Chapter 4.1: Memory Access

Microprocessors

Addressing modes | Displacement | Effective address | Segment:offset | byte ptr / word ptr | MOV rules | Video memory example | XCHG | 8/16-bit copying | Number bases

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_04_01_Memory_Access.html

Learning Outcomes

Addressing modes
Build valid memory operands from BX, SI, DI, BP and a displacement.

Address arithmetic
Compute the effective address, and the physical address from segment:offset.

Data size
State the access size with `byte ptr` / `word ptr`; read little-endian data.

MOV and XCHG
Tell valid operand combinations apart and trace copy and swap programs.

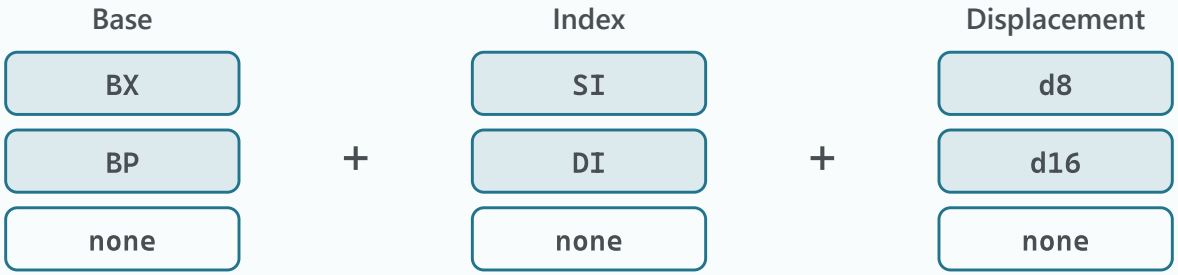
Memory Access

- Memory is addressed through the **BX, SI, DI and BP** registers.
- Combining these registers (and a constant) inside `[ ]` reaches different memory locations; these combinations are the **addressing modes**.

Register(s) only	+ d8 (8-bit displacement)	+ d16 (16-bit displacement)
`[BX + SI]`	`[BX + SI + d8]`	`[BX + SI + d16]`
`[BX + DI]`	`[BX + DI + d8]`	`[BX + DI + d16]`
`[BP + SI]`	`[BP + SI + d8]`	`[BP + SI + d16]`
`[BP + DI]`	`[BP + DI + d8]`	`[BP + DI + d16]`
`[SI]`	`[SI + d8]`	`[SI + d16]`
`[DI]`	`[DI + d8]`	`[DI + d16]`
`d16 (direct address)`	`[BP + d8]`	`[BP + d16]`
`[BX]`	`[BX + d8]`	`[BX + d16]`

These 24 combinations are all the memory choices of the mod/rm field in 8086 machine code. A bare `[BP]` is encoded by the assembler as `[BP + 0]` (d8 = 0).

The Combination Rule



at most one per column · BP chosen → segment SS, else DS

Rules

- **BX and BP** never appear in the same mode.
- **SI and DI** never appear in the same mode.
- Modes with BP use the SS segment; modes without BP use DS.

Examples

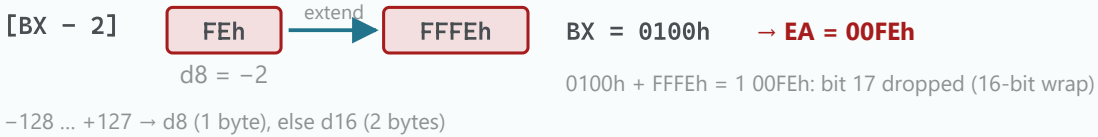
- `[BX+5]` → base + d8 (DS)
- `[BX+SI]` → base + index (DS)
- `[DI+BX-4]` → `[BX+DI-4]`, order does not matter (DS)
- `[BP+SI+300]` → base + index + d16 (SS)

Displacement

d8
8-bit signed displacement (−128 … +127).
Examples: `22`, `55h`, `-1`.

d16
16-bit signed displacement. Examples: `300`, `5517h`, `-259`.

- A displacement can be a number or the **offset of a variable**: `[BX + arr]`.
- The assembler folds several constants into **one**: `[BX + 2 + 3]` → `[BX + 5]`.
- It may be written inside or outside `[ ]`; the machine code is the same: `[BX+5]` = `5[BX]` = `[BX]+5`.
- It is signed, so it can be positive or negative: `[SI-1]`.
- The assembler picks the short form (d8) when the value is in −128…127, otherwise the long form (d16).



The effective address is computed in 16 bits and **wraps around**: with BX = FFFFh, `[BX+2]` → EA = 0001h.

Valid or Invalid?

Valid	Mode
`[BX]`	register indirect
`[1234h]`	direct address
`[SI+10]`	indexed
`[BP+DI-2]`	based-indexed (SS)
`ES:[BX+SI]`	segment override

Invalid	Why?
`[AX]`, `[CX]`, `[DX]`	cannot address memory
`[BL]`, `[BH]`	an 8-bit register is not an address
`[BX+BP]`	two base registers
`[SI+DI]`	two index registers
`[SP]`, `[BX*2]`	not on the 8086 (80386+)

The table generates every valid combination: base (BX/BP) + index (SI/DI) + displacement, each optional.

Kinds of Addressing Modes

Mode	Example	Where is the operand?
Immediate	`MOV AX, 5`	in the instruction (constant)
Register	`MOV AX, BX`	in a register
Direct	`MOV AX, [1000h]`	DS:1000h
Register indirect	`MOV AX, [BX]`	DS:BX
Based	`MOV AX, [BP+4]`	SS:BP+4
Indexed	`MOV AL, [SI+3]`	DS:SI+3
Based-indexed	`MOV AL, [BX+DI]`	DS:BX+DI
Based-indexed + displacement	`MOV AL, [BX+SI+8]`	DS:BX+SI+8

The first two never touch memory; the rest compute an effective address (EA). Physical address = segment × 10h + EA.

String, I/O and Relative Modes

Three more modes: the operand is not written at all, lives in an address space separate from memory, or is found relative to IP.

String mode

String instructions have **implicit** operands: source `DS:SI`, destination `ES:DI`. After each operation SI and DI increase when DF = 0 and decrease when DF = 1 (byte ±1, word ±2).

```
MOVSB ; ES:[DI] ← DS:[SI]
```

I/O mode

Ports live in a separate 64 K address space. Data passes only through **AL** or **AX**; a port 0–255 is given in the instruction, a higher one in **DX**.

```
IN  AL, 45h
OUT DX, AX
```

Relative mode

The instruction holds not the target but a **signed distance** added to IP: IP = address of next instruction + distance. Short jumps use an 8-bit distance (−128 … +127).

```
JNZ again ; 75 FA = −6
```

Mnemonics are written as one word (`MOVSB` , not `MOVS B`). OUT takes the port first, then the data register: `OUT 50h, AL` ; `IN A, 45` is 8085 syntax.

Code Trace: Six Addressing Modes

```
1  ORG 100h
▶  MOV BX, OFFSET table
3  MOV SI, 2
4  MOV AH, 7            ; immediate
5  MOV AL, [table+1]    ; direct
6  MOV CL, [BX]         ; indirect
7  MOV CH, [BX+4]       ; based
8  MOV DL, [BX+SI]      ; base+index
9  MOV DH, [BX+SI+3]    ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```

table (DS:0116h ...)

11h	22h	33h	44h	55h	66h
0116	0117	0118	0119	011A	011B

BX = 0116h SI = 0000h

AH	AL	CL
00h	00h	00h
CH	**DL**	**DH**
00h	00h	00h

BX = **0116h**: the offset of `table` (OFFSET is computed at assembly time).

Step 1 / 8

VARIABLES

`BX = 0116h` `SI = 0000h` `AX = 0000h` `CX = 0000h` `DX = 0000h`

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
▶  MOV SI, 2
4  MOV AH, 7           ; immediate
5  MOV AL, [table+1]   ; direct
6  MOV CL, [BX]        ; indirect
7  MOV CH, [BX+4]      ; based
8  MOV DL, [BX+SI]     ; base+index
9  MOV DH, [BX+SI+3]   ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```

table (DS:0116h ...)

11h	22h	33h	44h	55h	66h
0116	0117	0118	0119	011A	011B

BX = 0116h SI = 0002h

AH	AL	CL
00h	00h	00h
CH	**DL**	**DH**
00h	00h	00h

SI = 2: the index register.

Step 2 / 8

VARIABLES

BX = 0116h SI = 0002h AX = 0000h CX = 0000h DX = 0000h

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
3  MOV SI, 2
▶  MOV AH, 7            ; immediate
5  MOV AL, [table+1]    ; direct
6  MOV CL, [BX]         ; indirect
7  MOV CH, [BX+4]       ; based
8  MOV DL, [BX+SI]      ; base+index
9  MOV DH, [BX+SI+3]    ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```

table (DS:0116h ...)

11h	22h	33h	44h	55h	66h
0116	0117	0118	0119	011A	011B

immediate

`07h is in the instruction`

AH	AL	CL
07h	00h	00h
CH	**DL**	**DH**
00h	00h	00h

Immediate mode: 7 is inside the instruction; no memory access. AH = 07h.

Step 3 / 8

VARIABLES

BX = 0116h SI = 0002h AX = 0700h CX = 0000h DX = 0000h

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
3  MOV SI, 2
4  MOV AH, 7            ; immediate
▶  MOV AL, [table+1]    ; direct
6  MOV CL, [BX]         ; indirect
7  MOV CH, [BX+4]       ; based
8  MOV DL, [BX+SI]      ; base+index
9  MOV DH, [BX+SI+3]    ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```

table (DS:0116h ...)

11h	22h	33h	44h	55h	66h
0116	0117	0118	0119	011A	011B

direct

EA = table+1 = 0117h

physical = 07000h + 0117h = 07117h

AH	AL	CL
07h	22h	00h
CH	**DL**	**DH**
00h	00h	00h

Direct mode: the address is in the instruction: [0117h] → AL = **22h**.

Step 4 / 8

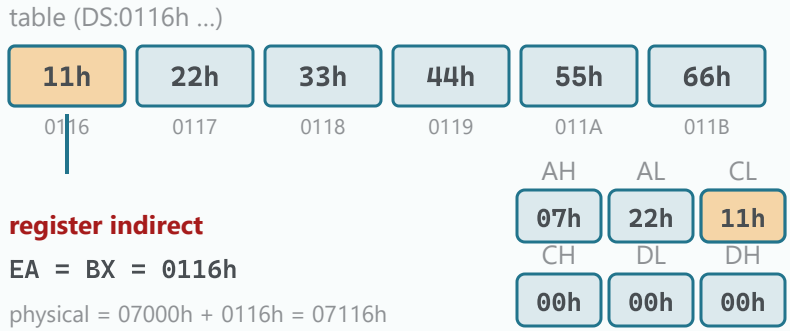
VARIABLES

BX = 0116h SI = 0002h AX = 0722h CX = 0000h DX = 0000h

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
3  MOV SI, 2
4  MOV AH, 7           ; immediate
5  MOV AL, [table+1]   ; direct
▶  MOV CL, [BX]        ; indirect
7  MOV CH, [BX+4]      ; based
8  MOV DL, [BX+SI]     ; base+index
9  MOV DH, [BX+SI+3]   ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```



Register indirect: EA = BX = 0116h → CL = **11h**.

Step 5 / 8

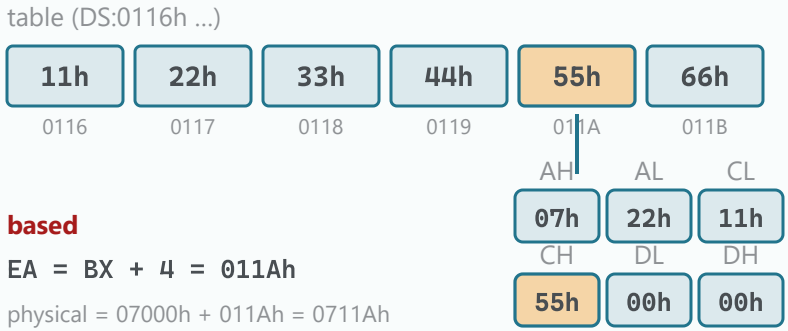
VARIABLES

BX = **0116h** SI = **0002h** AX = **0722h** CX = **0011h** DX = **0000h**

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
3  MOV SI, 2
4  MOV AH, 7           ; immediate
5  MOV AL, [table+1]   ; direct
6  MOV CL, [BX]        ; indirect
▶  MOV CH, [BX+4]      ; based
8  MOV DL, [BX+SI]     ; base+index
9  MOV DH, [BX+SI+3]   ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```



based

`EA = BX + 4 = 011Ah`

physical = 07000h + 011Ah = 0711Ah

Based: EA = BX + 4 = 011Ah → CH = **55h**.

Step 6 / 8

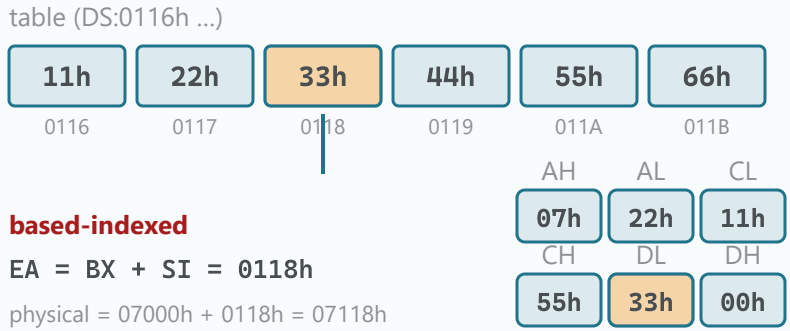
VARIABLES

`BX = 0116h` `SI = 0002h` `AX = 0722h` `CX = 5511h` `DX = 0000h`

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
3  MOV SI, 2
4  MOV AH, 7           ; immediate
5  MOV AL, [table+1]   ; direct
6  MOV CL, [BX]        ; indirect
7  MOV CH, [BX+4]      ; based
▶  MOV DL, [BX+SI]     ; base+index
9  MOV DH, [BX+SI+3]   ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```



Based-indexed: EA = BX + SI = 0118h → DL = **33h**.

Step 7 / 8

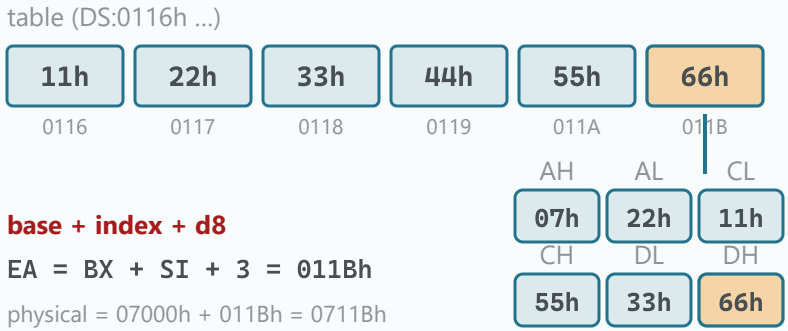
VARIABLES

BX = **0116h** SI = **0002h** AX = **0722h** CX = **5511h** DX = **0033h**

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Code Trace: Six Addressing Modes

