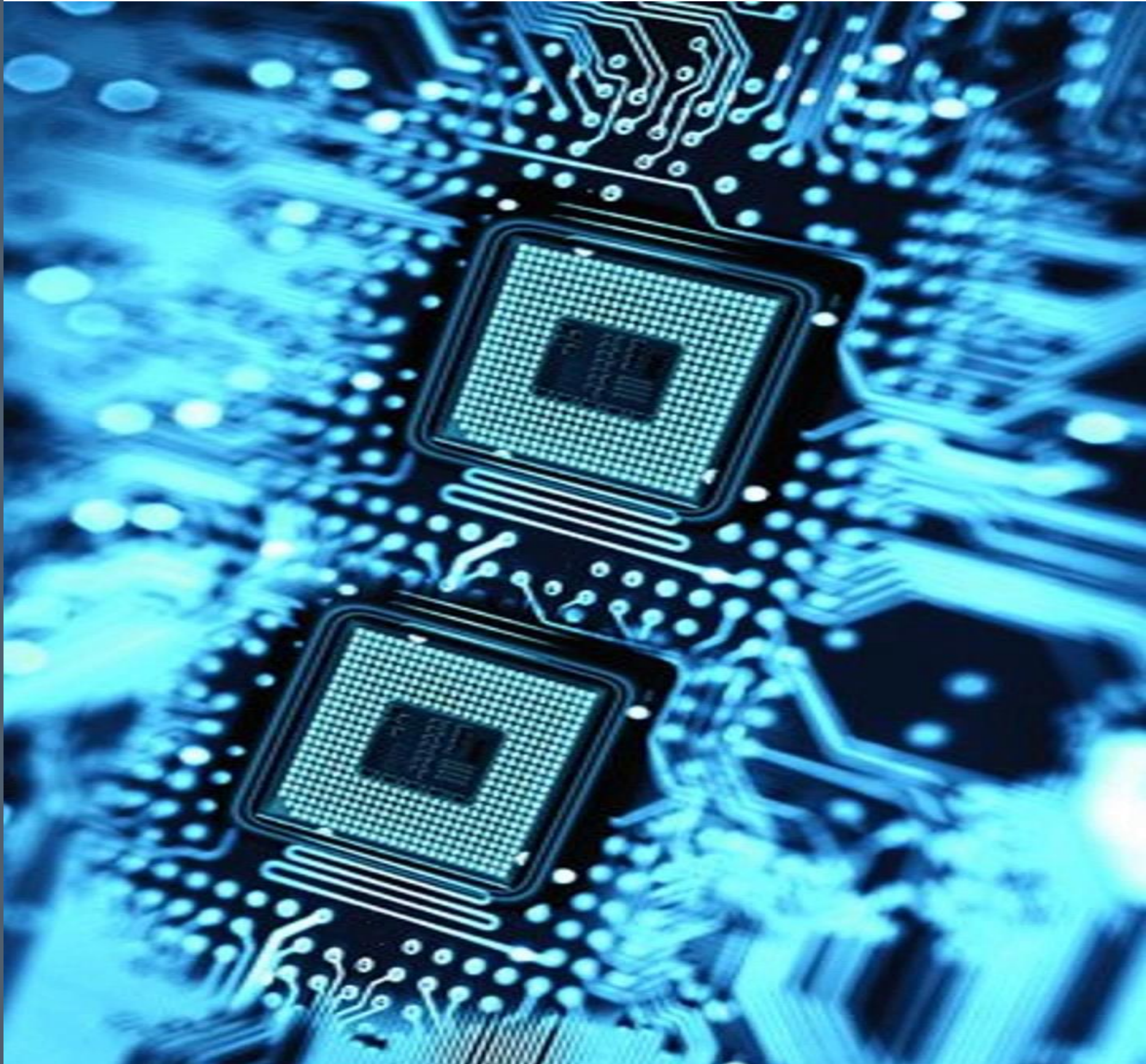
# Computer organization & architecture

Course by: Dr. Ahmed Sadek

Lab By: Mahmoud Badry

