

Objectives:

- String Primitive Instructions
- Selected String Procedures
- Two-Dimensional Arrays

Section 01 | String Primitive Instructions

The x86 instruction set has five groups of instructions for processing arrays of [byte](#), [word](#), and [dword](#).

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	
1	MOVSB, MOVSW, MOVSD	Move String Data:	Copy data from memory addressed by ESI to memory addressed by EDI .
2	CMPSB, CMPSW, CMPSD	Compare Strings:	Compare the contents of two memory locations addressed by ESI and EDI .
3	SCASB, SCASW, SCASD	Scan String:	Compare the accumulator (AL , AX , or EAX) to the contents of memory addressed by EDI .
4	STOSB, STOSW, STOSD	Store String Data:	Store the accumulator contents into memory addressed by EDI .
5	LODSB, LODSW, LODSD	Load Accumulator from String:	Load memory addressed by ESI into the accumulator.

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

Instruction	Description
MOVSB	Move (copy) bytes.
MOVSW	Move (copy) words.
MOVSD	Move (copy) double words.

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	Decrement	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:

Prefix	Description
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

```
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,LENGTHOF string1 ; set counter to length of string1
        rep movsb                 ; move bytes
        mov edx,OFFSET string2
        call WriteString
    exit
main ENDP
END main
```

Example: Copy an Array

```
INCLUDE Irvine32.inc
.data
    array1 BYTE 1,2,3,4,5,6,7,8
    array2 BYTE 8 DUP (0)
.code
    main PROC
        cld                      ; clear direction flag
        mov esi,OFFSET array1    ; ESI points to source (array1)
        mov edi,OFFSET array2    ; EDI points to target (array2)
        mov ecx,LENGTHOF array1 ; set counter to length of array1
        rep movsb                 ; move bytes
```

```

    mov ecx, LENGTHOF array1
    mov esi, 0

L1:
    movzx eax, array2[esi]
    call WriteDec
    call Crlf
    inc esi
    loop L1

    exit
main ENDP
END main

```

2. 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](#): You can use a repeat prefix with [CMPSB](#), [CMPSW](#), and [CMPSD](#). The Direction flag determines the incrementing or decrementing of [ESI](#) and [EDI](#).

Instruction	Description
CMPSB	Compare bytes.
CMPSW	Compare words.
CMPSD	Compare double words.

Example: Program to compare a Double Word in a source string with a Double Word in a target string.

```

INCLUDE Irvine32.inc

.data

    greater BYTE 'source > target',0
    lessOrEqual BYTE 'source < target',0
    equal BYTE 'source = target',0

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

.code
main PROC
    mov esi,OFFSET source

```

```

        mov edi,OFFSET target
        cmpsd                                ; compare doublewords
        ja L1
        je L2; jump if source > target
        mov edx,offset lessOrEqual;else print source <= target
        call writestring
        jmp end1

L1:
        mov edx,offset greater
        call writestring
        exit
L2:
        mov edx,offset equal
        call writestring
end1:
        exit
main ENDP
END main

```

3. SCASB, SCASW & SCASD

The **SCASB**, **SCASW**, and **SCASD** instructions compare a value in **AL/AX/EAX** to a **byte**, **word**, or **dword**, 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: 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:

```

INCLUDE Irvine32.inc

.data
    alpha BYTE "ABCDEFGHI",0

    string1 BYTE "Element found", 0
    string2 BYTE "Element not found", 0

.code
main PROC
    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 (as EDI is
                           ; pointing 1 byte beyond the actual
                           ; byte)

        mov edx, OFFSET string1
        call WriteString
        exit

quit:
        mov edx, OFFSET string2
        call WriteString
        exit
main ENDP
END main

```

4. 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**. **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.

Example: The following code initializes each byte in **string1** to **41h**

```

INCLUDE Irvine32.inc

.data
    count = 100
    string1 BYTE count DUP(?)

.code
main PROC
    mov al, 41h          ; 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

    mov edx, OFFSET string1
    call WriteString
    exit
main ENDP
END main

```

5. LODSB, LODSW & LODSD

The **LODSB**, **LODSW**, and **LODSD** instructions load a **byte**, **word** or **dword** 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
```

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

```
INCLUDE Irvine32.inc

.data
    array DWORD 1,2,3,4,5,6,7,8,9,10      ; test data
    multiplier DWORD 10                   ; test data
    string BYTE " ", 0

.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

        mov ecx, LENGTHOF array
        mov esi, OFFSET array
        mov edx, OFFSET string

