Contents

STRING OPERATIONS

String Primitive Instructions(x86)

String primitive instructions in the x86 architecture are specialized instructions designed to efficiently process **blocks (strings) of data** in memory.
They operate on **contiguous memory locations** and are commonly used for:

- Moving data
- Comparing memory blocks
- Scanning memory
- Loading from memory into registers
- Storing register values into memory

These instructions work closely with **index registers**, **segment registers**, and the **Direction Flag (DF)** to automatically update memory addresses after each operation.

Key Characteristics (Applies to All String Instructions)

- Operate on **memory, not high-level strings**
- Automatically increment or decrement pointers based on **DF**
- Often paired with **REP prefixes** (REP, REPE, REPNE) for looping
- Use **implicit registers** (no explicit operands written in instruction)

Common Registers Used

Table 9-1 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.

Groups of String Primitive Instructions

1. Move String Data — MOVS

Instructions:

- MOVSB – move byte
- MOVSW – move word
- MOVSD – move doubleword

Purpose

Copies data from a source memory location to a destination memory location.

Behavior

- Source address: DS:ESI
- Destination address: ES:EDI
- After execution:
  - ESI and EDI are automatically updated
  - Incremented if DF = 0
  - Decremented if DF = 1

Typical Use

- Copying arrays or memory buffers
- Used with REP to move blocks

2. Compare Strings — CMPS

Instructions:

- CMPSB
- CMPSW
- CMPSD

Purpose

Compares two memory locations element-by-element.

Behavior

- Compares DS:ESI with ES:EDI
- Internally subtracts: (Destination) – (Source)

- Sets **flags** (ZF, SF, CF, etc.)
- Does **not** store the result

Common Usage

- Used with REPE / REPNE to find mismatches
- Often used for string comparison logic

3. Scan String — SCAS

Instructions:

- SCASB
- SCASW
- SCASD

Purpose

Searches memory for a specific value.

Behavior

- Compares accumulator (AL/AX/EAX) with: ES:EDI
- Updates flags based on comparison
- Automatically updates EDI

Typical Use

- Finding a specific byte/word/dword in memory
- Used with REPNE to scan until a match is found

4. Store String Data — STOS

Instructions:

- STOSB
- STOSW
- STOSD

Purpose

Stores the accumulator value into memory.

Behavior

- Writes:
    - AL/AX/EAX → ES:EDI
- Updates EDI automatically

Common Usage

- Initializing memory blocks
- Clearing buffers (e.g., filling with zeros using REP STOSB)

5. Load Accumulator from String — LODS

Instructions:

- LODSB
- LODSW
- LODSD

Purpose

Loads data from memory into the accumulator.

Behavior

- Loads: DS:ESI → AL/AX/EAX
- Updates ESI automatically

Typical Use

- Sequentially reading data from memory
- Used in parsing routines

Direction Flag (DF) Impact

Use CLD to clear DF (forward processing)
Use STD to set DF (backward processing)

Common Pitfalls

- Forgetting to clear/set **Direction Flag**
- Assuming operands are explicit (they are **implicit**)
- Mixing up DS and ES segments
- Using string instructions without REP when looping is required
- Forgetting that flags are modified (especially with CMPS and SCAS)

Suggested Visualizations (For Study)

- **Memory flow diagram**: DS:ESI ---> ES:EDI
- **Register update flow** showing ESI/EDI movement based on DF
- **REP loop diagram** showing ECX countdown

Short note:

- String primitive instructions process **contiguous memory**
- They rely heavily on **implicit registers**
- Pointer movement depends on **Direction Flag**
- Best used with **REP prefixes** for efficiency
- Fundamental for low-level memory manipulation in x86

Using a Repeat Prefix

A string instruction normally works on **one memory value** (or one pair of values) at a time.

You can use a **repeat prefix** to make the instruction run **many times**.

The instruction repeats **until a specific condition is met**.

The most common repeat prefix is **REP**.

REP makes the instruction repeat **while the ECX register is greater than zero**.

Each time the instruction runs, **ECX is decreased by 1**.

When **ECX becomes zero**, the instruction **stops repeating**.

Example: using **REP** can copy **10 bytes** from the string1 buffer to the string2 buffer.

```
001 cld ; clear Direction flag
002 mov esi, OFFSET string1 ; ESI points to source
003 mov edi, OFFSET string2 ; EDI points to target
004 mov ecx, 10 ; set counter to 10
005 rep movsb ; move 10 bytes
```

String instructions use the **ESI** and **EDI** registers to move through memory.

Whether these registers **increase or decrease** depends on the **Direction Flag (DF)**.

The Direction Flag can be changed **manually** using special instructions.

CLD clears the Direction Flag.

When the Direction Flag is cleared, **ESI and EDI increment** (move forward in memory).

STD sets the Direction Flag.

When the Direction Flag is set, **ESI and EDI decrement** (move backward in memory).

Table 9-2 Direction Flag Usage in String Primitive Instructions.

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

When the **Direction Flag is clear**, **ESI and EDI increment** after each operation.

This means the string operation moves from a **lower memory address to a higher memory address**.

When the **Direction Flag is set**, **ESI and EDI decrement** after each operation.

This means the string operation moves from a **higher memory address to a lower memory address**.

The Direction Flag must be set **correctly before** using a string instruction.

If it is not set correctly, **ESI and EDI may move in the wrong direction**.

To copy a string normally from one buffer to another, the Direction Flag should be **cleared first**.

If the Direction Flag is set, the string would be copied **in reverse order**.

The following example shows how to use the **MOVSB** instruction to copy a string from one buffer to another.

```
009 ; Copy string1 to string2
010 cld ; clear Direction flag
011 mov esi, OFFSET string1 ; ESI points to source
012 mov edi, OFFSET string2 ; EDI points to target
013 mov ecx, 10 ; set counter to 10
014 rep movsb ; move 10 bytes
```

MOVSB, MOVSW, and MOVSD

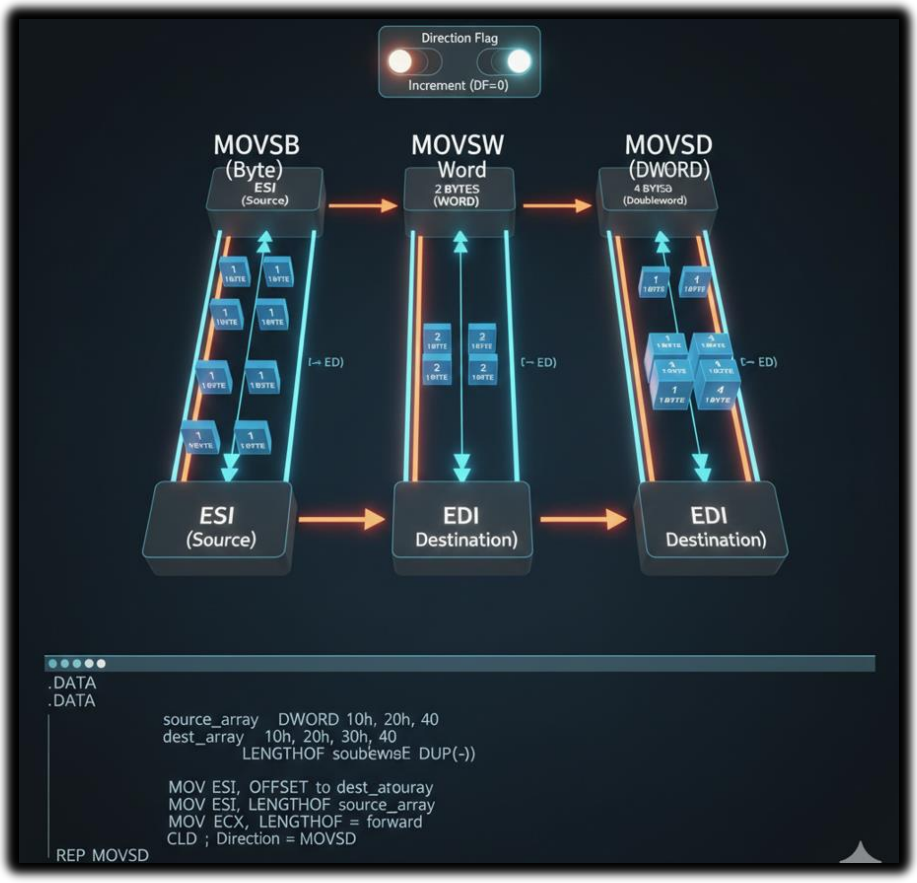
MOVSB, MOVSW, and MOVSD are used to copy data from one memory location to another.

They differ by the size of data they copy.

MOVSB copies bytes.

MOVSW copies words.

MOVSD copies doublewords.



All three instructions use ESI as the source address.

All three instructions use EDI as the destination address.

The Direction Flag controls whether ESI and EDI increment or decrement after each copy.

