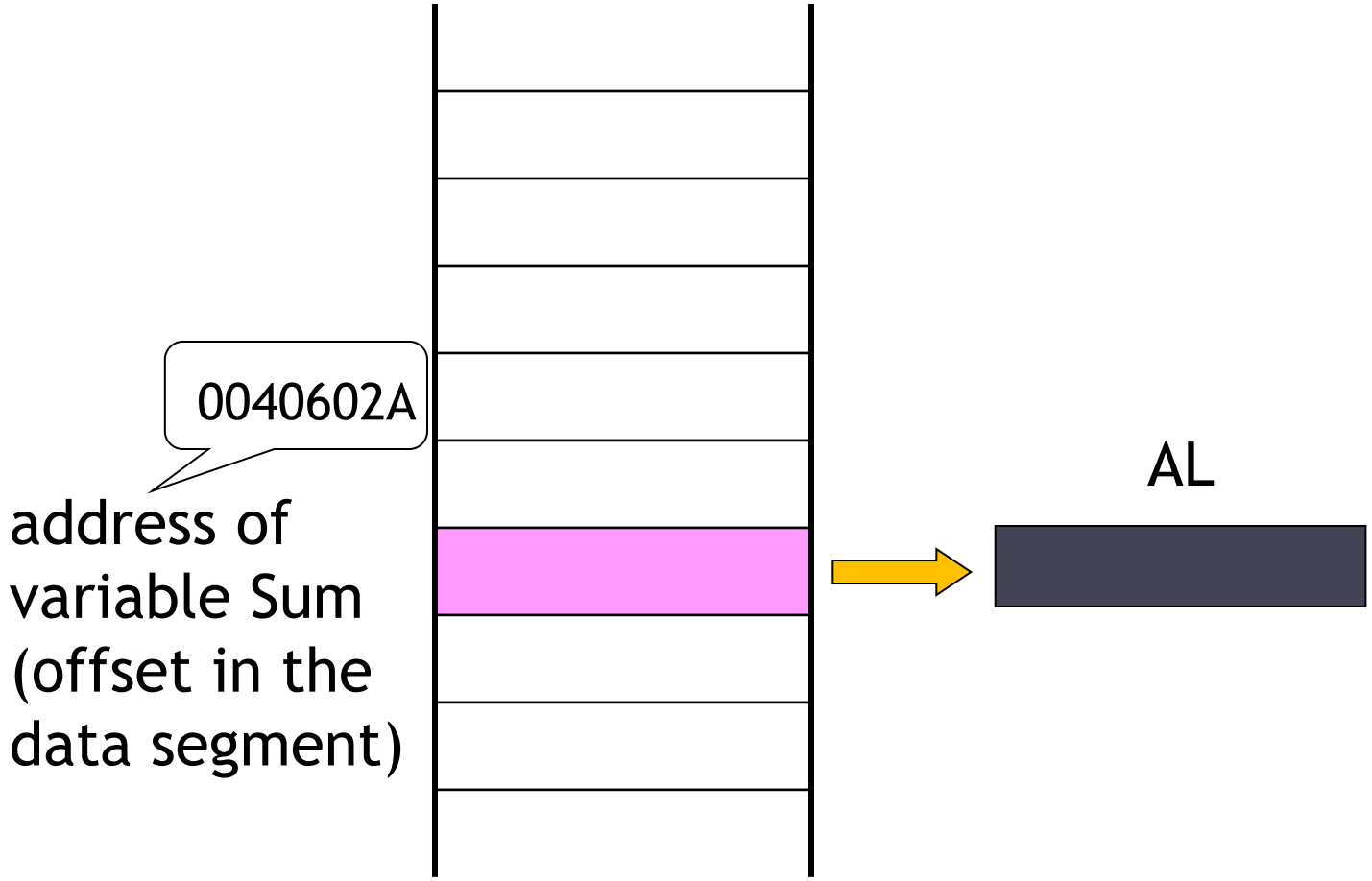
direct address



Direct address can be combined with a constant index:

```
mov al,Sum+2; [Sum+2] → al

mov al,Sum[2]; [Sum+2] → al
```



# Indirect register addressing modes

Register(s) in the instruction contains the address of the operand.

The register may be a base register and/or index register. A constant displacement may be added.

