EE-2003
Computer Organization & Assembly Language

Chapter 9: Strings & Arrays

Strings & Arrays

- String Primitive Instructions
- Selected String Procedures
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays

String Primitive Instructions

String Primitive Instructions

- MOVSB, MOVSW, and MOVSD
- CMPSB, CMPSW, and CMPSD
- SCASB, SCASW, and SCASD
- STOSB, STOSW, and STOSD
- LODSB, LODSW, and LODSD

MOVSB, MOVSW, & MOVSD

- The MOVSB, MOVSW, and MOVSD instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI.

- **Example:**

```
.data
source DWORD 0FFFFFFFFh
target DWORD ?
.code
mov esi, OFFSET source
mov edi, OFFSET target
movsd
```

MOVSB, MOVSW, & MOVSD

- ESI and EDI are automatically incremented or decremented:
  - MOVSB increments/decrements by 1
  - MOVSW increments/decrements by 2
  - MOVSD increments/decrements by 4

Direction Flag

- The Direction flag controls the incrementing or decrementing of ESI and EDI.
  - DF = clear (0): increment ESI and EDI
  - DF = set (1): decrement ESI and EDI

- The Direction flag can be explicitly changed using the CLD and STD instructions:

```
CLD     ; clear Direction flag
STD     ; set Direction flag
```

Using a Repeat Prefix

- By itself, a string primitive instruction processes only a single memory value or pair of values.
- REP (a repeat prefix) can be inserted just before MOVSB, MOVSW, or MOVSD
- ECX controls the number of repetitions
- Example: Copy 20 doublewords from source to target

```
.data
source DWORD 20 DUP(?)
target DWORD 20 DUP(?)
.code
cld                 ; direction = forward
mov ecx,LENGTHOF source    ; set REP counter
mov esi,OFFSET source
mov edi,OFFSET target
rep movsd
```

EXAMPLE

- Use MOVSD to delete the first element of the following doubleword array. All subsequent array values must be moved one position forward toward the beginning of the array: array DWORD 1,1,2,3,4,5,6,7,8,9,10

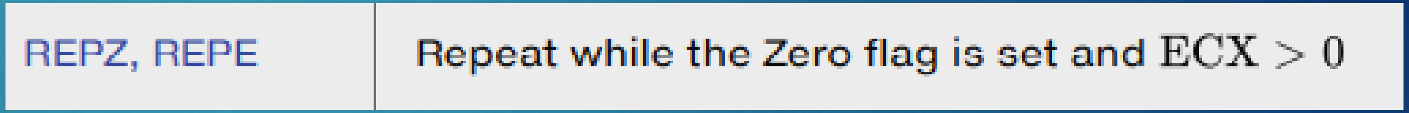
- **Solution:**

```
.data
array DWORD 1,1,2,3,4,5,6,7,8,9,10
.code
cld
mov ecx,(LENGTHOF array) - 1
mov esi,OFFSET array+4
mov edi,OFFSET array
rep movsd
```

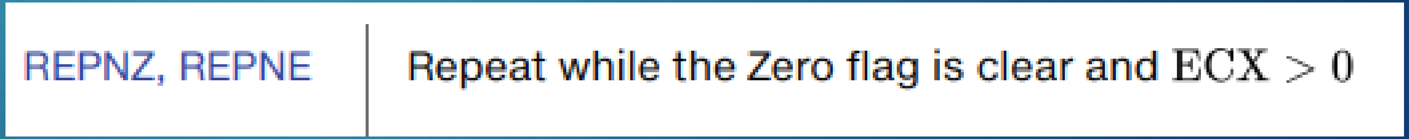
CMPSB, CMPSW, & CMPSD

- The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI
  - CMPSB compares bytes
  - CMPSW compares words
  - CMPSD compares doublewords

- Repeat prefix often used
  - REPE (REPZ)
  - REPNE (REPNZ)



REPZ, REPE	Repeat while the Zero flag is set and $\mathrm{ECX} > 0$



REPNZ, REPNE	Repeat while the Zero flag is clear and $\mathrm{ECX} > 0$

Comparing a Pair of Doublewords

- If source > target, the code jumps to label L1; otherwise, it jumps to label L2

- **Solution:**

```
.data
source DWORD 1234h
target DWORD 5678h
.code
mov esi, OFFSET source
mov edi, OFFSET target
cmpsd                  ; compare doublewords
ja L1                  ; jump if source > target
jmp L2                 ; jump if source <= target
```

Comparing a Pair of Doublewords

- Modify the program in the previous slide by declaring both source and target as WORD variables. Make any other necessary changes.
- **Solution:**

Comparing a Pair of Doublewords

- Use a REPE (repeat while equal) prefix to compare corresponding elements of two arrays
- **Solution:**

```
.data
COUNT = 20
source DWORD COUNT DUP(?)
target DWORD COUNT DUP(?)
.code
mov ecx, COUNT                ; repetition count
mov esi, OFFSET source
mov edi, OFFSET target
cld                           ; direction = forward
repe cmpsd                    ; repeat while equal
```

Example: Comparing Two Strings

- This program compares two strings (source and destination). It displays a message indicating whether the lexical value of the source string is less than the destination string.

- **Solution:**

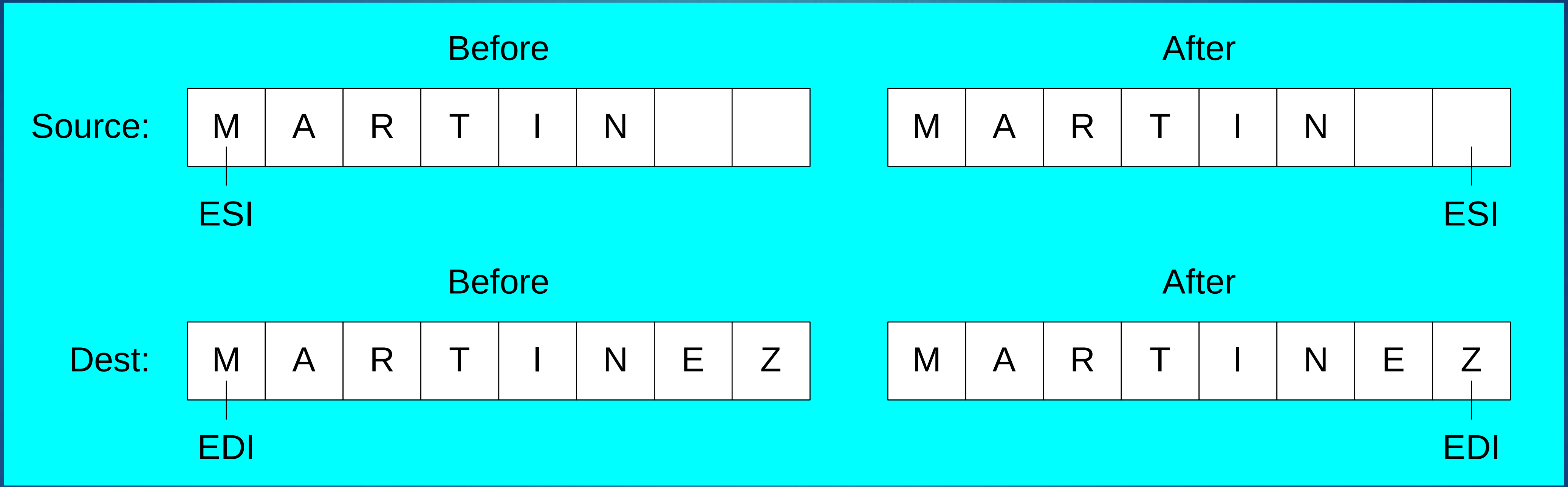
```
.data
source BYTE "MARTIN  "
dest   BYTE "MARTINEZ"
str1 BYTE "Source is smaller",0dh,0ah,0
str2 BYTE "Source is not smaller",0dh,0ah,0
```