The following code shows how to use MOVSD to copy a doubleword array from one buffer to another.

```
026 ; Copy source to target
027 cld ; clear Direction flag
028 mov ecx, LENGTHOF source ; set REP counter
029 mov esi, OFFSET source ; ESI points to source
030 mov edi, OFFSET target ; EDI points to target
031 rep movsd ; copy doublewords
```

CLD clears the Direction Flag. When the Direction Flag is clear, ESI and EDI increment after each operation.

REP makes the MOVSD instruction repeat while ECX is greater than zero.

ECX usually holds the number of elements to copy.

MOVSD copies 4 bytes (one doubleword) each time it runs.

After the code finishes, ESI and EDI point 4 bytes past the end of the arrays.

This happens because the registers are incremented after the final copy.

LENGTHOF is a macro that returns the number of elements in an array.

LENGTHOF is often used to set ECX before using REP.

ESI and EDI increment or decrement because string instructions automatically move through memory.

The Direction Flag controls which direction they move in memory.

CMPSB, CMPSW, and CMPSD

CMPSB, CMPSW, and CMPSD are used to compare two memory values.

They differ by the size of data they compare.

CMPSB compares bytes.

CMPSW compares words.

CMPSD compares doublewords.

All three instructions use ESI to point to the first operand.

All three instructions use EDI to point to the second operand.

After each comparison, ESI and EDI move to the next or previous memory location.

The Direction Flag controls whether ESI and EDI increment or decrement.

Example: Comparing Doublewords:

```
035 ; Compare source and target
036 mov esi, OFFSET source
037 mov edi, OFFSET target
038 cmpsd
039 ; compare doublewords
```

CMPSD compares two doubleword values.

If the two values are equal, the Zero Flag (ZF) is set.

If the source doubleword is greater than the target doubleword, the Carry Flag (CF) is set.

If the source is not greater than the target, the Carry Flag is cleared.

To compare multiple doublewords, you can use a repeat prefix with CMPSD.

```
042 mov esi, OFFSET source
043 mov edi, OFFSET target
044 cld ; clear Direction flag
045 mov ecx, LENGTHOF source ; repetition counter
046 repe cmpsd ; repeat while equal
```

REPE (Repeat While Equal) is used with compare instructions like CMPSD.

It repeats the comparison while the values are equal.

REPE stops when ECX becomes zero or when a mismatch is found.

ECX usually contains the number of elements to compare.

LENGTHOF is a macro that returns the number of elements in an array.

LENGTHOF is often used to initialize ECX before using REPE.

The Direction Flag controls whether ESI and EDI move forward or backward in memory.

The Direction Flag must be set correctly before using string compare instructions.

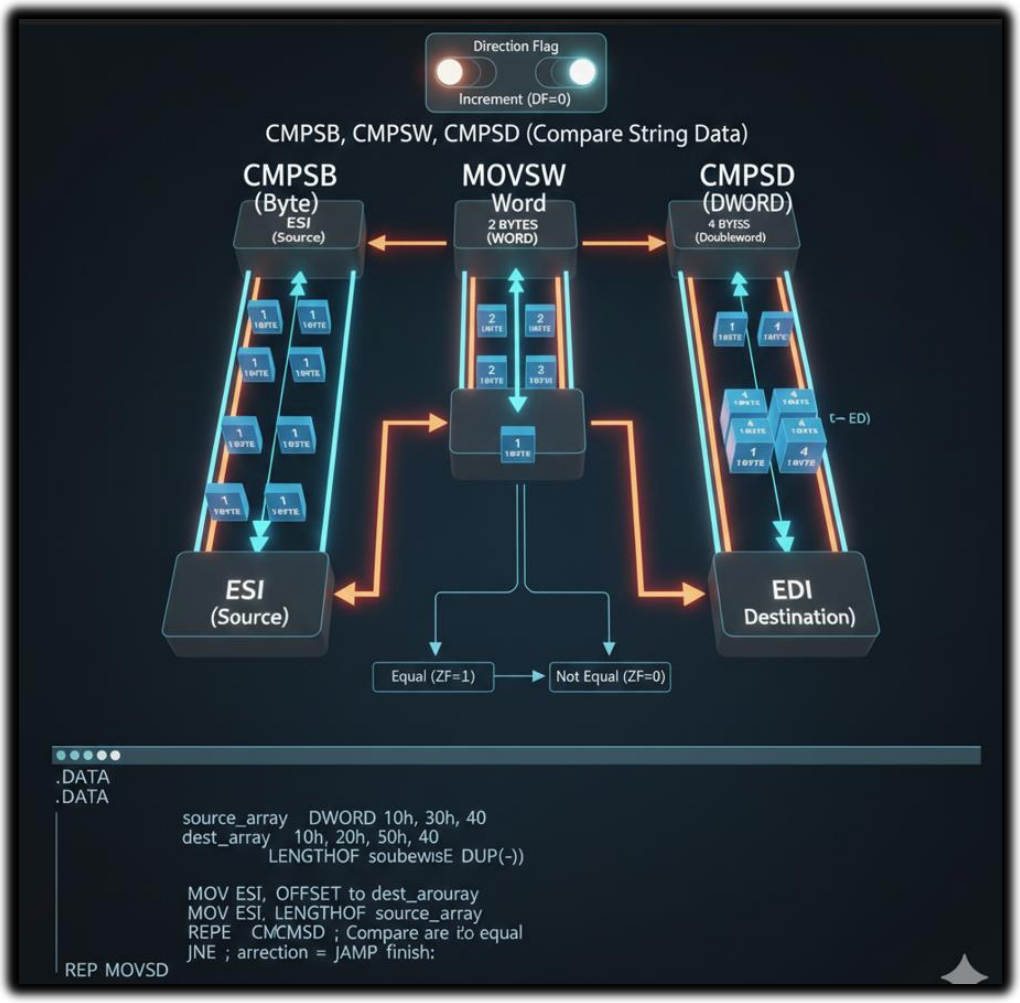
CMPSB, CMPSW, and CMPSD can be used for string-related operations.

Examples include searching for a character or comparing two strings.

The following example shows how to use CMPSB to search for the letter 'A' in a string.

```
050 mov esi, OFFSET string
051 mov edi, OFFSET 'A'
052 repe cmpsb ; repeat while equal
053 jne not_found ; jump if not equal
```

- The string is compared **one byte at a time** using **CMPSB**.
- If the letter **'A' is found**, the **Zero Flag is set**.
- When the Zero Flag is set, the **JNE instruction does not jump**.
- If the letter **'A' is not found**, the **Zero Flag is clear**.
- When the Zero Flag is clear, the **JNE instruction jumps** to the not_found label.
- **CMPSB**, **CMPSW**, and **CMPSD** are powerful instructions for **comparing memory operands**.



SCASB, SCASW, and SCASD

SCASB, SCASW, and SCASD compare a register value to memory:

SCASB: compares AL to a byte at EDI

SCASW: compares AX to a word at EDI

SCASD: compares EAX to a doubleword at EDI

These instructions are useful for searching for a single value in a string or array.

REPE (or REPZ) repeats the scan while ECX > 0 and the register matches memory.

REPNE (or REPNZ) repeats the scan while ECX > 0 and the register does not match memory.

ECX usually holds the number of elements to scan.

Example: SCASB can be used to search for the letter 'F' in the string alpha.

```
059 .data
060     alpha BYTE "ABCDEFGH",0
061 .code
062     mov edi, OFFSET alpha ; EDI points to the string
063     mov al, 'F' ; search for the letter F
064     mov ecx, LENGTHOF alpha ; set the search count
065     cld ; direction = forward
066     repne scasb ; repeat while not equal
067
068     ; Test if the loop stopped because ECX = 0 and the character in AL was not found
069     jnz quit
070
071     ; Found: back up EDI
072     dec edi
073
074     ; Otherwise, the letter F was found
```

REPNE makes the SCASB instruction repeat while the Zero Flag is clear and ECX > 0.

The Zero Flag is cleared when AL does not match the byte at EDI.

If AL matches the byte at EDI, the Zero Flag is set.

After the loop, the JNZ (jump if not zero) instruction is used to check why the loop stopped.

If the loop stopped because ECX = 0 and the character was not found, the Zero Flag is clear, and JNZ jumps to the quit label.

If the character 'F' was found, the DEC EDI instruction is used to move EDI back one position.

This ensures EDI points to the found character instead of the next memory location.

SCASB, SCASW, and SCASD are powerful for searching for a single value in a string or array.

STOSB, STOSW, and STOSD

STOSB, STOSW, and STOSD store a register value into memory at EDI.

STOSB stores AL as a byte.

STOSW stores AX as a word.

STOSD stores EAX as a doubleword.

EDI is incremented or decremented after each store based on the Direction Flag.

When combined with REP, these instructions can fill a string or array with a single value.

