AUBURN

UNIVERSITY

Arrays and …

Strings are just arrays of characters ….

Module Overview

Objectives

- Learn, understand, and define data (variables)
- Learn, understand, and define data arrays
- Copy, compare, and manipulate arrays with primitive array instructions.
- Extend to strings of characters

Requirements

- Know the registers
- Know the addressing modes (Indirect and Indexed)
- Know the instruction set (jumps and loops)

Data Definition

- Data definition statement
- Intrinsic data types
  - BYTE/SBYTE data
  - DUP Operator
  - WORD/SWORD data
  - DWORD/SDWORD data

Data Definition Statement

- **Syntax** of Data Definition Statement

  [name] datasize initializer [, initializer]

  Where ***name*** is an optional variable name

  datasize is a directive indicating the variable size

  initializer is ‘?’, an expression, or integer literal setting the initial value of the variable

- **Function**
  - This statement **reserves** (sets aside) storage in the memory for a variable. Note there is no difference between BYTE and SBYTE, WORD and SWORD, or DWORD and SDWORD. Assembly does not treat differently signed and unsigned variables: the difference in the definition is used only by the programmer to remember about whether the variable is signed or not.

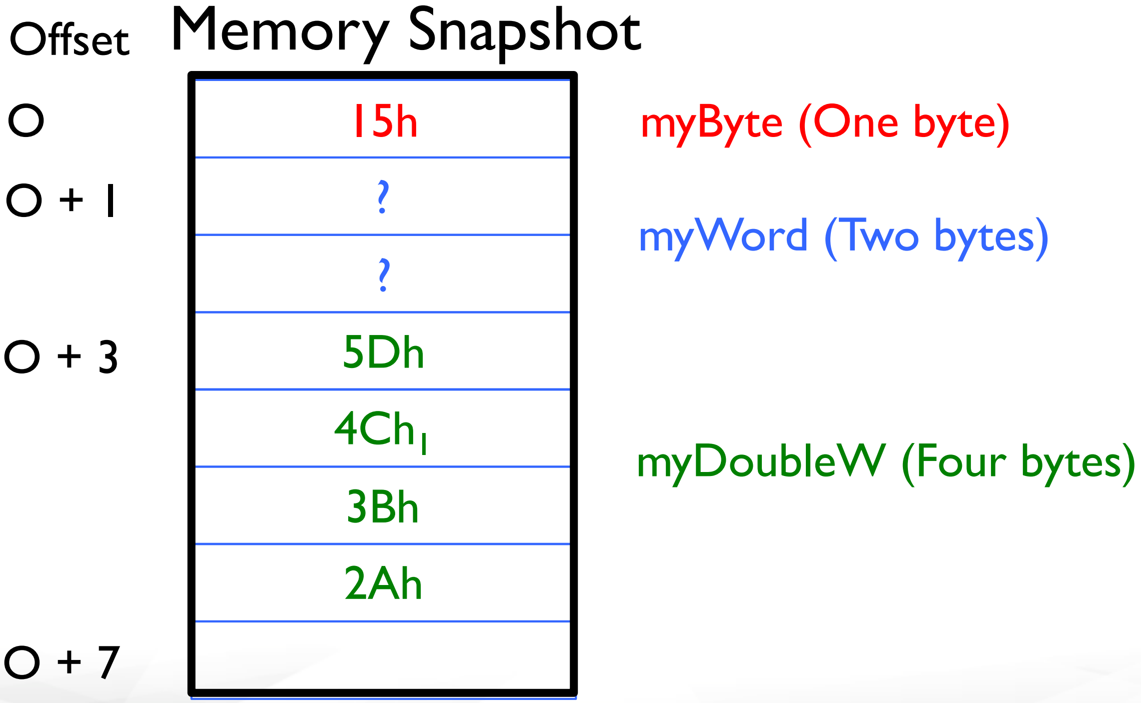
Datasize Directive	Data Type Reserved	Size (bytes)
BYTE	Unsigned 8-bit integer	1
SBYTE	**Signed** 8-bit integer	1
WORD	Unsigned 16-bit integer	2
SWORD	**Signed** 16-bit integer	2
DWORD	Unsigned 32-bit integer	4
SDWORD	**Signed** 32-bit integer	4

