Chapter 7

String Instructions

Summary:

Instruction	Type	Source	Destination	Flags	Prefix
STOS	B	al	es:di	N	rep (cx time)
	W	ax	es:di		
LODS	B	ds:si	al	N	loop
	W	ds:si	ax		
SCAS	B	al	es:di	Y	repe, repne
	W	ax	es:di		
MOVS	B	ds:si	es:di	N	rep (cx time)
	W	ds:si	es:di		
CMPS	B	ds:si	es:di	Y	repe, repne
	W	ds:si	es:di		

String Processing

- String instructions are a set of **five instructions** available in the 8088 microprocessor.
- They are designed to process data in blocks, making them ideal for working with large chunks of memory, such as the video memory used for screen operations.
- The five primary string instructions are: **STOS (store string)**, **LODS (load string)**, **CMPS (compare string)**, **SCAS (scan string)**, and **MOVS (move string)**.

REP Prefix:

- String instructions can operate on single memory cells individually. However, they can also be used with a special prefix called the **REP prefix**.
- The REP prefix enables these instructions to repeat their operations for a specified number of data elements.
- It's important to note that this repetition is not like a traditional loop; it's hard-coded into the processor.
- Using the REP prefix can significantly speed up operations on large blocks of memory and reduce code size.

Common Elements:

- These instructions share common elements, regardless of their specific function.

- They all use source and destination index registers, **SI (source index)** and **DI (destination index)**, to access memory.
- When an instruction needs a source of data from memory, it uses **DS:SI** by default. However, it's possible to override this and use another segment register as the source.
- When a destination for data is needed, **ES:DI** is used, and there's no override option.
- For working with byte data, **AL (accumulator)** holds the value, while **AX (accumulator** and extension) is used for word data.
- For example, the STOS instruction stores a register value in memory, so it uses either AL or AX, and ES:DI points to the destination memory.
- Conversely, the LODS instruction loads data from memory to a register, and the source is pointed to by DS:SI, using AL or AX for storage.

Direction Flag (DF):

- String instructions can work in both directions: from the start towards the end or from the end towards the start.
- The direction of movement is controlled by the **Direction Flag (DF)** in the flags register.
- When DF is cleared, it's called **auto-increment mode**, and the direction is from lower memory addresses to higher ones.
- When DF is set, it's called **auto-decrement mode**, and the direction is from higher addresses to lower ones.
- You can set or clear the direction flag using the `cld` **(clear direction flag)** and `std` **(set direction flag)** instructions.

Variants:

- Each string instruction has two variants: a **byte variant** and a **word variant**.
- For example, STOS has STOSB for bytes and STOSW for words.
- You can identify these variants by appending a **"B"** or **"W"** to the instruction name, which specifies whether they operate on byte or word data.

STOS

The STOS instruction is used to transfer a byte or word from the accumulator register (AL or AX) to a specific memory location pointed to by ES:DI. After the transfer, the DI register is updated to point to the next memory location in the string.

- The source data is always in the AL register for bytes or the AX register for words.
- If the Direction Flag (DF) is cleared (auto-increment mode), the DI register will be incremented by one or two, depending on whether STOSB or STOSW is used.
- If the Direction Flag (DF) is set (auto-decrement mode), the DI register will be decremented by one or two, again depending on the specific variant used.

- When the REP prefix is used before the STOS instruction, the operation is repeated a specified number of times, as indicated by the value in the CX register.
- The CX register, often referred to as the counter register, is crucial for controlling the number of repetitions.
- Regardless of whether the byte or word variant is used, the CX register always decrements by one during each repetition. Therefore, CX represents the count of repetitions, not the count of bytes.

LODS

The LODS instruction is used to transfer a byte or word from the source memory location pointed to by DS:SI to the accumulator register (AL or AX). After the transfer, the SI register is updated to point to the next memory location in the source string.

LODS is typically used within a loop construct and is not typically combined with the REP prefix. This is because the value previously loaded into the AL or AX register is overwritten with each iteration of the instruction, and only the last value from the block of memory remains in the register.

SCAS

The SCAS instruction is used to compare a source byte or word in the accumulator register (AL or AX) with a byte or word located at the destination string element addressed by ES:DI. This instruction updates the processor flags based on the result of the comparison. Additionally, DI is updated to point to the next memory location in the destination string.