Example: STOSB can initialize each byte in the string1 array to 0xFF.

```
; Initialize each byte in string1 to 0xFFh

.data
    Count = 100
    string1 BYTE Count DUP(?)
.code
    mov al, 0xFFh ; 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
```

REP makes the STOSB instruction repeat while ECX > 0.

CLD clears the Direction Flag.

When the Direction Flag is clear, EDI increments after each store.

If the Direction Flag is set, EDI decrements, and the array would be filled in reverse order.

After execution, each byte in the string1 array will be set to 0xFF.

STOSB, STOSW, and STOSD are powerful for filling all elements of a string or array with a single value.

```
mov al, 0xFFh
mov edi, OFFSET string1
mov ecx, LENGTHOF string1
cld
rep stosb
```

Move the value 0xFF into the AL register.

Move the value of EDI into ECX (to set the number of bytes to fill).

Clear the Direction Flag with CLD.

Repeat the STOSB instruction while ECX > 0.

After execution, each byte in the string1 array is set to 0xFF.

LODSB, LODSW, and LODSD

LODSB, LODSW, and LODSD load data from memory at ESI into a register:

LODSB → AL (byte)

LODSW → AX (word)

LODSD → EAX (doubleword)

ESI is incremented or decremented based on the Direction Flag.

REP is rarely used with LODS because each load overwrites the previous value in the accumulator.

LODS is typically used to load a single value from memory.

Example: LODSB can load a single byte from memory into AL.

```
101 ; Load a single byte from memory into the AL register
102 lod sb
```

LODSB/LODSW/LODSD load a value from memory at ESI into the accumulator (AL/AX/EAX).

After loading, ESI is incremented or decremented based on the Direction Flag.

Example: LODSB loads a byte from memory into AL, then ESI is incremented by 1.

Array Multiplication Example:

- LODSD loads a doubleword from an array into EAX.
- Multiply the value in EAX by a constant.
- STOSD stores the result back into memory at EDI.
- Repeat for each element of the array.

```
; Array Multiplication Example
.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
```

CLD clears the **Direction Flag**.

This ensures **ESI and EDI increment** after each operation.

ESI is set to the **offset of the source array**.

EDI is set to the **offset of the destination array**.

ECX is set to the **length of the array** and used as a **loop counter**.

The loop repeats **until ECX = 0**.

Each iteration of the loop:

- **LODSD** loads a doubleword from the source array (**ESI**) into **EAX**.
- **EAX** is multiplied by the value of the **multiplier variable**.
- **STOSD** stores the result into the destination array (**EDI**).

Clearing the Direction Flag ensures that **LODSD and STOSD access the next element** on each iteration.

Using **ECX as a loop counter** ensures the loop repeats **exactly as many times as the array length**.

LODSB, LODSW, and LODSD are powerful for **loading memory data into the accumulator**.

STRING PROCEDURES IN IRVINE32

The **Irvine32 library** provides procedures for working with **null-terminated strings**.

These procedures are similar to **C standard library functions**.

Str_copy

Str_copy copies a **source string** to a **target string**.

It takes **two arguments**:

1. Pointer to the **source string**
2. Pointer to the **target string**

Returns a **pointer to the target string**.

```
140 ; Copy a source string to a target string.
141 Str_copy PROTO,
142 source:PTR BYTE,
143 target:PTR BYTE
```

The following code shows how to use the **Str_copy procedure** to copy the string "Hello, world!" to the string "buffer":

```
145 mov eax, OFFSET "Hello, world!"
146 mov ebx, OFFSET buffer
147 call Str_copy
148 ; buffer now contains the string "Hello, world!"
```

Str_length

Str_length returns the **length of a string** (does **not** count the null byte).

It takes **one argument**: a **pointer to the string**.

The length is returned in the **EAX register**.

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

The following code shows how to use the Str_length procedure to get the length of the string "Hello, world!" and store it in the EAX register:

```
157 mov eax, OFFSET "Hello, world!"
158 call Str_length
159 ; EAX now contains the value 12, which is the length of the string "Hello, world!"
```

Str_compare

Str_compare compares **two strings**.

It sets the **Zero Flag (ZF)** and **Carry Flag (CF)** like the **CMP instruction**.

Takes **two arguments**:

1. Pointer to the **first string**
2. Pointer to the **second string**

```
162 ; Compare string1 to string2. Set the Zero and
163 ; Carry flags in the same way as the CMP instruction.
164 Str_compare PROTO,
165 string1:PTR BYTE,
166 string2:PTR BYTE
```

The following code shows how to use the Str_compare procedure to compare the strings "Hello, world!" and "Hello, world!":

```
170 mov eax, OFFSET "Hello, world!"
171 mov ebx, OFFSET "Hello, world!"
172 call Str_compare
173 ; The Zero flag will be set, indicating that the two strings are equal.
```

Str_trim

Str_trim removes a **trailing character** from a string.

Takes **two arguments**:

1. Pointer to the **string**
2. The **character to trim**

```
180 ; Trim a given trailing character from a string.
181 ; The second argument is the character to trim.
182 Str_trim PROTO,
183 pString:PTR BYTE,
184 char:BYTE
```

The following code shows how to use the Str_trim procedure to trim the trailing newline character from the string "Hello, world!\n":

```
185 mov eax, OFFSET "Hello, world!\n"
186 mov ebx, AL ; '\n' character
187 call Str_trim
188 ; The string "Hello, world!" is now stored in the memory location pointed to by EAX.
```

Str_ucase

- **Str_ucase** converts a string to **upper case**.
- Takes **one argument**: a **pointer to the string**.

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

The following code shows how to use the Str_ucase procedure to convert the string "hello, world!" to upper case:

```
195 mov eax, OFFSET "hello, world!"
196 call Str_ucase
197 ; The string "HELLO, WORLD!" is now stored in the memory location pointed to by EAX.
```

STR_COMPARE PROCEDURE

The Str_compare procedure compares two strings and sets the Carry and Zero flags in the same way as the CMP instruction. It takes two arguments: pointers to the two strings.

The Str_compare procedure works by comparing byte by byte of the two strings.

If a byte is found that is not equal in both strings, the Str_compare procedure sets the Carry flag and exits the loop.

If the end of both strings is reached, the Str_compare procedure sets the Zero flag and exits the loop.

The following is a more detailed explanation of the Str_compare procedure:

```
220 Str_compare PROC USES eax edx esi edi,
221     string1:PTR BYTE,
222     string2:PTR BYTE
223     ; Initialize esi and edi to point to the beginning of the strings.
224     mov esi, string1
225     mov edi, string2
226     ; Start of the loop to compare characters in the strings.
227 L1:
228     ; Load the characters from string1 and string2 into AL and DL.
229     mov al, [esi]
230     mov dl, [edi]
231     ; Check if we have reached the end of string1 (null terminator).
232     cmp al, 0
233     jne L2 ; If not, continue comparing.
234     ; If we have reached the end of string1, check if we have also reached the end of string2.
235     cmp dl, 0
236     jne L2 ; If not, continue comparing.
237     ; If we reach this point, both strings are equal, and we exit with ZF = 1.
238     jmp L3
239 L2:
240     ; Increment esi and edi to move to the next characters in the strings.
241     inc esi
242     inc edi
243     ; Compare the characters in AL and DL. If they are equal, continue the loop.
244     cmp al, dl
245     je L1 ; Characters are equal; continue comparing.
246
247     ; If characters are not equal, exit the loop with flags set.
248 L3:
249     ret
250 Str_compare ENDP
```

Line 1: The Str_compare procedure pushes the EAX, EDX, ESI, and EDI registers onto the stack. This is necessary because the procedure uses these registers.

Line 2: The Str_compare procedure moves the pointers to the two strings into the ESI and EDI registers, respectively.

Line 3: The Str_compare procedure enters a loop. On each iteration of the loop, the following steps are performed:

- The Str_compare procedure compares the bytes at the memory locations pointed to by the ESI and EDI registers.
- If the bytes are equal, the Str_compare procedure increments the ESI and EDI registers and continues to the next iteration of the loop.
- If the bytes are not equal, the Str_compare procedure sets the Carry flag and exits the loop.

Line 10: The Str_compare procedure checks if the end of the string pointed to by the ESI register has been reached. If the end of the string has been reached, the Str_compare procedure checks if the end of the string pointed to by the EDI register has also been reached.

Line 12: If the end of both strings has been reached, the Str_compare procedure sets the Zero flag and exits the loop.

Line 14: The Str_compare procedure increments the ESI and EDI registers so that they point to the next byte in the respective strings.

Line 16: The Str_compare procedure compares the bytes at the memory locations pointed to by the ESI and EDI registers. If the bytes are equal, the Str_compare procedure continues to the next iteration of the loop. If the bytes are not equal, the Str_compare procedure sets the Carry flag and exits the loop.

Line 20: The Str_compare procedure exits the loop and returns to the caller.