Example: Comparing Two Strings

```
.code
main PROC
cld                                 ; direction = forward
mov  esi, OFFSET source
mov  edi, OFFSET dest
mov  ecx, LENGTHOF source
repe cmpsb
jb   source_smaller
mov  edx,OFFSET str2                ; "source is not smaller"
jmp  done
source_smaller:
mov  edx,OFFSET str1                ; "source is smaller"
done:
call WriteString
exit
main ENDP
END main
```

Example: Comparing Two Strings

- The following diagram shows the final values of ESI and EDI after comparing the strings:



Before

Source: M A R T I N

ESI

Before

Dest: M A R T I N E Z

EDI

After

M A R T I N

ESI

After

M A R T I N E Z

EDI

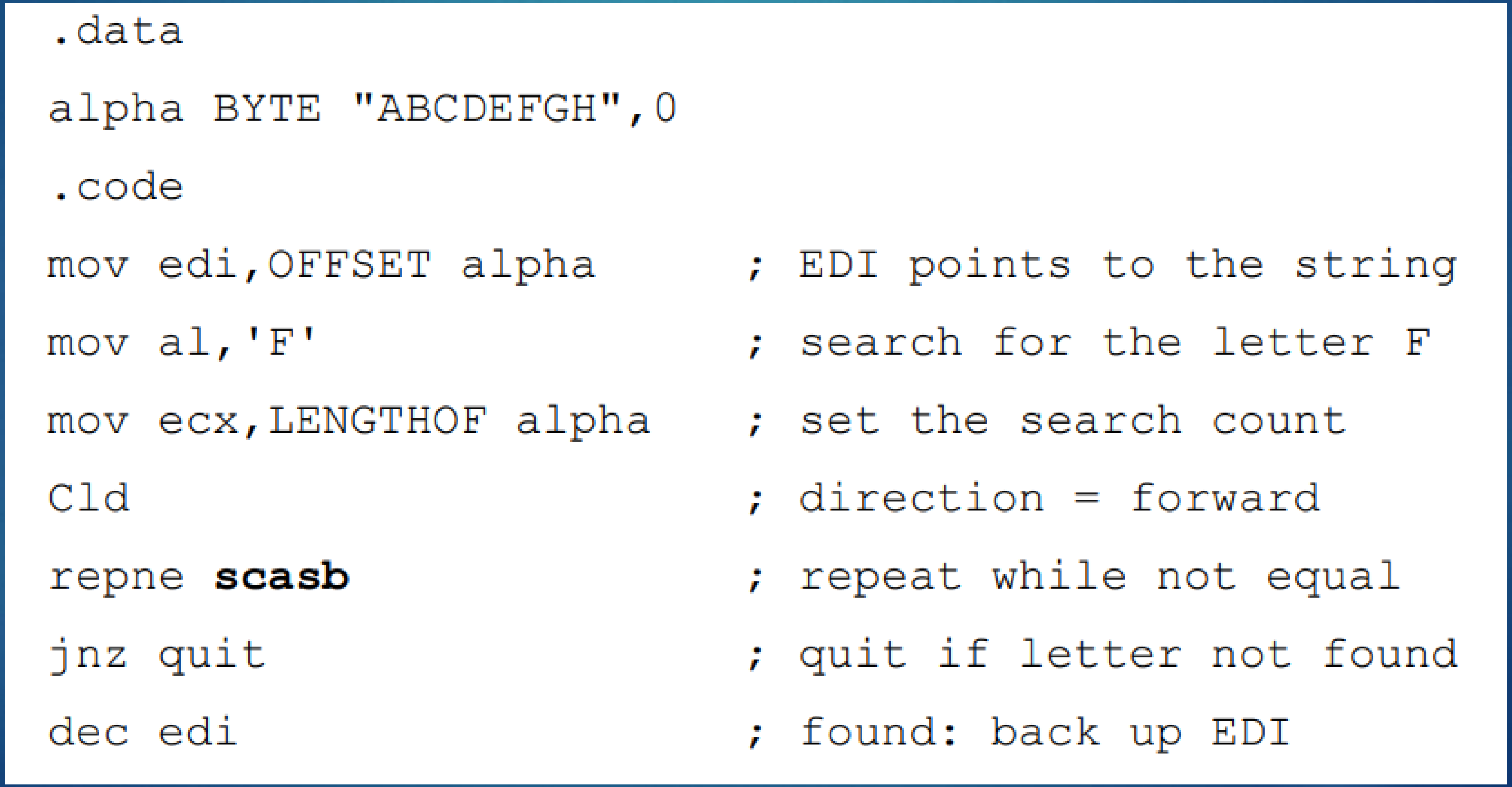
SCASB, SCASW, & SCASD

- The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or doubleword, respectively, addressed by EDI

- Useful types of searches:
  - Search for a specific element in a long string or array
  - Search for the first element that does not match a given value

SCASB, SCASW, & SCASD

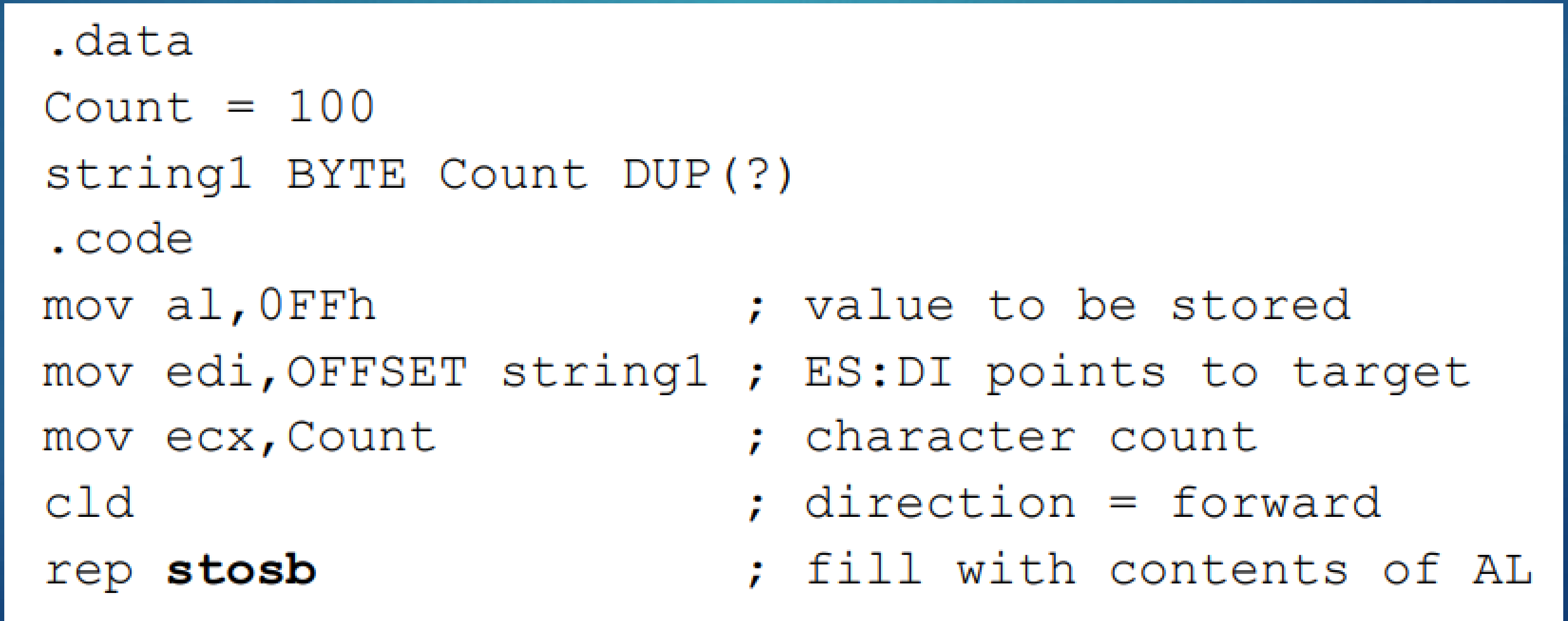
- Search for the letter 'F' in a string named alpha:
- **Solution:**



```
.data
alpha BYTE "ABCDEFGH",0
.code
mov edi,OFFSET alpha     ; EDI points to the string
mov al,'F'               ; search for the letter F
mov ecx,LENGTHOF alpha   ; set the search count
Cld                      ; direction = forward
repne scasb              ; repeat while not equal
jnz quit                 ; quit if letter not found
dec edi                  ; found: back up EDI
```