SCAS is frequently used to search for equality or inequality within a string of data. It can be used in conjunction with specific prefixes like **REPE (repeat while equal)** or **REPNE (repeat while not equal)** to specify the type of comparison desired. The primary purpose of SCAS is to search for a specific value **(byte in AL)** within a block of memory (destination string).

Here are two common use cases:

1. **Searching for Null-Terminated Strings:** SCAS can be used to search for a null (0) byte within a null-terminated string. This is often employed to determine the length of a string by counting the number of bytes processed until the null byte is encountered. In this case, REPNE (repeat while not equal) is used to keep searching until the null byte is found.
2. **Searching for Specific Values:** SCAS can be used to search for any specific byte or word within a block of memory. The instruction can be repeated with the REPE or REPNE prefix to locate the first occurrence or inequality of the desired value within the string.

MOVS

The MOVS instruction is used to transfer a byte or word from the source location at DS:SI to the destination location at ES:DI. After the transfer, both SI and DI are updated to point to the next memory locations. MOVS is primarily used to move blocks of memory from one place to another.

The direction flag (DF) is crucial when dealing with overlapping memory blocks. If the source and destination memory blocks overlap, the direction flag determines the direction in which the copy operation is performed. This is important to avoid corrupting the source data.

Here's how the direction flag affects MOVS:

1. **DF Clear (Auto-Increment Mode):** If the direction flag is cleared, the copy operation moves from lower addresses to higher addresses. This mode is suitable when the destination block is above the source block and prevents data corruption.
2. **DF Set (Auto-Decrement Mode):** If the direction flag is set, the copy operation moves from higher addresses to lower addresses. This mode is useful when the destination block is below the source block, preventing data corruption.

CMPS

The CMPS instruction is used to compare two blocks of memory, specifically the block at source location DS:SI and the block at destination location ES:DI. CMPS performs a byte-by-byte or word-by-word comparison of these two memory blocks without altering the source or destination data. It updates SI and DI accordingly.

CMPS is frequently used for comparing two blocks of memory to determine their equality or inequality. This instruction subtracts corresponding bytes or words from the source and destination and checks the result. If used with the REPE (repeat while equal) or REPNE (repeat while not equal) prefix, CMPS repeats the comparison operation as long as the blocks are equal or as long as they are different.

One common use of CMPS is to find a substring within a larger string. A substring is a smaller string contained within a larger one. For example, you can use CMPS to search for the substring "has" within the string "Mary has a little lamp." By employing CMPS with appropriate prefixes, you can perform the operation of a complex loop in a single instruction.

REP Prefix

REP repeats the following string instruction CX times. The use of CX is implied with the REP prefix. The decrement in CX doesn't affect any flags and the jump is also independent of the flags.

REPE and REPNE Prefixes

REPE or REPZ repeat the following string instruction while the zero flag is set and REPNE or REPNZ repeat the following instruction while the zero flag is not set. REPE or REPNE are used with the SCAS or CMPS instructions.

STOS EXAMPLE – CLEARING THE SCREEN

```
[org 0x0100]
jmp start

; Subroutine to clear the screen
clrscr:
    push es
    push ax
    push cx
    push di

    mov ax, 0xb800
    mov es, ax   ; Point es to video base
    xor di, di   ; Point di to top left column
    mov ax, 0x0720   ; Space char in normal attribute
    mov cx, 2000     ; Number of screen locations
    cld              ; Auto increment mode
    rep stosw        ; Clear the whole screen

    pop di
    pop cx
    pop ax
    pop es
    ret

start:
    call clrscr     ; Call clrscr subroutine
    mov ax, 0x4c00  ; Terminate program
    int 0x21
```

