# Chapter 2.2: 8086 Registers

## Microprocessors

The register set | AX · BX · CX · DX | SP · BP · SI · DI | Segment registers and IP

The FLAGS register | Status and control flags | Instructions that change flags

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/microprocessors/slides/Chapter_02_02_8086_Registers.html

INTRODUCTION

# Learning Outcomes

**Register set**

List the 14 registers of the 8086 by group and explain their 8- and 16-bit use.

**Special roles**

Know the implicit roles of AX, BX, CX, DX, SP, BP, SI and DI in instructions.

**Flags**

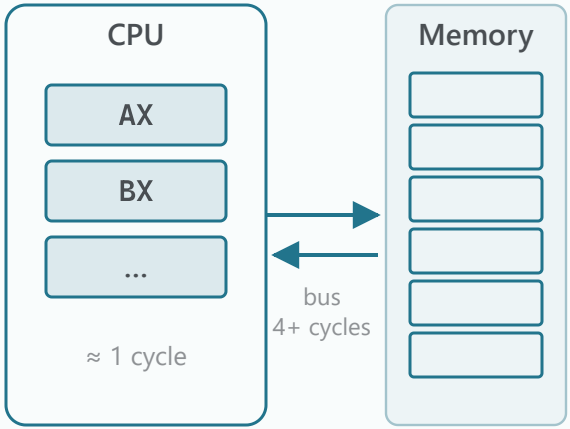
Explain which results set CF, ZF, SF, OF, PF and AF; know how DF, IF and TF control the processor.

**Register pairs**

Know which accesses use the pairs CS:IP, SS:SP, SS:BP, DS:SI and ES:DI.

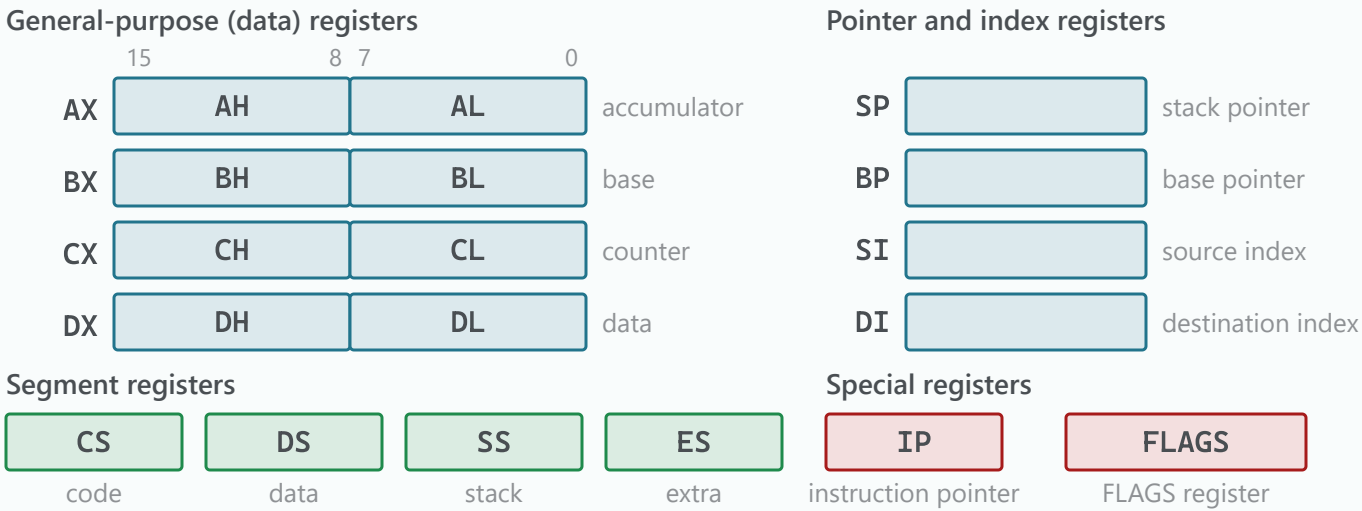
# What Is a Register?

- A **register** is a small, very fast storage cell *inside* the processor.
- 8086 registers are **16 bits** wide; four of them can also be split into two 8-bit halves.
- A memory access takes several clock cycles over the external bus; a register access is almost **free**.
- Registers hold temporary data, addresses, counters and results.
- Most instructions need at least one operand in a register: the 8086 has no **memory-to-memory** MOV or ADD.




The 8086 has **14** registers: 4 general-purpose data registers, 4 pointer/index registers, 4 segment registers, the instruction pointer (IP) and the FLAGS register.

# The 8086 Register Set




All are 16 bits. Only AX, BX, CX and DX split into halves ( `AH/AL` …); SP, BP, SI, DI and the segment registers have **no** 8-bit parts. In this course the eight registers AX…DI count as the **general-purpose registers**.

# General-Purpose Registers

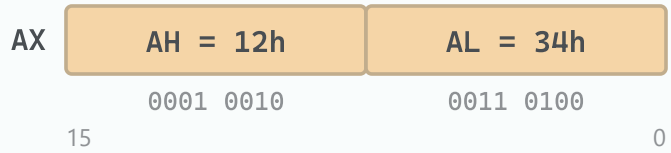
They hold temporary data. Most instructions accept any of them, but some instructions use a particular register **implicitly**.

| Register | Name | 8-bit halves | Implicit use (example) |
|---|---|---|---|
| `AX` | Accumulator | AH, AL | MUL, DIV, IN, OUT, LODS/STOS |
| `BX` | Base | BH, BL | memory addressing `[BX]`, XLAT |
| `CX` | Counter | CH, CL | LOOP, REP, shift count in CL |
| `DX` | Data | DH, DL | high half of MUL/DIV, port address |
| `SP` | Stack Pointer | — | PUSH, POP, CALL, RET |
| `BP` | Base Pointer | — | stack frame `[BP+n]` |
| `SI` | Source Index | — | string source DS:SI |
| `DI` | Destination Index | — | string destination ES:DI |

# AX = AH : AL — 8-Bit and 16-Bit Use

```
▶ MOV AX, 1234h    ; AH=12h, AL=34h
2 MOV AL, 0FFh     ; low half only
3 ADD AL, 1        ; 8-bit add
4 MOV AX, 12FFh
5 ADD AX, 1        ; 16-bit add
```

Writing one 8-bit half does **not change** the other. The carry of an 8-bit operation goes to CF, not into the other half.




AX gets 1234h: high byte **AH = 12h**, low byte **AL = 34h**.

Step 1 / 5

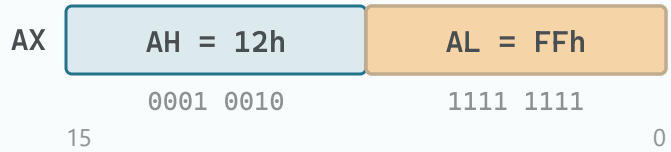
VARIABLES

AX = **1234h** AH = **12h** AL = **34h** CF = **0** ZF = **0**

# AX = AH : AL — 8-Bit and 16-Bit Use

```
1 MOV AX, 1234h    ; AH=12h, AL=34h
▶ MOV AL, 0FFh     ; low half only
3 ADD AL, 1        ; 8-bit add
4 MOV AX, 12FFh
5 ADD AX, 1        ; 16-bit add
```

Writing one 8-bit half does **not change** the other. The carry of an 8-bit operation goes to CF, not into the other half.




Only AL changes; AH is kept → AX = **12FFh**.