STOSB, STOSW, & STOSD

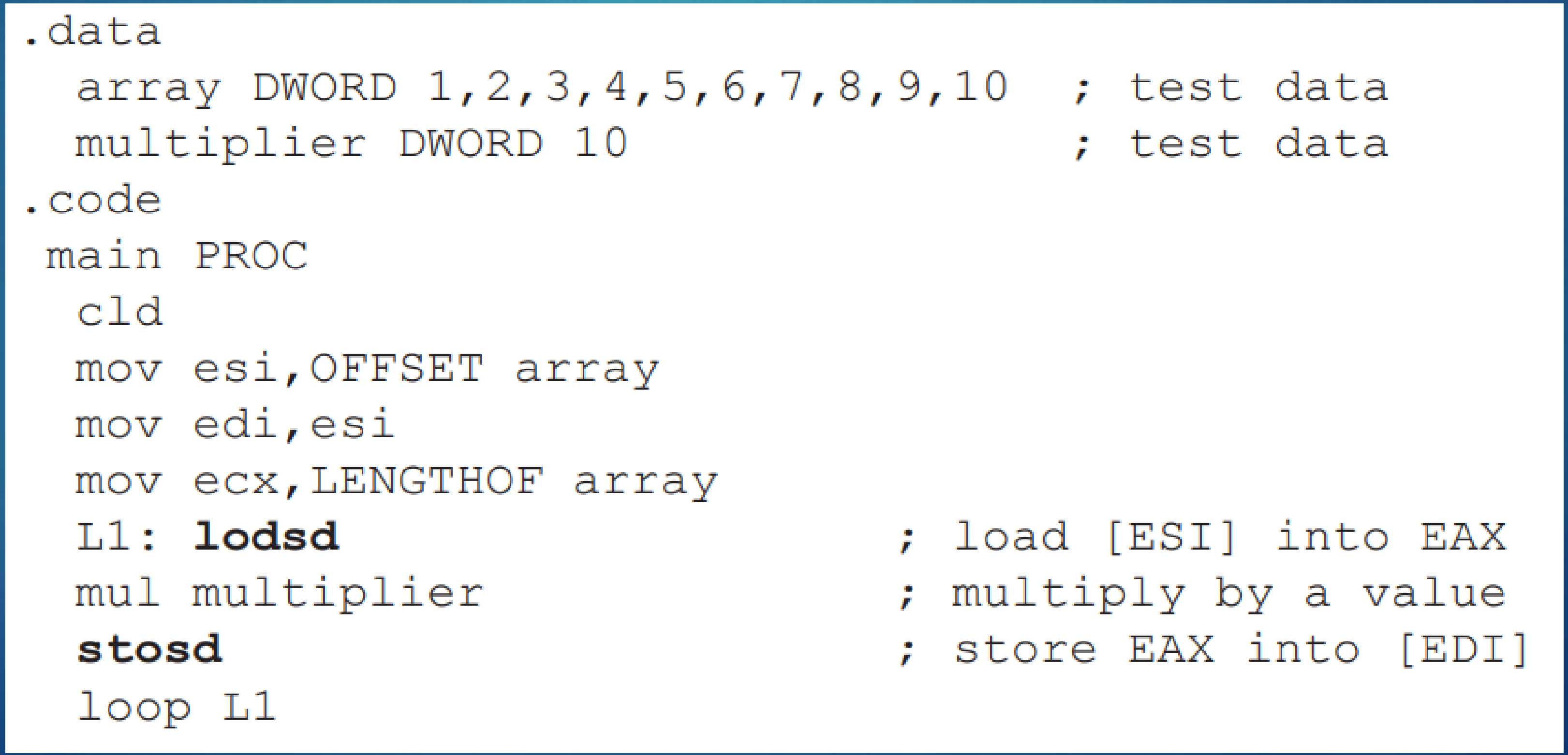
- The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI
- When used with the REP prefix, these instructions are useful for filling all elements of a string or array with a single value
- **Example**: fill an array with 0FFh



```
.data
Count = 100
string1 BYTE Count DUP(?)
.code
mov al,0FFh                ; value to be stored
mov edi,OFFSET string1 ; ES:DI points to target
mov ecx,Count              ; character count
cld                        ; direction = forward
rep stosb                  ; fill with contents of AL
```

LODSB, LODSW & LODSD

- LODSB, LODSW, and LODSD load a byte or word from memory at ESI into AL/AX/EAX, respectively.
- •The REP prefix is rarely used with LODS because each new value loaded into the accumulator overwrites its previous contents.
- • Instead, LODS is used to load a single value.
- **Example**:



```
.data
  array DWORD 1,2,3,4,5,6,7,8,9,10  ; test data
  multiplier DWORD 10               ; test data
.code
 main PROC
  cld
  mov esi,OFFSET array
  mov edi,esi
  mov ecx,LENGTHOF array
  L1: lodsd                   ; load [ESI] into EAX
  mul multiplier              ; multiply by a value
  stosd                       ; store EAX into [EDI]
  loop L1
```

