National University of Computer & Emerging Sciences, Karachi

EL-2003: Computer Organization & Assembly Language Lab Notes

STRINGS AND ARRAY

Fall 2022

String Primitive Instructions

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.

Using a Repeat Prefix

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

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

```
cld                         ; clear direction flag
mov   esi,OFFSET string1    ; ESI points to source
mov   edi,OFFSET string2    ; EDI points to target
mov   ecx,10                ; set counter to 10
rep   movsb                 ; move 10 bytes
```

ESI and EDI are automatically incremented when MOVSB repeats. This behavior is controlled by the CPU's Direction flag.

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

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

- You can use a repeat prefix with MOVSB, MOVSW, and MOVSD. The Direction flag determines whether ESI and EDI will be incremented or decremented.
- The size of the increment/decrement is shown in the following table:

Instruction	Value Added or Subtracted from ESI and EDI
MOVSB	1
MOVSW	2
MOVSD	4

Example: Copy Doubleword Array

Suppose we want to copy 20 doubleword integers from
source to **target**. After the array is copied, ESI and EDI point one position (4 bytes) beyond the end of each array:

```
.data
source DWORD 20 DUP(0FFFFFFFFh)
target DWORD 20 DUP(?)
.code
cld ; direction = forward
mov ecx,LENGTHOF source ; set REP counter
mov esi,OFFSET source ; ESI points to source
mov edi,OFFSET target ; EDI points to target
rep movsd ; copy doublewords
```

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:

CMPSB	Compare bytes
CMPSW	Compare words
CMPSD	Compare doublewords

You can use a repeat prefix with CMPSB, CMPSW, and CMPSD. The Direction flag determines the incrementing or decrementing of ESI and EDI.

Example: Comparing Doublewords

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

```
.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
```

To compare multiple doublewords, clear the Direction flag (forward direction), initialize ECX as a counter, and use a repeat prefix with CMPSD:

```
mov esi,OFFSET source
mov edi,OFFSET target
cld ; direction = forward
mov ecx,LENGTHOF source ; repetition counter
repe cmpsd ; repeat while equal
```

The REPE prefix repeats the comparison, incrementing ESI and EDI automatically until ECX equals zero or a pair of doublewords is found to be different.

SCASB, SCASW, and SCASD

- The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or doubleword, 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 matches each subsequent value in memory.
- The REPNE prefix scans until either AL/AX/EAX matches a value in memory or ECX = 0.

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.

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:

```
.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
```

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.

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 Example

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

```
; 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
```

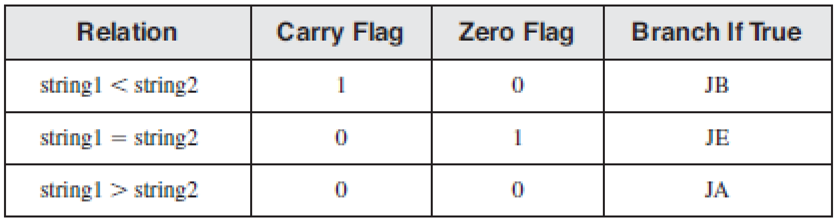
Selected String Procedures

we will demonstrate several procedures from the Irvine32 library that manipulate null-terminated strings. The procedures are clearly similar to functions in the standard C library:

```
; Copy a source string to a target string.
Str_copy PROTO,
source:PTR BYTE,
target:PTR BYTE

; Return the length of a string (excluding the null byte) in EAX.
Str_length PROTO,
pString:PTR BYTE

; Compare string1 to string2. Set the Zero and

; Carry flags in the same way as the CMP instruction.
Str_compare PROTO,
string1:PTR BYTE,
string2:PTR BYTE

; Trim a given trailing character from a string.

; The second argument is the character to trim.
Str_trim PROTO,
pString:PTR BYTE,
char:BYTE

; Convert a string to upper case.
Str_ucase PROTO,
pString:PTR BYTE
```

Str_compare Procedure

- ✓ The **Str_compare** procedure compares two strings.
  The calling format is
  INVOKE Str_compare, ADDR string1, ADDR string2
- ✓ 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 using the *string1* and *string2* arguments.



