Chapter 4.2: Variables

Microprocessors

DB / DW / DD | Naming rules | Memory layout | ORG 100h and .COM | Arrays and DUP | LEA and OFFSET | EQU constants | Adding and comparing | Doublewords

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_02_Variables.html

Learning Outcomes

Definitions

Define variables and arrays with DB, DW, DD, DUP and ?.

Memory layout

Work out the offsets and byte order the assembler gives each variable.

Addresses

Get a variable’s address with LEA and OFFSET and index into arrays.

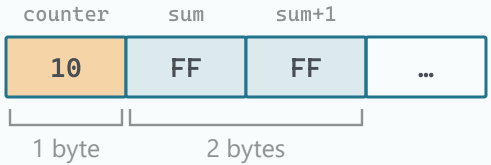
Programs

Trace programs that move, add and compare data between variables.

Variables

- A variable stands for a **memory location**.
- For a programmer the name `var1` is far easier to use than the address `5A73:235B`.
- The assembler supports two basic types:
  - **BYTE** (8 bits): `name DB value` (Define Byte)
  - **WORD** (16 bits): `name DW value` (Define Word)
- There is also `DD` (Define Doubleword, 32 bits): 4 bytes.

```
counter DB 10h      ; BYTE
sum     DW 0FFFFh   ; WORD
```



Names and Initial Values

- A variable name **must start with a letter**; letters and digits may follow.
- Register and instruction names (`AX` , `MOV` ...) cannot be variable names.
- A variable may have no name: it still has an **address**.
- It can hold any numeric value (binary, decimal, hex, character).
- Use `?` for a variable with no initial value (emu8086 stores 00).

Definition	?
`num1 DB 5`	valid
`1num DB 5`	starts with a digit
`ax DW 0`	register name
`DB 0Ah`	no name, but has an address
`result DW ?`	no initial value
`letter DB 'A'`	41h

Value Ranges

Directive	Size	Unsigned	Signed	Example
`DB`	1 byte	0 … 255	−128 … 127	`x DB -1` → FFh
`DW`	2 bytes	0 … 65535	−32768 … 32767	`y DW -2` → FE FF
`DD`	4 bytes	0 … 4,294,967,295	-2^{31} … $2^{31}-1$	`z DD 1` → 01 00 00 00

Memory stores only the bit pattern; **signed or unsigned** depends on the instruction that uses the value (JGE compares signed, JAE unsigned). `DB 300` does not fit in a byte → the assembler reports an error.

Code Trace: Is FFh 255 or −1?

```
1  ORG 100h
▶  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; signed: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```

memory

FF	FE	FF
x · 0110	y · 0111	y+1 · 0112

CF	OF	ZF
0	0	0

AX **00FFh** — unsigned 255, signed 255

BX **0000h** — unsigned 0, signed 0

AL: 255 / −1

In memory x = **FFh**, y = **FE FF** (FFFEh, little-endian). AL = x = FFh: 255 unsigned, −1 signed.

Step 1 / 6

VARIABLES

AX = 00FFh | BX = 0000h | CF = 0 | ZF = 0

DB/DW store only the bit pattern; the instruction that uses the value decides whether it is signed or unsigned.

Code Trace: Is FFh 255 or −1?

```
1  ORG 100h
2  MOV AL, x          ; FFh
▶  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; signed: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```

memory

FF	FE	FF
x · 0110	y · 0111	y+1 · 0112

CF	OF	ZF
1	0	1

unsigned overflow: CF

AX	0000h	unsigned 0 signed 0
BX	0000h	unsigned 0 signed 0

`AL: 0 / 0`

`ADD AL, 1` : unsigned, 255 + 1 = 256 does not fit in a byte → AL = **00h**, **CF = 1**. Signed, −1 + 1 = 0 is correct: OF = 0.

Step 2 / 6

VARIABLES

`AX = 0000h` `BX = 0000h` `CF = 1` `ZF = 1`

DB/DW store only the bit pattern; the instruction that uses the value decides whether it is signed or unsigned.

Code Trace: Is FFh 255 or −1?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
▶  MOV AL, x
5  CBW                ; signed: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```

memory

FF	FE	FF
x · 0110	y · 0111	y+1 · 0112

CF	OF	ZF
1	0	1

AX **00FFh** — unsigned 255, signed 255

BX **0000h** — unsigned 0, signed 0

AL: 255 / −1

AL is FFh again.

Step 3 / 6

VARIABLES

AX = 00FFh | BX = 0000h | CF = 1 | ZF = 1

DB/DW store only the bit pattern; the instruction that uses the value decides whether it is signed or unsigned.

Code Trace: Is FFh 255 or −1?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
▶  CBW                ; signed: -1
6  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```

memory

FF	FE	FF
x · 0110	y · 0111	y+1 · 0112

CF	OF	ZF
1	0	1

sign extension

AX	FFFFh	unsigned 65535 signed −1
BX	0000h	unsigned 0 signed 0

`AL: 255 / −1`

`CBW` treats AL as signed: MSB = 1 → AH = FFh, AX = **FFFFh** = −1 (16 bits).

Step 4 / 6

VARIABLES

AX = FFFFh | BX = 0000h | CF = 1 | ZF = 1

DB/DW store only the bit pattern; the instruction that uses the value decides whether it is signed or unsigned.

Code Trace: Is FFh 255 or −1?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; signed: -1
▶  MOV BX, y          ; FFFEh
7  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```

memory

FF	FE	FF
x · 0110	y · 0111	y+1 · 0112

CF	OF	ZF
1	0	1

AX **FFFFh** unsigned 65535, signed −1

BX **FFFEh** unsigned 65534, signed −2

`AL: 255 / −1`

BX = y = **FFFEh**: 65534 unsigned, −2 signed.

Step 5 / 6

VARIABLES

AX = **FFFFh** | BX = **FFFEh** | CF = **1** | ZF = **1**

DB/DW store only the bit pattern; the instruction that uses the value decides whether it is signed or unsigned.

Code Trace: Is FFh 255 or −1?

```
1  ORG 100h
2  MOV AL, x          ; FFh
3  ADD AL, 1          ; 255 + 1 ?
4  MOV AL, x
5  CBW                ; signed: -1
6  MOV BX, y          ; FFFEh
▶  NEG BX             ; -(-2)
8  RET
9  x DB -1
10 y DW -2
```



`NEG BX` : 0 − (−2) = **0002h**. The instruction chose the meaning: ADD/CF read the pattern as unsigned, CBW/NEG as signed.

Step 6 / 6

VARIABLES

AX = **FFFFh** BX = **0002h** CF = **1** ZF = **0**

DB/DW store only the bit pattern; the instruction that uses the value decides whether it is signed or unsigned.

Variable Type and Size Checking

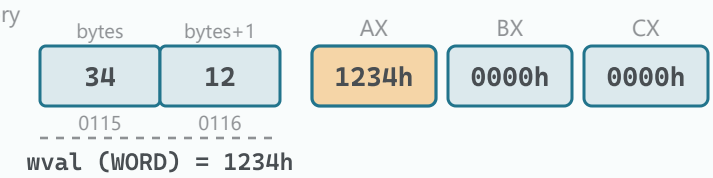
- The assembler keeps a **type** with every variable name: DB → BYTE (1), DW → WORD (2), DD → DWORD (4).
- If the other operand is a register, the sizes must match; otherwise assembly stops.
- With an immediate, the variable's type sets the size: `MOV w, 5` writes a word.
- Operands like `[BX]` have no type; with an immediate they need `BYTE PTR` or `WORD PTR`.
- `BYTE PTR` changes the type for that instruction only; the definition stays the same.

Instruction (w DW 1234h, b DB 5)	Result
`MOV AL, w`	error: 1 byte / 2 bytes
`MOV AL, BYTE PTR w`	AL = 34h (low byte)
`MOV AL, BYTE PTR w+1`	AL = 12h (high byte)
`MOV w, 5`	w = 0005h (word)
`MOV AX, b`	error: 2 bytes / 1 byte
`MOV [BX], 5`	error: size unknown
`MOV b, w`	error: memory to memory
`INC b`	b = 6 (byte increment)

Type checking happens at assembly time and catches many easy-to-miss errors. The processor knows no types at run time; it reads or writes the size the instruction encodes.

Code Trace: Two Names for the Same Memory (LABEL)

```
1 ORG 100h
▶ MOV AX, wval      ; WORD
3 MOV BL, bytes     ; low
4 MOV BH, bytes+1   ; high
5 MOV bytes, 0FFh   ; one byte
6 MOV CX, wval      ; word
7 RET
8 wval LABEL WORD
9 bytes DB 34h, 12h
```



read as a WORD

`wval LABEL WORD` reserves nothing; it only gives address 0115h a second name of type WORD. `MOV AX, wval` reads both bytes: AX = **1234h**.

Step 1 / 6

VARIABLES

`AX = 1234h` `BX = 0000h` `CX = 0000h`

name LABEL type gives the next byte a new name and type. To process a buffer both byte by byte and word by word, it reads better than writing PTR everywhere.

Code Trace: Two Names for the Same Memory (LABEL)

```
1 ORG 100h
2 MOV AX, wval        ; WORD
▶ MOV BL, bytes       ; low
4 MOV BH, bytes+1     ; high
5 MOV bytes, 0FFh     ; one byte
6 MOV CX, wval        ; word
7 RET
8 wval LABEL WORD
9 bytes DB 34h, 12h
```



`bytes` is the BYTE-typed name of the same address: BL = **34h**. No PTR needed.