# Data Transfers, Addressing, and Arithmetic

Chapter 4

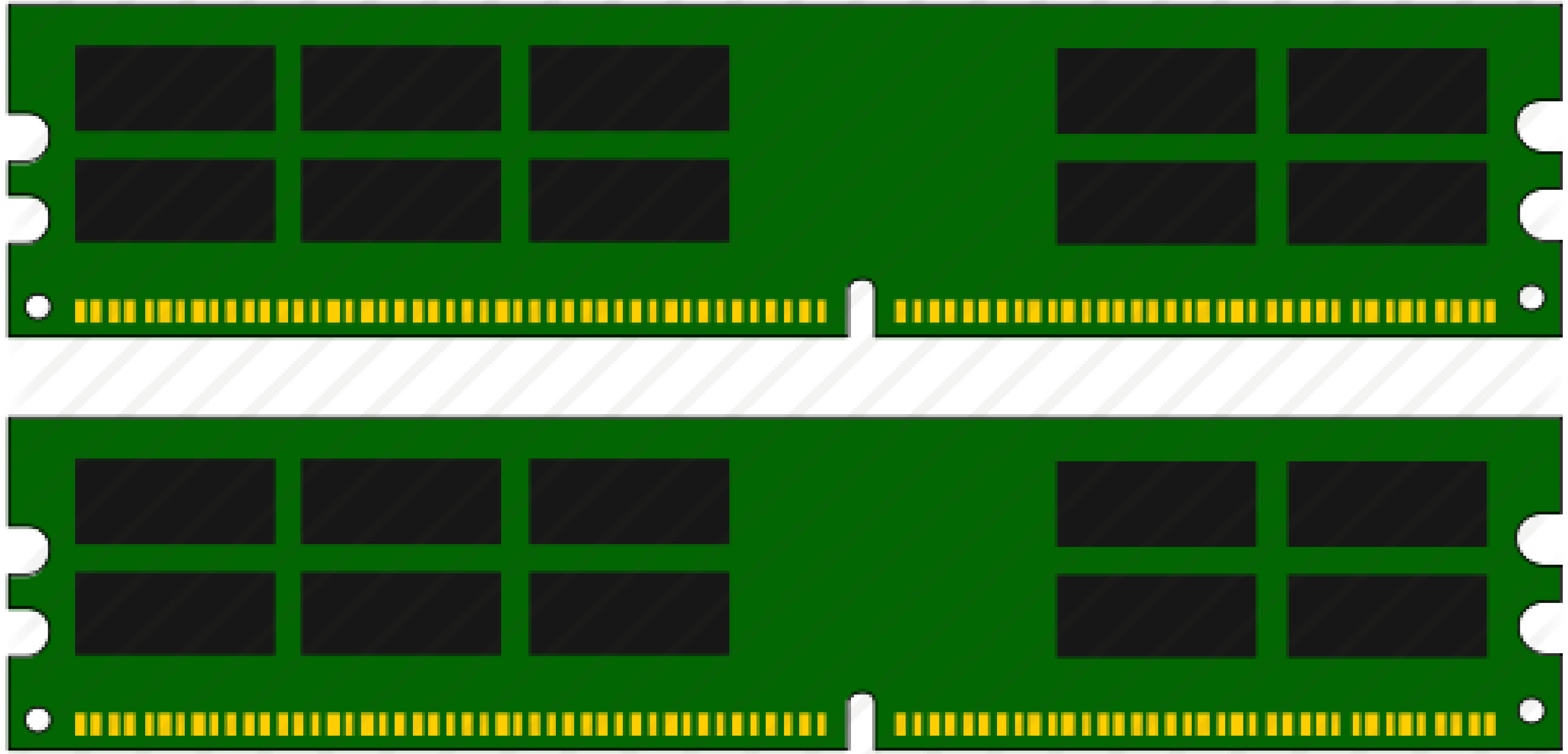


# About Chapter

- 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**.

