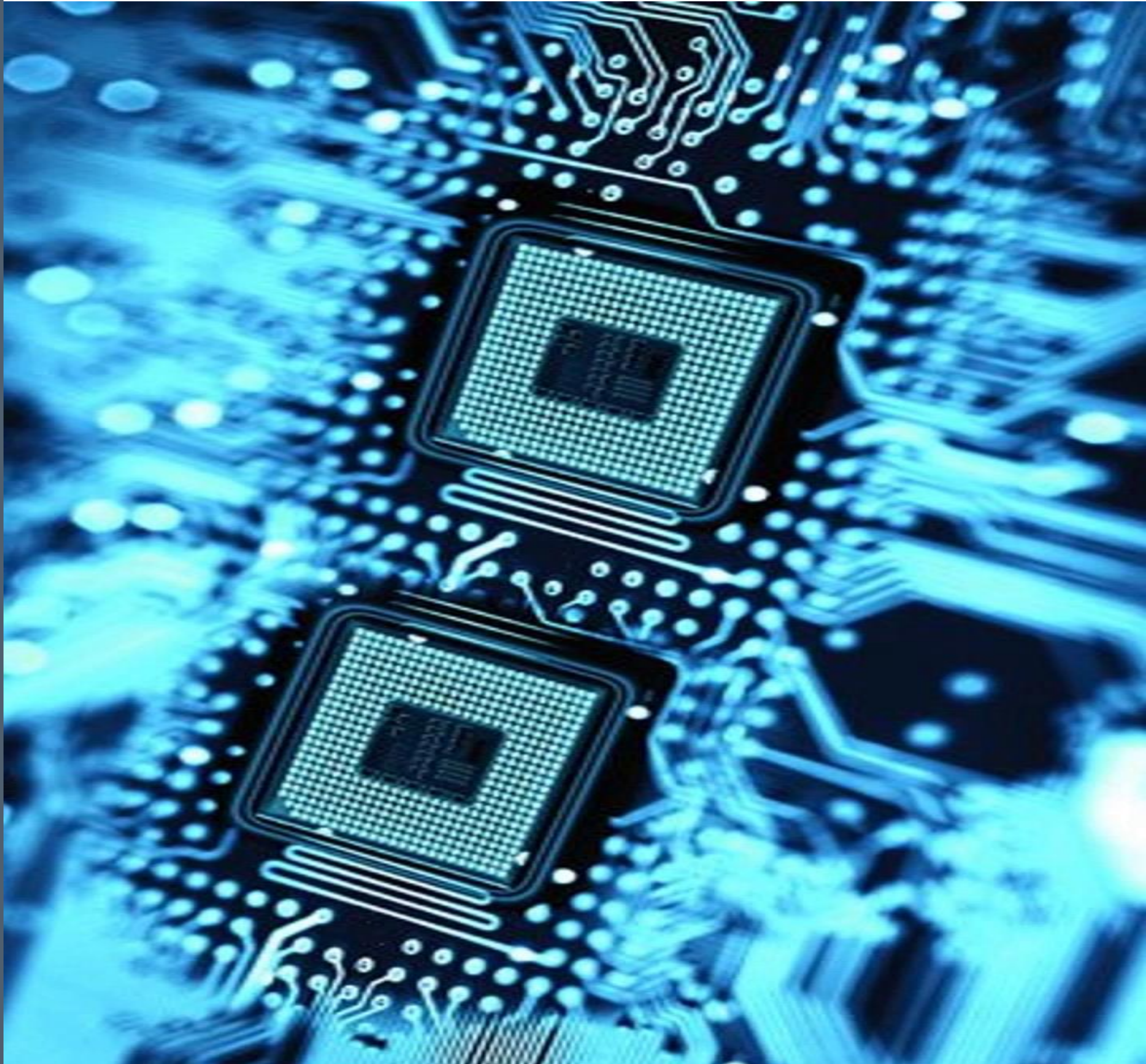
Computer organization & architecture

Course by: Dr. Ahmed Sadek

Lab By: Mahmoud Badry

Data Transfers, Addressing, and Arithmetic

Chapter 4

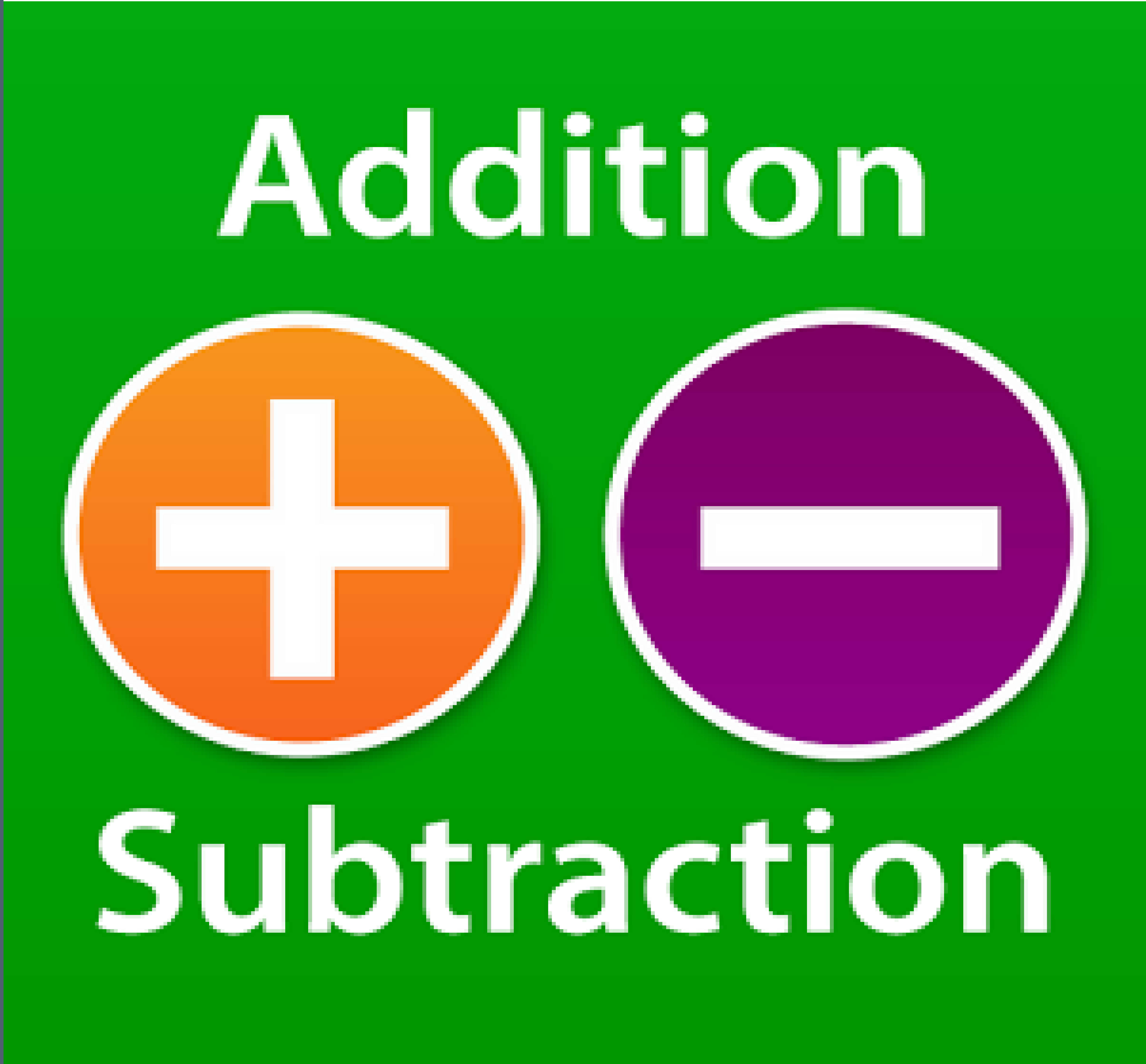


About Chapter

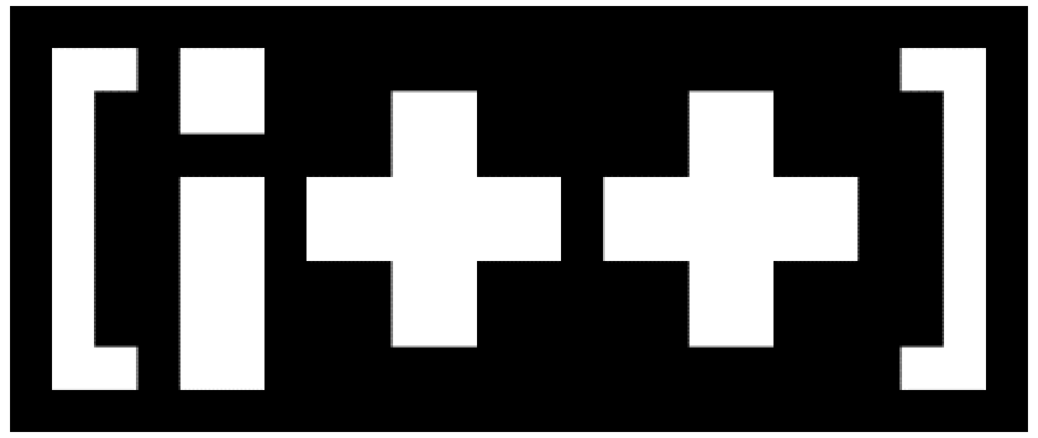
- In this chapter. you' re going to be exposed to a **surprising** amount of **detailed information**. You will encounter a major **difference** between **assembly** language and **high-level** languages.
- In assembly language, you can (and **must**) **control** every detail. You have **ultimate power**, and along with it, enormous **responsibility**.