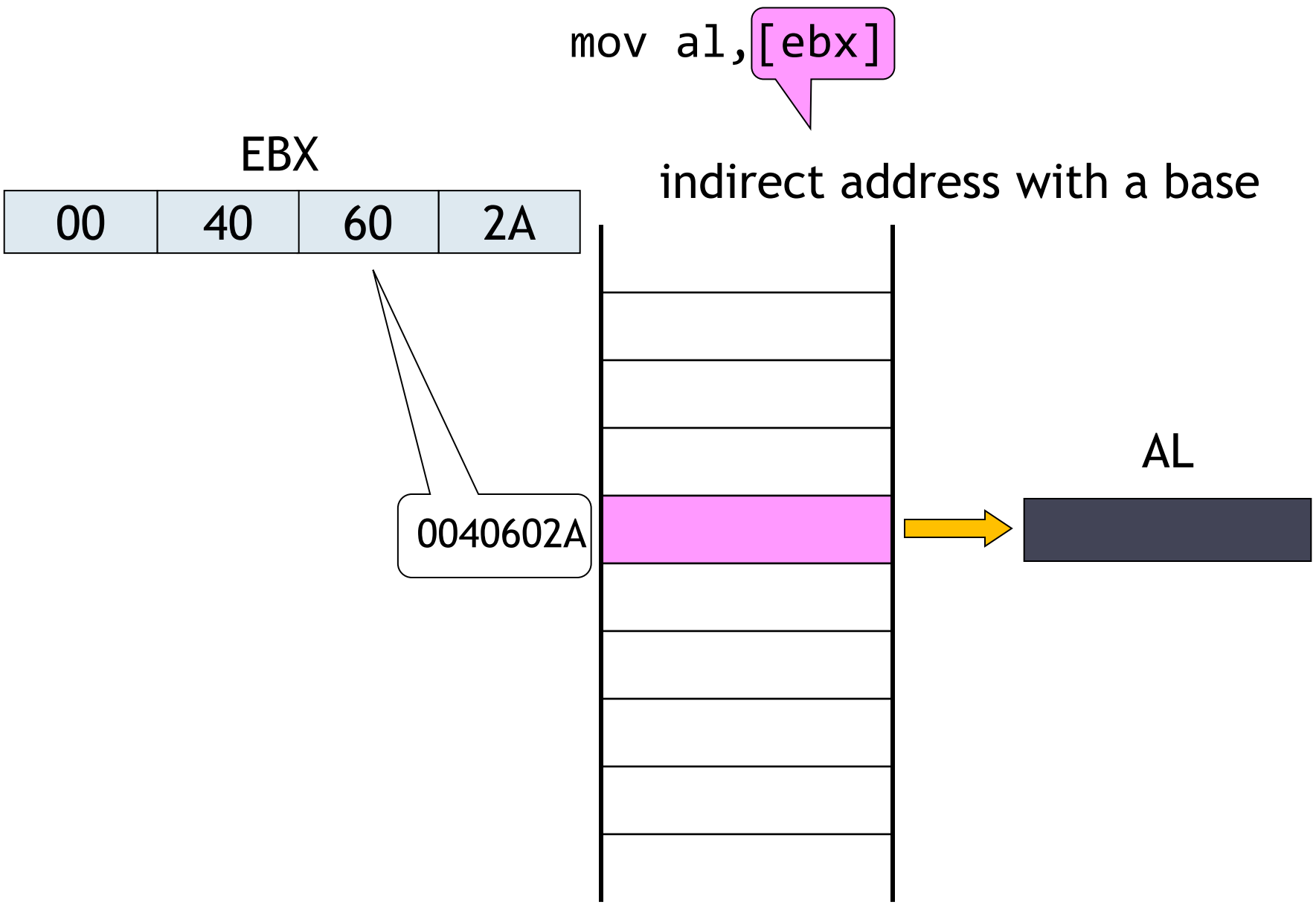
Address = base + index + displacement

Protected mode:

Every 32-bit general-purpose register can be used as a base register or (except ESP) as an index register.

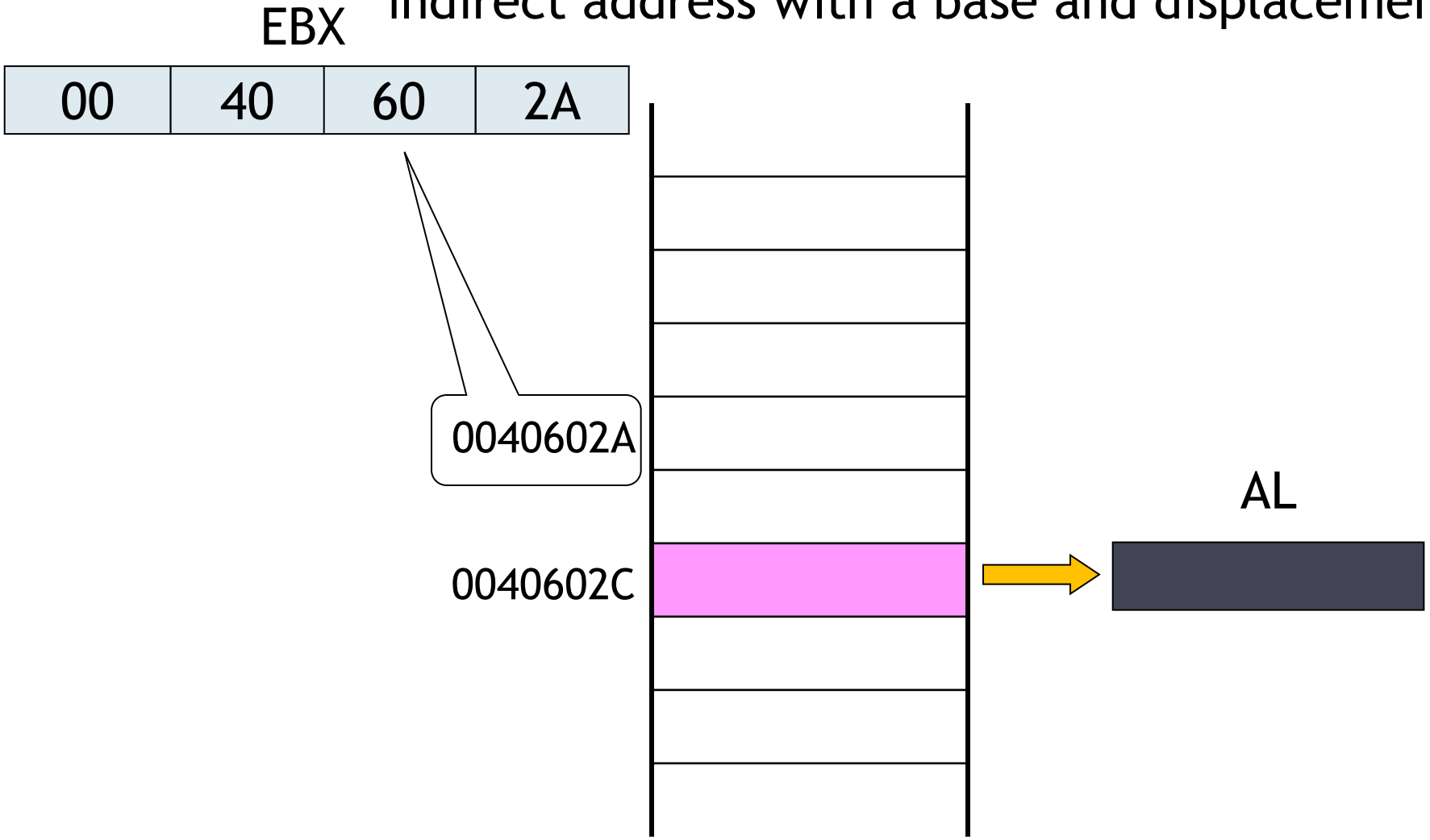
Which memory segment will be used depends on base register:

- EAX, EBX, ECX, EDX, ESI, EDI => data segment.
- EBP, ESP => stack segment.


