

# Assembly Language for Intel-Based Computers, 4<sup>th</sup> Edition

Kip R. Irvine

## Chapter 9: Strings and Arrays

# Chapter Overview

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

# 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, and MOVSD (1 of 2)

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

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

## MOVSb, MOVSW, and MOVSD (2 of 2)

- 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

- 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
```

## Your turn . . .

- 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
```

```
.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, 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 compares bytes
  - CMPSW compares words
  - CMPSD compares doublewords
- Repeat prefix (REP) is often used

# Comparing a Pair of Doublewords

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

```
.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 Arrays

Use a REPE (repeat while equal) prefix to compare corresponding elements of two arrays.

```
.data
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 (1 of 3)

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.

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

Screen  
output:

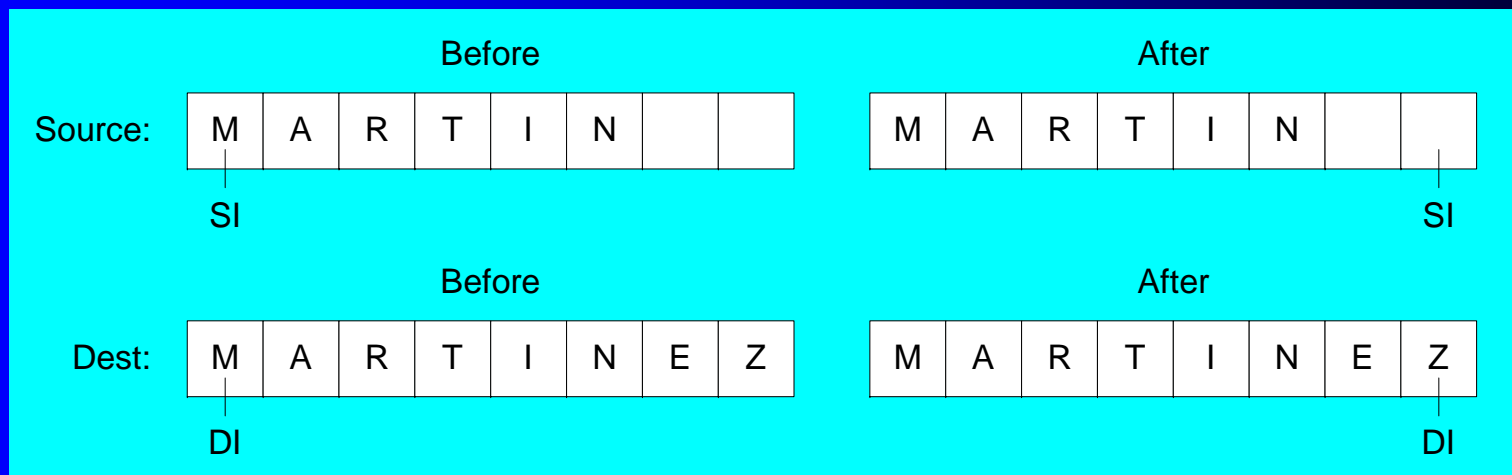
```
Source is smaller
```

## Example: Comparing Two Strings (2 of 3)

```
.code
main PROC
    cld                        ; direction = forward
    mov esi,OFFSET source
    mov edi,OFFSET dest
    mov cx,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 (3 of 3)

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



# 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.
- 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 Example

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

```
.data
alpha BYTE "ABCDEFGH",0
.code
mov edi,OFFSET alpha
mov al,'F'                ; search for 'F'
mov ecx,LENGTHOF alpha
cld
repne scasb               ; repeat while not equal
jnz quit
dec edi                  ; EDI points to 'F'
```

What is the purpose of the JNZ instruction?



# 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.
- 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, and LODSD

- The LODSB, LODSW, and LODSD instructions load a byte or word from memory at ESI into AL/AX/EAX, respectively.

```
.data
array 1,2,3,4,5,6,7,8,9
dest 9 DUP(?)
.code
    mov esi,OFFSET array
    mov edi,OFFSET dest
    mov ecx,LENGTHOF array
    cld
L1: lodsb
    or al,30h
    stosb
    loop L1
```

# Array Multiplication Example

Multiply each element of a doubleword array by a constant value.

```
.data
array DWORD 1,2,3,4,5,6,7,8,9,10
multiplier DWORD 10
.code
    cld                ; direction = up
    mov esi,OFFSET array ; source index
    mov edi,esi         ; destination index
    mov ecx,LENGTHOF array ; loop counter

L1: lodsd              ; copy [ESI] into EAX
    mul multiplier      ; multiply by a value
    stosd               ; store EAX at [EDI]
    loop L1
```