Addition and Subtraction

Section 2



INC and DEC Instructions



- The INC (**increment**) and DEC (**decrement**) instructions, respectively, add 1 and subtract 1 from a **single operand**.

```
INC reg/mem
DEC reg/mem
```

- Example:

```
.data
myWord DW 1000h
.Code
inc myWord                ;1001h
mov bx,myWord
dec bx                    ;1000h
```

ADD Instruction

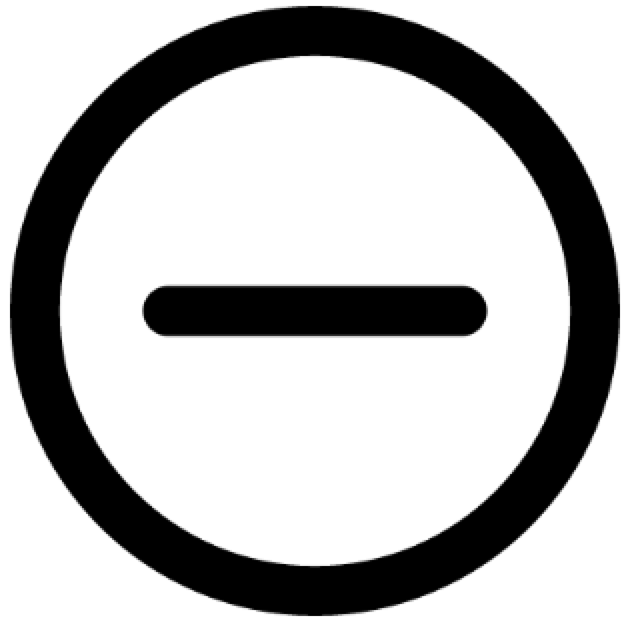
- The **ADD** instruction adds a **source** operand to a **destination** operand of the **same size**. The two operands cannot be memory operands.

```
ADD dest, source
```

- Example:

```
.data
var1 DW 1000h
var2 DW 2000h
.code
mov ax, var1
add ax, var2      ;3000h
```

SUB Instruction



- The **SUB** instruction subtracts a **source** operand from a **destination** operand of the **same size**. The two operands cannot be memory operands.

```
SUB dest, source
```

- Example:

```
.data
var1 DW 2000h
var2 DW 1000h
.code
mov ax, var1
sub ax, var2      ;1000h
```

- The **CPU** performs **subtraction** by first **negating** (Two’s complement) and **then adding**. For example, 4 - 1 is really 4 + (-1).