LODS EXAMPLE – STRING PRINTING

```
[org 0x0100]
jmp start

message: db 'hello world'   ; String to be printed
length: dw 11               ; Length of the string

; Subroutine to print a string
; Takes the x position, y position, attribute, address of string, and
; its length as parameters
printstr:
    push bp
    mov bp, sp
    push es
    push ax
    push cx
    push si
    push di

    mov ax, 0xb800
    mov es, ax         ; Point es to video base
    mov al, 80          ; Load al with columns per row
    mul byte [bp+10]   ; Multiply with y position
    add ax, [bp+12]    ; Add x position
    shl ax, 1          ; Turn into byte offset
    mov di, ax         ; Point di to required location
    mov si, [bp+6]     ; Point si to string
    mov cx, [bp+4]     ; Load length of string in cx
    mov ah, [bp+8]     ; Load attribute in ah

    cld                ; Auto increment mode

nextchar:
    lodsb              ; Load next char into al
    stosw              ; Print char/attribute pair
    loop nextchar      ; Repeat for the whole string

    pop di
    pop si
    pop cx
    pop ax
    pop es
    pop bp
    ret 10

start:
    call clrscr         ; Call the clrscr subroutine
    mov ax, 30
    push ax             ; Push x position
    mov ax, 20
    push ax             ; Push y position
    mov ax, 1           ; Blue on black attribute
    push ax             ; Push attribute
    mov ax, message
    push ax             ; Push address of message
    push word [length]  ; Push message length
    call printstr       ; Call the printstr subroutine
    mov ax, 0x4c00      ; Terminate program
    int 0x21
```

SCAS EXAMPLE – STRING LENGTH

```
[org 0x0100]
jmp start

message: db 'hello world', 0  ; Null-terminated string
```

```
; Subroutine to print a string
; Takes the x position, y position, attribute, and address of a null-terminated
printstr:
    push bp
    mov bp, sp
    push es
    push ax
    push cx
    push si
    push di
    push ds
    pop es          ; Load ds into es
    mov di, [bp+4]  ; Point di to string
    mov cx, 0xffff  ; Load maximum number in cx
    xor al, al      ; Load a zero in al
    repne scasb     ; Find zero in the string
    mov ax, 0xffff  ; Load maximum number in ax
    sub ax, cx      ; Find change in cx
    dec ax          ; Exclude null from length
    jz exit         ; No printing if the string is empty
    mov cx, ax      ; Load string length into cx
    mov ax, 0xb800
    mov es, ax      ; Point es to video base
    mov al, 80      ; Load al with columns per row
    mul byte [bp+8] ; Multiply with y position
    add ax, [bp+10] ; Add x position
    shl ax, 1       ; Turn into a byte offset
    mov di, ax      ; Point di to the required location
    mov si, [bp+4]  ; Point si to the string
    mov ah, [bp+6]  ; Load attribute into ah
    cld             ; Auto increment mode

nextchar:
    lodsb           ; Load the next character into al
    stosw           ; Print char/attribute pair
    loop nextchar   ; Repeat for the whole string

exit:
    pop di
    pop si
    pop cx
    pop ax
    pop es
    pop bp
    ret 8

start:
    call clrscr         ; Call the clrscr subroutine
    mov ax, 30
    push ax             ; Push x position
    mov ax, 20
    push ax             ; Push y position
    mov ax, 1           ; Blue on black attribute
    push ax             ; Push attribute
    mov ax, message
    push ax             ; Push address of the message
    call printstr       ; Call the printstr subroutine
    mov ax, 0x4c00      ; Terminate the program
    int 0x21
```

LES and LDS Instructions

The LES and LDS instructions are used to load a segment register and a general-purpose register from two consecutive memory locations. Here's how they work:

1. **LES (Load ES from memory)**:
   - **Syntax**: `LES destination, source`
   - The destination is a general-purpose register (e.g., SI, DI).
   - The source is the memory location containing a segment-offset pair.
   - The word at the higher memory address is loaded into ES, and the word at the lower memory address is loaded into the specified destination register.
   - Example: `LES SI, [BP+4]` loads SI from the memory location specified by BP+4 and loads ES from BP+6.
2. **LDS (Load DS from memory)**:
   - **Syntax**: `LDS destination, source`
   - Similar to LES, but it loads DS instead of ES.
   - The word at the higher memory address is loaded into DS, and the word at the lower memory address is loaded into the specified destination register.
   - Example: `LDS DI, [BX+2]` loads DI from the memory location specified by BX+2 and loads DS from BX+4.

These instructions are particularly useful when a subroutine receives a segment-offset pair as an argument, and you need to load them into the appropriate registers for further processing.

