본 강의 동영상 및 자료는 대한민국 저작권법을 준수합니다. 본 강의 동영상 및 자료는 상명대학교 재학생들의 수업 목적으로 제작·배포되는 것으로, 수업목적으로 내려받은 강의 동영상 및 자료는 수업목적 이외에 다른 용도로 사용할 수 없으며, 다른 장소 및 타인에게 복제, 전송하여 공유할 수 없습니다. 이를 위반해서 발생하는 모든 법적 책임은 행위 주체인 본인에게 있습니다.

8. Addressing Modes

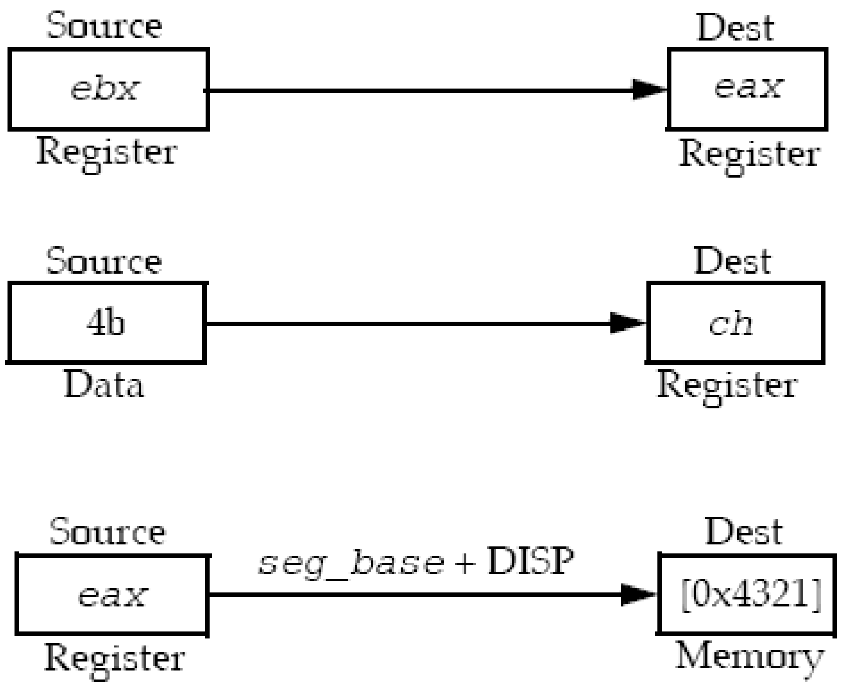
Data Addressing Modes

- Let's cover the data addressing modes using the ***mov*** instruction.
  - Data movement instructions move data (bytes, words and doublewords) between registers and between register / memory.
  - Only the ***movs*** (strings) instruction can have both operands in memory.
  - Most data transfer instructions do not change the **EFLAGS** register.
- Storage protocols
  - When an n-byte transfer is indicated by an address a, the memory bytes referred to are those at the address a, a+1, …, a+n-1
  - When an n-byte number is stored in memory, its bytes are stored in order of significance → little endian

Data Addressing Modes

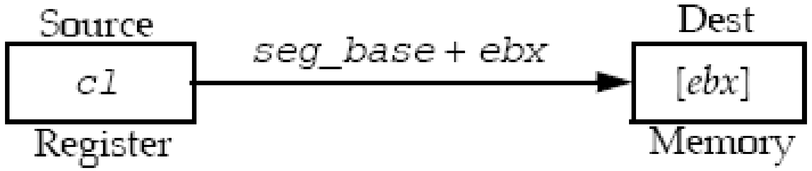
- Register
  - mov eax, ebx
- Immediate
  - mov ch, 0x4b
- Direct (eax), Displacement (other regs)
  - mov [0x1234], eax

Indirection operator!!

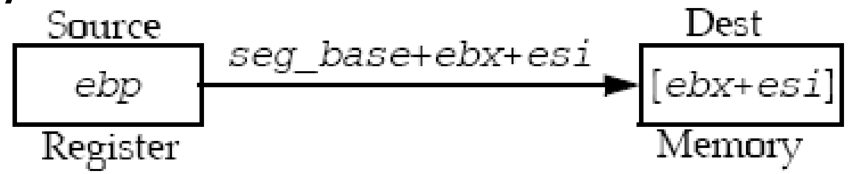


Data Addressing Modes

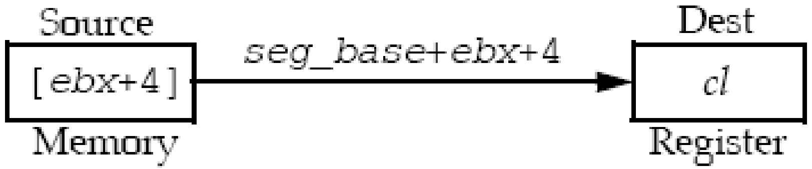
- Register Indirect
  - mov [ebx], cl
  - Any of eax, ebx, ecx, edx, ebp, edi or esi may be used.



- Base-plus-index
  - mov [ebx+esi], ebp
  - Any combination of eax, ebx, ecx, edx, ebp, edi or esi.