```
1  ORG 100h
2  MOV BX, OFFSET table
3  MOV SI, 2
4  MOV AH, 7           ; immediate
5  MOV AL, [table+1]   ; direct
6  MOV CL, [BX]        ; indirect
7  MOV CH, [BX+4]      ; based
8  MOV DL, [BX+SI]     ; base+index
▶  MOV DH, [BX+SI+3]   ; + d8
10 RET
11 table DB 11h,22h,33h,44h,55h,66h
```



base + index + d8

`EA = BX + SI + 3 = 011Bh`

physical = 07000h + 011Bh = 0711Bh

Base + index + d8: EA = BX + SI + 3 = 011Bh → DH = **66h**. No BP is used, so the segment is always DS.

Step 8 / 8

VARIABLES

BX = **0116h** SI = **0002h** AX = **0722h** CX = **5511h** DX = **6633h**

DS = 0700h is assumed for the .COM program; physical address = DS × 10h + EA.

Effective Address Step by Step

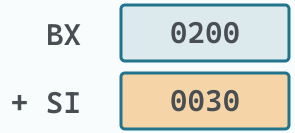
`MOV AL, [BX + SI + 4]` · DS = 1000h, BX = 0200h, SI = 0030h



Base register BX = 0200h.

Effective Address Step by Step

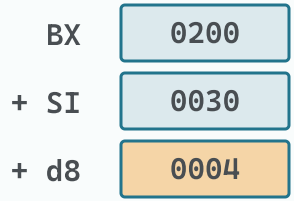
`MOV AL, [BX + SI + 4]` · DS = 1000h, BX = 0200h, SI = 0030h



Add the index register SI = 0030h.

Effective Address Step by Step

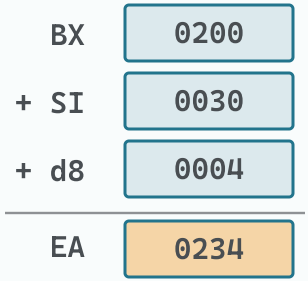
`MOV AL, [BX + SI + 4]` · DS = 1000h, BX = 0200h, SI = 0030h



8-bit displacement: +4.

Effective Address Step by Step

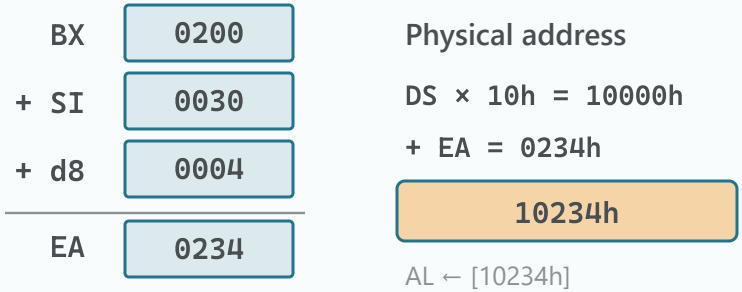
`MOV AL, [BX + SI + 4]` · DS = 1000h, BX = 0200h, SI = 0030h



Effective address EA = 0200h + 0030h + 0004h = **0234h** (16 bits).

Effective Address Step by Step

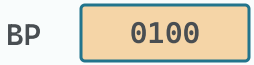
`MOV AL, [BX + SI + 4]` · DS = 1000h, BX = 0200h, SI = 0030h



No BP → segment **DS**. Physical address = 10000h + 0234h = **10234h**. AL gets the byte at that address.

Access Through BP

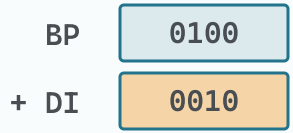
`MOV AX, [BP + DI - 2]` · SS = 2000h, BP = 0100h, DI = 0010h



Base register BP = 0100h.

Access Through BP

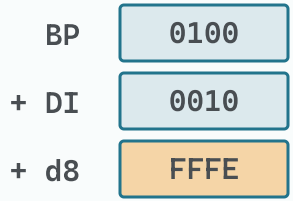
`MOV AX, [BP + DI − 2]` · SS = 2000h, BP = 0100h, DI = 0010h



Add the index register DI = 0010h.

Access Through BP

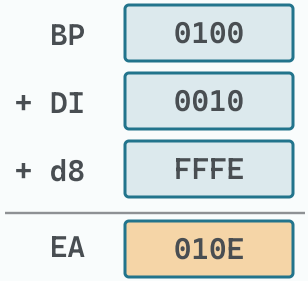
`MOV AX, [BP + DI − 2]` · SS = 2000h, BP = 0100h, DI = 0010h



Signed displacement −2 (FEh in machine code).

Access Through BP

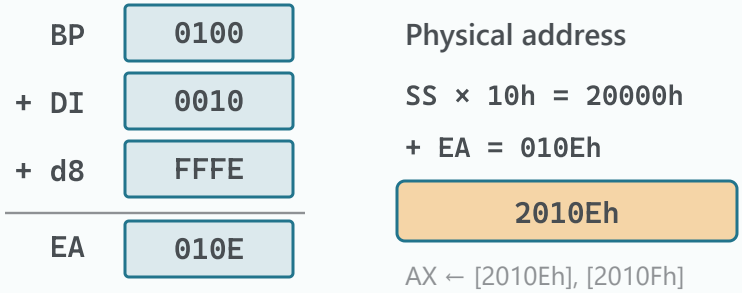
`MOV AX, [BP + DI − 2]` · SS = 2000h, BP = 0100h, DI = 0010h



EA = 0100h + 0010h − 2 = **010Eh**.

Access Through BP

`MOV AX, [BP + DI − 2]` · SS = 2000h, BP = 0100h, DI = 0010h



BP is used, so the segment is **SS**: 20000h + 010Eh = **2010Eh**. AX ← [2010Eh] (2 bytes: 2010Eh and 2010Fh).

ADDRESSING MODES

Effective Address Calculator

[BX + SI + 4] default segment

BX	BP	SI	DI
0200	0100	0030	0010

DS	ES	SS	CS
1000	B800	2000	0700

[BX + SI + 0004h]

EA = BX (0200h) + SI (0030h) + 0004h = **0234h**

DS × 10h + EA = 10000h + 0234h = **10234h**

Mode: **based-indexed + displacement** · displacement: d8 · segment: **DS** (default DS)

Register values are hex; the displacement may be decimal (`-2`) or hex (`1Ah`).

Addressing Mode by Data Structure

Data accessed	Suitable mode	Example
A single variable	direct	`MOV AX, count`
Element a pointer points to	register indirect	`MOV AL, [SI]`
Element i of an array (fixed array, variable index)	indexed + d16	`MOV AL, arr[SI]`
A field of a record (structure)	based + d8	`MOV AX, [BX+2]`
Matrix or array of records	base + index + d	`MOV AL, mat[BX+SI]`
Parameter or local variable on the stack	based, with BP (SS)	`MOV AX, [BP+4]`
Buffer in another segment	segment override	`MOV AL, ES:[DI]`

Base = which record / row

BX or BP holds the start of the structure; moving to another row updates only the base register.

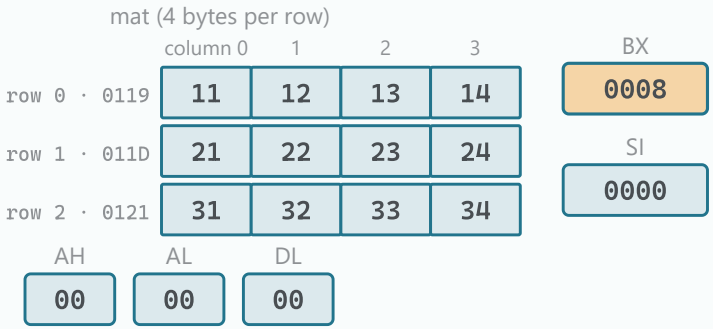
Index = element within the record

SI or DI steps through the loop; the displacement is the fixed position of the array or field.

The same data can be reached with several modes; choose by what is fixed and what changes at run time.

Code Trace: Accessing a Matrix Element

```
1  org 100h
▶  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



BX = 2 × 4 = **0008h**: row 2 starts 8 bytes into the matrix (row × row length).

Step 1 / 8

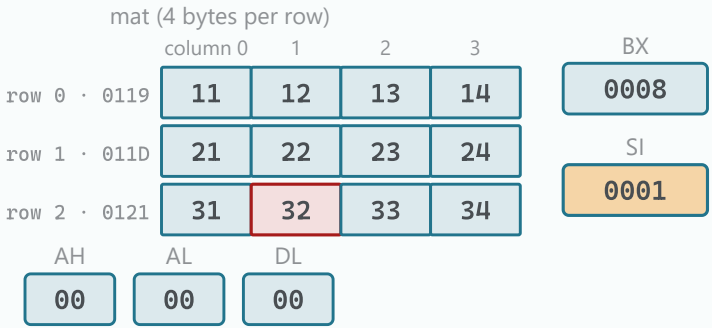
VARIABLES

BX = **0008h** SI = **0000h** AX = **0000h** DX = **0000h**

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



SI = 1: the column within the row. The base register selects the row, the index register the column.

Step 2 / 8

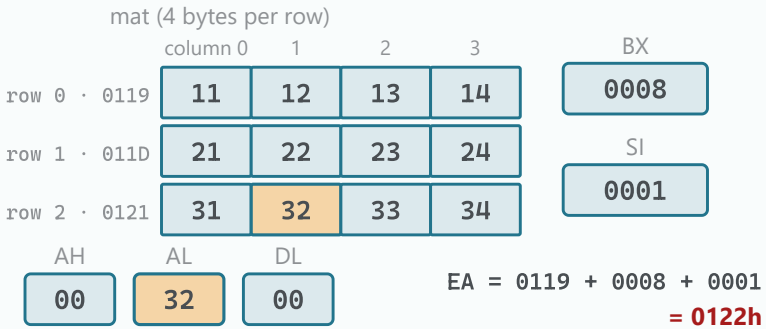
VARIABLES

BX = 0008h | SI = 0001h | AX = 0000h | DX = 0000h

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
▶  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11    db 21h,22h,23h,24h
12    db 31h,32h,33h,34h
```



`mat[bx+si]` : EA = 0119h + 8 + 1 = **0122h** → AL = **32h** (row 2, column 1). The displacement is the offset of `mat` (d16).

Step 3 / 8

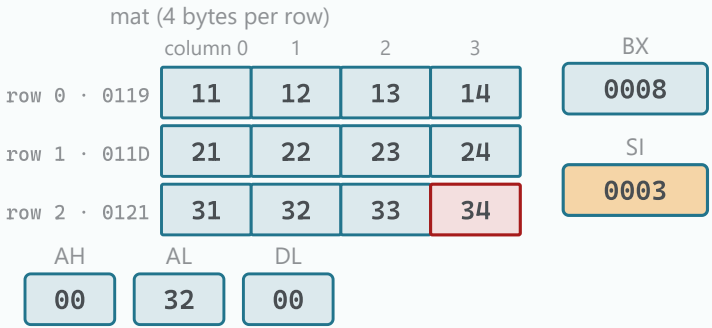
VARIABLES

BX = **0008h** | SI = **0001h** | AX = **0032h** | DX = **0000h**

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
▶  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



SI = 3: another column in the same row. BX is unchanged, so the row stays fixed.

Step 4 / 8

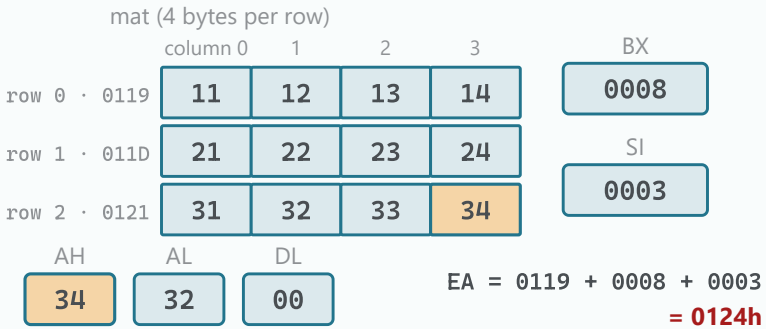
VARIABLES

BX = **0008h** | SI = **0003h** | AX = **0032h** | DX = **0000h**

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
▶  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



EA = 0124h → AH = **34h**. Same instruction; only the index changed.

Step 5 / 8

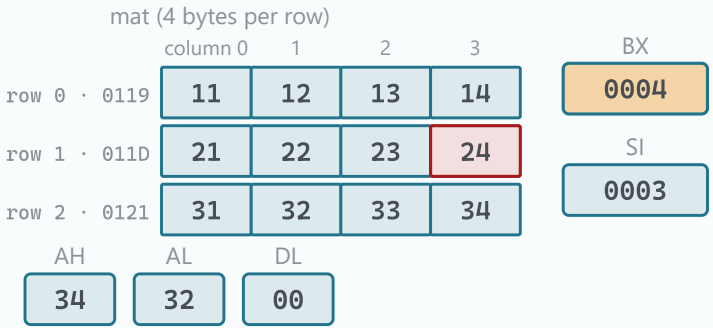
VARIABLES

BX = **0008h** | SI = **0003h** | AX = **3432h** | DX = **0000h**

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
▶  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



`sub bx, 4`: one row up (BX = 0004h). The column (SI = 3) is kept.

Step 6 / 8

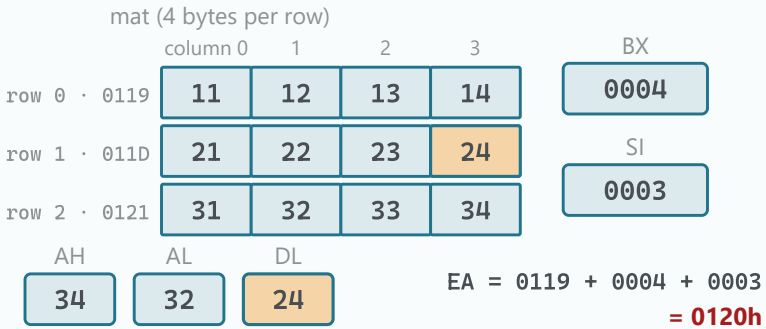
VARIABLES

BX = 0004h | SI = 0003h | AX = 3432h | DX = 0000h

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
▶  mov dl, mat[bx+si]   ; mat[1][3]
9  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



EA = 0119h + 4 + 3 = 0120h → DL = **24h** (row 1, column 3).

Step 7 / 8

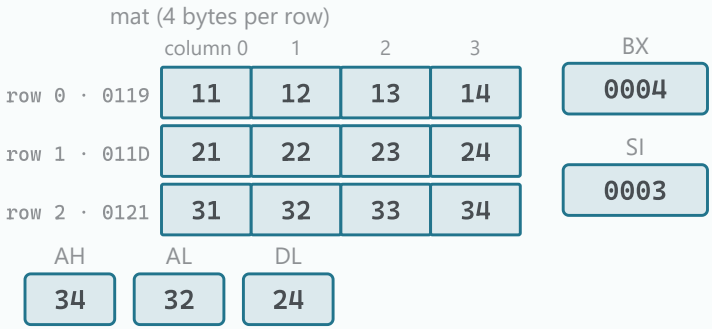
VARIABLES

BX = **0004h** | SI = **0003h** | AX = **3432h** | DX = **0024h**

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Code Trace: Accessing a Matrix Element