The following is a table of the flags affected by the Str_compare procedure:

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

This code defines the Str_compare procedure, which compares two strings character by character. It uses esi and edi to traverse the strings and compares characters while checking for the end of the strings.

The procedure exits with the Zero and Carry flags set according to the result of the comparison. If both strings are equal, ZF is set to 1; otherwise, ZF is set to 0.

The Carry flag is set to indicate the relation between the strings:

(JB for "string1 < string2," JA for "string1 > string2," JE for "string1 = string2")

STR_LENGTH PROCEDURE

The Str_length procedure returns the length of a string in the EAX register. It takes one argument: a pointer to the string.

The Str_length procedure works by scanning the string byte by byte until it reaches the null terminator.

The procedure increments the EAX register for each byte in the string. After reaching the null terminator, the procedure returns the value in the EAX register.

The following is a more detailed explanation of the Str_length procedure:

This code works by scanning the string byte by byte until it reaches the null terminator. The code increments the EAX register for each byte in the string. After reaching the null terminator, the code returns the value in the EAX register.

To use the Str_length procedure, you would pass the address of the string to the procedure as an argument.

```
255 ;-----------------------------------------------------
256 ; Str_length Procedure
257 ; Calculates the length of a null-terminated string.
258 ; Returns the length of the string in EAX.
259 ;-----------------------------------------------------
260 Str_length PROC USES edi,
261     pString:PTR BYTE   ; pointer to the string
262
263     mov edi, pString   ; Initialize edi with the pointer to the string.
264     mov eax, 0         ; Initialize eax to 0, which will store the character count.
265
266 L1:
267     cmp BYTE PTR [edi], 0   ; Check if the current character is the null terminator (end of string).
268     je L2                   ; If it's the end of the string, exit the loop.
269
270     inc edi                 ; Move to the next character in the string.
271     inc eax                 ; Increment the character count by 1.
272
273     jmp L1                  ; Repeat the loop to process the next character.
274
275 L2:
276     ret                     ; Return with the length of the string in EAX.
277
278 Str_length ENDP
```

Line 1: The Str_length procedure pushes the EDI register onto the stack. This is necessary because the procedure uses this register.

Line 2: The Str_length procedure moves the pointer to the string into the EDI register.

Line 3: The Str_length procedure moves the value 0 into the EAX register. This will be used to store the length of the string.

Line 4: The Str_length procedure enters a loop. On each iteration of the loop, the following steps are performed:

- The Str_length procedure compares the byte at the memory location pointed to by the EDI register to the null terminator (0x00).
- If the byte is equal to the null terminator, the Str_length procedure exits the loop.
- If the byte is not equal to the null terminator, the Str_length procedure increments the EDI register and the EAX register.

Line 10: The Str_length procedure exits the loop and returns to the caller.

The following is an example of how to use the Str_length procedure:

```
; Get the length of the string "Hello, world!"
mov eax, OFFSET "Hello, world!"
call Str_length

; The EAX register will now contain the value 12, which is the length of the string "Hello, world!"
```

STR_COPY PROCEDURE

The Str_copy procedure copies a null-terminated string from a source location to a target location.

It takes two arguments: pointers to the source and target strings, respectively.

The Str_copy procedure works by first calling the Str_length procedure to get the length of the source string.

The Str_copy procedure then copies the source string to the target string byte by byte using the REP MOVSB instruction.

The following is the MASM code for the Str_copy procedure:

```
;-----------------------------------------------------
; Str_copy Procedure
; Copies a string from the source to the target.
; Requires: the target string must contain enough space
; to hold a copy of the source string.
;-----------------------------------------------------
Str_copy PROC USES eax ecx esi edi,
    source:PTR BYTE,    ; source string
    target:PTR BYTE     ; target string

    INVOKE Str_length, source   ; Calculate the length of the source string and store it in EAX.
    mov ecx, eax                ; Copy the length into ECX for REP count.
    inc ecx                     ; Add 1 for the null byte at the end of the string.

    mov esi, source             ; Initialize esi with the source pointer.
    mov edi, target             ; Initialize edi with the target pointer.

    cld                         ; Set the direction flag to forward.

    rep movsb                   ; Use REP to copy the string byte by byte from source to target.

    ret                         ; Return when the string is copied.

Str_copy ENDP
```

This code defines the Str_copy procedure, which copies a null-terminated string from the source to the target.

It uses the Str_length procedure to calculate the length of the source string, sets up pointers to both source and target strings, and then uses the rep movsb instruction to copy the string byte by byte.

The procedure returns once the entire string is copied to the target.

To use the Str_copy procedure, you would pass the addresses of the source and target strings to the procedure as arguments.

For example, to copy the string "Hello, world!" from one location to another, you would use the following code:

```
mov eax, OFFSET "Hello, world!"
mov ebx, OFFSET target_string
call Str_copy

; The target_string variable will now contain a copy of the string "Hello, world!"
```

STR_TRIM PROCEDURE

The Str_trim procedure removes all occurrences of a specified trailing character from the end of a null-terminated string. For example, it can trim trailing spaces from a string.

It takes two parameters:

pString - Pointer to the null-terminated ASCII string to trim.

char - The ASCII character to trim from the end of the string It does not return anything, but modifies the string in-place by truncating it.

Logic

It handles several cases:

1. Empty string - nothing to do
2. String with trailing character(s) - remove them
3. String with only the trailing char - truncate to empty string
4. String without any trailing char - leave unchanged
5. String with trailing char(s) followed by other chars - remove only trailing char(s)

To trim the string, it inserts a null byte (\0) after the last character to keep. Any chars after the null become insignificant.

It uses these steps:

1. Get length of string
2. Check if length is 0 and exit if so (empty string case)
3. Initialize loop counter to string length

4. Point to last character
5. Loop backwards until beginning

- Check if current char matches trailing char to trim
- If yes, decrement counter to keep backing up
- If no, insert null byte after current char and exit loop
- Insert null byte truncates the string

```
360 Str_trim PROC USES eax ecx edi,
361             pString:PTR BYTE, ; string pointer
362             char: BYTE        ; trailing char to trim
363
364
365     mov  edi,pString        ; point EDI to string
366     INVOKE Str_length,edi   ; get length in EAX
367
368     cmp eax,0               ; is length 0?
369     je L3                   ; yes, exit
370
371     mov ecx,eax             ; ECX = length
372     dec eax                 ; EAX = length - 1
373     add edi,eax             ; point to last char
374
375 L1:
376     mov al,[edi]            ; load character
377     cmp al,char             ; compare to trailing char
378     jne L2                  ; no match, insert null
379     dec edi                 ; match, keep backing up
380     loop L1
381
382 L2:
383     mov BYTE PTR [edi+1],0 ; insert null byte
384
385 L3:
386     ret
387 Str_trim ENDP
```

Line 1: The Str_trim procedure pushes the EAX, ECX, and EDI registers onto the stack. This is necessary because the procedure uses these registers.

Line 2: The Str_trim procedure moves the pointer to the string to be trimmed into the EDI register.

Line 3: The Str_trim procedure calls the Str_length procedure to get the length of the string. The length of the string is stored in the EAX register.

Line 4: The Str_trim procedure compares the length of the string to 0. If the length of the string is equal to 0, then the string is empty and the procedure exits.

Line 6: The Str_trim procedure moves the length of the string to the ECX register. This will be used as the loop counter.

Line 7: The Str_trim procedure decrements the EAX register. This will be used to point to the last character in the string.

Line 8: The Str_trim procedure adds the EAX register to the EDI register. This will point the EDI register to the last character in the string.

Line 9: The Str_trim procedure enters a loop. On each iteration of the loop, the following steps are performed:

- The Str_trim procedure moves the byte at the memory location pointed to by the EDI register into the AL register.
- The Str_trim procedure compares the AL register to the character to be trimmed.
- If the two characters are equal, then the Str_trim procedure decrements the EDI register and continues to the next iteration of the loop.
- If the two characters are not equal, then the Str_trim procedure breaks out of the loop.

Line 14: The Str_trim procedure inserts a null byte at the memory location pointed to by the EDI register plus one. This will terminate the string at the last character that is not the character to be trimmed.

Line 15: The Str_trim procedure exits the loop.

Line 16: The Str_trim procedure pops the EAX, ECX, and EDI registers from the stack.

Line 17: The Str_trim procedure returns.