- Register relative
  - mov cl, [ebx+4]
  - A second variation includes: mov eax, [ebx+ARR]



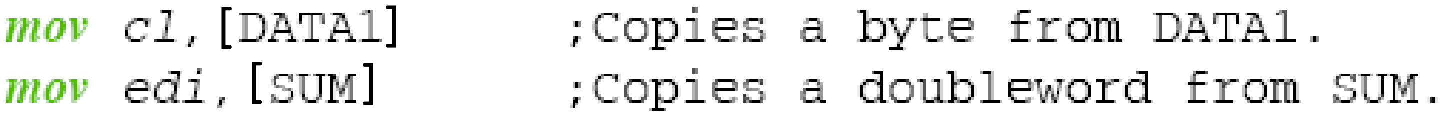
X86 Indirect Addressing Modes

BASE + (INDEX * SCALE) + DISPLACEMENT

$$\left\{\begin{matrix}\text{none}\\ \text{EAX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBX}\\ \text{ESP}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{matrix}\right\} + \left\{\begin{matrix}\text{none}\\ \text{EAX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBX}\\ -\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{matrix}\right\} * \left\{\begin{matrix}1\\ 2\\ 4\\ 8\end{matrix}\right\} + \left\{\begin{matrix}\text{None}\\ \text{8-bit}\\ \text{32-bit}\end{matrix}\right\}$$

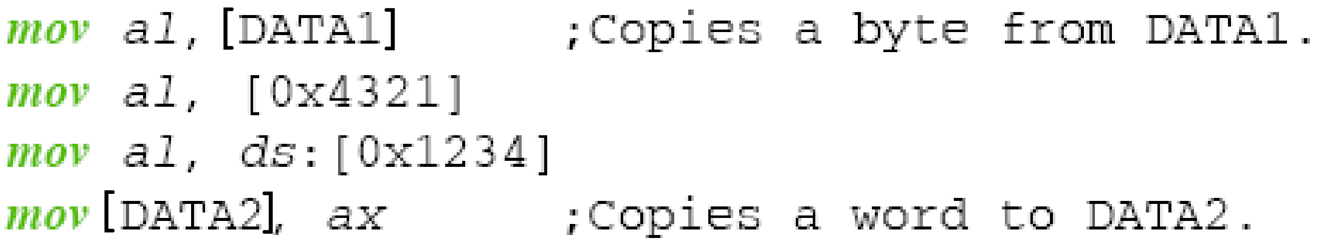
Displacement Addressing

- Displacement addressing
  - Displacement instructions are encoded with up to 7 bytes (32 bit register and a 32 bit displacement).
  - To access a statically allocated scalar operand



```
mov cl, [DATA1]        ;Copies a byte from DATA1.
mov edi, [SUM]         ;Copies a doubleword from SUM.
```

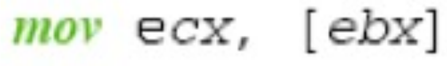
- Direct addressing
  - Transfers between memory and *al*, *ax* and *eax*.
  - Usually encoded in 3 bytes, sometime 4:



```
mov al, [DATA1]        ;Copies a byte from DATA1.
mov al, [0x4321]
mov al, ds:[0x1234]
mov [DATA2], ax        ;Copies a word to DATA2.
```

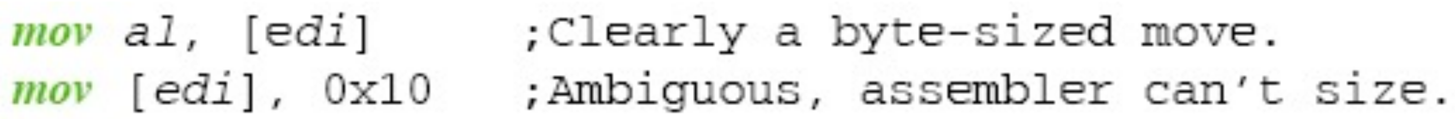
Register Indirect Addressing

- Offset stored in a register is added to the segment register. Used for dynamic storage of variables and data structures



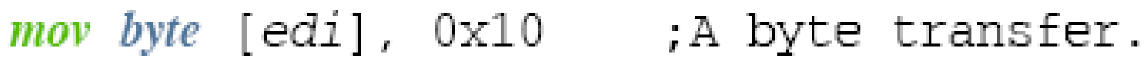
```
mov ecx, [ebx]
```

- The memory to memory ***mov*** is allowed with string instructions.
  - Any register EXCEPT **esp** for the 80386 and up.
  - For **eax**, **ebx**, **ecx**, **edx**, **edi** and **esi**: The data segment is the default.
  - For **ebp**: The stack segment is the default.
  - Some versions of register indirect require special assembler directives ***byte, word,*** or ***dword***