Data Definition Statement : Example I (1/4)

```
; datadefinitions.asm - Example
.386
.model flat, stdcall
.stack 4096
EXITProcess PROTO, dwExitCode:DWORD
.data ; This is where data definition starts
; Recall [name] datasize initializer [, initializer]
myByte    BYTE 5; the name is myByte, initializer is 5
myWord    WORD ?; Variable myWord will have a random value
myDoubleW DWORD 4*56; Variable will be intialized with 224
......
.code
```

Data Definition Statement : Example I (2/4)

```
; .......
.data ; This is where data definition starts
.........
myByte    BYTE 15h; the name is myByte, initializer is 5
myWord    WORD ?; Variable myWord will have a random value
myDoubleW DWORD 2A3B4C5Dh ;myDoubleW initialized to 2A3B4C5Dh
```



Offset

Memory Snapshot

O

15h

myByte (One byte)

O + 1

?

?

myWord (Two bytes)

O + 3

5Dh

4Ch

3Bh

2Ah

myDoubleW (Four bytes)

O + 7

Data Definition Statement : Example I (3/4)

```
.........
.data ; This is where data definition starts
myByte    BYTE 15h; the name is myByte, initializer is 5
myWord    WORD ?; Variable myWord will have a random value
myDoubleW DWORD 2A3B4C5Dh ;myDoubleW initialized to 2A3B4C5Dh
```

Fact : Consecutive Variables are stored contiguously

```
   ; Code to access the variables
   mov al, myByte ; al ← 15h
   mov dx, myWord ; dx ← unknown (will be known at runtime)
   mov ecx, myDoubleW; ecx ← 2A3B4C5Dh
```

Note: *source* and *destination* operands must be of the **same** size

Data Definition Statement : Example I (4/4)

```
.........
.data ; This is where data definition starts
myByte    BYTE 15h; the name is myByte, initializer is 15h
myWord    WORD ?; Variable myWord will have a random value
myDoubleW DWORD 2A3B4C5Dh ;myDoubleW initialized to 2A3B4C5Dh
```

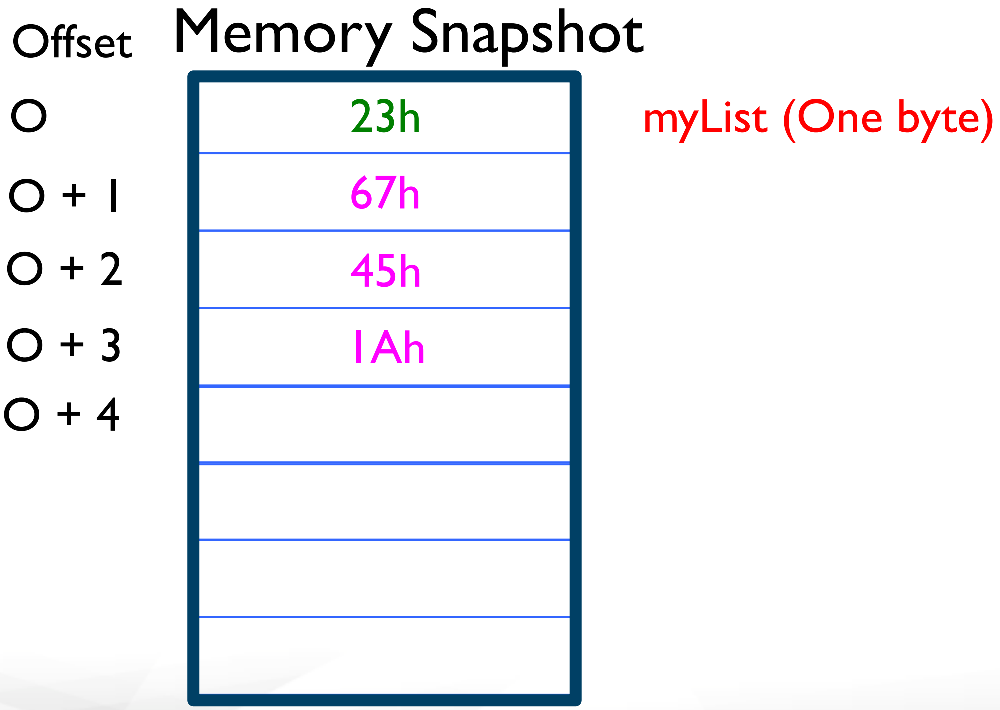
Another way to access these variables (using **indirect** and **indexed** addressing modes)

```
; Code to access the variables
mov ebx, OFFSET myByte ; ebx ← offset of myByte (O)
mov al, [ebx] ; al ← 15h (using indirect addressing mode)
mov ax, [ebx+1]; ax ← ??  (indexed ...)
mov eax, [ebx+3]; eax ← 2A3B4C5Dh
```