```
1  org 100h
2  mov bx, 2*4          ; row 2
3  mov si, 1            ; column 1
4  mov al, mat[bx+si]   ; mat[2][1]
5  mov si, 3            ; column 3
6  mov ah, mat[bx+si]   ; mat[2][3]
7  sub bx, 4            ; row 1
8  mov dl, mat[bx+si]   ; mat[1][3]
▶  ret
10 mat db 11h,12h,13h,14h
11     db 21h,22h,23h,24h
12     db 31h,32h,33h,34h
```



`ret` : AX = **3432h**, DL = 24h. Three elements were read with a single addressing pattern.

Step 8 / 8

VARIABLES

BX = **0004h** SI = **0003h** AX = **3432h** DX = **0024h**

Row-major order: address of element[row][col] = mat + row × number of columns + col. For a word matrix, both terms are doubled.

Effective Address Calculation Time

EA components	Clocks
`[BX]`, `[BP]`, `[SI]`, `[DI]`	5
direct `[d16]`	6
`[BP+DI]`, `[BX+SI]`	7
`[BP+SI]`, `[BX+DI]`	8
one register + d8/d16	9
`[BP+DI+d]`, `[BX+SI+d]`	11
`[BP+SI+d]`, `[BX+DI+d]`	12
segment override	+2

On the 8086 the execution unit computes the effective address with a separate addition; that time is added to the instruction's own time.

- `MOV AX, BX` : 2 clocks (no memory).
- `MOV CL, [BX]` : 8 + 5 = **13** clocks.
- `MOV CL, [BX+DI+4]` : 8 + 12 = **20** clocks.
- A word at an odd address needs +4 clocks for the second bus cycle.

Keeping a value that a loop reads often in a register removes this calculation from every pass.

Figures are from the table in the Intel 8086 user's manual; from the 80286 on, address calculation has its own unit and these differences shrink.

Segment and Offset

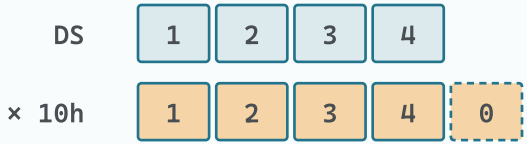
- The value in a segment register (CS, DS, SS, ES) is the **segment**; the value in an addressing register (BX, SI, DI, BP) is the **offset**.
- The segment selects a **block** of memory and the offset a **specific location** in it; together they form the physical address.



DS = **1234h**, SI = **7890h**. Notation: `1234:7890`.

Segment and Offset

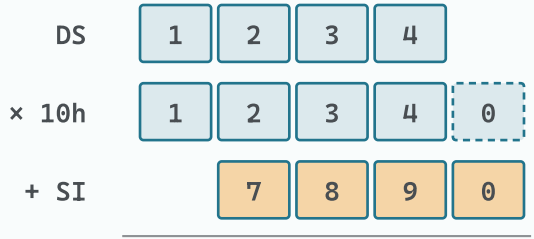
- The value in a segment register (CS, DS, SS, ES) is the **segment**; the value in an addressing register (BX, SI, DI, BP) is the **offset**.
- The segment selects a **block** of memory and the offset a **specific location** in it; together they form the physical address.



The segment is multiplied by 10h (shifted left 4 bits): **12340h**.

Segment and Offset

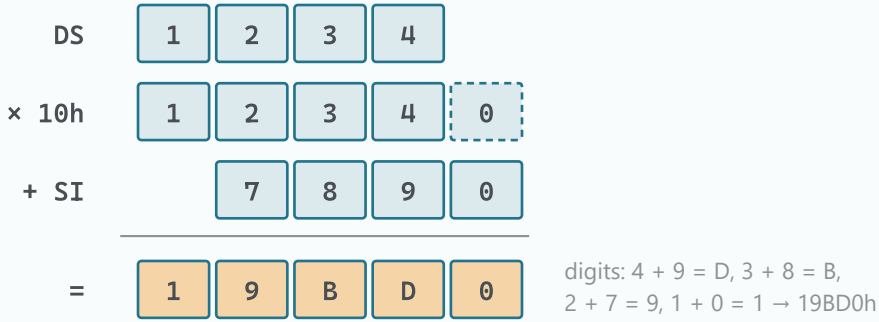
- The value in a segment register (CS, DS, SS, ES) is the **segment**; the value in an addressing register (BX, SI, DI, BP) is the **offset**.
- The segment selects a **block** of memory and the offset a **specific location** in it; together they form the physical address.



The offset is aligned for the addition: **7890h**.

Segment and Offset

- The value in a segment register (CS, DS, SS, ES) is the **segment**; the value in an addressing register (BX, SI, DI, BP) is the **offset**.
- The segment selects a **block** of memory and the offset a **specific location** in it; together they form the physical address.



12340h + 7890h = physical address **19BD0h**.

Address Lines and Memory Capacity

n address lines select 2^n locations. The number of data lines sets the bits per location:

capacity = 2^n × data width

- 8086: 20 lines → 2^{20} = 1,048,576 locations = **1 MB** (00000h–FFFFFh).
- A segment's limit comes from the offset: 16 bits → 2^{16} = **64 KB**.
- The reverse question: how many lines at least? Take the first power of 2 that covers the capacity: for 6 GB, 2^{32} = 4 GB is not enough, so **33** lines (8 GB).

Address / data lines	Capacity
12 / 8	4096 × 1 B = 4 KB
16 / 4	64 K × 4 bits = 32 KB
16 / 8	64 KB (one segment)
20 / 8	1 MB (8086)
24 / 16	16 M × 2 B = 32 MB
32 / 8	4 GB (80386)

On a memory chip the number of data lines is the word width; for a capacity in bytes, divide the bit count by 8.

Working Backwards: DS:SI from a Physical Address

We want to reach physical address **2A5C3h** through DS:SI. There is no single answer; we find the range DS can take.

Largest DS

Make SI as small as possible: DS = first four digits = **2A5Ch**, SI = 2A5C3h − 2A5C0h = **0003h**.

Smallest DS

SI can be at most FFFFh: DS × 10h ≥ 2A5C3h − FFFFh = 1A5C4h → DS ≥ 1A5C.4h → **1A5Dh**, SI = **FFF3h**.

How many pairs?

Each step up in DS lowers SI by 10h: 2A5Ch − 1A5Dh + 1 = 1000h = **4096** pairs.



Always check with the forward calculation: 1A5Dh × 10h + FFF3h = 1A5D0h + FFF3h = 2A5C3h.

Segment Override Prefix

Writing a **segment override prefix** before the brackets replaces the default segment:

```
MOV AX, 0B800h
MOV ES, AX          ; ES = video memory
MOV BX, 0
MOV AL, ES:[BX]     ; read ES:0000
MOV AL, [BX]        ; read DS:0000
MOV DL, SS:[SI]     ; read SS:SI
```

The prefix is a 1-byte code placed before the instruction in machine code (ES: → 26h, CS: → 2Eh, SS: → 36h, DS: → 3Eh).

When is it needed?

- To reach data in another segment (e.g. video memory) without changing DS.
- To reach stack data through a register other than BP: `SS:[BX]`.
- To read a table stored in the code: `CS:[SI]`.

IP always pairs with CS and SP always with SS; these cannot be overridden.

Code Trace: Video Memory Through ES

```
1 ORG 100h
▶ MOV AX, 0B800h
3 MOV ES, AX          ; ES = video
4 MOV BX, 00A0h       ; row 1
5 MOV BYTE PTR ES:[BX], 'K'
6 MOV BYTE PTR ES:[BX+1], 1Eh
7 MOV AL, [BX]        ; DS:00A0h
8 MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```



AX = B800h: the segment of the color text screen. In video memory one cell = character + color.

Step 1 / 7

VARIABLES

DS = **0700h** ES = **0700h** BX = **0000h** AL = **00h** DL = **00h**

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Video Memory Through ES

```
1 ORG 100h
2 MOV AX, 0B800h
▶ MOV ES, AX          ; ES = video
4 MOV BX, 00A0h       ; row 1
5 MOV BYTE PTR ES:[BX], 'K'
6 MOV BYTE PTR ES:[BX+1], 1Eh
7 MOV AL, [BX]        ; DS:00A0h
8 MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```



ES = **B800h**. DS still holds the program's segment (0700h), so variable access is unaffected.

Step 2 / 7

VARIABLES

DS = **0700h** · ES = **B800h** · BX = **0000h** · AL = **00h** · DL = **00h**

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Video Memory Through ES

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV ES, AX          ; ES = video
▶ MOV BX, 00A0h       ; row 1
5 MOV BYTE PTR ES:[BX], 'K'
6 MOV BYTE PTR ES:[BX+1], 1Eh
7 MOV AL, [BX]        ; DS:00A0h
8 MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```

DS = 0700h

07000h + 00A0h = 070A0h

00h
+0

ES = B800h

B8000h + 00A0h = B80A0h

20h	07h
+0	+1

screen, row 1: 

AL	DL
00h	00h

BX = 00A0h = 160 = (1 × 80 + 0) × 2: the first cell of row 1.

Step 3 / 7

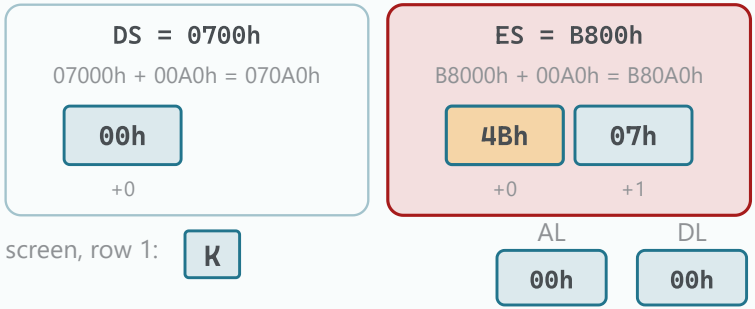
VARIABLES

DS = 0700h · ES = B800h · BX = 00A0h · AL = 00h · DL = 00h

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Video Memory Through ES

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV ES, AX          ; ES = video
4 MOV BX, 00A0h       ; row 1
▶ MOV BYTE PTR ES:[BX], 'K'
6 MOV BYTE PTR ES:[BX+1], 1Eh
7 MOV AL, [BX]        ; DS:00A0h
8 MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```



The `ES:` prefix (byte 26h in machine code): B8000h + 00A0h = **B80A0h** ← 'K' (4Bh).

Step 4 / 7

VARIABLES

`DS = 0700h` `ES = B800h` `BX = 00A0h` `AL = 00h` `DL = 00h`

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Video Memory Through ES

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV ES, AX          ; ES = video
4 MOV BX, 00A0h       ; row 1
5 MOV BYTE PTR ES:[BX], 'K'
▶ MOV BYTE PTR ES:[BX+1], 1Eh
7 MOV AL, [BX]        ; DS:00A0h
8 MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```



B80A1h ← **1Eh**: yellow on blue. A yellow K appears on the screen.

Step 5 / 7

VARIABLES

`DS = 0700h` `ES = B800h` `BX = 00A0h` `AL = 00h` `DL = 00h`

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Video Memory Through ES

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV ES, AX          ; ES = video
4 MOV BX, 00A0h       ; row 1
5 MOV BYTE PTR ES:[BX], 'K'
6 MOV BYTE PTR ES:[BX+1], 1Eh
▶ MOV AL, [BX]        ; DS:00A0h
8 MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```



No prefix → default **DS**: 07000h + 00A0h = 070A0h is read (the program's PSP area, 00h here). Same offset, different byte.

Step 6 / 7

VARIABLES

DS = **0700h** ES = **B800h** BX = **00A0h** AL = **00h** DL = **00h**

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Video Memory Through ES

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV ES, AX          ; ES = video
4 MOV BX, 00A0h       ; row 1
5 MOV BYTE PTR ES:[BX], 'K'
6 MOV BYTE PTR ES:[BX+1], 1Eh
7 MOV AL, [BX]        ; DS:00A0h
▶ MOV DL, ES:[BX]     ; B800:00A0h
9 RET
```

DS = 0700h

07000h + 00A0h = 070A0h

00h
+0

screen, row 1: **K**

ES = B800h

B8000h + 00A0h = B80A0h

4Bh	1Eh
+0	+1

AL	DL
00h	4Bh

`ES:[BX]` goes to video memory again: DL = **4Bh** ('K').

Step 7 / 7

VARIABLES

DS = **0700h** · ES = **B800h** · BX = **00A0h** · AL = **00h** · DL = **4Bh**

The video cell is assumed to start as 20h 07h (space, gray). The prefix affects only its own instruction; DS does not change.

Code Trace: Reaching Data Through BP

```
 1 data segment
 2 pad db 3 dup(0)
 3 x db 'D'
 4 data ends
 5 stk segment stack
 6 dw 64 dup(0)
 7 stk ends
 8 code segment
 9 assume cs:code, ds:data, ss:stk
10 start:
▶ mov ax, data
12 mov ds, ax
13 mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]       ; DS:BP
16 mov si, offset x
17 mov cl, [si]          ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```

data: DS = 0700h

phys. 07100h…

0000	0001	0002	0003
00	00	00	44

stack: SS = 0711h

phys. 07110h…

0000	0001	0002	0003
00	00	00	00

BP	SI	AL	BL	CL
0000	0000	10	00	00

AX = **0710h**: the paragraph number of the `data` segment (set by the loader). When an EXE starts, DS does not point to its data.

Step 1 / 8

VARIABLES

DS = **0700h** SS = **0711h** BP = **0000h** SI = **0000h** AL = **10h** BL = **00h** CL = **00h**

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
 1 data segment
 2 pad db 3 dup(0)
 3 x db 'D'
 4 data ends
 5 stk segment stack
 6 dw 64 dup(0)
 7 stk ends
 8 code segment
 9 assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
▶ mov ds, ax
13 mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]       ; DS:BP
16 mov si, offset x
17 mov cl, [si]          ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```



DS = 0710h. The stack segment SS = **0711h** is a separate block; in this program DS ≠ SS.

Step 2 / 8

VARIABLES

DS = **0710h** | SS = **0711h** | BP = **0000h** | SI = **0000h** | AL = **10h** | BL = **00h** | CL = **00h**

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
 1 data segment
 2 pad db 3 dup(0)
 3 x db 'D'
 4 data ends
 5 stk segment stack
 6 dw 64 dup(0)
 7 stk ends
 8 code segment
 9 assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
12 mov ds, ax
▶ mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]       ; DS:BP
16 mov si, offset x
17 mov cl, [si]          ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```

data: DS = 0710h

phys. 07100h…

0000	0001	0002	0003
00	00	00	44

stack: SS = 0711h

phys. 07110h…

0000	0001	0002	0003
00	00	00	00

BP	SI	AL	BL	CL
0003	0000	10	00	00

BP = **0003h**: the offset of `x`. The offset is right; the trouble is the segment.

Step 3 / 8

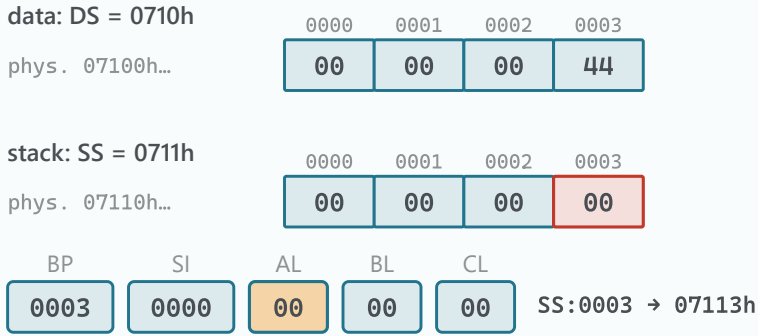
VARIABLES

DS = **0710h** · SS = **0711h** · BP = **0003h** · SI = **0000h** · AL = **10h** · BL = **00h** · CL = **00h**

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
1  data segment
2  pad db 3 dup(0)
3  x db 'D'
4  data ends
5  stk segment stack
6  dw 64 dup(0)
7  stk ends
8  code segment
9  assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
12 mov ds, ax
13 mov bp, offset x
▶ mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]      ; DS:BP
16 mov si, offset x
17 mov cl, [si]         ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```



`[bp]` pairs with **SS** by default: 07113h is read, AL = **00h**. Not 'D', but a byte from the stack.

Step 4 / 8

VARIABLES

DS = **0710h** · SS = **0711h** · BP = **0003h** · SI = **0000h** · AL = **00h** · BL = **00h** · CL = **00h**

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
1  data segment
2  pad db 3 dup(0)
3  x db 'D'
4  data ends
5  stk segment stack
6  dw 64 dup(0)
7  stk ends
8  code segment
9  assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
12 mov ds, ax
13 mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
▶ mov bl, ds:[bp]        ; DS:BP
16 mov si, offset x
17 mov cl, [si]          ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```

data: DS = 0710h

phys. 07100h…

0000	0001	0002	0003
00	00	00	44

stack: SS = 0711h

phys. 07110h…

0000	0001	0002	0003
00	00	00	00

BP	SI	AL	BL	CL
0003	0000	00	44	00

DS:0003 → 07103h

`ds:[bp]` : the prefix overrides the default, 07103h is read → BL = **44h** ('D').

Step 5 / 8

VARIABLES

DS = **0710h** · SS = **0711h** · BP = **0003h** · SI = **0000h** · AL = **00h** · BL = **44h** · CL = **00h**

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
 1 data segment
 2 pad db 3 dup(0)
 3 x db 'D'
 4 data ends
 5 stk segment stack
 6 dw 64 dup(0)
 7 stk ends
 8 code segment
 9 assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
12 mov ds, ax
13 mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]       ; DS:BP
▶ mov si, offset x
17 mov cl, [si]          ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```

data: DS = 0710h

phys. 07100h…

0000	0001	0002	0003
00	00	00	44

stack: SS = 0711h

phys. 07110h…

0000	0001	0002	0003
00	00	00	00

BP	SI	AL	BL	CL
0003	0003	00	44	00

The same offset goes into SI: SI = 0003h.

Step 6 / 8

VARIABLES

DS = 0710h · SS = 0711h · BP = 0003h · SI = 0003h · AL = 00h · BL = 44h · CL = 00h

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
1  data segment
2  pad db 3 dup(0)
3  x db 'D'
4  data ends
5  stk segment stack
6  dw 64 dup(0)
7  stk ends
8  code segment
9  assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
12 mov ds, ax
13 mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]       ; DS:BP
16 mov si, offset x
▶  mov cl, [si]          ; DS:SI
18 mov ax, 4C00h
19 int 21h
20 code ends
21 end start
```

data: DS = 0710h

phys. 07100h…

0000	0001	0002	0003
00	00	00	44

stack: SS = 0711h

phys. 07110h…

0000	0001	0002	0003
00	00	00	00

BP	SI	AL	BL	CL
0003	0003	00	44	44

DS:0003 → 07103h

`[si]` uses **DS** by default: CL = **44h**. Use BX/SI/DI for data and BP for the stack.