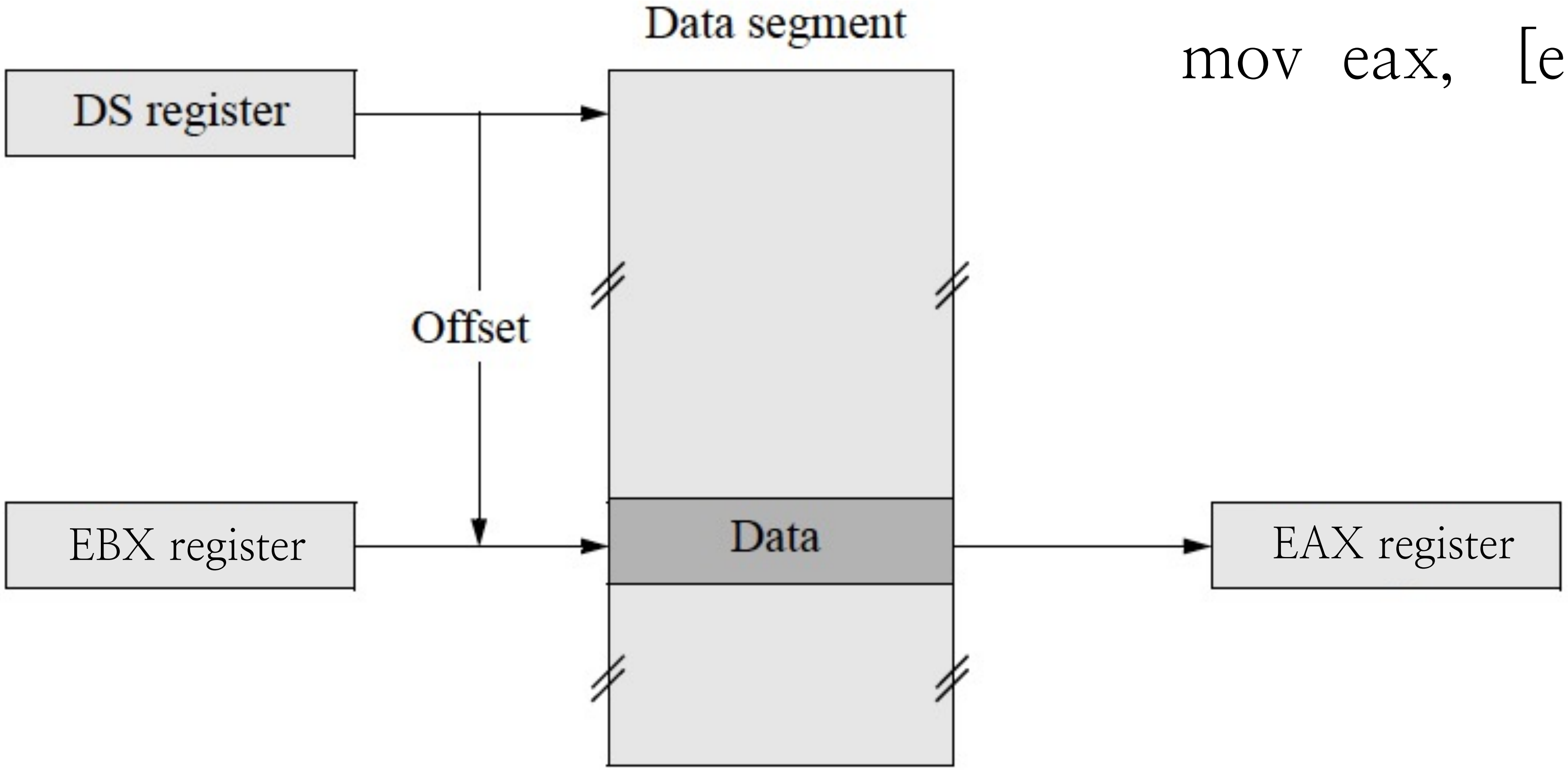
```
mov al, [edi]         ;Clearly a byte-sized move.
mov [edi], 0x10       ;Ambiguous, assembler can't size.
```

  - Does [*edi*] address a byte, a word or a double-word? Use



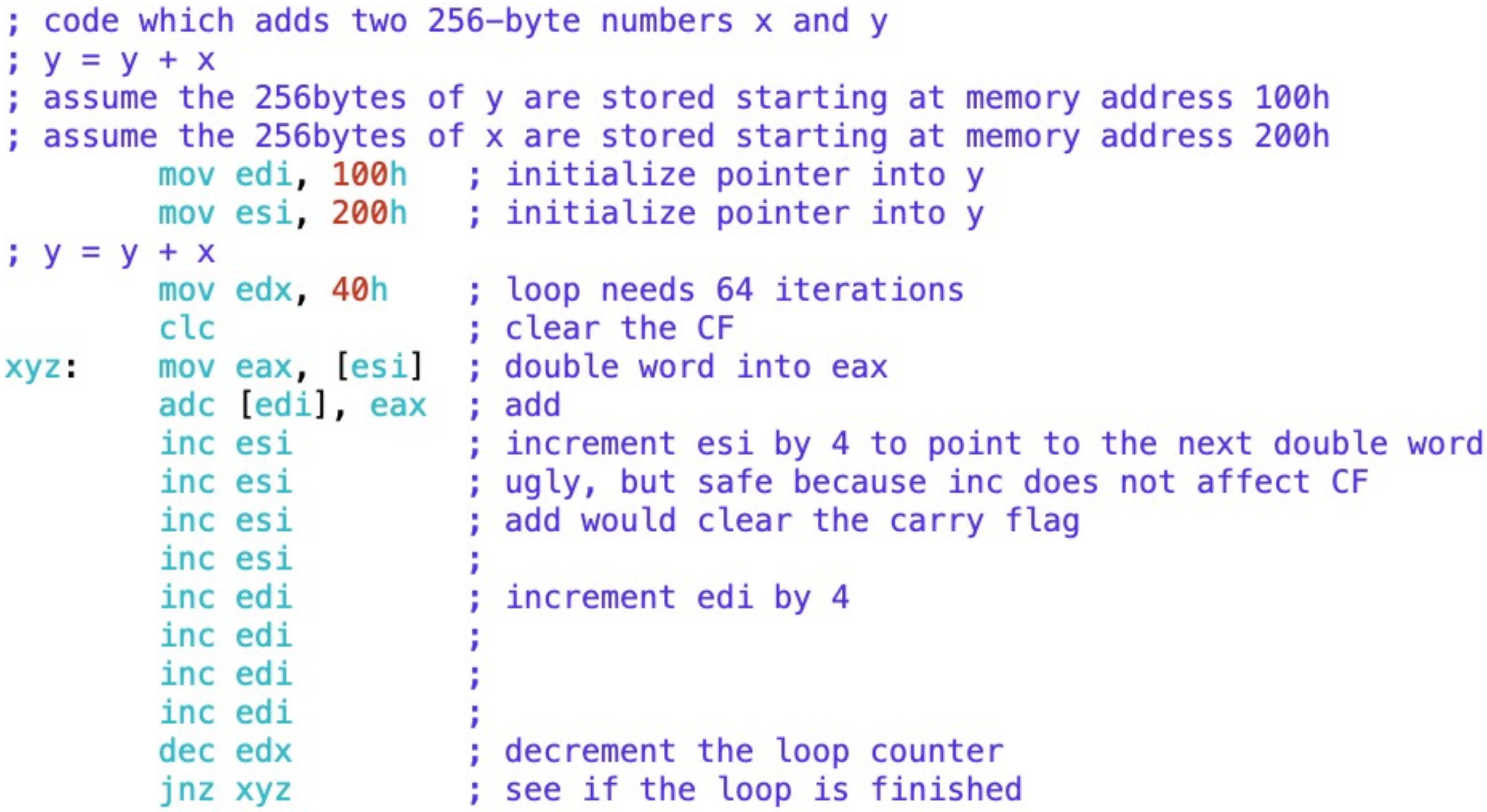
```
mov byte [edi], 0x10      ;A byte transfer.
```

