**Lab Session 11**



**String Handling Instructions**

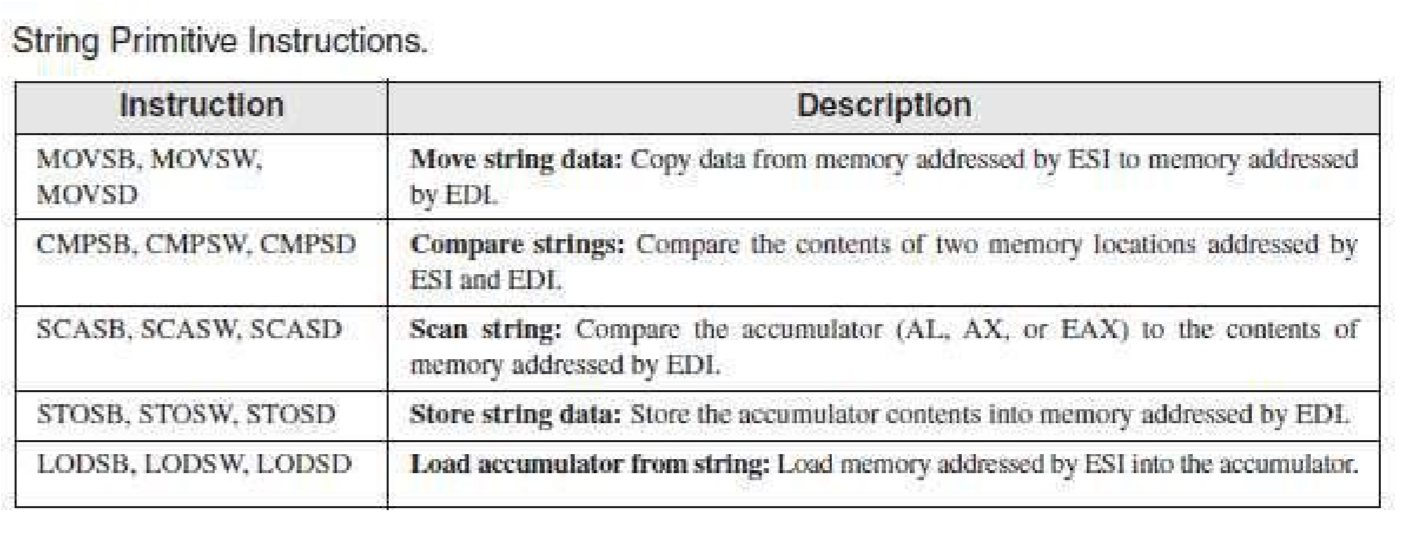


# Primitive String Instructions

* The x86 instruction set has five groups of instructions for processing arrays of bytes, words, and double words.

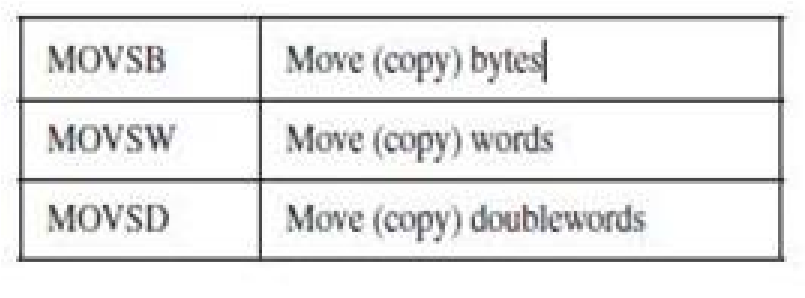
* Each instruction implicitly uses ESI, EDI, or both registers to address memory.

* 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.



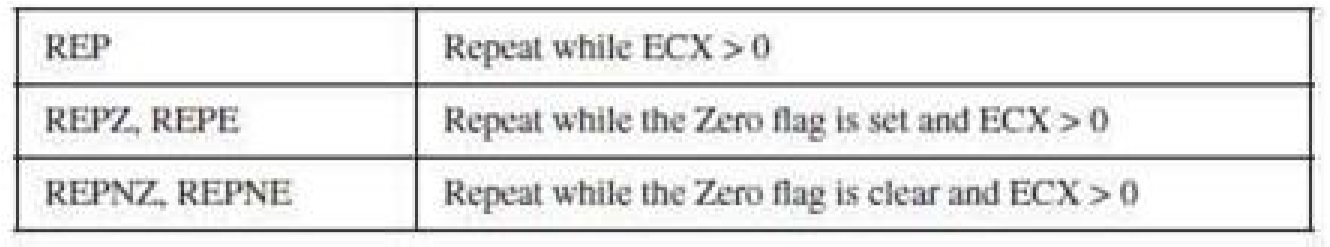
**1. MOVSB, MOVSW, and MOVSD**

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):