Step 7 / 8

VARIABLES

DS = **0710h** SS = **0711h** BP = **0003h** SI = **0003h**

AL = **00h** BL = **44h** CL = **44h**

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Code Trace: Reaching Data Through BP

```
1  data segment
2  pad db 3 dup(0)
3  x db 'D'
4  data ends
5  stk segment stack
6  dw 64 dup(0)
7  stk ends
8  code segment
9  assume cs:code, ds:data, ss:stk
10 start:
11 mov ax, data
12 mov ds, ax
13 mov bp, offset x
14 mov al, [bp]          ; reads SS:BP!
15 mov bl, ds:[bp]       ; DS:BP
16 mov si, offset x
17 mov cl, [si]          ; DS:SI
▶  mov ax, 4C00h
▶  int 21h
20 code ends
21 end start
```

data: DS = 0710h

phys. 07100h…

0000	0001	0002	0003
00	00	00	44

stack: SS = 0711h

phys. 07110h…

0000	0001	0002	0003
00	00	00	00

BP	SI	AL	BL	CL
0003	0003	00	44	44

AH = 4Ch, AL = 0: the program returns to DOS with exit code 0.

Step 8 / 8

VARIABLES

DS = 0710h · SS = 0711h · BP = 0003h · SI = 0003h · AL = 00h · BL = 44h · CL = 44h

In a .COM program CS = DS = SS, so this bug stays hidden; it shows up in an .EXE program with a separate stack segment.

Changing a Segment Register

```
PUSH DS             ; save DS
MOV  AX, 0B800h
MOV  DS, AX         ; temporary
MOV  BYTE PTR [0], 'Z'
POP  DS             ; variables back

MOV  AX, new_stack
MOV  SS, AX         ; right after it
MOV  SP, 0400h      ; set SP
```

DS

Variable names are only offsets; once DS changes, the same name reaches a byte in another segment. Wrap a temporary change in PUSH DS / POP DS, or use ES.

SS and SP

Change them together. After a MOV or POP into a segment register, the 8086 accepts no interrupt until the next instruction finishes, so SS:SP is never left half-set.

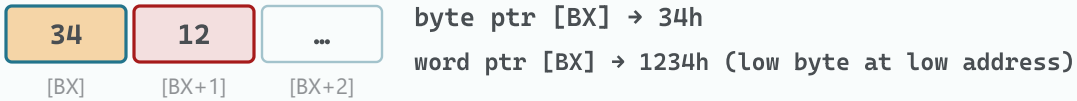
CS

Cannot be written with MOV; a far JMP/CALL/RET or INT/IRET changes CS and IP together.

ES is used by default only as the string destination, which makes it the safest way to reach a second data block (video memory, a buffer).

Specifying the Data Type

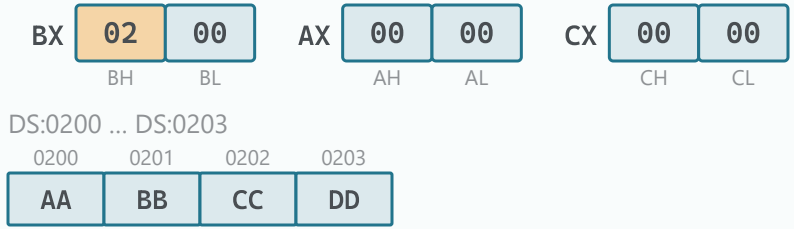
- **Type specifiers** tell the assembler the size of the data being accessed:
  - `byte ptr [BX]` : a byte (8-bit) access.
  - `word ptr [BX]` : a two-byte (16-bit, one word) access.
- emu8086 also accepts shorter forms:
  - `b.[SI]` : the value at the address in SI, as a byte.
  - `w.[DI]` : the value at the address in DI, as a word.
- Often the assembler infers the type **itself**: if the other operand is a register, its size wins.



Instruction	Where does the size come from?
`MOV [BX], AL`	AL is 8 bits → a byte is written
`MOV [BX], AX`	AX is 16 bits → a word is written
`MOV [BX], 5`	**ambiguous!** → `MOV BYTE PTR [BX], 5` or `MOV WORD PTR [BX], 5`

Code Trace: byte ptr and word ptr

```
▶ MOV BX, 0200h
2 MOV WORD PTR [BX], 1234h
3 MOV BYTE PTR [BX], 0FFh
4 MOV WORD PTR [BX], 0FFh
5 MOV AX, [BX]          ; word
6 MOV CL, [BX+1]        ; byte
```



BX = 0200h. Memory still holds earlier values (AA BB CC DD).

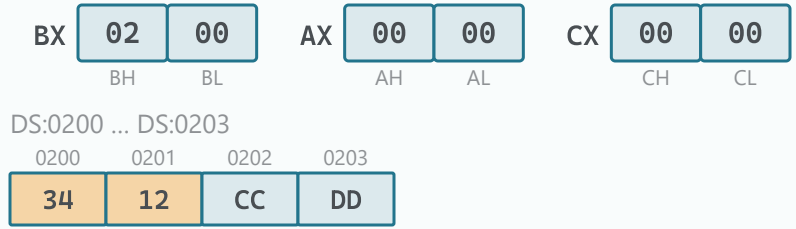
Step 1 / 6

VARIABLES

`BX = 0200h` `AX = 0000h` `CL = 00h` `w[0200] = BBAAh` `w[0202] = DDCCh`

Code Trace: byte ptr and word ptr

```
1 MOV BX, 0200h
▶ MOV WORD PTR [BX], 1234h
3 MOV BYTE PTR [BX], 0FFh
4 MOV WORD PTR [BX], 0FFh
5 MOV AX, [BX]          ; word
6 MOV CL, [BX+1]        ; byte
```



word ptr: 2 bytes are written. The low byte (34h) goes to the low address: **34 12**.

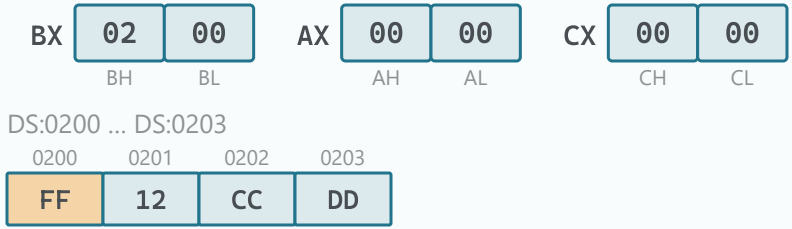
Step 2 / 6

VARIABLES

`BX = 0200h` `AX = 0000h` `CL = 00h` `w[0200] = 1234h` `w[0202] = DDCCh`

Code Trace: byte ptr and word ptr

```
1 MOV BX, 0200h
2 MOV WORD PTR [BX], 1234h
▶ MOV BYTE PTR [BX], 0FFh
4 MOV WORD PTR [BX], 0FFh
5 MOV AX, [BX]          ; word
6 MOV CL, [BX+1]        ; byte
```



byte ptr: only 0200h changes → FF 12. 0201h is untouched.

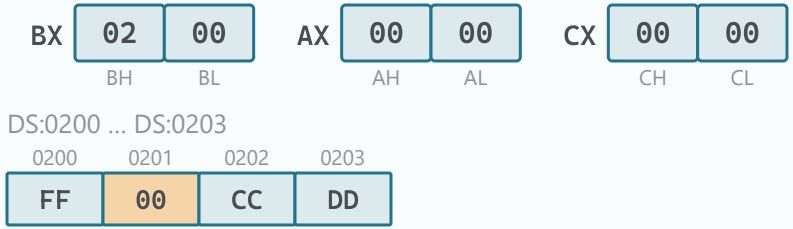
Step 3 / 6

VARIABLES

BX = **0200h** · AX = **0000h** · CL = **00h** · w[0200] = **12FFh** · w[0202] = **DDCCh**

Code Trace: byte ptr and word ptr

```
1 MOV BX, 0200h
2 MOV WORD PTR [BX], 1234h
3 MOV BYTE PTR [BX], 0FFh
▶ MOV WORD PTR [BX], 0FFh
5 MOV AX, [BX]          ; word
6 MOV CL, [BX+1]        ; byte
```



Same constant, **word ptr**: 00FFh is written → **FF 00**. 0201h changed too!

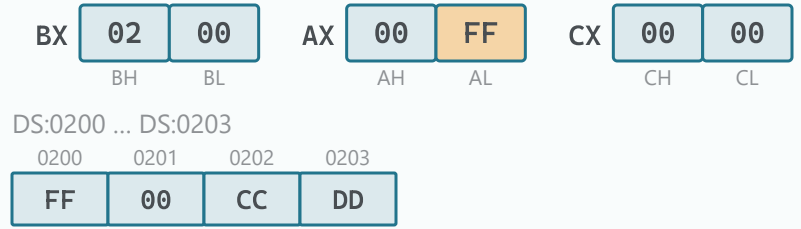
Step 4 / 6

VARIABLES

`BX = 0200h` `AX = 0000h` `CL = 00h` `w[0200] = 00FFh` `w[0202] = DDCCh`

Code Trace: byte ptr and word ptr

```
1 MOV BX, 0200h
2 MOV WORD PTR [BX], 1234h
3 MOV BYTE PTR [BX], 0FFh
4 MOV WORD PTR [BX], 0FFh
▶ MOV AX, [BX]          ; word
6 MOV CL, [BX+1]        ; byte
```



AX is 16 bits, so the size is clear: AX = **00FFh** (0201h is the high byte).

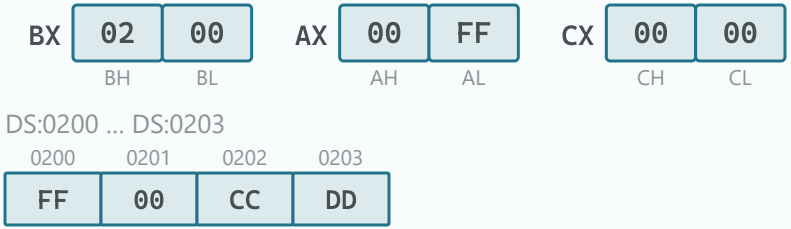
Step 5 / 6

VARIABLES

`BX = 0200h` `AX = 00FFh` `CL = 00h` `w[0200] = 00FFh` `w[0202] = DDCCh`

Code Trace: byte ptr and word ptr

```
1 MOV BX, 0200h
2 MOV WORD PTR [BX], 1234h
3 MOV BYTE PTR [BX], 0FFh
4 MOV WORD PTR [BX], 0FFh
5 MOV AX, [BX]          ; word
▶ MOV CL, [BX+1]        ; byte
```



CL is 8 bits: the byte at [BX+1] = 0201h → CL = **00h**.

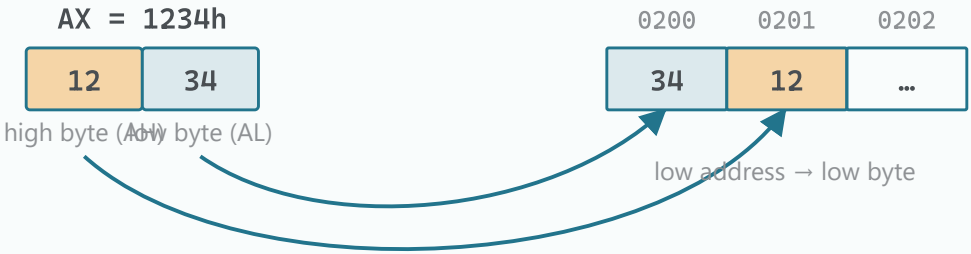
Step 6 / 6

VARIABLES

`BX = 0200h` `AX = 00FFh` `CL = 00h` `w[0200] = 00FFh` `w[0202] = DDCCh`

DATA TYPE

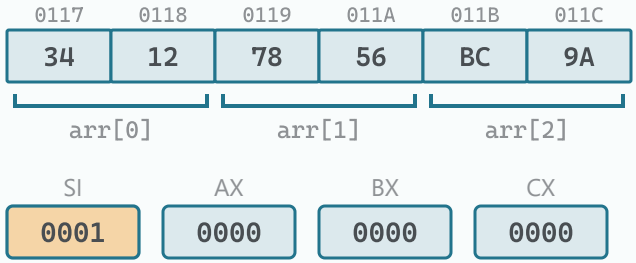
Little-Endian Order



- The 8086 is **little-endian**: the **low byte** of a 16-bit value is stored at the **low address**, the high byte at the next one.
- `MOV [0200h], AX` → [0200h] = AL, [0201h] = AH. `MOV AX, [0200h]` does the reverse.
- That is why words look “reversed” in the emu8086 memory window: 1234h → `34 12`.
- The rule also applies to constants in machine code: `MOV AX, 1234h` → `B8 34 12`.

Code Trace: The Word-Array Index Bug

```
1  org 100h
▶ mov si, 1              ; index 1
3  mov ax, arr[si]       ; BUG!
4  shl si, 1             ; index × 2
5  mov bx, arr[si]       ; arr[1]
6  mov si, 2             ; index 2
7  shl si, 1             ; SI = 4
8  mov cx, arr[si]       ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



SI = 1, meaning “the second element”. But SI is a **byte** distance.

Step 1 / 7

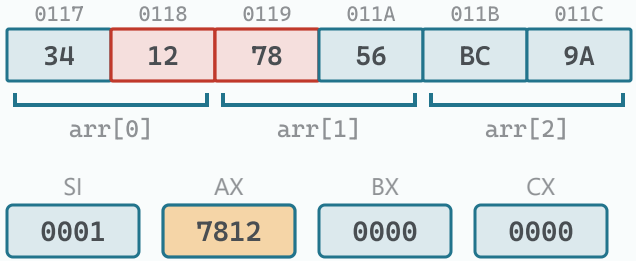
VARIABLES

`SI = 0001h` `AX = 0000h` `BX = 0000h` `CX = 0000h`

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

Code Trace: The Word-Array Index Bug

```
1  org 100h
2  mov si, 1            ; index 1
▶  mov ax, arr[si]      ; BUG!
4  shl si, 1            ; index × 2
5  mov bx, arr[si]      ; arr[1]
6  mov si, 2            ; index 2
7  shl si, 1            ; SI = 4
8  mov cx, arr[si]      ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



EA = 0117 + 0001 = 0118

`arr[si]` = 0118h: the high byte of arr[0] and the low byte of arr[1] are read → AX = **7812h**. Neither 1234h nor 5678h.

Step 2 / 7

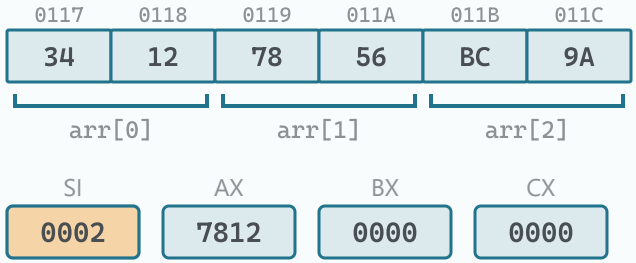
VARIABLES

`SI = 0001h` `AX = 7812h` `BX = 0000h` `CX = 0000h`

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

Code Trace: The Word-Array Index Bug

```
1  org 100h
2  mov si, 1            ; index 1
3  mov ax, arr[si]      ; BUG!
▶  shl si, 1            ; index × 2
5  mov bx, arr[si]      ; arr[1]
6  mov si, 2            ; index 2
7  shl si, 1            ; SI = 4
8  mov cx, arr[si]      ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



`shl si, 1` : index × 2 → SI = 2. In a word array, byte distance = index × 2.

Step 3 / 7

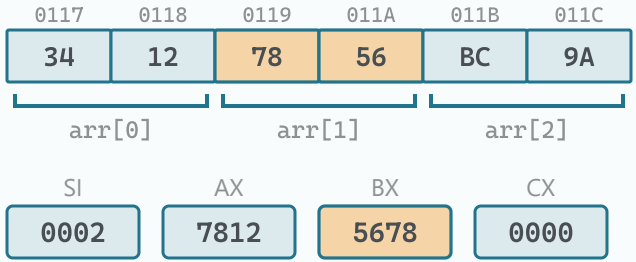
VARIABLES

SI = 0002h | AX = 7812h | BX = 0000h | CX = 0000h

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

Code Trace: The Word-Array Index Bug

```
1  org 100h
2  mov si, 1            ; index 1
3  mov ax, arr[si]      ; BUG!
4  shl si, 1            ; index × 2
▶  mov bx, arr[si]      ; arr[1]
6  mov si, 2            ; index 2
7  shl si, 1            ; SI = 4
8  mov cx, arr[si]      ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



EA = 0117 + 0002 = 0119

EA = 0119h → BX = **5678h** = arr[1]. The right element.