mov al,[ebx]
EBX
00
40
60
2A
indirect address with a base
0040602A
AL

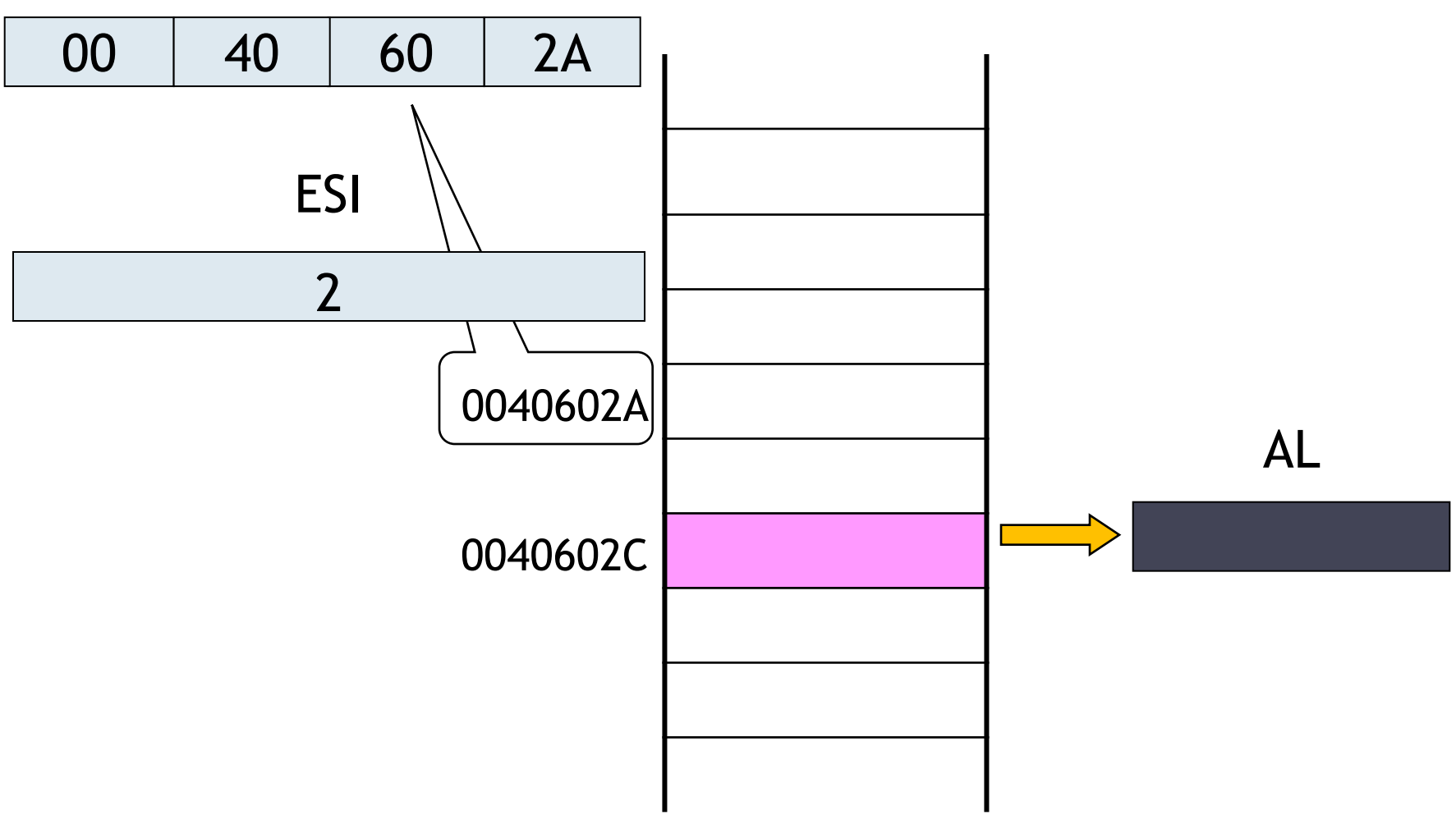
```
mov al,[ebx+2]
```

indirect address with a base and displacement



```
mov al,[ebx+esi]
```

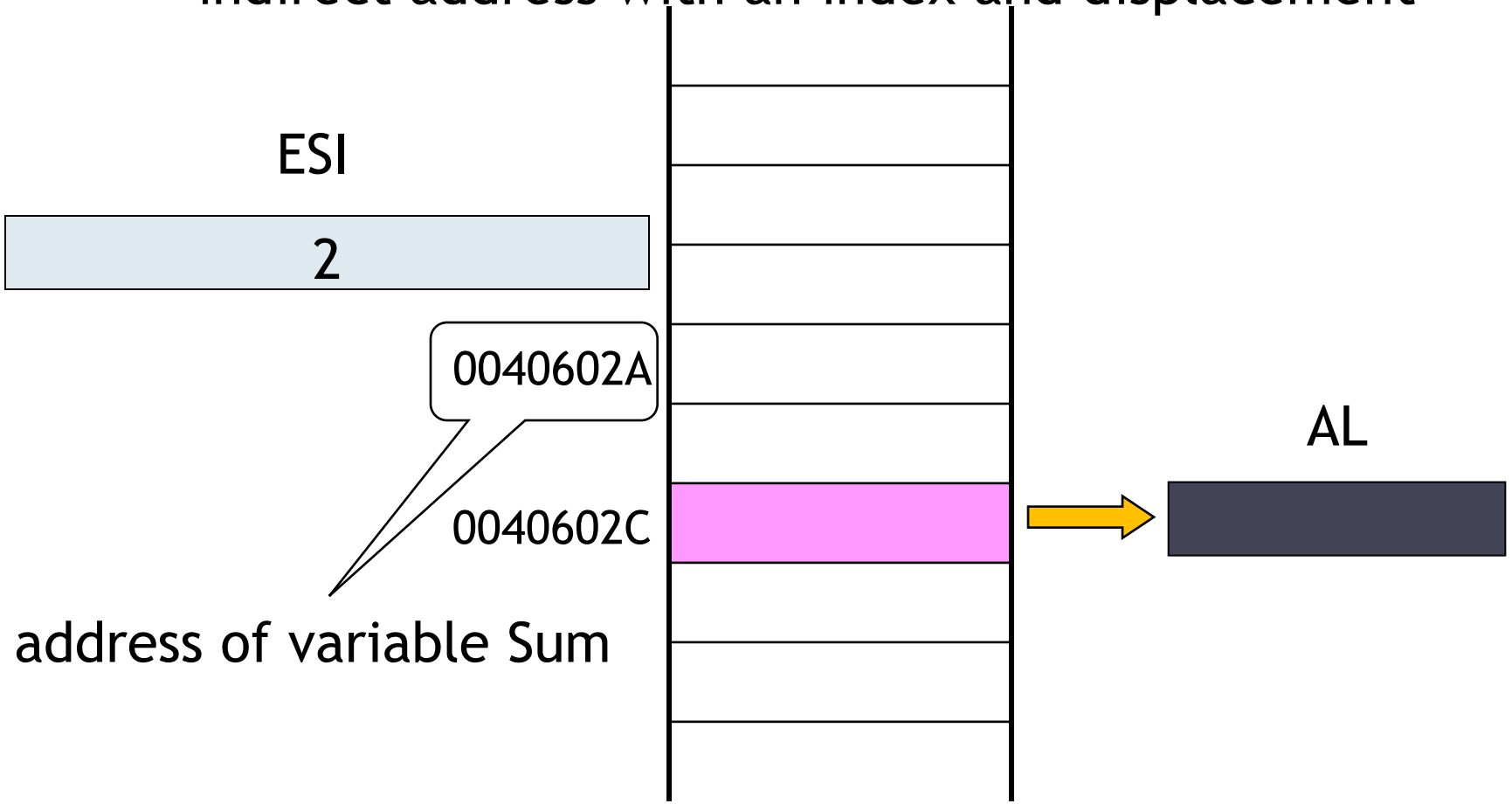
indirect address with a base and index



The name of a variable is taken as a displacement:

```
mov al,[Sum + esi]   or
mov al,Sum[esi]
```

indirect address with an index and displacement



Index register can be multiplied by 2, 4 or 8 to make the access to word, dword, and qword arrays faster.

- Example:

Table is an array of type word. Store the first word to AX:

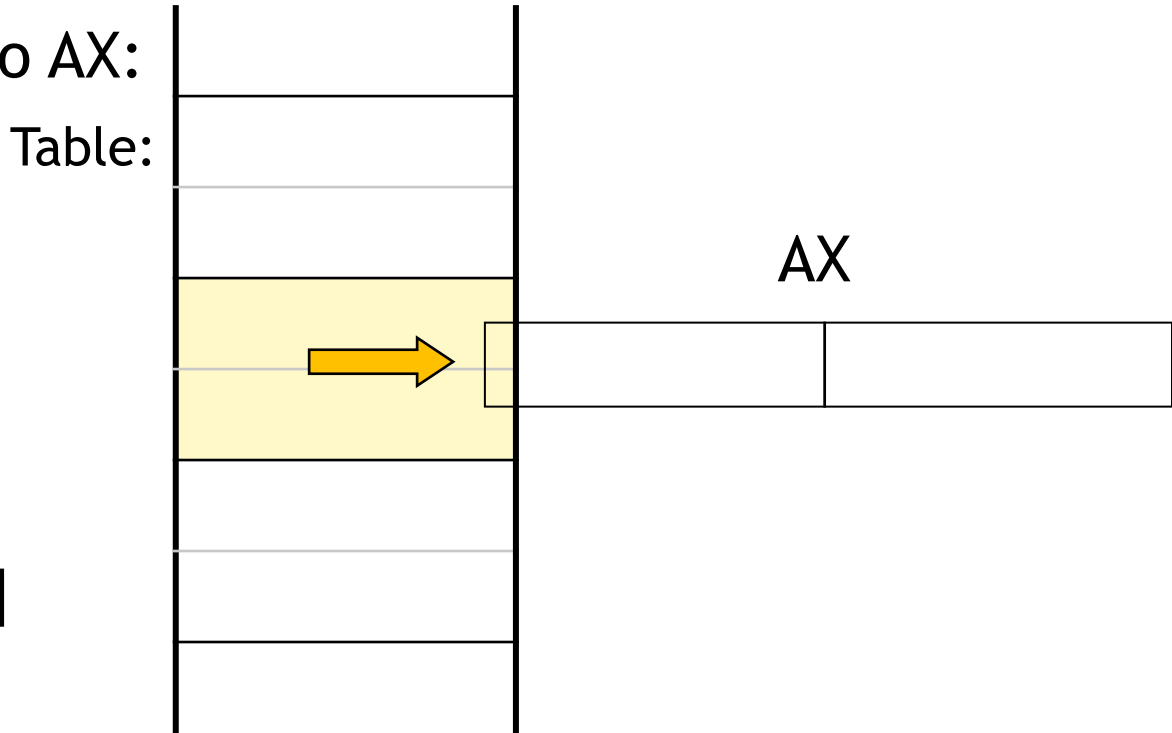
```
mov esi,0
mov ax,[Table+esi]
```

Store the second word to AX:

```
inc esi
inc esi
mov ax,[Table+esi]
```

or

```
inc esi
mov ax,[Table+2*esi]
```



Indirect addressing is useful when operating with data arrays:

- base register – points to the array (contains the address of the first element)
- index register – contains the index of an element

Coding:

```
mov al,[Str+edi] → 8A 87 00 60 40 00
mov al,[edx+edi] → 8A 04 17
```

better!

```
mov [Str],'*' → C6 05 00 60 40 00 2A
mov byte ptr [edx],'*' → C6 02 2A
```

# The basic elements of an assembly language program

Statements – one per line.