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

Str_length Procedure

The **Str_length** procedure returns the length of a string in the EAX register. When you call it, pass the string's offset.
INVOKE Str_length, ADDR myString

Str_copy Procedure

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.

The syntax for calling Str_copy is
INVOKE Str_copy, ADDR *source*, ADDR *target*

Str_trim Procedure

The **Str_trim** procedure removes all occurrences of a selected trailing character from a nullterminated string.
The syntax for calling it is
INVOKE Str_trim, ADDR *string, char_to_trim*

Let's look at some code that tests the Str_trim procedure. The INVOKE statement passes the address of a string to Str_trim:

```
.data
string_1 BYTE "Helloeee",0
.code

INVOKE Str_trim,ADDR string_1,'e'
```

Input String	Expected Modified String
"Hello##"	"Hello"
"#"	"" (empty string)
"Hello"	"Hello"
"H"	"H"
"#H"	"#H"

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:

INVOKE Str_ucase, ADDR myString

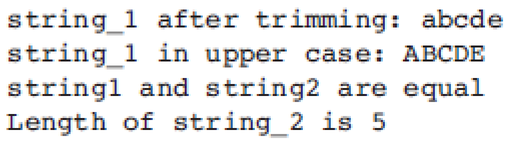
String Library Demo Program

```
; String Library Demo (StringDemo.asm)
; This program demonstrates the string-handling procedures in
; the book's link library.
INCLUDE Irvine32.inc
.data
string_1 BYTE "abcde////",0
string_2 BYTE "ABCDE",0
msg0 BYTE "string_1 in upper case: ",0
msg1 BYTE "string_1 and string_2 are equal",0
msg2 BYTE "string_1 is less than string_2",0
msg3 BYTE "string_2 is less than string_1",0
msg4 BYTE "Length of string_2 is ",0
msg5 BYTE "string_1 after trimming: ",0
.code
main PROC
call trim_string
call upper_case
call compare_strings
call print_length
exit
main ENDP

trim_string PROC
; Remove trailing characters from string_1.
INVOKE Str_trim, ADDR string_1, '/'
mov edx,OFFSET msg5
call WriteString
```

```
mov edx,OFFSET string_1
call WriteString
call Crlf
ret
trim_string ENDP

upper_case PROC
; Convert string_1 to upper case.
mov edx,OFFSET msg0
call WriteString
INVOKE Str_ucase, ADDR string_1
mov edx,OFFSET string_1
call WriteString
call Crlf

ret
upper_case ENDP


compare_strings PROC
; Compare string_1 to string_2.
INVOKE Str_compare, ADDR string_1, ADDR string_2
.IF ZERO?
mov edx,OFFSET msg1
.ELSEIF CARRY?
mov edx,OFFSET msg2 ; string 1 is less than...
.ELSE
mov edx,OFFSET msg3 ; string 2 is less than...
.ENDIF
call WriteString
call Crlf
ret
compare_strings ENDP

print_length PROC
; Display the length of string_2.
mov edx,OFFSET msg4
call WriteString
INVOKE Str_length, ADDR string_2
call WriteDec
call Crlf
ret
print_length ENDP
END main
```

Trailing characters are removed from string_1 by the call to Str_trim. The string is converted to upper case by calling the Str_ucase procedure.