Step 2 / 5

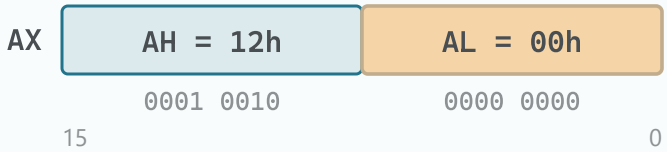
VARIABLES

AX = **12FFh** AH = **12h** AL = **FFh** CF = **0** ZF = **0**

# AX = AH : AL — 8-Bit and 16-Bit Use

```
1 MOV AX, 1234h    ; AH=12h, AL=34h
2 MOV AL, 0FFh     ; low half only
▶ ADD AL, 1        ; 8-bit add
4 MOV AX, 12FFh
5 ADD AX, 1        ; 16-bit add
```

Writing one 8-bit half does **not change** the other. The carry of an 8-bit operation goes to CF, not into the other half.




FFh + 1 = 100h does not fit in 8 bits: AL = **00h**, CF = 1, ZF = 1. The carry **does not reach AH**; AX = 1200h.

Step 3 / 5

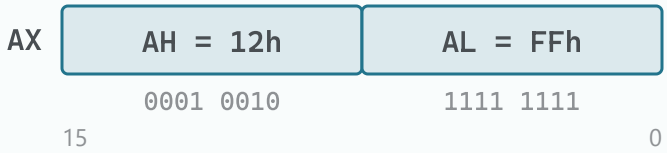
VARIABLES

AX = **1200h** AH = **12h** AL = **00h** CF = **1** ZF = **1**

# AX = AH : AL — 8-Bit and 16-Bit Use

```
1 MOV AX, 1234h    ; AH=12h, AL=34h
2 MOV AL, 0FFh     ; low half only
3 ADD AL, 1        ; 8-bit add
▶ MOV AX, 12FFh
5 ADD AX, 1        ; 16-bit add
```

Writing one 8-bit half does **not change** the other. The carry of an 8-bit operation goes to CF, not into the other half.




AX is set back to 12FFh.

Step 4 / 5

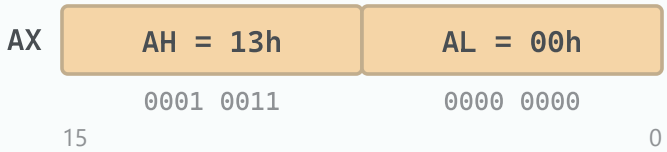
VARIABLES

AX = 12FFh | AH = 12h | AL = FFh | CF = 1 | ZF = 1

# AX = AH : AL — 8-Bit and 16-Bit Use

```
1 MOV AX, 1234h    ; AH=12h, AL=34h
2 MOV AL, 0FFh     ; low half only
3 ADD AL, 1        ; 8-bit add
4 MOV AX, 12FFh
▶ ADD AX, 1        ; 16-bit add
```

Writing one 8-bit half does **not change** the other. The carry of an 8-bit operation goes to CF, not into the other half.




This time the add is 16-bit: 12FFh + 1 = **1300h**; the carry moved into AH, CF = 0.

Step 5 / 5

VARIABLES

AX = 1300h  AH = 13h  AL = 00h  CF = 0  ZF = 0

# AX — Accumulator

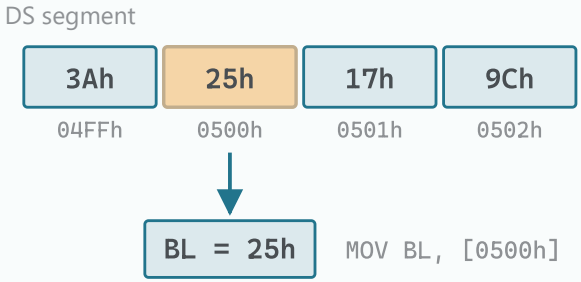
- 16 bits wide; splits into two 8-bit parts, **AH** (high) and **AL** (low).
- 8-bit instructions can use AH and AL separately.
- Used mostly in arithmetic and logic instructions.
- Some instructions use AX/AL **implicitly**: `MUL`, `DIV`, `IN` / `OUT`, `CBW`, `DAA`, `LODSB` / `STOSB`.
- With AX, many instructions encode one byte **shorter** (e.g. `ADD AX, imm16`).

```
ADD AX, AX      ; AX = AX + AX
MOV AX, 0005h
ADD AX, AX      ; AX = 000Ah
MOV AH, 4Ch     ; DOS: exit
INT 21h
```

Adding AX to itself is the same as multiplying it by 2.

# BX — Base

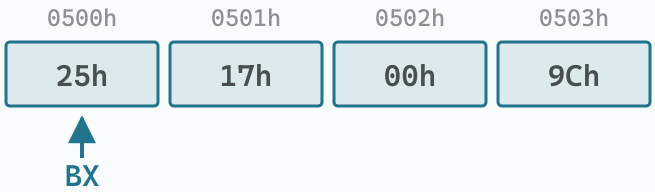
- 16 bits; splits into two 8-bit halves, **BH** and **BL**.
- The only data register that can hold an **offset** in indirect memory addressing: `[BX]`, `[BX+SI]`, `[BX+4]`.
- Holds the base address of arrays and tables (matrices).
- Addressing as `[AX]`, `[CX]` or `[DX]` does **not exist** on the 8086.




`MOV BL, [500h]` does not put the *number* 500h in BL; it reads the **byte** at DS:0500h. To load the number, drop the brackets: `MOV BX, 0500h`.

# Indirect Access with BX

```
1 MOV BX, 0500h      ; BX = address
2 MOV AL, [BX]       ; AL = [0500h]
3 MOV DL, [0501h]    ; direct address
4 ADD AL, DL         ; AL = 25h + 17h
5 MOV [BX+2], AL     ; [0502h] = AL
```




BX gets the number **0500h**; BX can now serve as an address (a pointer).

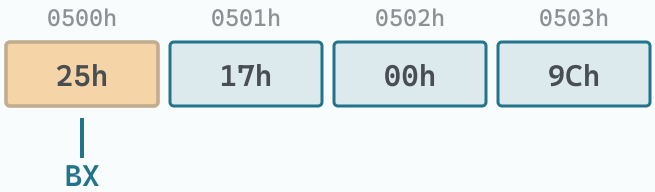
Step 1 / 5

VARIABLES

BX = **0500h** AL = **00h** DL = **00h**

# Indirect Access with BX

```
1 MOV BX, 0500h      ; BX = address
▶ MOV AL, [BX]       ; AL = [0500h]
3 MOV DL, [0501h]    ; direct address
4 ADD AL, DL         ; AL = 25h + 17h
5 MOV [BX+2], AL     ; [0502h] = AL
```




`[BX]` : the byte BX points to is read → AL = **25h**.

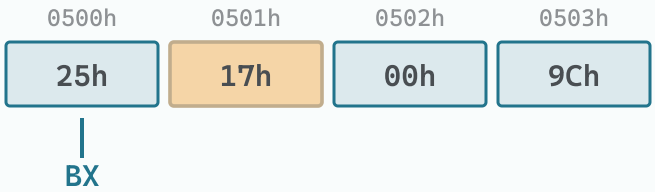
Step 2 / 5

VARIABLES

BX = **0500h** AL = **25h** DL = **00h**

# Indirect Access with BX

```
1 MOV BX, 0500h      ; BX = address
2 MOV AL, [BX]       ; AL = [0500h]
▶ MOV DL, [0501h]    ; direct address
4 ADD AL, DL         ; AL = 25h + 17h
5 MOV [BX+2], AL     ; [0502h] = AL
```




The same kind of access works with a constant address: DL = [0501h] = **17h**.

