Chapter 11: Instruction Encoding and String Instructions

Microprocessors

Instruction classes · Machine code · ModR/M · MOV forms · XLAT · LEA · LDS · LES · String instructions · REP prefixes · JMP encoding · Processor control · Flag effects

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_11_Instruction_Encoding_and_String_Instructions.html

Learning Outcomes

Instruction classes
Sort the 8086's roughly 100 instructions into six functional classes and describe what each class is for.

Machine code
Work out an instruction's machine code from its opcode, D/W bits and ModR/M byte.

String instructions
Use MOVS, CMPS, SCAS, LODS and STOS with REP prefixes; track how DF, SI, DI and CX change.

Reference
Know XLAT, LEA, LDS, LES and the processor control instructions; read from the flag table which flags an instruction changes.

The 8086 Instruction Set at a Glance

1. Data transfer

Copy data between registers, memory, the stack and I/O.

`MOV PUSH POP XCHG IN OUT XLAT LEA LDS LES LAHF SAHF PUSHF POPF`

2. Arithmetic

The four operations, BCD adjust, sign extension.

`ADD ADC INC AAA DAA SUB SBB DEC NEG CMP AAS DAS MUL IMUL AAM DIV IDIV AAD CBW CWD`

3. Logic, shift, rotate

Bit-level operations.

`NOT AND OR XOR TEST SHL SAL SHR SAR ROL ROR RCL RCR`

4. String processing

Scan and copy memory blocks through SI/DI.

`MOVSB/W CMPSB/W SCASB/W LODSB/W STOSB/W REP REPE REPZ REPNE REPNZ`

5. Control transfer

Branches, loops, procedures and interrupts.

`JMP Jcc (JE JNE JA JB JG JL …) LOOP LOOPE LOOPNE JCXZ CALL RET RETF INT INTO IRET`

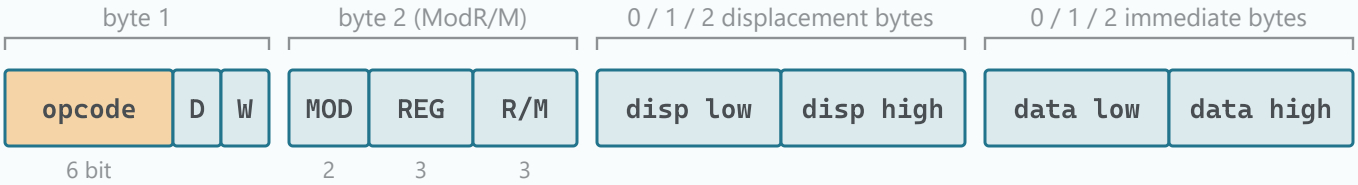
6. Processor control

Set/clear flags and control the processor state.

`CLC STC CMC CLD STD CLI STI HLT WAIT ESC LOCK NOP`

INSTRUCTION ENCODING

Machine Code Format



Instructions are 1–6 bytes long; prefixes (segment, REP, LOCK) can go in front.

D (direction)
1: the REG field is the **destination**; 0: the REG field is the **source**.

W (word)
0: byte (8-bit) operation; 1: word (16-bit) operation.

ModR/M
MOD says whether the operand is a register or memory, R/M picks the addressing form, REG the other register.

Common instructions have shorter special forms: `inc ax` = **40h** (1 byte), `mov ax, imm16` = **B8h** + 2 bytes, `add al, imm8` = **04h** + 1 byte.

The REG and MOD Fields

REG	W = 0	W = 1	SREG
`000`	AL	AX	ES (00)
`001`	CL	CX	CS (01)
`010`	DL	DX	SS (10)
`011`	BL	BX	DS (11)
`100`	AH	SP	
`101`	CH	BP	
`110`	DH	SI	
`111`	BH	DI	

MOD	Meaning
`00`	Memory, no displacement (R/M = 110 means a **direct** 16-bit address)
`01`	Memory + 8-bit displacement (sign-extended)
`10`	Memory + 16-bit displacement
`11`	Register: R/M is a REG code too

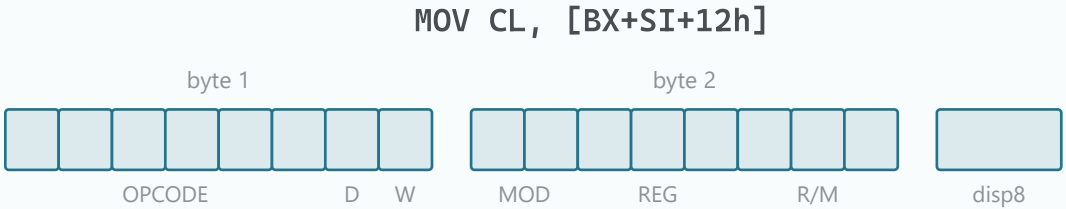
Example: `mov ax, bx` → **8B C3**: 100010 **1 1** | **11** 000 011 (D = 1, destination REG = AX, R/M = BX). The same instruction can also be encoded as **89 D8** (D = 0).

The R/M Field: Effective Address (EA)

R/M	MOD = 00	MOD = 01	MOD = 10	Default segment
`000`	[BX+SI]	[BX+SI+d8]	[BX+SI+d16]	DS
`001`	[BX+DI]	[BX+DI+d8]	[BX+DI+d16]	DS
`010`	[BP+SI]	[BP+SI+d8]	[BP+SI+d16]	**SS**
`011`	[BP+DI]	[BP+DI+d8]	[BP+DI+d16]	**SS**
`100`	[SI]	[SI+d8]	[SI+d16]	DS
`101`	[DI]	[DI+d8]	[DI+d16]	DS
`110`	**[d16] direct**	[BP+d8]	[BP+d16]	DS / **SS**
`111`	[BX]	[BX+d8]	[BX+d16]	DS

- Only **BX, BP, SI, DI** can be address registers; `[ax]`, `[cx]`, `[si+di]` are invalid.
- For `[bp]` the assembler emits `[bp+0]` (MOD = 01, d8 = 0), because MOD = 00, R/M = 110 is reserved for a direct address.
- A segment prefix overrides the default: `mov al, es:[bx]` → prefix **26h**.

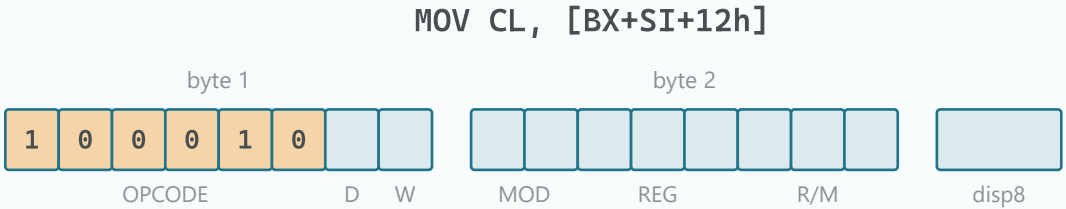
Encoding Step by Step: `MOV CL, [BX+SI+12h]`



Instruction: move a byte from memory into CL. Form: **MOV reg, r/m** (opcode + ModR/M + displacement).

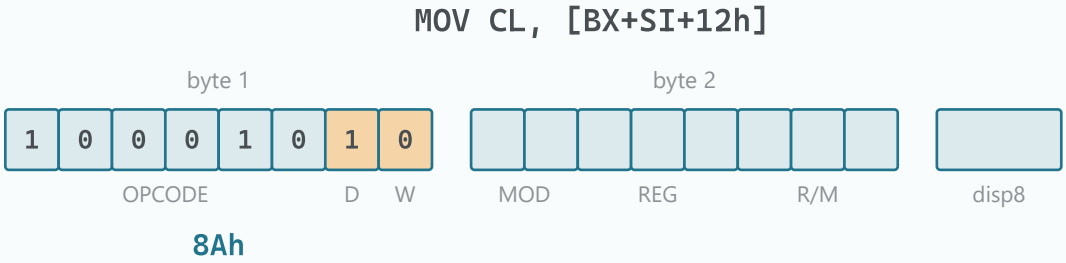
Step 1 / 5

Encoding Step by Step: `MOV CL, [BX+SI+12h]`



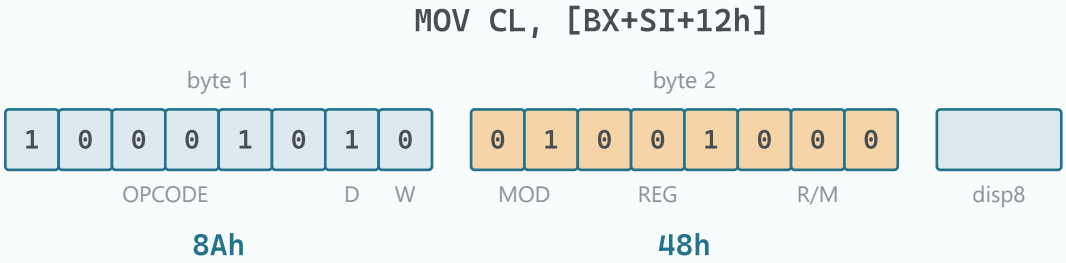
Opcode of the MOV r/m ↔ reg family: **100010**.