# Data Transfer Instructions

Section 1



# Operand Types

| Operand | Description |
|---|---|
| r8 | 8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL |
| r16 | 16-bit general-purpose register: AX. BX, CX. DX. SI |
| r32 | 32-bitgeneral-purpose register: EAX. EBX, ECX, EDX |
| reg | Any general-purpose register |
| sreg | 16-bit segment register: CS,DS,SS,ES,FS,DS |
| imm | 8-,16-,or 32-bit immediate value |
| imm8 | 8-bit immediate byte value |
| imm16 | 16-bit immediate word value |
| imm32 | 32-bit immediate double word value |
| r/m8 | 8-bit operand which can be an 8-bit general register or memory byte |
| r/m16 | l6-bit operand which can be a 16-bitgeneral register or memory word |
| r/m32 | 32-bit operand which can be a32-bit general register or memory double word |
| mem | An 8-, 16-,or 32-bit memory operand |

# Direct Memory Operands

- Regarding this **example**:

```
.data
var1 DB 10h
```

- Suppose **var1** were located at **offset** 0102h. Then a **machine-level** instruction **referencing** this data would be assembled as:

```
mov al,[0102h]
```

- While it might be **possible** to **write** programs that used **numeric addresses** as **operands**, it is much **easier** to use **symbolic** names such as var1.

```
mov al,var1
```

```
OR mov al,[var1]
```

# MOV Instruction

- The **MOV** instruction **copies** data **from** a **source** operand to a **destination** operand.

  `MOV destination, source`

  Equals `destination = source;`
- In nearly **all assembly** language instructions, the **left-hand** operand is the **destination** and the **right-hand** operand is the **source**.
- MOV is very **flexible** in its use of operands, as long as the following **rules** are **observed**:
  - **Both operands** must be the **same size**.
  - **Both operands** cannot be **memory** operands.
  - **CS**, **EIP**, and **IP** **cannot** be **destination** operands.
  - An **immediate value cannot** be moved to a **segment** register.

- **Here is a list of the general variants of MOV, excluding segment registers:**

```
MOV  reg, reg
MOV  mem, reg
MOV  reg, mem
MOV  mem, imm
MOV  reg, imm
```

- **Segment registers** are **not modified** by programs **running** in **Protected mode**. The following **options** are available, with the **exception** that **CS** cannot be a target operand:

```
MOV r/m16,  sreg
MOV sreg,  r /m16
```

# Copying Smaller Values to Larger Ones

- For **unsigned** values, must **first** move **zero** and then move the small value:

```
.data
count DB 1
.code
mov cx, 0
mov cl, count
```