Step 4 / 7

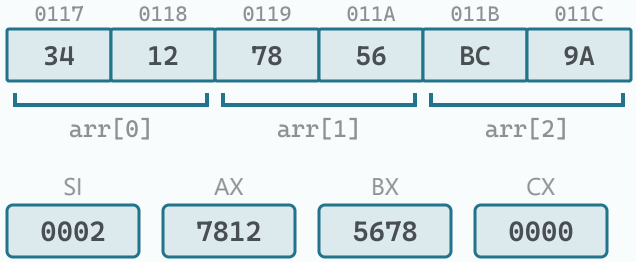
VARIABLES

SI = **0002h** AX = **7812h** BX = **5678h** CX = **0000h**

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

Code Trace: The Word-Array Index Bug

```
1  org 100h
2  mov si, 1            ; index 1
3  mov ax, arr[si]      ; BUG!
4  shl si, 1            ; index × 2
5  mov bx, arr[si]      ; arr[1]
▶  mov si, 2            ; index 2
7  shl si, 1            ; SI = 4
8  mov cx, arr[si]      ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



SI = 2: this time index 2 (the third element).

Step 5 / 7

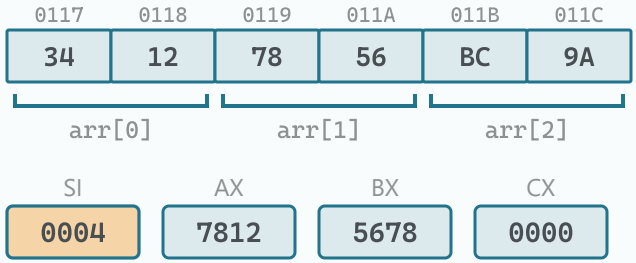
VARIABLES

`SI = 0002h` `AX = 7812h` `BX = 5678h` `CX = 0000h`

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

Code Trace: The Word-Array Index Bug

```
1  org 100h
2  mov si, 1            ; index 1
3  mov ax, arr[si]      ; BUG!
4  shl si, 1            ; index × 2
5  mov bx, arr[si]      ; arr[1]
6  mov si, 2            ; index 2
▶  shl si, 1            ; SI = 4
8  mov cx, arr[si]      ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



`shl si, 1` : SI = **4**. A shift is the shortest way to multiply by 2.

Step 6 / 7

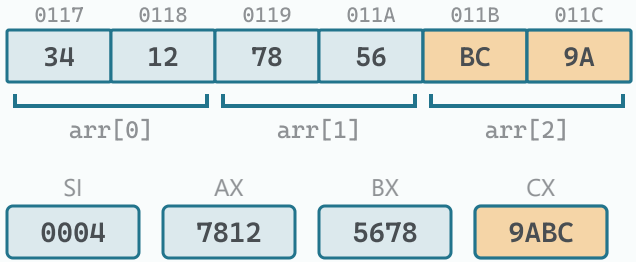
VARIABLES

`SI = 0004h` `AX = 7812h` `BX = 5678h` `CX = 0000h`

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

Code Trace: The Word-Array Index Bug

```
1  org 100h
2  mov si, 1            ; index 1
3  mov ax, arr[si]      ; BUG!
4  shl si, 1            ; index × 2
5  mov bx, arr[si]      ; arr[1]
6  mov si, 2            ; index 2
7  shl si, 1            ; SI = 4
▶  mov cx, arr[si]      ; arr[2]
9  ret
10 arr dw 1234h, 5678h, 9ABCh
```



`EA = 0117 + 0004 = 011B`

EA = 011Bh → CX = **9ABCh** = arr[2].

Step 7 / 7

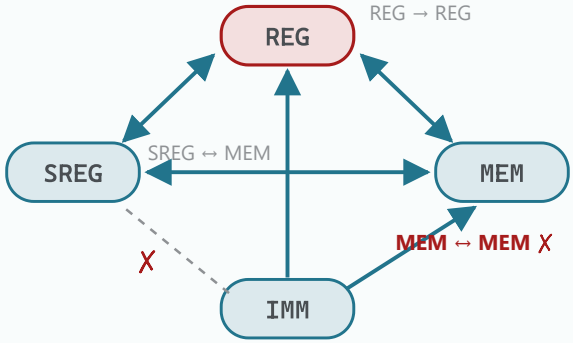
VARIABLES

`SI = 0004h` `AX = 7812h` `BX = 5678h` `CX = 9ABCh`

When a loop walks a word array, advance SI with `ADD SI, 2` on every pass; `INC SI` is for byte arrays only.

The MOV Instruction

- `MOV dest, source` **copies** the source value into the destination; the source is unchanged.
- **Source**: an immediate (constant), a general-purpose register, a segment register or a memory location.
- **Destination**: a general-purpose register, a segment register (not CS) or a memory location.
- It cannot load **CS or IP**.
- Both operands must be the **same size**: both bytes or both words.
- Only one operand may be memory.
- MOV **does not affect the flags**.



MOV AX, BX

Copies the value in BX into AX. BX is unchanged.

MOV [SI], 10h

Writes 10h to the address in SI (DS:SI). The size is ambiguous, so the correct form is `MOV BYTE PTR [SI], 10h`.

Supported Operands

General-purpose

```
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
```

Segment registers

```
MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
```

REG	AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP
SREG	DS, ES, SS and CS (CS only as the **second** operand, the source)
memory	`[BX]`, `[BX+SI+7]`, a variable...
immediate	`5`, `-24`, `3Fh`, `10001101b`...

There is no `MOV SREG, immediate` or `MOV SREG, SREG`; go through a general register first.

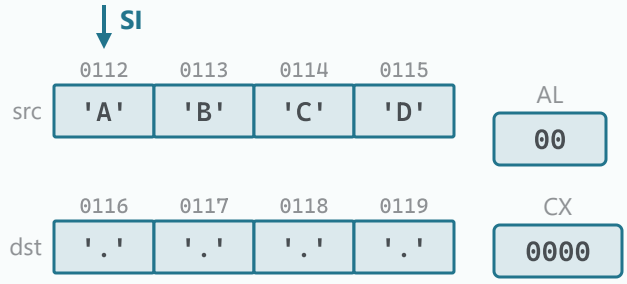
MOV: Valid or Not?

Instruction	?	Instruction	?
`MOV [BX], [SI]`	1	`MOV AX, CS`	6
`MOV DS, ES`	2	`MOV SP, BX`	7
`MOV DS, 100h`	3	`MOV BL, DX`	8
`MOV CS, AX`	4	`MOV AL, [SI+BP]`	9
`MOV [BX], 5`	5	`MOV 5, AX`	10

▶ **Answers**

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



SI = **0112h**: the offset of the source.

Step 1 / 12

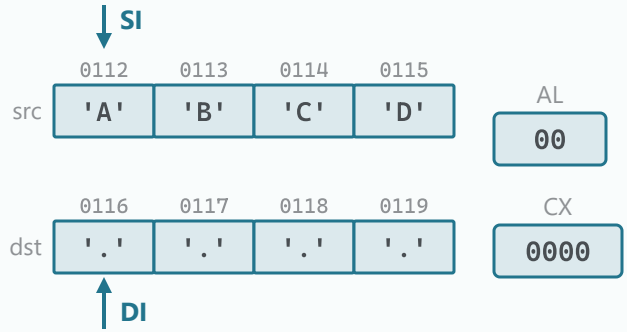
VARIABLES

`SI = 0112h` `DI = 0000h` `AL = 00h` `CX = 0000h`

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
▶  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



DI = **0116h**: the offset of the destination. Two pointers, two separate buffers.

Step 2 / 12

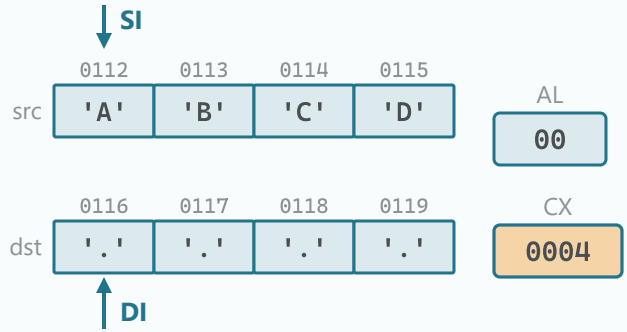
VARIABLES

SI = **0112h** | DI = **0116h** | AL = **00h** | CX = **0000h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
▶  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



CX = 4: the number of bytes to copy (the LOOP counter).

Step 3 / 12

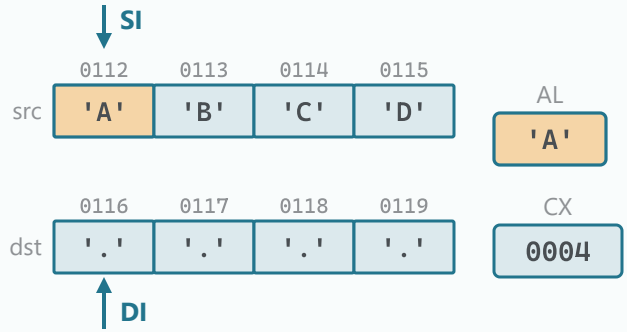
VARIABLES

SI = **0112h** DI = **0116h** AL = **00h** CX = **0004h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
▶  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



`mov al, [si]`: memory to register → AL = 'A' (41h). There is no memory-to-memory MOV, so AL is the stopover.

Step 4 / 12

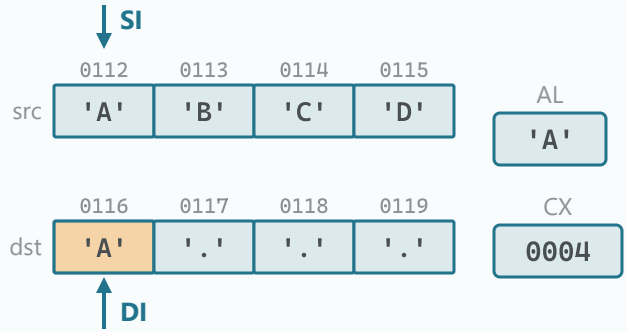
VARIABLES

SI = **0112h** DI = **0116h** AL = **41h** CX = **0004h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
▶  mov [di], al          ; AL → memory
8  inc si
9  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



`mov [di], al` : register to memory → [0116] = 41h.

Step 5 / 12

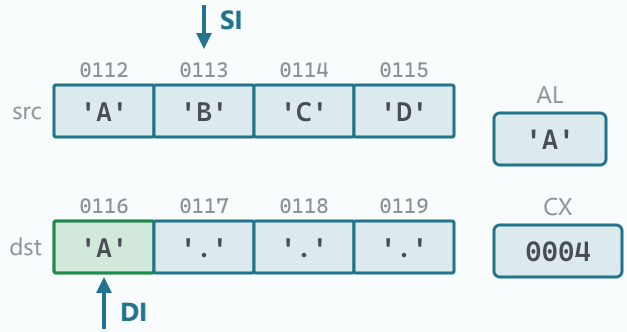
VARIABLES

SI = **0112h** DI = **0116h** AL = **41h** CX = **0004h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB` : DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
▶  inc si
9  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



`inc si` : the source pointer moves to the next byte.

Step 6 / 12

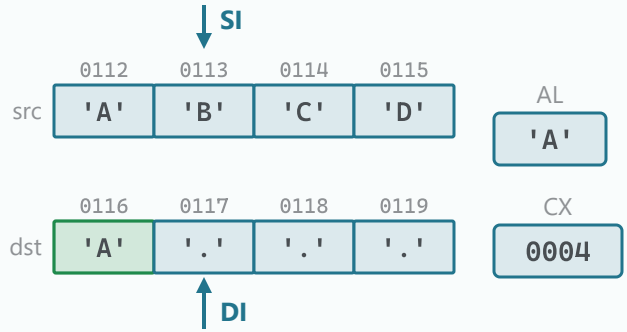
VARIABLES

SI = 0113h | DI = 0116h | AL = 41h | CX = 0004h

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB` : DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
▶  inc di
10 loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



`inc di` : so does the destination pointer. The two pointers advance together.

Step 7 / 12

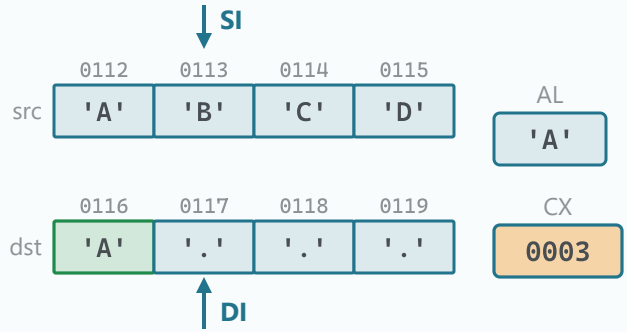
VARIABLES

SI = **0113h** | DI = **0117h** | AL = **41h** | CX = **0004h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB` : DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
▶  loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



`loop copy` : CX = 3 ≠ 0 → back to the top.

Step 8 / 12

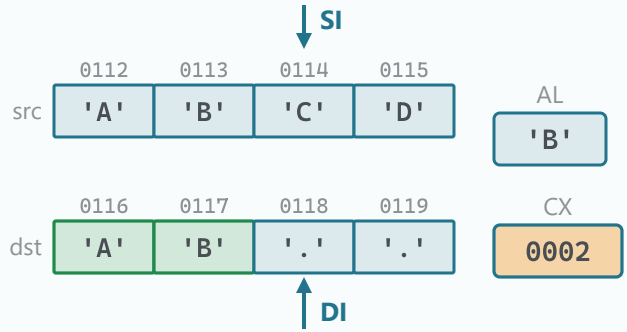
VARIABLES

SI = **0113h** DI = **0117h** AL = **41h** CX = **0003h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB` : DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
▶  loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



Pass 2 (summary): 'B' copied, SI = 0114h, DI = 0118h, CX = 2.

Step 9 / 12

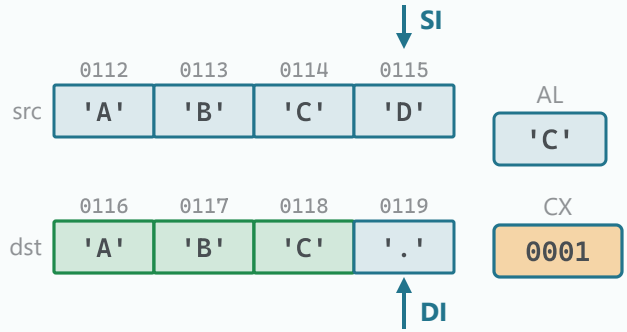
VARIABLES

SI = 0114h | DI = 0118h | AL = 42h | CX = 0002h

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
▶ loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



Pass 3 (summary): 'C' copied, CX = 1.

Step 10 / 12

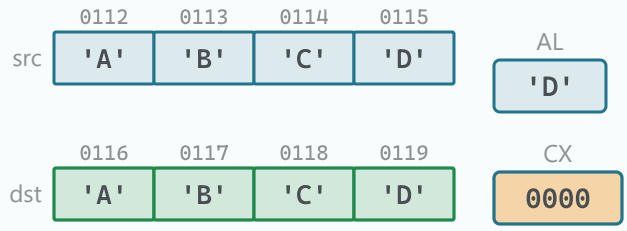
VARIABLES

SI = 0115h | DI = 0119h | AL = 43h | CX = 0001h

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
▶  loop copy
11 ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



Pass 4 (summary): 'D' copied, CX = **0** → LOOP falls through. The destination now holds **ABCD**.

Step 11 / 12

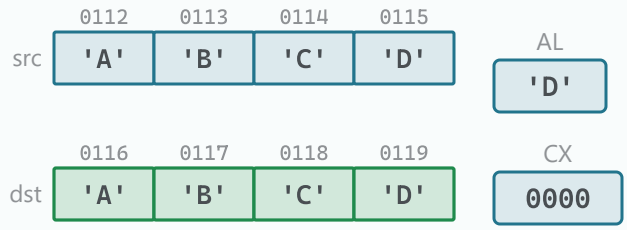
VARIABLES

SI = **0116h** DI = **011Ah** AL = **44h** CX = **0000h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB`: DS:SI → ES:DI).