NEG Instruction

-1

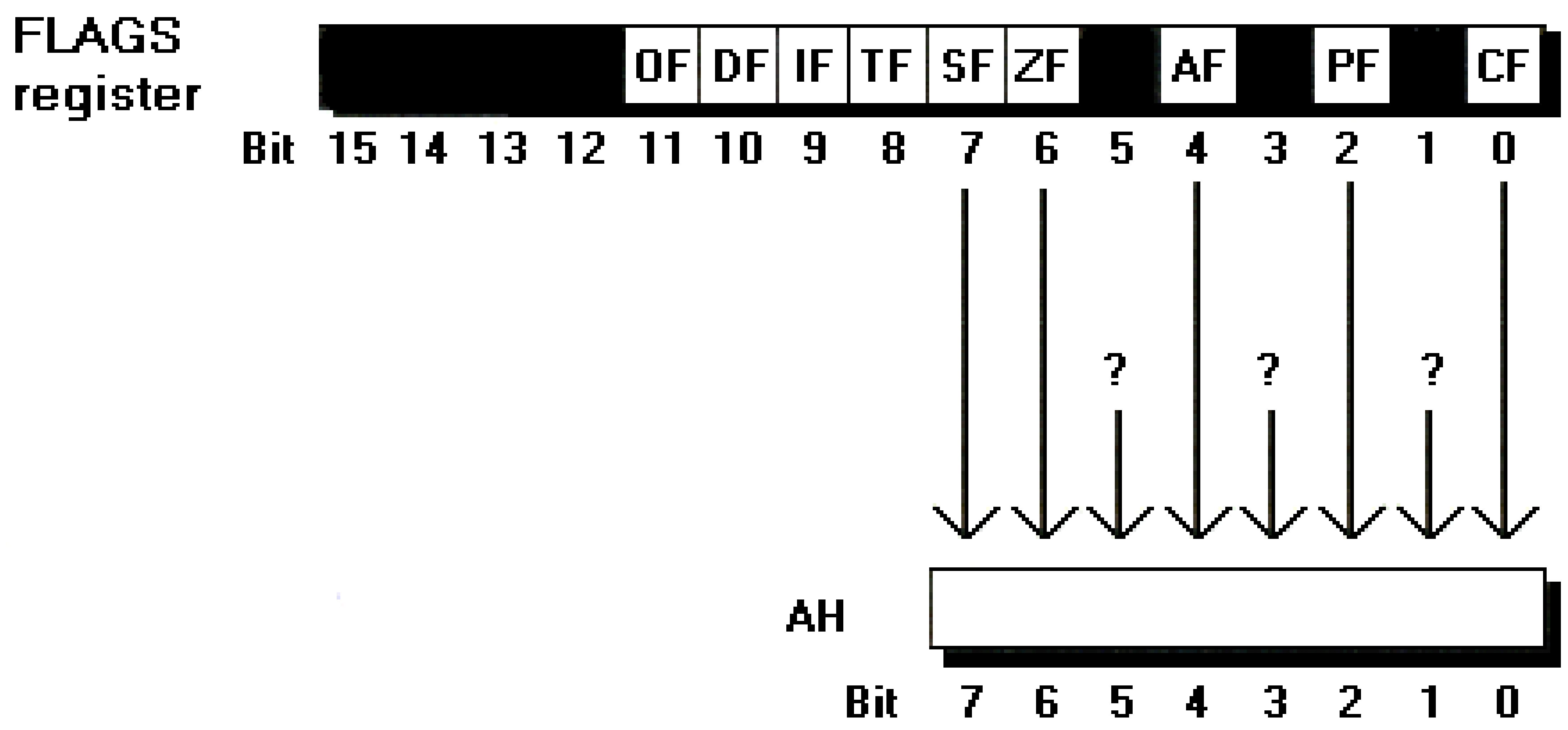
- The **NEG** (negate) instruction **reverses** the **sign** of a number by **converting** the **number** to its **two's complement**.

  NEG reg

  NEG mem

- Recall that the **two's complement** of a number can be found by **reversing** all the **bits** in the destination operand and **adding 1**.

Regarding ADD, SUB, INC, Dec and NEG instructions



- The **Carry**, **Zero**, **Sign**, **Overflow**, **Auxiliary Carry**, and **Parity** flags are **changed** according to the value of the **destination** operand.

Zero and Sign Flags

- The **Zero** flag is set when the **destination** operand of an **arithmetic** instruction is assigned a value of **zero**.

```
Mov cx,1
sub cx,1                ;CX = 0, ZF = 1
mov ax,0FFFFh
inc ax                  ;ax = 0, ZF = 1
inc ax                  ;ax = 1, ZF = 0
```

- The **Sign** flag is set when the **result** of an arithmetic operation is **negative**

```
mov CX,0
sub CX,1                ;CX = -1, SF = 1
add CX,2                ;cx = 1, SF = 0
```

Carry Flag (unsigned arithmetic)

- The **Carry** flag is significant only when the CPU **performs unsigned** arithmetic. If the result of an **unsigned operation** is too **large** (or too **small**) for the **destination** operand , the **Carry** flag is **set**.
- Example (Too **big**):

```
mov al,0FFh
add al,1                    ;CF = 1, AL = 0
```

- Example (Too **small**):

```
mov al,1
sub al,2                    ;AL = -1, CF = 1
```

- The **INC** and **DEC** instructions **don't affect** the **Carry** flag.