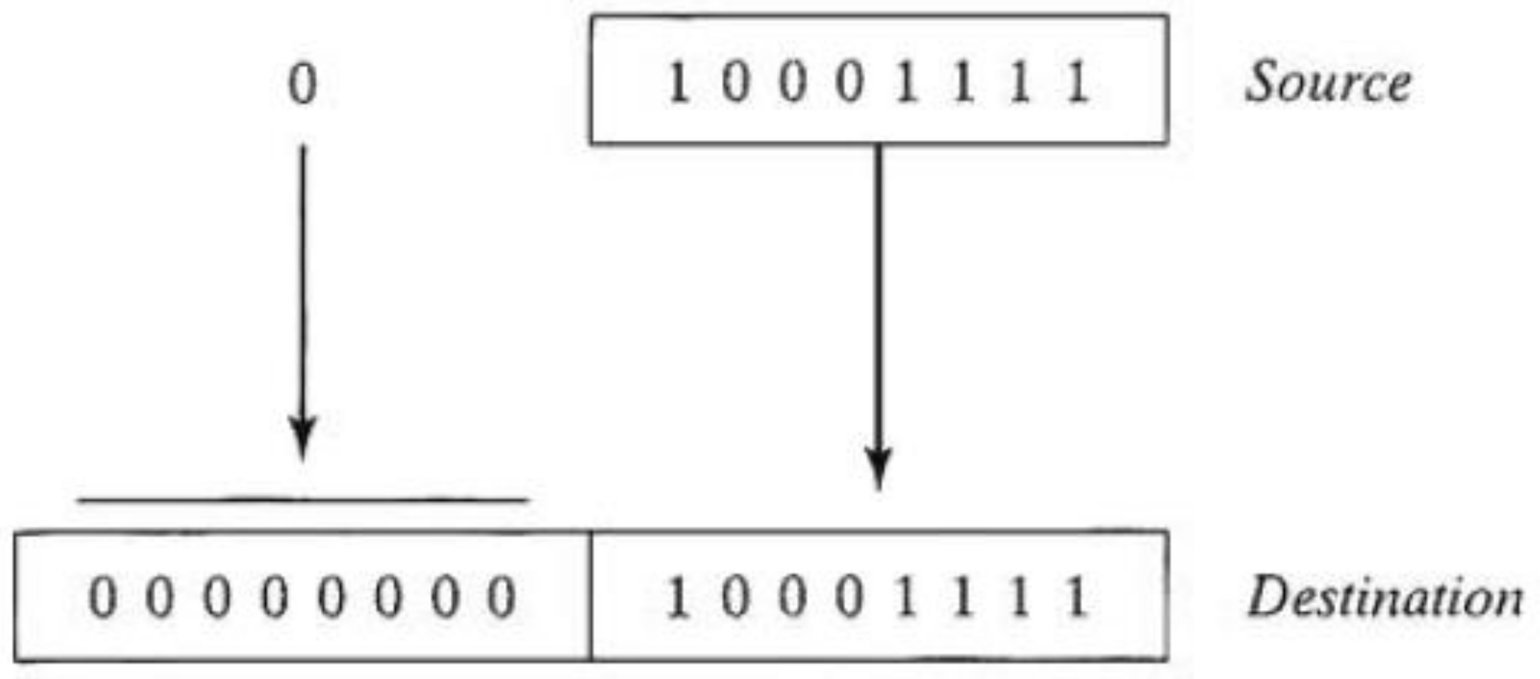
- What happens if we try the **same approach** with a **signed** integer

```
.data
signedVal DB -16      ;FFF0h
.code
mov cx, 0
mov cl, signedVal
;CX = 0000FFF0h (+65520)
```

- To **solve** this **problem** we should do this:

```
mov cx, OFFFFFFFFh
mov cl, signedVal
;ECX = FFFFFFFOh (-16)
```

