String Processing

Chapter 10

S. Dandamudi

Outline

- String representation
  - Using string length
  - Using a sentinel character
- String instructions
  - Repetition prefixes
  - Direction flag
  - String move instructions
  - String compare instructions
  - String scan instructions
- Illustrative examples
  - LDS and LES instructions
- Examples
  - str_len
  - str-cpy
  - str_cat
  - str_cmp
  - str_chr
  - str_cnv
- Indirect procedure call
- Performance: Advantage of string instructions

String Representation

- Two types
  - Fixed-length
  - Variable-length
- Fixed length strings
  - Each string uses the same length
    - Shorter strings are padded (e.g. by blank characters)
    - Longer strings are truncated
  - Selection of string length is critical
    - Too large ==> inefficient
    - Too small ==> truncation of larger strings

String Representation (cont'd)

- Variable-length strings
  - Avoids the pitfalls associated with fixed-length strings
- Two ways of representation
  - Explicitly storing string length (used in PASCAL)

```
string      DB     'Error message'
str_len     DW     $-string
```

    - $ represents the current value of the location counter
      - $ points to the byte after the last character of **`string`**
  - Using a sentinel character (used in C)
    - Uses NULL character
      - Such NULL-terminated strings are called *ASCIIZ strings*

String Instructions

- Five string instructions

```
LODS   LOaD String        source
STOS   STOre String  destination
MOVS   MOVe String        source & destination
CMPS   CoMPare String     source & destination
SCAS   SCAn String        destination
```

- Specifying operands
  - 32-bit segments:

    DS:ESI = source operand ES:EDI = destination operand
  - 16-bit segments:

    DS:SI = source operand ES:DI = destination operand

String Instructions (cont'd)

- Each string instruction
  - Can operate on 8-, 16-, or 32-bit operands
  - Updates index register(s) automatically
    - Byte operands: increment/decrement by 1
    - Word operands: increment/decrement by 2
    - Doubleword operands: increment/decrement by 4
- Direction flag
  - DF = 0: Forward direction (increments index registers)
  - DF = 1: Backward direction (decrements index registers)
- Two instructions to manipulate DF

  `std` set direction flag (DF = 1)

  `cld` clear direction flag (DF = 0)

Repetition Prefixes

- String instructions can be repeated by using a repetition prefix
- Two types
  - Unconditional repetition

    `rep` REPeat
  - Conditional repetition

    `repe/repz` REPeat while Equal
    REPeat while Zero

    `repne/repnz` REPeat while Not Equal
    REPeat while Not Zero

Repetition Prefixes (cont'd)

rep

while (ECX ≠ 0)
execute the string instruction
ECX := ECX–1
end while

- ECX register is first checked
  - If zero, string instruction is not executed at all
  - More like the **JECXZ** instruction

Repetition Prefixes (cont'd)

`repe/repz`

while (ECX ≠ 0)
    execute the string instruction
    ECX := ECX–1
    if (ZF = 0)
    then
      exit loop
    end if
end while

- Useful with **`cmps`** and **`scas`** string instructions

Repetition Prefixes (cont'd)

`repne/repnz`

while (ECX ≠ 0)
    execute the string instruction
    ECX := ECX–1
    if (ZF = 1)
    then
      exit loop
    end if
end while

String Move Instructions

- Three basic instructions
  movs, lods, and stos
- Move a string (movs)
- Format

```
movs    dest_string,source_string
movsb        ; operands are bytes
movsw        ; operands are words
movsd        ; operands are doublewords
```

- First form is not used frequently
  - Source and destination are assumed to be pointed by DS:ESI and ES:EDI, respectively

String Move Instructions (cont'd)

movsb --- move a byte string

```
ES:EDI:= (DS:ESI)    ; copy a byte
if (DF=0)            ; forward direction
then
        ESI := ESI+1
        EDI := EDI+1
else                 ; backward direction
        ESI := ESI–1
        EDI := EDI–1
end if
```

Flags affected: none

String Move Instructions (cont'd)

Example

```
.DATA
string1     db     'The original string',0
strLen      EQU    $ - string1
.UDATA
string2     resb     80
.CODE
     .STARTUP
     mov     AX,DS              ; set up ES
     mov     ES,AX              ;  to the data segment
     mov     ECX,strLen         ; strLen includes NULL
     mov     ESI,string1
     mov     EDI,string2
     cld                        ; forward direction
     rep     movsb
```

String Move Instructions (cont'd)

Load a String (LODS)

- Copies the value from the source string at DS:ESI to
  - AL (**`lodsb`**)
  - AX (**`lodsw`**)
  - EAX (**`lodsd`**)

- Repetition prefix does not make sense
  - It leaves only the last value in AL, AX, or EAX register

String Move Instructions (cont'd)

`lodsb` --- load a byte string

```
AL := (DS:ESI)          ; copy a byte
if (DF=0)               ; forward direction
then
        ESI := ESI+1
else                    ; backward direction
        ESI := ESI–1
end if
```

Flags affected: none

String Move Instructions (cont'd)

Store a String (STOS)

- Performs the complementary operation
- Copies the value in
  - AL (**`lodsb`**)
  - AX (**`lodsw`**)
  - EAX (**`lodsd`**)

  to the destination string at ES:EDI

- Repetition prefix can be used if you want to initialize a block of memory

String Move Instructions (cont'd)

stosb --- store a byte string

```
ES:EDI := AL  ; copy a byte
if (DF=0)             ; forward direction
then
        EDI := EDI+1
else                  ; backward direction
        EDI := EDI–1
end if
```