Encoding Step by Step: `MOV CL, [BX+SI+12h]`



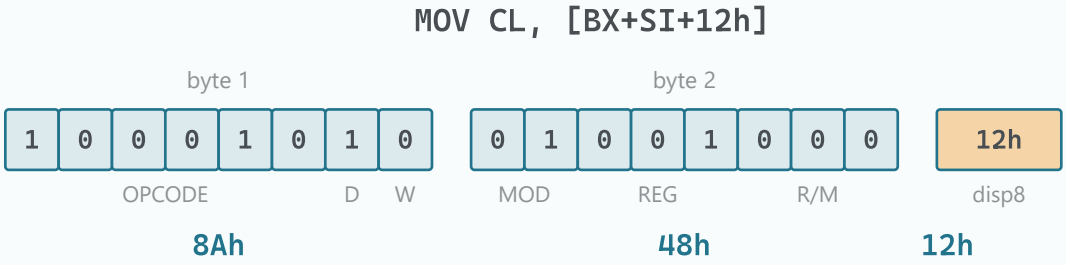
D = 1: the REG field is the **destination**. **W = 0**: byte operation. Byte 1 = 1000 1010b = **8Ah**.

Encoding Step by Step: `MOV CL, [BX+SI+12h]`



MOD = 01: memory + 8-bit displacement. **REG = 001** (CL). **R/M = 000** ([BX+SI]). Byte 2 = 01 001 000b = **48h**.

Encoding Step by Step: `MOV CL, [BX+SI+12h]`



Machine code: 8A 48 12 (3 bytes)

12h is appended as an 8-bit displacement. Result: **8A 48 12**. With a 16-bit displacement it would be MOD = 10 and 4 bytes.

Sample Instructions and Their Lengths

Instruction	Machine code	Bytes	Note
`nop`	`90`	1	really `xchg ax, ax`
`inc ax` / `push bx`	`40` / `53`	1	register code inside the opcode (40h+r, 50h+r)
`mov al, 5`	`B0 05`	2	B0h+r, immediate byte
`int 21h`	`CD 21`	2	interrupt number in the instruction
`mov ax, 1234h`	`B8 34 12`	3	low byte first (little-endian)
`mov cl, [bx+si+12h]`	`8A 48 12`	3	ModR/M + d8
`mov es:[bx], al`	`26 88 07`	3	26h = ES prefix
`add [bx+di+1000h], dx`	`01 91 00 10`	4	MOD = 10, d16
`mov word ptr [1234h], 5678h`	`C7 06 34 12 78 56`	6	longest form: d16 + imm16
`rep movsb`	`F3 A4`	2	F3h = REP prefix

Prefixes: 26h ES, 2Eh CS, 36h SS, 3Eh DS (segment); F3h REP/REPE, F2h REPNE; F0h LOCK. To see machine code in emu8086, open the listing (.lst) file after assembling.

Encoding Forms of MOV

Form	Opcode (8 / 16 bit)	Followed by	Example	Code
register → r/m	`88` / `89`	ModR/M (+ disp)	`mov [bx], al`	`88 07`
r/m → register	`8A` / `8B`	ModR/M (+ disp)	`mov cx, [si]`	`8B 0C`
immediate → r/m	`C6` / `C7`	ModR/M (REG = 000) + disp + imm	`mov byte ptr [bx], 5`	`C6 07 05`
immediate → register	`B0+r` / `B8+r`	imm8 / imm16	`mov si, 1000h`	`BE 00 10`
memory → AL / AX	`A0` / `A1`	16-bit address	`mov ax, [2000h]`	`A1 00 20`
AL / AX → memory	`A2` / `A3`	16-bit address	`mov [1000h], al`	`A2 00 10`
segment register → r/m16	`8C`	ModR/M (REG = sreg)	`mov ax, es`	`8C C0`
r/m16 → segment register	`8E`	ModR/M (REG = sreg)	`mov ds, ax`	`8E D8`

One instruction can have several encodings: `mov al, [1000h]` = **A0 00 10** (3 bytes) or **8A 06 00 10** (4 bytes). The assembler picks the short one.

Encoding Forms of MOV (continued)

Segment register codes (sreg)

ES = **00**, CS = **01**, SS = **10**, DS = **11**; they go into the ModR/M REG field as 0 + sreg. `mov ds, ax`: **11 011 000** = D8h.

Forms that don't exist

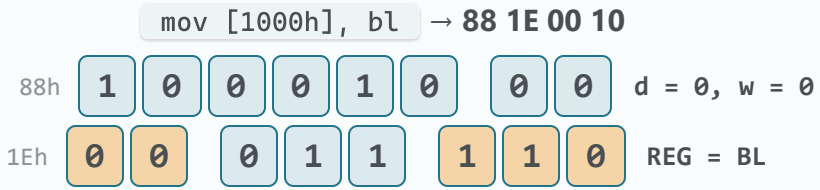
No immediate → segment register (`mov ds, 1234h` is invalid), no memory → memory. `mov cs, …` is not used; CS changes only through JMP/CALL/RET/INT.

`mov ds, ax` → `8E D8`



ES 00 · CS 01 · SS 10 · DS 11 (REG field = 0 + sreg)

MOD = 00, R/M = 110: Not [BP] but [d16]



MOD = 00 and R/M = 110 → a **2-byte address** follows: 00 10 = 1000h. The instruction is **4 bytes**. For [BP] the assembler uses MOD = 01, d8 = 0.

Instruction	Code	Note
`mov [1000h], al`	`A2 00 10`	short form for AL, 3 bytes
`mov [1000h], al`	`88 06 00 10`	general form, 4 bytes (same effect): ModR/M 06h = 00 000 110
`mov [1000h], bl`	`88 1E 00 10`	no short form for BL
`mov [bp], bl`	`88 5E 00`	5Eh = 01 011 110: [BP+d8], d8 = 0; segment SS

A decoder that wrongly took 1Eh as [BP] would read `88 1E` as a 2-byte `mov [bp], bl`, and the address bytes would become a new instruction: `00 10 = add [bx+si], dl`. Every later boundary shifts and memory gets corrupted.

Same Bytes, Different Instructions: Decoding from the Wrong Offset

from offset 0



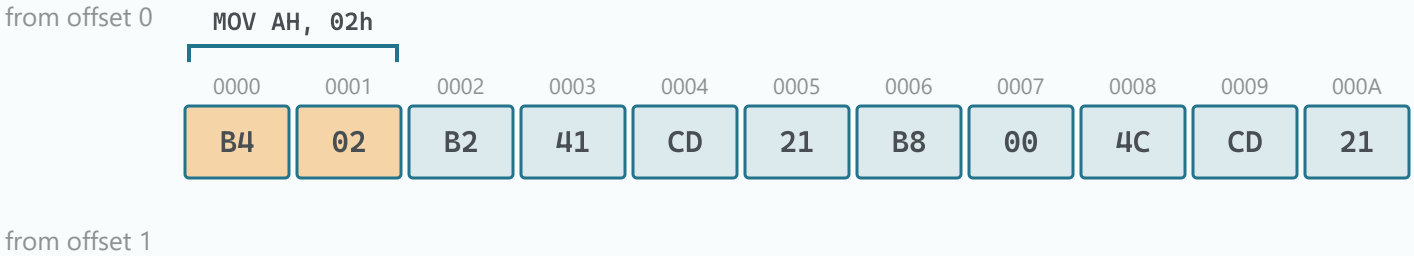
from offset 1

11 bytes in memory. Code or data? The CPU can't tell: whatever byte CS:IP points to, it takes as an **opcode**.