LES AND LDS EXAMPLE

```
[org 0x0100]
jmp start

message: db 'hello world', 0 ; Null-terminated string

; Subroutine to calculate the length of a string
; Takes the segment and offset of a string as parameters
strlen:
    push bp
    mov bp, sp
    push es
    push cx
    push di
    les di, [bp+4]    ; Point es:di to string
    mov cx, 0xffff    ; Load maximum number in cx
    xor al, al        ; Load a zero in al
    repne scasb        ; Find zero in the string
    mov ax, 0xffff    ; Load maximum number in ax
    sub ax, cx        ; Find change in cx
    dec ax            ; Exclude null from length
    pop di
    pop cx
    pop es
    pop bp
    ret 4

; Subroutine to print a string
; Takes the x position, y position, attribute, and address of a null-terminated
printstr:
    push bp
    mov bp, sp
    push es
    push ax
    push cx
    push si
    push di
    push ds              ; Push segment of string
    mov ax, [bp+4]
    push ax              ; Push offset of string
    call strlen          ; Calculate string length
    cmp ax, 0            ; Is the string empty
    jz exit              ; No printing if the string is empty
    mov cx, ax           ; Save length in cx
    mov ax, 0xb800
    mov es, ax              ; Point es to the video base
    mov al, 80              ; Load al with columns per row
    mul byte [bp+8]         ; Multiply with y position
    add ax, [bp+10]         ; Add x position
    shl ax, 1               ; Turn into a byte offset
    mov di, ax              ; Point di to the required location
    mov si, [bp+4]          ; Point si to the string
    mov ah, [bp+6]          ; Load attribute into ah
    cld                     ; Auto increment mode

nextchar:
    lodsb                   ; Load the next char into al
    stosw                   ; Print char/attribute pair
    loop nextchar           ; Repeat for the whole string

exit:
    pop di
    pop si
    pop cx
    pop ax
    pop es
    pop bp
    ret 8

start:
    call clrscr                ; Call the clrscr subroutine
    mov ax, 30
    push ax                    ; Push x position
    mov ax, 20
    push ax                    ; Push y position
    mov ax, 0x71               ; Blue on white attribute
    push ax                    ; Push attribute
    mov ax, message
    push ax                    ; Push address of message
    call printstr              ; Call the printstr subroutine
    mov ax, 0x4c00             ; Terminate the program
    int 0x21
```

MOVS EXAMPLE – SCREEN SCROLLING

```
[org 0x0100]
jmp start

; Subroutine to scroll up the screen
; Takes the number of lines to scroll as a parameter
scrollup:
    push bp
    mov bp, sp
    push ax
    push cx
    push si
    push di
    push es
    push ds

    ; Calculate the source position
    mov ax, 80              ; Load characters per row in ax
    mul byte [bp+4]         ; Calculate source position
    mov si, ax              ; Load source position in si
    push si                 ; Save position for later use
    shl si, 1               ; Convert to byte offset
    mov cx, 2000            ; Number of screen locations
    sub cx, ax              ; Count of words to move
    mov ax, 0xb800
    mov es, ax              ; Point es to video base
    mov ds, ax              ; Point ds to video base
    xor di, di              ; Point di to the top left column
    cld                     ; Set auto increment mode
    rep movsw               ; Scroll up

    ; Clear the scrolled space with spaces in normal attribute
    mov ax, 0x0720
    pop cx                   ; Count of positions to clear
    rep stosw
    pop ds
    pop es
    pop di
    pop si
    pop cx
    pop ax
    pop bp
    ret 2

; Subroutine to scroll down the screen
; Takes the number of lines to scroll as a parameter
scrolldown:
    push bp
    mov bp, sp
    push ax
    push cx
    push si
    push di
    push es
    push ds

    ; Calculate the source position
    mov ax, 80              ; Load characters per row in ax
    mul byte [bp+4]         ; Calculate source position
    push ax                 ; Save position for later use
    shl ax, 1               ; Convert to byte offset
    mov si, 3998            ; Last location on the screen
    sub si, ax              ; Load source position in si
    pop ax
    push ax
    mov cx, 2000            ; Number of screen locations
    sub cx, ax              ; Count of words to move
    mov ax, 0xb800
    mov es, ax              ; Point es to video base
    mov ds, ax              ; Point ds to video base
    mov di, 3998            ; Point di to the lower right column
    std                     ; Set auto decrement mode
    rep movsw               ; Scroll down

    ; Clear the scrolled space with spaces in normal attribute
    mov ax, 0x0720
    pop cx                   ; Count of positions to clear
    rep stosw
    pop ds
    pop es
    pop di
    pop si
    pop cx
    pop ax
    pop bp
    ret 2

start:
    mov ax, 5
    push ax                  ; Push the number of lines to scroll
    call scrollup            ; Call the scrollup subroutine

    ; You can also use the following line to scroll down instead of scrolling up
    ; call scrolldown
    mov ax, 0x4c00          ; Terminate the program
    int 0x21
```

