
Lecture 7: Arrays and Strings

Dragan Mirkovic
Department of Computer Science
University of Houston

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Announcements

- Quiz #4 on Tuesday 3/30/2004
 - Subroutines Ch. 6 & 13
- Today we will start the arrays and strings
 - Chapters 10 and 17 in text

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Introduction

- Array: a contiguous collection of entries of the same length in memory
 - A simple definition of a possibly large amount of data
 - Simplified addressing of array elements: $A[j]$
 - Understanding of addressing is crucial
- We have already seen how to use pointers in the previous lectures and labs
- In this lecture we will give more details on the pointer arithmetic and the use of pointers in assembly language
- The ideas are similar to C/C++ pointers
 - In fact C/C++ pointers were designed to mimic the assembly language (or hardware)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Addressing

- In high-level languages, an identifier usually refers to the contents of a memory location
- In C if we define `int A`, and then use `x = A`; we are referring to the value of the variable `A`
- In order to specify the address of a variable we can use the address operator `&`:
 - `p = &A;` `p` contains the address of the variable `A`
 - The value changes in the course of the program, but the address `&A` does not;
- In assembly language
 - `A DB 24`
 - Defines a memory location named `A`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Address operator

- Assembler equivalent to '&' operator in C/C++?
- Depends on the memory model used
 - Segmented addresses require two operators: **SEG** and **OFFSET**
- Examples:
 - When you specify: `A DW 0A0Bh`;
 - Assembler will assign a 2 byte memory location to the symbol **A**, for example: `DS:0008,9` or `0B8E:0008,9`.
 - The location 0008 will contain 0B and 0009 will contain 0A (little-endian byte order).

```
0B8E:0000 04 00 B8 08 00 B8 02 00-0B 0A 08 00 09 00 90 08
```

- Similarly: `B DD 01020304h`; will result in:

```
0B8E:0000 04 00 B8 08 00 B8 02 00-0B 0A 04 03 02 01 90 08
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Address operator examples

- Examples:
 - The command: `mov ax, A`; will result in: `ah = 0A, al = 0B`.

```
0B8C:0010 A10800 MOV AX,[0008] DS:0008=0A0B
AX=0A0B
```

- The command: `mov ax, OFFSET A`; will result in:
`ax = OFFSET` part of the address of **A**, `ah = 00, al = 08`.

```
0B8C:0013 B80800 MOV AX,0008
AX=0008
```

- The command: `mov ax, SEG A`; will result in:
`ax = SEGMENT` part of the address of **A**,
`ah = 0B, al = 8E`.

```
0B8C:0016 B88E0B MOV AX,0B8E
AX=0B8E
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

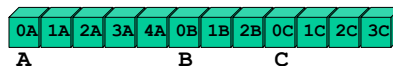
Address arithmetic

- In high-level languages we can use arrays to refer to the data of the same type by a single name and a numeric index:
`A[3], A[17], C[i][j],...`
- In assembly:
`A DB 0Ah, 1Ah, 2Ah, 3Ah, 4Ah`
A contains 0Ah, A+1 contains 1Ah,
A+2 contains 2Ah, etc.
- What is the pointer operator in assembly?
- [a] denotes the the content of the memory at the address a, equivalent to *a in C/C++
- Caution:
`mov al, A + 3; sets al to [A + 3] = 3Ah`
and NOT to [A] + 3 = 0Dh!!

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Address arithmetic example

- Example:
A DB 0Ah, 1Ah, 2Ah, 3Ah, 4Ah
B DB 0Bh, 1Bh, 2Bh
C DB 0Ch, 1Ch, 2Ch, 3Ch



- The commands:
`mov al, [C-1]`
`mov al, [B+2]`
`mov al, [A+7]`
- Result in: `al = 02Bh`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Rules for Address Expressions

- Basic rules:
 - The arithmetic operations on addresses are performed by the assembler.
 - All the values should be known at the assembly time.
- There is expressions may contain a large number of operators (+, -, (), [], *, /, OFFSET, SEG,...) and symbolic names (A, B, Myname, Whatever, ...).
 - Assembler will perform the calculations and replace the expression with a single result.
- Examples: Let


```
A DW      0a0bh
B DD      01020304h
D DW      9
```