Step 1 / 9

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

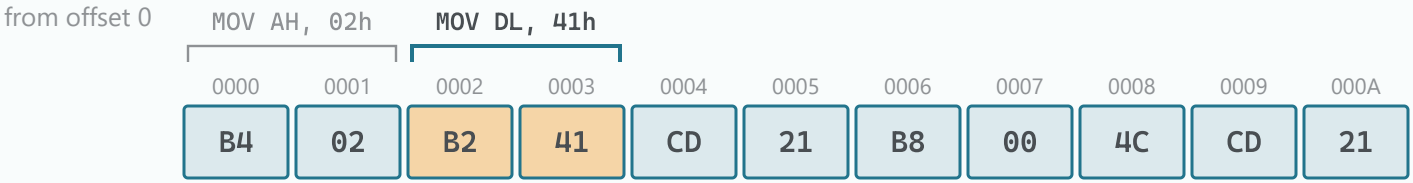
Same Bytes, Different Instructions: Decoding from the Wrong Offset



Offset 0: B4h = MOV AH, imm8 (B0h + 4) → takes 1 immediate byte: `mov ah, 02h`.

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

Same Bytes, Different Instructions: Decoding from the Wrong Offset

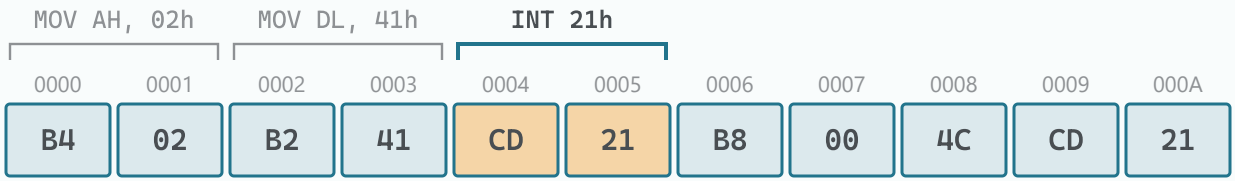


B2h = MOV DL, imm8 → `mov dl, 41h` (41h = 'A').

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

Same Bytes, Different Instructions: Decoding from the Wrong Offset

from offset 0

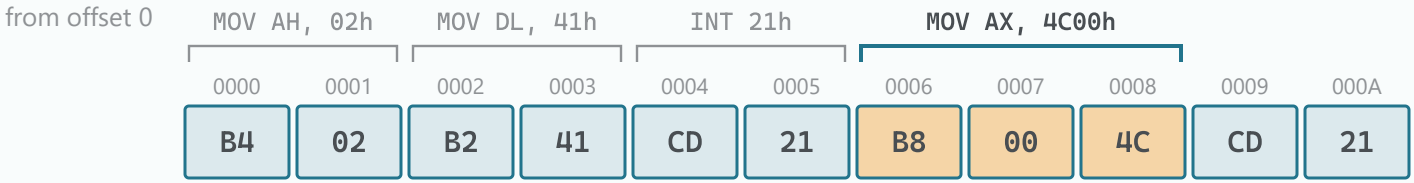


from offset 1

CDh = INT imm8 → `int 21h`: AH = 02h, DOS prints the 'A' in DL.

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

Same Bytes, Different Instructions: Decoding from the Wrong Offset

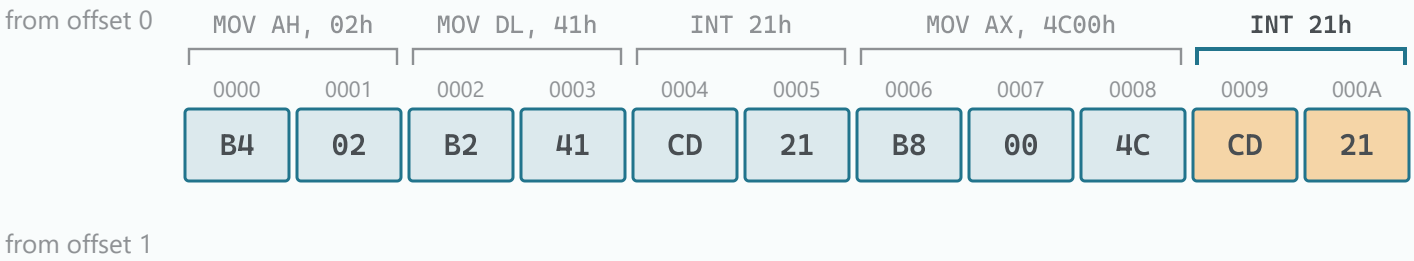


B8h = MOV AX, imm16 → 2 bytes, little-endian: `mov ax, 4C00h`.

Step 5 / 9

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

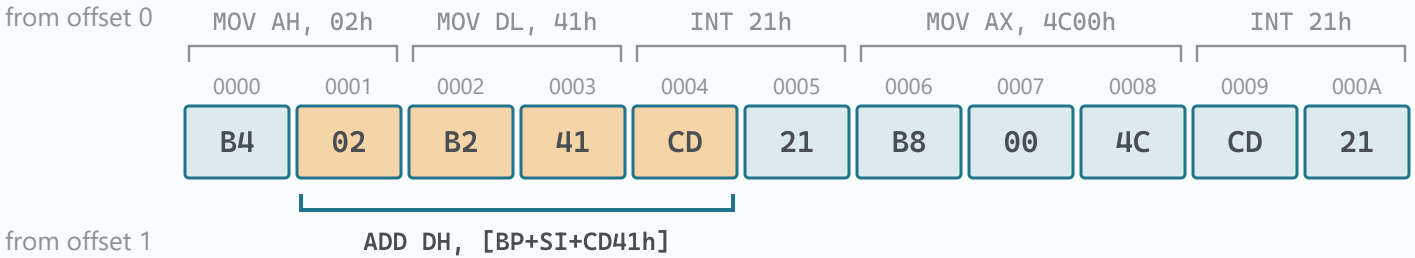
Same Bytes, Different Instructions: Decoding from the Wrong Offset



`int 21h`, AH = 4Ch: the program exits. Decoding from offset 0 gives the code the programmer wrote.

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

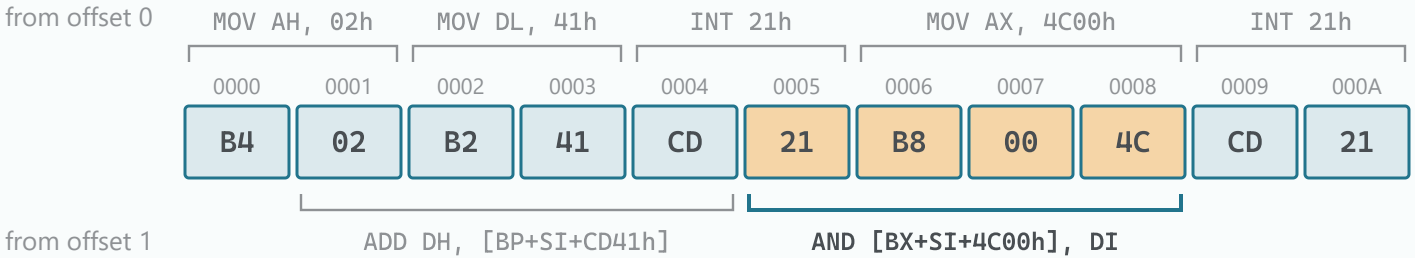
Same Bytes, Different Instructions: Decoding from the Wrong Offset



Now shift IP by one byte (offset 1). 02h = ADD r8, r/m8; ModR/M B2h = **10 110 010** → MOD = 10 (d16), REG = DH, R/M = [BP+SI]; d16 = CD41h → 4 bytes: the 'A' and INT's opcode are swallowed.