The format of a line:

[identifier] instruction/directive [operands] [; comment]

# Identifier

- name of a variable = symbolic address of a data object (offset in the data segment)
- label, name of a procedure = symbolic address of an instruction (offset in the code segment)
- symbolic constant
- name of a segment

# Make your labels meaningful!

- Find out whether register AL contains a code for a lower case letter. If yes, convert it to the corresponding upper case letter.

```
cmp al,'a'
jb N1
cmp al,'z'
ja N1
sub al,20h
N1:   …
```

Better:

```
cmp al,'a'
jb NotALowerCaseLetter
cmp al,'z'
ja NotALowerCaseLetter
sub al,20h
NotALowerCaseLetter:
        …
```

# Comment your code!

Comment through multiple lines:

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

or:

```
COMMENT /*
      This line is a comment.
      This line is also a comment.
*/
```

# Directives

They are not compiled to a machine code, but they can:

- define constants, variables, labels, segments, procedures and macros
- enable compilation of instructions from the enhanced instruction set, e.g.:

  ```
  .686P; enables assembly of all instructions for
    the Pentium Pro processor
  ```

  (in include file SmallWin.inc)

  ```
  .MMX; enables assembly of MMX instructions
  ```

- control the contents and format of the program listing (report from the compilation)
- control conditional compilation (e.g. directives IF, ENDIF, ELSE)

# Operands

Possible operands:

- registers
- addresses (see addressing modes)
- numbers, symbolic constants

An instruction may have:

- none operand
- 1 operand
- 2 operands
  - the right one (source) : register, memory, number
  - the left one: register, memory

  If the operands are registers or memory locations, they must be of the same types:

  ~~mov ax,bl~~

  Two memory operands are not allowed in the instruction!
- 3 operands
  - imul register, register/memory, number

# Variables

- symbolic addresses of data items (offsets in the data segment)
- defined by directives DB, DW, DD, DF, DQ, DT.

Syntax:

[name of the variable] Dx expression [,expression] ...

Directive Dx:

- determines the variable type (according to the letter x)
- allocates the space in memory (one or more data items)
- initializes the contents of the memory locations (does not initialize, if the expression is ?)

| Directive | Size of the allocated memory in bytes | Variable type | Variable may contain |
|---|---|---|---|
| DB | 1 | byte | Signed integer in the range ⟨-128; 127⟩<br>Unsigned integer in the range ⟨0; 255⟩<br>Character |
| DW | 2 | word | Signed integer in the range ⟨-32 768; 32 767⟩<br>Unsigned integer in the range ⟨0; 65 535⟩<br>16-bit offset |
| DD | 4 | dword | Signed integer<br>Unsigned integer<br>Single precision floating point number in the range about $\pm 10^{38}$<br>Far pointer in 16-bit mode, i.e. address in the segment:offset form<br>32-bit offset |

| Directive | Size of the allocated memory in bytes | Variable type | Variable may contain |
|---|---|---|---|
| DF | 6 | fword | Signed integer<br>Unsigned integer<br>Far pointer in 32-bit mode, i.e. address in the segment:offset form |
| DQ | 8 | qword | Signed integer<br>Unsigned integer<br>Double precision floating point number in the range about $\pm 10^{308}$ |
| DT | 10 | tbyte | Signed integer<br>Unsigned integer<br>Packed BCD number<br>Extended precision floating point number in the range about $\pm 10^{4932}$ |

```
.data
;contents of memory locations from the offset 4069D0h
One   DB –1,12+1     ; FF 0D
      DB ’abcd’      ; 61 62 63 64
      DB 3 dup(?)    ; ?? ?? ??
      DW -32768      ; 00 80
      DD 0.3         ; 9A 99 99 3E
      DD 0ABCDEF23h; 23 EF CD AB
      DD One         ; D0 69 40 00
      DT 10          ; 0A 00 00 00 00 00 00 00 00 00
```

Multi-byte data are stored in the reverse order of bytes.

4069D0h is offset of variable One

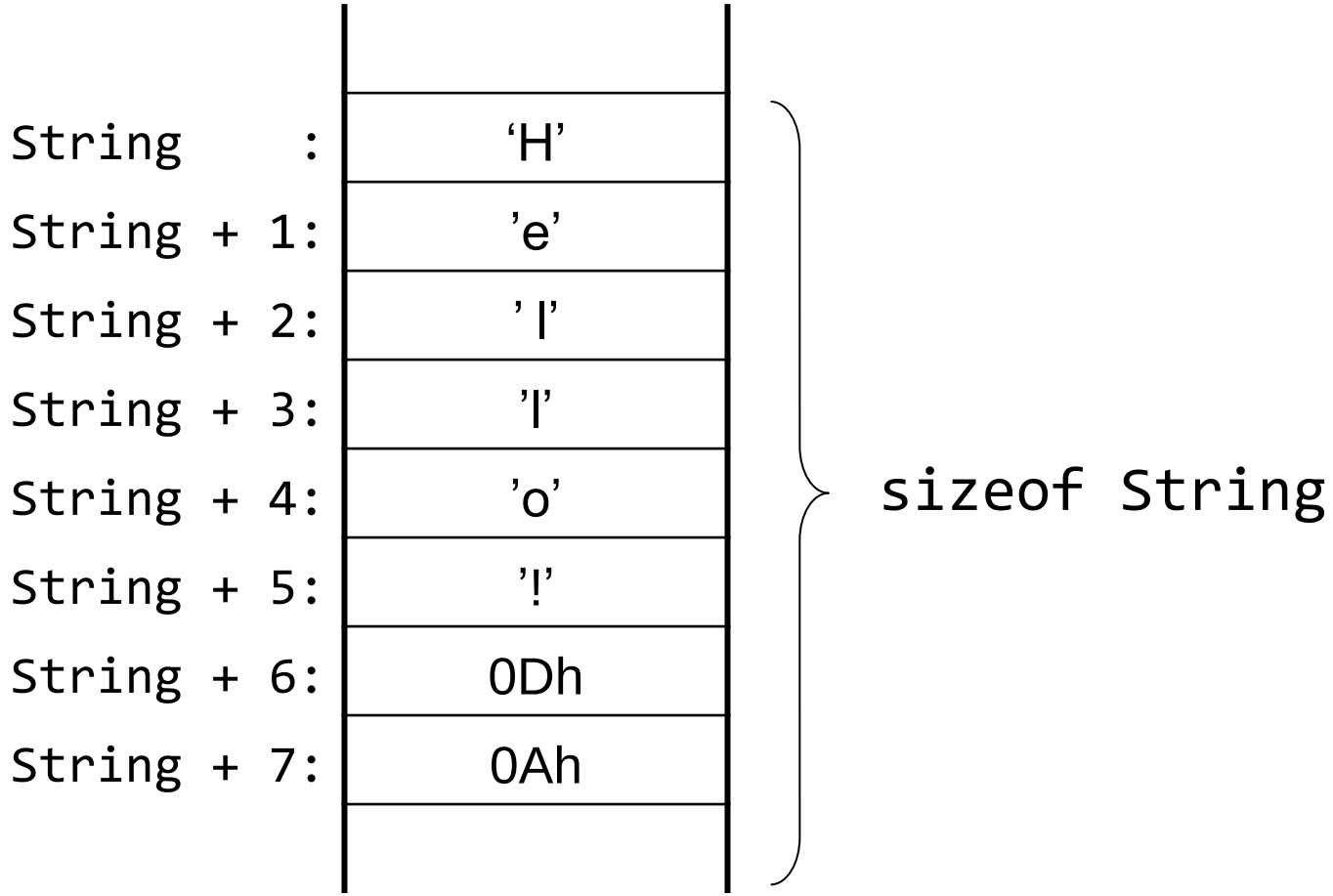
| Operator | Purpose | |
|---|---|---|
| offset | Gets offset of the variable. | |
| type | Returns the value | according to the type of the variable |
| | 1 | byte |
| | 2 | word |
| | 4 | dword |
| | 6 | fword |
| | 8 | qword |
| | 10 | tbyte |
| length | Gets the number of data items allocated to the variable by the first expression. | |
| size | Gets the number of bytes allocated to the variable by the first expression, i.e. the value length * type. | |
| lengthof | Gets the number of data items allocated to the variable. | |
| sizeof | Gets the number of bytes allocated to the variable, i.e. lengthof * type. | |
| ptr | Overwrites the type of the variable. | |

```
Value DW 1234h
Vector DB 5,6,7
Table DW 5 dup(?),1000

mov al,type Value; al = 2
mov bl,type Vector; bl = 1
mov cl,type Table; cl = 2
mov al,length Value; al = 1
mov bl,length Vector; bl = 1
mov cl,length Table; cl = 5
mov cl,size Table; cl = 10
mov cl,lengthof Table; cl = 6
mov cl,sizeof Table; cl = 12
```