Step 2 / 6

VARIABLES

`AX = 1234h` `BX = 0034h` `CX = 0000h`

name LABEL type gives the next byte a new name and type. To process a buffer both byte by byte and word by word, it reads better than writing PTR everywhere.

Code Trace: Two Names for the Same Memory (LABEL)

```
1 ORG 100h
2 MOV AX, wval        ; WORD
3 MOV BL, bytes       ; low
▶ MOV BH, bytes+1     ; high
5 MOV bytes, 0FFh     ; one byte
6 MOV CX, wval        ; word
7 RET
8 wval LABEL WORD
9 bytes DB 34h, 12h
```



`bytes+1` is the second byte: BH = **12h**. BX = 1234h, thanks to little-endian order.

Step 3 / 6

VARIABLES

AX = **1234h** BX = **1234h** CX = **0000h**

name LABEL type gives the next byte a new name and type. To process a buffer both byte by byte and word by word, it reads better than writing PTR everywhere.

Code Trace: Two Names for the Same Memory (LABEL)

```
1 ORG 100h
2 MOV AX, wval        ; WORD
3 MOV BL, bytes       ; low
4 MOV BH, bytes+1     ; high
▶ MOV bytes, 0FFh     ; one byte
6 MOV CX, wval        ; word
7 RET
8 wval LABEL WORD
9 bytes DB 34h, 12h
```



one byte written

The type is BYTE, so `MOV bytes, 0FFh` writes only address 0115h: memory is **FF 12**.

Step 4 / 6

VARIABLES

`AX = 1234h` `BX = 1234h` `CX = 0000h`

name LABEL type gives the next byte a new name and type. To process a buffer both byte by byte and word by word, it reads better than writing PTR everywhere.

Code Trace: Two Names for the Same Memory (LABEL)

```
1 ORG 100h
2 MOV AX, wval        ; WORD
3 MOV BL, bytes       ; low
4 MOV BH, bytes+1     ; high
5 MOV bytes, 0FFh     ; one byte
▶ MOV CX, wval        ; word
7 RET
8 wval LABEL WORD
9 bytes DB 34h, 12h
```



Read through the WORD name, the change shows: CX = **12FFh**. Two names, one memory location.

Step 5 / 6

VARIABLES

AX = **1234h** BX = **1234h** CX = **12FFh**

name LABEL type gives the next byte a new name and type. To process a buffer both byte by byte and word by word, it reads better than writing PTR everywhere.

Code Trace: Two Names for the Same Memory (LABEL)

```
1 ORG 100h
2 MOV AX, wval        ; WORD
3 MOV BL, bytes       ; low
4 MOV BH, bytes+1     ; high
5 MOV bytes, 0FFh     ; one byte
6 MOV CX, wval        ; word
▶ RET
8 wval LABEL WORD
9 bytes DB 34h, 12h
```



RET. `LABEL` produced no bytes; the program grew only by the two-byte definition.

Step 6 / 6

VARIABLES

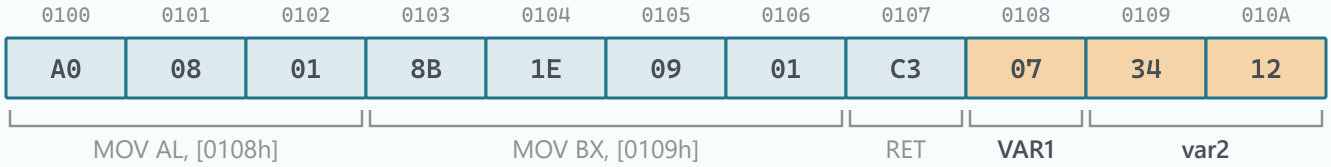
AX = **1234h** BX = **1234h** CX = **12FFh**

name LABEL type gives the next byte a new name and type. To process a buffer both byte by byte and word by word, it reads better than writing PTR everywhere.

Example Code

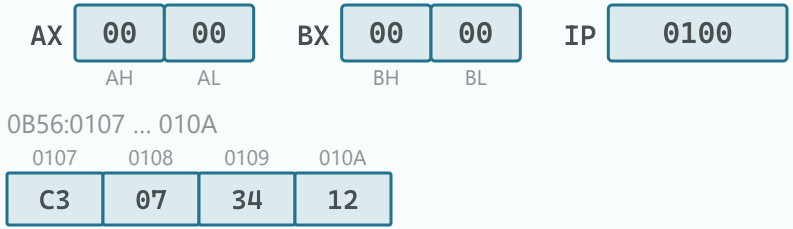
```
ORG 100h
MOV AL, var1    ; AL ← var1
MOV BX, var2    ; BX ← var2
RET             ; end program
VAR1 DB 7       ; BYTE, 7
var2 DW 1234h   ; WORD, 1234h
```

- Code first, variables **after RET**: the CPU never reaches them.
- The assembler turns the source into a **sequence of bytes** and replaces variable names with memory locations.
- It is case-insensitive: `VAR1` and `var1` are the same variable.



Code Trace: var1 and var2

```
1 ▶ ORG 100h
2   MOV AL, var1    ; AL ← var1
3   MOV BX, var2    ; BX ← var2
4   RET             ; end program
5   VAR1 DB 7       ; BYTE, 7
6   var2 DW 1234h   ; WORD, 1234h
```



The program is loaded at 0B56:0100; DS = CS = 0B56h, IP = 0100h.

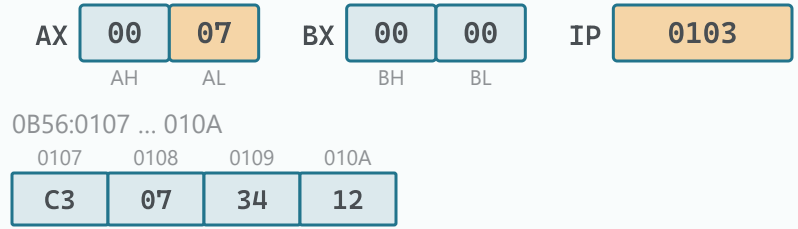
Step 1 / 4

VARIABLES

`IP = 0100h` `AL = 00h` `BX = 0000h`

Code Trace: var1 and var2

```
1 ORG 100h
▶ MOV AL, var1    ; AL ← var1
3 MOV BX, var2    ; BX ← var2
4 RET             ; end program
5 VAR1 DB 7       ; BYTE, 7
6 var2 DW 1234h   ; WORD, 1234h
```



`MOV AL, [0108h]` : the value of VAR1 → AL = **07h**. IP = 0103h.

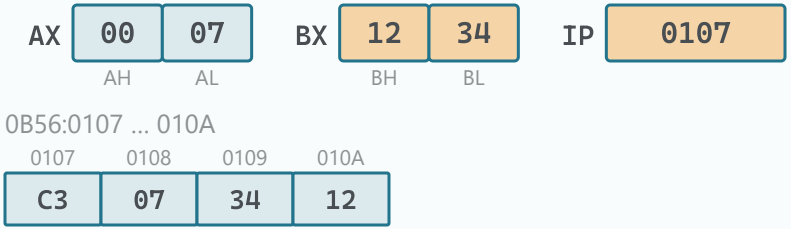
Step 2 / 4

VARIABLES

`IP = 0103h` `AL = 07h` `BX = 0000h`

Code Trace: var1 and var2

```
1 ORG 100h
2 MOV AL, var1    ; AL ← var1
▶ MOV BX, var2    ; BX ← var2
4 RET             ; end program
5 VAR1 DB 7       ; BYTE, 7
6 var2 DW 1234h   ; WORD, 1234h
```



`MOV BX, [0109h]` : 34h is the low byte, 12h the high byte → BX = **1234h**. IP = 0107h.

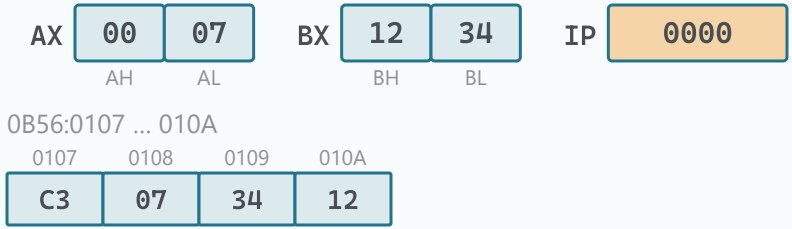
Step 3 / 4

VARIABLES

`IP = 0107h` `AL = 07h` `BX = 1234h`

Code Trace: var1 and var2

```
1 ORG 100h
2 MOV AL, var1    ; AL ← var1
3 MOV BX, var2    ; BX ← var2
▶ RET             ; end program
5 VAR1 DB 7       ; BYTE, 7
6 var2 DW 1234h   ; WORD, 1234h
```



RET (C3 at 0107h) pops 0000h: IP = **0000h**, and the INT 20h in the PSP ends the program. Everything from 0108h on is data.

Step 4 / 4

VARIABLES

IP = **0000h** AL = **07h** BX = **1234h**

What Does the Assembler Do?