Here is an example of how to use the Str_trim procedure:

```
390 ; Trim all trailing spaces from the string "Hello, world!   "
391 mov eax, OFFSET "Hello, world!   "
392 call Str_trim
393 ; The EAX register will now contain a pointer to the string "Hello, world!"
```

STR_UCASE PROCEDURE

```
;-----------------------------------------------------
; Str_ucase Procedure
; Converts a null-terminated string to uppercase.
; Returns: nothing
;-----------------------------------------------------
Str_ucase PROC USES eax esi,
    pString:PTR BYTE  ; Pointer to the string

    mov esi, pString  ; Initialize esi with the address of the string.
L1:
    mov al, [esi]    ; Load the character from the string.
    cmp al, 0         ; Check if it's the end of the string.
    je L3             ; If yes, exit the loop.
    cmp al, 'a'       ; Compare the character with 'a'.
    jb L2             ; If it's below 'a', go to L2.
    cmp al, 'z'       ; Compare the character with 'z'.
    ja L2             ; If it's above 'z', go to L2.

    and BYTE PTR [esi], 11011111b
    ; Convert the character to uppercase by clearing the 6th bit.
L2:
    inc esi           ; Move to the next character.
    jmp L1            ; Repeat the loop.
L3:
    ret               ; Return when the entire string is converted to uppercase.
Str_ucase ENDP
```

The first part of the code sets up the procedure. It expects a pointer to a string (null-terminated) in pString, and it initializes esi with the address of this string.

Here's what's happening within the loop:

mov al, [esi] loads the character at the memory address pointed to by esi into the al register.

cmp al, 0 checks if the character is the null terminator (end of the string). If it is, the loop exits (je L3).

cmp al, 'a' compares the character with the ASCII value of 'a'. If the character is below 'a', it jumps to L2.

cmp al, 'z' compares the character with the ASCII value of 'z'. If the character is above 'z', it also jumps to L2.

and BYTE PTR [esi], 11011111b converts the character to uppercase by clearing the 6th bit (bit 5) in the ASCII code. This changes lowercase letters to uppercase.

L2: marks the location to which the code jumps when the character is not a lowercase letter, ensuring it's skipped.

inc esi increments the esi register to point to the next character in the string.

jmp L1 jumps back to the beginning of the loop, continuing the process until the end of the string is reached.

Finally, L3: is the label that is reached when the end of the string is detected. At this point, the loop exits, and the procedure returns.

The entire string is converted to uppercase by iterating through each character, and no value is returned; the original string is modified in memory.

STRING LIBRARY DEMO PROGRAM

The "String Library Demo" program demonstrates the usage of string-handling procedures from the Irvine32 library. It performs the following tasks:

Trimming trailing characters from string_1 using the Str_trim procedure.

Converting string_1 to uppercase using the Str_ucase procedure.

Comparing string_1 to string_2 using the Str_compare procedure.

Displaying the length of string_2 using the Str_length procedure. Here's the code with detailed explanations:

```
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   ; Remove trailing characters from string_1.
   call upper_case    ; Convert string_1 to uppercase.
   call compare_strings ; Compare string_1 to string_2.
   call print_length  ; Display the length of string_2.
   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
      mov edx, OFFSET msg3
   .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
```

The program's output is as follows:

After trimming string_1, it displays "string_1 after trimming: abcde."

After converting string_1 to uppercase, it displays "string_1 in upper case: ABCDE."

It then compares string_1 and string_2 and displays one of the messages depending on the result.

Finally, it displays the length of string_2.

This program showcases the use of various string-handling procedures from the Irvine32 library and provides informative messages for each step.

Strings using Irvine64

```
INCLUDE Irvine64.inc
.data
    source BYTE "AABCDEFGAABCDFG", 0
    ; size = 15
    target BYTE 20 DUP(0)

.code
    Str_compare PROTO
    Str_length PROTO
    Str_copy PROTO
    ExitProcess PROTO

    main PROC
        mov rcx, OFFSET source
        call Str_length
        ; Returns length in RAX
        mov rsi, OFFSET source
        mov rdi, OFFSET target
        call Str_copy
        ; We just copied the string, so they should be equal.
        call Str_compare
        ; ZF = 1, strings are equal
        ; Change the first character of the target string, and
        ; compare them again.
        mov BYTE PTR [rdi], 'B'
        call Str_compare
        ; CF = 1, source < target
        mov ecx, 0
        call ExitProcess
    main ENDP
```

Actual Implementation of Procedures:

To have a complete working program, you need to provide the actual implementation of the Str_compare, Str_length, and Str_copy procedures.

These procedures are essential for the functionality of your program. They should be implemented with appropriate assembly code to perform the desired operations.

I'll provide you with the implementation of these procedures in Irvine64 assembly:

; Str_compare Procedure
; Compares two strings
; Receives:

```
; RSI points to the source string
; RDI points to the target string
; Returns:
; Sets ZF if the strings are equal
; Sets CF if source < target

Str_compare PROC
   ; Implementation of Str_compare
   ; ...
   ret
Str_compare ENDP

; Str_length Procedure
; Gets the length of a string
; Receives: RCX points to the string
; Returns: length of string in RAX

Str_length PROC
   ; Implementation of Str_length
   ; ...
   ret
Str_length ENDP

; Str_copy Procedure
; Copies a source string to a location indicated by a target pointer
; Receives:
; RSI points to the source string
; RDI points to the target string
; Returns: nothing

Str_copy PROC
   ; Implementation of Str_copy
   ; ...
   ret
Str_copy ENDP
```

Output and Display:

In the provided test program, there is no code for displaying the results of these operations. You should add code to display whether the strings are equal, the length of the string, and the comparison results. For example, you can use WriteString and WriteDec functions to display these results:

```
mov rsi, OFFSET msg1
call WriteString  ; Display result message
call Crlf
; Check ZF and CF flags to determine equality or comparison result
; Display results accordingly
```

Irvine64 Library Setup:

The Irvine64 library needs to be included and set up properly in your assembly environment. You should have instructions at the beginning of your program to include the Irvine64 library and set it up. This usually involves specifying the paths and configurations for the Irvine64 library. Here is an example:

```
INCLUDE Irvine64.inc  ; Include Irvine64 library

.data
; Your data declarations go here

.code
main PROC
; Your program's main code goes here

main ENDP

END main
```

Actual Program:

```
INCLUDE Irvine64.inc

; -----------------------------------------------
; Str_compare
; Compares two strings
; Receives:
; RSI points to the source string
; RDI points to the target string
; Returns:
; Sets ZF if the strings are equal
; Sets CF if source < target
; -----------------------------------------------
Str_compare PROC
    USES rax rdx rsi rdi

L1:
    mov al, [rsi]
    mov dl, [rdi]
    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 rsi          ; Point to next
    inc rdi
    cmp al, dl       ; Characters equal?
    je L1            ; Yes, continue loop
                    ; No: Exit with flags set

L3:
    ret

Str_compare ENDP

; -------------------------------------------------
; Str_copy
; Copies a source string to a location indicated by a target pointer
; Receives:
; RSI points to the source string
; RDI points to the location where the copied string will be stored
; Returns: nothing
; -------------------------------------------------
Str_copy PROC
    USES rax rcx rsi rdi

    mov rcx, rsi    ; Get length of the source string
    call Str_length ; Returns length in RAX
    mov rcx, rax    ; Loop counter
    inc rcx          ; Add 1 for the null byte
    cld             ; Direction = up
    rep movsb        ; Copy the string
    ret
```

```
587 Str_copy ENDP
588
589 ; ---------------------------------------------
590 ; Str_length
591 ; Gets the length of a string
592 ; Receives: RCX points to the string
593 ; Returns: length of the string in RAX
594 ; ---------------------------------------------
595 Str_length PROC
596     USES rdi
597
598     mov rdi, rcx    ; Get the pointer
599     mov eax, 0      ; Character count
600 L1:
601     cmp BYTE PTR [rdi], 0  ; End of string?
602     je L2                  ; Yes: quit
603     inc rdi                ; No: Point to the next
604     inc rax                ; Add 1 to count
605     jmp L1
606 L2:
607     ret                   ; Return count in RAX
608
609 Str_length ENDP
```

```
611 .data
612     source BYTE "AABCDEFGAABCDFG",0
613     target BYTE 20 dup(0)
614
615 .code
616     main PROC
617         mov rcx, offset source
618         call Str_length      ; Returns length in RAX
619         mov rsi, offset source
620         mov rdi, offset target
621         call Str_copy
622         ; We just copied the string, so they should be equal.
623         call Str_compare
624         ; ZF = 1, strings are equal
625         ; Change the first character of the target string, and compare them again.
626         mov target, 'B'
627         call Str_compare
628         ; CF = 1, source < target
629         mov ecx, 0
630         call ExitProcess
631
632     main ENDP
633
634 END main
```

Explanation:

Irvine64 Library: The INCLUDE Irvine64.inc statement includes the Irvine64 library, providing access to Irvine's assembly functions and features.

USES Keyword: In the Str_compare and Str_copy procedures, the USES keyword is used to specify registers that will be pushed onto the stack and popped off the stack upon return from the procedure. This helps maintain the calling conventions.

Str_compare Procedure: Compares two strings pointed to by RSI and RDI. It sets the Zero Flag (ZF) if the strings are equal and the Carry Flag (CF) if the source is less than the target.