**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:



Example #1: 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 | ; changed String |

call writestring main ENDP

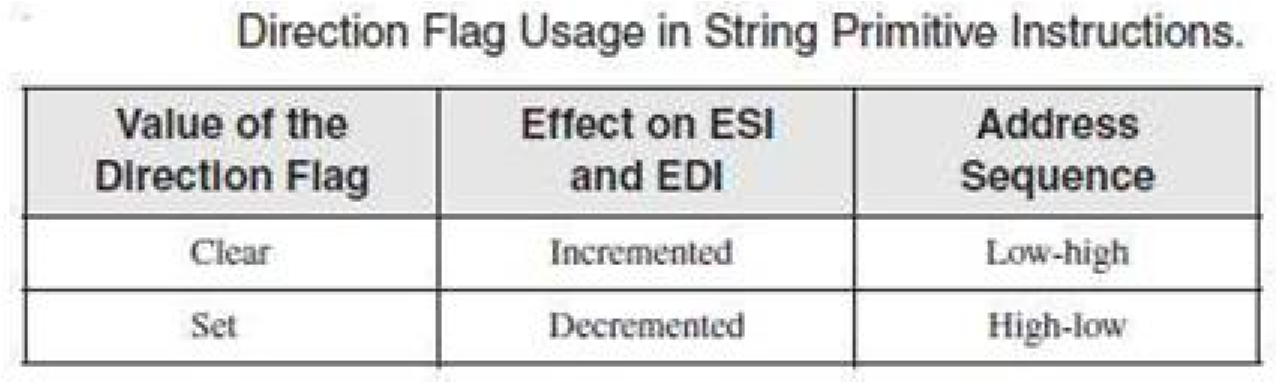
END main

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

**Direction Flag**

|  |  |
| --- | --- |
| **CLD** | **;clear Direction flag (forward direction)** |
| **STD** | **;set Direction flag (reverse direction)** |

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:



**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.

Example # 2: 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 #3: Scan for a Matching Character

INCLUDE Irvine32.inc

.data

alpha BYTE "ABCDEFGH",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 |

quit: exit main ENDP

END main

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:

.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 inc esi ; point to next byte

Example#4: 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  .code main PROC |  |  | ; test data |
| Cld |  |  | ; direction = forward |
| mov esi,OFFSET array |  |  | ; source index |
| mov edi,esi |  |  | ; destination index |
| mov ecx,LENGTHOF array  L1: |  |  | ; loop counter |
| Lodsd |  |  | ; load [ESI] into EAX |
| mul multiplier |  |  | ; multiply by a value |
| Stosd  loop L1 mov esi , OFFSET array mov ecx, LENGTHOF array mov ebx, TYPE array |  |  | ; store EAX into [EDI] |
| call dumpmem |  |  | ; updated array Display |

Exit main ENDP END main

# String Procedures

## 1. STR\_CPY

The Str\_copy procedure copies a null-terminated string from a source location to a target location.

**Syntax:** INVOKE Str\_copy, ADDR source, ADDR target

## 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

## 3. STR\_COMPARE

The Str\_compare procedure compares two strings. It affects the CF and ZF as shown in the following table.

**Syntax:** INVOKE Str\_compare, ADDR string1, ADDR string2

## 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

## 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

# String Library Demo Program

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 .ELSE | ; string 1 is less than... |
| mov edx,OFFSET msg3  .ENDIF call WriteString  call Crlf  ret  compare\_strings ENDP    print\_length PROC | ; string 2 is less than... |

; Display the length of string\_2.