Overflow Flag (signed arithmetic)

- Is **set** when an **arithmetic** operation **generates** a **signed** result that **cannot fit** in the **destination** operand.
- Example:

```
.data
var DB +127
.code
mov bl, var
add bl, 1                ;OF = 1, BL = 080h
```

- Example

```
.data
var DB -128
.code
mov bl, var
sub bl, 1                ;OF = 1, BL = 07Fh
```

Overflow Flag (signed arithmetic)

- There is a very **easy** way to tell if **signed overflow** has occurred when **adding two operands**. **Overflow** has occurred if:
  - **Two positive** operands were **added** and their sum is **negative**.
  - **Two negative** operands were **added** and their sum is **positive**.
- Example (NEG):

```
mov al, -128        ;AL = 1000 0000b
neg al              ;AL = 1000 0000b, OF = 1
```

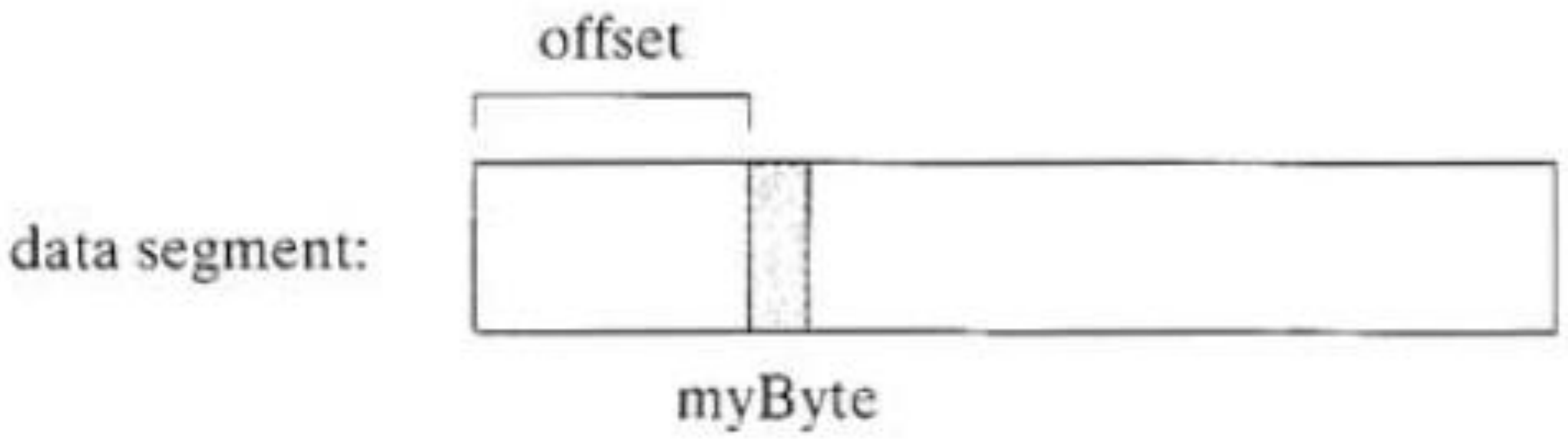
Data-Related Operators and Directives

Section 3

Data-Related Operators and Directives

- **Operators** and **directives**, as we said earlier, are **not part** of the Intel **instruction set**. They are only **understood** by the **assembler**.
- **MASM** has a number of **operators** that are effective tools for **describing** and **addressing** variables:
  - The **OFFSET** operator returns the **distance** of a variable from the beginning of its enclosing **segment**.
  - The **TYPE** operator returns the **size** (in bytes) of each **element** in an array,
  - The **LENGTHOF** operator returns the **number** of **elements** in an array .
  - The **SIZEOF** operator returns the **number** of **bytes** used by an array **initializer**.
  - The **LABEL** directive provides a way to **redefine** the same **variable** with different size attributes.
- The operators and directives in this chapter represent only a **small subset** of the **operators supported** by **MASM**.