Offset	Memory Snapshot
O	15h
O + 1	??
O + 3	2A3B4C5Dh
O + 7	

Data Definition Statement : Example II (1/2)

```
……….
.data ; This is where data definition starts
; Recall [name] datasize initializer [, initializer]
myList BYTE 23h,67h,45h,1Ah; reserve and initialize 4 bytes
                  ; the name myList refers ONLY to the first
                    ; element containing 23h
```



Memory Snapshot

Offset	Memory Snapshot	
O	23h	myList (One byte)
O + 1	67h	
O + 2	45h	
O + 3	1Ah	
O + 4		

Data Definition Statement : Example II (2/2)

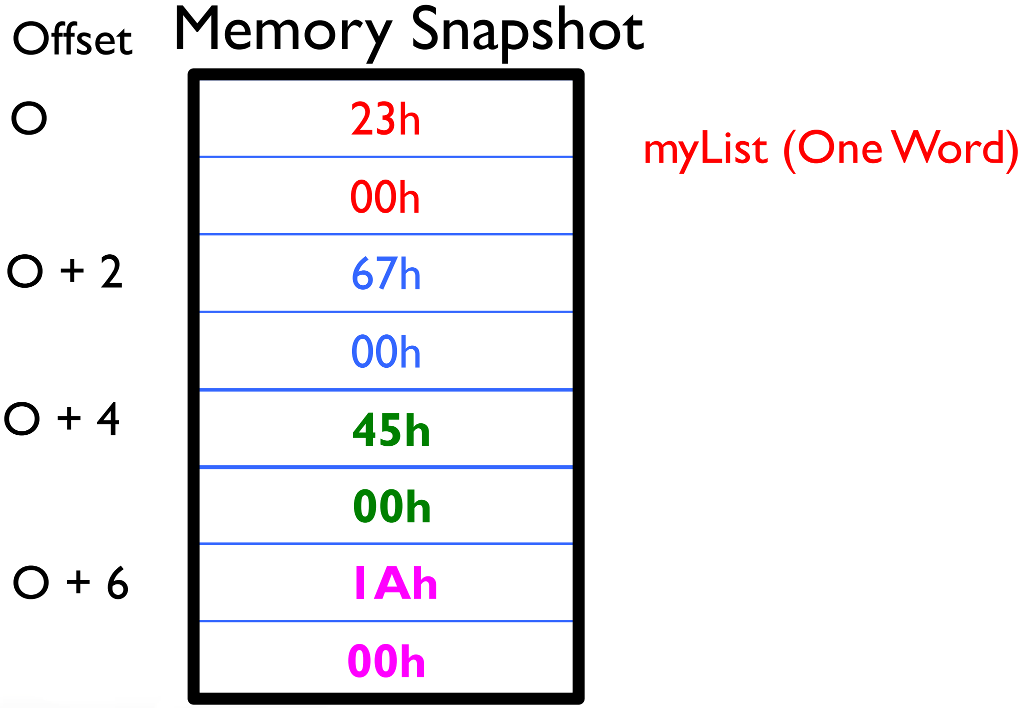
```
.........
.data ; This is where data definition starts
myList BYTE 23h,67h,45h,1Ah; reserve and initialize 4 bytes
                       ; myList refers ONLY to the first
              ; element containing 23h


;Accessing data
mov cl, myList    ; cl ← 23h
mov esi, OFFSET myList ; esi ← offset (O)of myList
mov al, [esi+2] ; al ← 45h
mov ax, [esi+1];  ax ← 4567h
mov eax, [esi] ; eax ← 1A456723h
```