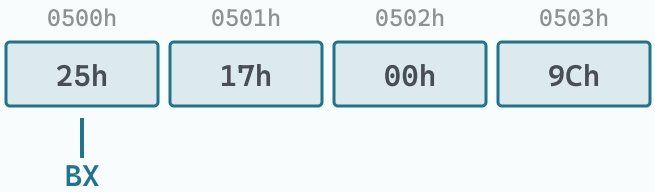
Step 3 / 5

VARIABLES

BX = **0500h** AL = **25h** DL = **17h**

# Indirect Access with BX

```
1 MOV BX, 0500h      ; BX = address
2 MOV AL, [BX]       ; AL = [0500h]
3 MOV DL, [0501h]    ; direct address
▶ ADD AL, DL         ; AL = 25h + 17h
5 MOV [BX+2], AL     ; [0502h] = AL
```




AL = 25h + 17h = **3Ch**.

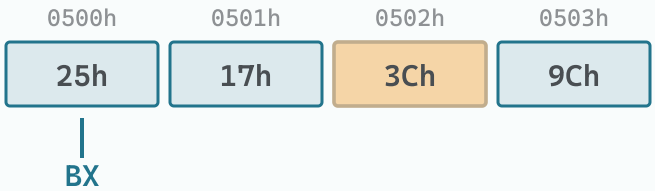
Step 4 / 5

VARIABLES

BX = **0500h** AL = **3Ch** DL = **17h**

# Indirect Access with BX

```
1 MOV BX, 0500h      ; BX = address
2 MOV AL, [BX]       ; AL = [0500h]
3 MOV DL, [0501h]    ; direct address
4 ADD AL, DL         ; AL = 25h + 17h
▶ MOV [BX+2], AL     ; [0502h] = AL
```




AL is written to `[BX+2]` = 0502h. Memory changed; BX did not.

Step 5 / 5

**VARIABLES**

BX = **0500h** AL = **3Ch** DL = **17h**

# CX — Counter

- 16 bits; splits into two 8-bit halves, **CH** and **CL**.
- Used for loops ( `LOOP` ), repeats ( `REP` ) and counting.
- `LOOP label` : first **CX ← CX − 1**, then jump to the label if CX ≠ 0. Flags are not affected.
- In shift and rotate instructions, a count greater than 1 goes in **CL**: `SHL AX, CL` .
- `JCXZ` : jump if CX = 0 (guards against an empty loop).

```
MOV CX, 0005h   ; 5 passes
NEXT:
  ; ... body ...
  LOOP NEXT      ; CX--, jump if CX≠0
```

Entering the loop with CX = 0 makes the first LOOP set CX to FFFFh, and the loop runs **65536** times.

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

CX = 5: the loop will run 5 times.

Step 1 / 13

VARIABLES

CX = 0005h  AX = 0000h  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
▶ MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

The sum starts at AX = 0.

Step 2 / 13

VARIABLES

CX = **0005h** AX = **0000h** DX = **0000h**

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
▶ AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

AX = AX + CX → 0 + 5 = **5**.

Step 3 / 13

VARIABLES

CX = **0005h** AX = **0005h** DX = **0000h**

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
▶ LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

LOOP: CX = 5 → 4; not zero, jump to **AGAIN**.

Step 4 / 13

VARIABLES

CX = 0004h  AX = 0005h  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
▶ AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

AX = AX + CX → 5 + 4 = **9**.

Step 5 / 13

VARIABLES

CX = 0004h  AX = 0009h  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
▶ LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

LOOP: CX = 4 → 3; not zero, jump to **AGAIN**.

Step 6 / 13

VARIABLES

CX = 0003h  AX = 0009h  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
▶ AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

AX = AX + CX → 9 + 3 = **12**.

Step 7 / 13

VARIABLES

CX = 0003h  AX = 000Ch  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
▶ LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

LOOP: CX = 3 → 2; not zero, jump to **AGAIN**.

Step 8 / 13

VARIABLES

CX = 0002h  AX = 000Ch  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
▶ AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

AX = AX + CX → 12 + 2 = **14**.

Step 9 / 13

VARIABLES

CX = 0002h  AX = 000Eh  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
▶ LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

LOOP: CX = 2 → 1; not zero, jump to **AGAIN**.

Step 10 / 13

VARIABLES

CX = 0001h  AX = 000Eh  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
▶ AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

AX = AX + CX → 14 + 1 = **15**.

Step 11 / 13

VARIABLES

CX = **0001h** AX = **000Fh** DX = **0000h**

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
▶ LOOP AGAIN         ; CX--, jump if ≠0
5 MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

LOOP: CX = 1 → **0**; no jump, the loop exits.

Step 12 / 13

VARIABLES

CX = 0000h  AX = 000Fh  DX = 0000h

# CX and LOOP: 5 + 4 + 3 + 2 + 1

```
1 MOV CX, 5          ; counter
2 MOV AX, 0          ; sum
3 AGAIN: ADD AX, CX
4 LOOP AGAIN         ; CX--, jump if ≠0
▶ MOV DX, AX         ; result
```

The loop body runs with CX = 5, 4, 3, 2, 1; it does **not** run with CX = 0.

The result is copied to DX: 5+4+3+2+1 = 15 = **000Fh**.

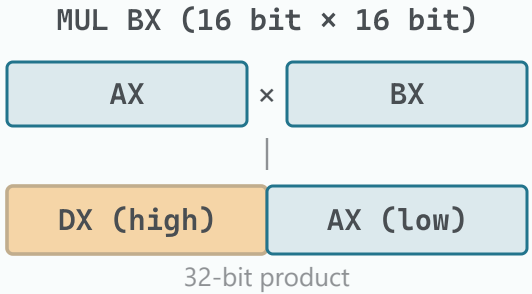
Step 13 / 13

VARIABLES

CX = **0000h** AX = **000Fh** DX = **000Fh**

# DX — Data

- 16 bits; splits into two 8-bit halves, **DH** and **DL**.
- A 16-bit multiply gives a 32-bit result: **DX** holds the high word, **AX** the low word → `DX:AX`.
- In a 16-bit divide the dividend is `DX:AX`; the quotient goes to AX, the remainder to DX.
- I/O port numbers above 255 are given in DX: `OUT DX, AL`.
- In DOS calls (INT 21h) it holds a string address: `LEA DX, msg`.




Example: `MUL BX` → `DX:AX = AX × BX`. The 8-bit `MUL BL` gives `AX = AL × BL` and leaves DX alone.

# DX:AX with MUL

```
▶ MOV AX, 1234h
2 MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
5 MOV BL, 2
6 MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

AX = 1234h.

Step 1 / 8

VARIABLES

AX = **1234h** BX = **0000h** DX = **0000h** CF = **0** OF = **0**

# DX:AX with MUL

```
1 MOV AX, 1234h
▶ MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
5 MOV BL, 2
6 MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

BX = 0100h (256).