mov edx,OFFSET msg4

call WriteString

INVOKE Str\_length, ADDR string\_2 call WriteDec call Crlf

ret

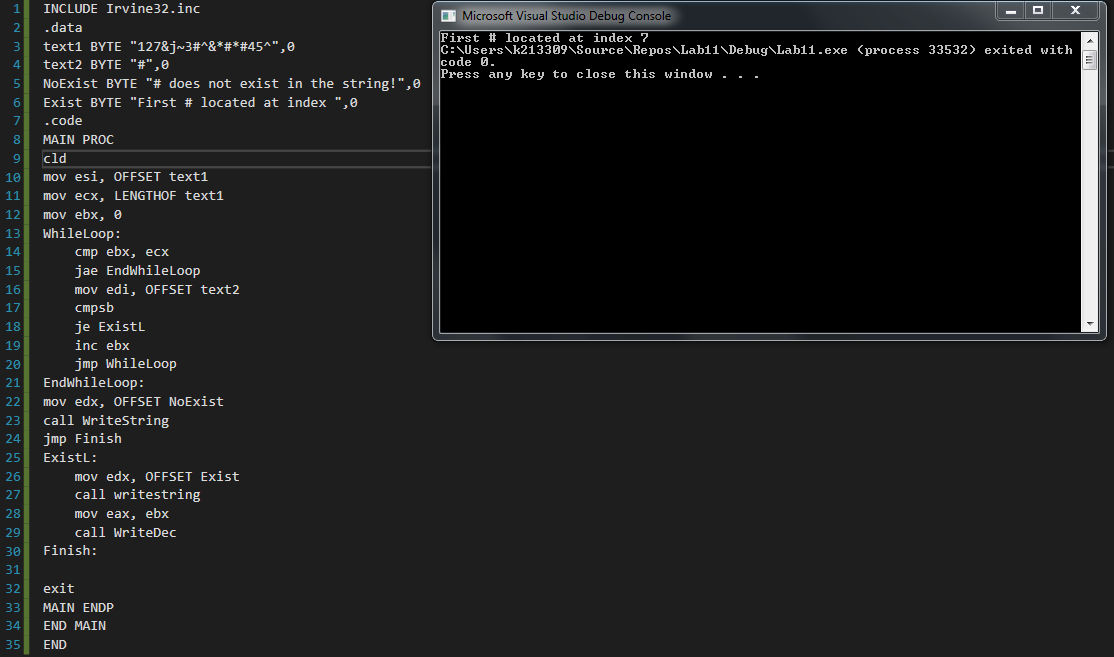
print\_length ENDP

END main

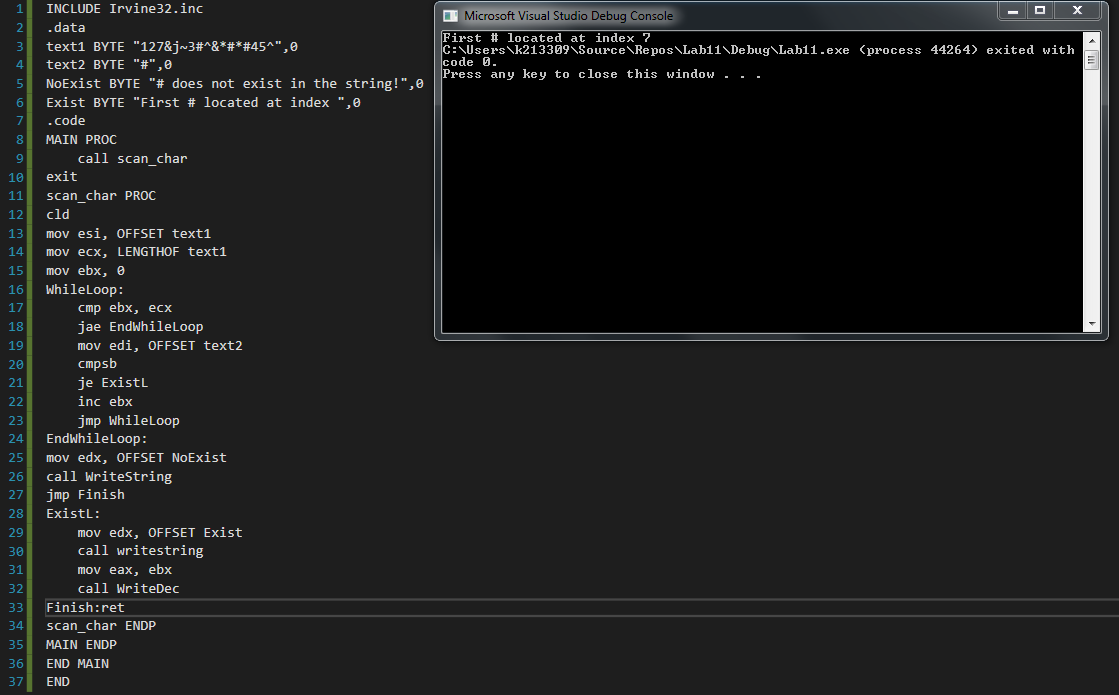
**Exercise**

1. **Write a program to find the index(location) of the first occurrence of the character ‘#’ in the given string.**