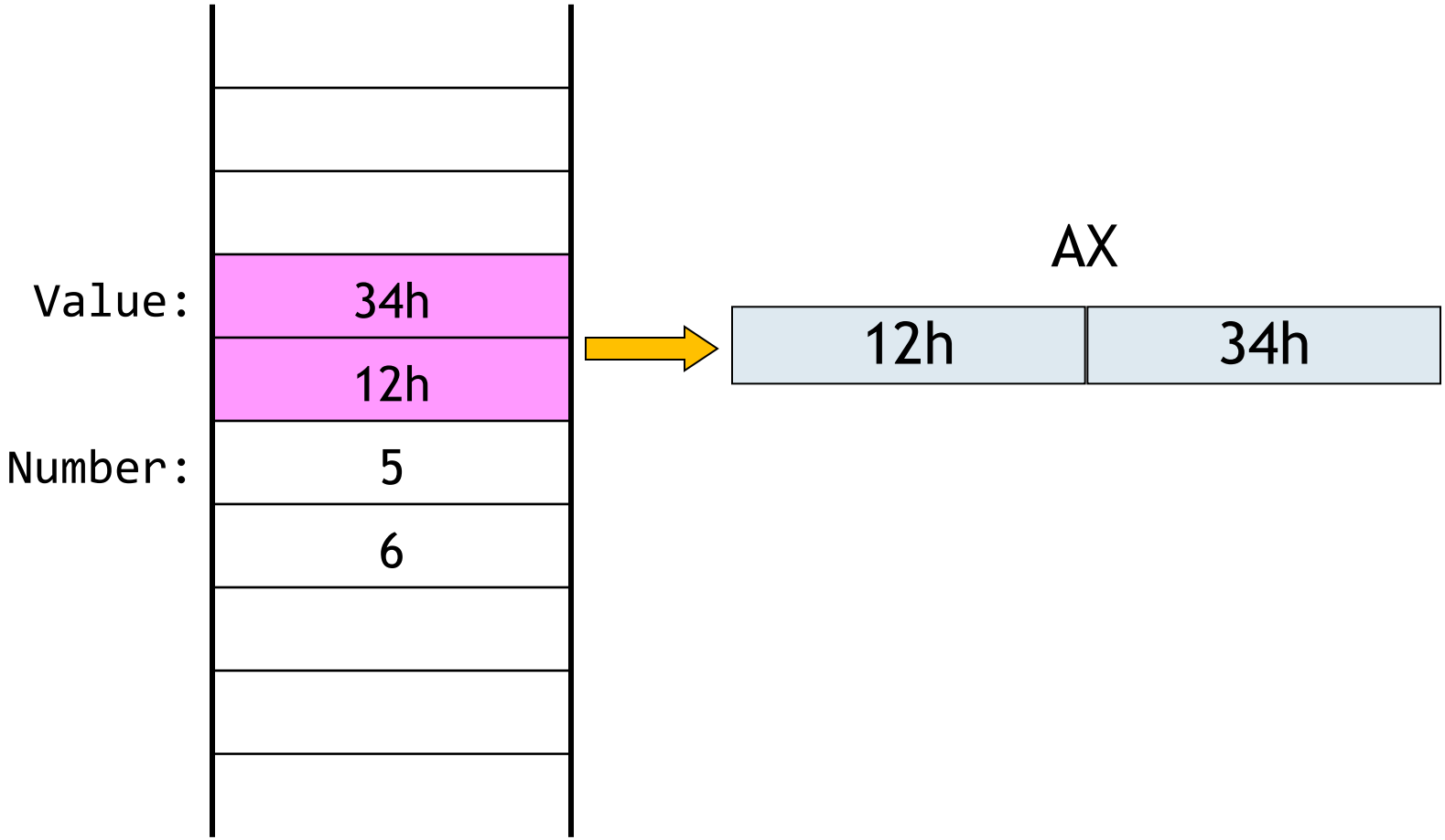
length returns value > 1 only if the variable has been defined using operator dup

- Display the string „Hello!“ stored in variable String (without termination character).