OFFSET Operator



- The **OFFSET** operator return s the **offset** of a **data label**. The **offset** represents the **distance**, in **bytes**, of the label from the **beginning** of the **data segment**.
- Example:

```
.data
val1  DB 1,2,3
val2  DB ?
.code
mov si,OFFSET val2            ;si = DS - val2
mov si,OFFSET val1+1          ;si = DS - val1[1]
```

LABEL Directive

- The **LABEL** directive lets you **insert** a **label** and give it a **size** attribute **without allocating** any **storage**.
- One **common use** of LABEL is to provide an **alternative name** and **size** attribute for some **existing variable** in the data segment.
- **Example:**

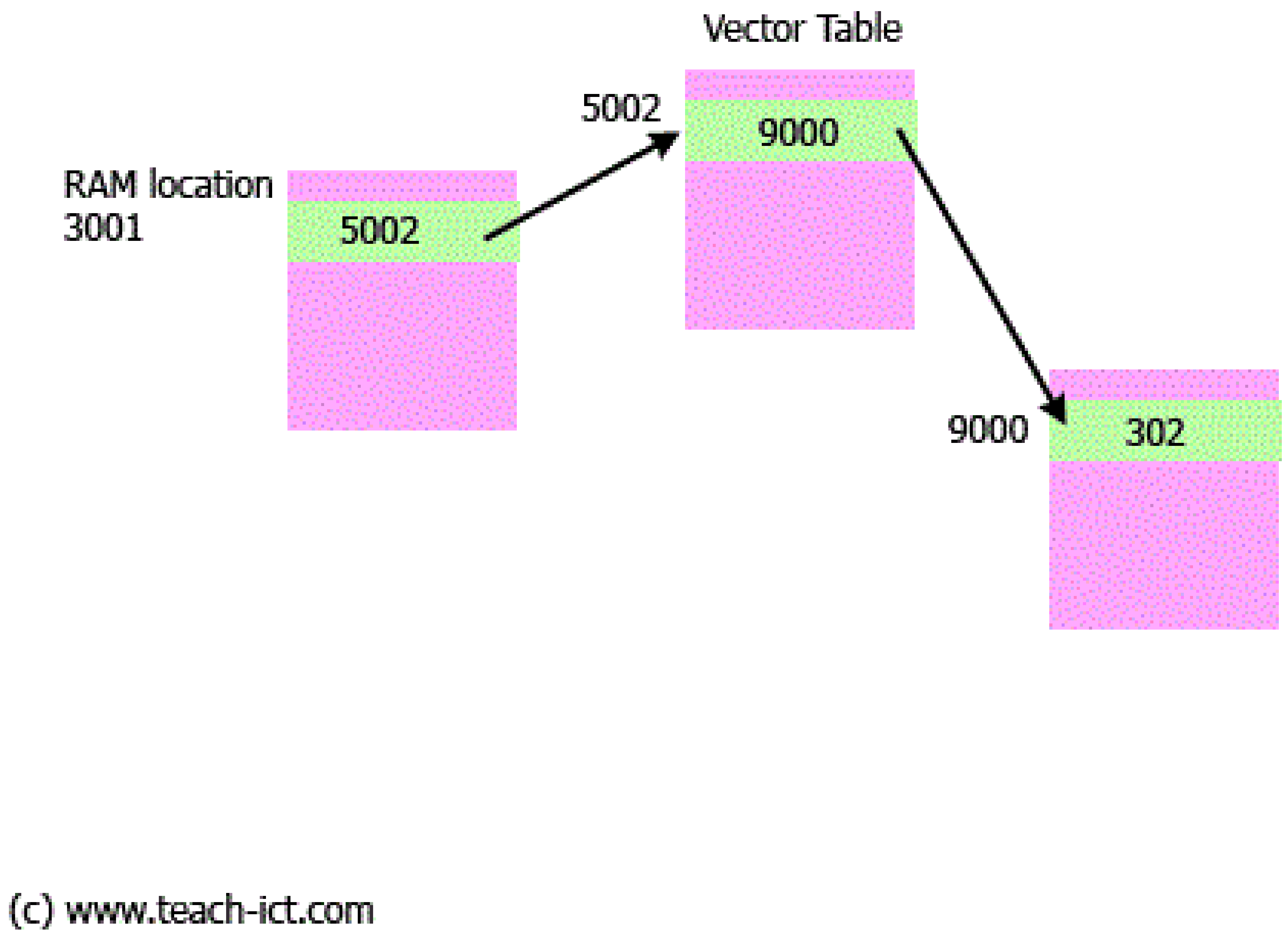
```
.data
val1  LABEL  DB
val2  DW  0FFAAh
.code
mov  si,val1
mov  bl  ,[si]              ;bx  =  AA
mov  bl  ,[si+1]            ;bx  =  FF
```