Str_copy Procedure: Copies a source string (pointed to by RSI) to a location indicated by the target pointer (RDI). It calculates the length of the source string using Str_length, then uses rep movsb to perform the copy.

Str_length Procedure: Calculates the length of a null-terminated string. It receives a pointer in RCX, and the result is returned in RAX.

.data Section: Data declarations for source and target strings.

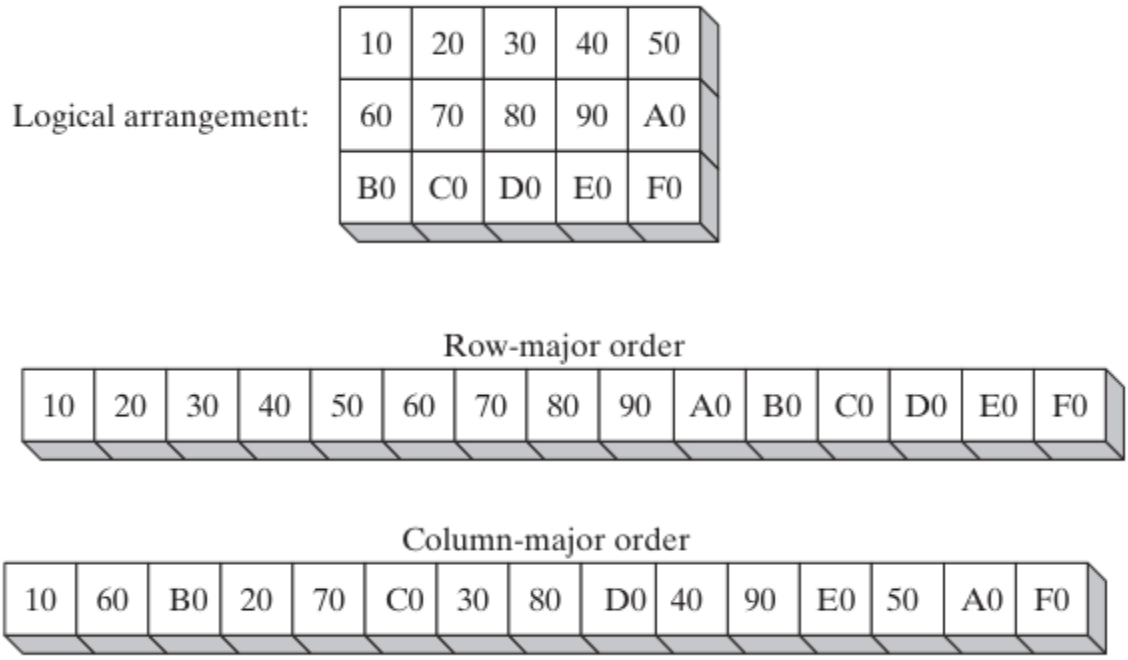
.code Section: The main procedure demonstrates the use of these string procedures, copying the string, comparing strings, and changing a character for comparison.

This code illustrates how to use these string procedures in Irvine64 assembly, focusing on the Irvine64 register usage, stack management, and proper procedure calling conventions.

It's essential to configure your environment correctly to work with Irvine64 and ensure you have the Irvine64 library properly set up.

2D ARRAYS

Row-major order and **column-major order** are two different ways of storing a two-dimensional array in memory. The main difference is the order in which the elements of the array are stored.

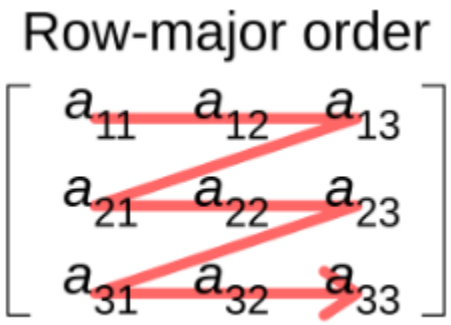


Row-major order

Row-major order is the most common method, and it is the method used by most high-level programming languages. In row-major order, the elements of each row are stored contiguously in memory.

```
Row-major order:
[0][0] [0][1] [0][2]
[1][0] [1][1] [1][2]
[2][0] [2][1] [2][2]
```

This means that the first element of the first row is stored at the beginning of the memory block, followed by the second element of the first row, and so on. The last element of the first row is followed by the first element of the second row, and so on. This continues until the last element of the last row is stored.

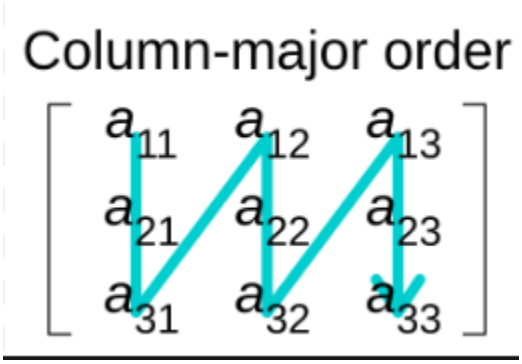


Column-major order

Column-major order is less common, but it is used in some applications, such as linear algebra. In column-major order, the elements of each column are stored contiguously in memory.

```
Column-major order:
[0][0] [1][0] [2][0]
[0][1] [1][1] [2][1]
[0][2] [1][2] [2][2]
```

This means that the first element of the first column is stored at the beginning of the memory block, followed by the first element of the second column, and so on. The last element of the first column is followed by the second element of the second column, and so on. This continues until the last element of the last column is stored.



Which order to use?

The choice of which order to use depends on the application. Row-major order is generally more efficient for accessing elements of the array by row, while column-major order is more efficient for accessing elements of the array by column.

How to implement a two-dimensional array in MASM To implement a two-dimensional array in MASM, you can use either row-major order or column-major order. However, it is important to be **consistent with the ordering that you choose.**

Implementations

To implement a two-dimensional array in row-major order, you can use the following steps:

- Allocate a block of memory for the array.
- The size of the memory block depends on the size of the array and the data type of the elements of the array.
- Initialize the elements of the array. To access an element of the array, you can use the following formula:

```
element[i][j] = array[i * number_of_columns + j]
```

where i is the row index and j is the column index.

For example, the following code implements a two-dimensional array of integers in row-major order:

```
; Declare a two-dimensional array of integers.
array dw 100 dup(0)

; Initialize the elements of the array.
mov ebx, 1
mov ecx, 100
mov edi, array
loop:
mov dword ptr [edi], ebx
inc ebx
inc edi
loop loop

; Access an element of the array.
mov eax, array[1 * 10 + 2]
```

Explanation:

- The array is declared as 100 contiguous DWORDs initialized to 0. This creates a 10x10 array since each DWORD is 2 bytes (10 * 10 = 100).
- EBX is used as a counter from 1 to 100 to store sequential values in the array.
- EDI points to the start of the array and is incremented each iteration to move through the elements.

- ECX counts iterations from 1 to 100 to fill all elements.
- Array elements are accessed using **row*numCols + col***. *Here **row 1**, **col 2** is at offset* **10 + 2 = 12.**
- The elements are stored in row-major order - row 1, then row 2, etc sequentially in memory.

So this shows a typical way to declare, initialize, and access a 2D array in MASM using row-major layout.

Base-Index Operands

A base-index operand is a type of operand that allows you to access memory using the sum of two register values: the base register and the index register.

Base-index operands are very useful for accessing arrays, because they allow you to calculate the address of an element of an array using the row and column indices of the element.

To use a base-index operand, you must first load the base register with the address of the array.

You can then load the index register with the row and/or column index of the element of the array that you want to access.

Finally, you can use the base-index operand to access the element of the array.

The following example shows how to use a base-index operand to access an element of a two-dimensional array:

```
672 ; Declare a two-dimensional array of integers.
673 array dw 1000h, 2000h, 3000h
674
675 ; Load the base register with the address of the array.
676 mov ebx, OFFSET array
677
678 ; Load the index register with the row and column indices of the element that we want to access.
679 mov esi, 1 ; row index
680 mov edi, 2 ; column index
681
682 ; Calculate the address of the element using the base-index operand.
683 mov eax, [ebx + esi * 2 + edi * 4]
684
685 ; Display the value of the element.
686 mov edx, 0 ; service number
687 mov ecx, 1 ; buffer offset
688 mov al, [eax] ; buffer byte
689 int 21h ; system call
```

The above code will display the value 2000h, which is the value of the element at row 1, column 2 of the array.

<u>Column-Major Order vs. Row-Major Order</u>

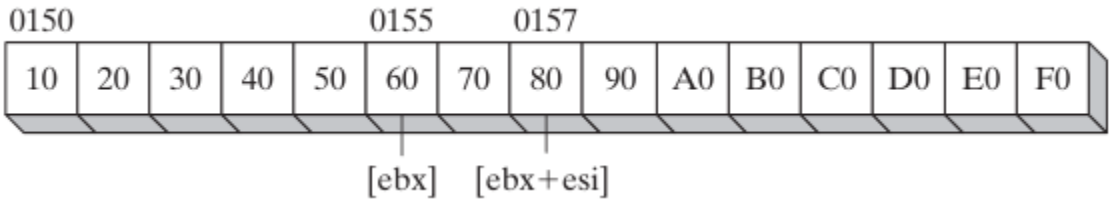
Two-dimensional arrays can be stored in memory in two different ways: column-major order and row-major order.