Offset	Memory Content	
O	23h	myList
O + 1	67h	
O + 2	45h	
O + 3	1Ah	

Data Definition Statement : Example III (1/2)

```
..........
.data
myList WORD 23h,67h,45h, 1Ah; reserve and initialize 4 words
                        ; myList refers ONLY to the first
              ; word containing 0023h
```

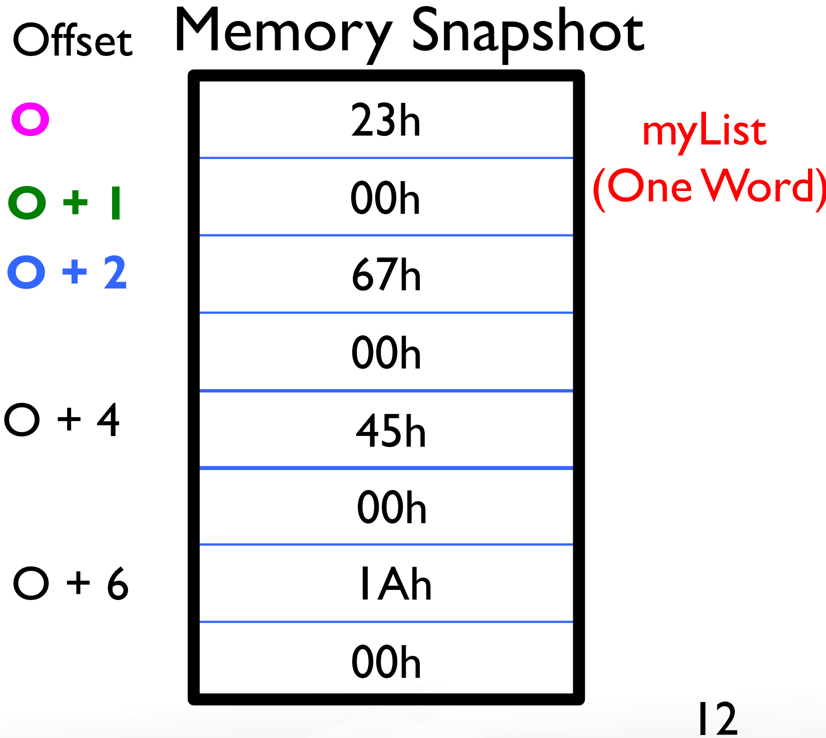


Offset	Memory Snapshot
O	23h
	00h
O + 2	67h
	00h
O + 4	45h
	00h
O + 6	1Ah
	00h

myList (One Word)

Data Definition Statement : Example III (2/2)

```
.........
.data
myList WORD 23h,67h,45h, 1Ah; reserve and initialize 4 bytes
                    ; myList refers ONLY to the first
           ; word containing 23h
;Accessing data
mov cx, myList ; cx ← 0023h
mov esi, OFFSET myList
mov al, [esi+2] ; al ← 67h
mov ax, [esi+1];  ax ← 6700h
mov eax, [esi] ; eax ←00670023h
```