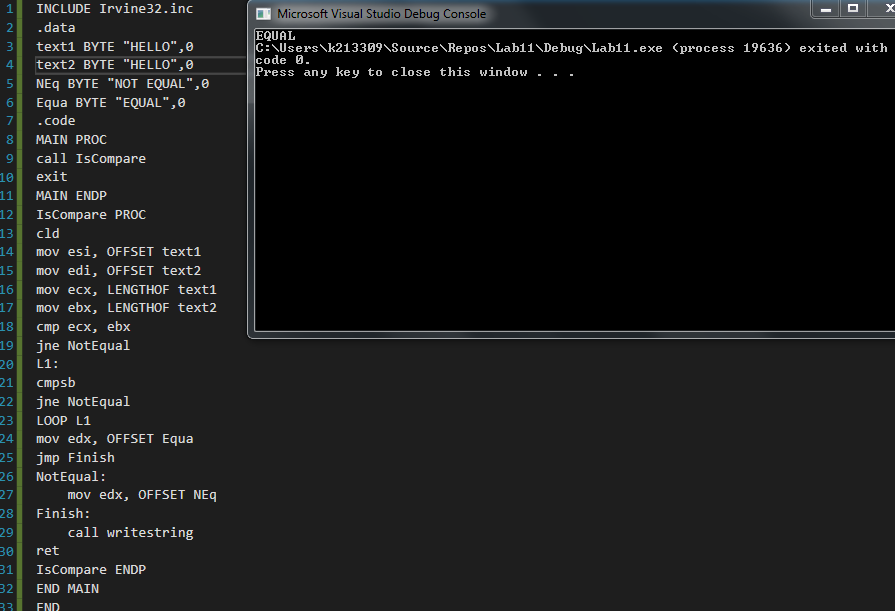
**Str1 BYTE ‘127&j~3#^&\*#\*#45^’,0**



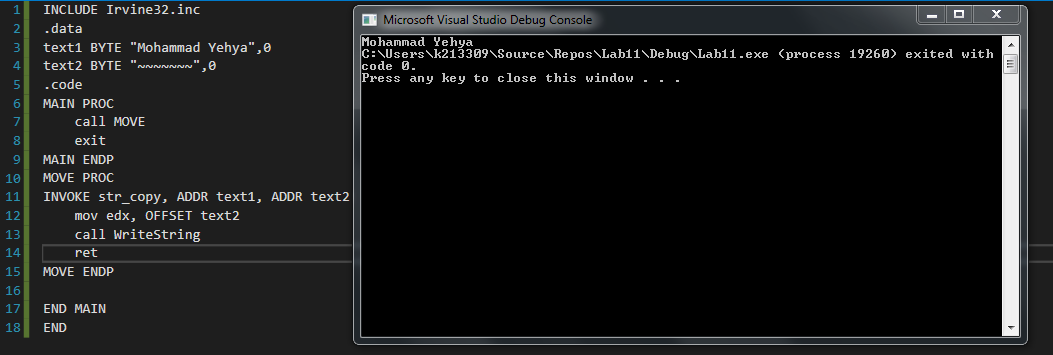
1. **Repeat the task 1 by creating a procedure named scan\_char. Call the procedure to find the index(location) of the first occurrence of the character ‘#’ in the given string.**



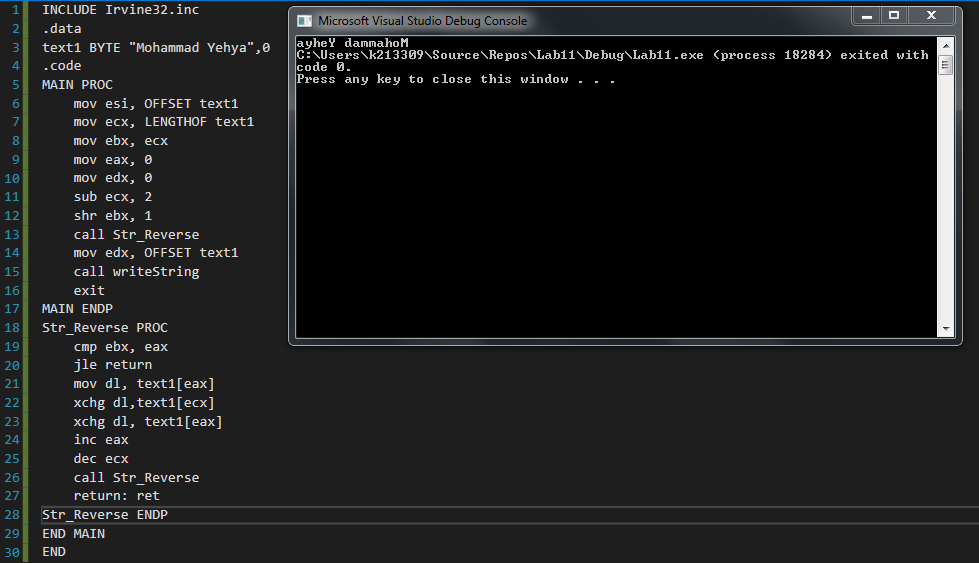
1. **Create IsCompare procedure to compare two strings.**



1. **Create *Move* procedure to perform move operation on two strings.**



1. **Create a Str\_Reverse procedure to reverse strings.**



1. **Create a procedure that Loads an array of integer by multiplying it with 3.** **Display updated Array.**