Code Trace: Block Copy Through a Register

```
1  org 100h
2  mov si, offset src
3  mov di, offset dst
4  mov cx, 4
5  copy:
6  mov al, [si]          ; memory → AL
7  mov [di], al          ; AL → memory
8  inc si
9  inc di
10 loop copy
▶  ret
12 src db 'ABCD'
13 dst db 4 dup('.')
```



`ret` : SI and DI point just past the buffers (0116h, 011Ah).

Step 12 / 12

VARIABLES

SI = **0116h** DI = **011Ah** AL = **44h** CX = **0000h**

Each byte costs two memory accesses and a register. String instructions do the same job in one instruction (`REP MOVSB` : DS:SI → ES:DI).

Example Program

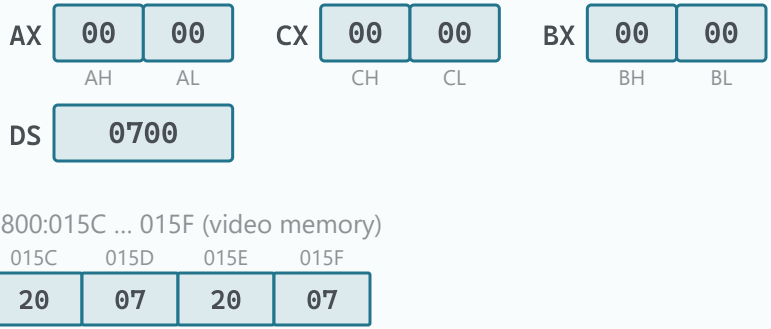
```
ORG 100h            ; for a .com program
MOV AX, 0B800h      ; AX = B800h
MOV DS, AX          ; DS = B800h
MOV CL, 'A'         ; CL = 41h
MOV CH, 11011111b   ; CH = DFh
MOV BX, 15Eh        ; BX = 015Eh
MOV [BX], CX        ; B800:015E ← CX
RET                 ; back to DOS
```

- The program writes the letter 'A' at a given position in **video memory**.
- `MOV` copies a value from source to destination.
- `ORG` sets the program's starting address (it produces no machine code).
- `;` starts a comment.
- DS cannot take an immediate, so B800h goes into AX first.

Instruction	Machine code	Instruction	Machine code
`MOV AX, 0B800h`	`B8 00 B8`	`MOV CH, 11011111b`	`B5 DF`
`MOV DS, AX`	`8E D8`	`MOV BX, 15Eh`	`BB 5E 01`
`MOV CL, 'A'`	`B1 41`	`MOV [BX], CX` · `RET`	`89 0F` · `C3`

Code Trace: Writing 'A' on the Screen

```
▶ ORG 100h
2 MOV AX, 0B800h
3 MOV DS, AX
4 MOV CL, 'A'
5 MOV CH, 11011111b
6 MOV BX, 15Eh
7 MOV [BX], CX
8 RET
```



The screen is blank: each cell is **20h** (space) + **07h** (gray on black).

Step 1 / 8

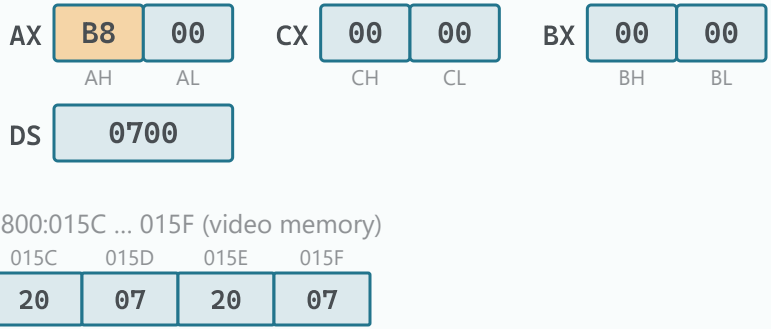
VARIABLES

`AX = 0000h` `DS = 0700h` `CH = 00h` `CL = 00h` `BX = 0000h`

`[B800:015E] = 20h` `[B800:015F] = 07h`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
▶ MOV AX, 0B800h
3 MOV DS, AX
4 MOV CL, 'A'
5 MOV CH, 11011111b
6 MOV BX, 15Eh
7 MOV [BX], CX
8 RET
```



AX = **B800h** (the segment of the color text screen).

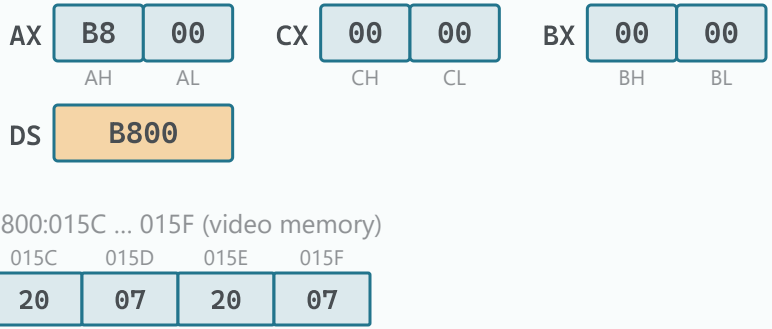
Step 2 / 8

VARIABLES

`AX = B800h` `DS = 0700h` `CH = 00h` `CL = 00h` `BX = 0000h` `[B800:015E] = 20h` `[B800:015F] = 07h`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
2 MOV AX, 0B800h
▶ MOV DS, AX
4 MOV CL, 'A'
5 MOV CH, 11011111b
6 MOV BX, 15Eh
7 MOV [BX], CX
8 RET
```



DS = AX = **B800h**. From now on [ ] accesses go to video memory.

Step 3 / 8

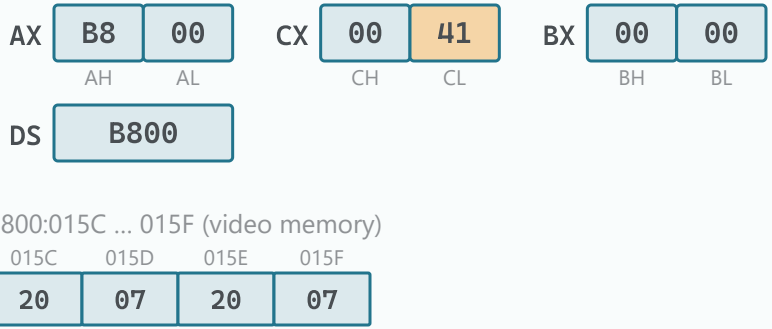
VARIABLES

`AX = B800h` `DS = B800h` `CH = 00h` `CL = 00h` `BX = 0000h`

`[B800:015E] = 20h` `[B800:015F] = 07h`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV DS, AX
▶ MOV CL, 'A'
5 MOV CH, 11011111b
6 MOV BX, 15Eh
7 MOV [BX], CX
8 RET
```



CL = 'A' = **41h** (its ASCII code).

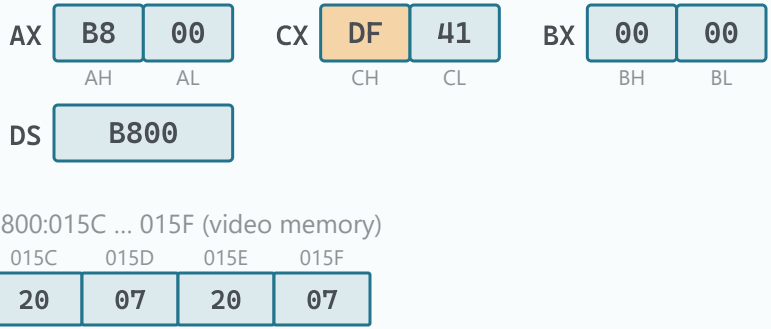
Step 4 / 8

VARIABLES

`AX = B800h` `DS = B800h` `CH = 00h` `CL = 41h` `BX = 0000h` `[B800:015E] = 20h` `[B800:015F] = 07h`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV DS, AX
4 MOV CL, 'A'
▶ MOV CH, 11011111b
6 MOV BX, 15Eh
7 MOV [BX], CX
8 RET
```



CH = 11011111b = **DFh**: the color (attribute) byte.

Step 5 / 8

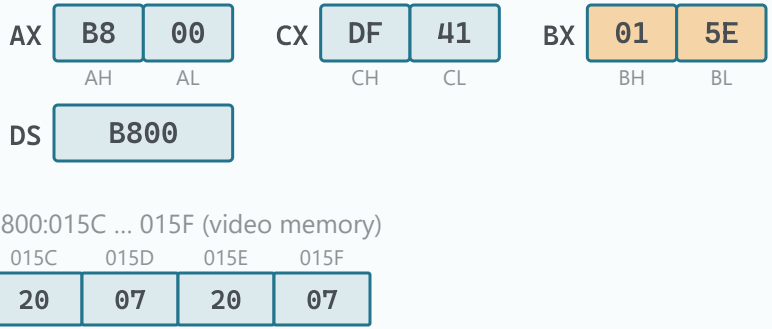
VARIABLES

`AX = B800h` `DS = B800h` `CH = DFh` `CL = 41h` `BX = 0000h`

`[B800:015E] = 20h` `[B800:015F] = 07h`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV DS, AX
4 MOV CL, 'A'
5 MOV CH, 11011111b
▶ MOV BX, 15Eh
7 MOV [BX], CX
8 RET
```



BX = **015Eh** = 350: the offset of the screen position.

Step 6 / 8

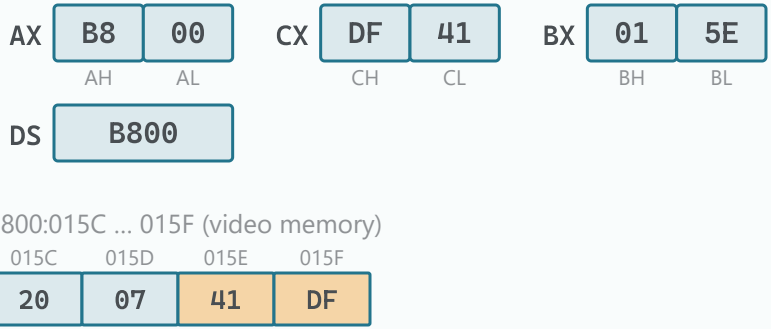
VARIABLES

`AX = B800h` `DS = B800h` `CH = DFh` `CL = 41h` `BX = 015Eh`

`[B800:015E] = 20h` `[B800:015F] = 07h`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV DS, AX
4 MOV CL, 'A'
5 MOV CH, 11011111b
6 MOV BX, 15Eh
▶ MOV [BX], CX
8 RET
```



CX = DF41h is written as a word: [015Eh] = **41h** ('A'), [015Fh] = **DFh** (color). Physical: B8000h + 15Eh = **B815Eh**.

Step 7 / 8

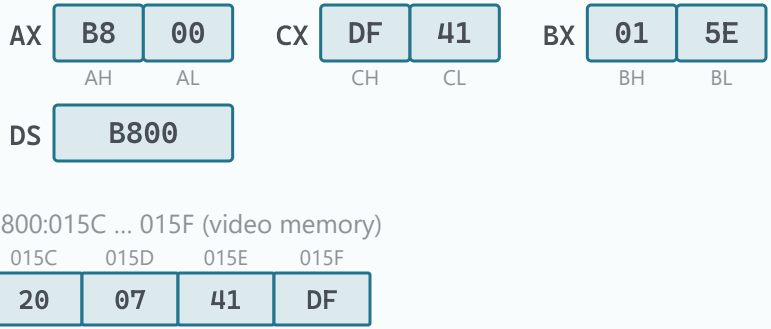
VARIABLES

`AX = B800h` `DS = B800h` `CH = DFh` `CL = 41h` `BX = 015Eh`

`[B800:015E] = 41h` `[B800:015F] = DFh`

Code Trace: Writing 'A' on the Screen

```
1 ORG 100h
2 MOV AX, 0B800h
3 MOV DS, AX
4 MOV CL, 'A'
5 MOV CH, 11011111b
6 MOV BX, 15Eh
7 MOV [BX], CX
▶ RET
```



RET: back to DOS; the 'A' stays on the screen.

Step 8 / 8

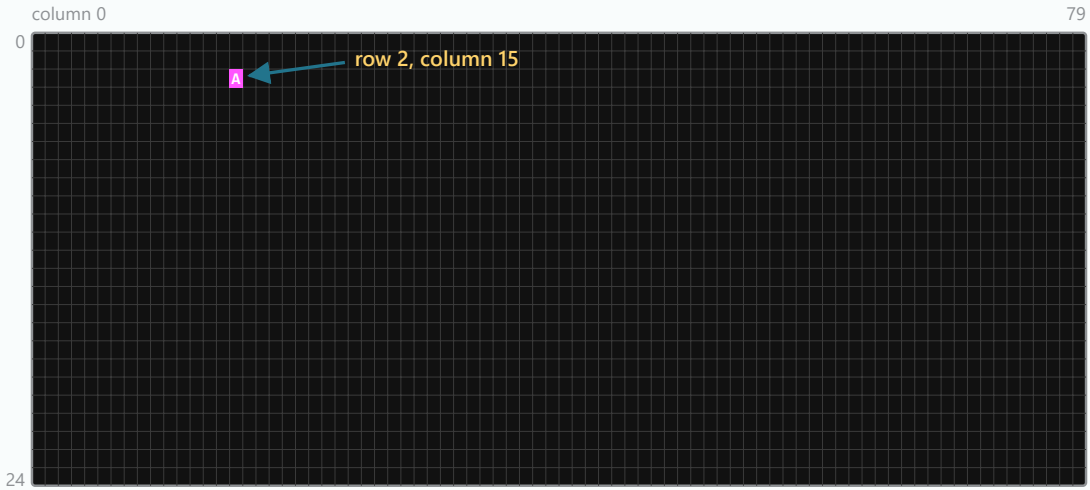
VARIABLES

`AX = B800h` `DS = B800h` `CH = DFh` `CL = 41h` `BX = 015Eh`

`[B800:015E] = 41h` `[B800:015F] = DFh`

EXAMPLE PROGRAM

Text-Mode Video Memory



Position

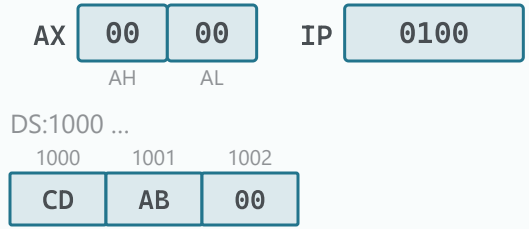
On the 80×25 text screen each cell is **2 bytes**: character + color. Offset = (row × 80 + column) × 2 → (2 × 80 + 15) × 2 = 350 = **15Eh**.

Color byte DFh = 1101 1111b

Bit 7: blink (bright background in emu8086) · bits 6–4 = 101: magenta background · bits 3–0 = 1111: bright white text.

Moving a Value from Memory into a Register

```
1 ORG 100h
2 MOV AX, [1000h]   ; A1 00 10
3 RET
```



DS = 0700h. DS:1000h holds the bytes **CD AB**. IP = 0100h.

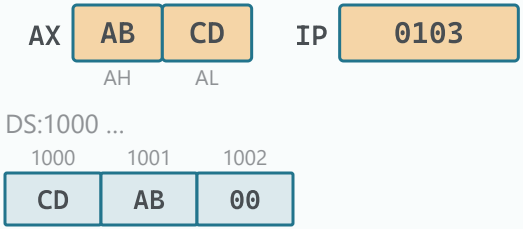
Step 1 / 3

VARIABLES

`AX = 0000h` `IP = 0100h` `w[1000h] = ABCDh`

Moving a Value from Memory into a Register

```
1 ORG 100h
▶ MOV AX, [1000h]   ; A1 00 10
3 RET
```



`MOV AX,[1000h]` : two bytes are read, little-endian → AX = **ABCDh**. The instruction is 3 bytes → IP = 0103h.