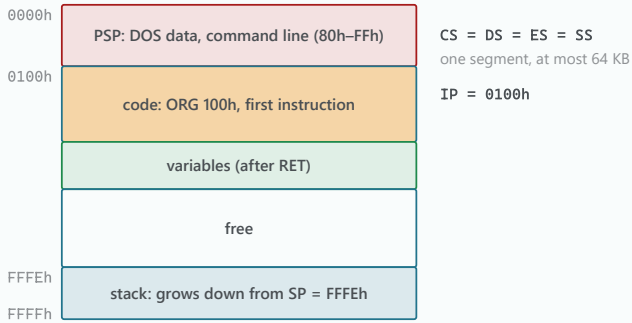
- It turns the source into a sequence of bytes and replaces variable names with **offsets**: `MOV AL, var1` → `MOV AL, [0108h]`.
- When a .COM file is loaded, **DS gets the value of CS** (so do ES and SS).
- An **offset** is the distance from the start of the segment.
- `VAR1` is at offset **0108h**; its full address is **0B56:0108**.
- `var2` is at offset **0109h**; its full address is **0B56:0109**.
  - As a WORD it takes 2 bytes (0109h and 010Ah).
  - The low byte sits at the low address: **34h comes before 12h** (little-endian).

The segment value (0B56h) depends on where DOS loads the program and may differ on every run; the offsets are fixed at assembly time.

MEMORY LAYOUT

The ORG 100h Directive

- Tells the assembler the **address where the program will be loaded**; it is used when names are replaced with addresses.
- Used for **.COM** files; most examples in this course use the emu8086 .COM template.
- The operating system keeps data about the program in the **first 256 bytes** of the segment (the PSP); the command-line arguments live there.
- **.EXE** files load at offset 0000h and use a separate segment for variables.
- Directives are **not** translated into machine code.



MEMORY LAYOUT

Alignment: Where Does a Word Start?

even address: 1 bus cycle



16-bit data bus
carries two bytes at once

odd address: 2 bus cycles



cycle 1: 0201 · cycle 2: 0202
+4 clocks per word

```
a   DB 5          ; 0200h
    EVEN          ; 0201h: pad
w   DW 1234h      ; 0202h (even)
```

- The 8086 data bus is 16 bits: a word at an even address is read in one bus cycle.
- A word at an odd address needs two cycles; the 8086 handles it, only slower (4 clocks per access).
- MASM/TASM's `EVEN` rounds the location counter up to even, adding a pad byte if needed. Definitions are only shifted, not reordered.
- On the 8088 (8-bit bus) every word takes two cycles anyway; alignment doesn't matter there.

Example: `MOV AX, w` takes 10 clocks on the 8086, 14 if w is odd-addressed. Read 1000 times in a loop: 4000 clocks, 0.8 ms at 5 MHz. Bytes need no alignment.

MEMORY LAYOUT

Where Do Variables Go?

Place	Example	Watch out for
After the code	`RET` · `x DB 5`	Execution ends at RET and never reaches the data. The most common layout for .COM.
Before the code	`JMP start` · `x DB 5` · `start:`	The JMP skips the data; forget it and the data bytes are executed as instructions.
Separate data segment (.EXE)	`.DATA` · `x DB 5`	The program must begin with `MOV AX, @data` · `MOV DS, AX`.

If DS is not loaded

When an .EXE starts, DS points to the PSP. `MOV AL, x` uses the right offset but the wrong segment: a wrong value comes back with no error.

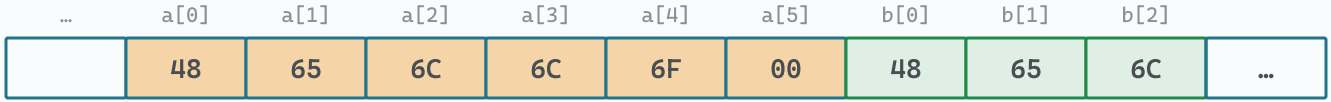
Why the segment matters

In a .COM file CS = DS, so code and data can share a segment. In an .EXE the code is in CS and the data in DS; variable addresses are relative to DS.

Arrays

- Think of an array as a **chain of variables**: a group of values stored in consecutive memory locations.
- Quoted text is converted automatically into a **byte array** (ASCII codes).
- Elements are reached through memory addresses: the first element is at the array's address, the rest follow.

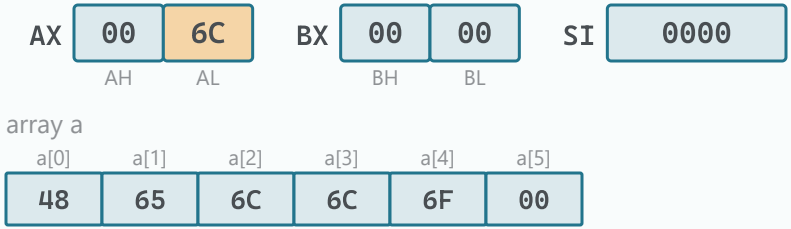
```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```



a and b hold the same bytes: 48h 'H', 65h 'e', 6Ch 'l', 6Ch 'l', 6Fh 'o', 00h (end of string).

Code Trace: Accessing Array Elements

```
1 ORG 100h
▶ MOV AL, a[3]   ; a + 3
3 MOV SI, 0
4 MOV BL, a[SI]  ; a + 0
5 INC SI
6 MOV BH, a[SI]  ; a + 1
7 MOV a[SI], 'E' ; a[1] ← 45h
8 RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



`a[3]` means **a + 3** bytes: AL = **6Ch** ('l').

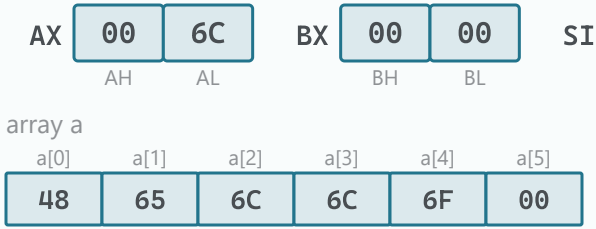
Step 1 / 7

VARIABLES

`AL = 6Ch` `SI = 0000h` `BL = 00h` `BH = 00h`

Code Trace: Accessing Array Elements

```
1 ORG 100h
2 MOV AL, a[3]    ; a + 3
▶ MOV SI, 0
4 MOV BL, a[SI]   ; a + 0
5 INC SI
6 MOV BH, a[SI]   ; a + 1
7 MOV a[SI], 'E' ; a[1] ← 45h
8 RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



Index register SI = 0.

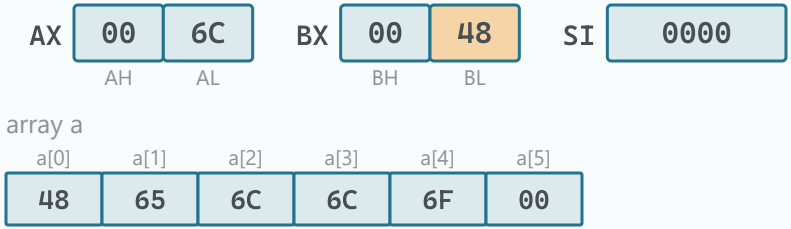
Step 2 / 7

VARIABLES

AL = 6Ch | SI = 0000h | BL = 00h | BH = 00h

Code Trace: Accessing Array Elements

```
1 ORG 100h
2 MOV AL, a[3]   ; a + 3
3 MOV SI, 0
4 MOV BL, a[SI]  ; a + 0
5 INC SI
6 MOV BH, a[SI]  ; a + 1
7 MOV a[SI], 'E' ; a[1] ← 45h
8 RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



`a[SI]` = [a + SI]: BL = **48h** ('H').

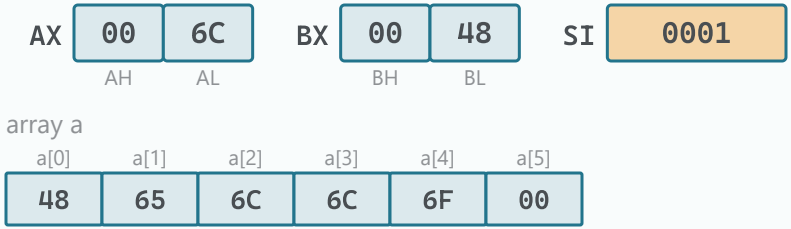
Step 3 / 7

VARIABLES

AL = **6Ch** · SI = **0000h** · BL = **48h** · BH = **00h**

Code Trace: Accessing Array Elements

```
1 ORG 100h
2 MOV AL, a[3]   ; a + 3
3 MOV SI, 0
4 MOV BL, a[SI]  ; a + 0
▶ INC SI
6 MOV BH, a[SI]  ; a + 1
7 MOV a[SI], 'E' ; a[1] ← 45h
8 RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



SI = 1: the next byte.

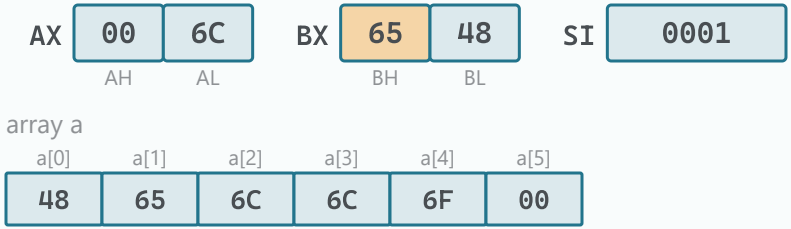
Step 4 / 7

VARIABLES

AL = 6Ch | SI = 0001h | BL = 48h | BH = 00h

Code Trace: Accessing Array Elements

```
1 ORG 100h
2 MOV AL, a[3]   ; a + 3
3 MOV SI, 0
4 MOV BL, a[SI]  ; a + 0
5 INC SI
▶ MOV BH, a[SI]  ; a + 1
7 MOV a[SI], 'E' ; a[1] ← 45h
8 RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



BH = a[1] = **65h** ('e'). BX = 6548h.