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

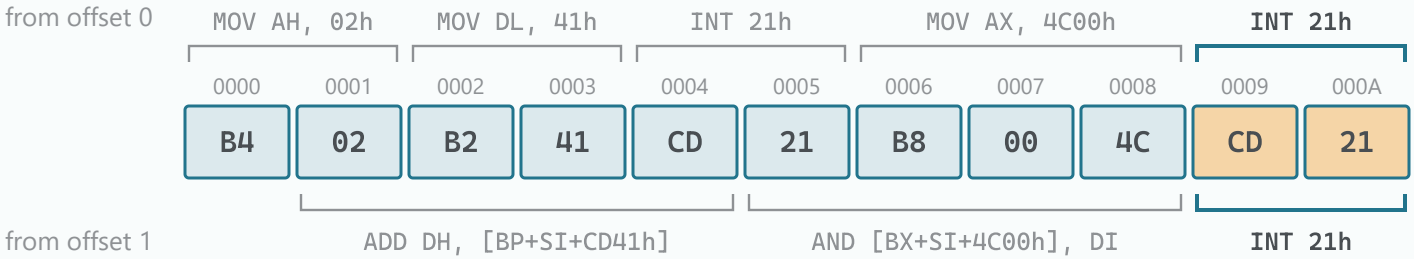
Same Bytes, Different Instructions: Decoding from the Wrong Offset



21h = AND r/m16, r16; ModR/M B8h = **10 111 000** → [BX+SI+d16], REG = DI; d16 = 4C00h. This instruction **writes to memory**.

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

Same Bytes, Different Instructions: Decoding from the Wrong Offset



The final CD 21 lands in the same place in both decodings: the decoding has **resynchronized**. But AX = 4C00h was never loaded, so INT 21h runs with whatever AH holds. The meaning comes not from the bytes but from where IP starts.

The 8086 has no hardware distinction between code and data: a JMP into data or a miscomputed IP executes the bytes as instructions. A disassembler, too, has to know where to start.

Data Transfer Instructions

Instruction	Operation	Example
`MOV d, s`	d ← s (copy)	`mov bx, [si]`
`PUSH s` / `POP d`	SP ← SP−2, [SS:SP] ← s / d ← [SS:SP], SP ← SP+2	`push ax`
`XCHG a, b`	a ↔ b (one must be a register)	`xchg al, [bx]`
`IN` / `OUT`	AL/AX ↔ I/O port	`in al, 60h`
`XLAT`	AL ← [DS:BX + AL]	table lookup
`LEA r, m`	r ← **address** of m (EA)	`lea si, msg`
`LDS` / `LES r, m`	r ← [m], DS/ES ← [m+2] (far pointer)	`les di, farptr`
`LAHF` / `SAHF`	AH ← low byte of FLAGS / the reverse	
`PUSHF` / `POPF`	push FLAGS / pop FLAGS	

Data transfer instructions **don't affect** the flags; the exceptions are SAHF and POPF, which write the flags directly.

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

BX gets the offset of the lookup table: **BX = 011Dh**.

Step 1 / 14

VARIABLES

AL = **00h** BX = **011Dh** CL = **00h** DL = **00h** AH = **00h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
▶  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

The byte to print: **AL = 3Ch** (0011 1100b).

Step 2 / 14

VARIABLES

AL = **3Ch** BX = **011Dh** CL = **00h** DL = **00h** AH = **00h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
▶  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

On the 8086 a shift count above 1 must be in **CL**.

Step 3 / 14

VARIABLES

AL = **3Ch** BX = **011Dh** CL = **04h** DL = **00h** AH = **00h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
▶  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

`shr al, cl` : 0011 1100 → 0000 0011. **AL = 03h** (high nibble).

Step 4 / 14

VARIABLES

AL = **03h** BX = **011Dh** CL = **04h** DL = **00h** AH = **00h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
▶ xlat                ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

`xlat` : AL ← [DS:BX + AL] = hex_tab[3] = **'3' (33h)**.

Step 5 / 14

VARIABLES

AL = **33h** BX = **011Dh** CL = **04h** DL = **00h** AH = **00h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
▶  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

The character is copied to DL (the INT 21h/02h argument).

Step 6 / 14

VARIABLES

AL = **33h** BX = **011Dh** CL = **04h** DL = **33h** AH = **00h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
▶  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

AH = 02h: the DOS **print character** service.

Step 7 / 14

VARIABLES

AL = **33h** BX = **011Dh** CL = **04h** DL = **33h** AH = **02h**

OUTPUT

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
▶  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

DOS prints **'3'** (AL also takes DL's value).

Step 8 / 14

VARIABLES

AL = **33h** BX = **011Dh** CL = **04h** DL = **33h** AH = **02h**

OUTPUT

```
3
```

Code Trace: Printing a Byte in Hex with XLAT

```
 1 org 100h
 2 mov bx, offset hex_tab
 3 mov al, num        ; AL = 3Ch
 4 mov cl, 4
 5 shr al, cl         ; high nibble
 6 xlat               ; AL = [BX+AL]
 7 mov dl, al
 8 mov ah, 2
 9 int 21h            ; print DL
▶mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

Reload the same byte: AL = 3Ch.

Step 9 / 14

VARIABLES

AL = **3Ch** BX = **011Dh** CL = **04h** DL = **33h** AH = **02h**

OUTPUT

```
3
```

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
▶ and al, 0Fh         ; low nibble
12 xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

`and al, 0Fh` : the high nibble is masked off → **AL = 0Ch**.

Step 10 / 14

VARIABLES

AL = **0Ch** BX = **011Dh** CL = **04h** DL = **33h** AH = **02h**

OUTPUT

```
3
```

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
▶ xlat
13 mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

hex_tab[12] = **'C' (43h)**.

Step 11 / 14

VARIABLES

AL = **43h** BX = **011Dh** CL = **04h** DL = **33h** AH = **02h**

OUTPUT

```
3
```

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
▶ mov dl, al
14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

DL = 43h. AH is still 02h, so no need to reload it.

Step 12 / 14

VARIABLES

AL = **43h** BX = **011Dh** CL = **04h** DL = **43h** AH = **02h**

OUTPUT

```
3
```

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num        ; AL = 3Ch
4  mov cl, 4
5  shr al, cl         ; high nibble
6  xlat               ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h            ; print DL
10 mov al, num
11 and al, 0Fh        ; low nibble
12 xlat
13 mov dl, al
▶ 14 int 21h
15 ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

'C' is printed → output **3C**.

Step 13 / 14

VARIABLES

AL = **43h** BX = **011Dh** CL = **04h** DL = **43h** AH = **02h**

OUTPUT

```
3C
```

Code Trace: Printing a Byte in Hex with XLAT

```
1  org 100h
2  mov bx, offset hex_tab
3  mov al, num       ; AL = 3Ch
4  mov cl, 4
5  shr al, cl        ; high nibble
6  xlat              ; AL = [BX+AL]
7  mov dl, al
8  mov ah, 2
9  int 21h           ; print DL
10 mov al, num
11 and al, 0Fh       ; low nibble
12 xlat
13 mov dl, al
14 int 21h
▶ ret
16 num  db 3Ch
17 hex_tab db '0123456789ABCDEF'
```

The program ends. XLAT translates through a table of up to 256 bytes in a single instruction.

Step 14 / 14

VARIABLES

AL = **43h** BX = **011Dh** CL = **04h** DL = **43h** AH = **02h**

OUTPUT

```
3C
```

Loading Addresses: LEA, LDS, LES

- `lea si, msg` ≡ `mov si, offset msg`; the difference: LEA also handles addresses computed **at run time**: `lea si, [bx+di+4]`.
- LEA doesn't touch memory or the flags, so it doubles as an adder: `lea ax, [bx+si]` → AX = BX + SI.
- `lds si, ptr`: SI ← [ptr], DS ← [ptr+2]. `les di, ptr` does the same with ES.
- A far pointer is stored in memory **offset first, segment second**.

```
mov bx, 0010h
mov di, 0004h
lea ax, [bx+di+2]  ; AX = 0016h
mov ax, [bx+di+2]  ; AX = [0016h]

; video memory: B800:0000
les di, screen     ; DI=0000h ES=B800h
mov al, 'A'
stosb              ; 'A' in the corner
lds si, screen     ; SI=0000h DS=B800h
lodsb              ; AL = 'A' (corner)
ret
screen dw 0000h, 0B800h
```

LEA gives the **address**, MOV the **contents**: the difference between `lea ax, [bx+di+2]` and `mov ax, [bx+di+2]` is an exam favourite.

Address or Contents? LEA, MOV and LES

```
1  org 100h
▶ lea bx, vals          ; BX ← address
3  mov di, 2
4  lea ax, [bx+di+2]    ; AX ← EA
5  mov ax, [bx+di+2]    ; AX ← [EA]
6  les di, screen        ; far pointer
7  mov al, 'A'
8  stosb                ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



LEA gives the address of the **vals** variable: BX = 0114h. Memory is not read.