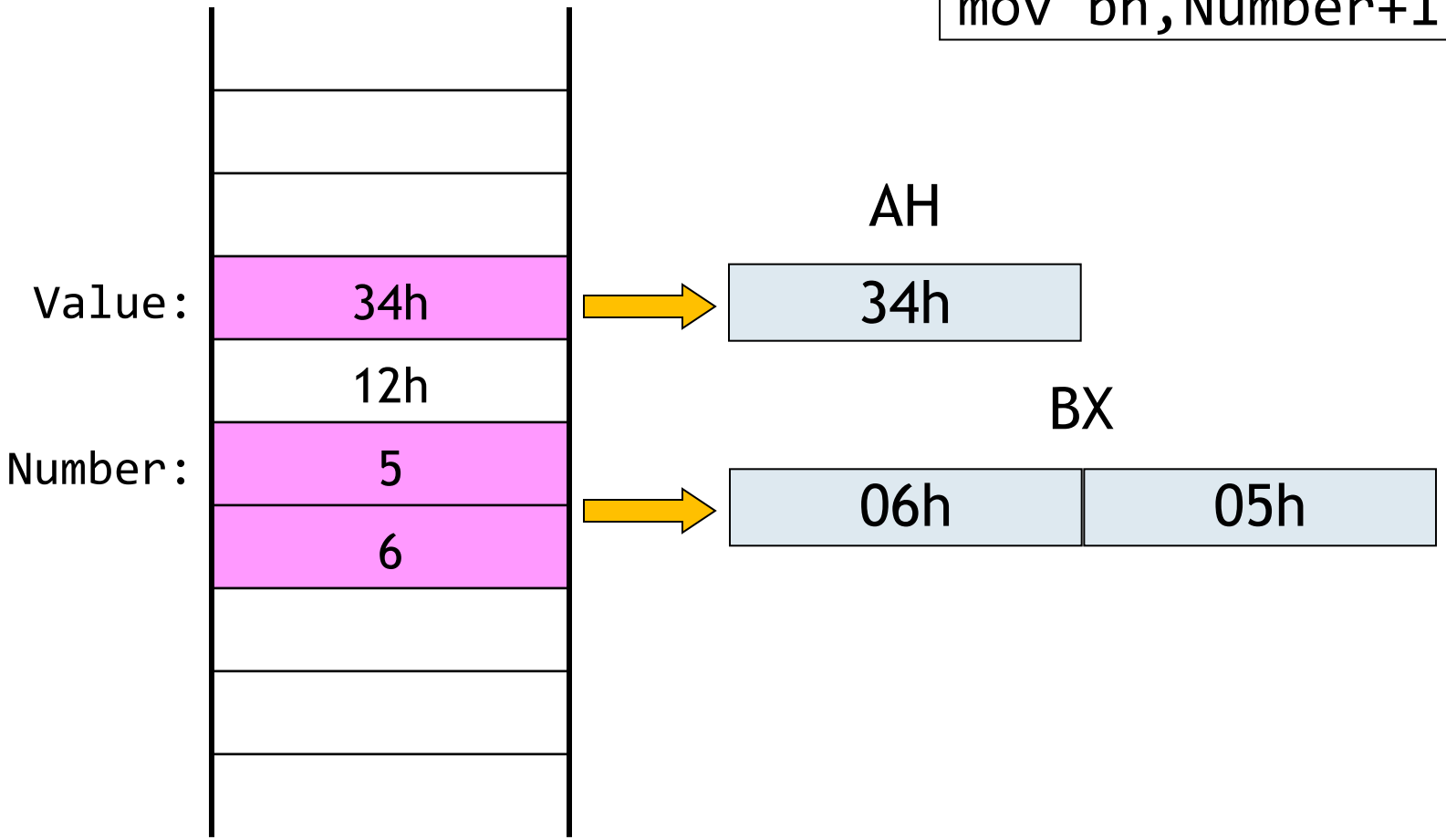
```
TITLE MASM String(main.asm)
INCLUDE Irvine32.inc
.data
String DB “Hello!",0Dh,0Ah
.code
main PROC
  mov edx,offset String ; point edx to the memory location for
                        ; the first character
  mov edi,0; the 1st character has index 0
  mov ecx,lengthof String
Display:
  mov al,[edx+edi]; copy the character at offset edx+edi to al
  call WriteChar; display the character whose ASCII code is in
                ;al
  inc edi; increment index by 1 (to the next character)
  loop Display; ecx = ecx – 1, if ecx > 0, jump to Display
exit
main ENDP
END main
```

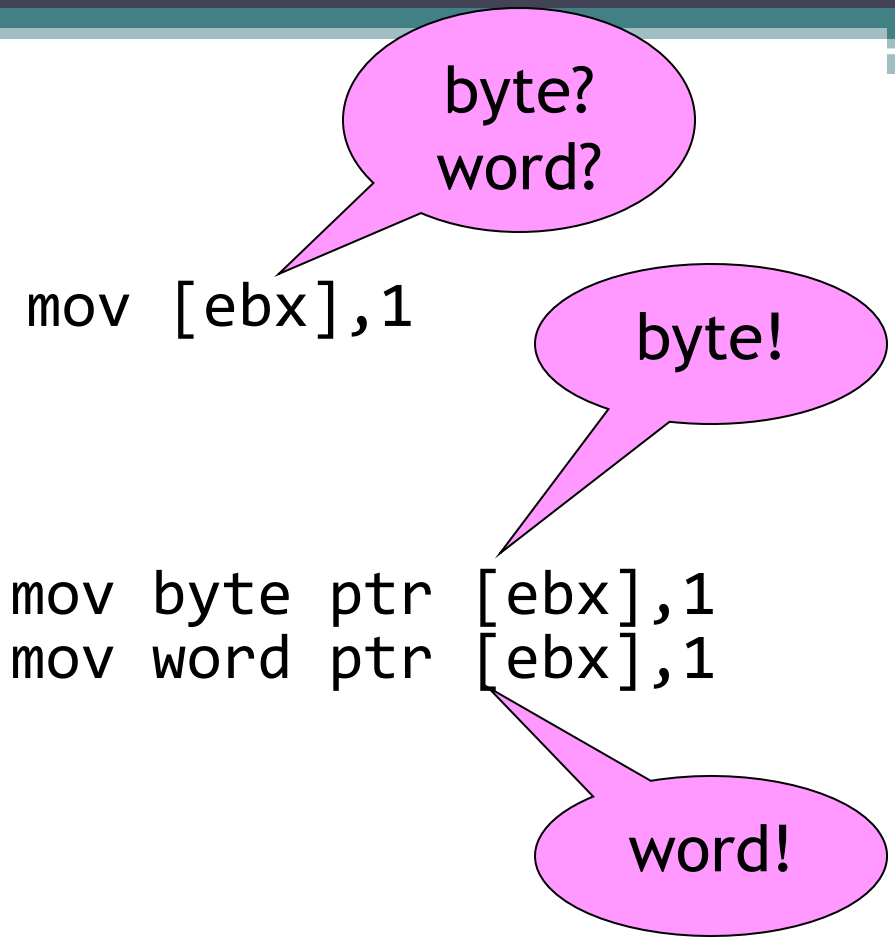
```
Value DW 1234h
Number DB 5,6
mov ax,Value; al = 34h, ah = 12h
```



```
mov ah,byte ptr Value; ah = 34h
mov bx,word ptr Number; bx = 605h
```

