EL213 Computer Organization & Assembly Language

Lab 11

String Handling Instructions

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

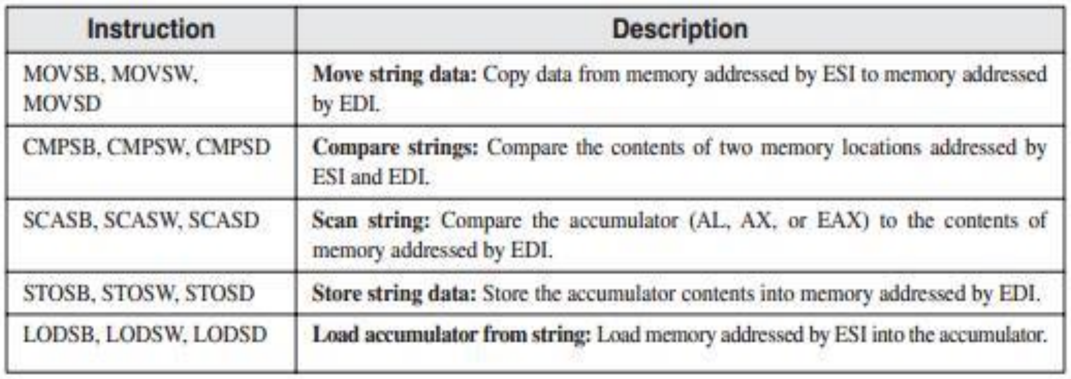
LAB 11

Learning Objectives

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

String Primitive Instructions

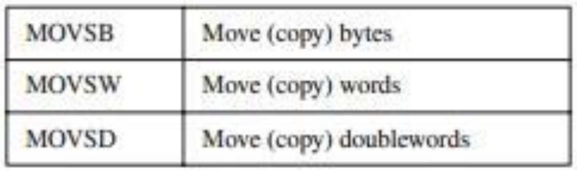
- The x86 instruction set has five groups of instructions for processing arrays of bytes, words, and double words.
- Each instruction implicitly uses ESI, EDI, or both registers to address memory.
- References to the accumulator imply the use of AL, AX, or EAX, depending on the instruction data size. String primitives execute efficiently because they automatically repeat and increment array indexes.



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.

1. MOVSB, MOVSW, and 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. The two registers are either incremented or decremented automatically (based on the value of the Direction flag).



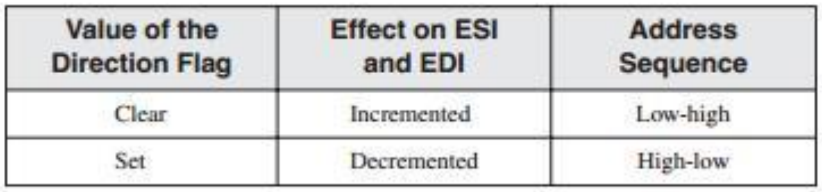
MOVSB	Move (copy) bytes
MOVSW	Move (copy) words
MOVSD	Move (copy) doublewords

We can use a repeat prefix with MOVSB, MOVSW, and MOVSD. The Direction flag determines whether ESI and EDI will be incremented or decremented.

- **Direction Flag**:

String primitive instructions increment or decrement ESI and EDI based on the state of the Direction flag. The Direction flag can be explicitly modified using the CLD and STD instructions:

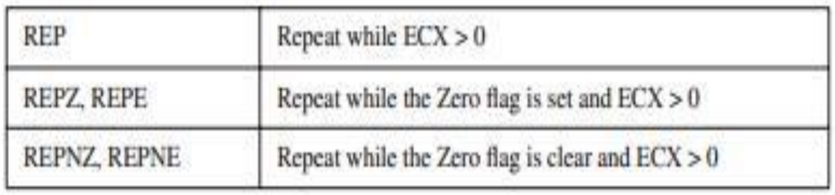
```
CLD            ; clear Direction flag (forward direction)
STD            ; set Direction flag (reverse direction)
```



Value of the Direction Flag	Effect on ESI and EDI	Address Sequence
Clear	Incremented	Low-high
Set	Decremented	High-low

Using a Repeat Prefix

By itself, a string primitive instruction processes only a single memory value or pair of values. If you add a repeat prefix, the instruction repeats, using ECX as a counter. The repeat prefix permits you to process an entire array using a single instruction. The following repeat prefixes are used:



REP	Repeat while ECX > 0
REPZ, REPE	Repeat while the Zero flag is set and ECX > 0
REPNZ, REPNE	Repeat while the Zero flag is clear and ECX > 0

Example: Copy a String

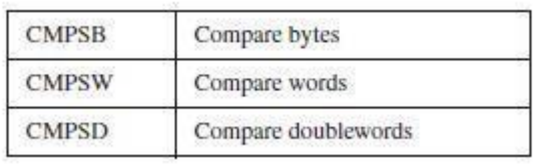
In the following example, MOVSB moves 10 bytes from string1 to string2. The repeat prefix first tests ECX > 0 before executing the MOVSB instruction. If ECX = 0, the instruction is ignored, and control passes to the next line in the program. If ECX > 0, ECX is decremented and the instruction repeats:

```
INCLUDE Irvine32.inc
.data
string1 BYTE ‘this is first string’ ,0
string2 BYTE ‘this is second string’,0

.code
main PROC
cld                                ; clear direction flag
mov esi,OFFSET string1             ; ESI points to source
mov edi,OFFSET string2             ; EDI points to target
mov ecx,sizeof string1             ; set counter to 10
rep movsb                          ; move bytes
mov edx,offset string2
```

```
call writestring
main ENDP
END main
```

2. CMPSB, CMPSW, and CMPSD

The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI: You can use a repeat prefix with CMPSB, CMPSW, and CMPSD. The Direction flag determines the incrementing or decrementing of ESI and EDI.



CMPSB	Compare bytes
CMPSW	Compare words
CMPSD	Compare doublewords

Example: Comparing Doublewords

Suppose you want to compare a pair of double words using CMPSD. In the following example, source has a smaller value than target, so the JA instruction will not jump to label L1.

```
INCLUDE Irvine32.inc
.data
greater BYTE 'source > target',0
lessOrEqual BYTE 'source <target',0

source BYTE 'abcd',0
target BYTE 'abc',0

.code
main PROC
mov esi,OFFSET source
mov edi,OFFSET target
cmpsd                              ; compare doublewords
ja L1                              ; jump if source > target
mov edx,offset lessOrEqual         ;else print source <= target
call writestring
jmp endd

L1:
mov edx,offset greater
call writestring

endd:
Exit
main ENDP
END main
```

3. SCASB, SCASW, and SCASD

The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or double word, respectively, addressed by EDI. The instructions are useful when looking for a single value in a string or array. Combined with the REPE (or REPZ) prefix, the string or array is scanned while ECX > 0 and the value in AL/ AX/ EAX match each subsequent value in memory. The REPNE prefix scans until either AL/AX/EAX matches a value in memory or ECX = 0.

Example: Scan for a Matching Character

In the following example we search the string alpha, looking for the letter F. If the letter is found, EDI points one position beyond the matching character. If the letter is not found, JNZ exits:

```
.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
```

JNZ was added after the loop to test for the possibility that the loop stopped because ECX = 0 and the character in AL was not found.

4. **STOSB, STOSW, and STOSD**

The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI. EDI is incremented or decremented based on the state of the Direction flag. When used with the REP prefix, these instructions are useful for filling all elements of a string or array with a single value. For example, the following code initializes each byte in string1 to 0FFh:

Example:

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

5. LODSB, LODSW, and LODSD

The LODSB, LODSW, and LODSD instructions load a byte or word from memory at ESI into AL/AX/EAX, respectively. ESI is incremented or decremented based on the state of the Direction flag. 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. In the next example, LODSB substitutes for the following two instructions (assuming the Direction flag is clear):

```
mov al,[esi] ; move byte into
AL inc esi ; point to next byte
```

Array Multiplication:

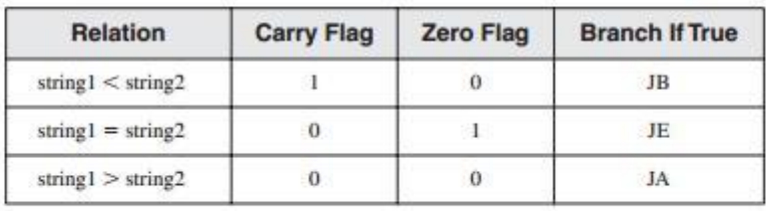
The following program multiplies each element of a doubleword array by a constant value. LODSD and STOSD work together:

```
TITLE Multiply an Array (Mult.asm)
;   This program multiplies each element of an array
;   of 32-bit integers by a constant value.

INCLUDE Irvine32.inc
.data
array DWORD 1,2,3,4,5,6,7,8,9,10            ; test data
multiplier DWORD 10                         ; test data
.code
main PROC
cld                                         ; direction = forward
mov esi,OFFSET array                        ; source index
mov edi,esi                                 ; destination index
mov ecx,LENGTHOF array                      ; loop counter
L1:
Lodsd                                       ; load [ESI] into EAX
mul multiplier                              ; multiply by a value
Stosd                                       ; store EAX into [EDI]
loop L1
Exit
main ENDP
END main
```

➢ **String Procedures**

1. STR_COMPARE

It compares the strings in forward order, starting at the first byte. The comparison is case sensitive because ASCII codes are different for uppercase and lowercase letters. The procedure does not return a value, but the Carry and Zero flags can be interpreted as shown in Table



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

Syntax: INVOKE Str_compare, ADDR string1, ADDR string2

Example:

```
INCLUDE Irvine32.inc
.data
string1 BYTE 'abcd' ,0
string2 BYTE 'abc',0
.code
main PROC
Str_compare PROTO,

INVOKE Str_compare, ADDR string1, ADDR
string2 call dumpRegs

exit
main ENDP
END main
```

2. STR_LENGTH

The Str_length procedure returns the length of a string in the EAX register. When you call it, pass the string’s offset.

Syntax: INVOKE Str_length, ADDR myString

Example:

```
INCLUDE Irvine32.inc
.data
string1 BYTE 'Hello World' ,0

.code
main PROC
mov eax,0
Str_length PROTO,

INVOKE Str_length, ADDR string1
call dumpRegs

exit
main ENDP
END main
```

3. STR_CPY

The Str_copy procedure copies a null-terminated string from a source location to a target location. Before calling this procedure, you must make sure the target operand is large enough to hold the copied string.

Syntax: INVOKE Str_copy, ADDR source, ADDR target

Example:

```
INCLUDE Irvine32.inc
.data
string_1 BYTE "COAL",0
string_2 BYTE " ", 0

.code
main PROC

INVOKE Str_copy,ADDR string_1,ADDR string_2

mov edx,OFFSET string_2
call WriteString

exit
main ENDP
END main
```

4. STR_TRIM PROCEDURE

The Str_trim procedure removes all occurrences of a selected trailing character from a null-terminated string.

Syntax:

INVOKE Str_trim, ADDR string, char_to_trim

Example:

```
INCLUDE Irvine32.inc
.data
string_1 BYTE "Hellooo",0

.code
main PROC

INVOKE Str_trim,ADDR string_1,'o'
mov edx,OFFSET string_1
call WriteString

exit
main ENDP
END main
```

5. STR_UCASE PROCEDURE

The Str_ucase procedure converts a string to all uppercase characters. It returns no value. When you call it, pass the offset of a string.

Syntax: INVOKE Str_ucase, ADDR myString

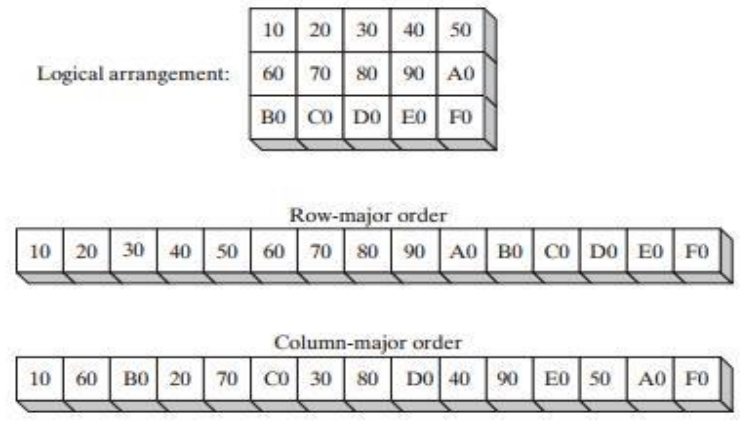
Example:

```
INCLUDE Irvine32.inc
.data
string_1 BYTE "Coal",0

.code
main PROC

INVOKE Str_ucase,ADDR string_1
mov edx,OFFSET string_1
call WriteString

exit
main ENDP
END main
```

➢ Two-Dimensional Arrays

From an assembly language programmer's perspective, a two-dimensional array is a high-level abstraction of a one-dimensional array. High-level languages select one of two methods of arranging the rows and columns in memory: row-major order and column-major order

Row-Major and Column-Major Ordering.