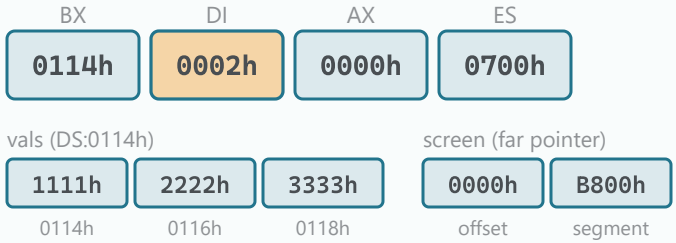
Step 1 / 8

VARIABLES

BX = **0114h** DI = **0000h** AX = **0000h** ES = **0700h**

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
▶  mov di, 2
4  lea ax, [bx+di+2]     ; AX ← EA
5  mov ax, [bx+di+2]     ; AX ← [EA]
6  les di, screen         ; far pointer
7  mov al, 'A'
8  stosb                 ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



DI = 2.

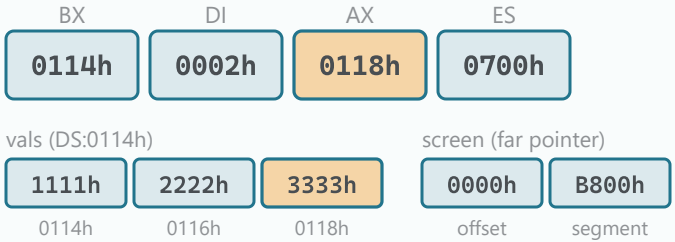
Step 2 / 8

VARIABLES

BX = 0114h | DI = 0002h | AX = 0000h | ES = 0700h

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
3  mov di, 2
▶  lea ax, [bx+di+2]     ; AX ← EA
5  mov ax, [bx+di+2]     ; AX ← [EA]
6  les di, screen         ; far pointer
7  mov al, 'A'
8  stosb                 ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



EA = BX + DI + 2 = 0118h (address)

LEA AX, [BX+DI+2]: only the effective address is computed → AX = 0114h + 2 + 2 = **0118h**. Memory is untouched, no flags change.

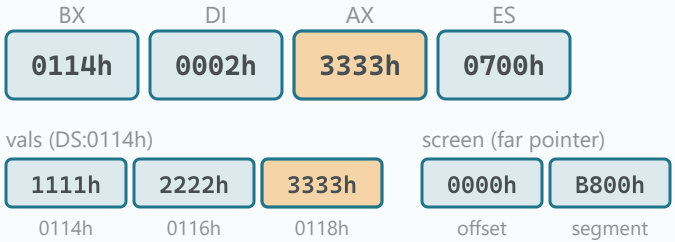
Step 3 / 8

VARIABLES

BX = **0114h** DI = **0002h** AX = **0118h** ES = **0700h**

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
3  mov di, 2
4  lea ax, [bx+di+2]     ; AX ← EA
▶  mov ax, [bx+di+2]     ; AX ← [EA]
6  les di, screen         ; far pointer
7  mov al, 'A'
8  stosb                 ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



[EA] = 3333h (contents)

MOV AX, [BX+DI+2]: the **contents** at the same EA are read → AX = **3333h** (the 3rd word of vals).

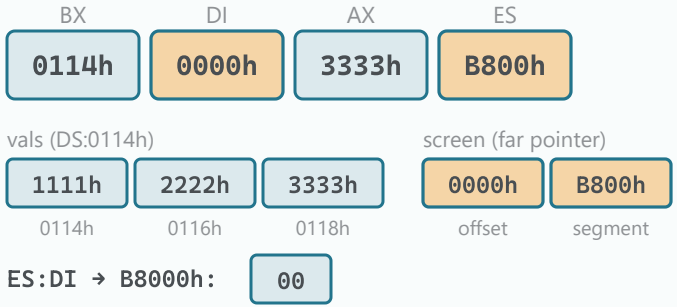
Step 4 / 8

VARIABLES

BX = 0114h DI = 0002h AX = 3333h ES = 0700h

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
3  mov di, 2
4  lea ax, [bx+di+2]     ; AX ← EA
5  mov ax, [bx+di+2]     ; AX ← [EA]
▶  les di, screen         ; far pointer
7  mov al, 'A'
8  stosb                 ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



LES DI, screen: the far pointer in memory (offset first, segment second) → DI = **0000h**, ES = **B800h**.

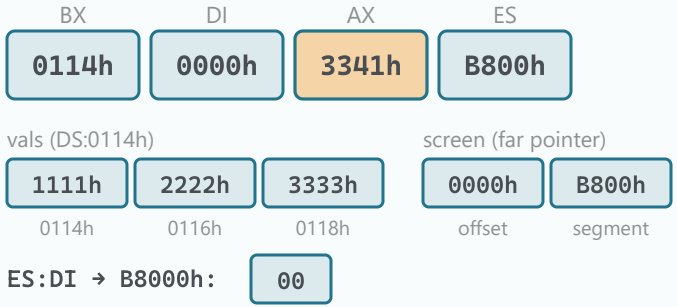
Step 5 / 8

VARIABLES

BX = **0114h** DI = **0000h** AX = **3333h** ES = **B800h**

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
3  mov di, 2
4  lea ax, [bx+di+2]     ; AX ← EA
5  mov ax, [bx+di+2]     ; AX ← [EA]
6  les di, screen         ; far pointer
▶  mov al, 'A'
8  stosb                 ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



AL = 41h ('A').

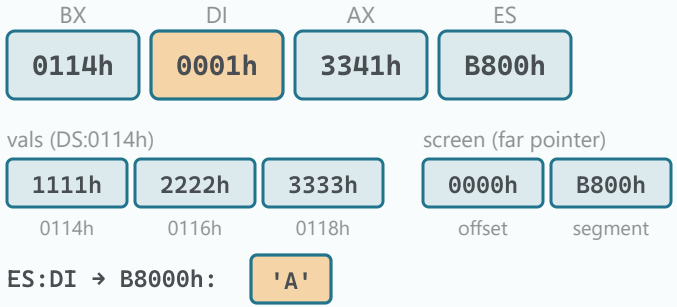
Step 6 / 8

VARIABLES

BX = 0114h | DI = 0000h | AX = 3341h | ES = B800h

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
3  mov di, 2
4  lea ax, [bx+di+2]     ; AX ← EA
5  mov ax, [bx+di+2]     ; AX ← [EA]
6  les di, screen         ; far pointer
7  mov al, 'A'
▶  stosb                 ; ES:[DI] ← AL
9  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



STOSB: ES:[DI] = B800h × 10h + 0000h = **B8000h** ← 41h; DF = 0, so DI = 1. An 'A' in the top-left corner of the screen.

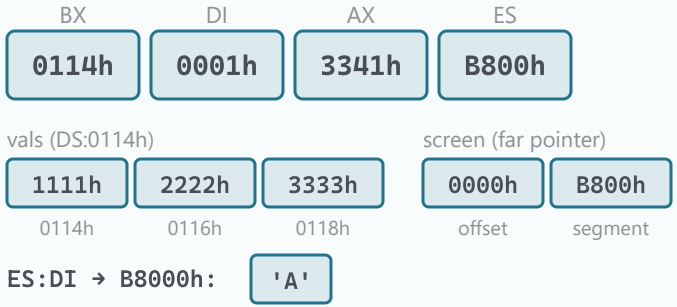
Step 7 / 8

VARIABLES

BX = 0114h | DI = 0001h | AX = 3341h | ES = B800h

Address or Contents? LEA, MOV and LES

```
1  org 100h
2  lea bx, vals          ; BX ← address
3  mov di, 2
4  lea ax, [bx+di+2]     ; AX ← EA
5  mov ax, [bx+di+2]     ; AX ← [EA]
6  les di, screen         ; far pointer
7  mov al, 'A'
8  stosb                 ; ES:[DI] ← AL
▶  ret
10 vals  dw 1111h, 2222h, 3333h
11 screen dw 0000h, 0B800h
```



RET: the program ends.