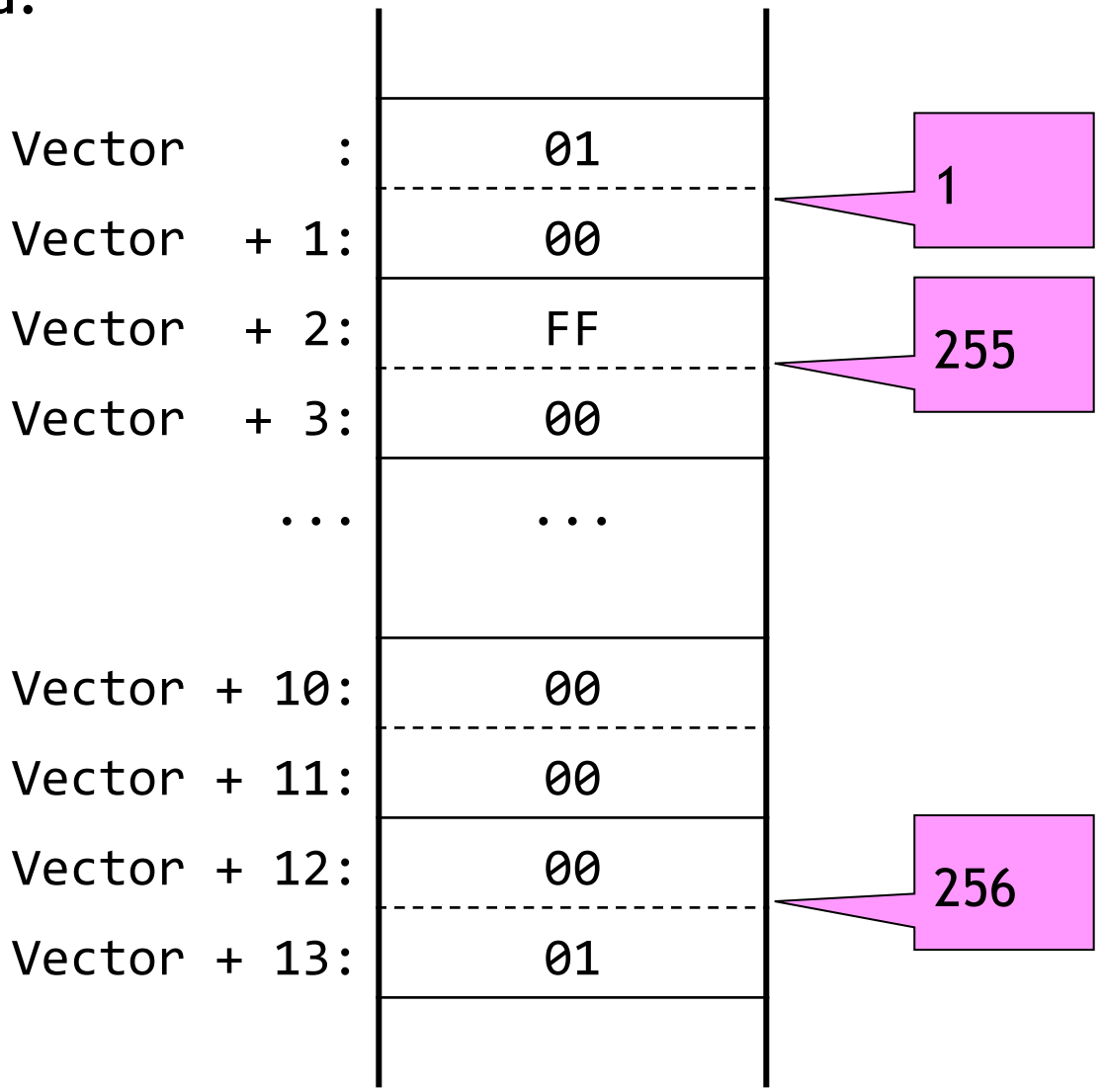
the same as
```
mov bl,Number
mov bh,Number+1
```






Similar problem: `inc [ebx]`

- Find out, how many zero components are in variable Vector of type word.



```
TITLE MASM Index_v1 (main.asm)
INCLUDE Irvine32.inc
.data
Vector DW 1,255,0,0,0,0,256
.code
main PROC
  mov bl,0; counter
  mov ecx,lengthof Vector ; save the number of components to ecx
  mov edx,offset Vector; point edx to Vector
  mov edi,0; the 1st component has index 0
Compare: cmp word ptr [edx+edi],0
  jne Continue
  inc bl
Continue:
  inc edi; increment index by 2
  inc edi
  loop Compare
Finish:
  exit
main ENDP
END main
```

```
TITLE MASM Index_v1 (main.asm)
INCLUDE Irvine32.inc
.data
Vector DW 1,255,0,0,0,0,256
.code
main PROC
  mov bl,0; counter
  mov ecx,lengthof Vector ; save the number of components to ecx
  mov edx,offset Vector; point edx to Vector
  mov edi,0; the 1st element has index 0
Compare: cmp word ptr [edx+2*edi],0
  jne Continue
  inc bl
Continue:
  inc edi; increment index

  loop Compare
Finish:
  exit
main ENDP
END main
```

better! (one less inc)

# Symbolic constants

- make the orientation in the program and its modification easier
- defined by directive EQU