Step 5 / 7

VARIABLES

AL = **6Ch** SI = **0001h** BL = **48h** BH = **65h**

Code Trace: Accessing Array Elements

```
1 ORG 100h
2 MOV AL, a[3]   ; a + 3
3 MOV SI, 0
4 MOV BL, a[SI]  ; a + 0
5 INC SI
6 MOV BH, a[SI]  ; a + 1
▶ MOV a[SI], 'E' ; a[1] ← 45h
8 RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```

AX		BX		SI
00	6C	65	48	0001
AH	AL	BH	BL	

array a

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
48	45	6C	6C	6F	00

The array is of type byte, so the size is known: a[1] ← **45h** ('E'). The array now reads “HEllo”.

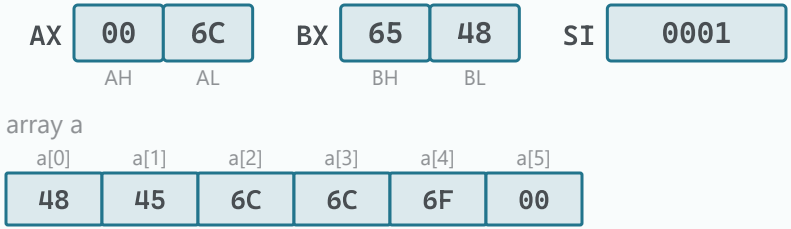
Step 6 / 7

VARIABLES

AL = **6Ch** SI = **0001h** BL = **48h** BH = **65h**

Code Trace: Accessing Array Elements

```
1 ORG 100h
2 MOV AL, a[3]   ; a + 3
3 MOV SI, 0
4 MOV BL, a[SI]  ; a + 0
5 INC SI
6 MOV BH, a[SI]  ; a + 1
7 MOV a[SI], 'E' ; a[1] ← 45h
▶ RET
9 a DB 48h,65h,6Ch,6Ch,6Fh,0
```



RET. The array definition sits after the code, where nothing is executed.

Step 7 / 7

VARIABLES

AL = **6Ch** SI = **0001h** BL = **48h** BH = **65h**

Access Patterns and DUP

Reaching an element

- A constant index in brackets: `MOV AL, a[3]`
- Through the memory index registers (BX, SI, DI, BP): `MOV SI, 3` · `MOV AL, a[SI]`
- Equivalent forms: `a[SI]` = `[a + SI]` = `[SI + a]`
- The index counts **bytes**, not elements.

The DUP operator

For large arrays: `count DUP(value(s))`

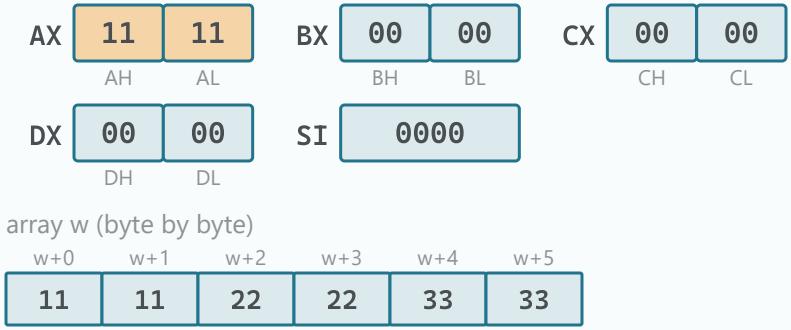
- `c DB 5 DUP(9)` = `c DB 9, 9, 9, 9, 9`
- `d DB 5 DUP(1, 2)` = `d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2`
- `buffer DB 100 DUP(?)` : 100 uninitialized bytes
- `table DW 10 DUP(0)` : 10 words = 20 bytes



DW can define **arrays of numbers** (`w DW 1000h, 2000h`); what it cannot define is **text strings** (`DW 'Hello'` is invalid). Values above 255 or below −128 need DW.

Code Trace: Indexing a DW Array

```
1 ORG 100h
▶ MOV AX, w[0]   ; element 1
3 MOV BX, w[2]   ; element 2
4 MOV SI, 4
5 MOV CX, w[SI]  ; element 3
6 MOV DX, w[1]   ; careful!
7 RET
8 w DW 1111h, 2222h, 3333h
```



w[0]: the first word → AX = **1111h**.

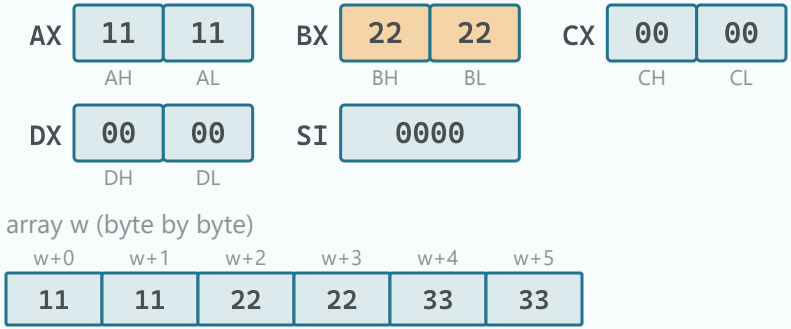
Step 1 / 5

VARIABLES

AX = **1111h** BX = **0000h** SI = **0000h** CX = **0000h** DX = **0000h**

Code Trace: Indexing a DW Array

```
1 ORG 100h
2 MOV AX, w[0]   ; element 1
▶ MOV BX, w[2]   ; element 2
4 MOV SI, 4
5 MOV CX, w[SI]  ; element 3
6 MOV DX, w[1]   ; careful!
7 RET
8 w DW 1111h, 2222h, 3333h
```



The second element is at **w[2]** (each element is 2 bytes): BX = 2222h.

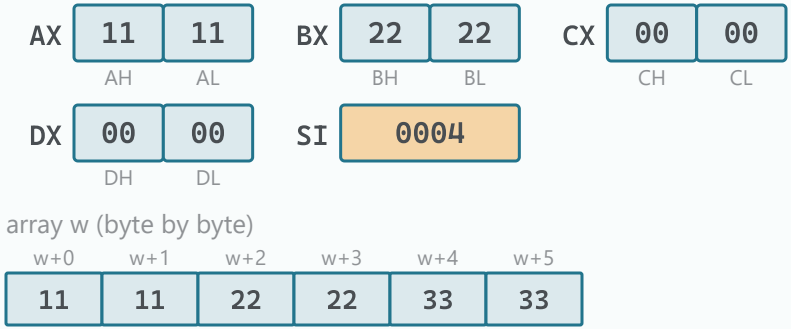
Step 2 / 5

VARIABLES

`AX = 1111h` `BX = 2222h` `SI = 0000h` `CX = 0000h` `DX = 0000h`

Code Trace: Indexing a DW Array

```
1 ORG 100h
2 MOV AX, w[0]   ; element 1
3 MOV BX, w[2]   ; element 2
▶ MOV SI, 4
5 MOV CX, w[SI]  ; element 3
6 MOV DX, w[1]   ; careful!
7 RET
8 w DW 1111h, 2222h, 3333h
```



The index of element i = **2 × i**: for the third element, SI = 4.

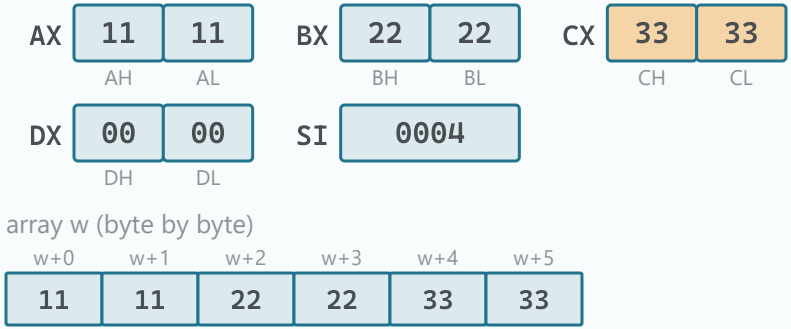
Step 3 / 5

VARIABLES

AX = **1111h** BX = **2222h** SI = **0004h** CX = **0000h** DX = **0000h**

Code Trace: Indexing a DW Array

```
1 ORG 100h
2 MOV AX, w[0]   ; element 1
3 MOV BX, w[2]   ; element 2
4 MOV SI, 4
▶ MOV CX, w[SI]  ; element 3
6 MOV DX, w[1]   ; careful!
7 RET
8 w DW 1111h, 2222h, 3333h
```



CX = w[SI] = **3333h**.

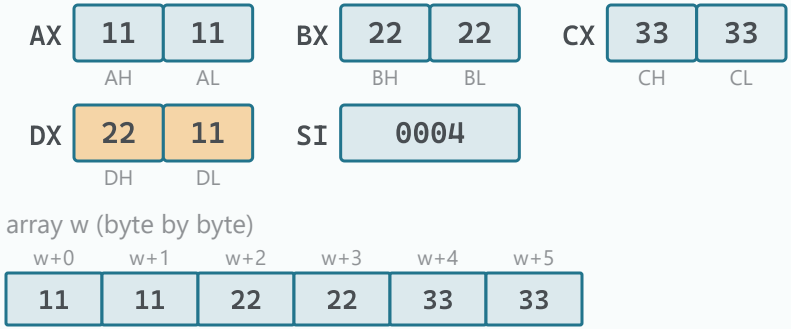
Step 4 / 5

VARIABLES

AX = **1111h** BX = **2222h** SI = **0004h** CX = **3333h** DX = **0000h**

Code Trace: Indexing a DW Array