Memory Snapshot

Offset	
O	23h
O + 1	00h
O + 2	67h
	00h
O + 4	45h
	00h
O + 6	1Ah
	00h

Data Definition Statement : DUP Operator

How to reserve tens, hundreds, or thousands of data items?

```
.data
myList BYTE 100 DUP (?); reserve 100 bytes not initialized
myListW WORD 40 DUP (0); reserve 40 words initialized to 0
anotherL SDWORD 250 DUP (-1); reserve 250 signed double
words                                     ; initialized to -1 (i.e.,
FFFFFFFFh)
```

- **Summary**:
  - with the DUP operator, we can reserve and initialize ARRAYS of bytes, words or double words. (There are many other intrisic data types).

Manipulating Arrays

- Simple functions on arrays
  - Copy one array to another location
  - Count the number of occurrences of a value ***v*** in an array
  - Compare two arrays

Copy An Array: Version 1 (Naïve)

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
mov esi,OFFSET mySource
mov edi,OFFSET myTarget
mov al,[esi] ; al ← first element of mySource
mov [edi],al ; [edi] (first element of myTarget) ← al
mov al,[esi+1] ; al ← second element of mySource
mov [edi+1],al ; [edi] (second element of myTarget) ← al
mov al,[esi+2] ; al ← third element of mySource
mov [edi+2],al ; [edi] (third element of myTarget) ← al
……….
; How about if the array has 10,000 elements?
```

Copy An Array: Version II (Use LOOP)

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 8 ; 8 is the number of elements to copy
myLoop :  mov al, [esi] ; al ← current element of mySource
          mov [edi], al ; [edi] ← al
          inc esi ; update esi to refer the next element
          inc edi ; update edi to refer the next element
          loop myLoop ; jump to myLoop if ECX > 0
```

; Later, we will see a better version using primitive array instructions

Count the Number of Occurrences In An Array

Problem Statement:

1) Count the number of occurrences of the value in AL in the array mySource

2) Store the number of occurrences in the register DL.

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
          mov al, 2Ch ; Looking for the occurrences of 2Ch
          mov esi,OFFSET mySource
          mov dl, 0 ; initialize DL (0 occurrences!!)
          mov ecx, 8 ; 8 is the number of elements to scan
myLoop :  cmp al, [esi]; compare[esi] and 2Ch
          jnz keepGoing ; if ([esi] != 2Ch)jump to
KeepGoing
          inc dl
keepGoing: inc esi ; update esi to refer the next
          loop myLoop
```

Compare Two Arrays

Problem Statement:

1) Compare two arrays (stop at the first difference)

2) If they are the same, return 0 in Register DL, otherwise return 1 in DL.

```
. data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 0Ah,1Bh,2Ch,3Dh,66h,5Fh,10h,43h
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 8 ; 8 elements to compare at most
          mov DL,0
myLoop :  mov al, [esi] ; al ← current element of mySource
          cmp[edi], al ; compare[esi] and al
          jz keepGoing ; if ([esi] == 2Ch)jump to
KeepGoing
          inc DL ; found a difference then DL ←DL + 1
          jmp Done   ; Done! no need to keep comparing
keepGoing:inc esi ; update esi to refer the next
          inc edi ; update esi to refer the next
          loop myLoop
Done :
```

Array Primitive Instructions

- Arrays primitive instructions:
  - **MOVSB**, **MOVSW**, **MOVSD**
  - **CMPSB**, **CMPSW**, **CMPSD**
  - Repeat Prefix (**REP**)
  - EFLAGS Direction Flag D (**CLD**, **STD**)

Move : MOVSB, MOVSW, and MOVSD

Syntax (Size)	**Function (if Flag D = 0) : Forward**	**Function (if Flag D = 1) : Reverse**
MOVSB (byte)	[edi] ← [esi]; esi ← esi **+ 1**; edi ←edi **+ 1**	[edi] ← [esi]; esi ← esi **- 1**; edi ←edi **- 1**
MOVSW (word)	[edi] ← [esi]; esi ← esi **+ 2**; edi ←edi **+ 2**	[edi] ← [esi]; esi ← esi **- 2**; edi ←edi **- 2**
MOVSD (double)	[edi] ← [esi]; esi ← esi **+ 4**; edi ←edi **+ 4**	[edi] ← [esi]; esi ← esi **- 4**; edi ←edi **- 4**

MOVSB Usage (1/2)

- Let us improve the program (Version II) to copy arrays

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 8 ; 8 is the number of elements to copy
myLoop :  mov al, [esi] ; al ← current element of mySource
          mov [edi], al ; [edi] ← al
          inc esi ; update esi to refer the next
          inc edi ; update edi to refer the next
          loop myLoop
```