Register Indirect Addressing



mov eax, [ebx]

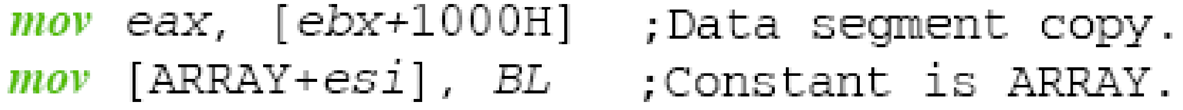
Register Indirect Addressing



```
; code which adds two 256-byte numbers x and y
; y = y + x
; assume the 256bytes of y are stored starting at memory address 100h
; assume the 256bytes of x are stored starting at memory address 200h
        mov edi, 100h   ; initialize pointer into y
        mov esi, 200h   ; initialize pointer into y
; y = y + x
        mov edx, 40h    ; loop needs 64 iterations
        clc             ; clear the CF
xyz:    mov eax, [esi]  ; double word into eax
        adc [edi], eax  ; add
        inc esi         ; increment esi by 4 to point to the next double word
        inc esi         ; ugly, but safe because inc does not affect CF
        inc esi         ; add would clear the carry flag
        inc esi         ;
        inc edi         ; increment edi by 4
        inc edi         ;
        inc edi         ;
        inc edi         ;
        dec edx         ; decrement the loop counter
        jnz xyz         ; see if the loop is finished
```

Register Relative Addressing

- Effective address computed as: seg_base + base + constant.
- Same default segment rules apply with respect to **ebp**, **ebx**, **edi** and **esi**.
- Displacement constant is any *32-bit* signed value.

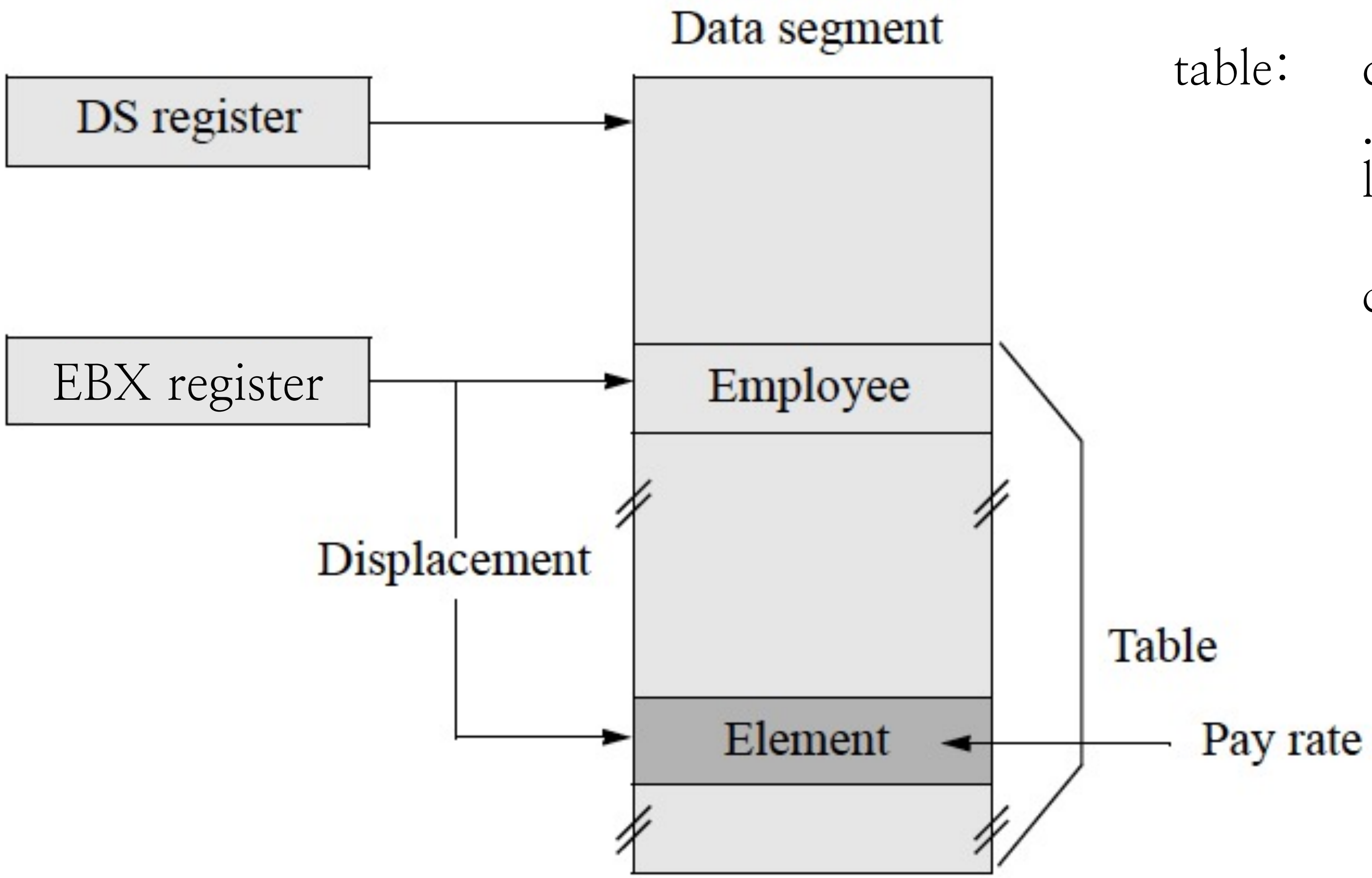


```
mov eax, [ebx+1000H]   ;Data segment copy.
mov [ARRAY+esi], BL    ;Constant is ARRAY.
```