In **column-major order,** the elements of each column are stored contiguously in memory.

In **row-major order,** the elements of each row are stored contiguously in memory.

Most high-level programming languages use row-major order to store two-dimensional arrays.

Therefore, if you are writing assembly language code that will be used with a high-level language, you should use row-major order to store your two-dimensional arrays.



To access an elements of the array above, you can use the following formula:

```
element[row_index][column_index] = array[row_index * number_of_columns + column_index]
```

where row_index is the row index of the element and column_index is the column index of the element.

For example, to access the element at row 1, column 2 of the array, you would use the following code:

```
698 mov ebx, OFFSET array ; load the base register with the address of the array
699 mov esi, 1 ; load the index register with the row index
700 mov al, [ebx + esi * 2] ; access the element at row 1, column 2
```

The al register will now contain the value of the element at row 1, column 2 of the array.

Calculating a Row Sum

The calc_row_sum procedure calculates the sum of a selected row in a matrix of 8-bit integers. It receives the following inputs:

- **EBX:** The offset of the matrix in memory.
- **EAX:** The index of the row to calculate the sum for.
- **ECX:** The size of each row in the matrix, in bytes.
- The procedure returns the sum of the row in the EAX register.

The procedure works by first calculating the offset of the row in the matrix. This is done by multiplying the row index by the row size.

The procedure then adds this offset to the base address of the matrix to get the address of the first element in the row.

The procedure then iterates over the row, adding each element to the accumulator. The accumulator is initialized to 0 before the loop starts.

The loop iterates until the end of the row is reached.

After the loop finishes, the procedure returns the sum in the accumulator in the EAX register.

Here is a more detailed explanation of the code:

```
;-----------------------------------------------------
; calc_row_sum
; Calculates the sum of a row in a byte matrix.
; Receives: EBX = table offset, EAX = row index,
; ECX = row size, in bytes.
; Returns: EAX holds the sum.
;-----------------------------------------------------
calc_row_sum PROC USES ebx ecx edx esi

; Calculate the offset of the row.
mul ecx ; row index * row size

; Add the offset to the base address of the matrix to get the address of the first element in the row.
add ebx,eax ; row offset

; Initialize the accumulator.
mov eax,0 ; accumulator

; Set the column index to 0.
mov esi,0 ; column index

; Loop over the row, adding each element to the accumulator.
L1:
movzx edx,BYTE PTR[ebx + esi] ; get a byte
add eax,edx ; add to accumulator
inc esi ; next byte in row
loop L1

; Return the sum in the accumulator.
ret
calc_row_sum ENDP
```

The BYTE PTR operand size in the MOVZX instruction is required to clarify the operand size. The MOVZX instruction converts a byte to a doubleword.

The BYTE PTR operand size tells the assembler that the operand at the address specified by the ebx + esi register is a byte.

The calc_row_sum procedure is a useful example of how to use base-index addressing to access elements of a two-dimensional array and to perform common tasks on arrays, such as calculating the sum of a row.

Scale Factors

A scale factor is a multiplier that is used to scale the index operand when accessing elements of an array using base-index addressing.

The scale factor is required because the size of the elements in an array can vary.

For example, if an array contains bytes, then the scale factor is 1. If an array contains words, then the scale factor is 2. If an array contains doublewords, then the scale factor is 4.

The following table shows the scale factor for different types of data:

Data type	Scale factor
Byte	1
Word	2
Doubleword	4
Quadword	8

To use a scale factor, you simply multiply the index operand by the scale factor before adding it to the base address of the array.

This will give you the address of the element in the array at the specified index.

For example, the following code accesses the element at row 1, column 2 of an array of words:

```
745 ; tableW is an array of words.
746 mov ebx, OFFSET tableW ; load the base register with the address of the array
747 mov esi, 2 ; load the index register with the column index
748 mov ax, [ebx + esi * TYPE tableW] ; access the element at row 1, column 2
```

The TYPE tableW operand size in the MOV instruction tells the assembler that the elements in the array are words.

Therefore, the scale factor is 2. The esi * TYPE tableW operand is multiplied by 2 before it is added to the base address of the array.

This gives us the address of the element at row 1, column 2.

Scale factors can be useful for writing efficient code to access arrays.

By using scale factors, you can avoid having to keep track of the size of the elements in the array. This can make your code more readable and maintainable.

Base-Index-Displacement Operands

A base-index-displacement operand is a type of operand that allows you to access memory using the sum of the following values:

- **A displacement.**
- **A base register.**
- **An index register.**
- **An optional scale factor**.

Base-index-displacement operands are well suited for processing two-dimensional arrays. The displacement can be an array name, the base operand can hold the row offset, and the index operand can hold the column offset.

The following example shows how to use a base-index-displacement operand to access an element of a two-dimensional array of doublewords:

```
759 ; tableD is a two-dimensional array of doublewords.
760 ; Rowsize is the size of each row in the array, in bytes.
761
762 mov ebx, Rowsize ; load the base register with the row offset
763 mov esi, 2 ; load the index register with the column offset
764 mov eax, tableD[ebx + esi * TYPE tableD] ; access the element at row 1, column 2
```

The tableD[ebx + esi * TYPE tableD] operand specifies the address of the element at row 1, column 2 of the array.

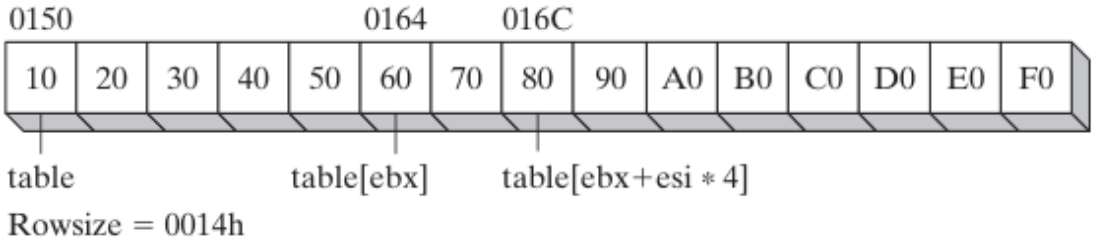
The ebx register contains the row offset, and the esi register contains the column offset.

The TYPE tableD operand size tells the assembler that the elements in the array are doublewords. Therefore, the scale factor is 4.

Base-index-displacement operands can be useful for writing efficient code to access arrays.

By using base-index-displacement operands, you can avoid having to keep track of the size of the elements in the array and the offset of each row in the array.

This can make your code more readable and maintainable.



The diagram you provided shows the positions of the EBX and ESI registers relative to the array tableD. The EBX register contains the row offset, and the ESI register contains the column offset.

The tableD array begins at offset 0150h. The EBX register contains the value 20h, which is the size of each row in the array, in bytes. Therefore, the EBX register points to the beginning of the second row in the array.

The ESI register contains the value 2, which is the column index of the element that we want to access. Therefore, the ESI register points to the third element in the second row of the array.

Base-index-displacement operands are a powerful tool for accessing arrays in assembly language.

Base-Index Operands in 64-Bit Mode.

The program below is a short program that uses a procedure named get_tableVal to locate a value in a two-dimensional table of 64-bit integers. The program demonstrates how to use base-index-displacement operands in 64-bit mode.

```
0804 ; Two-dimensional arrays in 64-bit mode (TwoDimArrays.asm)
0805 ; Prototypes for procedures
0806 Crlf proto
0807 WriteInt64 proto
0808 ExitProcess proto
0809
0810 .data
0811     table QWORD 1,2,3,4,5      ; Define a two-dimensional array with 3 rows and 5 columns
0812     RowSize = ($ - table)      ; Calculate the size of one row in bytes
0813
0814     QWORD 6,7,8,9,10          ; Define the second row
0815     QWORD 11,12,13,14,15       ; Define the third row
0816
0817 .code
0818 main PROC
0819     ; Set row and column indices
0820     mov rax, 1         ; Row index (zero-based)
0821     mov rsi, 4         ; Column index (zero-based)
0822
0823     call get_tableVal  ; Call the get_tableVal procedure to retrieve the value
0824     call WriteInt64    ; Display the retrieved value
0825     call Crlf          ; Insert a line break
0826
0827     mov ecx, 0
0828     call ExitProcess   ; End the program
0829
0830 main ENDP
```

The main procedure performs the following steps:

It loads the row index (1) into the RAX register.

It loads the column index (4) into the RSI register.

It calls the get_tableVal procedure to get the value at the specified row and column in the table array.