Program Output

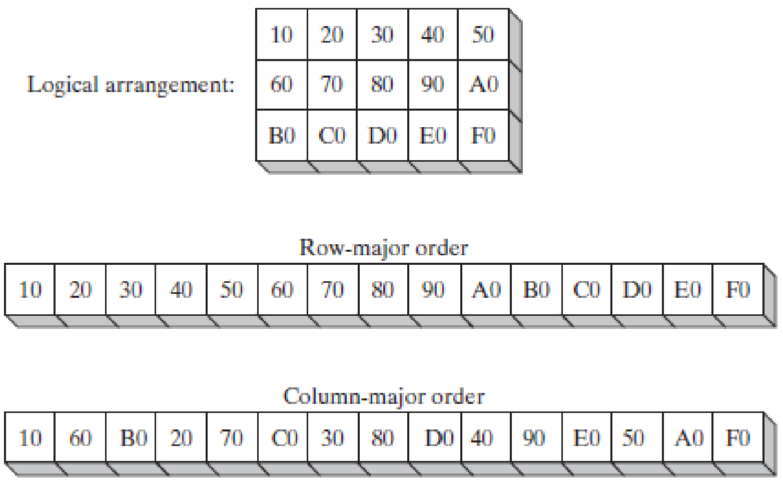


```
string_1 after trimming: abcde
string_1 in upper case: ABCDE
string1 and string2 are equal
Length of string_2 is 5
```

Two-Dimensional Arrays

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.



Base-Index Operands

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

When accessing a two-dimensional array in row-major order

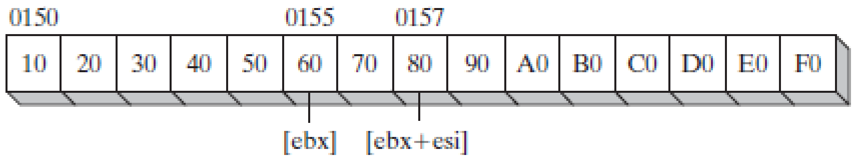
The row offset is held in the base register and the column offset is in the index register. The following table,
for example, has three rows and five columns:

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

- 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.
- 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 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
```

Suppose the array is located at offset 0150h. Then the effective address represented by EBX _
ESI is 0157h. Following figure shows how adding EBX and ESI produces the offset of the byte at tableB [1,2]. If the effective address points outside the program's data region, a runtime error occurs.



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
```

The scale factor used in this example (TYPE tableW) is equal to 2. Similarly, you must use a scale factor of 4 if the array contains doublewords:

```
tableD DWORD 10h, 20h, ...etc.
.code
mov eax,[ebx + esi*TYPE tableD]
```

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*]

Doubleword Array Example

The following two-dimensional array holds three rows of five doublewords:

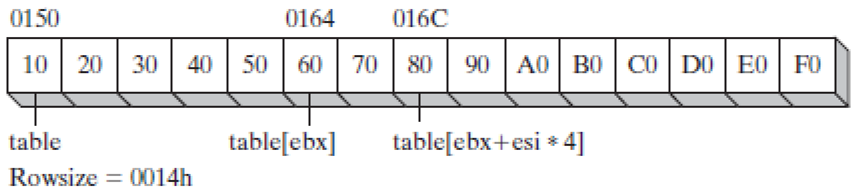
```
tableD DWORD 10h, 20h, 30h, 40h, 50h
Rowsize = ($ - tableD)
DWORD 60h, 70h, 80h, 90h, 0A0h
DWORD 0B0h, 0C0h, 0D0h, 0E0h, 0F0h
```

- Rowsize is equal to 20 (14h). Assuming that the coordinates are zero based, the entry at row 1, column 2 contains 80h.
- To access this entry, we set EBX to the row index and ESI to the column index:

```
mov ebx,Rowsize ; row index
mov esi,2 ; column index
mov eax,tableD[ebx + esi*TYPE tableD]
```

Suppose **tableD** begins at offset 0150h. Offsets are in hexadecimal.

Base-index-displacement example.


0150
0164
016C
10
20
30
40
50
60
70
80
90
A0
B0
C0
D0
E0
F0
table
table[ebx]
table[ebx+esi * 4]
Rowsize = 0014h