Register Relative Addressing

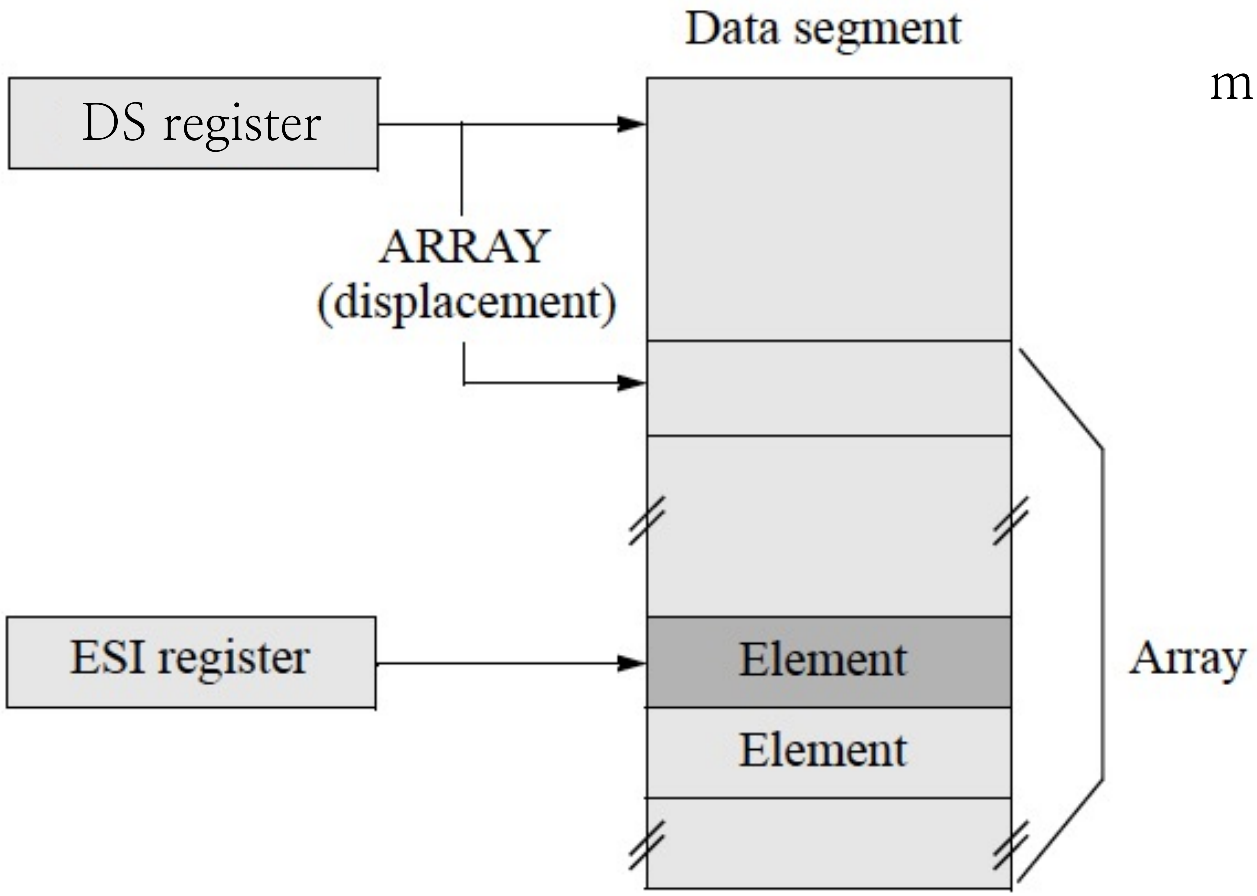
- Base+displacement
  - An index into an array when the element size is not 2, 4, or 8 bytes; The displacement encodes the static offset to the beginning of the array, while the base register holds the results of a calculation to determine the offset to a specific element within the array
  - To access a field of a record; the base register holds the address of the beginning of the record, while the displacement is an static offset to the field
  - A important special case is access to parameters in a procedure activation record (the base register in this case is EBP)
- (Index*scale)+displacement
  - Index into a static array when the element size is 2, 4, or 8 bytes