Step 8 / 8

VARIABLES

BX = 0114h DI = 0001h AX = 3341h ES = B800h

Arithmetic, Logic and Shift Instructions: A Summary

Group	Instructions	Description
Addition	`ADD ADC INC`	ADC adds the carry (CF) too; INC leaves CF alone
Subtraction	`SUB SBB DEC NEG CMP`	SBB subtracts the borrow (CF) too; NEG = 0 − d; CMP doesn't store the result
Multiplication	`MUL IMUL`	unsigned / signed; implicit AL or AX
Division	`DIV IDIV`	unsigned / signed; quotient + remainder
BCD adjust	`DAA DAS AAA AAS AAM AAD`	corrects a binary result to decimal
Sign extension	`CBW CWD`	AL → AX, AX → DX:AX
Logic	`AND OR XOR NOT TEST`	clear, set, invert bits; TEST doesn't store the result
Shift / rotate	`SHL SAL SHR SAR ROL ROR RCL RCR`	count is 1 or CL; the bit shifted out goes to CF

The CPU adds and subtracts with **one circuit** and doesn't know whether a number is signed or unsigned, so it sets flags for both readings: **CF** flags unsigned overflow, **OF** signed overflow.

STRING PROCESSING

String Instructions

Instruction	Operation	Updates	Flags
`MOVSB / MOVSW`	[ES:DI] ← [DS:SI]	SI, DI	–
`CMPSB / CMPSW`	[DS:SI] − [ES:DI]	SI, DI	like CMP
`SCASB / SCASW`	AL/AX − [ES:DI]	DI	like CMP
`LODSB / LODSW`	AL/AX ← [DS:SI]	SI	–
`STOSB / STOSW`	[ES:DI] ← AL/AX	DI	–

Source / destination

The source is **DS:SI** (a segment prefix can change it), the destination is always **ES:DI**.

Direction flag

`cld` → DF = 0, addresses **increase**; `std` → DF = 1, they **decrease**.

Step

±1 for the B (byte) form, ±2 for the W (word) form.

In EXE programs ES must be set up separately (`mov ax, @data` / `mov es, ax`); in COM programs DS = ES = CS.

Repeat Prefixes: REP, REPE, REPNE

Prefix	Repeat while	Used with
`REP`	CX ≠ 0	MOVS, STOS
`REPE` / `REPZ`	CX ≠ 0 and ZF = 1	CMPS, SCAS: "while equal"
`REPNE` / `REPNZ`	CX ≠ 0 and ZF = 0	CMPS, SCAS: "while not equal"

1. If CX = 0, stop (the instruction may never run).

Step 1 / 4

```
; copy a 100-byte block
cld
lea si, src
lea di, dst
mov cx, 100
rep movsb

; fill the screen with blanks
mov ax, 0B800h
mov es, ax
xor di, di
mov ax, 0720h    ; ' ' + attribute
mov cx, 2000     ; 80×25
rep stosw
```

Repeat Prefixes: REP, REPE, REPNE

Prefix	Repeat while	Used with
`REP`	CX ≠ 0	MOVS, STOS
`REPE` / `REPZ`	CX ≠ 0 and ZF = 1	CMPS, SCAS: "while equal"
`REPNE` / `REPNZ`	CX ≠ 0 and ZF = 0	CMPS, SCAS: "while not equal"

1. If CX = 0, stop (the instruction may never run).

2. Execute the string instruction once, update SI/DI.


```
; copy a 100-byte block
cld
lea si, src
lea di, dst
mov cx, 100
rep movsb

; fill the screen with blanks
mov ax, 0B800h
mov es, ax
xor di, di
mov ax, 0720h    ; ' ' + attribute
mov cx, 2000     ; 80×25
rep stosw
```

Repeat Prefixes: REP, REPE, REPNE

Prefix	Repeat while	Used with
`REP`	CX ≠ 0	MOVS, STOS
`REPE` / `REPZ`	CX ≠ 0 and ZF = 1	CMPS, SCAS: "while equal"
`REPNE` / `REPNZ`	CX ≠ 0 and ZF = 0	CMPS, SCAS: "while not equal"

1. If CX = 0, stop (the instruction may never run).

2. Execute the string instruction once, update SI/DI.

3. CX ← CX − 1 (flags unchanged).


```
; copy a 100-byte block
cld
lea si, src
lea di, dst
mov cx, 100
rep movsb

; fill the screen with blanks
mov ax, 0B800h
mov es, ax
xor di, di
mov ax, 0720h   ; ' ' + attribute
mov cx, 2000    ; 80×25
rep stosw
```

Repeat Prefixes: REP, REPE, REPNE

Prefix	Repeat while	Used with
`REP`	CX ≠ 0	MOVS, STOS
`REPE` / `REPZ`	CX ≠ 0 and ZF = 1	CMPS, SCAS: "while equal"
`REPNE` / `REPNZ`	CX ≠ 0 and ZF = 0	CMPS, SCAS: "while not equal"

1. If CX = 0, stop (the instruction may never run).

2. Execute the string instruction once, update SI/DI.

3. CX ← CX − 1 (flags unchanged).

4. For REPE/REPNE, test the ZF condition; stop if it fails, otherwise go back to 1.


```
; copy a 100-byte block
cld
lea si, src
lea di, dst
mov cx, 100
rep movsb

; fill the screen with blanks
mov ax, 0B800h
mov es, ax
xor di, di
mov ax, 0720h    ; ' ' + attribute
mov cx, 2000     ; 80×25
rep stosw
```

Code Trace: Copying with MOVSB

```
1  org 100h
▶  cld
3  lea si, src     ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb              ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp         ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

DF = 0: copy forward.

Step 1 / 13

VARIABLES

SI = **0000h** DI = **0000h** CX = **0000h**

destination = **'....'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
▶  lea si, src      ; DS:SI source
4  lea di, dst       ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb               ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp          ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

SI = 010Eh: source string.

Step 2 / 13

VARIABLES

SI = 010Eh | DI = 0000h | CX = 0000h | destination = '....'

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb               ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp          ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

DI = 0112h: destination area ('....').

Step 3 / 13

VARIABLES

SI = **010Eh** | DI = **0112h** | CX = **0000h**

destination = **'....'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
▶  mov cx, 4
6  copy_lp:
7  movsb              ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp         ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

Byte count CX = 4.

Step 4 / 13

VARIABLES

SI = **010Eh** DI = **0112h** CX = **0004h**

destination = **'....'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
▶  movsb            ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp     ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

`movsb` : one byte copied; SI = 010Fh, DI = 0113h. Flags unchanged.

Step 5 / 13

VARIABLES

SI = **010Fh** DI = **0113h** CX = **0004h**

destination = **'A...'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb             ; [DI]←[SI], SI++,
   DI++
▶  loop copy_lp      ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

CX = 0003h.

Step 6 / 13

VARIABLES

SI = **010Fh** DI = **0113h** CX = **0003h**

destination = **'A...'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
▶  movsb            ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp     ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

`movsb` : one byte copied; SI = 0110h, DI = 0114h. Flags unchanged.

Step 7 / 13

VARIABLES

SI = **0110h** DI = **0114h** CX = **0003h**

destination = **'AB..'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb             ; [DI]←[SI], SI++, DI++
▶  loop copy_lp      ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

CX = 0002h.

Step 8 / 13

VARIABLES

SI = **0110h** DI = **0114h** CX = **0002h**

destination = **'AB..'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
▶  movsb            ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp     ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

`movsb` : one byte copied; SI = 0111h, DI = 0115h. Flags unchanged.

Step 9 / 13

VARIABLES

SI = **0111h** DI = **0115h** CX = **0002h**

destination = **'ABC.'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb             ; [DI]←[SI], SI++,
   DI++
▶  loop copy_lp      ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

CX = 0001h.

Step 10 / 13

VARIABLES

SI = **0111h** DI = **0115h** CX = **0001h**

destination = **'ABC.'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
▶  movsb            ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp     ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

`movsb` : one byte copied; SI = 0112h, DI = 0116h. Flags unchanged.

Step 11 / 13

VARIABLES

SI = **0112h** DI = **0116h** CX = **0001h**

destination = **'ABCD'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb             ; [DI]←[SI], SI++, DI++
▶  loop copy_lp      ; = rep movsb
9  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

CX = 0000h.

Step 12 / 13

VARIABLES

SI = **0112h** DI = **0116h** CX = **0000h**

destination = **'ABCD'**

Code Trace: Copying with MOVSB