Flags affected: none

String Move Instructions (cont'd)

Example: Initializes **`array1`** with -1

```
.UDATA
array1    resw    100
.CODE
    .STARTUP
    mov     AX,DS           ; set up ES
    mov     ES,AX           ;  to the data segment
    mov     ECX,100
    mov     EDI,array1
    mov     AX,-1
    cld                     ; forward direction
    rep     stosw
```

String Move Instructions (cont'd)

- In general, repeat prefixes are not useful with **`lods`** and **`stos`**
- Used in a loop to do conversions while copying

```
        mov     ECX,strLen
        mov     ESI,string1
        mov     EDI,string2
        cld                         ; forward direction
    loop1:
        lodsb
        or      AL,20H
        stosb
        loop    loop1
    done:
```

String Compare Instruction

cmpsb --- compare two byte strings

Compare two bytes at DS:ESI and ES:EDI and set flags

```
if (DF=0)              ; forward direction
then
        ESI := ESI+1
        EDI := EDI+1
else                   ; backward direction
        ESI := ESI–1
        EDI := EDI–1
end if
```

Flags affected: As per **cmp** instruction (DS:ESI)–(ES:EDI)

String Compare Instruction (cont'd)

```
.DATA
string1     db     'abcdfghi',0
strLen      EQU    $ - string1
string2     db     'abcdefgh',0
.CODE
    .STARTUP
    mov     AX,DS              ; set up ES
    mov     ES,AX              ;  to the data segment
    mov     ECX,strLen
    mov     ESI,string1
    mov     EDI,string2
    cld                        ; forward direction
    repe    cmpsb
    dec     ESI
    dec     EDI     ; ESI & EDI pointing to the last character that differs
```

String Compare Instruction (cont'd)

```
.DATA
string1     db     'abcdfghi',0
strLen      EQU    $ - string1 - 1
string2     db     'abcdefgh',0
.CODE
     .STARTUP
     mov    AX,DS              ; set up ES
     mov    ES,AX              ;  to the data segment
     mov    EECX,strLen
     mov    ESI,string1 + strLen - 1
     mov    EDI,string2 + strLen - 1
     std                       ; backward direction
     repne  cmpsb
     inc    ESI  ; ESI & EDI pointing to the first character that matches
     inc    EDI  ; in the backward direction
```

String Scan Instruction

scasb --- Scan a byte string

```
Compare AL to the byte at ES:EDI & set flags
if (DF=0)                ; forward direction
then
        EDI := EDI+1
else                     ; backward direction
        EDI := EDI–1
end if
```

Flags affected: As per **cmp** instruction (DS:ESI)–(ES:EDI)

- **scasw** uses AX and **scasd** uses EAX instead of AL

String Scan Instruction (cont'd)

Example 1

```
.DATA
string1     db     'abcdefgh',0
strLen      EQU    $ - string1
.CODE
     .STARTUP
     mov     AX,DS          ; set up ES
     mov     ES,AX          ;  to the data segment
     mov     ECX,strLen
     mov     EDI,string1
     mov     AL,'e'         ; character to be searched
     cld                    ; forward direction
     repne   scasb
     dec     EDI            ; leaves EDI pointing to e in string1
```

String Scan Instruction (cont'd)

Example 2

```
.DATA
string1     db     '       abc',0
strLen      EQU    $ - string1
.CODE
     .STARTUP
     mov     AX,DS          ; set up ES
     mov     ES,AX          ;  to the data segment
     mov     ECX,strLen
     mov     EDI,string1
     mov     AL,' '         ; character to be searched
     cld                    ; forward direction
     repe    scasb
     dec     EDI          ; EDI pointing to the first non-blank character a
```

Illustrative Examples

LDS and LES instructions

- String pointer can be loaded into DS/SI or ES/DI register pair by using `lds` or `les` instructions
- Syntax

```
lds     register,source
les     register,source
```

  - **`register`** should be a 32-bit register
  - **`source`** is a pointer to a 48-bit memory operand
    - **`register`** is typically ESI in `lds` and EDI in `les`

Illustrative Examples (cont'd)

- Actions of **lds** and **les**

```
lds
   register := (source)
         DS := (source+4)
les
   register := (source)
         ES := (source+4)
```

- Pentium also supports **lfs**, **lgs**, and **lss** to load the other segment registers

Illustrative Examples (cont'd)

- Seven popular string processing routines are given as examples
  - **`str_len`**
  - **`str-cpy`**
  - **`str_cat`**
  - **`str_cmp`**
  - **`str_chr`**
  - **`str_cnv`**

Indirect Procedure Call

- Direct procedure calls specify the offset of the first instruction of the called procedure
- In indirect procedure call, the offset is specified through memory or a register
  - If BX contains pointer to the procedure, we can use

```
call    EBX
```

  - If the word in memory at **`target_proc_ptr`** contains the offset of the called procedure, we can use

```
call    [target_proc_ptr]
```

- These are similar to direct and indirect jumps

Performance: Advantage of String Instructions

- Two chief advantages of string instructions
  - Index registers are automatically updated
  - Can operate two operands in memory
- Example: Copy data from **`array1`** to **`array2`**

```
cld
rep    movsd
```

  - Assumes:

    DS:ESI points to **`array1`**

    ES:EDI points to **`array2`**

    ECX contains the array size

Performance: Advantage of String Instructions (cont'd)

50,000-element array-to-array copy