If we implement a two-dimensional array in assembly language, you can choose either ordering method. The x86 instruction set includes two operand types, base-index and base-index-displacement, both suited to array applications.

- **Base-Index Operands**

A base-index operand adds the values of two registers (called base and index), producing an off-set address:

[base+ index]

The square brackets are required. In 32-bit mode, any 32-bit general-purpose registers may be used as base and index registers.

Example:

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

Two-Dimensional Array:

When accessing a two-dimensional array in row-major order, the row offset is held in the base register and the column offset is in the index register. The following table, for example, has three rows and five columns:

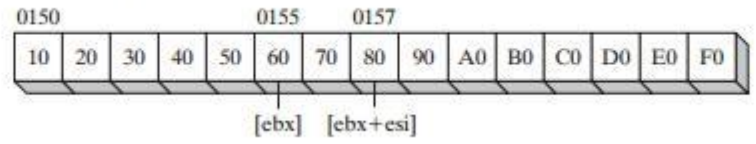
```
tableB  BYTE   10h,  20h,  30h,  40h,  50h
Rowsize = ($ - tableB)
        BYTE   60h,  70h,  80h,  90h, 0A0h
        BYTE  0B0h, 0C0h, 0D0h, 0E0h, 0F0h
```

Suppose we want to access particular number from an array.

Assuming that the coordinates are zero based, the entry at row 1, column 2 contains 80h. We set EBX to the table's offset, add (Rowsize * row_index) to calculate the row offset, and set ESI to the column index:

```
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.



by using Scale Factors

If you're writing code for an array of WORD, multiply the index operand by a scale factor of 2. The following example locates the value at row 1, column 2:

```
tableW  WORD   10h,  20h,  30h,  40h,  50h
RowsizeW = ($ - tableW)
        WORD   60h,  70h,  80h,  90h, 0A0h
        WORD  0B0h, 0C0h, 0D0h, 0E0h, 0F0h
.code
row_index = 1
column_index = 2
mov   ebx,OFFSET tableW              ; table offset
add   ebx,RowSizeW * row_index       ; row offset
mov   esi,column_index
mov   ax,[ebx + esi*TYPE tableW]     ; AX = 0080h
```

- **Base-Index-Displacement Operands:**

A base-index-displacement operand combines a displacement, a base register, an index register, and an optional scale factor to produce an effective address. Here are the formats:

[base+ index+ displacement]

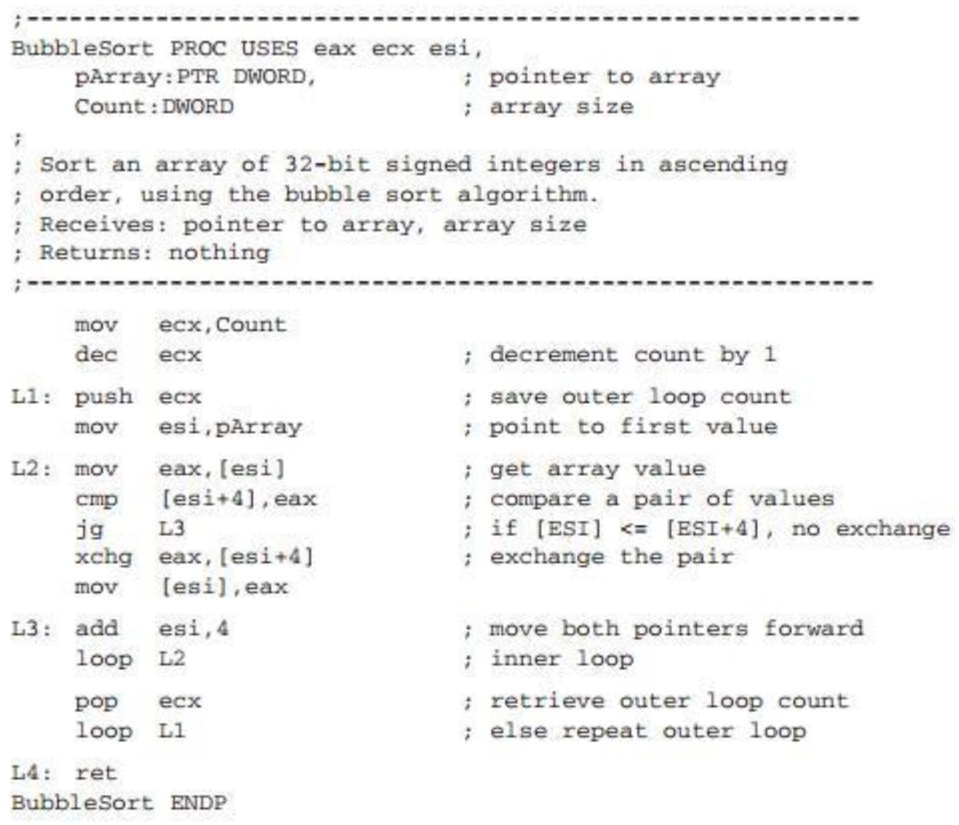
Displacement [base+ index]

Displacement can be the name of a variable or a constant expression. In 32-bit mode, any general-purpose 32-bit registers may be used for the base and index.

➢ Searching and Sorting Integer Arrays

- **Bubble Sort**

A bubble sort compares pairs of array values, beginning in positions 0 and 1. If the compared values are in reverse order, they are exchanged.