```
1  org 100h
2  cld
3  lea si, src      ; DS:SI source
4  lea di, dst      ; ES:DI destination
5  mov cx, 4
6  copy_lp:
7  movsb               ; [DI]←[SI], SI++,
   DI++
8  loop copy_lp        ; = rep movsb
▶  ret
10 src db 'ABCD'
11 dst  db 4 dup('.')
```

dst = **'ABCD'**. `mov cx, 4` + `rep movsb` does the same in one instruction.

Step 13 / 13

VARIABLES

`SI = 0112h` `DI = 0116h` `CX = 0000h`

`destination = 'ABCD'`

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
▶ cld                 ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`cld` : DF = 0 → string instructions **increment** SI/DI.

Step 1 / 25

VARIABLES

`AL = 00h` `SI = 0000h` `DI = 0000h` `CX = 0000h`

`msg = 'micro'`

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
▶  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

SI points to the source: SI = 0110h (msg).

Step 2 / 25

VARIABLES

AL = **00h** | SI = **0110h** | DI = **0000h** | CX = **0000h**

msg = **'micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
▶  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb                ; AL = [SI], SI++
8  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

The destination is the same string: DI = 0110h. In a COM program ES = DS, so ES:DI points to the same place.

Step 3 / 25

VARIABLES

AL = **00h** SI = **0110h** DI = **0110h** CX = **0000h**

msg = **'micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
▶  mov cx, 5
6  next_ch:
7  lodsb                ; AL = [SI], SI++
8  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

5 characters to process.

Step 4 / 25

VARIABLES

AL = **00h** SI = **0110h** DI = **0110h** CX = **0005h**

msg = **'micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
▶  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`lodsb` : AL = [DS:SI] = 6Dh, then SI = 0111h.

Step 5 / 25

VARIABLES

AL = **6Dh** SI = **0111h** DI = **0110h** CX = **0005h**

msg = **'micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb                ; AL = [SI], SI++
▶  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

Clear bit 5: AL = 4Dh.

Step 6 / 25

VARIABLES

AL = 4Dh | SI = 0111h | DI = 0110h | CX = 0005h | msg = 'micro'

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
▶  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`stosb` : [ES:DI] ← AL, then DI = 0111h.

Step 7 / 25

VARIABLES

AL = **4Dh** SI = **0111h** DI = **0111h** CX = **0005h**

msg = **'Micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
▶  loop next_ch
11 ret
12 msg db 'micro'
```

CX = 0004h.

Step 8 / 25

VARIABLES

AL = **4Dh** SI = **0111h** DI = **0111h** CX = **0004h**

msg = **'Micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
▶  lodsb                ; AL = [SI], SI++
8  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`lodsb` : AL = [DS:SI] = 69h, then SI = 0112h.

Step 9 / 25

VARIABLES

AL = **69h** SI = **0112h** DI = **0111h** CX = **0004h**

msg = **'Micro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb                ; AL = [SI], SI++
▶  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

Clear bit 5: AL = 49h.

Step 10 / 25

VARIABLES

AL = 49h | SI = 0112h | DI = 0111h | CX = 0004h | msg = 'Micro'

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
▶  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`stosb` : [ES:DI] ← AL, then DI = 0112h.

Step 11 / 25

VARIABLES

AL = **49h** SI = **0112h** DI = **0112h** CX = **0004h**

msg = **'MIcro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
▶  loop next_ch
11 ret
12 msg db 'micro'
```

CX = 0003h.

Step 12 / 25

VARIABLES

AL = **49h** SI = **0112h** DI = **0112h** CX = **0003h**

msg = **'MIcro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
▶  lodsb                ; AL = [SI], SI++
8  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`lodsb` : AL = [DS:SI] = 63h, then SI = 0113h.

Step 13 / 25

VARIABLES

AL = **63h** SI = **0113h** DI = **0112h** CX = **0003h**

msg = **'MIcro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb                ; AL = [SI], SI++
▶  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

Clear bit 5: AL = 43h.

Step 14 / 25

VARIABLES

AL = 43h | SI = 0113h | DI = 0112h | CX = 0003h | msg = 'MIcro'

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
▶  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`stosb` : [ES:DI] ← AL, then DI = 0113h.

Step 15 / 25

VARIABLES

AL = **43h** SI = **0113h** DI = **0113h** CX = **0003h**

msg = **'MICro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
▶  loop next_ch
11 ret
12 msg db 'micro'
```

CX = 0002h.

Step 16 / 25

VARIABLES

AL = **43h** SI = **0113h** DI = **0113h** CX = **0002h**

msg = **'MICro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
▶  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`lodsb` : AL = [DS:SI] = 72h, then SI = 0114h.

Step 17 / 25

VARIABLES

AL = **72h** SI = **0114h** DI = **0113h** CX = **0002h**

msg = **'MICro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
▶  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

Clear bit 5: AL = 52h.

Step 18 / 25

VARIABLES

AL = **52h** SI = **0114h** DI = **0113h** CX = **0002h**

msg = **'MICro'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
▶  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`stosb` : [ES:DI] ← AL, then DI = 0114h.

Step 19 / 25

VARIABLES

AL = **52h** SI = **0114h** DI = **0114h** CX = **0002h**

msg = **'MICRo'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
▶ loop next_ch
11 ret
12 msg db 'micro'
```

CX = 0001h.

Step 20 / 25

VARIABLES

AL = **52h** SI = **0114h** DI = **0114h** CX = **0001h**

msg = **'MICRo'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
▶  lodsb                ; AL = [SI], SI++
8  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`lodsb` : AL = [DS:SI] = 6Fh, then SI = 0115h.

Step 21 / 25

VARIABLES

AL = **6Fh** SI = **0115h** DI = **0114h** CX = **0001h**

msg = **'MICRo'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
▶  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

Clear bit 5: AL = 4Fh.

Step 22 / 25

VARIABLES

AL = **4Fh** | SI = **0115h** | DI = **0114h** | CX = **0001h**

msg = **'MICRo'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
▶  stosb              ; [DI] = AL, DI++
10 loop next_ch
11 ret
12 msg db 'micro'
```

`stosb` : [ES:DI] ← AL, then DI = 0115h.

Step 23 / 25

VARIABLES

AL = **4Fh** SI = **0115h** DI = **0115h** CX = **0001h**

msg = **'MICRO'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                ; DF = 0: forward
3  lea si, msg
4  mov di, si         ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb              ; AL = [SI], SI++
8  and al, 0DFh       ; upper case
9  stosb              ; [DI] = AL, DI++
▶  loop next_ch
11 ret
12 msg db 'micro'
```

CX = 0000h.

Step 24 / 25

VARIABLES

AL = **4Fh** SI = **0115h** DI = **0115h** CX = **0000h**

msg = **'MICRO'**

Code Trace: Converting to Upper Case with LODSB + STOSB

```
1  org 100h
2  cld                  ; DF = 0: forward
3  lea si, msg
4  mov di, si           ; write in place
5  mov cx, 5
6  next_ch:
7  lodsb                ; AL = [SI], SI++
8  and al, 0DFh         ; upper case
9  stosb                ; [DI] = AL, DI++
10 loop next_ch
▶  ret
12 msg db 'micro'
```

msg = **'MICRO'**. A LODS/STOS pair is the standard pattern for transforming an array in place.

Step 25 / 25

VARIABLES

`AL = 4Fh` `SI = 0115h` `DI = 0115h` `CX = 0000h` `msg = 'MICRO'`

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
▶ cld
3  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
7  jne not_found
8  dec di               ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

Scan forward.

Step 1 / 9

VARIABLES

`AL = 00h` `DI = 0000h` `CX = 0000h` `ZF = 0`

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
▶  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
7  jne not_found
8  dec di               ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

DI = 0113h: the string to scan must be at ES:DI.

Step 2 / 9

VARIABLES

AL = 00h | DI = 0113h | CX = 0000h | ZF = 0

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
▶  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
7  jne not_found
8  dec di               ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

At most 7 comparisons.

Step 3 / 9

VARIABLES

AL = **00h** | DI = **0113h** | CX = **0007h** | ZF = **0**

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
4  mov cx, 7          ; length
▶  mov al, 'h'        ; target
6  repne scasb        ; search
7  jne not_found
8  dec di             ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

Character to find: AL = 'h' (68h).

Step 4 / 9

VARIABLES

AL = **68h** DI = **0113h** CX = **0007h** ZF = **0**

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
▶  repne scasb          ; search
7  jne not_found
8  dec di               ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

`repne scasb` : compares against 'p','u','s','h'; on the 4th comparison **ZF = 1** and it stops. Each pass does DI++ and CX−−: **DI = 0117h, CX = 0003h**.

Step 5 / 9

VARIABLES

AL = **68h** DI = **0117h** CX = **0003h** ZF = **1**

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
▶  jne not_found
8  dec di               ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

ZF = 1 (found) → no jump.

Step 6 / 9

VARIABLES

`AL = 68h` `DI = 0117h` `CX = 0003h` `ZF = 1`

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
7  jne not_found
▶  dec di               ; match position
9  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

DI points **one past** the match; step back: DI = 0116h.

Step 7 / 9

VARIABLES

AL = 68h | DI = 0116h | CX = 0003h | ZF = 0

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
7  jne not_found
8  dec di               ; match position
▶  sub di, offset msg
10 not_found:
11 ret
12 msg db 'pushing'
```

Subtract the start: **DI = 3** → 'h' is at index 3 (counting from 0).

