# Laboratory Manual for
# Computer Organization and Assembly Language


Department of Computer Science
National University of Computer and Engineering Sciences
Chiniot-Faisalabad Campus

## Outline

- Using Data Segment
- Multiple Initializer/Array
- Little Endian and Big Endian Orders
- Symbolic Constants
- OFFSET Operator
- PTR Operator

# 1 Using Data Segment

## 1.1 DATA Segment

It is a large continuous chunk of memory in RAM which is used for storing variables.

1. Directive is **.data** for data segment.
2. All variables must be declared, and memory space for each allocated.
3. Data definition directive can be followed by a single value, or a list of values separated by commas.
4. Different data definition directives for different size types of memory.
   - **DB** - Define Byte (8 bits)
   - **DW** - Define Word (16 bits)
   - **DD** - Define Double Word (32 bits)
   - **DQ** - Define Quad Word (64 bits)

## 1.2 Data Definition Statement

A data definition statement sets aside storage in memory for a variable, with an optional name.

A data definition has the following syntax:

[`name`] `directive initializer` [`, initializer`]...

```
VAL1 DB 'A'
VAL2 DB 0
VAL3 DB 255
VAL4 DB ?
VAL5 DB 10H
VAL6 DB 20H
VAL7 DB -128
VAL8 DW 1234H
```

A question mark (?) initializer leaves the variable uninitialized.

## 1.3 Multiple Initializers

If multiple initializers are used in the same data definiton, its label refers only to the offset of the first initializer.

```
LIST DB 10, 20, 30, 40
```

Not all data definitions require labels.

```
LIST1 DB 10, 20, 30, 40
      DB 50, 60, 70, 80
      DB 90, 100, 110, 120
```

Within a single data definition, its initializers can use different radixes.

```
LIST2 DB 10, 32, 41H, 00100010B
LIST3 DB 0AH, 20H, 'A', 22H
```

### 1.3.1 Defining Strings

```
GREETING1 DB "Good afternoon", 0
GREETING2 DB "Good Night", 0
GREETING3 DB "Welcome to the"
          DB "Assembly Language Programming", 0DH, 0AH, 0
```

The hexadecimal codes 0DH and 0AH are alternately called CR/CL (carriage-return line-feed) or `end-of-line characters`. When written to standard output, they move the cursor to the left-column of the line following the current line.

### 1.3.2 DUP Operator

The DUP operator allocates storage for multiple data items, using a constant expression as a counter.

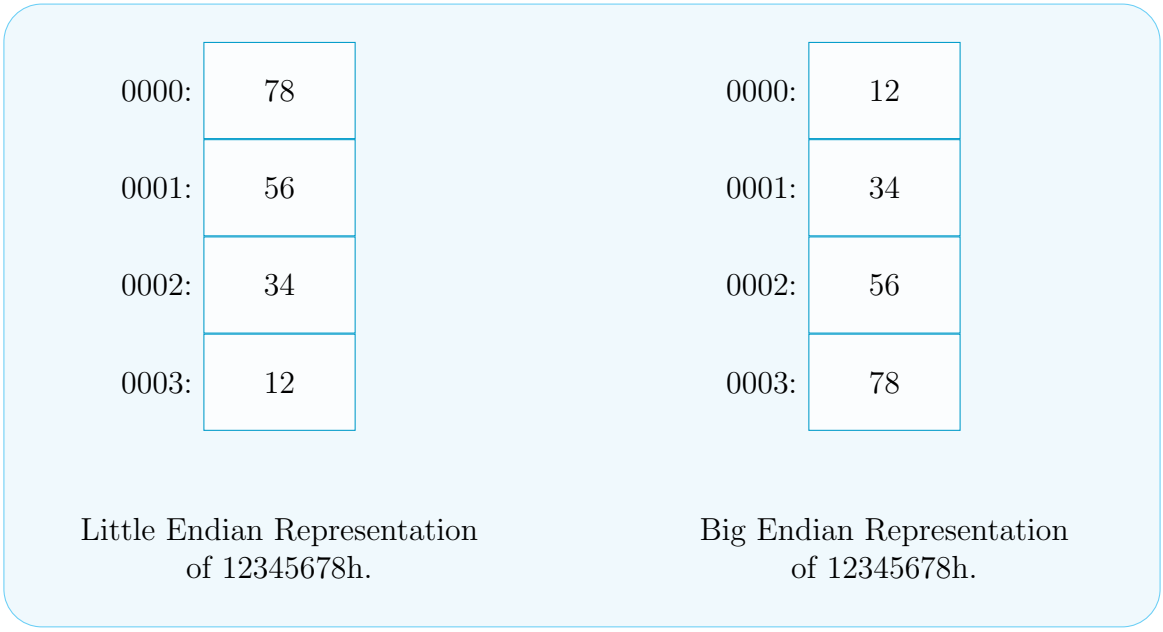
It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data.

```
arr1 DB 20 DUP(1)
arr2 DB 10 DUP(?)
arr3 DB 3 DUP("STACK")
```