CLD

MOVSB

MOVSB Usage (2/2)

- Let us improve the program (Version II) to copy arrays

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          CLD ; Clear D to go forward with MOVSB
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 8 ; 8 is the number of elements to copy
myLoop :  MOVSB
          loop myLoop
```

MOVSW Usage

- Let us improve **further** the program (Version II) to copy arrays

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          CLD ; Clear D to go forward with MOVSW
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, ~~8~~ 4 ; 4 words to copy
myLoop :  ~~MOVSB~~ MOVSW
          loop myLoop
```

MOVSD Usage

- Let us improve **further** the program (Version II) to copy arrays

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          CLD ; Clear D to go forward with MOVSD
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, ~~8~~ 2 ; 2 double words to copy
myLoop :  ~~MOVSB~~ MOVSD
          loop myLoop
```

Repeat Prefix **REP (1/3)**

Syntax: REP *Instruction*

Function: repeat *Instruction* while ECX > 0

Let us improve **further** the program (Version II) to copy arrays **using REP**

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          CLD
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 2 ; 2 double words to copy
myLoop :  MOVSD
          loop myLoop
```

REP MOVSD

Repeat Prefix **REP (2/3)**

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          CLD
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 2 ; 2 double words to copy
~~myLoop :~~   REP MOVSD
          ~~loop myLoop~~
```

Repeat Prefix **REP (3/3)**

1) There other Repeat Prefixes:

REPZ, REPE	**Repeat while the Zero Flag (ZR) is set and ECX > 0**
REPNZ,REPNE	**Repeat while the Zero Flag (ZR) is clear and ECX > 0**

2) REP could be used with MOVSB and MOVSW

```
.data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 8 DUP(?)
; Let us copy mySource onto myTarget
          CLD
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 4 ; 4 words to copy
~~myLoop :~~   REP MOVSW
          ~~loop myLoop~~
```

Compare CMPSB, CMPSW, and CMPSD

Syntax (Size)	Function (if Flag D = 0) : Forward	Function (if Flag D = 1) : Reverse
CMPSB (byte)	CMP [esi],[edi]; esi ← esi + 1; edi ←edi + 1	CMP [esi],[edi]; esi ← esi - 1; edi ←edi - 1
CMPSW (word)	CMP [esi],[edi]; esi ← esi + 2; edi ←edi + 2	CMP [esi],[edi]; esi ← esi - 2; edi ←edi - 2
CMPSD (double)	CMP [esi],[edi]; esi ← esi + 4; edi ←edi + 4	CMP [esi],[edi]; esi ← esi - 4; edi ←edi - 4

Use CMPSB to Compare Two Arrays (1/3)

Problem Statement:

1) Compare two arrays (stop at the first difference)

2) If they are the same, return 0 in Register DL, otherwise return 1 in DL.