Base + Displacement



```
table:   db       15, 7, 6, 10, 4
         ...
         lea   ebx, table    ; loads the effective
                             ;addr of TABLE into BX
         cmp  eax, [ebx+4]
```

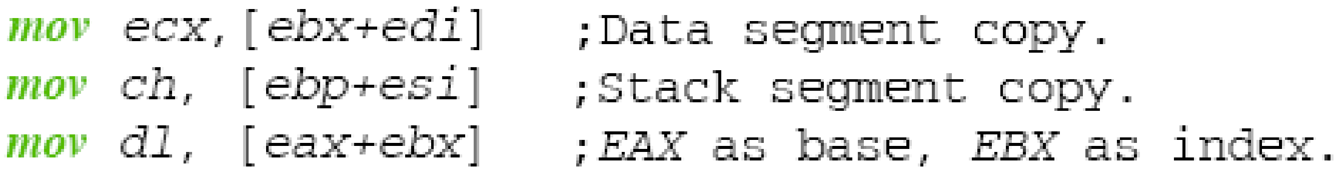
Index * Scale + Displacement



```
mov  eax,   [esi*4+ARRAY]
```

Base-Plus-Index Addressing

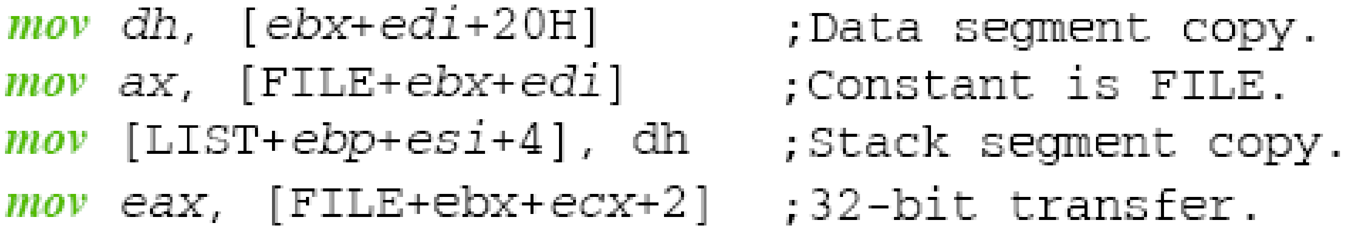
- Effective address computed as: seg_base + base + index.
- Base registers: Holds starting location of an array.
  - ebp, esp (stack) / ebx, … (data)
- Index registers: Holds offset location.
  - edi, esi, Any 32-bit register except esp.
- frequently used to access the elements of a dynamic array. A dynamic array is an array whose base address can change during program execution.



```
mov ecx,[ebx+edi]    ;Data segment copy.
mov ch, [ebp+esi]    ;Stack segment copy.
mov dl, [eax+ebx]    ;EAX as base, EBX as index.
```

Base-Plus-Index-plus-displacement Addressing

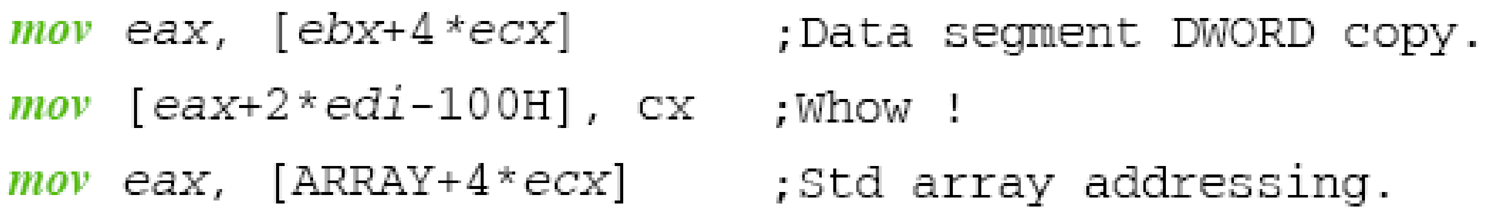
- Effective address computed as: seg_base + base + index + constant.
- Designed to be used as a mechanism to address a two-dimensional array (the displacement holds the address of the beginning of the array)
- One of several instances of an array of records (displacement is an offset to a field within the record)



```
mov dh, [ebx+edi+20H]        ;Data segment copy.
mov ax, [FILE+ebx+edi]       ;Constant is FILE.
mov [LIST+ebp+esi+4], dh     ;Stack segment copy.
mov eax, [FILE+ebx+ecx+2]    ;32-bit transfer.
```