# 2 Little-Endian and Big-Endian Orders

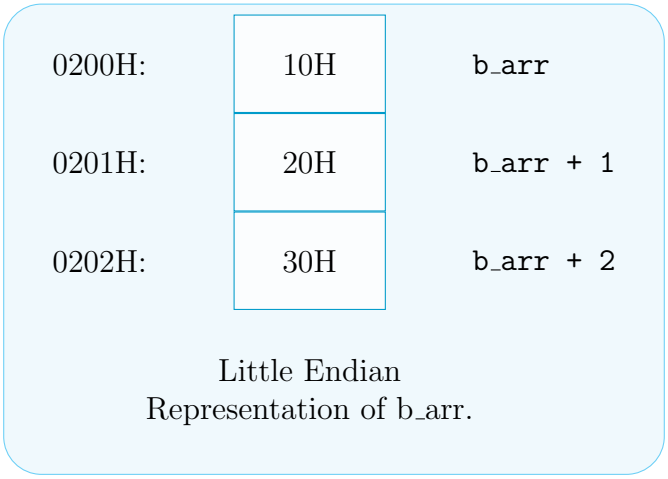
Little-endian and Big-endian are terms that describe the order in which a sequence of bytes are stored in computer memory. Big-endian is an order in which the "big end" (most significant value in the sequence) is stored at the lowest storage address and least significant value in sequence is stored at highest storage address. While little-endian is an order in which the "little end" (least significant value in the sequence) is stored at lowest storage address while most significant value is stored at highest storage address.

x86 processors store and retrieve data from memory using `little endian order (low to high)`.



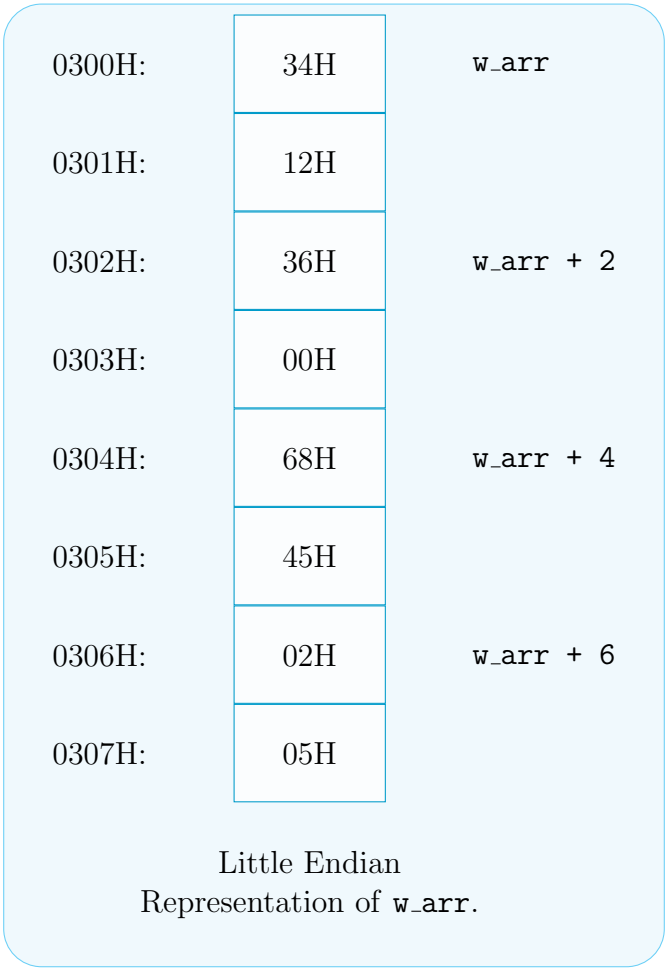

```
b_arr DB 10H, 20H, 30H
```

The name `b_arr` is associated with first of these bytes, `b_arr + 1` with the second byte, `b_arr + 2` with the third byte. If assembler assigns the offset address 0200H to `b_arr`, then the little-endian representation of `b_arr` would look like this:




```
w_arr DW 1234H, 36H, 4568H, 502H
```

If the `w_arr` starts at address 0300H. It's little-endian representation would look like this:




Little Endian
Representation of `w_arr`.