Explanation:

1. The code is organized into two main subroutines: `scrollup` for scrolling up and `scrolldown` for scrolling down.
2. In the `scrollup` subroutine:
   - The number of lines to scroll is passed as a parameter on the stack and retrieved using the `bp` register.
   - The source position is calculated based on the number of lines to scroll.
   - The source and destination addresses are set up to define the region of the screen to scroll.
   - The direction flag (`cld`) is set to auto-increment mode.
   - The REP prefix is used with `movsw` to efficiently move the memory block upward.
   - After scrolling, the scrolled region is cleared with spaces in normal attribute.
3. In the `scrolldown` subroutine:
   - Similar to `scrollup`, this subroutine calculates the source position and sets up source and destination addresses.
   - The direction flag is set to auto-decrement mode (`std`) since scrolling down requires copying from the bottom to the top.
   - The REP prefix is used with `movsw` to move the memory block downward.
   - The scrolled region is cleared with spaces in normal attribute.
4. The `start` section:
   - It calls the `scrollup` subroutine to perform the scrolling operation.
   - You can comment out the `call scrollup` line and uncomment the `call scrolldown` line if you want to scroll down instead.
   - Finally, the program terminates using `int 0x21` with an exit code.

CMPS EXAMPLE – STRING COMPARISON

```
[org 0x0100]
jmp start

msg1: db 'hello world', 0
msg2: db 'hello WORLD', 0
msg3: db 'hello world', 0

; Subroutine to compare two strings
; Takes segment and offset pairs of two strings to compare
```

```
strcmp:
    push bp
    mov bp, sp
    push cx
    push si
    push di
    push es
    push ds

    lds si, [bp+4]   ; Point ds:si to the first string
    les di, [bp+8]   ; Point es:di to the second string

    push ds          ; Push the segment of the first string
    push si          ; Push the offset of the first string
    call strlen      ; Calculate the length of the first string
    mov cx, ax       ; Save the length in cx

    push es          ; Push the segment of the second string
    push di          ; Push the offset of the second string
    call strlen      ; Calculate the length of the second string
    cmp cx, ax       ; Compare the lengths of both strings
    jne exitfalse    ; If they are unequal, return 0

    add cx, 1
    mov ax, 1        ; Store 1 in ax to be returned
    repe cmpsb       ; Compare both strings
    jcxz exitsimple  ; If they are successfully compared, jump to exitsimple

exitfalse:
    mov ax, 0        ; Store 0 to mark them as unequal

exitsimple:
    pop ds
    pop es
    pop di
    pop si
    pop cx
    pop bp
    ret 8

start:
    push ds          ; Push the segment of the first string
    mov ax, msg1
    push ax          ; Push the offset of the first string
    push ds          ; Push the segment of the second string
    mov ax, msg2
    push ax             ; Push the offset of the second string
    call strcmp         ; Call the strcmp subroutine

    push ds             ; Push the segment of the first string
    mov ax, msg1
    push ax             ; Push the offset of the first string
    push ds             ; Push the segment of the third string
    mov ax, msg3
    push ax             ; Push the offset of the third string
    call strcmp         ; Call the strcmp subroutine

    mov ax, 0x4c00      ; Terminate the program
    int 0x21
```

Explanation:

1. The code starts by defining three null-terminated strings (`msg1`, `msg2`, and `msg3`).
2. The `strcmp` subroutine is designed to compare two strings. It does so by comparing the lengths of the strings first and then comparing the characters one by one.
3. The subroutine takes the segment and offset pairs of two strings to compare.
4. It first calculates the length of both strings using the `strlen` subroutine, which calculates the length of null-terminated strings.
5. If the lengths are different, it exits with a return value of 0, indicating that the strings are not equal.
6. If the lengths are the same, it uses the `repe cmpsb` instruction to compare the strings character by character. If they match, it sets the return value (`ax`) to 1; otherwise, it remains 0.
7. The `start` section demonstrates the use of the `strcmp` subroutine to compare the strings `msg1` with `msg2` and `msg1` with `msg3`.
8. Finally, the program terminates with `int 0x21`.