Step 2 / 8

VARIABLES

AX = **1234h** BX = **0100h** DX = **0000h** CF = **0** OF = **0**

# DX:AX with MUL

```
1 MOV AX, 1234h
2 MOV BX, 0100h
▶ MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
5 MOV BL, 2
6 MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

1234h × 0100h = **00123400h** → DX = 0012h, AX = 3400h. The high half is nonzero, so CF = OF = 1.

Step 3 / 8

VARIABLES

AX = **3400h** BX = **0100h** DX = **0012h** CF = **1** OF = **1**

# DX:AX with MUL

```
1 MOV AX, 1234h
2 MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
▶ MOV AX, 00FFh
5 MOV BL, 2
6 MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

AX = 00FFh; the old result is still in DX.

Step 4 / 8

VARIABLES

AX = 00FFh | BX = 0100h | DX = 0012h | CF = 1 | OF = 1

# DX:AX with MUL

```
1 MOV AX, 1234h
2 MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
▶ MOV BL, 2
6 MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

BL = 02h (BX = 0102h, BH unchanged).

Step 5 / 8

VARIABLES

AX = **00FFh** | BX = **0102h** | DX = **0012h** | CF = **1** | OF = **1**

# DX:AX with MUL

```
1 MOV AX, 1234h
2 MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
5 MOV BL, 2
▶ MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

8-bit product: FFh × 02h = **01FEh** → AX. DX is unchanged; AH ≠ 0, so CF = OF = 1.

Step 6 / 8

VARIABLES

AX = **01FEh** BX = **0102h** DX = **0012h** CF = **1** OF = **1**

# DX:AX with MUL

```
1 MOV AX, 1234h
2 MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
5 MOV BL, 2
6 MUL BL          ; AX = AL*BL
▶ MOV DX, 03F8h   ; port address
8 OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

This time DX is a port number: 03F8h (the COM1 serial port).

Step 7 / 8

**VARIABLES**

AX = **01FEh** BX = **0102h** DX = **03F8h** CF = **1** OF = **1**

# DX:AX with MUL

```
1 MOV AX, 1234h
2 MOV BX, 0100h
3 MUL BX          ; DX:AX = AX*BX
4 MOV AX, 00FFh
5 MOV BL, 2
6 MUL BL          ; AX = AL*BL
7 MOV DX, 03F8h   ; port address
▶ OUT DX, AL      ; write to port
```

MUL is an unsigned multiply; use `IMUL` for signed. After MUL, SF, ZF, PF and AF are **undefined**.

FEh in AL is sent to port 03F8h. Ports above 255 require DX.

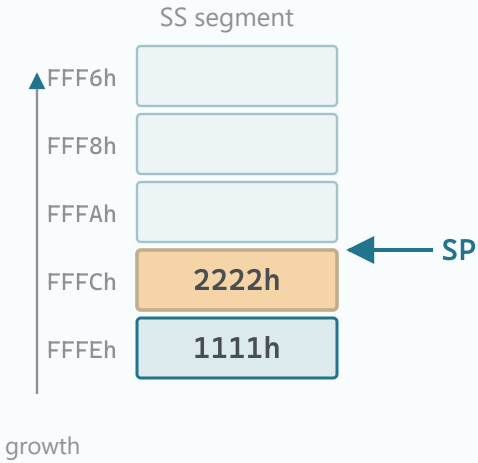
Step 8 / 8

**VARIABLES**

AX = **01FEh** BX = **0102h** DX = **03F8h** CF = **1** OF = **1**

# SP — Stack Pointer

- The 16-bit **stack pointer** holds the offset of the **top** item on the stack.
- The address is always relative to the **stack segment**: the physical address is `SS:SP`.
- The stack is **LIFO**: `PUSH` adds, `POP` removes; on the 8086 items are always **16 bits**.
- The stack grows **downward** in memory: `PUSH` first subtracts 2 from SP, then writes; `POP` first reads, then adds 2 to SP.
- "SP = FFFEh on an empty stack" holds for **.COM** programs (DOS/emu8086 set SP to FFFEh); it is not a general rule.



# BP — Base Pointer

- The 16-bit **base pointer** marks a **fixed position** on the stack.
- Memory accesses through BP use the **SS** segment by default: `[BP+4]` → `SS:BP+4`.
- Used to access a procedure's stack **parameters** and local variables.
- SP keeps changing with PUSH/POP; BP stays **fixed** during the procedure, so the parameter positions are known.

```
ADDP PROC
  PUSH BP
  MOV  BP, SP      ; frame base
  MOV  AX, [BP+4] ; 1st param
  POP  BP
  RET  2
ADDP ENDP
```

The classic procedure prologue: `PUSH BP` + `MOV BP, SP`.

# SI and DI — Index Registers

### SI — Source Index

- 16 bits; an offset in the data segment (**DS**).
- Points to the data to be read.
- The **source** in string instructions: `MOVSB`, `CMPSB`, `LODSB` → `DS:SI`.
- Example: `MOV SI, 2000h` — points SI at a memory location.

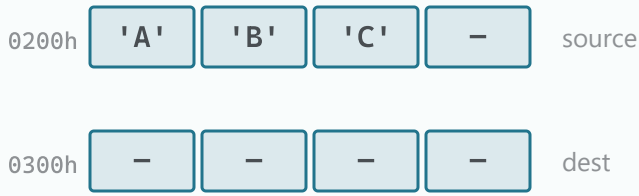
### DI — Destination Index

- 16 bits; in string instructions, an offset in the extra segment (**ES**).
- Points to where data will be written.
- The **destination** in string instructions: `MOVSW`, `STOSB`, `SCASB` → `ES:DI`.
- Example: `MOV DI, 3000h` — points DI at a memory location.

Outside string instructions, `[SI]` and `[DI]` are plain index registers using the **DS** segment. After each string step SI and DI move up or down by 1 (byte) or 2 (word), depending on the **DF** flag.

# Copying a String with SI and DI

```
▶ CLD                ; DF=0: addresses go up
2 MOV SI, 0200h      ; source
3 MOV DI, 0300h      ; destination
4 MOV CX, 3          ; byte count
5 REP MOVSB          ; ES:[DI] ← DS:[SI]
```




CLD: DF = 0 → SI and DI will **increase** on each step.

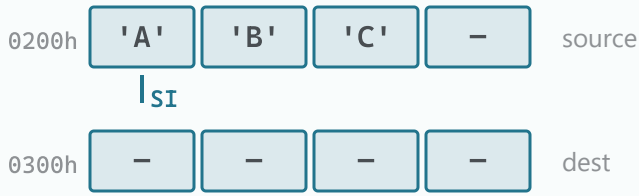
Step 1 / 7

**VARIABLES**

SI = **0000h** DI = **0000h** CX = **0000h** DF = **0**

# Copying a String with SI and DI

```
1 CLD                ; DF=0: addresses go up
▶ MOV SI, 0200h      ; source
3 MOV DI, 0300h      ; destination
4 MOV CX, 3          ; byte count
5 REP MOVSB          ; ES:[DI] ← DS:[SI]
```




SI = 0200h: start of the string “ABC” to copy.

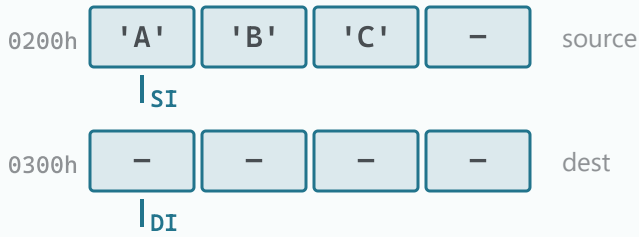
Step 2 / 7

**VARIABLES**

SI = 0200h  DI = 0000h  CX = 0000h  DF = 0

# Copying a String with SI and DI