```
;----------------------------------------------------------
BubbleSort PROC USES eax ecx esi,
    pArray:PTR DWORD,           ; pointer to array
    Count:DWORD                 ; array size
;
; Sort an array of 32-bit signed integers in ascending
; order, using the bubble sort algorithm.
; Receives: pointer to array, array size
; Returns: nothing
;-----------------------------------------------------------

    mov   ecx,Count
    dec   ecx                   ; decrement count by 1

L1: push  ecx                   ; save outer loop count
    mov   esi,pArray            ; point to first value

L2: mov   eax,[esi]             ; get array value
    cmp   [esi+4],eax           ; compare a pair of values
    jg    L3                    ; if [ESI] <= [ESI+4], no exchange
    xchg  eax,[esi+4]           ; exchange the pair
    mov   [esi],eax

L3: add   esi,4                 ; move both pointers forward
    loop  L2                    ; inner loop

    pop   ecx                   ; retrieve outer loop count
    loop  L1                    ; else repeat outer loop

L4: ret
BubbleSort ENDP
```

Exercise

1. Create a procedure named *Scan_String* to find the index of the first occurrence of the character '#' in the given string.

```
Str1 BYTE '127&j~3#^&*#*#45^',0
```

2. Modify the above procedure to take *offset of string1* and the *character to be searched* as argument.

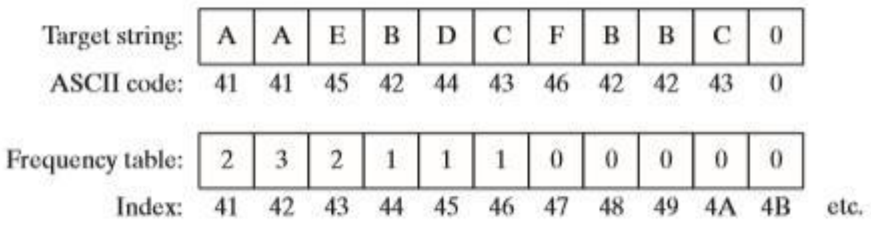
3. Create *IsCompare* procedure to compare two strings.

4. Create a Str_Reverse procedure to reverse strings.

5. Create a procedure that Loads an array of integer by multiplying it with Load(offset array, byte no)

6. Write the procedure to get_frequency Find the frequency of characters:

```
.data
        target BYTE "AAEBDCFBBC",0
        freqTable DWORD 256 DUP(0)
 .code

        INVOKE Get_frequencies, ADDR target, ADDR freqTable
```



Target string:	A	A	E	B	D	C	F	B	B	C	0	
ASCII code:	41	41	45	42	44	43	46	42	42	43	0	
Frequency table:	2	3	2	1	1	1	0	0	0	0	0	
Index:	41	42	43	44	45	46	47	48	49	4A	4B	etc.