## Your turn . . .

- 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(?)
```

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

# 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

# Str\_compare Procedure

- Compares *string1* to *string2*, setting the Carry and Zero flags accordingly
- Prototype:

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

For example, if  $\text{string1} > \text{string2}$ ,  $\text{CF}=0$ ,  $\text{ZF}=0$

Or, if  $\text{string1} < \text{string2}$ ,  $\text{CF}=1$ ,  $\text{ZF}=0$

# Str\_compare 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: inc  esi                ; point to next  
    inc  edi  
    cmp  al,dl              ; chars equal?  
    je   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:

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

Example:

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



# Str\_length 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:

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

# Str\_copy 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:

```
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 Source Code

```
Str_trim PROC USES eax ecx edi,  
    pString:PTR BYTE,           ; points to string  
    char:BYTE                   ; char to remove  
    mov     edi,pString  
    INVOKE  Str_length,edi       ; returns length in EAX  
    cmp     eax,0               ; zero-length string?  
    je      L2                  ; yes: exit  
    mov     ecx,eax             ; no: counter = string length  
    dec     eax  
    add     edi,eax             ; EDI points to last char  
    mov     al,char             ; char to trim  
    std     ; direction = reverse  
    repe    scasb               ; skip past trim character  
    jne     L1                  ; removed first character?  
    dec     edi                 ; adjust EDI: ZF=1 && ECX=0  
L1: mov     BYTE PTR [edi+2],0  ; insert null byte  
L2: ret  
Str_trim ENDP
```

# Str\_ucase Procedure

- The Str\_ucase procedure converts a string to all uppercase characters. It returns no value.
- Prototype:

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

Example:

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

# Str\_ucase 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

- Base-Index Operands
- Base-Index Displacement

# 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.
- Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name. )

# Structure Application

A common application of base-index addressing has to do with addressing arrays of structures (Chapter 10). The following defines a structure named COORD containing X and Y screen coordinates:

```
COORD STRUCT
    X WORD ?           ; offset 00
    Y WORD ?           ; offset 02
COORD ENDS
```

Then we can define an array of COORD objects:

```
.data
setOfCoordinates COORD 10 DUP(<>)
```

# Structure Application

The following code loops through the array and displays each Y-coordinate:

```
    mov     ebx,OFFSET setOfCoordinates
    mov     esi,2                ; offset of Y value
    mov     eax,0
L1:  mov     ax,[ebx+esi]
     call   WriteDec
     add    ebx,SIZEOF COORD
     loop   L1
```

# Base-Index-Displacement Operand

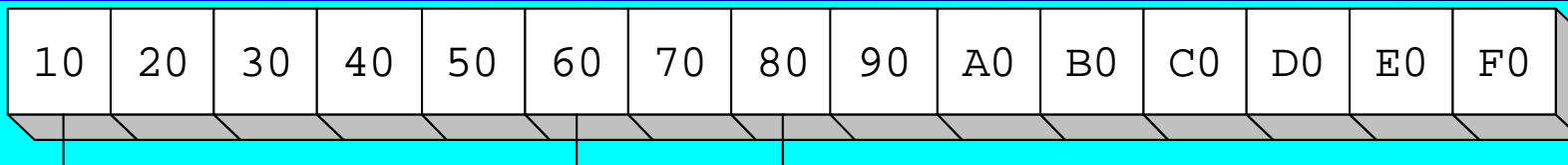
- A **base-index-displacement** operand adds base and index registers to a constant, producing an **effective address**. Any two 32-bit general-purpose registers may be used.
- Common formats:

$[ \textit{base} + \textit{index} + \textit{displacement} ]$   
 $\textit{displacement} [ \textit{base} + \textit{index} ]$

# Two-Dimensional Table Example

Imagine a table with three rows and five columns.