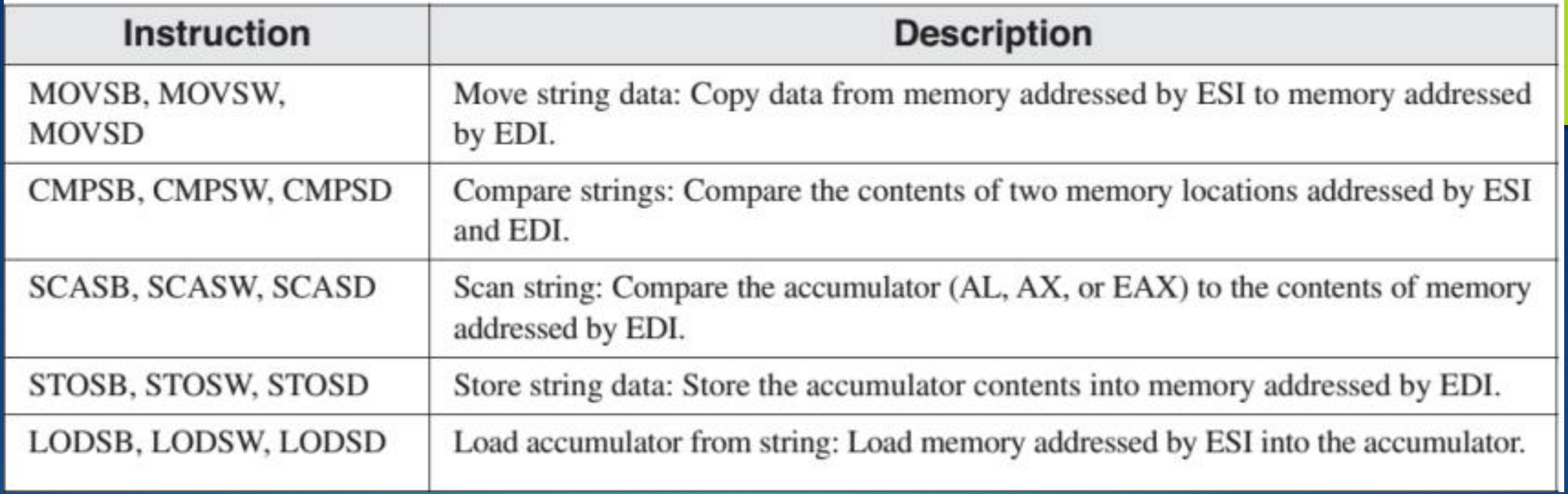
Exercixe

- Write a program that converts each unpacked binary-coded decimal byte belonging to an array into an ASCII decimal byte and copies it to a new array.

```
.data
array BYTE 1,2,3,4,5,6,7,8,9
dest  BYTE (LENGTHOF array) DUP(?)
```

- **Solution**:

```
.code
    mov esi,OFFSET array
    mov edi,OFFSET dest
    mov ecx,LENGTHOF array
     cld                             ; direction = up
L1:lodsb                             ; load into AL
    or al,30h                        ; convert to ASCII
    stosb                            ; store into memory
    loop L1
```



Instruction	Description
MOVSB, MOVSW, MOVSD	Move string data: Copy data from memory addressed by ESI to memory addressed by EDI.
CMPSB, CMPSW, CMPSD	Compare strings: Compare the contents of two memory locations addressed by ESI and EDI.
SCASB, SCASW, SCASD	Scan string: Compare the accumulator (AL, AX, or EAX) to the contents of memory addressed by EDI.
STOSB, STOSW, STOSD	Store string data: Store the accumulator contents into memory addressed by EDI.
LODSB, LODSW, LODSD	Load accumulator from string: Load memory addressed by ESI into the accumulator.

- Although they are called string primitives, they are not limited to character arrays.

- Each instruction implicitly uses ESI, EDI, or both registers to address memory.

- String primitives execute efficiently because they automatically repeat and increment array indexes.

Selected String Procedures

PROTO DIRECTIVE

- Creates a procedure prototype

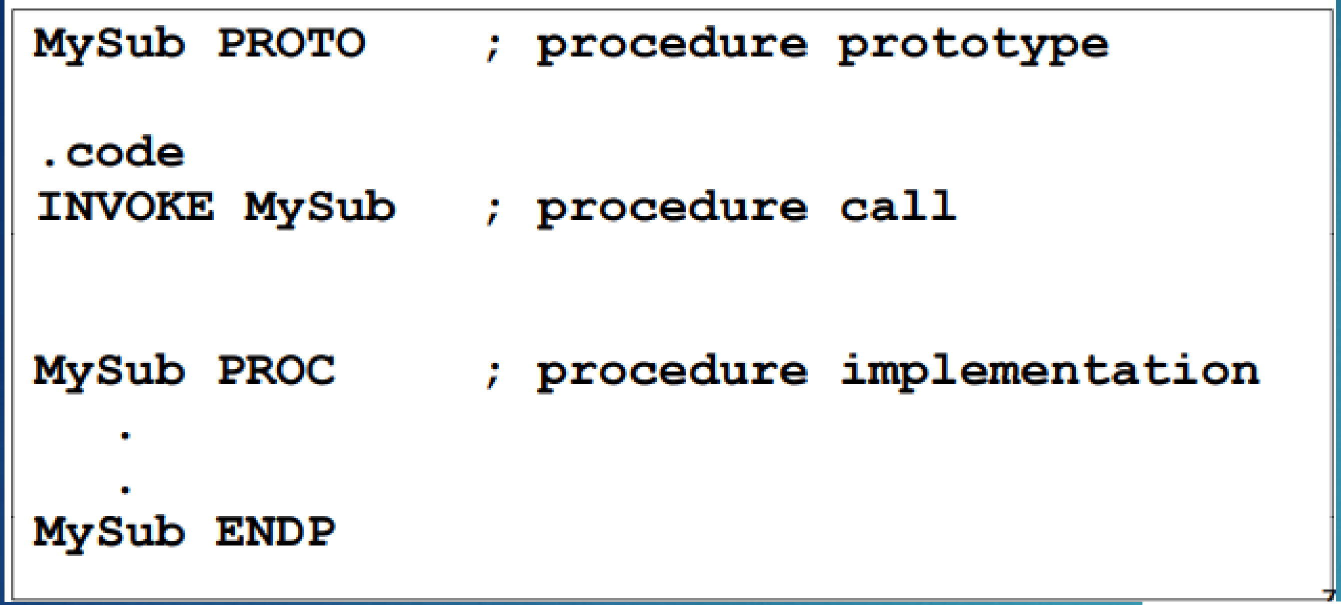
label PROTO paramList

- Every procedure called by the INVOKE directive must have a prototype.

- A complete procedure definition can also serve as its own prototype.

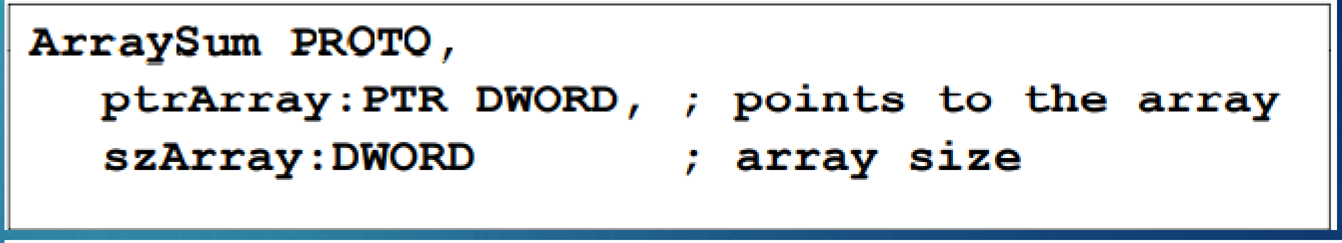
- Standard configuration: PROTO appears at top of the program listing, INVOKE appears in the code segment, and the procedure implementation occurs later in the program.