        L2:
            mov eax, [esi]
            call WriteDec
            call WriteString
            add esi, 4
        loop L2
        exit
    main ENDP
END main
```

Section 02 | 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 'abcdeee' ,0
    string2 BYTE 'abcdee',0

    prompt1 BYTE "string1 is greater.",0
    prompt2 BYTE "strings are equal.",0
    prompt3 BYTE "string1 is smaller.",0

.code
main PROC

    INVOKE Str_Compare, ADDR string1, ADDR string2
    JA st1_greater
    JE equal
    JB st1_smaller

    st1_greater:
        mov edx, OFFSET prompt1
        call WriteString
        jmp end1
```

```

equal:
    mov edx, OFFSET prompt2
    call WriteString
    jmp end1

st1_smaller:
    mov edx, OFFSET prompt3
    call WriteString

end1:
    call Crlf
    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_Compare, ADDR string1
```

[Example:](#)

```

INCLUDE Irvine32.inc

.data

    string1 BYTE 'Hello World' ,0

.code
main PROC

    mov eax,0

    INVOKE Str_Length, ADDR string1
    call WriteDec

    exit
main ENDP
END main

```

3. STR_COPY

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

    source BYTE "Computer Organization & Assembly Language", 0
    target BYTE "                ", 0

.code
    main PROC
        INVOKE Str_Copy, ADDR source, ADDR target
        mov edx, OFFSET target
        call WriteString

        exit
    main ENDP
END main
```

4. STR_TRIM

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 BYTE "Hellooo", 0

.code
    main PROC

        INVOKE Str_Trim, ADDR string, "o"
```

```
    mov edx, OFFSET string
    call WriteString

    exit
main ENDP
END main
```

5. STR_UCASE

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 string
```

Example:

```
INCLUDE Irvine32.inc

.data
    string BYTE "Computer Organization & Assembly Language", 0

.code
main PROC

    INVOKE Str_Ucase, ADDR string
    mov edx, OFFSET string
    call WriteString

    exit
main ENDP
END main
```

Section 03 | 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

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
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

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.

1. Base-Index Operands

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

$$[\text{base} + \text{index}]$$

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 3 rows and 5 columns:

	col 0	col 1	col 2	col 3	col 4
row 0	10	12	14	16	18
row 1	20	22	24	26	28
row 2	30	32	34	36	38

The table is in row-major order and the constant [rowSize](#) is calculated by the assembler as the number of bytes in each table row.

Row-Major Order

10	12	14	16	18	20	22	24	26	28	30	32	34	36	38
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Suppose we want to locate a particular entry in the table using row and column coordinates. Assuming that the coordinates are zero based, the entry at row 1, column 2 contains [24](#). We set [EBX](#) to the table's [offset](#), add [\(rowSize * rowIndex\)](#) to calculate the row [offset](#), and set [ESI](#) to the column index:

```
INCLUDE Irvine32.inc

.data
    table      BYTE 10, 12, 14, 16, 18
    rowSize = ($ - table)
                  BYTE 20, 22, 24, 26, 28
                  BYTE 30, 32, 34, 36, 38

.code
main PROC

    ; accessing element "24" of row 1, column 2
    rowIndex = 1           ; row 1
```

```

columnIndex = 2           ; column 2

    mov ebx, OFFSET table      ; table OFFSET
    add ebx, rowSize * rowIndex ; row OFFSET
    mov esi, columnIndex
    mov al, [ebx + esi]        ; al = table[1,2] = 24

    movzx eax, al
    call WriteDec

    exit
main ENDP
END main

```

If you're writing code for an array of **WORD** or **DWORD**, multiply the index operand by a scale factor of **2** or **4**.

The following example locates the value at row 1, column 2:

```

INCLUDE Irvine32.inc

.data
    table      WORD 10, 12, 14, 16, 18
    rowSize = ($ - table)
                  WORD 20, 22, 24, 26, 28
                  WORD 30, 32, 34, 36, 38

.code
main PROC

    ; accessing element "24" of row 1, column 2
    rowIndex = 1           ; row 1
    columnIndex = 2          ; column 2

    mov ebx, OFFSET table      ; table OFFSET
    add ebx, rowSize * rowIndex ; row OFFSET
    mov esi, columnIndex
    mov ax, [ebx + esi * TYPE table] ; ax = table[1,2] = 24

    movzx eax, ax
    call WriteDec

    exit
main ENDP
END main

```

2. 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.

Example: The following two-dimensional array holds 3 rows of 5 doublewords:

```
INCLUDE Irvine32.inc

.data
    table      DWORD 10, 12, 14, 16, 18
    rowSize = ($ - table)
                DWORD 20, 22, 24, 26, 28
                DWORD 30, 32, 34, 36, 38

.code
main PROC

    ; accessing element "24" of row 1, column 2

    rowNum = 1
    columnIndex = 2

    mov ebx, rowSize * rowNum
    mov esi, columnIndex

    mov eax, table[ebx + esi * TYPE table]
    call WriteDec

    exit
main ENDP
END main
```