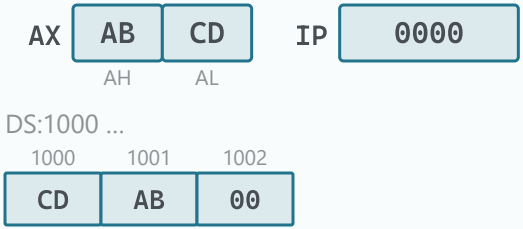
Step 2 / 3

VARIABLES

`AX = ABCDh` `IP = 0103h` `w[1000h] = ABCDh`

Moving a Value from Memory into a Register

```
1 ORG 100h
2 MOV AX, [1000h]   ; A1 00 10
▶ RET
```



RET: IP = 0000h (the INT 20h in the PSP). Memory is unchanged; MOV only **copies**. Physical address: 07000h + 1000h = 08000h.

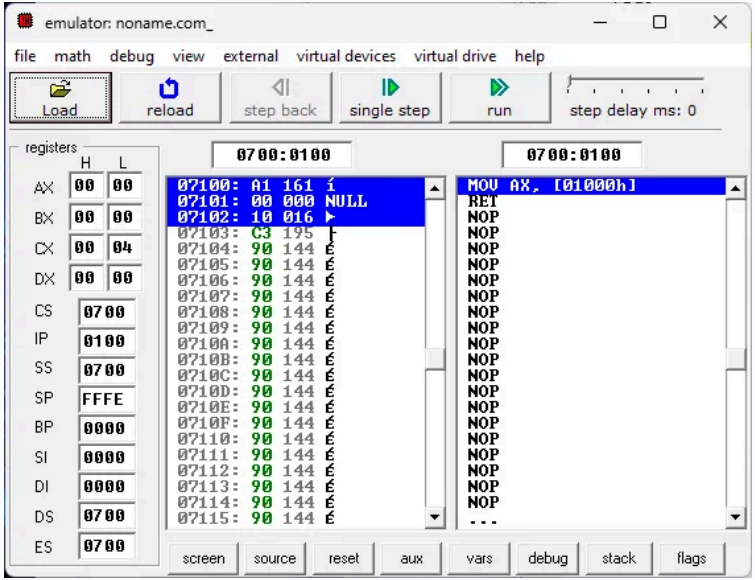
Step 3 / 3

VARIABLES

`AX = ABCDh` `IP = 0000h` `w[1000h] = ABCDh`

The View in emu8086

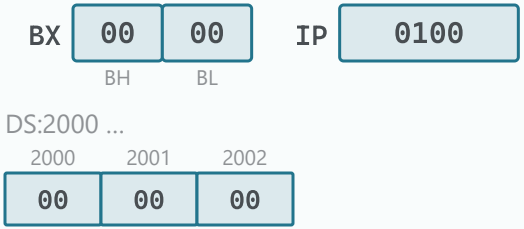
- Left: registers (H/L halves shown separately). `CS = DS = 0700h`, `IP = 0100h`, `SP = FFFEh`.
- Middle: memory (physical address `07100h` = 0700:0100). `A1 00 10` are the 3 bytes of the instruction.
- Right: the disassembled code: `MOV AX, [01000h]`, then `RET` (C3).
- **single step** executes the instruction; changed registers are highlighted.
- Memory is empty here, so AX stays 0000h on this screen.



The emu8086 emulator window

Moving a Register Value into Memory

```
1 ORG 100h
2 MOV BX, 1234h      ; BB 34 12
3 MOV [2000h], BX    ; 89 1E 00 20
4 RET
```



DS:2000h is empty (00 00).

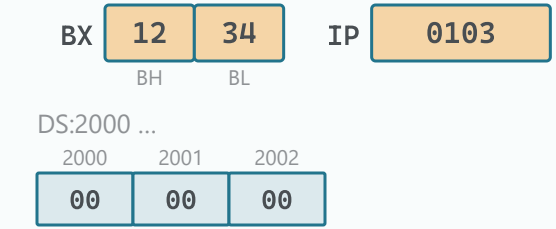
Step 1 / 4

VARIABLES

`BX = 0000h` `IP = 0100h` `w[2000h] = 0000h`

Moving a Register Value into Memory

```
1 ORG 100h
▶ MOV BX, 1234h      ; BB 34 12
3 MOV [2000h], BX    ; 89 1E 00 20
4 RET
```



BX = **1234h**; IP = 0103h.

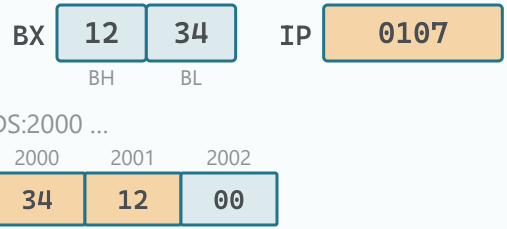
Step 2 / 4

VARIABLES

`BX = 1234h` `IP = 0103h` `w[2000h] = 0000h`

Moving a Register Value into Memory

```
1 ORG 100h
2 MOV BX, 1234h      ; BB 34 12
▶ MOV [2000h], BX    ; 89 1E 00 20
4 RET
```



BX is written to memory: [2000h] = **34h**, [2001h] = **12h**. The instruction is 4 bytes → IP = 0107h.

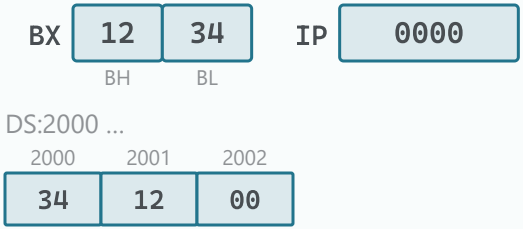
Step 3 / 4

VARIABLES

`BX = 1234h` `IP = 0107h` `w[2000h] = 1234h`

Moving a Register Value into Memory

```
1 ORG 100h
2 MOV BX, 1234h      ; BB 34 12
3 MOV [2000h], BX    ; 89 1E 00 20
▶ RET
```



RET: IP = 0000h. BX is unchanged. Physical address: 07000h + 2000h = **09000h**.

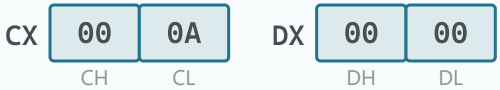
Step 4 / 4

VARIABLES

`BX = 1234h` `IP = 0000h` `w[2000h] = 1234h`

Swapping Two Registers

```
▶ MOV CX, 10   ; CX = 000Ah
2 MOV DX, 20   ; DX = 0014h
3 XCHG CX, DX  ; swap
```



CX = 10 = **000Ah** (numbers are decimal by default).

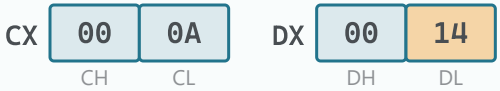
Step 1 / 3

VARIABLES

`CX = 000Ah` `DX = 0000h`

Swapping Two Registers

```
1 MOV CX, 10   ; CX = 000Ah
▶ MOV DX, 20   ; DX = 0014h
3 XCHG CX, DX  ; swap
```



DX = 20 = **0014h**.

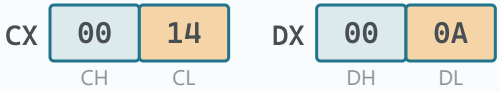
Step 2 / 3

VARIABLES

CX = 000Ah DX = 0014h

Swapping Two Registers

```
1 MOV CX, 10   ; CX = 000Ah
2 MOV DX, 20   ; DX = 0014h
▶ XCHG CX, DX  ; swap
```



`XCHG` swaps the two operands **in one instruction**: CX = 0014h, DX = 000Ah. Flags are not affected.

Step 3 / 3

VARIABLES

`CX = 0014h` `DX = 000Ah`

XCHG Rules

XCHG dest, source

- Register ↔ register: `XCHG AX, BX`
- Register ↔ memory: `XCHG AL, [SI]`
- **No** memory ↔ memory.
- No segment registers or immediates.
- Sizes must match; flags are unchanged.
- `XCHG AX, reg` is a one-byte instruction (`XCHG AX, AX` = `NOP` = 90h).

Without XCHG a swap would need a **temporary** register:

```
MOV AX, CX     ; AX ← CX (backup)
MOV CX, DX     ; CX ← DX
MOV DX, AX     ; DX ← old CX
```

That costs 3 instructions and an extra register; XCHG does it in one.

Code Trace: Swapping with XCHG

```
1  ORG 100h
▶ MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX          ; backup
8  MOV CX, DX
9  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```



SI = 0116h: the address of `num`; in memory num = **A5h**.

Step 1 / 9

VARIABLES

`SI = 0116h` `AX = 0000h` `CX = 0000h` `DX = 0000h` `num = A5h`

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
▶  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX          ; backup
8  MOV CX, DX
9  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```

AX	CX	DX	num · 0116h
005Ah	0000h	0000h	A5h

AL = **5Ah**.

Step 2 / 9

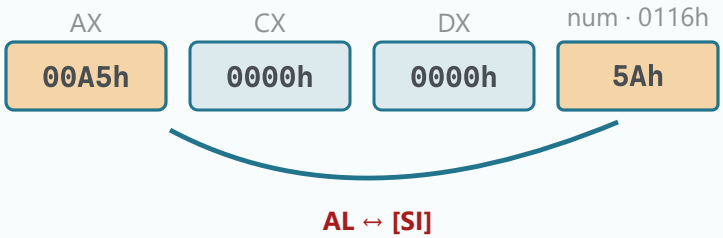
VARIABLES

SI = **0116h** · AX = **005Ah** · CX = **0000h** · DX = **0000h** · num = **A5h**

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
▶  XCHG AL, [SI]        ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX           ; backup
8  MOV CX, DX
9  MOV DX, AX           ; swapped
10 XCHG CX, DX          ; one instr.
11 RET
12 num DB 0A5h
```



`XCHG AL, [SI]` : AL ↔ memory. AL = **A5h**, num = **5Ah**. No temporary register needed.

Step 3 / 9

VARIABLES

`SI = 0116h` `AX = 00A5h` `CX = 0000h` `DX = 0000h` `num = 5Ah`

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
▶  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX          ; backup
8  MOV CX, DX
9  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```

AX	CX	DX	num · 0116h
00A5h	1111h	0000h	5Ah

CX = 1111h.

Step 4 / 9

VARIABLES

SI = 0116h · AX = 00A5h · CX = 1111h · DX = 0000h · num = 5Ah

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
▶  MOV DX, 2222h
7  MOV AX, CX          ; backup
8  MOV CX, DX
9  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```

AX	CX	DX	num · 0116h
00A5h	1111h	2222h	5Ah

DX = 2222h. Now the same swap without XCHG.

Step 5 / 9

VARIABLES

SI = 0116h · AX = 00A5h · CX = 1111h · DX = 2222h · num = 5Ah

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
▶  MOV AX, CX          ; backup
8  MOV CX, DX
9  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```

AX	CX	DX	num · 0116h
1111h	1111h	2222h	5Ah

1. AX ← CX

3 instr. + spare register

`MOV AX, CX` : a backup of CX in AX (AX's old value is lost).

Step 6 / 9

VARIABLES

`SI = 0116h` `AX = 1111h` `CX = 1111h` `DX = 2222h` `num = 5Ah`

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX          ; backup
▶  MOV CX, DX
9  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```

AX	CX	DX	num · 0116h
1111h	2222h	2222h	5Ah

2. CX ← DX

3 instr. + spare register

`MOV CX, DX` : CX = 2222h. The old CX survives only in AX.

Step 7 / 9

VARIABLES

`SI = 0116h` `AX = 1111h` `CX = 2222h` `DX = 2222h` `num = 5Ah`

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX          ; backup
8  MOV CX, DX
▶  MOV DX, AX          ; swapped
10 XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```

AX	CX	DX	num · 0116h
1111h	2222h	1111h	5Ah

3. DX ← AX

3 instr. + spare register

`MOV DX, AX` : DX = 1111h. The swap took 3 instructions and a register.

Step 8 / 9

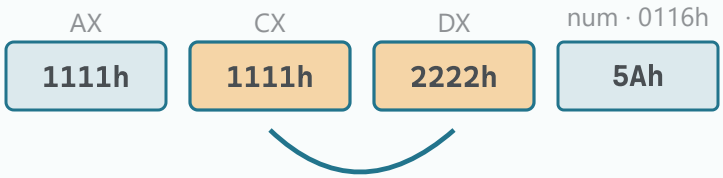
VARIABLES

`SI = 0116h` `AX = 1111h` `CX = 2222h` `DX = 1111h` `num = 5Ah`

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Code Trace: Swapping with XCHG

```
1  ORG 100h
2  MOV SI, OFFSET num
3  MOV AL, 5Ah
4  XCHG AL, [SI]       ; AL ⇔ [SI]
5  MOV CX, 1111h
6  MOV DX, 2222h
7  MOV AX, CX          ; backup
8  MOV CX, DX
9  MOV DX, AX          ; swapped
▶  XCHG CX, DX         ; one instr.
11 RET
12 num DB 0A5h
```



CX ↔ DX: 1 instruction

`XCHG CX, DX`: swapped back in one instruction: CX = **1111h**, DX = **2222h**; AX is untouched.

Step 9 / 9

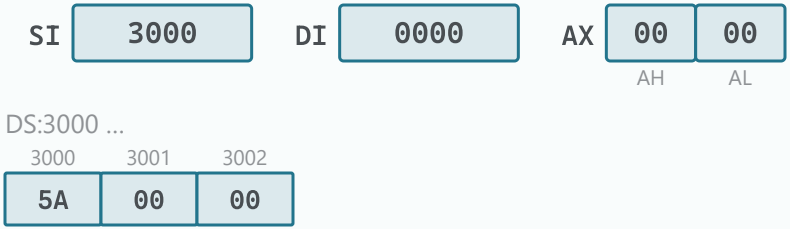
VARIABLES

`SI = 0116h` `AX = 1111h` `CX = 1111h` `DX = 2222h` `num = 5Ah`

XCHG leaves the flags alone; there is no memory ↔ memory or segment-register exchange.

Copying One Register into Another

```
1 MOV SI, 3000h  ; SI = address
2 MOV DI, SI     ; DI = SI
3 MOV AL, [SI]   ; DS:3000h
4 MOV [DI+1], AL ; DS:3001h
```



SI gets a **memory address** (an offset): 3000h.

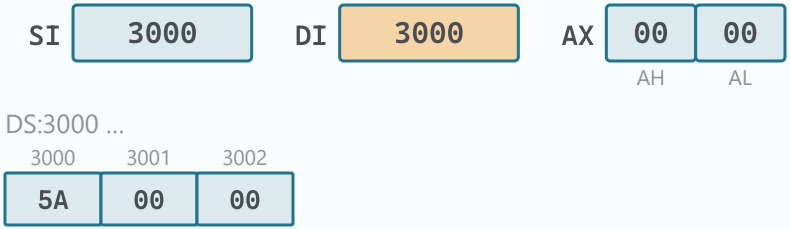
Step 1 / 4

VARIABLES

SI = 3000h DI = 0000h AL = 00h

Copying One Register into Another

```
1 MOV SI, 3000h  ; SI = address
▶ MOV DI, SI     ; DI = SI
3 MOV AL, [SI]   ; DS:3000h
4 MOV [DI+1], AL ; DS:3001h
```



DI = SI = **3000h**. What is copied is the address, not the data in memory.

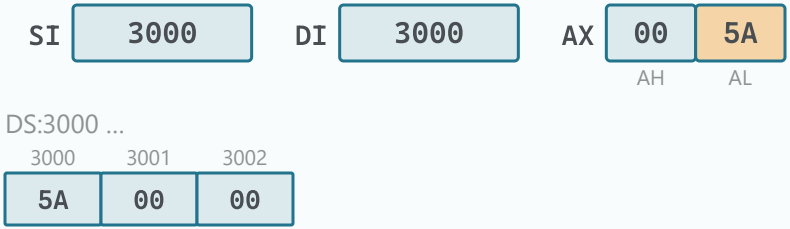
Step 2 / 4

VARIABLES

SI = 3000h DI = 3000h AL = 00h

Copying One Register into Another

```
1 MOV SI, 3000h  ; SI = address
2 MOV DI, SI     ; DI = SI
▶ MOV AL, [SI]   ; DS:3000h
4 MOV [DI+1], AL ; DS:3001h
```



SI and DI now point to the same cell: AL = [3000h] = **5Ah**.