```
1 CLD                ; DF=0: addresses go up
2 MOV SI, 0200h      ; source
▶ MOV DI, 0300h      ; destination
4 MOV CX, 3          ; byte count
5 REP MOVSB          ; ES:[DI] ← DS:[SI]
```




DI = 0300h: start of the destination area.

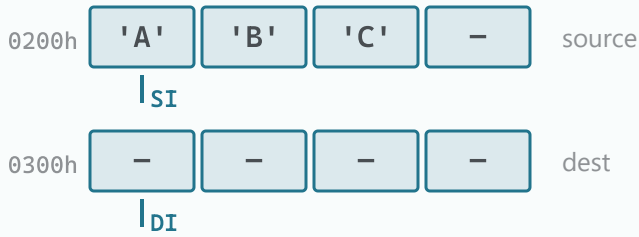
Step 3 / 7

**VARIABLES**

SI = 0200h  DI = 0300h  CX = 0000h  DF = 0

# Copying a String with SI and DI

```
1 CLD                ; DF=0: addresses go up
2 MOV SI, 0200h      ; source
3 MOV DI, 0300h      ; destination
▶ MOV CX, 3          ; byte count
5 REP MOVSB          ; ES:[DI] ← DS:[SI]
```




CX = 3: REP repeats at most 3 times.

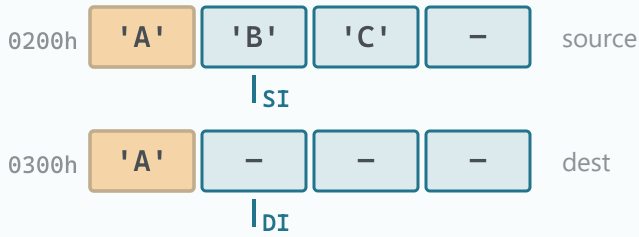
Step 4 / 7

**VARIABLES**

SI = 0200h  DI = 0300h  CX = 0003h  DF = 0

# Copying a String with SI and DI

```
1 CLD                ; DF=0: addresses go up
2 MOV SI, 0200h      ; source
3 MOV DI, 0300h      ; destination
4 MOV CX, 3          ; byte count
▶ REP MOVSB          ; ES:[DI] ← DS:[SI]
```




Pass 1: 'A' is copied; SI = 0201h, DI = 0301h, CX = 2.

Step 5 / 7

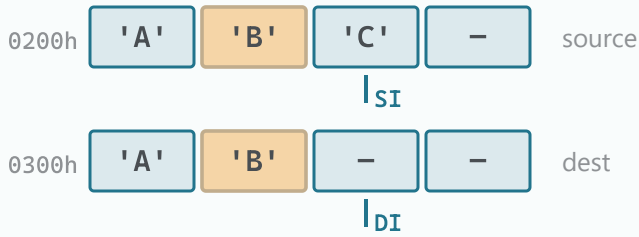
**VARIABLES**

SI = 0201h  DI = 0301h  CX = 0002h

DF = 0

# Copying a String with SI and DI

```
1 CLD                ; DF=0: addresses go up
2 MOV SI, 0200h      ; source
3 MOV DI, 0300h      ; destination
4 MOV CX, 3          ; byte count
▶ REP MOVSB          ; ES:[DI] ← DS:[SI]
```




Pass 2: 'B' is copied; SI = 0202h, DI = 0302h, CX = 1.

Step 6 / 7

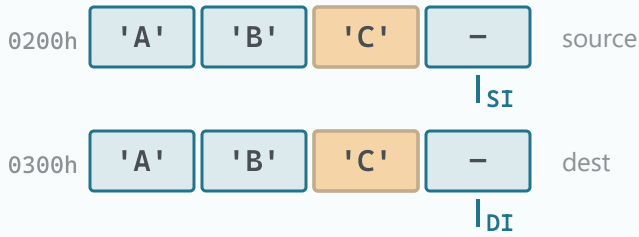
VARIABLES

SI = 0202h  DI = 0302h  CX = 0001h

DF = 0

# Copying a String with SI and DI