SEG:OFST	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0B8F:0000	00	00	0B	0A	04	03	02	01	09	00	02	01	09	00	13	2E
	A				B				D							

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Examples

Command	Result	
mov ax, A;	MOV AX, [0002]	AX=0A0B
mov ax, SEG A	MOV AX, 0B8F	AX=0B8F
mov ax, OFFSET A	MOV AX, 0002	AX=0002
mov ax, OFFSET B	MOV AX, 0004	AX=0004
mov ax, OFFSET D	MOV AX, 0008	AX=0008
mov ax, A + 1	MOV AX, [0003]	AX=040A
mov ax, B - A	MOV AX, 0002	AX=0002
mov ax, D - A	MOV AX, 0006	AX=0006
mov ax, A - D	MOV AX, FFFA	AX=FFFA
mov ax, 2*(D - A)	MOV AX, 000C	AX=000C
mov ax, A + (D - A)/2	MOV AX, [0005]	AX=0203

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Byte order

- IBM PC is little-endian:
 - The least significant bytes (little end) is stored 1st (at lower addresses).
 - Big-endian machines (Risc, Sun, HP PA, ...) use the opposite order.
- Problems when using binary files on the machines with different byte order
 - Reading a binary file with data saved in the wrong order requires a byte swapping
 - Java Virtual Machine is big-endian. On a little- endian machine data bytes need to be swapped.
 - From 80486 up there is a `bswap` instruction:
`bswap reg32` will swap bytes in a 32-bit register

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Example

- If `B` is defined as

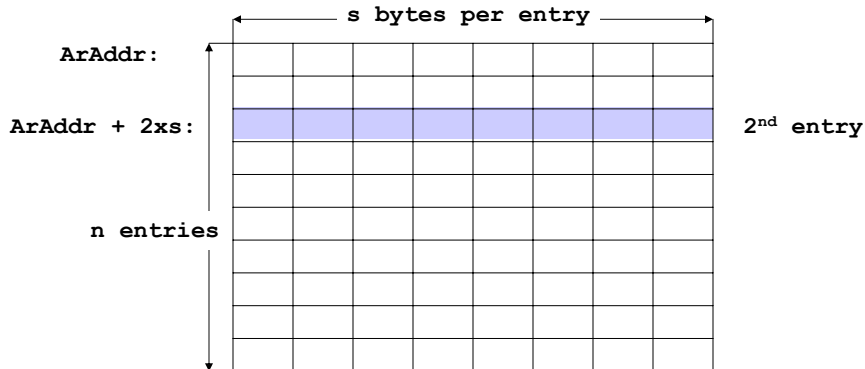
```
B DD 01020304h
mov eax, B
bswap eax
```
- Will result in:

```
0A47:0010 66A10E00 MOV EAX, DWORD PTR [000E]
EAX = 01020304
0A47:0014 660FC8 BSWAP EAX
EAX = 04030201
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Arrays

- Array: a contiguous collection of entries of the same length in memory



D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Arrays...

- In assembly, it is natural to use **0-origin indexing** similar to C/C++ and Java.
- The location of the i th entry is given by
 $\langle \text{array address} \rangle + i \times s$
- You can declare an uninitialized array **A** of **N** elements each being **s** bytes long using

```
N EQU ...
s EQU ...
A DB N DUP (s DUP (?)) ; or N*s DUP (?)
```
- In high-level languages $A[i]$ denotes the i th element of the array ($A[i] = [A + i*s]$)
- In assembler $A[i] = [A + i]$ is the i th byte in the array
 - **The user must take into account the element size!!**

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Using Arrays

1. Constant subscripts:

- First define three different arrays:

```
A DB 8 DUP (0A)
B DW 8 DUP (0B1Bh)
D DD 8 DUP (0D1D2D3Dh)
```

```
0B93:0000 0A 0A 0A 0A 0A 0A 0A 0A 1B 0B 1B 0B 1B 0B 1B 0B
0B93:0010 1B 0B 1B 0B 1B 0B 1B 0B 3D 2D 1D 0D 3D 2D 1D 0D
0B93:0020 3D 2D 1D 0D 3D 2D 1D 0D 3D 2D 1D 0D 3D 2D 1D 0D
0B93:0030 3D 2D 1D 0D 3D 2D 1D 0D 00 00 00 00 00 00 00 00
```

- Next, we initialize a few elements in them

```
mov [A + 2], 2;      A[2] = 2
mov [B + 2*3], 3;    B[3] = 3
mov [D + 4*5], 5;    D[5] = 5
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Examples

- Here are the results:

```
0B93:0000 0A 0A 02 0A 0A 0A 0A 0A 1B 0B 1B 0B 1B 0B 03 00
0B93:0010 1B 0B 1B 0B 1B 0B 1B 0B 3D 2D 1D 0D 3D 2D 1D 0D
0B93:0020 3D 2D 1D 0D 3D 2D 1D 0D 3D 2D 1D 0D 05 00 00 00
0B93:0030 3D 2D 1D 0D 3D 2D 1D 0D 00 00 00 00 00 00 00 00
```

- Variable subscripts:
 - Require addressing of the form $[A + s*[n]]$
 - It is necessary to use registers, ex. **bx**, **bp**, **si**, and **di** (index registers).
 - Only these registers can be used as index registers in assembly instructions
 - You should use **bp** for stack indexing only!

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Variable subscripts

- The possible form of the operand is

$$[\text{<address expression> + } \left. \begin{array}{c} \text{bx} \\ \text{bp} \\ \text{si} \\ \text{di} \end{array} \right\}]$$

- Example:
[bx], [A + si], [B - 5 + di], [2 + bp], [D + 4*3 + bx], ...
are correct, but:
[A + 4*di] is not!
- Example. Let B be an array of bytes and N a word variable. Write a code to set B[N] = 27 and B[N+3] = 11.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Solution

```
mov    di, N
mov    [B + di], 27;    B[N] = 27
add    di, 3
mov    [B + di], 11;    B[N+3] = 11
```

- Or

```
mov    di, N
mov    [B + 3 + di], 11; B[N+3] = 11
```
- Note that in the 2nd solution, the addition is done at assembly time.
- Now if B is an array of words we have to take into account the size of each array element!

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Example

- Example. Let W be an array of words and N a word variable. Write a code to set $W[N+3] = 27$.
- Solution:
 - Each entry is 2 bytes long and $W[N+3]$ starts at $W + ([N] + 3) * 2$.
 - We can compute in an index register and then use $[W + bx]$ to set the value.
 - Multiplication, however needs ax and we have to write something like

```
mov ax, N
add ax, 3
mov bx, 2
mul bx
mov [W + ax], 27
```
 - This is too complicated! There is an easier way to multiply (and divide by 2 in the binary system).

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Shift operations

- Shifting of the contents of a double word, word or byte left or right is equivalent to multiplying or dividing by powers of two respectively.
- The 80x86 processors have instructions to perform byte shifts:

```
shl/shr mem/reg, cl/const
```

 - You can shift either register or a memory location
 - The amount of shift is either constant or a variable in cl .
- Example:

```
mov ax, factor;    load factor into ax
shl ax, 1;         ax = 2*factor
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Solution using shl

```
mov  bx, N
add  bx, 3
shl  bx, 1;          bx = 2*(N+3)
mov  [B + bx], 27; W[N+3] = 27
```

- Or we can precalculate the addition part in the following way:

```
W + ([N] + 3) * 2 = W + 2 * 3 + 2 * [N]
mov  bx, N
shl  bx, 1;          bx = 2 * N
mov  [B + 2 * 3 + bx], 27; W[N+3] = 27
```

- Reminder: The notation $\mathbf{A}[i]$ is legal in the assembly language, but is usually different from in $\mathbf{A}[i]$ high-level languages and should be used with caution.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Arrays and loops

- Array elements are typically processed in a loop
- Example:
 - Write a code to set $w[n] = 0$, for $n = 0, \dots, 99$, where W is an array of 100 words.
- Solution:

```
mov  cx, 100
mov  di, 0
L0:  mov  [W + di], 0
      add  di, 2
      dec  cx
      jnz  L0
```

- Index method (di contains the index of W)
- Equivalent to $W[j] = 0$ in C/C++

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Pointer method

- Solution:

```
mov    cx, 100
mov    di, OFFSET W; di = pointer to W
L0:   mov    WORD PTR [di], 0;
      add    di, 2
      dec    cx
      jnz   L0
```

- This is similar to the C/C++ pointer method
- `di` contains the actual address of the `i`th element in `w`
- Equivalent to `*pW = 0` in C/C++

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Base indexing

- More general form of indexing:
[address-expression + {bx/bp} + {si/di}]
– **Base indexing** or double indexing.
- Example:
 - Array copy: `w` and `x` are arrays of 100 words. Write the code to copy `w` into `x`.

```
mov    cx, 100
mov    bx, 0
L0:   mov    ax, [W + bx]
      mov    [X + bx], ax
      add    bx, 2
      dec    cx
      jnz   L0

mov    cx, 100
mov    si, OFFSET W
mov    di, OFFSET X
L0:   mov    ax, [si]
      mov    [di], ax
      add    si, 2
      add    di, 2
      dec    cx
      jnz   L0
```

– equivalent to `X[i] = W[i]` in C/C++

– equivalent to `*pX = *pW` in C/C++

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Example

```
mov    cx, 100                mov    cx, 100
mov    si, OFFSET W           mov    si, OFFSET W
L0:    mov    ax, [si]         L0:    mov    bx, X - W
      mov    [X-W+si], ax     mov    ax, [si]
      add    si, 2            mov    [bx+si], ax
      dec    cx              add    si, 2
      jnz   L0               dec    cx
                                   jnz   L0
```

- Different methods will generate different code
- The 1st method generates the following instructions:

```
0B91:0016 8B870A00    MOV    AX, [BX+000A]    DS:000A=0A0B
0B91:001A 8987D200    MOV    [BX+00D2],AX    DS:00D2=0000
```

- Each instruction here takes 4 bytes

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Example...

- The 2nd method generates the following instructions:

```
0B91:0019 8B04                MOV    AX, [SI]        DS:000C=0A0B
0B91:001B 8905                MOV    [DI],AX        DS:00D4=0000
```

- Each instruction here takes only 2 bytes, but we need 2 adds

- The 3rd and the 4th method generate:

```
0B91:0016 8B04                MOV    AX, [SI]        DS:0008=0A0B
0B91:0018 8984C800           MOV    [SI+00C8],AX    DS:00D0=0000
```

- The 1st instruction is 2 bytes and the second is 4 bytes

```
0B91:0019 8B04                MOV    AX, [SI]        DS:000A=0A0B
0B91:001B 8900                MOV    [BX+SI],AX     DS:00D2=0000
```

- Each instruction here takes only 2 bytes

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Indexing with extended registers

- From 80386 on a powerful and flexible indexing was allowed using the extended registers
- The addressing combinations available are:

$$[\text{<address expression>} + \left. \begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esi} \\ \text{edi} \\ \text{esp} \\ \text{ebp} \end{array} \right\} + \left. \begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \end{array} \right\} * \left. \begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esi} \\ \text{edi} \\ \text{ebp} \end{array} \right\}]$$

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Indexing with extended registers

- Any of the 8 extended registers can be used as a base register
- Any of the extended registers except `esp` can be used as an index register
- The index register can be automatically multiplied by a scale factor of 2, 4, or 8 indicating the size of the array entry
- When running in 16 bit mode the final result should be number in the range 0-64k.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Array copy example

- Here we use the extended registers to copy the **w** array into the **x** array as before

```
mov    eax, 99
mov    ebx, OFFSET W
mov    ecx, OFFSET X
CopyLoop:
mov    edx, [ebx + 2 * eax]
mov    [ecx + 2 * eax], edx
dec    eax
jge    CopyLoop
```

- The scaling is included in the addressing instructions

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

The lea trick

- The load effective address instruction `lea` is of the form