```
1 ORG 100h
2 MOV AX, w[0]   ; element 1
3 MOV BX, w[2]   ; element 2
4 MOV SI, 4
5 MOV CX, w[SI]  ; element 3
▶ MOV DX, w[1]   ; careful!
7 RET
8 w DW 1111h, 2222h, 3333h
```



w[1] reads from the middle of an element: bytes 11h, 22h → DX = **2211h**. A common mistake!

Step 5 / 5

VARIABLES

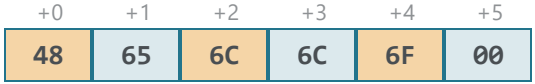
AX = **1111h** BX = **2222h** SI = **0004h** CX = **3333h**

DX = **2211h**

From Definition to Bytes

```
b DB 'Hello', 0
```

DB list | DW | DUP | DW ?, −1 | DD | error



b: 6 bytes · item size 1 byte · color change = one item

Type a DB / DW / DD line to see its bytes. Numbers (decimal, `h`, `b`, `o`), `'text'` (DB only), `?` and `n DUP(…)` are supported. Above each cell: the byte offset from the start of the variable.

Strings and Terminators

```
s1 DB 'Welcome', 0              ; zero-terminated
s2 DB 'Welcome$'                ; '$'-terminated (INT 21h / AH = 09h)
s3 DB 13, 10, 'Line', 13, 10, '$'  ; CR LF: carriage return + line feed
s4 DB 7, 'Welcome'              ; length-prefixed: first byte = 7
s5 DB "Ali's", 0                ; single quote inside double quotes
```

Format	How is the end found?	Where is it used?
zero-terminated	when the byte read is 0	emu8086.inc `PRINT_STRING`, the C language
'$'-terminated	when the byte read is 24h	the DOS print-string service
length-prefixed	as many bytes as the first byte says	the DOS buffered-input buffer (AH = 0Ah)

- Space taken in memory = number of characters + terminator: 8 bytes for s1. Forget the terminator and printing runs on, treating the bytes after the string as characters.
- A '$'-terminated string cannot print '$' itself. Codes for Turkish letters (ç, ğ, ş…) depend on the code page in use; they are not in ASCII's 0–127 range.

Character Constants and ASCII Arithmetic

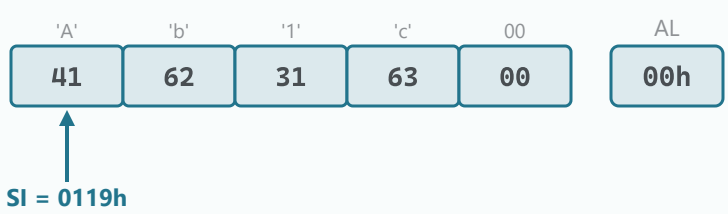
Characters	ASCII codes
'0' … '9'	30h … 39h
'A' … 'Z'	41h … 5Ah
'a' … 'z'	61h … 7Ah
space · '$'	20h · 24h
CR · LF	0Dh · 0Ah

- To the assembler a quoted character is just a number: `MOV AL, 'A'` = `MOV AL, 41h`.
- Digit character to number: `SUB AL, '0'` ('7' = 37h → 7). The other way: `ADD AL, '0'`.
- Upper and lower case differ by **20h**, i.e. only bit 5: `SUB AL, 20h` or `AND AL, 0DFh` converts to upper case, `OR AL, 20h` to lower case.

`AND AL, 0DFh` is only correct for letters. Applied to '1' (31h) it gives 11h and ruins the digit, so first check that AL is a letter ('a' ≤ AL ≤ 'z').

Code Trace: Converting a String to Upper Case

```
1  ORG 100h
▶  LEA SI, s
3  again:
4  MOV AL, [SI]
5  CMP AL, 0          ; terminator?
6  JE done
7  CMP AL, 'a'
8  JB next
9  CMP AL, 'z'
10 JA next
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 next:
14 INC SI
15 JMP again
16 done:
17 RET
18 s DB 'Ab1c', 0
```



`LEA SI, s` : SI = **0119h**, the first byte of the string 'Ab1c' with terminator 0.

Step 1 / 6

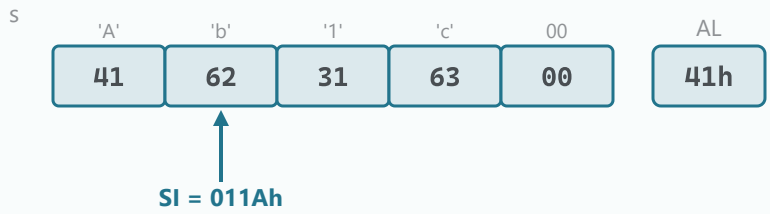
VARIABLES

`SI = 0119h` `AL = 00h` `ZF = 0`

The string was changed in place: each byte is read and, if it is a letter, written back to the same address minus 20h. A terminator (0) took the place of a length.

Code Trace: Converting a String to Upper Case

```
1  ORG 100h
2  LEA SI, s
3  again:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; terminator?
▶  JE done
▶  CMP AL, 'a'
▶  JB next
9  CMP AL, 'z'
10 JA next
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 next:
▶  INC SI
15 JMP again
16 done:
17 RET
18 s DB 'Ab1c', 0
```



'A' (41h): not 0; `CMP AL, 'a'` → 41h < 61h, `JB` jumps. Already upper case, so nothing is written; SI moves on.

Step 2 / 6

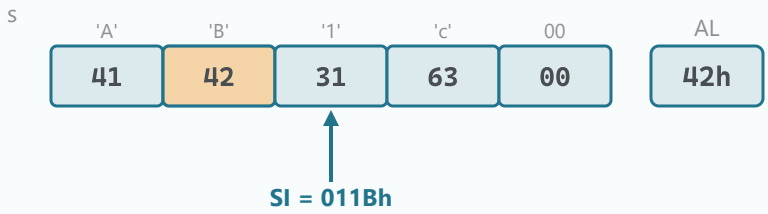
VARIABLES

`SI = 011Ah` `AL = 41h` `ZF = 0`

The string was changed in place: each byte is read and, if it is a letter, written back to the same address minus 20h. A terminator (0) took the place of a length.

Code Trace: Converting a String to Upper Case

```
1  ORG 100h
2  LEA SI, s
3  again:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; terminator?
▶  JE done
▶  CMP AL, 'a'
▶  JB next
▶  CMP AL, 'z'
▶  JA next
▶  SUB AL, 20h        ; 'a' - 'A'
▶  MOV [SI], AL
13 next:
▶  INC SI
▶  JMP again
16 done:
17 RET
18 s DB 'Ab1c', 0
```



'b' (62h): 'a' ≤ 62h ≤ 'z' → `SUB AL, 20h` = **42h** ('B'), and `MOV [SI], AL` writes it back.

Step 3 / 6

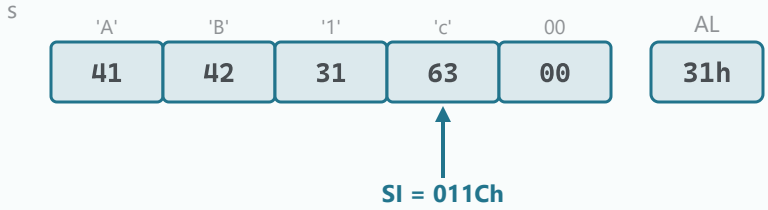
VARIABLES

`SI = 011Bh` `AL = 42h` `ZF = 0`

The string was changed in place: each byte is read and, if it is a letter, written back to the same address minus 20h. A terminator (0) took the place of a length.

Code Trace: Converting a String to Upper Case

```
1  ORG 100h
2  LEA SI, s
3  again:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; terminator?
▶  JE done
▶  CMP AL, 'a'
▶  JB next
9  CMP AL, 'z'
10 JA next
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 next:
▶  INC SI
▶  JMP again
16 done:
17 RET
18 s DB 'Ab1c', 0
```



'1' (31h): below 'a', skipped. Without the letter check, 31h − 20h = 11h would have ruined the digit.

Step 4 / 6

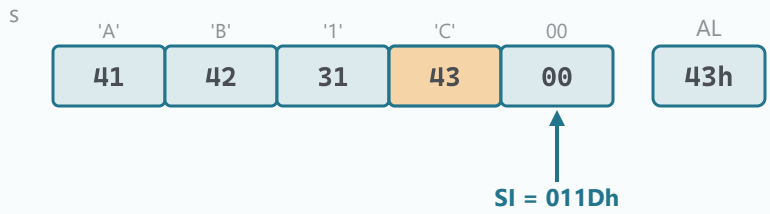
VARIABLES

`SI = 011Ch` `AL = 31h` `ZF = 0`

The string was changed in place: each byte is read and, if it is a letter, written back to the same address minus 20h. A terminator (0) took the place of a length.

Code Trace: Converting a String to Upper Case