EXAMPLE

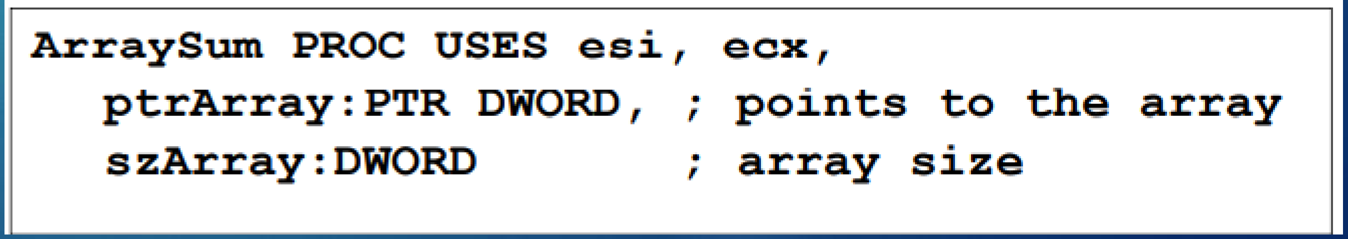


```
MySub PROTO     ; procedure prototype

.code
INVOKE MySub    ; procedure call


MySub PROC      ; procedure implementation
   .
   .
MySub ENDP
```

- Prototype for the ArraySum procedure, showing its parameter list:



```
ArraySum PROTO,
   ptrArray:PTR DWORD,  ; points to the array
   szArray:DWORD        ; array size
```

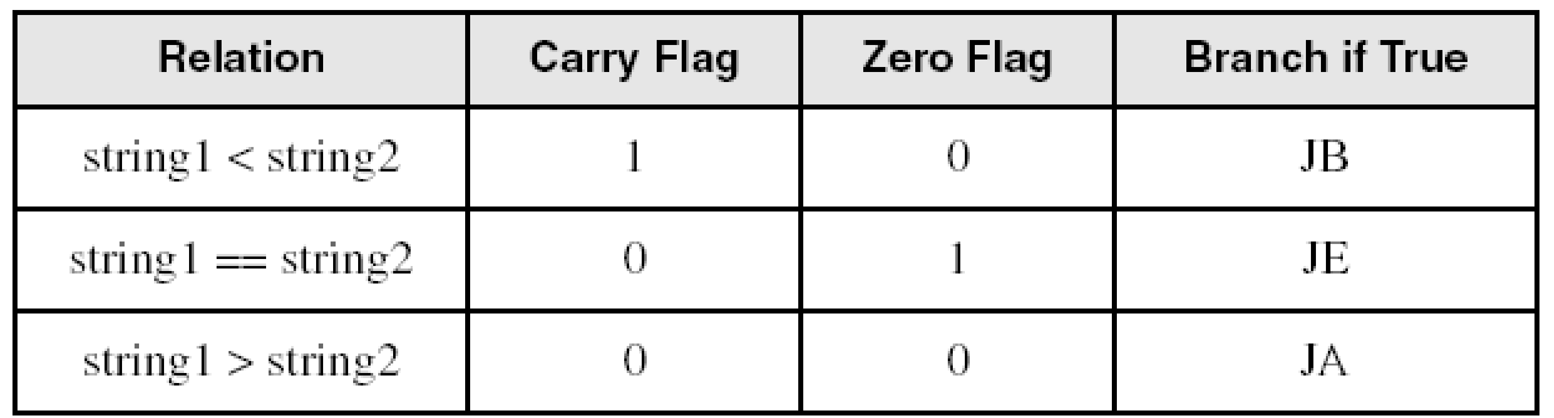


```
ArraySum PROC USES esi, ecx,
   ptrArray:PTR DWORD,  ; points to the array
   szArray:DWORD        ; array size
```

STR_COMPARE PROCEDURE

- Compares string1 to string2, setting the Carry and Zero flags accordingly
- **PROTOTYPE (PROTO):**

```
Str_compare PROTO,
  string1:PTR BYTE,            ; pointer to string
  string2:PTR BYTE             ; pointer to string
```



Relation	Carry Flag	Zero Flag	Branch if True
string1 < string2	1	0	JB
string1 == string2	0	1	JE
string1 > string2	0	0	JA

Str_compare Procedure Source Code

```
Str_compare PROC USES eax edx esi edi,
    string1:PTR BYTE, string2:PTR BYTE
  mov esi,string1
  mov edi,string2
L1: mov  al,[esi]
  mov  dl,[edi]
  cmp  al,0            ; end of string1?
  jne  L2                    ; no
  cmp  dl,0            ; yes: end of string2?
  jne  L2                    ; no
  jmp  L3                    ; yes, exit with ZF = 1
L2:cmp  al,dl                ; chars equal?
Jne L3
  inc  esi             ; point to next
  inc  edi
jmp   L1                     ; yes: continue loop
L3: ret
Str_compare ENDP
```

Str_length Procedure

- Calculates the length of a null-terminated string and returns the length in the EAX register
- **PROTOTYPE (PROTO):**

```
Str_length PROTO,
 pStringPTR BYTE                          ; pointer to string
```

- **Example:**

```
.data
myString BYTE "abcdefg",0
.code
   INVOKE Str_length,  ADDR myString          ; EAX = 7
```

Str_length Procedure Source Code

```
Str_length PROC USES edi,
    pString:PTR BYTE            ; pointer to string

    mov edi, pString
    mov eax,0                   ; character count
L1:
    cmp byte ptr [edi],0        ; end of string?
    je  L2                      ; yes: quit
    inc edi                     ; no: point to next
    inc eax                     ; add 1 to count
    jmp L1
L2: ret
Str_length ENDP
```

Str_copy Procedure

- Copies a null-terminated string from a source location to a target location
- **PROTOTYPE (PROTO):**

```
Str_copy PROTO,
  source:PTR BYTE,    ; pointer to string
  target:PTR BYTE     ; pointer to string
```

- **Example**:

```
.data
source BYTE "hello",0
target BYTE SIZEOF source DUP (?)

.code
     INVOKE Str_copy,  ADDR source,  ADDR target
```

Str_copy Procedure Source Code

```
Str_copy PROC USES eax ecx esi edi,
    source:PTR BYTE,              ; source string
    target:PTR BYTE               ; target string

    INVOKE Str_length,source      ; EAX = length source
    mov ecx,eax                   ; REP count
    inc ecx                       ; add 1 for null byte
    mov esi,source
    mov edi,target
    cld                           ; direction = up
    rep movsb                     ; copy the string
    ret
Str_copy ENDP
```

Str_trim Procedure

- The Str_trim procedure removes all occurrences of a selected trailing character from a null-terminated string
- **PROTOTYPE (PROTO)**:

```
Str_trim PROTO,
  pString:PTR BYTE,    ; points to string
  char:BYTE            ; char to remove
```