```
lea    reg, mem;
```

- Address of `mem` is loaded into the register

- Similar to

```
mov    reg, OFFSET mem
```

- **Only** `lea` can obtain an address that is calculated at runtime. (It sets no flags)
- All standard address calculations are allowed for `mem`

- Examples:

```
lea    eax, [eax + 4];    eax += 4
lea    eax, [eax + 2 * eax];  eax *= 3
lea    eax, [eax + 4 * eax];  eax *= 5
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

String Operations

- A string is an array of characters.
- Initializing a string like "Helo", allocates and initializes a byte for each character in the string.
- An initialized string can be no longer than 255 characters.
- Initializers can span multiple lines, which is indicated by a comma at the end of the line.
- Declaring and referencing strings:

```
str1 BYTE "This is a long sting that does not",  
      "fit on one line."
```
- If the data directive is not **BYTE** a string may initialize only the first element, and the initializer value must fit into the specified size.

```
wstr WORD "OK" ; is fine.  
wstr WORD "BAD" ; doesn't fit.
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Useful Operators

- **LENGTHOF**, **SIZEOF**, **TYPE**,... can be useful when working with arrays and strings
- Examples:

```
array WORD 40 DUP (3)  
msg BYTE "This string extends",  
        "over three",  
        "lines."  
  
larray EQU LENGTHOF array; 40 elements  
sarray EQU SIZEOF array; 80 bytes  
tarray EQU TYPE array; 2 bytes/element  
lmsg EQU LENGTHOF msg; 37 elements  
smsg EQU SIZEOF msg; 37 bytes  
tmsg EQU TYPE msg; 1 bytes/element
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

String Processing Instructions

- The 80x86 architecture has a number of instructions specifically designed for processing strings of bytes and arrays of words or double words
- There are many common operations that need to be performed on a whole array:
 - Moving the array
 - Copying the array
 - Searching for a particular element in the array
- A typical array processing operation is

Aloop: ...

```
mov al, [si];  
inc si;  
...  
jmp Aloop
```

There is a single instruction
with the same effect:
lodsb: load string byte

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

The String Instructions

- There are five string operations:

<code>movsb/movsw/movsd</code>	move string (memory to memory)
<code>lodsb/lodsw/lodsd</code>	load string (memory to register)
<code>stosb/stosw/stosd</code>	store string (register to memory)
<code>cmpsb/cmpsw/cmpsd</code>	compare string (memory to memory)
<code>scasb/scasw/scasd</code>	scan string (memory to register)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

The String Instructions Properties

- All of them have the following characteristics:
 - 3 versions: byte, word and double word indicated by `b`, `w`, or `d`,
 - The register used is always `ax`, `eax` or `eax`.
 - The source is always `ds:[si]` (or `ds:[esi]` for 32-bit)
 - The destination is always `es:[di]` (or `es:[edi]`)
 - `si` and `di` are automatically incremented or decremented by 1 (byte), 2 (word), or 4 (dword) according to the `D` (direction) flag.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

The Direction Flag

- The `D` flag controls whether to increment or decrement `si` and/or `di`
- The `D` flag is normally 0 (auto increment)
- The `D` flag can be set and cleared by
 - `cld` ; Clear `D` flag = auto increment
 - `std` ; Set `D` flag = auto decrement
- Warning:
 - Many library routines assume the `D` flag is 0!
- Rules:
 - Auto Increment: `cld ; ...`
 - Auto Decrement: `pushf; std ; ... ; popf;`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Move String Operations

- Here is a detailed description of instructions for moving the string elements:

```
movsb { mov  es:[di], BYTE PTR ds:[si]
       { inc  si
       { inc  di

movsw { mov  es:[di], WORD PTR ds:[si]
       { add  si, 2
       { add  di, 2

movsd { mov  es:[di], DWORD PTR ds:[si]
       { add  si, 4
       { add  di, 4
```

;NOTE: mov memory to memory is illegal!