```
1  ORG 100h
2  LEA SI, s
3  again:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; terminator?
▶  JE done
▶  CMP AL, 'a'
▶  JB next
▶  CMP AL, 'z'
▶  JA next
▶  SUB AL, 20h        ; 'a' - 'A'
▶  MOV [SI], AL
13 next:
▶  INC SI
▶  JMP again
16 done:
17 RET
18 s DB 'Ab1c', 0
```



'c' (63h) → **43h** ('C'). The string is now 'AB1C'.

Step 5 / 6

VARIABLES

SI = 011Dh AL = 43h ZF = 0

The string was changed in place: each byte is read and, if it is a letter, written back to the same address minus 20h. A terminator (0) took the place of a length.

Code Trace: Converting a String to Upper Case

```
1  ORG 100h
2  LEA SI, s
3  again:
▶  MOV AL, [SI]
▶  CMP AL, 0          ; terminator?
▶  JE done
7  CMP AL, 'a'
8  JB next
9  CMP AL, 'z'
10 JA next
11 SUB AL, 20h        ; 'a' - 'A'
12 MOV [SI], AL
13 next:
14 INC SI
▶  JMP again
16 done:
▶  RET
18 s DB 'Ab1c', 0
```



terminator: ZF = 1

The byte read is 0: `CMP AL, 0` → ZF = 1, `JE done`. The loop stopped at the terminator without knowing the length.

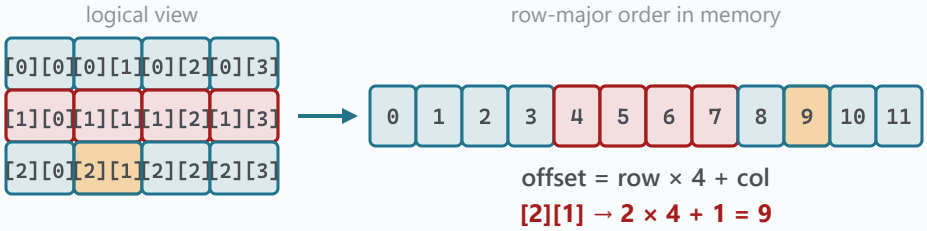
Step 6 / 6

VARIABLES

`SI = 011Dh` `AL = 00h` `ZF = 1`

The string was changed in place: each byte is read and, if it is a letter, written back to the same address minus 20h. A terminator (0) took the place of a length.

Two-Dimensional Arrays (Tables)



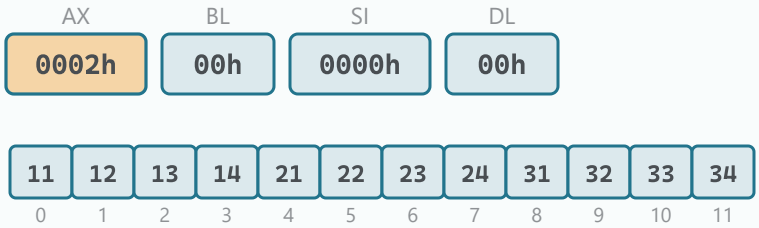

```
mat DB 3 DUP(4 DUP(0)) ; 3 × 4
tab DW 3 DUP(4 DUP(0)) ; 24 bytes
```

- Memory is one-dimensional; a table is laid out in **row-major order**: all columns of row 0, then row 1…
- Offset = (row × number of columns + column) × element size.
- In a word table, [1][2] is at (1 × 4 + 2) × 2 = **12** bytes.
- Nested DUP reserves rows and columns together: 3 times (4 times 0).

If the number of columns is a power of 2, a shift replaces the multiply: two `SHL` s for row × 4. The number of rows does not affect the layout; it only bounds the row index.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
▶ MOV AL, row          ; AL = 2
4  MOV BL, NCOLS       ; BL = 4
5  MUL BL              ; AX = 2 × 4
6  ADD AL, col         ; + 1
7  MOV SI, AX          ; SI = 9
8  MOV DL, mat[SI]     ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



AL ← row = **2**. Target: mat[2][1]. The table is at 0112h, 3 rows × 4 columns of bytes.

Step 1 / 7

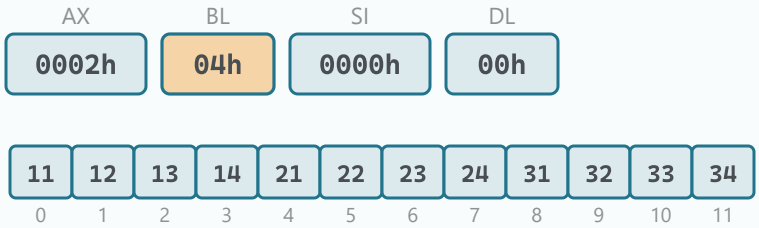
VARIABLES

AX = **0002h** BL = **00h** SI = **0000h** DL = **00h**

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
3  MOV AL, row          ; AL = 2
4  MOV BL, NCOLS        ; BL = 4
5  MUL BL               ; AX = 2 × 4
6  ADD AL, col          ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



`NCOLS EQU 4` is encoded in the instruction: BL = **04h** (number of columns).

Step 2 / 7

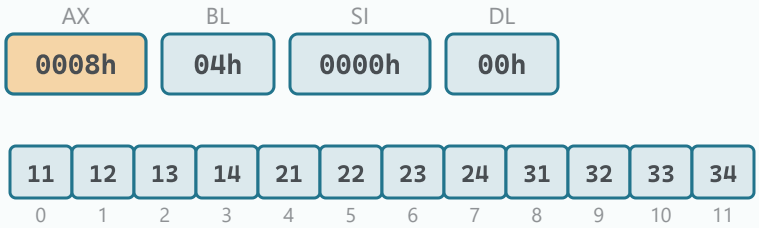
VARIABLES

AX = **0002h** | BL = **04h** | SI = **0000h** | DL = **00h**

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
3  MOV AL, row          ; AL = 2
4  MOV BL, NCOLS        ; BL = 4
▶  MUL BL               ; AX = 2 × 4
6  ADD AL, col          ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



`MUL BL` : AX = AL × BL = 2 × 4 = **0008h**. Row 2 starts at byte 8 of the table.

Step 3 / 7

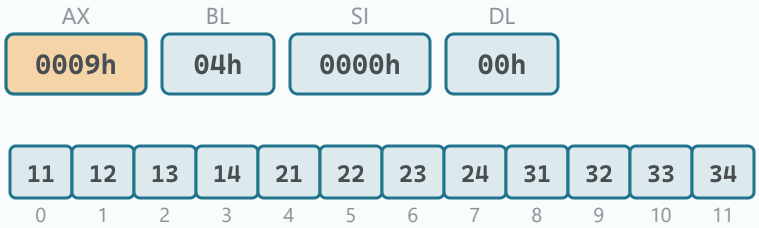
VARIABLES

AX = 0008h | BL = 04h | SI = 0000h | DL = 00h

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
3  MOV AL, row          ; AL = 2
4  MOV BL, NCOLS        ; BL = 4
5  MUL BL               ; AX = 2 × 4
▶  ADD AL, col          ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



`ADD AL, col`: 8 + 1 = **9**. AH is already 0, so AX = 0009h.

Step 4 / 7

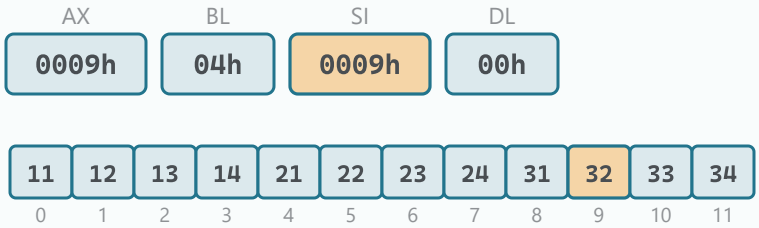
VARIABLES

AX = **0009h** BL = **04h** SI = **0000h** DL = **00h**

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
3  MOV AL, row          ; AL = 2
4  MOV BL, NCOLS        ; BL = 4
5  MUL BL               ; AX = 2 × 4
6  ADD AL, col          ; + 1
▶  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



The offset goes into an index register: SI = **0009h**. AL cannot be an index; only BX, SI, DI, BP can address memory.

Step 5 / 7

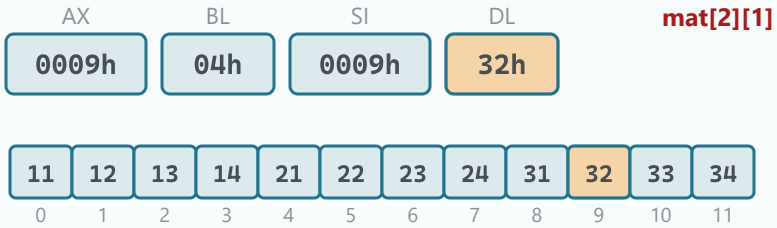
VARIABLES

AX = 0009h | BL = 04h | SI = 0009h | DL = 00h

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
3  MOV AL, row          ; AL = 2
4  MOV BL, NCOLS        ; BL = 4
5  MUL BL               ; AX = 2 × 4
6  ADD AL, col          ; + 1
7  MOV SI, AX           ; SI = 9
▶  MOV DL, mat[SI]      ; mat[2][1]
9  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



`mat[SI]` = [0112h + 9]: DL = **32h**. Row 2, column 1.

Step 6 / 7

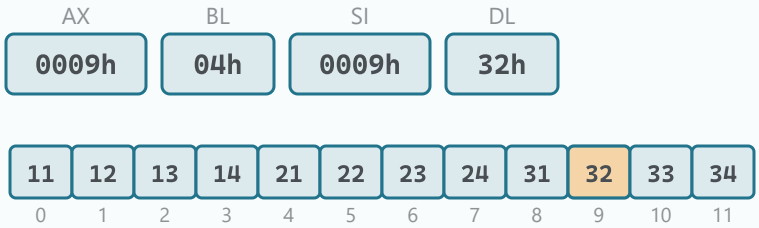
VARIABLES

AX = **0009h** BL = **04h** SI = **0009h** DL = **32h**

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

Code Trace: Reading mat[2][1]