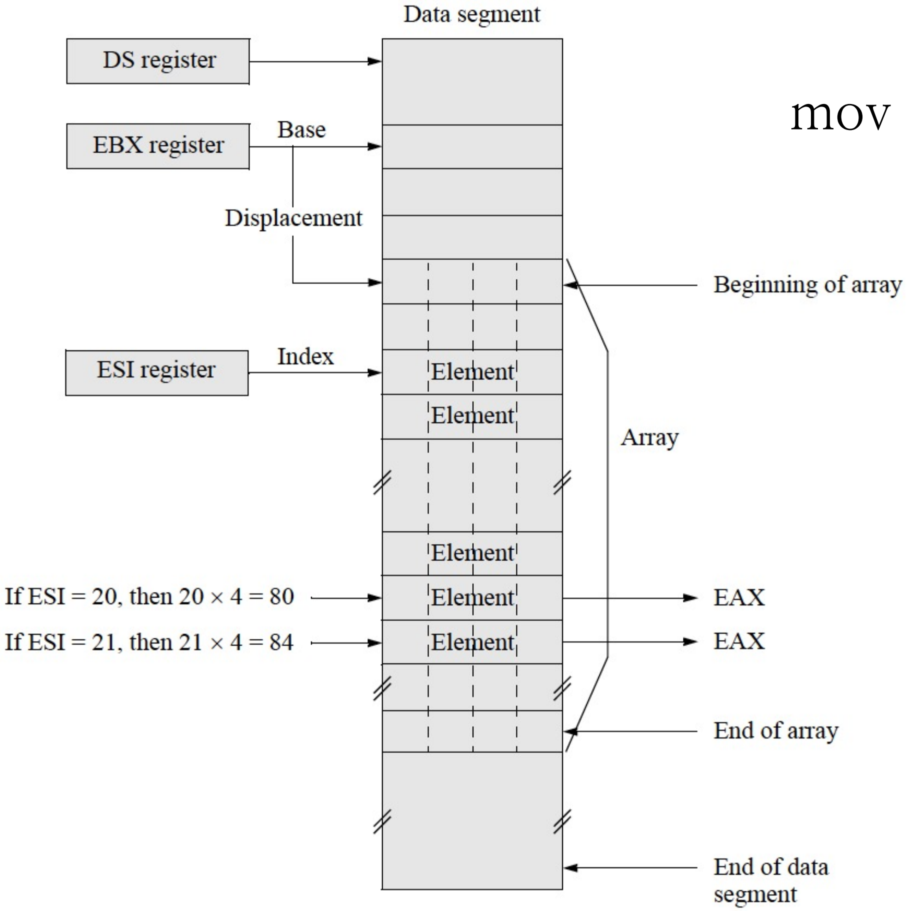
Scaled-Index Addressing

- Effective address computed as: seg_base + base + constant*index
- Indexing two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size



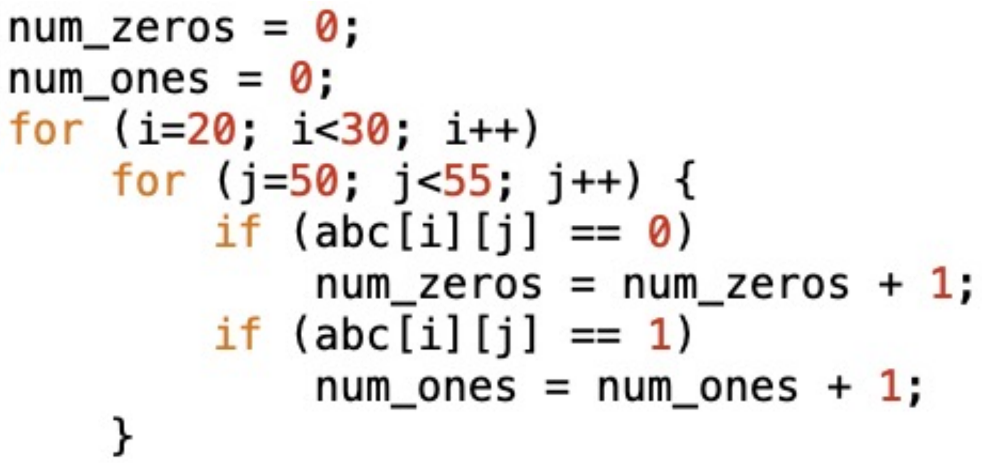
```
mov eax, [ebx+4*ecx]          ;Data segment DWORD copy.
mov [eax+2*edi-100H], cx      ;Whow !
mov eax, [ARRAY+4*ecx]        ;Std array addressing.
```