- **Example**:

```
.data
val1  LABEL  DW
val2  DB  0ffh
val3  DB  0aah
.code
mov  si,  OFFSET  val1
mov  bx  ,[si]               ;BX  =  0aaffh
```

Indirect Addressing

Section 4 (Self-study)



Indirect Operands

- An **indirect operand** can be any 16-bit **general-purpose register** (AX, BX, CX, DX, SI, Di, BP, and SP) surrounded by **brackets**. The register is assumed to contain the **offset** of some data .
- **Example:**

```
.data
val1 DB 010h
.code
mov si, OFFSET val1
```

- If a **MOV** instruction uses the **indirect operand** as the source. the **pointer** in SI is **dereferenced** and a **byte** is moved to AL:

```
mov al,[si]      ; AL = val1 = 10h
```

- Or, if the **indirect operand** is the **destination** operand . a **new value** is **placed** in **memory** at the location **pointed** to by the **register**:

```
mov [si],bl    ; val1= BL
```

- **General Protection Fault** happens when the **CPU executes** a *general protection (GP)* **fault**.
- Example:

```
;SI uninitialized
mov ax , [si] ;GP fault happens.
```

Using PTR with Indirect Operands

- The **size** of an **operand** is often **not clear** from the context **of** an **instruction**. Consider the following instruction:

```
inc [si]    ;assemble doesn’t know the size of SI
```

- **Solution** to **use PTR** directive:

```
inc WORD PTR [si]
```

- PTR keyword is used with variables like this:
  - BYTE PTR
  - WORD PTR

Indirect Operands and arrays

- Indirect operands are so **useful** with arrays:

.data

```
arrayB DB 10h, 20h, 30h
.code
mov si,OFFSET arrayB
mov al , [si]           ;AL 10h
inc si
mov al , [si]            ;AL 20h
inc si
mov al,[si]              ;AL = 30h
```

- If the array was an **array** of **words**, **replace** inc si with add si,2 and so on, because **size** of **word** is **two bytes**.

Indexed Operands

- An **indexed operand** adds a **constant** to a **register** to generate an effective **address**. Form of indexed operands:

  ```
  constant[reg]
  [constant + reg]
  ```