It calls the WriteInt64 procedure to display the value in the RAX register.

It calls the ExitProcess procedure to end the program.

The get_tableVal procedure performs the following steps:

It loads the row offset into the RBX register.

It multiplies the row offset by the size of a quadword to get the offset of the row in the array.

It adds the column offset to the row offset to get the offset of the element in the array.

It loads the value at the specified offset into the RAX register.

It returns from the procedure.

The get_tableVal procedure uses a base-index-displacement operand to access the element in the array.

The base operand is the RBX register, which contains the row offset.

The index operand is the RSI register, which contains the column offset.

The scale factor is omitted, because the elements in the array are quadwords, and the size of a quadword is 1.

The program you provided demonstrates how to use base-index-displacement operands in 64-bit mode.

Base-index-displacement operands are a powerful tool for accessing arrays in assembly language.

SEARCHING AND SORTING ALGORITHMS

Bubble sort

The bubble sort algorithm is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. The algorithm starts at the beginning of the array and compares the first two elements.

If the first element is greater than the second element, the two elements are swapped. The algorithm then moves on to the next two elements and repeats the process.

The algorithm continues to iterate through the array until it reaches the end.

The following is a pseudocode implementation of the bubble sort algorithm:

```
0839 bubble_sort(array):
0840   for i in range(len(array) - 1):
0841     for j in range(len(array) - i - 1):
0842       if array[j] > array[j + 1]:
0843         array[j], array[j + 1] = array[j + 1], array[j]
```

The bubble sort algorithm is a simple and straightforward algorithm, but it is not very efficient for large arrays.

This is because the algorithm has to compare every pair of elements in the array for each iteration. For an array of size n, the bubble sort algorithm has a **time complexity of $O(n^2)$.**

Analysis of bubble sort performance.

The following table shows the sort times for various array sizes, assuming that 1000 array elements can be sorted in 0.1 second:

Array Size	Time (seconds)
1,000	0.1
10,000	10.0
100,000	1000
1,000,000	100,000 (27.78 hours)

As you can see, the sort time increases quadratically with the array size. This means that the bubble sort algorithm is not very efficient for large arrays.

The bubble sort algorithm is a simple and straightforward sorting algorithm, but it is not very efficient for large arrays.

If you need to sort a large array, you should use a more efficient sorting algorithm, such as the quicksort algorithm or the merge sort algorithm.

```
0847 ; Bubble sort algorithm in MASM
0848 section .data
0849     array: dw 5, 3, 2, 1, 4
0850 section .code
0851     start:
0852
0853     mov esi, array
0854     mov ecx, 5 ; length of the array
0855     L1:
0856     mov edi, esi
0857     add edi, 4
0858     L2:
0859     cmp [esi], [edi]
0860     jg L3 ; swap if esi > edi
0861     xchg [esi], [edi]
0862     add esi, 4
0863     cmp esi, array + ecx * 4 - 4
0864     jne L2
0865     loop L1
0866     ; array is now sorted
0867     exit
```

The array section in the data segment initializes an array with values to be sorted.

The code begins by setting up registers, with esi pointing to the start of the array and ecx containing the length of the array (in this case, 5).

The outer loop, labeled as L1, iterates through the array. This corresponds to the outer loop counter (cx1) in the notes.

Inside the outer loop, the inner loop labeled as L2 is used to compare and swap elements, corresponding to the inner loop counter (cx2) in the notes.

The comparison is done using cmp, and if the current element ([esi]) is greater than the next element ([edi]), a swap is performed using xchg.

The code ensures that the inner loop (L2) iterates through the entire array by comparing esi to the end of the array (array + ecx * 4 - 4).

After completing the inner loop for a given pass through the array, it uses loop to decrement the outer loop counter and repeats the process until the outer loop counter is equal to 0.

Once the sorting is complete, the array is in ascending order.

C++ version:

```
0870 int BinSearch(int values[], const int searchVal, int count) {
0871     int first = 0;
0872     int last = count - 1;
0873 
0874     while (first <= last) {
0875         int mid = (last + first) / 2;
0876 
0877         if (values[mid] < searchVal)
0878             first = mid + 1;
0879         else if (values[mid] > searchVal)
0880             last = mid - 1;
0881         else
0882             return mid;  // success
0883     }
0884 
0885     return -1;  // not found
0886 }
```

Assembly version:

```
; BinarySearch
; Searches an array of signed integers for a single value.
; Receives: Pointer to array, array size, search value.
; Returns: If a match is found, EAX = the array position of the
; matching element; otherwise, EAX = -1.
BinarySearch PROC USES ebx edx esi edi,
   pArray:PTR DWORD,
   Count:DWORD,
   searchVal:DWORD

   LOCAL first:DWORD,
      last:DWORD,
      mid:DWORD

   mov first, 0
   mov eax, Count
   dec eax
   mov last, eax
   mov edi, searchVal
   mov ebx, pArray
```

```
L1:
   mov eax, first
   cmp eax, last
   jg L5

   mov eax, last
   add eax, first
   shr eax, 1
   mov mid, eax

   mov esi, mid
   shl esi, 2
   mov edx, [ebx+esi]

   cmp edx, edi
   jge L2
   mov eax, mid
   inc eax
   mov first, eax
   jmp L4

L2:
   cmp edx, edi
   jle L3
   mov eax, mid
   dec eax
   mov last, eax
   jmp L4

L3:
   mov eax, mid
   jmp L9

L4:
   jmp L1

L5:
   mov eax, -1

L9:
   ret
```

```
BinarySearch ENDP
```

Program 2:

```
; Bubble Sort and Binary Search (BinarySearchTest.asm)
; Bubble sort an array of signed integers and perform a binary search.

INCLUDE Irvine32.inc
INCLUDE BinarySearch.inc  ; Include procedure prototypes

LOWVAL = -5000
HIGHVAL = +5000
ARRAY_SIZE = 50

.data
array DWORD ARRAY_SIZE DUP(?)

.code
main PROC
   call Randomize  ; Initialize random number generator

   ; Fill an array with random signed integers
   INVOKE FillArray, ADDR array, ARRAY_SIZE, LOWVAL, HIGHVAL

   ; Display the array
   INVOKE PrintArray, ADDR array, ARRAY_SIZE
   call WaitMsg

   ; Perform a bubble sort and redisplay the array
   INVOKE BubbleSort, ADDR array, ARRAY_SIZE
   INVOKE PrintArray, ADDR array, ARRAY_SIZE

   ; Demonstrate a binary search
   call AskForSearchVal

   ; Perform the binary search and display the results
   INVOKE BinarySearch, ADDR array, ARRAY_SIZE, eax
   call ShowResults

   exit
main ENDP
```

```
; Prompt the user for a signed integer
AskForSearchVal PROC
   .data
   prompt BYTE "Enter a signed decimal integer in the range of -5000 to +5000 to find in
the array: ",0
   .code
   call Crlf
   mov edx, OFFSET prompt
   call WriteString
   call ReadInt
   ret
AskForSearchVal ENDP

; Display the resulting value from the binary search
ShowResults PROC
   .data
   msg1 BYTE "The value was not found.",0
   msg2 BYTE "The value was found at position ",0
   .code
   .IF eax == -1
      mov edx, OFFSET msg1
      call WriteString
   .ELSE
      mov edx, OFFSET msg2
      call WriteString
      call WriteDec
   .ENDIF
   call Crlf
   call Crlf
   ret
ShowResults ENDP

END main
```

The provided assembly code, BinarySearchTest.asm, is a program that demonstrates the use of the bubble sort and binary search functions to work with an array of signed integers. Here's an overview of what the code does:

Initialization:

It starts by including necessary libraries and defining constants for the minimum and maximum values (LOWVAL and HIGHVAL) and the size of the array (ARRAY_SIZE).

The code also defines the array array and contains the .data and .code sections.

Main Procedure (main):

Calls the Randomize function to initialize the random number generator.

Invokes the FillArray procedure to fill the array with random signed integers within the specified range (LOWVAL to HIGHVAL).

Displays the original content of the array using the PrintArray procedure and waits for a message (WaitMsg).

Performs a bubble sort on the array using the BubbleSort procedure to sort the integers in ascending order.

Displays the sorted array using the PrintArray procedure. Prompts the user to enter a signed integer with the AskForSearchVal procedure.

AskForSearchVal Procedure:

Prompts the user to enter a signed integer within the specified range. Reads the integer entered by the user and returns it in the EAX register.

ShowResults Procedure:

Displays the result of the binary search.

If the binary search returns -1, indicating that the value was not found, it prints "The value was not found."

If the binary search returns the position of the value in the array, it prints "The value was found at position X," where X is the position.

In summary, this program generates a random array of signed integers, sorts the array using the bubble sort algorithm, and then performs a binary search on the sorted array to find a user-specified value.

It displays the results of the binary search, indicating whether the value was found and, if so, at what position in the array.