```
1  NCOLS EQU 4
2  ORG 100h
3  MOV AL, row          ; AL = 2
4  MOV BL, NCOLS        ; BL = 4
5  MUL BL               ; AX = 2 × 4
6  ADD AL, col          ; + 1
7  MOV SI, AX           ; SI = 9
8  MOV DL, mat[SI]      ; mat[2][1]
▶  RET
10 mat DB 11h, 12h, 13h, 14h
11     DB 21h, 22h, 23h, 24h
12     DB 31h, 32h, 33h, 34h
13 row DB 2
14 col DB 1
```



RET. The constant is named NCOLS on purpose: `COL EQU 4` would be the same name as `col DB 1`, since the assembler ignores case, and give a “duplicate definition” error.

Step 7 / 7

VARIABLES

`AX = 0009h` `BL = 04h` `SI = 0009h` `DL = 32h`

Row and column were read from variables at run time; the same code works for every element. In a word table the result would also be multiplied by 2.

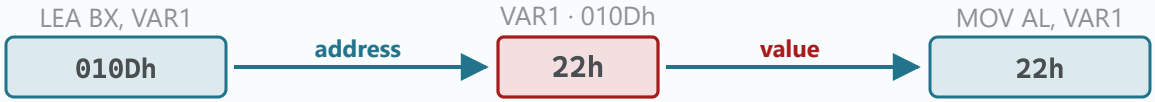
Getting a Variable's Address

LEA (Load Effective Address)

Loads the **address** (offset) of a variable into a register: `LEA BX, VAR1`. It also works for indexed variables: `LEA BX, a[SI]`.

OFFSET

An assembler operator; it supplies the variable's offset **value** at assembly time: `MOV BX, OFFSET VAR1`.

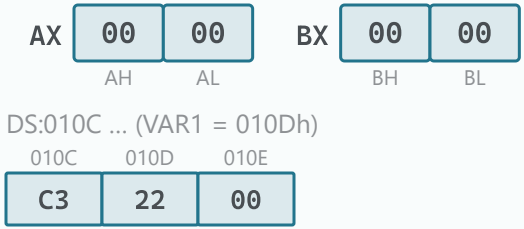


A variable's address is very handy for **passing parameters to a procedure**: instead of a large array, the procedure gets just its 16-bit address.

Instruction	What goes into BX?
`MOV BX, VAR1` (VAR1 DW)	the **value** of VAR1 (its contents)
`LEA BX, VAR1` · `MOV BX, OFFSET VAR1`	the **address** of VAR1 (its offset)

Code Trace: Updating a Variable Through LEA

```
1 ORG 100h
2 MOV AL, VAR1          ; value
3 LEA BX, VAR1          ; address
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; new value
6 RET
7 VAR1 DB 22h
8 END
```



The code starts at 0100h; the assembler placed VAR1 at **010Dh** (12 bytes of code + RET).

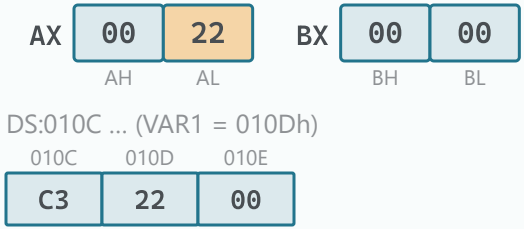
Step 1 / 6

VARIABLES

`AL = 00h` `BX = 0000h` `VAR1 = 22h`

Code Trace: Updating a Variable Through LEA

```
1 ORG 100h
▶ MOV AL, VAR1          ; value
3 LEA BX, VAR1          ; address
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; new value
6 RET
7 VAR1 DB 22h
8 END
```



AL = the **value** of VAR1 = 22h.

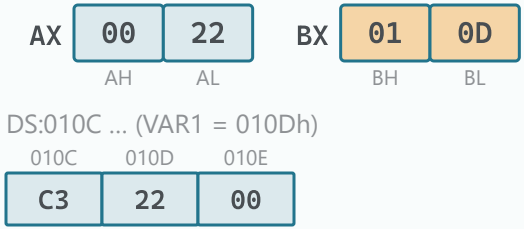
Step 2 / 6

VARIABLES

AL = 22h BX = 0000h VAR1 = 22h

Code Trace: Updating a Variable Through LEA

```
1 ORG 100h
2 MOV AL, VAR1          ; value
▶ LEA BX, VAR1          ; address
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; new value
6 RET
7 VAR1 DB 22h
8 END
```



BX = the **address** of VAR1 = 010Dh. BX is now a pointer to VAR1.

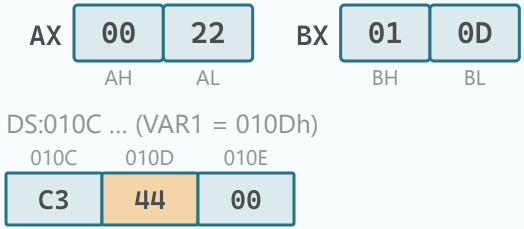
Step 3 / 6

VARIABLES

`AL = 22h` `BX = 010Dh` `VAR1 = 22h`

Code Trace: Updating a Variable Through LEA

```
1 ORG 100h
2 MOV AL, VAR1          ; value
3 LEA BX, VAR1          ; address
▶ MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; new value
6 RET
7 VAR1 DB 22h
8 END
```



VAR1 is updated through [BX]: [010Dh] = **44h**.

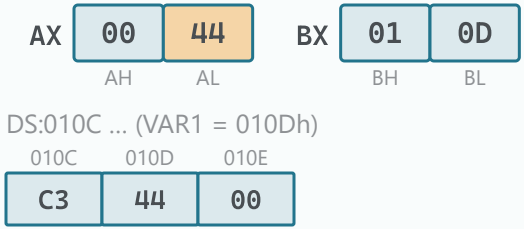
Step 4 / 6

VARIABLES

AL = **22h** BX = **010Dh** VAR1 = **44h**

Code Trace: Updating a Variable Through LEA

```
1 ORG 100h
2 MOV AL, VAR1          ; value
3 LEA BX, VAR1          ; address
4 MOV BYTE PTR [BX], 44h
▶ MOV AL, VAR1          ; new value
6 RET
7 VAR1 DB 22h
8 END
```



VAR1 is read again: AL = **44h**. A variable can be reached by its name or through a pointer.

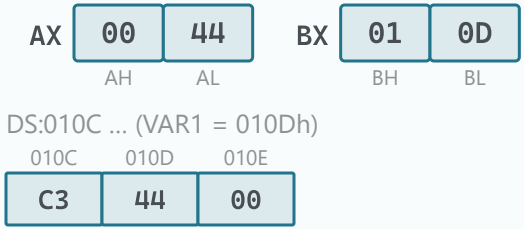
Step 5 / 6

VARIABLES

AL = 44h BX = 010Dh VAR1 = 44h

Code Trace: Updating a Variable Through LEA