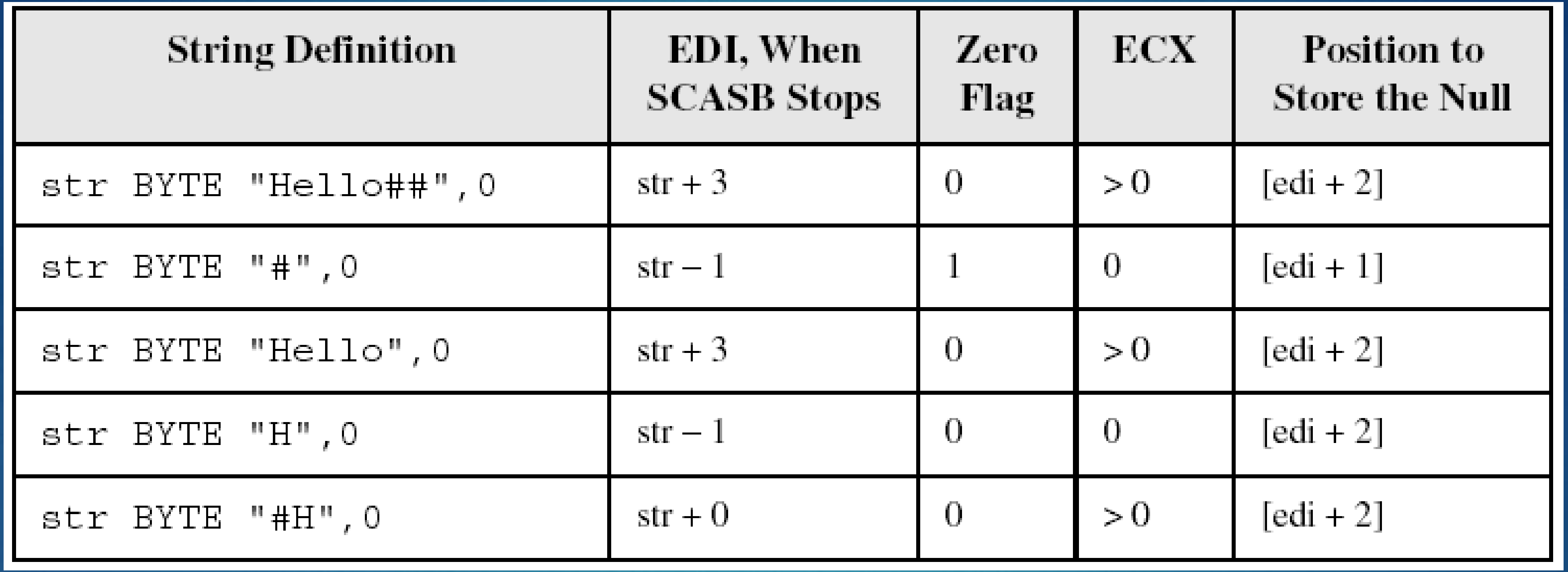
- **Example**:

```
.data
myString BYTE "Hello###",0
.code
   INVOKE Str_trim, ADDR myString, '#'

myString = "Hello"
```

Str_trim Procedure

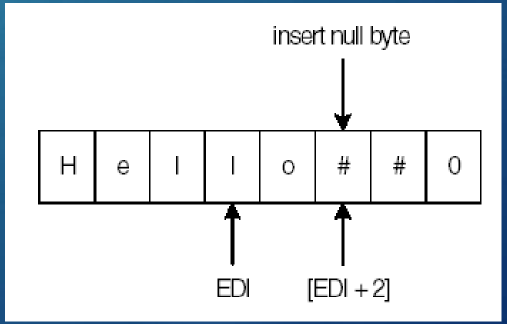
- Str_trim checks a number of possible cases (shown here with # as the trailing character:
- The string is empty
- The string contains other characters followed by one or more trailing characters, as in "Hello##"
- The string contains only one character, the trailing character, as in "#"
- The string contains no trailing character, as in "Hello" or "H"
- The string contains one or more trailing characters followed by one or more nontrailing characters, as in "#H" or "###Hello"

Str_trim Procedure



String Definition	EDI, When SCASB Stops	Zero Flag	ECX	Position to Store the Null
`str BYTE "Hello##",0`	str + 3	0	> 0	[edi + 2]
`str BYTE "#",0`	str – 1	1	0	[edi + 1]
`str BYTE "Hello",0`	str + 3	0	> 0	[edi + 2]
`str BYTE "H",0`	str – 1	0	0	[edi + 2]
`str BYTE "#H",0`	str + 0	0	> 0	[edi + 2]

Using the first definition in the table, position of EDI when SCASB stops:



Str_trim Procedure Source Code

```
Str_trim PROC USES eax ecx edi,
 pString:PTR BYTE,                  ; points to string
 char: BYTE                         ; character to remove
 mov edi,pString                    ; prepare to call  Str_length
 INVOKE Str_length,edi              ; returns the length in EAX
 cmp eax,0                          ; is the length equal to zero?
 je L3 ; yes: exit now
 mov ecx,eax                        ; no: ECX = string length
 dec eax
 add edi,eax                        ; point to last character
L1: mov al,[edi]                    ; get a character
 cmp al,char                        ; is it the delimiter?
 jne L2                             ; no: insert null byte
 dec edi                            ; yes: keep backing up
 loop L1                            ; until beginning reached
L2: mov BYTE PTR [edi+1],0          ; insert a null byte
L3: ret
Stmr_trim ENDP
```

Str_ucase Procedure

- The Str_ucase procedure converts a string to all uppercase characters. It returns no value.
- **PROTOTYPE (PROTO)**:

```
Str_ucase PROTO,
  pString:PTR BYTE          ; pointer to string
```

- **Example**:

```
.data
myString BYTE "Hello",0
.code
    INVOKE Str_ucase, ADDR myString
```

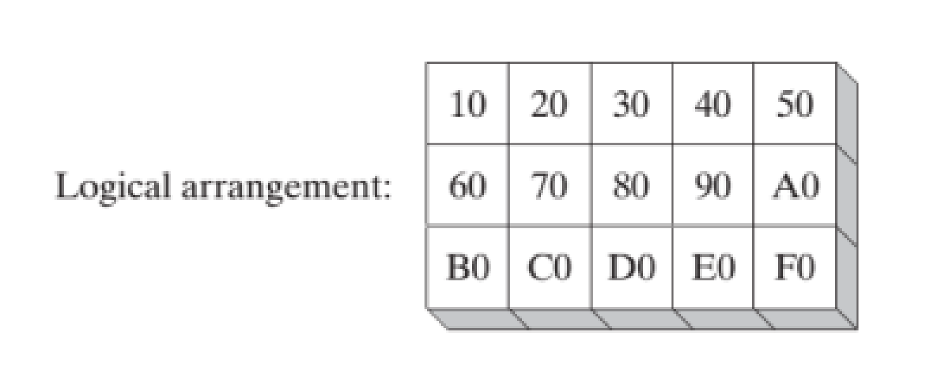
Str_ucase Procedure Source Code

```
Str_ucase PROC USES eax esi,
     pString:PTR BYTE
     mov esi,pString

L1:  mov al,[esi]                     ; get char
     cmp al,0                         ; end of string?
     je  L3                           ; yes: quit
     cmp al,'a’                       ; below "a"?
     jb  L2
     cmp al,'z’                       ; above "z"?
     ja  L2
     and BYTE PTR [esi],11011111b     ; convert the char

L2:  inc esi                          ; next char
     jmp L1

L3: ret
Str_ucase ENDP
```

Two-Dimensional Arrays

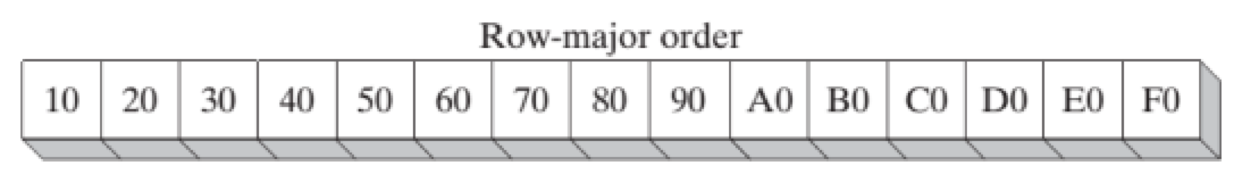
- The two methods of arranging the rows and columns in memory: row-major order and column-major order.

- If you implement a two-dimensional array in assembly language, you can choose either ordering method.