```
. data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 0Ah,1Bh,2Ch,3Dh,66h,5Fh,10h,43h
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 8 ; 8 elements to compare at most
          mov dl,0
myLoop :  mov al, [esi] ; al ← current element of mySource
          cmp[edi], al ; compare[esi] and al
          jz keepGoing ; if ([esi] == 2Ch)jump to 
KeepGoing
          inc DL ; found a difference then DL ←DL + 1
          jmp Done    ; Done! no need to keep comparing
keepGoing:inc esi ; update esi to refer the next
          inc edi ; update esi to refer the next
          loop myLoop
Done :
```

```
cld
repe cmpsb
jz Done
mov dl, 1
```

Use CMPSB to Compare Two Arrays (2/3)

Problem Statement:

1) Compare two arrays (stop at the first difference)

2) If they are the same, return 0 in Register DL, otherwise return 1 in DL.

```
. data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 0Ah,1Bh,2Ch,3Dh,66h,5Fh,10h,43h
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, 8 ; 8 elements to compare at most
          mov DL,0
          cld
          repe cmpsb     ;repeat while equality and ecx >
0
          jz Done        ;if finished on an equality
          mov dl, 01h    ;there is a difference
Done :
```

Use CMPSD to Compare Two Arrays (3/3)

Problem Statement:

1) Compare two arrays (stop at the first difference)

2) If they are the same, return 0 in Register DL, otherwise return 1 in DL.

Let us compare double words instead of bytes.

```
. data
mySource BYTE 0Ah,1Bh,2Ch,3Dh,4Eh,5Fh,10h,43h
myTarget BYTE 0Ah,1Bh,2Ch,3Dh,66h,5Fh,10h,43h
          mov esi,OFFSET mySource
          mov edi,OFFSET myTarget
          mov ecx, ~~8~~ 2 ; ~~8~~ 2 elements to compare at most
          mov DL,0
          cld
          repe ~~cmpsb~~ cmpsd ;repeat while equality and ecx > 0
          jz Done         ;if finished on an equality
          mov dl, 01h     ;there is a difference
Done :
```

Strings

- Strings are **arrays** of characters!!!
- All what we did so far does apply to strings
- Difference resides essentially in
  1. having a termination character '\0' and
  2. the **initialization**: the assembler offers some convenience to initialize a string. (As humans, we prefer manipulate the characters rather than their codes (ASCII or others))

Data Definition Statement : What is Special About Strings?

Using what we know so far, here is how to reserve memory for the string : “ Hello World!”

```
.data
myGreeting BYTE 48h,65h,6Ch,6Ch,6Fh,20h,57h,6Fh,72h,6Ch,64h,21h,0
; note that you just need to know the ASCII code of each character
; note also the 0 at the end: in most languages strings terminate
; with the null character ‘\0’.
; This way (of reserving initializing strings) is not convenient.
;The assembler offers a convenient way to initialize strings
.data
newGreeting BYTE “Hello World!”,0 ;using characters instead of codes.
```

Summary:

- Fundamentally, strings are simply **arrays**.
- Strings can be reserved and manipulated the same way arrays are manipulated
- What is special about **strings**? Other than the initialization, **nothing**! A string is an array of characters. If a character is coded using one byte, that a string is an array of bytes. By convention, a string is terminated with the null character ‘\0’

Module Wrap Up

- Learn, understand, and define data (variables)
  - Intrinsic data types: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD
  - Initialization and access to variables (using indirect or indexed addressing mode)
- Learn, understand, and define data arrays
  - DUP operator
  - Manipulate arrays with what we know:
    - Naïve program to **copy** an array
    - Improve the program using LOOP to copy an array
    - Program to **count** the occurrences of a number in an array
    - Program to **compare** arrays
- Copy, compare, and manipulate arrays with primitive array instructions.
  - MOVSB, MOVSW, and MOVSD
    - Improve the program to **copy** an array by using MOVSB, MOVSW, or MOVSD
  - CMPSB, CMPSW, and CMPSD
    - Improve the program to compare two arrays by using CMPSB, CMPSW, or CMPSD
- Extend to strings of characters
  - A string of characters is an array!
  - For most languages, it must terminate with the null character \0 (i.e., 0)
  - Initialization is made convenient: can use characters instead of ASCII codes to initialize