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Load String Operations

- There is a set of instructions on 80x86 for loading the string elements (bytes, words or dwords) into the accumulator

```
lodsb { mov  al, ds:[si]
       { inc  si

lodsw { mov  ax, ds:[si]
       { add  si, 2

lodsd { mov  eax, ds:[si]
       { add  si, 4
```

- In the 32-bit addressing mode the source is at address `ds:[esi]`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Store String Operations

- There is a set of instructions on 80x86 for storing the string elements (bytes, words or dwords) to memory

```
stosb      {  mov   es:[di], al
             {  inc   di
stosw      {  mov   es:[di], ax
             {  add   di, 2
stosd      {  mov   es:[di], eax
             {  add   di, 4
```

- In the 32-bit addressing mode the source is at address `ds:[esi]`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Compare String Instructions

- The set of instructions for comparison of the two strings of bytes, words or dwords is:

```
cmpsb      {  cmp   es:[di], BYTE PTR ds:[si]
             {  inc   si
             {  inc   di
cmpsw      {  cmp   es:[di], WORD PTR ds:[si]
             {  add   si, 2
             {  add   di, 2
cmpsd      {  cmp   es:[di], DWORD PTR ds:[si]
             {  add   si, 4
             {  add   di, 4
```

;NOTE: `cmp` memory to memory is illegal!

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Scan String Instructions

- There is a set of instructions on 80x86 for scanning the string elements (bytes, words or dwords) to find a value specified in the accumulator register

```
scasb      {  cmp    al, es:[di]
             {  inc    di
```

```
scasw      {  cmp    ax, es:[di]
             {  add    di, 2
```

```
scasd      {  cmp    eax, es:[di]
             {  add    di, 4
```

- In the 32-bit addressing mode the destination string is at address `es:[edi]`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Repeated Instructions

- There are various repeat (`rep`) instructions that can be added as a prefix to a string instruction
- It repeats the string instruction until some condition is satisfied
- Example:

```
rep movsb;
```

- Will repeat the move string operation until `cx` is zero, as in:

```
        jcxz   LDone
L0:
        movsb
        dec    cx
        jnz    L0
Ldone:
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

The REP Instruction Prefixes

- In addition to the standard rep instruction there are:
 - `repe/repz` repeat while the equal (or zero) comparison is true and count is not zero.
 - `repne/repnz` repeat while the equal (or zero) comparison is not true and count is not zero.
- Example: `repe/repz` and `repne/repnz` are equivalent to

```
    jcxz  Done
strLoop:
    strop
    jnz  Done
    dec  cx
    jnz  strLoop
Done:
```

```
    jcxz  Done
strLoop:
    strop
    jz   Done
    dec  cx
    jnz  strLoop
Done:
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Comments and Examples

- The string processing instructions and the flags:
 - `movs`, `lods`, and `stos` do not change the flags.
 - `cmps` and `scas` are changing the flags.
- The rep prefixes act 'correctly' when cx is initially zero.
 - The loop is not performed at all.
- Example:
 - Code the function `LenStr` which returns the length of the C-string pointed by `ds:si`.
 - The length is the number of characters in the string exclusive of the terminating 0.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Solution without REP

```
    push    es;           save current es
    push    ds;
    pop     es;           es = ds
    mov     di, si ;      scasb uses es:di
    mov     al, 0 ;       scan for zero
    cld
LenLoop:
    scasb
    jne     LenLoop
    mov     ax, di ;      ax
    sub     ax, si ;      ax = di - si;
    dec     ax;           ax = di - si - 1;
    mov     si, di
    pop     es
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Solution with REP

```
    push    es
    push    ds
    pop     es;
    mov     di, si ;      scasb uses es:di
    mov     al, 0 ;       scan for zero
    cld
    mov     cx, 0ffffh;   Big enough number.
    repne  scasb
    mov     ax, di ;
    sub     ax, si ;      ax = di - si;
    dec     ax;           ax = di - si - 1;
    mov     si, di
    pop     es
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

Homework

- Problems:
 - Exercises 10.2: 1, 2 (pp. 270)
 - Exercises 10.4: 3 (pp. 277)
 - Exercises 17.1: 1 (pp. 504)
 - Exercises 17.2: 2 (pp. 509)