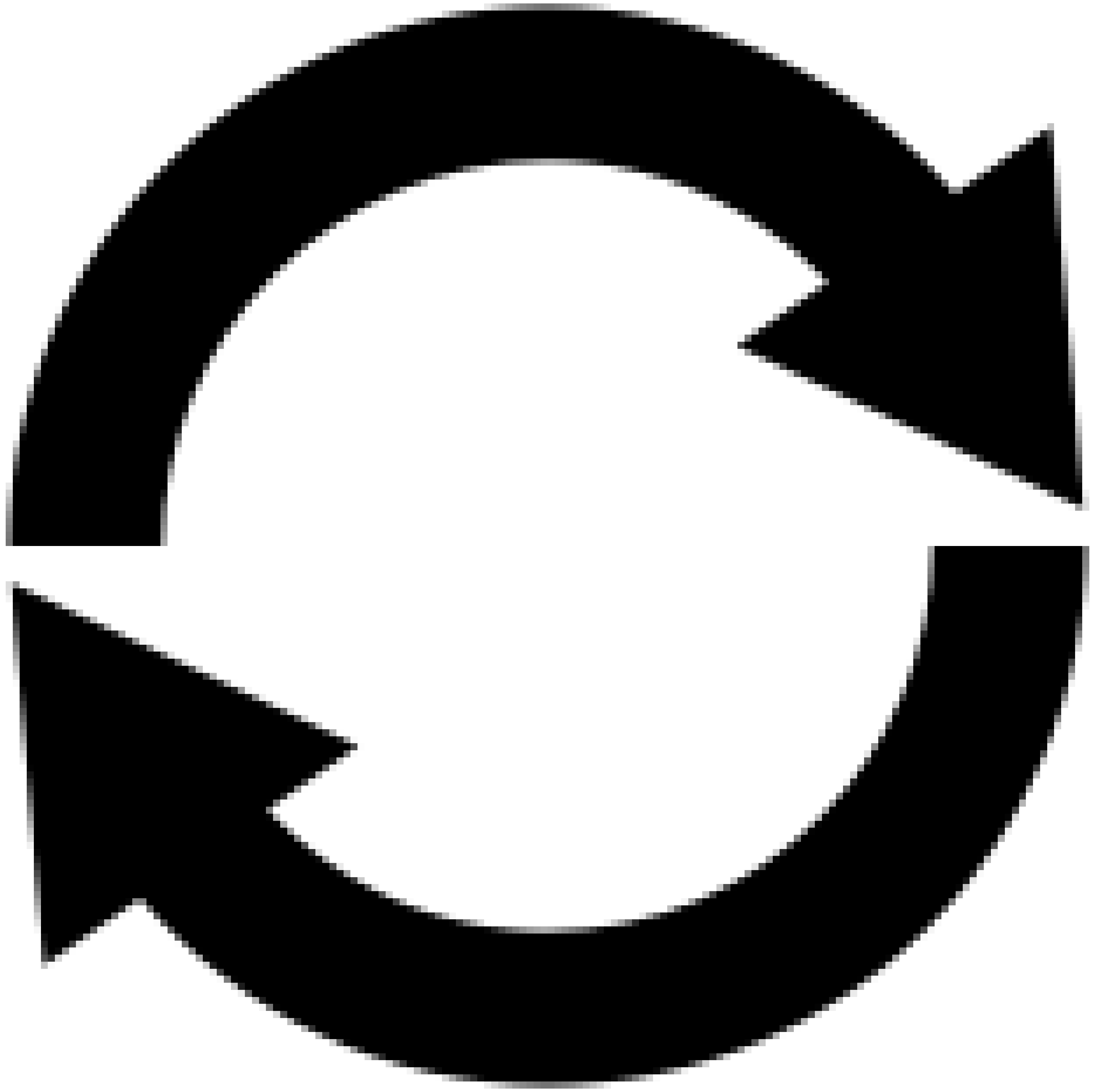
  The above commands are equal.

- Example:

  ```
  .data
  arrayB DB 10h,20h,30h
  . code
  mov si,0
  mov AL, [arrayB + si]     ;AL=10h
  inc esi
  mov AL, arrayB [si]       ;AL=20h
  ```

JMP and LOOP Instructions

Section 5



JMP and LOOP Instructions

- The **CPU automatically** loads and **executes** programs **sequentially**. As each instruction is **decoded** and **executed**. the **CPU** has already **incremented** the **instruction pointer** to the **offset** of the next instruction.
- **Real-life programs** are **not** that **simple**, it contains (**IF** – **go to** - **loops**)
- A **transfer of control**, *or branch.* is a way of **altering** the **order** in which **statements** are **executed**. There are **two types** of **transfer**:
    - **Unconditional Transfer**: The program branches to a new location in **all cases**, The **JMP** instruction is a good example.
    - **Conditional Transfer**: The program branches if a certain **condition** is **true**, **LOOP** is a good example.

JMP Instruction

- The **JMP** instruction causes an **unconditional transfer** to a target location inside the code segment. Its **syntax** is like this:

```
JMP targetLabel
```

- When the **CPU executes** this **instruction**. the **offset** of **target label** is **moved** into the **instruction pointer**. causing **execution** to immediately continue at the **new location**.
- Example (**infinite** loop):

```
top:
…
…
jmp top          ;repeat the endless loop
```

LOOP Instruction

- The **LOOP** instruction provides a simple way to **repeat** a **block** of **statements** a specific number of times. **CX** is **automatically used** as a **counter** and is **decremented** each time the **loop repeats**. It's **syntax** is

  ```
  LOOP destination
  ```

- The **execution** of the **LOOP** instruction involves **two** steps : **First** , it **subtracts** 1 from **CX** .**Next**. it **compares CX** to **zero**. If **CX** is **not equal** to **zero**, a **jump** is **taken** to the **label identified** by **destination** . Otherwise, if **CX** equals **zero**, **no jump** takes place and **control passes** to the **instruction following** the loop.

- **Example:**

  ```
  .data
  mov ax,0
  mov cx,5
  L1 :   inc ax
         loop L1   ;Loops 5 times
  ```

- A **common** programming **error** is to inadvertently **initialize CX** to **zero before** beginning a **loop** . If this happens, the LOOP instruction decrements **CX** to **0FFFFh**.

THANKS