Step 8 / 9

VARIABLES

AL = **68h** DI = **0003h** CX = **0003h** ZF = **0**

Code Trace: Finding a Character with REPNE SCASB

```
1  org 100h
2  cld
3  lea di, msg
4  mov cx, 7            ; length
5  mov al, 'h'          ; target
6  repne scasb          ; search
7  jne not_found
8  dec di               ; match position
9  sub di, offset msg
10 not_found:
▶ ret
12 msg db 'pushing'
```

Had it not been found, the scan would end with CX = 0 and ZF = 0, and `jne not_found` would jump.

Step 9 / 9

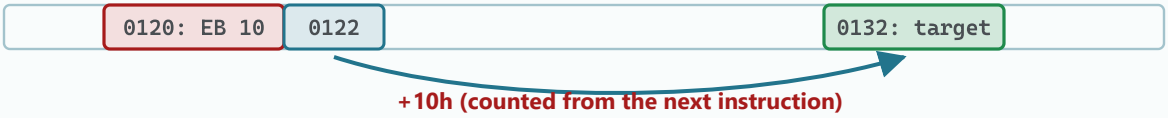
VARIABLES

AL = **68h** DI = **0003h** CX = **0003h** ZF = **0**

Unconditional Jump: JMP

Kind	Code	Range	Example
Short	`EB d8`	−128 … +127 bytes	`jmp short again`
Near	`E9 d16`	same segment, ±32 K	`jmp start`
Far	`EA IP CS`	any segment	`jmp far ptr dst`
Indirect near	`FF /4`	IP ← register/memory	`jmp bx` , `jmp jmp_tab[bx]`
Indirect far	`FF /5`	CS:IP ← memory (4 bytes)	`jmp dword ptr [si]`

- Short and near jumps are **relative**: target = address of the next instruction + displacement, so the code can be moved in memory.
- Indirect jumps build a **jump table** (switch-case): `shl bx, 1` / `jmp hex_tab[bx]` .
- `EB FE` jumps to itself (an endless loop): FEh = −2 and the instruction is 2 bytes.



Processor Control Instructions

Instruction	Effect
`CLC` / `STC`	CF ← 0 / CF ← 1
`CMC`	CF ← ¬CF
`CLD` / `STD`	DF ← 0 / 1 (string direction)
`CLI` / `STI`	IF ← 0 / 1 (INTR interrupts)

Instruction	Effect
`HLT`	halt until an interrupt or RESET
`WAIT`	wait until the TEST̄ pin goes active (8087 sync)
`ESC`	pass the instruction to the coprocessor (8087)
`LOCK`	prefix: LOCK̄ pin active for the next instruction → bus locked
`NOP`	do nothing (90h), 3 clocks

CLC/STC

A common way to return success/failure from a procedure: `stc` on error, and the caller does `jc fail`.

CLI/STI

Protect a critical section (e.g. changing SS:SP) from interrupts. NMI can't be masked.

LOCK

In a multiprocessor system `lock xchg` builds an atomic semaphore.

Flag Effects: Arithmetic and Logic Instructions

Instruction	OF	SF	ZF	AF	PF	CF
`ADD ADC SUB SBB CMP NEG`	*	*	*	*	*	*
`CMPS SCAS`	*	*	*	*	*	*
`INC DEC`	*	*	*	*	*	–
`MUL IMUL`	*	?	?	?	?	*
`DIV IDIV`	?	?	?	?	?	?
`DAA DAS`	?	*	*	*	*	*
`AAA AAS`	?	?	?	*	?	*
`AAM AAD`	?	*	*	?	*	?
`AND OR XOR TEST`	0	*	*	?	*	0
`SHL SAL SHR SAR`	*[1]	*	*	?	*	*
`ROL ROR RCL RCR`	*[1]	–	–	–	–	*

***** set by the result · **0** cleared · **–** unchanged · **?** undefined (don't rely on it) · [1] OF is defined only for a shift count of 1. DF, IF, TF are not affected by these instructions.

Flag Effects: Other Instructions

Instructions	Flags affected
`MOV XCHG LEA LDS LES XLAT PUSH POP IN OUT LAHF PUSHF CBW CWD NOT`	**none**
`MOVS LODS STOS`, `REP`	none
`JMP Jcc LOOP LOOPE LOOPNE JCXZ CALL RET`	none (only read)
`SAHF`	SF ZF AF PF CF ← AH
`POPF`, `IRET`	all (from the stack)
`INT n`, `INTO`	IF ← 0, TF ← 0 (old FLAGS on the stack)
`CLC STC CMC`	CF
`CLD STD`	DF
`CLI STI`	IF

A classic mistake: putting a flag-changing instruction (e.g. `inc`, `add`) between `cmp` and `jcc`. Only flag-neutral instructions such as MOV, PUSH, LEA may go in between.

Lab: The Instruction Set

Printing hex with XLAT

CODE · **45 bytes**

```
 1 org 100h
 2 mov bx, offset hex_tab
 3 mov al, num          ; AL = 3…
 4 mov cl, 4
 5 shr al, cl           ; high …
 6 xlat                 ; AL = …
 7 mov dl, al
 8 mov ah, 2
 9 int 21h              ; print…
10 mov al, num
11 and al, 0Fh          ; low n…
12 xlat
13 mov dl, al
14 int 21h
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	002D	CH	00	CL	2D
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**

```
0118 D0 CD 21 C3 3C 30 31 32
0120 33 34 35 36 37 38 39 41
0128 42 43 44 45 46 00 00 00
0130 00 00 00 00 00 00 00 00
0138 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: 45 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (002Dh). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

Encoding

An instruction is 1–6 bytes: opcode (D, W) + ModR/M + displacement + immediate; prefixes go in front.

ModR/M

MOD = 11 register, 00/01/10 memory; MOD = 00, R/M = 110 is a direct address, so [BP] is encoded with d8 = 0.

Data transfer

XLAT: AL ← [DS:BX + AL]. LEA gives the address, MOV the contents; LDS/LES load a far pointer. No flags change.

Strings

DS:SI → ES:DI, DF sets the direction, REP/REPE/REPNE repeat up to CX times; with CX = 0 the instruction never runs.

JMP

Short (EB d8) and near (E9 d16) jumps are relative; far and indirect jumps use absolute addresses.

Flag effects

Moves, string moves and jumps leave the flags alone; never put a flag-changing instruction between CMP and Jcc.

SUMMARY

Exercises

1. Encode `mov dx, [bp+di+8]` (opcode 100010, d = 1, w = 1).
2. DS = 1000h, BX = 0200h, AL = 05h: from which physical address does `xlat` load AL?
3. `farptr dd` holds the bytes 34 12 00 B8. What are ES and DI after `les di, farptr`?
4. How many bytes does `rep movsb` copy when CX = 0?
5. CLD, CX = 5, SI = 0100h, DI = 0200h; `repe cmpsb` finds a mismatch at byte 3. CX, SI, DI and ZF afterwards?
6. A `jmp short` at 0150h jumps to 0130h. What is its machine code?
7. Count the spaces in a string with `repne scasb`.

▶ **Hints / answers**

SUMMARY

Exercises (2)

8. Which instruction is `8A 1E 34 12`? What do the same bytes decode to one byte later (from offset 1)?
9. Encode `mov es, dx` and `mov al, [0020h]`.

▶ **Hints / answers**

References

- Intel, *The 8086 Family User's Manual*, 1979 (Chapter 2: the instruction set, Chapter 4: instruction encoding).
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009.
- Kip R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014.
- Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009 (in Turkish).
- emu8086 help: *8086 instructions* and *8086 assembler tutorial*.

The End

Chapter 11: Instruction Encoding and String Instructions

Previous: Chapter 10.2: Device Control