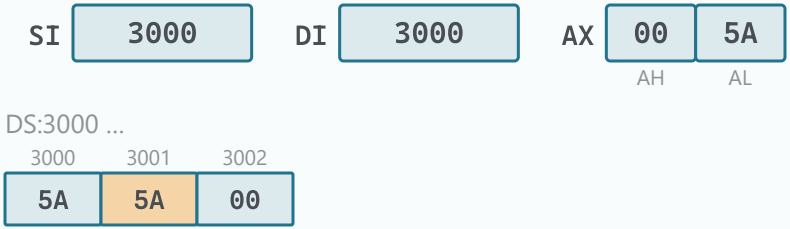
Step 3 / 4

VARIABLES

SI = 3000h DI = 3000h AL = 5Ah

Copying One Register into Another

```
1 MOV SI, 3000h  ; SI = address
2 MOV DI, SI     ; DI = SI
3 MOV AL, [SI]   ; DS:3000h
▶ MOV [DI+1], AL ; DS:3001h
```



AL is written to [DI+1] = 3001h → **5A 5A**.

Step 4 / 4

VARIABLES

SI = **3000h** DI = **3000h** AL = **5Ah**

Writing a Constant to a Memory Address

```
1 ORG 100h
2 MOV BYTE PTR [4000h], 65  ; 'A'
3 MOV WORD PTR [4002h], 65
4 RET
```

IP 0100

DS:4000 ...

4000	4001	4002	4003
00	99	00	99

DS:4000h ... 4003h: 00 99 00 99.

Step 1 / 4

VARIABLES

IP = 0100h [4000h] = 00h [4003h] = 99h

Writing a Constant to a Memory Address

```
1 ORG 100h
▶ MOV BYTE PTR [4000h], 65  ; 'A'
3 MOV WORD PTR [4002h], 65
4 RET
```

IP **0105**

DS:4000 …

4000	4001	4002	4003
41	99	00	99

65 = 41h = 'A' is written as a single byte. Machine code **C6 06 00 40 41** (5 bytes) → IP = 0105h. [4001h] is unchanged.

Step 2 / 4

VARIABLES

`IP = 0105h` `[4000h] = 41h` `[4003h] = 99h`

Writing a Constant to a Memory Address

```
1 ORG 100h
2 MOV BYTE PTR [4000h], 65  ; 'A'
▶ MOV WORD PTR [4002h], 65
4 RET
```

IP **010B**

DS:4000 …

4000	4001	4002	4003
41	99	41	00

WORD PTR: 0041h is written → [4002h] = 41h, [4003h] = **00h** (the 99h is overwritten). The instruction is 6 bytes → IP = 010Bh.

Step 3 / 4

VARIABLES

IP = **010Bh** [4000h] = **41h** [4003h] = **00h**

Writing a Constant to a Memory Address

```
1 ORG 100h
2 MOV BYTE PTR [4000h], 65  ; 'A'
3 MOV WORD PTR [4002h], 65
▶ RET
```

IP 0000

DS:4000 ...

4000	4001	4002	4003
41	99	41	00

RET: IP = 0000h. A constant has no size of its own; the **PTR** specifier sets it.

Step 4 / 4

VARIABLES

IP = 0000h [4000h] = 41h [4003h] = 00h

Copying a Register's Low Byte into Another

```
1 MOV AX, 1234h   ; AH = 12h, AL = 34h
2 MOV BX, 0FFFFh
3 MOV BL, AL      ; BL only
```



AX = **1234h**: AH = 12h, AL = 34h.

Step 1 / 3

VARIABLES

AX = **1234h** BX = **0000h** BL = **00h**

Copying a Register's Low Byte into Another

```
1 MOV AX, 1234h    ; AH = 12h, AL = 34h
▶ MOV BX, 0FFFFh
3 MOV BL, AL       ; BL only
```



BX = FFFFh, to make the effect visible.

Step 2 / 3

VARIABLES

AX = **1234h** BX = **FFFFh** BL = **FFh**

Copying a Register's Low Byte into Another

```
1 MOV AX, 1234h    ; AH = 12h, AL = 34h
2 MOV BX, 0FFFFh
▶ MOV BL, AL       ; BL only
```



AL (the **low byte** of AX) is copied to BL: BL = 34h. BH is unchanged → BX = **FF34h**.

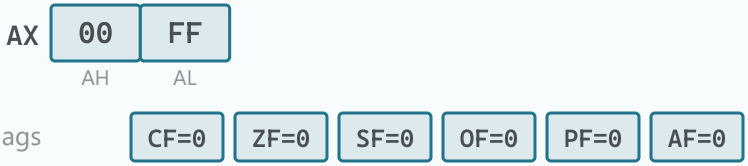
Step 3 / 3

VARIABLES

AX = **1234h** BX = **FF34h** BL = **34h**

Adding to a Register's High Byte

```
▶ MOV AX, 00FFh    ; AH = 00h, AL = FFh
2 ADD AH, 01h      ; AH only
3 ADD AL, 01h      ; AL only
4 MOV AX, 01FFh
5 ADD AX, 1        ; 16 bits
```



AX = **00FFh**.

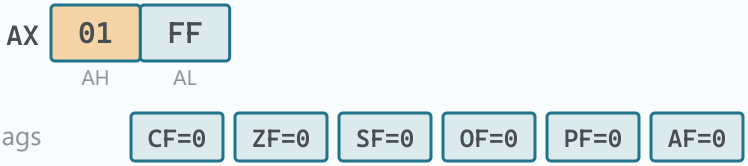
Step 1 / 5

VARIABLES

AX = **00FFh** AH = **00h** AL = **FFh** CF = **0** ZF = **0** SF = **0** OF = **0** PF = **0** AF = **0**

Adding to a Register's High Byte

```
1 MOV AX, 00FFh    ; AH = 00h, AL = FFh
▶ ADD AH, 01h      ; AH only
3 ADD AL, 01h      ; AL only
4 MOV AX, 01FFh
5 ADD AX, 1        ; 16 bits
```



AH = 00h + 01h = 01h → AX = **01FFh**. Nonzero result, no carry.

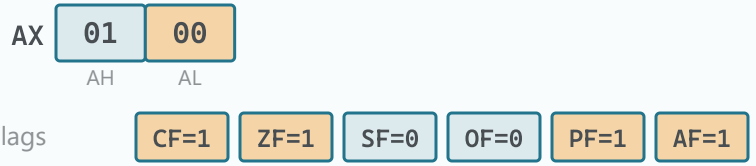
Step 2 / 5

VARIABLES

AX = **01FFh** | AH = **01h** | AL = **FFh** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0** | AF = **0**

Adding to a Register's High Byte

```
1 MOV AX, 00FFh    ; AH = 00h, AL = FFh
2 ADD AH, 01h      ; AH only
▶ ADD AL, 01h      ; AL only
4 MOV AX, 01FFh
5 ADD AX, 1        ; 16 bits
```



AL = FFh + 1 = 00h, **CF = 1**, ZF = 1. The carry does **not** reach AH: AX = **0100h**.

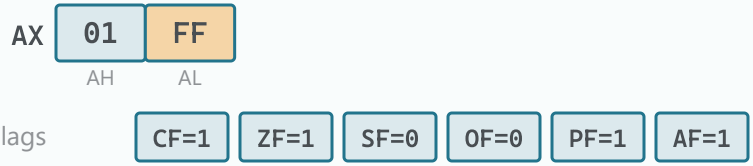
Step 3 / 5

VARIABLES

AX = 0100h | AH = 01h | AL = 00h | CF = 1 | ZF = 1 | SF = 0 | OF = 0 | PF = 1 | AF = 1

Adding to a Register's High Byte

```
1 MOV AX, 00FFh    ; AH = 00h, AL = FFh
2 ADD AH, 01h      ; AH only
3 ADD AL, 01h      ; AL only
▶ MOV AX, 01FFh
5 ADD AX, 1        ; 16 bits
```



For comparison, AX is 01FFh again. MOV leaves the flags alone.

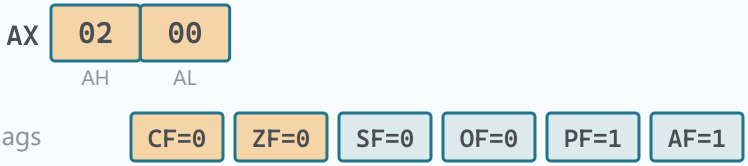
Step 4 / 5

VARIABLES

AX = 01FFh | AH = 01h | AL = FFh | CF = 1 | ZF = 1 | SF = 0 | OF = 0 | PF = 1 | AF = 1

Adding to a Register's High Byte

```
1 MOV AX, 00FFh    ; AH = 00h, AL =
  FFh
2 ADD AH, 01h      ; AH only
3 ADD AL, 01h      ; AL only
4 MOV AX, 01FFh
▶ ADD AX, 1        ; 16 bits
```



16-bit addition: 01FFh + 1 = **0200h**; the carry passed from AL into AH, CF = 0.

Step 5 / 5

VARIABLES

AX = 0200h AH = 02h AL = 00h CF = 0 ZF = 0

SF = 0 OF = 0 PF = 1 AF = 1

Loading Binary, Octal and Hex Values

```
1 mov al, 00000101b ; binary
2 mov bl, 0ah       ; hex
3 mov cl, 10o       ; octal
4 add al, bl        ; 5 + 10 = 15
5 sub al, cl        ; 15 - 8 = 7
```



00000101b = **5** → AL = 05h.

Step 1 / 5

VARIABLES

AL = **05h** BL = **00h** CL = **00h** CF = **0** ZF = **0**

SF = **0** OF = **0** PF = **0** AF = **0**

Loading Binary, Octal and Hex Values

```
1 mov al, 00000101b ; binary
▶ mov bl, 0ah       ; hex
3 mov cl, 10o       ; octal
4 add al, bl        ; 5 + 10 = 15
5 sub al, cl        ; 15 - 8 = 7
```



0Ah = **10** → BL = 0Ah. It starts with a letter, hence the leading 0.

Step 2 / 5

VARIABLES

AL = **05h** | BL = **0Ah** | CL = **00h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0** | AF = **0**

Loading Binary, Octal and Hex Values

```
1 mov al, 00000101b ; binary
2 mov bl, 0ah       ; hex
▶ mov cl, 10o       ; octal
4 add al, bl        ; 5 + 10 = 15
5 sub al, cl        ; 15 - 8 = 7
```



10o (octal) = 1×8 + 0 = **8** → CL = 08h.

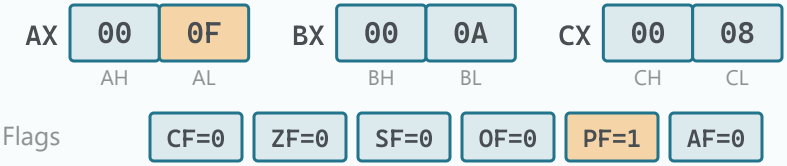
Step 3 / 5

VARIABLES

AL = **05h** | BL = **0Ah** | CL = **08h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0** | AF = **0**

Loading Binary, Octal and Hex Values

```
1 mov al, 00000101b ; binary
2 mov bl, 0ah       ; hex
3 mov cl, 10o       ; octal
▶ add al, bl        ; 5 + 10 = 15
5 sub al, cl        ; 15 - 8 = 7
```



AL = 5 + 10 = 15 = **0Fh**. PF = 1 (0000 1111: four 1 bits).

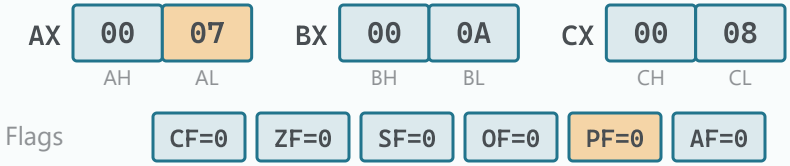
Step 4 / 5

VARIABLES

AL = 0Fh | BL = 0Ah | CL = 08h | CF = 0 | ZF = 0 | SF = 0 | OF = 0 | PF = 1 | AF = 0

Loading Binary, Octal and Hex Values

```
1 mov al, 00000101b ; binary
2 mov bl, 0ah       ; hex
3 mov cl, 10o       ; octal
4 add al, bl        ; 5 + 10 = 15
▶ sub al, cl        ; 15 - 8 = 7
```



AL = 15 − 8 = 7 = **07h**. No borrow → CF = 0; PF = 0 (three 1 bits).

Step 5 / 5

VARIABLES

AL = 07h | BL = 0Ah | CL = 08h | CF = 0 | ZF = 0 | SF = 0 | OF = 0 | PF = 0 | AF = 0

MOV EXAMPLES

Writing Numbers

Base	Suffix	Example	Decimal value
Decimal	none or `d`	`25`, `25d`	25
Binary	`b`	`00011001b`	25
Octal	`o` or `q`	`31o`	25
Hexadecimal	`h`	`19h`, `0FFh`	25, 255
Character	`' '`	`'A'`	65 (41h)

A hex number that starts with a letter (A–F) needs a leading **0**: `0Ah`, `0B800h`. Otherwise `Ah` clashes with the AH register and `B800h` with a label. emu8086 also accepts the `0x1000` form.

MOV EXAMPLES

Common Memory-Access Mistakes

Wrong	What happens?	Correct
`MOV SI, arr`	SI gets the value of the first element, not the array's address.	`MOV SI, OFFSET arr` or `LEA SI, arr`
`INC SI` (word array)	Reads from the middle of a word: halves of two elements are combined.	`ADD SI, 2`
`MOV AL, [BP]` for a variable	Reads from the SS segment; in an .EXE the wrong byte comes back.	`[BX]` , `[SI]` or `DS:[BP]`
`MOV [BX], 5`	Size unknown; the assembler reports an error or guesses.	`MOV BYTE PTR [BX], 5`
`MOV AX, B800h`	B800h is taken for a label name.	`MOV AX, 0B800h`
Changing DS and leaving it	Every later variable access goes to another segment.	`PUSH DS` … `POP DS` or ES
Expecting 3412h from `MOV AX, [0200h]`	Little-endian forgotten: if memory holds **34 12**, AX = 1234h.	Low address = low byte

Most of these are not assembly errors: the program runs but reads from the wrong address. In emu8086, check the EA and the physical address in the memory window to find them.

Lab: Memory Access

Addressing modes

CODE · **28 bytes**

```
 1 ORG 100h
 2 MOV BX, OFFSET table
 3 MOV SI, 2
 4 MOV AH, 7           ; immed…
 5 MOV AL, [table+1]   ; direct
 6 MOV CL, [BX]        ; indir…
 7 MOV CH, [BX+4]      ; based
 8 MOV DL, [BX+SI]     ; base+…
 9 MOV DH, [BX+SI+3]   ; + d8
10 RET
11 table DB 11h,22h,33h,44h,5…
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	001C	CH	00	CL	1C
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**

```
0110 8A 10 8A 70 03 C3 11 22
0118 33 44 55 66 00 00 00 00
0120 00 00 00 00 00 00 00 00
0128 00 00 00 00 00 00 00 00
0130 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: 28 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (001Ch). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

Addressing
[base + index + displacement]; base BX/BP, index SI/DI, displacement d8/d16, each optional.

Segment choice
SS if BP is used, otherwise DS; a prefix overrides it (ES:[BX]).

Physical address
segment × 10h + EA; 1234:7890 → 19BD0h.

Size
byte ptr / word ptr (b. / w.); a register operand sets the size. Little-endian.

MOV
Copies and leaves the flags alone; no memory↔memory, no immediate into a segment register, no CS/IP destination.

XCHG
Swaps two operands in one instruction; no segment registers or immediates.

SUMMARY

Exercises

1. DS = 3000h, BX = 0010h, SI = 0005h: EA and physical address of `[BX+SI+20h]`?
2. SS = 4000h, BP = FFFEh: EA and physical address of `[BP+4]`?
3. Which of `[BX+BP]`, `[DI+SI]`, `[BX+DI+300]`, `[DX]` are valid?
4. If [0500h] = 78h and [0501h] = 56h, what is AX after `MOV AX,[0500h]`? AL after `MOV AL,[0501h]`?
5. Write code that puts a yellow 'X' on green at row 10, column 40 of the screen (color 2Eh).
6. How would you change the previous program to write to video memory through ES without touching DS?

▶ **Answers**

References

- emu8086 documentation, *8086 assembler tutorial for beginners*, part 2: memory access, MOV instruction.
- Intel, *8086/8088 User's Manual*, addressing modes and instruction encoding.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, Chapter 3 (Addressing Modes).
- IBM PC documentation on text-mode video memory (B800h) and the attribute byte.

The End

Chapter 4.1: Memory Access

Previous: Chapter 3.2: Assembly Language · Next: Chapter 4.2: Variables