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

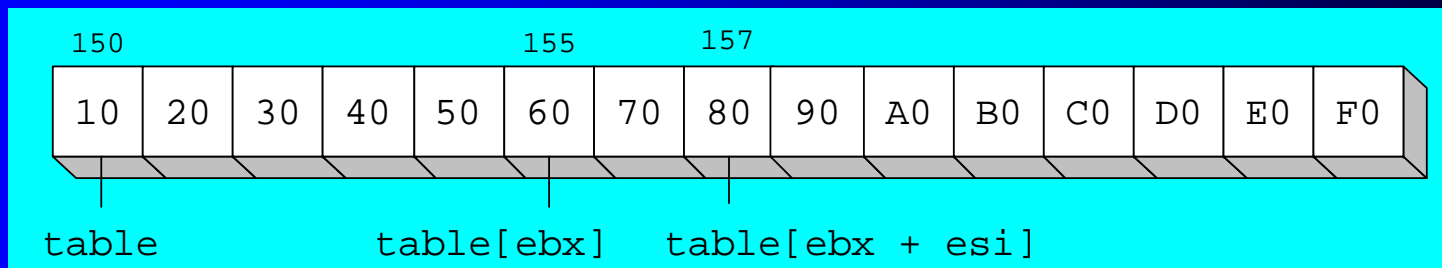


# Two-Dimensional Table Example

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]
```



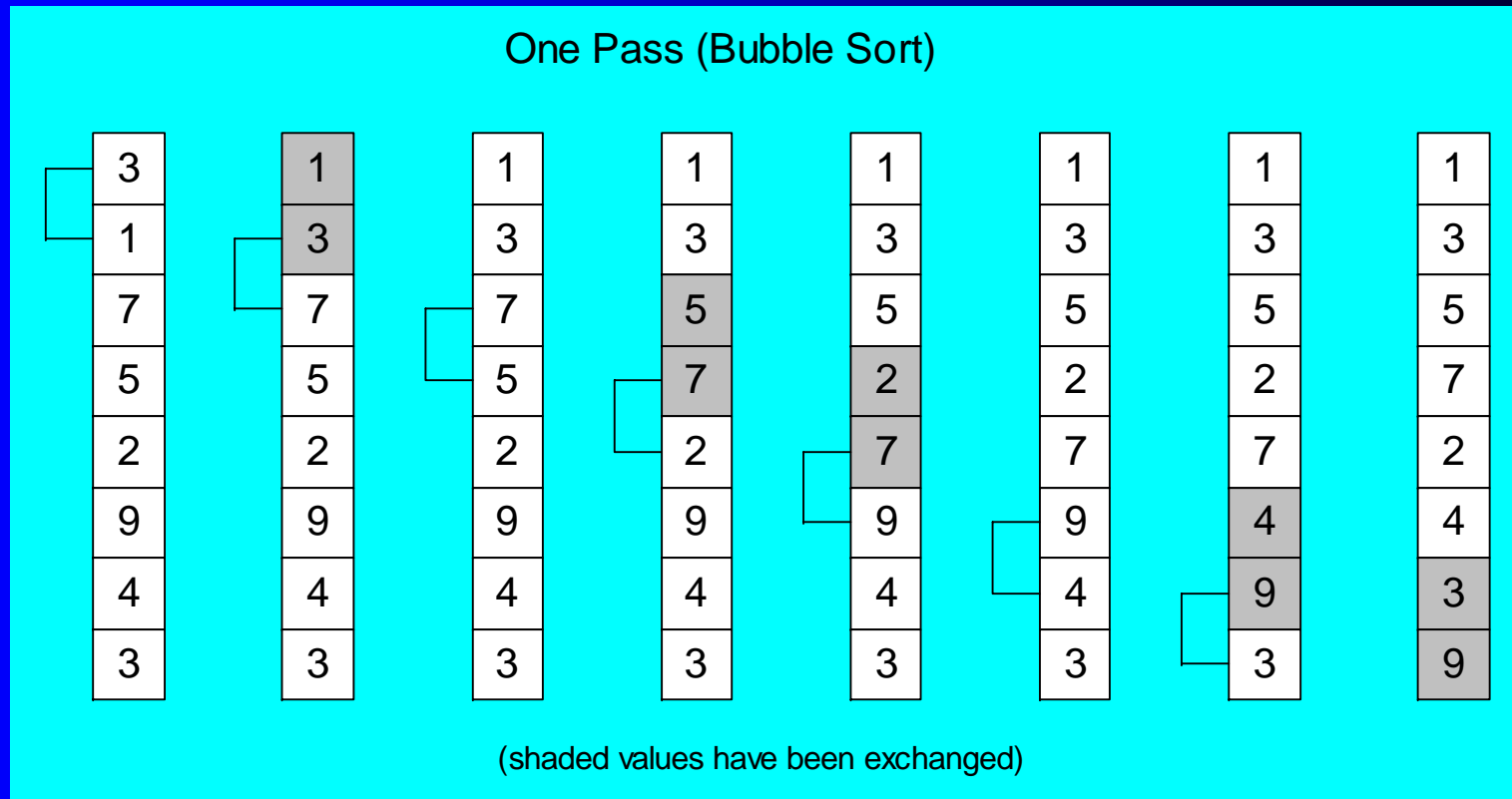
# Searching and Sorting Integer Arrays

- Bubble Sort
  - A simple sorting algorithm that works well for small arrays
- Binary Search
  - A simple searching algorithm that works well for large arrays of values that have been placed in either ascending or descending order



# Bubble Sort

Each pair of adjacent values is compared, and exchanged if the values are not ordered correctly:



# Bubble Sort Pseudocode

N = array size, cx1 = outer loop counter, cx2 = inner loop counter:

```
cx1 = N - 1
while( cx1 > 0 )
{
    esi = addr(array)
    cx2 = cx1
    while( cx2 > 0 )
    {
        if( array[esi] < array[esi+4] )
            exchange( array[esi], array[esi+4] )
        add esi, 4
        dec cx2
    }
    dec cx1
}
```

# Bubble Sort Implementation

```
BubbleSort PROC USES eax ecx esi,  
    pArray:PTR DWORD,Count:DWORD  
    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  
    jge    L3                ; if [esi] <= [edi], skip  
    xchg   eax,[esi+4]        ; else 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
```

# Binary Search

- Searching algorithm, well-suited to large ordered data sets
- Divide and conquer strategy
- Each "guess" divides the list in half
- Classified as an  $O(\log n)$  algorithm:
  - As the number of array elements increases by a factor of  $n$ , the average search time increases by a factor of  $\log n$ .

# Binary Search Estimates

Array Size (n)	Maximum Number of Comparisons: $(\log_2 n) + 1$
64	7
1,024	11
65,536	17
1,048,576	21
4,294,967,296	33

# Binary Search Pseudocode

```
int BinSearch(int values[],
              const int searchVal, int count)
{
    int first = 0;
    int last = count - 1;
    while( first <= last )
    {
        int mid = (last + first) / 2;
        if( values[mid] < searchVal )
            first = mid + 1;
        else if( values[mid] > searchVal )
            last = mid - 1;
        else
            return mid;          // success
    }
    return -1;                  // not found
}
```

# Binary Search Implementation (1 of 3)


```
BinarySearch PROC uses ebx edx esi edi,  
    pArray:PTR DWORD,      ; pointer to array  
    Count:DWORD,           ; array size  
    searchVal:DWORD        ; search value  
  
LOCAL first:DWORD,        ; first position  
    last:DWORD,           ; last position  
    mid:DWORD             ; midpoint  
    mov     first,0        ; first = 0  
    mov     eax,Count      ; last = (count - 1)  
    dec     eax  
    mov     last,eax  
    mov     edi,searchVal  ; EDI = searchVal  
    mov     ebx,pArray     ; EBX points to the array  
L1:                               ; while first <= last  
    mov     eax,first  
    cmp     eax,last  
    jg      L5             ; exit search
```

## Binary Search Implementation (2 of 3)

```
; mid = (last + first) / 2
    mov    eax,last
    add    eax,first
    shr    eax,1
    mov    mid,eax

; EDX = values[mid]
    mov    esi,mid
    shl    esi,2           ; scale mid value by 4
    mov    edx,[ebx+esi]   ; EDX = values[mid]

; if ( EDX < searchval(EDI) )
;   first = mid + 1;
    cmp    edx,edi
    jge    L2
    mov    eax,mid         ; first = mid + 1
    inc    eax
    mov    first,eax
    jmp    L4              ; continue the loop
```





## Binary Search Implementation (3 of 3)

```
; else if( EDX > searchVal(EDI) )
;   last = mid - 1;
L2: cmp     edx,edi                ; (could be removed)
     jle     L3
     mov     eax,mid              ; last = mid - 1
     dec     eax
     mov     last,eax
     jmp     L4                  ; continue the loop

; else return mid
L3: mov     eax,mid              ; value found
     jmp     L9                  ; return (mid)

L4: jmp     L1                  ; continue the loop
L5: mov     eax,-1              ; search failed
L9: ret
BinarySearch ENDP
```