Scaled-Index Addressing



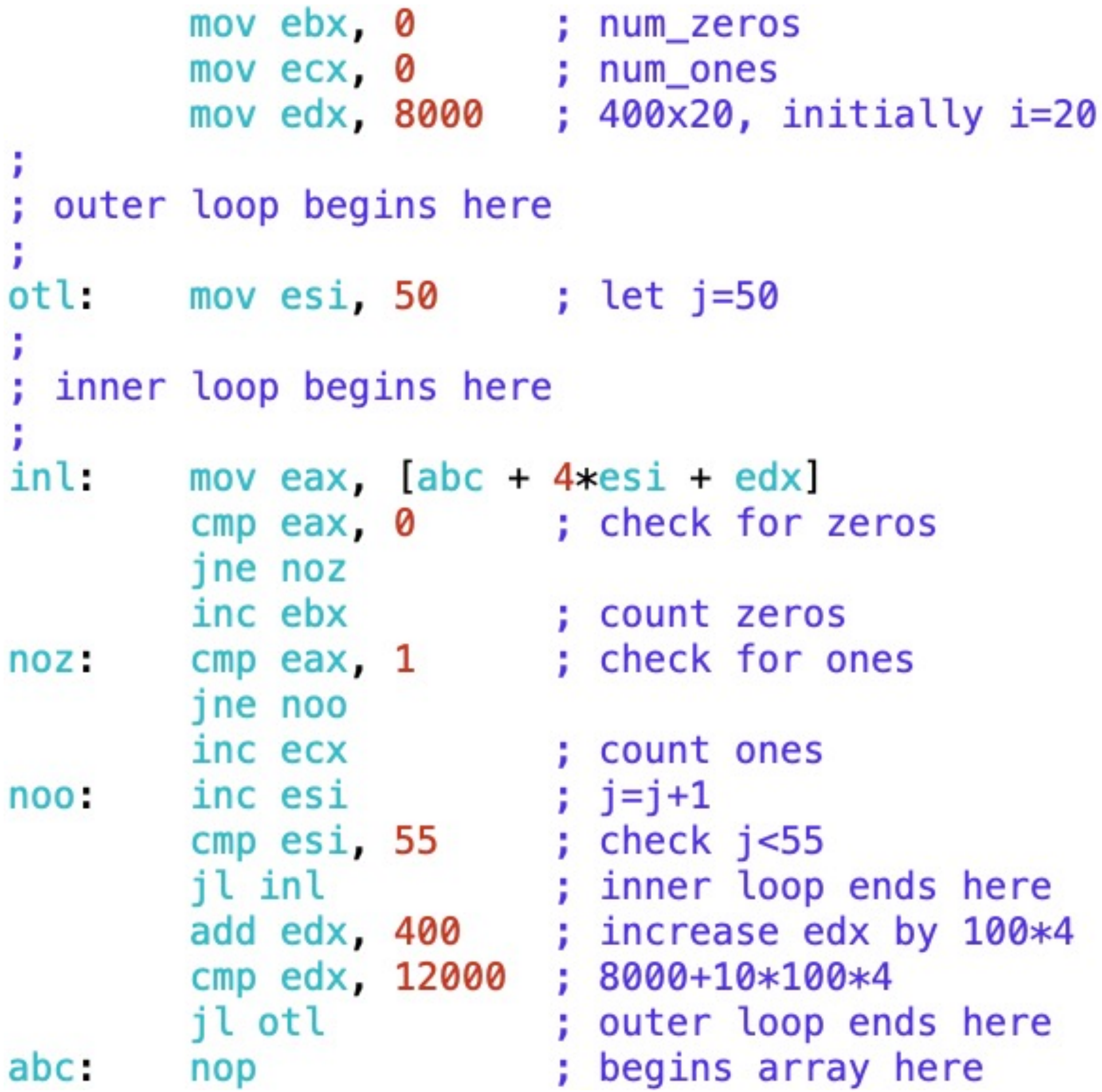
mov eax, [ebx+esi*4+DISPLACEMENT]

Arrays



```
num_zeros = 0;
num_ones = 0;
for (i=20; i<30; i++)
    for (j=50; j<55; j++) {
        if (abc[i][j] == 0)
            num_zeros = num_zeros + 1;
        if (abc[i][j] == 1)
            num_ones = num_ones + 1;
    }
```

$$\begin{array}{cc} & \begin{array}{cccccc} \text{ABC} & & 50 & & 54 & 99 \end{array} \\ \begin{array}{c} \\ \\ 20 \\ 29 \end{array} & \begin{bmatrix} x & & & & & \\ \vdots & & & & & \\ x & \cdots & x & \cdots & x & x \\ x & \cdots & x & \cdots & x & x \end{bmatrix} \end{array}$$



```
        mov ebx, 0      ; num_zeros
        mov ecx, 0      ; num_ones
        mov edx, 8000   ; 400x20, initially i=20
;
; outer loop begins here
;
otl:    mov esi, 50     ; let j=50
;
; inner loop begins here
;
inl:    mov eax, [abc + 4*esi + edx]
        cmp eax, 0      ; check for zeros
        jne noz
        inc ebx         ; count zeros
noz:    cmp eax, 1      ; check for ones
        jne noo
        inc ecx         ; count ones
noo:    inc esi         ; j=j+1
        cmp esi, 55     ; check j<55
        jl inl          ; inner loop ends here
        add edx, 400    ; increase edx by 100*4
        cmp edx, 12000  ; 8000+10*100*4
        jl otl          ; outer loop ends here
abc:    nop             ; begins array here
```

IA-32 SW Developer’s man	Lecture note	Application
displacement	Direct Displacement	To access a statically allocated scalar operand
base	Register indirect	Used for dynamic storage of variables and data structures
Base+displacement	Register relative	- An index into an array when the element size is not 2, 4, or 8 bytes (the displacement encodes the static offset to the beginning of the array; The base register holds the results of a calculation to determine the offset to a specific element within the array) - To access a field of a record (the base register holds the address of the beginning of the record, while the displacement is an static offset to the field) - A special case is access to parameters in a procedure activation record (the base register in this case is EBP)
(Index*scale)+displacement		- Index into a static array when the element size is 2, 4, or 8 bytes
Base+Index+Displacement	Base relative-plus-index	- A two-dimensional array (the displacement holds the address of the beginning of the array) - One of several instances of an array of records (displacement is an offset to a field within the record
	Base-plus-index	Dynamic array ??
Base+(Index*scale)+Displacement	Scaled index	Indexing 2-dimensional array when the elements of the array are 2, 4, or 8 bytes in size