# 3 Symbolic Constants

A symbolic constant (or symbol definition) is created by associating an identifier (a symbol) with an integer expression or some text.

Symbols do not reserve storage. They are used only by the assembly when scanning a program, and they cannot change at runtime.

## 3.1 Equal-Sign Directive

The `equal-sign directive` associates a symbol name with an integer expression.

The syntax is

```
name = expression
```

When a program is assembled, all occurrences of `name` are replaced by `expression` during the assembler's preprocessor step.

```
.data
COUNT = 50
.code
mov cx, COUNT
```

**Why Use Symbols?** We might have skipped the COUNT symbol entirely and simply coded the MOV instruction with the literal 500, but experience has shown that programs are easier to read and maintain if symbols are used. Suppose COUNT were used many times throughout a program. At a later time, we could easily redefine its value: COUNT = 600

```
.data
ESC_KEY = 27
.code
mov al, ESC_KEY          ; good style
mov al, 27               ; poor style
```

### 3.1.1 Current Location Counter

The $ operator (current location counter) returns the offset associated with the current program statement.

### 3.1.2 Calculating the Sizes of Arrays and Strings

In the following example, **listsize** is calculated by subtracting the offset of **list** from the current location counter $.

```
list db 10, 20, 30, 40
listsize = ($ - list)
mystring db "Hello World!", 0
stringsize = ($ - mystring)
```

When calculating the number of elements in an array containing values other than bytes, you should always divide the total array size (in bytes) by the size of the individual array elements.

The following code, for example, divides the address range by 2 because each word in the

array occupies 2 bytes (16 bits).

```
wlist dw 1000h, 2000h, 3000h, 4000h
wlistsize = ($ - wlist)/2
```

Similarly, each element of an array of doublewords is 4 bytes long, so its overall length must be divided by four to produce the number of array elements.

```
dwlist dd 10000000h, 20000000h, 30000000h
dwlistsize = ($ - dwlist)/4
```

The current location counter $ must follow immediately after the array or string.

## 3.2 EQU Directive

The `EQU directive` associates a symbolic name with an integer expression or some arbitrary text.

There are three formats:

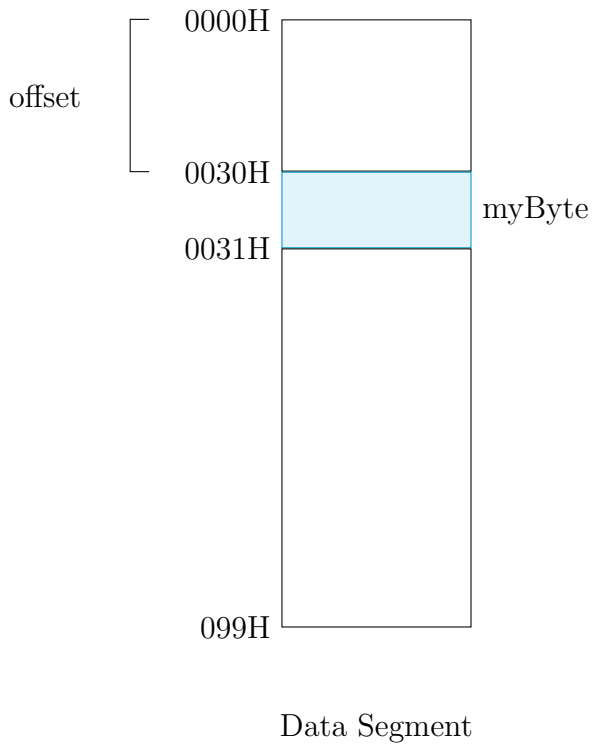
```
name EQU expression

name EQU symbol

name EQU <text>
```

# 4 OFFSET Operator

The OFFSET operator returns the offset of a data label. The offset represents the distance, in bytes, of the label from the begining of the data segment.




```
.data
val1 dw 0105h
val3 dd 12345678h
val2 dw 205h
.code
mov ax, offset val1
mov bx, offset val2
mov cx, offset val3
```

If data segment starts at address `0000h`, then the offset address of `val1` is `0000h`. And the offset address of `val3` is `0002h` because `val1` occupies `2 bytes` space. And the offset address of `val2` is `0006h` because the combine space of `val1` and `val3` is `6 bytes`.

# 5 PTR Operator

You can use the PTR operator to override the declared size of an operand. This is only necessary when you're trying to access the variable using a size attribute that's different from the one used to declare the variable.

```
.data
val3 dd 12345678h
.code
mov al, BYTE PTR val3
mov bx, WORD PTR val3
mov cx, WORD PTR val3+2
```