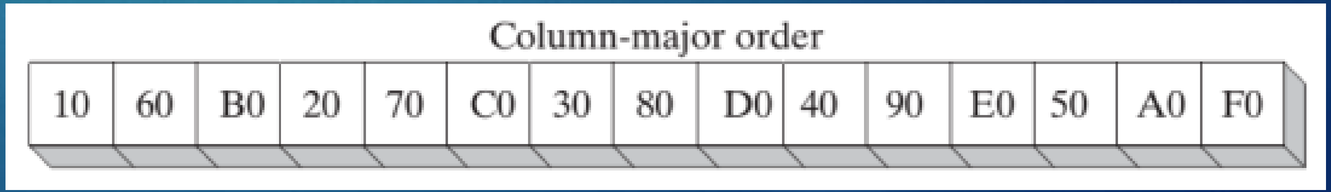
Logical arrangement:

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0



Row-major order

10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0



Column-major order

10	60	B0	20	70	C0	30	80	D0	40	90	E0	50	A0	F0

THE TWO OPERAND TYPES

- Base-Index Operands
- Base-Index Displacement Operands

Base-Index Operand

- A base-index operand adds the values of two registers (called base and index), producing an effective address
  - Any two 32-bit general-purpose registers may be used (Note: esp is not used for general purpose)

- [base + index]

Base-Index-Displacement Operand

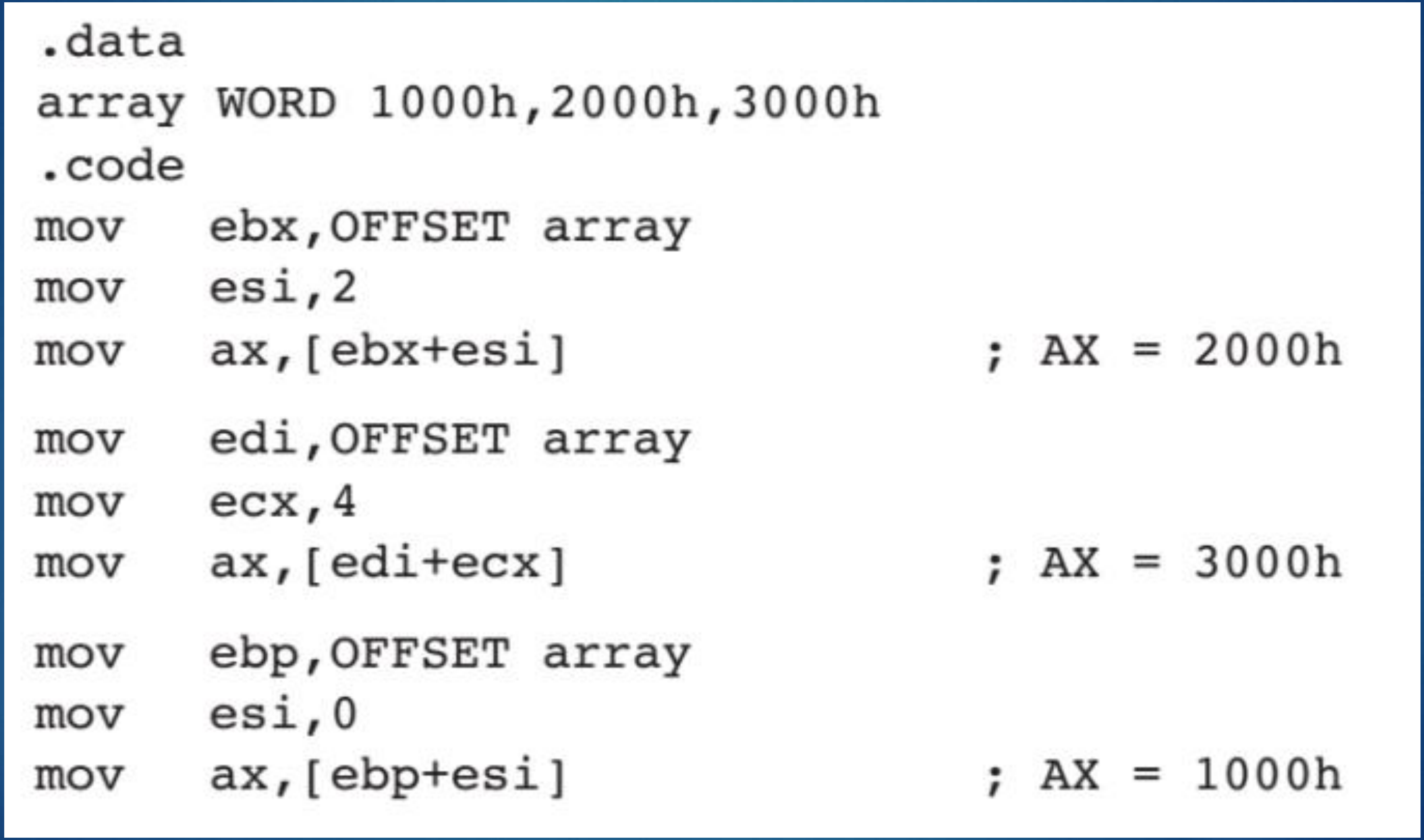
- **A base-index-displacement operand combines a displacement, a base register, an index register, and an optional scale factor to produce an effective address.**

- Displacement can be the name of a variable or a constant expression.
- Common Format:

[base + index + displacement]

displacement[base + index]