```
1 CLD                ; DF=0: addresses go
  up
2 MOV SI, 0200h      ; source
3 MOV DI, 0300h      ; destination
4 MOV CX, 3          ; byte count
▶ REP MOVSB          ; ES:[DI] ← DS:[SI]
```




Pass 3: 'C' is copied; CX = 0 → REP ends.

Step 7 / 7

**VARIABLES**

SI = 0203h  DI = 0303h  CX = 0000h

DF = 0

# Segment Registers and IP

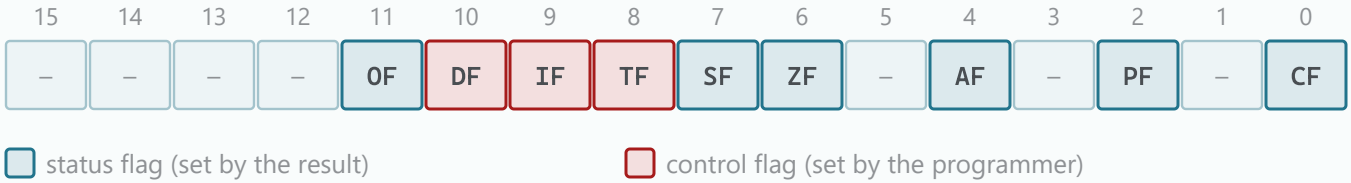
- **CS** (code), **DS** (data), **SS** (stack), **ES** (extra): each holds the start of a 64 KB segment.
- **IP** (instruction pointer): the offset within CS of the **next instruction** to execute.
- IP cannot be written with `MOV`; only JMP, CALL, RET and interrupts change it.
- A segment register cannot take an immediate value: you need `MOV DS, AX`.

| Pair | Use |
|---|---|
| `CS:IP` | next instruction |
| `SS:SP` | top of stack |
| `SS:BP` | stack frame |
| `DS:BX`, `DS:SI`, `DS:DI` | data |
| `DS:SI → ES:DI` | string instructions |

**Physical address** (20 bits) = segment × 16 + offset. Multiplying by 16 appends a **0** to the hex number: with DS = 2000h and BX = 0500h, `[BX]` → 20000h + 0500h = **20500h**.

# The FLAGS Register

- Also called the **status register**; **9** of its 16 bits are used.
- Describes the result of the **last arithmetic/logic instruction** executed.
- Determines whether **conditional jumps** (JZ, JC, JG ...) are taken.




**6 status flags**

CF, PF, AF, ZF, SF, OF: set (1) or cleared (0) by the hardware according to the result.

**3 control flags**

DF, IF, TF: change *how the processor runs*; the programmer sets them with instructions such as CLD/STD and CLI/STI.

# SF — Sign Flag

- The sign flag tells whether the result is **negative or positive**.
- It is a copy of the result's **most significant bit** (MSB):
- MSB = 1 → the number is negative (read as signed) → **SF = 1**.
- MSB = 0 → the number is positive or zero → **SF = 0**.
- The MSB is bit 7 for 8 bits, bit 15 for 16 bits.

| Result (8 bits) | MSB | SF |
|---|---|---|
| `00h … 7Fh` | 0 | 0 |
| `80h … FFh` | 1 | 1 |
| `35h` = 0011 0101b | 0 | 0 |
| `C2h` = 1100 0010b | 1 | 1 |

The processor does not know whether a number is signed or unsigned; SF is just a copy of the MSB. Reading the result as two's complement (signed) is the programmer's job.

# ZF and PF

### ZF — Zero Flag

- Tells whether the result is **zero**.
- If the result is 0, **ZF = 1**; otherwise ZF = 0.
- `CMP AX, BX` sets ZF = 1 on equality → `JE` / `JZ` jumps.
- Example: `SUB AL, AL` → AL = 00h, ZF = 1.

### PF — Parity Flag

- PF = 1 if the number of 1s in the **low 8 bits** of the result is **even**; PF = 0 if it is odd.
- Even in a 16-bit operation only the low byte counts.
- Example: 03h = 0000 0011b → two 1s → PF = 1.
- Example: 07h = 0000 0111b → three 1s → PF = 0.




ZF = 1 means “the result is zero”, so the flag and the result look inverted. PF was once used for error checking in serial communication.

# AF — Auxiliary Carry Flag

- Auxiliary carry flag: **AF = 1** if a carry (in addition) or borrow (in subtraction) passes **from bit 3 to bit 4**; otherwise 0.
- In other words, it shows a carry out of the low **nibble** (4 bits).
- Specific to **BCD** (binary-coded decimal, each digit 0–9) arithmetic: `DAA`, `DAS`, `AAA` and `AAS` use this flag.




There is **no conditional jump on AF** and no instruction that sets or clears it directly; still, `LAHF` and `PUSHF` can read it, `SAHF` and `POPF` can write it.

# CF — Carry Flag

- Carry flag: in an n-bit operation, **CF = 1** if the result **does not fit in n bits** (a carry leaves the MSB); otherwise 0.
- A **borrow** in subtraction (smaller minus larger) also sets CF = 1.
- It signals overflow for **unsigned** numbers.
- In shifts and rotates the bit shifted out goes to CF.
- The programmer can change it too: `STC` (1), `CLC` (0), `CMC` (complement).
- In multi-byte addition, `ADC` carries it into the next byte.

| Operation (8 bits) | Result | CF |
|---|---|---|
| `FFh + 01h` | 00h | 1 |
| `80h + 80h` | 00h | 1 |
| `70h + 10h` | 80h | 0 |
| `05h - 07h` | FEh | 1 (borrow) |
| `07h - 05h` | 02h | 0 |

# OF — Overflow Flag

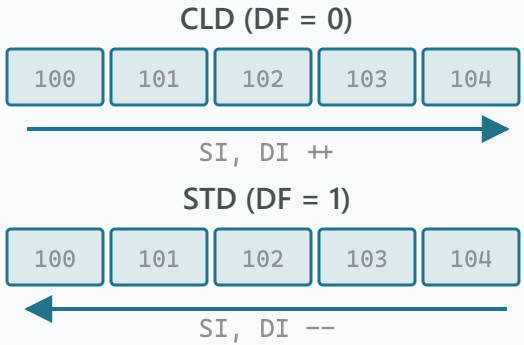
- Overflow flag: OF = 1 if the result, read as **signed**, **does not fit** in the operand size; otherwise 0.
- The signed range is −128 … +127 for 8 bits and −32768 … +32767 for 16 bits.
- Rule for addition: OF = 1 when two **positives** give a negative, or two **negatives** give a positive. Adding numbers of opposite sign never overflows.
- In hardware: OF = 1 when the carry **into** the MSB differs from the carry **out of** it.

| Operation (8 bits) | Unsigned view | Signed view | Result | CF | OF |
|---|---|---|---|---|---|
| `05h + 03h` | 5 + 3 = 8 | 5 + 3 = 8 | 08h | 0 | 0 |
| `7Fh + 01h` | 127 + 1 = 128 | 127 + 1 ≠ −128 | 80h | 0 | **1** |
| `FFh + 01h` | 255 + 1 ≠ 0 | −1 + 1 = 0 | 00h | **1** | 0 |
| `80h + 80h` | 128 + 128 ≠ 0 | −128 + (−128) ≠ 0 | 00h | **1** | **1** |

CF signals unsigned overflow, OF signed overflow; the two are independent.

# DF — Direction Flag

- The direction flag sets the direction in which **string** instructions step through memory.
- **DF = 0**: low to high addresses; SI/DI **increase** on each step.
- **DF = 1**: high to low addresses; SI/DI **decrease**.
- The step is 1 for byte instructions and 2 for word instructions.
- `CLD` → DF = 0, `STD` → DF = 1.
- Direction matters when copying overlapping areas: if the destination lies ahead of the source, copy **from the end** (STD).



# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
▶ mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
6 rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```

arr (010Eh–0112h)




**DF = 0 · CX = 0**

"KLMN" will move one place right; source and destination **overlap**. SI = 0111h: the **last** source byte.

Step 1 / 9

VARIABLES

SI = **0111h** DI = **0000h** CX = **0** DF = **0**

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
▶ mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
6 rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```

arr (010Eh–0112h)




**DF = 0 · CX = 0**

DI = 0112h: the last destination byte (one to the right).

Step 2 / 9

VARIABLES

SI = **0111h** DI = **0112h** CX = **0** DF = **0**

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
▶ mov cx, 4
5 std                  ; DF = 1
6 rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```

arr (010Eh–0112h)




`DF = 0 · CX = 4`

CX = 4 bytes to copy.

Step 3 / 9

**VARIABLES**

`SI = 0111h` `DI = 0112h` `CX = 4` `DF = 0`

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
▶ std                  ; DF = 1
6 rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```

arr (010Eh–0112h)




`DF = 1 · CX = 4`

`std`: **DF = 1**; string instructions will **decrement** SI and DI on each step.

Step 4 / 9

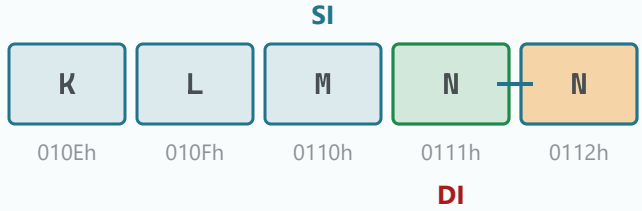
**VARIABLES**

`SI = 0111h` `DI = 0112h` `CX = 4` `DF = 1`

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
▶ rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```

arr (010Eh–0112h)




**DF = 1 · CX = 3**

MOVSB 1: [0111h] → [0112h] ('N'); SI, DI down by 1, CX = 3.

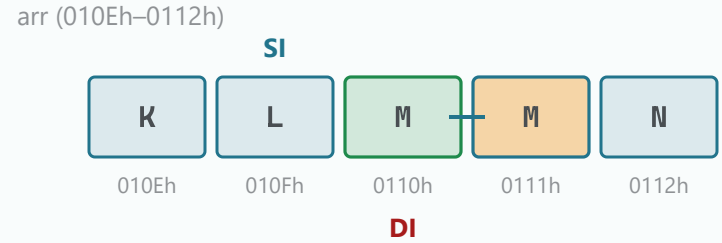
Step 5 / 9

VARIABLES

SI = **0110h** DI = **0111h** CX = **3** DF = **1**

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
▶ rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```




`DF = 1 · CX = 2`

MOVSB 2: [0110h] → [0111h] ('M'); SI, DI down by 1, CX = 2.

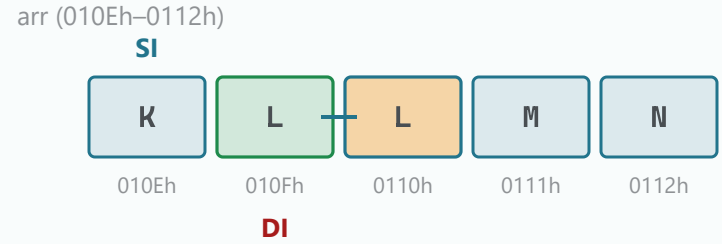
Step 6 / 9

**VARIABLES**

`SI = 010Fh` `DI = 0110h` `CX = 2` `DF = 1`

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
▶ rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```




`DF = 1 · CX = 1`

MOVSB 3: [010Fh] → [0110h] ('L'); SI, DI down by 1, CX = 1.

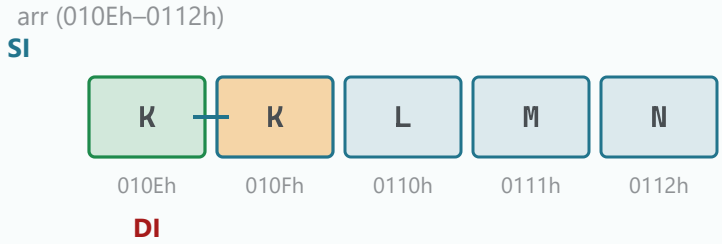
Step 7 / 9

**VARIABLES**

`SI = 010Eh` `DI = 010Fh` `CX = 1` `DF = 1`

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
▶ rep movsb
7 cld
8 ret
9 arr db 'KLMN', 0
```