# MOVZX Instruction (move with zero-extend)

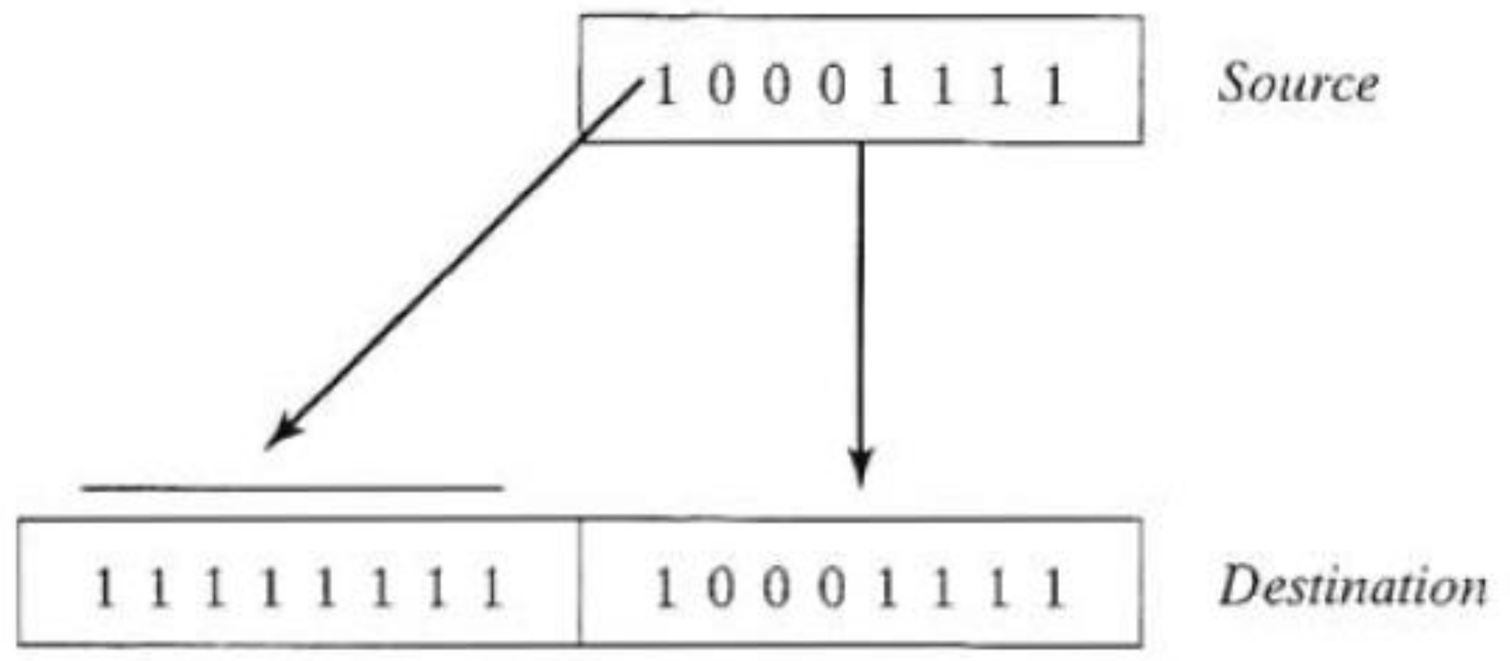


- **Copies** the contents of a **source** operand into a **destination** operand and **zero-extends** the value to either **16** or **32** bits.

```
MOVZX r32, r/m8
MOVZX r32, r/m16
MOVZX r16, r/m8
```

- **Not working** in 8086.

# MOVSX Instruction (move with sign-extend)



- **Copies** the contents of a **source** operand into a **destination** operand and **sign-extends** the value to either **16** or **32** bits.

```
MOVSX r32, r/m8
MOVSX r32, r/m16
MOVSX r16, r/m8
```

- **Not working** in 8086.

# LAHF and SAHF Instructions

- The LAHF (**load** status **flags** into **AH**) instruction **copies** the low byte of the **EFLAGS** register into **AH**. The following flags are copied: **Sign**, **Zero**, **Auxiliary Carry**, **Parity**, and **Carry**. Using this instruction:

```
.data
saveflags DB ?
.code
lahf
mov saveflags, ah
```

- The SAHF (**store AH** into status **flags**) instruction copies **AH** into the low byte of the **EFLAGS** register:

```
mov ah, saveflags
sahf
```

- **Same flag registers** are **evolved**.

# XCHG Instruction

- The XCHG (**exchange data**) instruction **exchanges** the **contents** of two **operands**. There are three variants:

```
XCHG reg, reg
XCHG reg ,mem
XCHG mem, reg
```

- The **rules** for **operands** in the XCHG instruction are the **same as** those for the **MOV** instruction, except that **XCHG** does **not accept immediate** operands.

- **Examples:**

```
xchg ax, bx
xchg ah, al
xchg varl, bx
```

- Exchange **memory** variables:

```
mov ax , vall
xchg ax, val2
mov vall ,ax
```

# Direct-Offset Operands

- Let's begin with an **array of bytes** named arrayB:

```
arrayB DB 10h ,20h , 30h ,40h, 50h
```

- To get **first element** of array:

```
mov al, [arrayB]          ;AL = 10h
```

- To get the **second** element:

```
mov al, [arrayB+1]          ;AL = 20h
```

- To get the **third** element:

```
mov al, [arrayB+2]          ;AL = 20h
```

- What about this?

```
mov al, [arrayB+20]
```

- What about word arrays:

```
.data
arrayW DW 100h,200h,300h
.code
mov ax, [arrayW]          ; AX = 100h
mov ax, [arrayW+ 2]       ; AX = 200h
```

- What about double word?

```
.data
arrayW DD 10000h,20000h,30000h
.code
mov eax, [arrayW]          ; AX = 10000h
mov eax, [arrayW+ 4]       ; AX = 20000h
```

# THANKS