BASE-INDEX OPERANDS



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

mov   edi,OFFSET array
mov   ecx,4
mov   ax,[edi+ecx]               ; AX = 3000h

mov   ebp,OFFSET array
mov   esi,0
mov   ax,[ebp+esi]               ; AX = 1000h
```

Two-Dimensional Array Example

- Imagine a table with three rows and five columns. The data can be arranged in any format on the page:

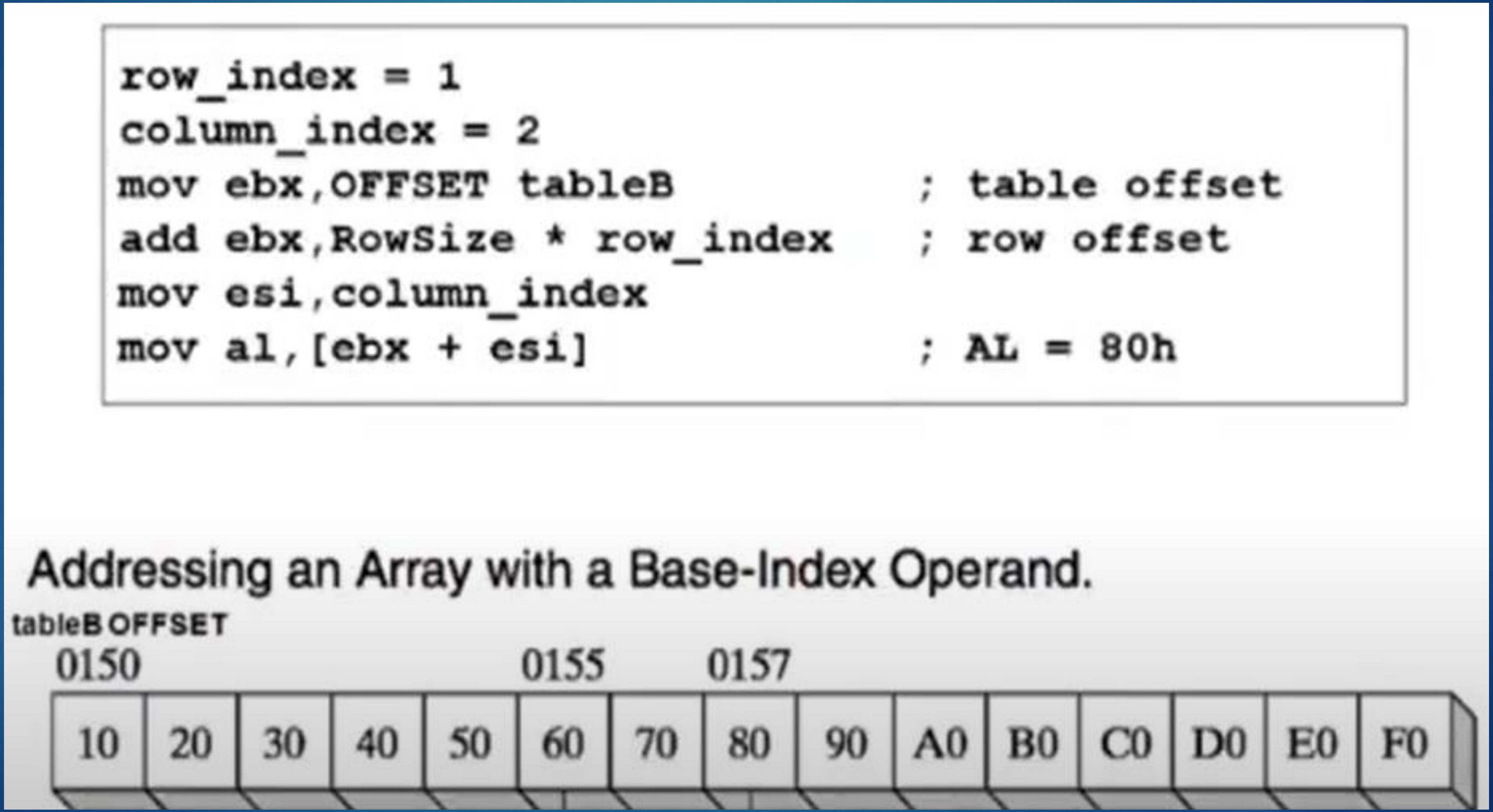
```
table  BYTE   10h,   20h,   30h,   40h,   50h
       BYTE   60h,   70h,   80h,   90h, 0A0h
       BYTE 0B0h, 0C0h, 0D0h, 0E0h, 0F0h
NumCols = 5
```

- Alternative format:

```
table  BYTE  10h,20h,30h,40h,50h,60h,70h,80h,90h,0A0h,0B0h,0C0h,0D0h,0E0h,0F0h
NumCols = 5
```

Two-Dimensional Array Example

- The following code loads the table element stored in row 1, column 2:



```
row_index = 1
column_index = 2
mov ebx,OFFSET tableB             ; table offset
add ebx,RowSize * row_index       ; row offset
mov esi,column_index
mov al,[ebx + esi]                ; AL = 80h
```

Addressing an Array with a Base-Index Operand.

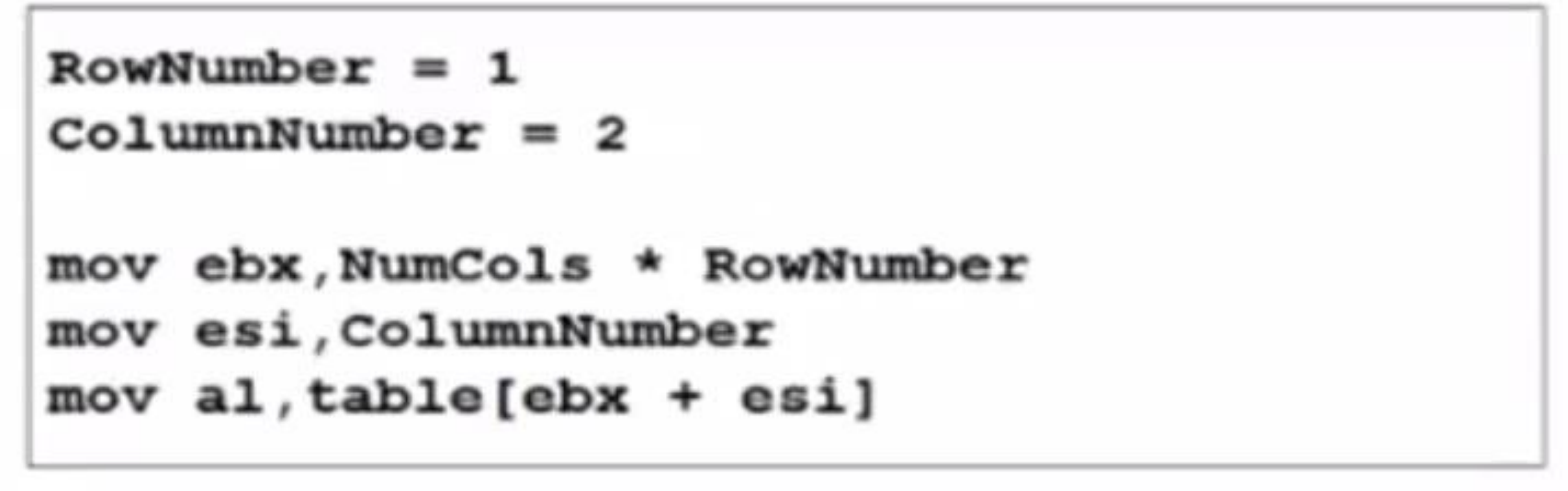
tableB OFFSET

0150 0155 0157

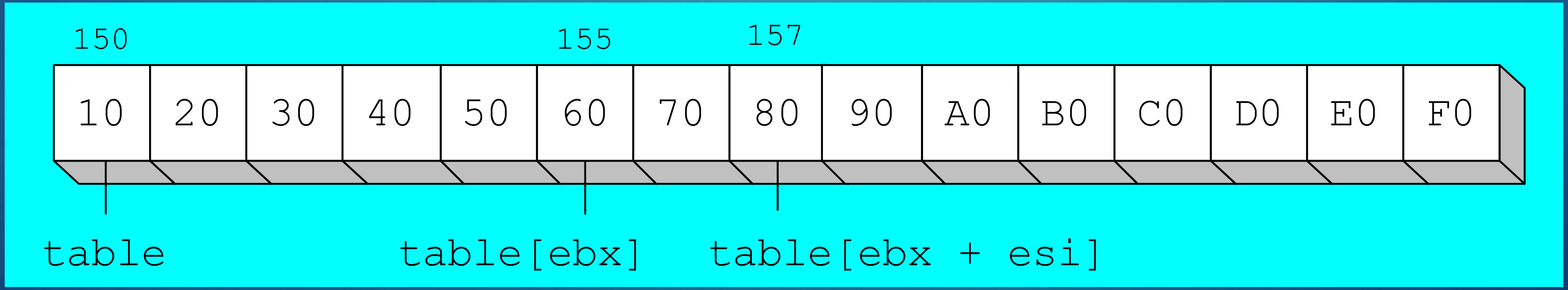
10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0

Two-Dimensional Array Example

- The following code loads the table element stored in row 1, column 2:



```
RowNumber = 1
ColumnNumber = 2

mov ebx,NumCols * RowNumber
mov esi,ColumnNumber
mov al,table[ebx + esi]
```



Selected String Procedures

- The following string procedures may be found in the Irvine32 and Irvine16 libraries:
  - Str_compare Procedure
  - Str_length Procedure
  - Str_copy Procedure
  - Str_trim Procedure
  - Str_ucase Procedure