`DF = 1 · CX = 0`

MOVSB 4: [010Eh] → [010Fh] ('K'); SI, DI down by 1, CX = 0. Result **KKLMN**: the array shifted correctly.

Step 8 / 9

VARIABLES

`SI = 010Dh` `DI = 010Eh` `CX = 0` `DF = 1`

# Code Trace: Overlapping Copy with DF = 1

```
1 org 100h
2 mov si, offset arr+3 ; source
3 mov di, offset arr+4 ; destination
4 mov cx, 4
5 std                  ; DF = 1
6 rep movsb
▶ cld
8 ret
9 arr db 'KLMN', 0
```




`DF = 0 · CX = 0`

`cld` restores DF = 0. Copying forward with CLD would carry 'K' into every byte and give **KKKKK**: when the destination lies ahead of the source, copy backward.

Step 9 / 9

VARIABLES

`SI = 010Dh` `DI = 010Eh` `CX = 0` `DF = 0`

# IF and TF

### IF — Interrupt Flag

- Controls external hardware **interrupts** (the INTR input).
- **IF = 1**: maskable interrupt requests from peripherals are serviced.
- **IF = 0**: these requests are ignored (masked).
- `STI` → IF = 1, `CLI` → IF = 0.
- NMI (non-maskable interrupt) and software interrupts ( `INT n` ) are **not affected** by IF.

### TF — Trap Flag

- Used for on-chip **debugging**.
- **TF = 1**: single-step mode; after each instruction an internal **INT 1** occurs and the single-step ISR (interrupt service routine) runs.
- This lets you examine a program one instruction at a time (DEBUG, emu8086 “single step”).
- **TF = 0**: normal execution.
- There is no direct instruction; use `PUSHF` → change bit 8 → `POPF` .



# Instructions That Change Flags

| Instruction | Effect | Instruction | Effect |
|---|---|---|---|
| `CLC` | CF = 0 | `CLD` | DF = 0 |
| `STC` | CF = 1 | `STD` | DF = 1 |
| `CMC` | CF = ¬CF | `CLI` / `STI` | IF = 0 / 1 |
| `LAHF` | AH ← SF ZF – AF – PF – CF | `SAHF` | AH → low flag byte |
| `PUSHF` | push FLAGS | `POPF` | pop into FLAGS |

| Instruction group | Effect on flags |
|---|---|
| ADD, ADC, SUB, SBB, CMP, NEG | CF, PF, AF, ZF, SF, OF all updated |
| INC, DEC | all updated except CF |
| AND, OR, XOR, TEST | CF = OF = 0; PF, ZF, SF updated; AF undefined |
| MOV, PUSH, POP, XCHG, LEA, LOOP, JMP | flags **not affected** |

# Code Trace: Reading and Writing Flags

```
1  org 100h
▶ mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
4  lahf             ; AH ← flags
5  cmc              ; flip CF
6  clc              ; CF = 0
7  stc              ; CF = 1
8  mov ah, 0C1h     ; 1100 0001b
9  sahf             ; flags ← AH
10 ret
```

flags (low byte)

| SF | ZF | – | AF | – | PF | – | CF |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

AH

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**AH = 00h**

AL = FFh. Low byte of the flags: SF ZF – AF – PF – CF.

Step 1 / 8

**VARIABLES**

AH = **00h** SF = **0** ZF = **0** AF = **0** PF = **0** CF = **0**

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
▶  add al, 1          ; CF ZF AF PF = 1
4  lahf               ; AH ← flags
5  cmc                ; flip CF
6  clc                ; CF = 0
7  stc                ; CF = 1
8  mov ah, 0C1h       ; 1100 0001b
9  sahf               ; flags ← AH
10 ret
```

flags (low byte)

| SF | ZF | – | AF | – | PF | – | CF |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

AH

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

**AH = 00h**

FFh + 1 = 00h: **CF = 1**, **ZF = 1**, AF = 1 (F + 1), PF = 1 (00h has zero 1s, even).

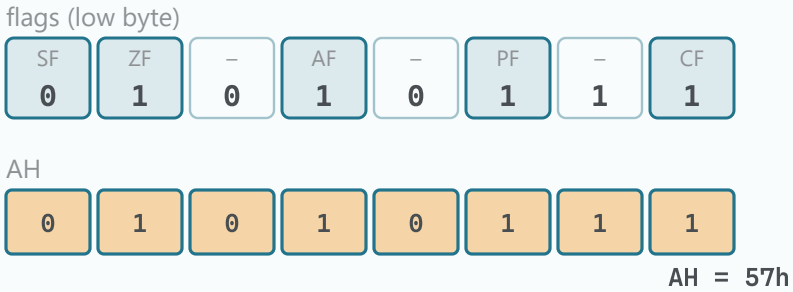
Step 2 / 8

**VARIABLES**

AH = **00h** · SF = **0** · ZF = **1** · AF = **1** · PF = **1** · CF = **1**

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
▶  lahf             ; AH ← flags
5  cmc              ; flip CF
6  clc              ; CF = 0
7  stc              ; CF = 1
8  mov ah, 0C1h     ; 1100 0001b
9  sahf             ; flags ← AH
10 ret
```




`lahf`: the low flag byte is copied to AH → AH = **57h** = 0101 0111b (bit 1 is always 1).

Step 3 / 8

**VARIABLES**

AH = **57h** | SF = **0** | ZF = **1** | AF = **1** | PF = **1** | CF = **1**

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
4  lahf             ; AH ← flags
▶  cmc              ; flip CF
6  clc              ; CF = 0
7  stc              ; CF = 1
8  mov ah, 0C1h     ; 1100 0001b
9  sahf             ; flags ← AH
10 ret
```

flags (low byte)

| SF | ZF | – | AF | – | PF | – | CF |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

AH

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

**AH = 57h**

`cmc` : CF flips, 1 → **0**. The other flags stay.

Step 4 / 8

**VARIABLES**

`AH = 57h` `SF = 0` `ZF = 1` `AF = 1` `PF = 1` `CF = 0`

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
4  lahf             ; AH ← flags
5  cmc              ; flip CF
▶  clc              ; CF = 0
7  stc              ; CF = 1
8  mov ah, 0C1h     ; 1100 0001b
9  sahf             ; flags ← AH
10 ret
```

flags (low byte)

| SF | ZF | – | AF | – | PF | – | CF |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

AH

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

**AH = 57h**

`clc` : CF = 0 (it already was).

Step 5 / 8

**VARIABLES**

`AH = 57h` `SF = 0` `ZF = 1` `AF = 1` `PF = 1` `CF = 0`

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
4  lahf             ; AH ← flags
5  cmc              ; flip CF
6  clc              ; CF = 0
▶  stc              ; CF = 1
8  mov ah, 0C1h     ; 1100 0001b
9  sahf             ; flags ← AH
10 ret
```

flags (low byte)

| SF | ZF | – | AF | – | PF | – | CF |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

AH

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

**AH = 57h**

`stc` : **CF = 1**. Sets up the carry before multi-byte ADC/SBB.

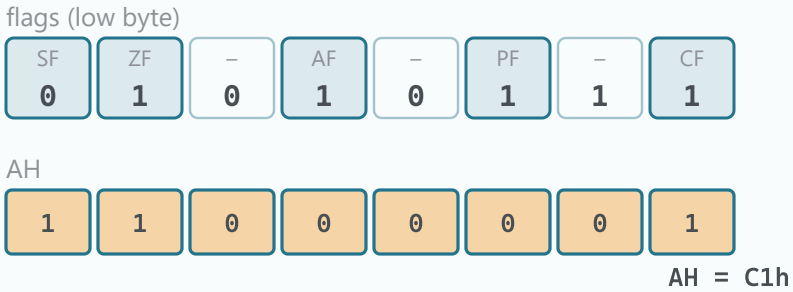
Step 6 / 8

**VARIABLES**

AH = **57h** · SF = **0** · ZF = **1** · AF = **1** · PF = **1** · CF = **1**

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
4  lahf             ; AH ← flags
5  cmc              ; flip CF
6  clc              ; CF = 0
7  stc              ; CF = 1
▶  mov ah, 0C1h     ; 1100 0001b
9  sahf             ; flags ← AH
10 ret
```




AH = C1h = 1100 0001b: bit 7 (SF), bit 6 (ZF), bit 0 (CF) = 1.

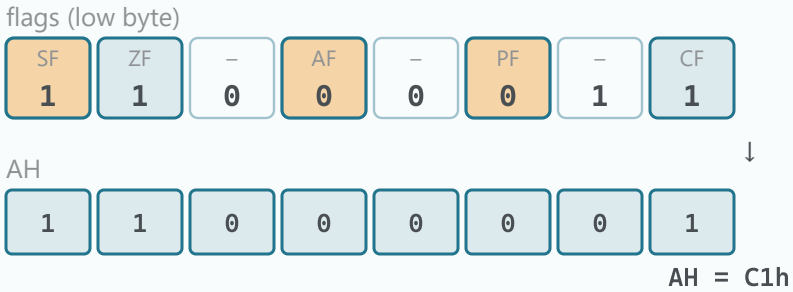
Step 7 / 8

**VARIABLES**

AH = **C1h** SF = **0** ZF = **1** AF = **1** PF = **1** CF = **1**

# Code Trace: Reading and Writing Flags

```
1  org 100h
2  mov al, 0FFh
3  add al, 1        ; CF ZF AF PF = 1
4  lahf             ; AH ← flags
5  cmc              ; flip CF
6  clc              ; CF = 0
7  stc              ; CF = 1
8  mov ah, 0C1h     ; 1100 0001b
▶  sahf             ; flags ← AH
10 ret
```




`sahf` : AH is written to the low flag byte → **SF = 1, ZF = 1, CF = 1**, AF = PF = 0. OF, DF, IF and TF are not affected.

Step 8 / 8

**VARIABLES**

AH = **C1h** | SF = **1** | ZF = **1** | AF = **0** | PF = **0** | CF = **1**

# Lab: Registers and Flags

AX = AH : AL

**CODE · 14 bytes**

```
 1 ORG 100h
 2 MOV AX, 1234h    ; AH=12h, …
 3 MOV AL, 0FFh     ; low half…
 4 ADD AL, 1        ; 8-bit add
 5 MOV AX, 12FFh
 6 ADD AX, 1        ; 16-bit a…
 7 RET
```

**REGISTERS**

| | | | | | |
|---|---|---|---|---|---|
| AX | 0000 | AH | 00 | AL | 00 |
| BX | 0000 | BH | 00 | BL | 00 |
| CX | 000E | CH | 00 | CL | 0E |
| DX | 0000 | DH | 00 | DL | 00 |

| | | | | | |
|---|---|---|---|---|---|
| SI | 0000 | DI | 0000 | BP | 0000 |
| SP | FFFE | CS | 0700 | IP | 0100 |
| DS | 0700 | ES | 0700 | SS | 0700 |

**FLAGS**

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

CS:IP = 0700:0100 · step 0

**MEMORY DS = 0700h**

```
0100 B8 34 12 B0 FF 04 01 B8
0108 FF 12 05 01 00 C3 00 00
0110 00 00 00 00 00 00 00 00
0118 00 00 00 00 00 00 00 00
0120 00 00 00 00 00 00 00 00
```

**STACK SS:SP**

```
FFF6 0000
FFF8 0000
FFFA 0000
FFFC 0000
FFFE 0000 ◀ SP
```

**OUTPUT**

Program loaded: 14 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (000Eh). The Step button executes instructions one at a time; the highlighted line is the next instruction.

# Summary

### Registers

AX, BX, CX and DX are data registers that split into halves; SP/BP serve the stack, SI/DI strings and indexing, CS/DS/SS/ES the segments, and IP holds the next instruction.

### Implicit roles

MUL/DIV → AX, DX · LOOP/REP → CX · `[BX]` → DS · `[BP]` → SS · strings → DS:SI, ES:DI · ports → DX.

### Flags

CF unsigned and OF signed overflow; ZF zero; SF the MSB; PF parity of the low byte; AF nibble carry. DF, IF and TF are control flags.

### Register pairs

CS:IP next instruction, SS:SP top of stack, SS:BP stack frame, DS:SI → ES:DI string copy. A segment register cannot take a constant.

SUMMARY

# Exercises

1. With AX = 3A7Fh, what is AX after `MOV AL, 0` followed by `INC AH`?
2. Find AL, CF, ZF, SF, OF and PF after `MOV AL, 9Ch` / `ADD AL, 64h`.
3. `MOV AL, 50h` / `ADD AL, 50h`: what are CF and OF? Why do they differ?
4. With SP = 0100h, what is SP after 3 PUSHes and 1 POP?
5. How many times does a LOOP loop run after `MOV CX, 0`?

▶ **Answers**

# References


- Intel, *The 8086 Family User's Manual*, 1979.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009.
- Muhammad Ali Mazidi, Janice G. Mazidi, *The x86 PC: Assembly Language, Design, and Interfacing*, 5th ed., 2010.
- emu8086 documentation: the *8086 instructions* and *Addressing modes* sections.
- Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz* (in Turkish).

# The End

## Chapter 2.2: 8086 Registers

Previous: Chapter 2.1: 8086 Architecture · Next: Chapter 3.1: Programming Languages