```
1 ORG 100h
2 MOV AL, VAR1          ; value
3 LEA BX, VAR1          ; address
4 MOV BYTE PTR [BX], 44h
5 MOV AL, VAR1          ; new value
▶ RET
7 VAR1 DB 22h
8 END
```



RET. `END` is the directive that marks the end of the source file.

Step 6 / 6

VARIABLES

`AL = 44h` `BX = 010Dh` `VAR1 = 44h`

Are LEA and OFFSET the Same?

The same program with OFFSET

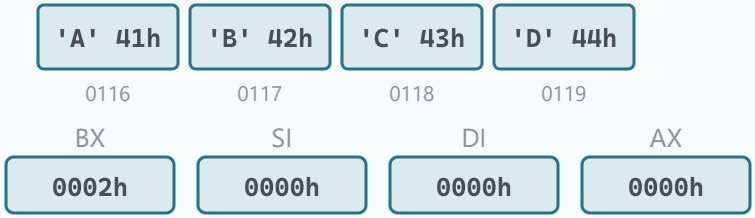
```
ORG 100h
MOV AL, VAR1
MOV BX, OFFSET VAR1
MOV BYTE PTR [BX], 44h
MOV AL, VAR1
RET
VAR1 DB 22h
END
```

- In emu8086, `LEA BX, VAR1` and `MOV BX, OFFSET VAR1` assemble to the **same machine code**: `BB 0D 01` (3 bytes, `MOV BX, 010Dh`). VAR1 = 010Dh in both programs.
- A real 8086 assembler such as MASM encodes LEA as its own instruction: `8D 1E 0D 01` (4 bytes).
- The difference: `LEA BX, a[SI]` computes the address **at run time** (a + SI); OFFSET gives only a constant address.
- Only **BX, SI, DI, BP** may appear inside square brackets.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
▶ MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```

a



BX = 2: the index. The array `a` starts at 0116h: 'A' 'B' 'C' 'D'.

Step 1 / 8

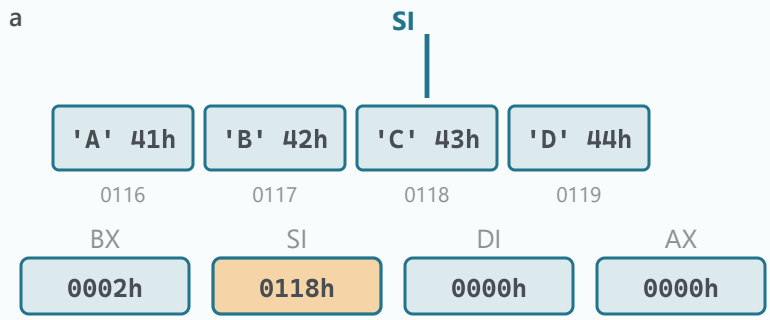
VARIABLES

BX = **0002h** SI = **0000h** DI = **0000h** AX = **0000h**

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
▶  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



`LEA SI, a[BX]` : the address is computed **at run time**: 0116h + 2 = **0118h**. Memory is not read.

Step 2 / 8

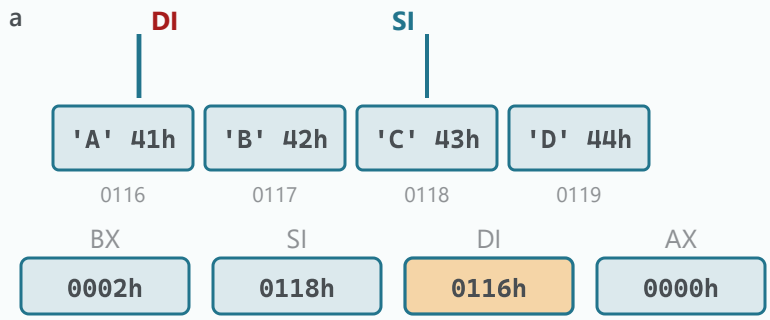
VARIABLES

`BX = 0002h` `SI = 0118h` `DI = 0000h` `AX = 0000h`

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
▶  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



`OFFSET a` is a constant known at assembly time: DI = **0116h**. It cannot include BX.

Step 3 / 8

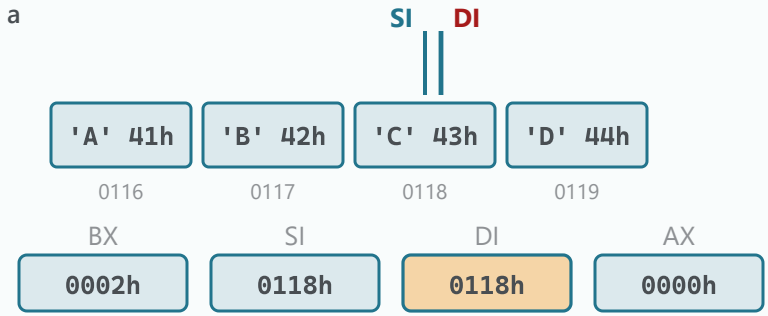
VARIABLES

BX = **0002h** SI = **0118h** DI = **0116h** AX = **0000h**

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
▶  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



With OFFSET the same address needs an extra `ADD`: DI = **0118h** = SI.

Step 4 / 8

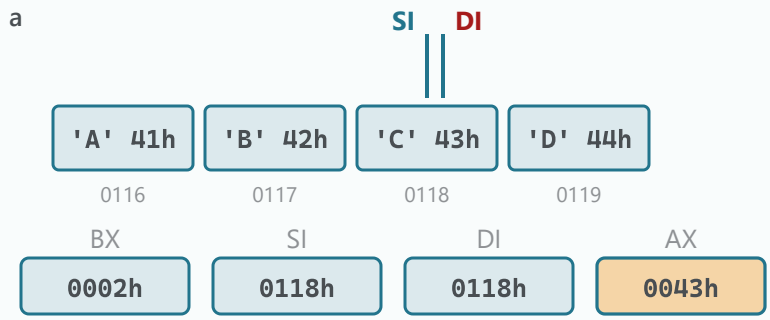
VARIABLES

BX = **0002h** | SI = **0118h** | DI = **0118h** | AX = **0000h**

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
▶  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



AL = [SI] = **43h** ('C').

Step 5 / 8

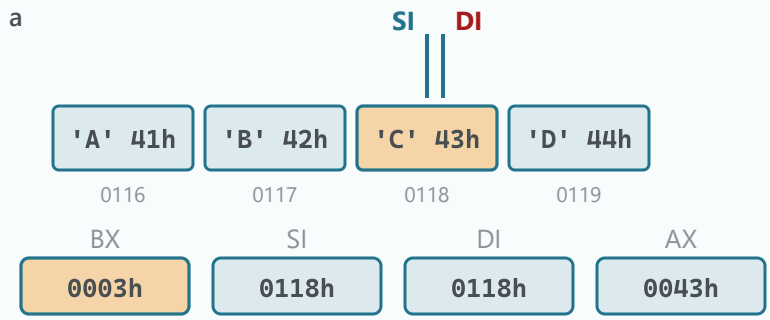
VARIABLES

BX = 0002h | SI = 0118h | DI = 0118h | AX = 0043h

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
▶  INC BX
8  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



BX = 3.

Step 6 / 8

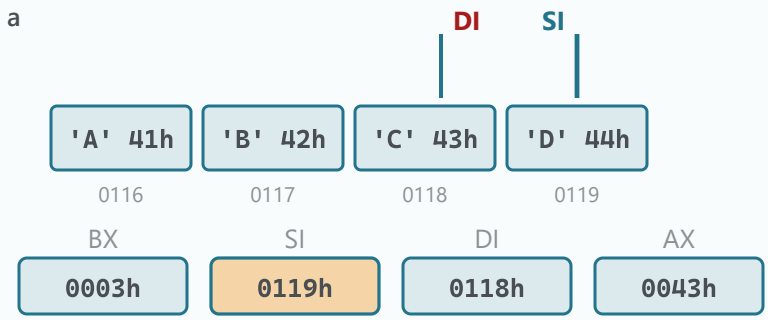
VARIABLES

BX = 0003h SI = 0118h DI = 0118h AX = 0043h

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
7  INC BX
▶  LEA SI, a[BX]       ; new address
9  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



`LEA` recomputes with the current BX: SI = **0119h**.

Step 7 / 8

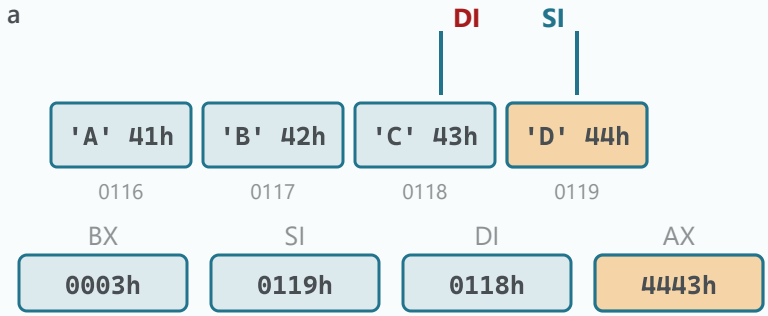
VARIABLES

BX = **0003h** SI = **0119h** DI = **0118h** AX = **0043h**

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Code Trace: LEA a[BX] vs. OFFSET a

```
1  ORG 100h
2  MOV BX, 2
3  LEA SI, a[BX]       ; SI = a + BX
4  MOV DI, OFFSET a    ; fixed address
5  ADD DI, BX          ; add by hand
6  MOV AL, [SI]
7  INC BX
8  LEA SI, a[BX]       ; new address
▶  MOV AH, [SI]
10 RET
11 a DB 'ABCD'
```



AH = [SI] = **44h** ('D'). AX = 4443h.

Step 8 / 8

VARIABLES

BX = 0003h SI = 0119h DI = 0118h AX = 4443h

LEA computes the address from registers at run time; OFFSET gives only a constant address known at assembly time.

Pointer Variables

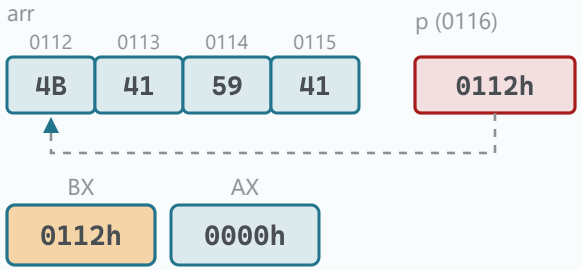
```
ORG 100h
MOV BX, p           ; 010Bh
LEA SI, p           ; 010Fh
MOV DI, OFFSET arr
RET
arr   DB 'KAYA'     ; 010Bh
p     DW arr        ; 010Fh, holds 010Bh
far_p DD arr        ; offset + segment
```

- If the value of a DW is a variable name, the **address** of that variable is stored: p is a pointer variable.
- The pointer lives in memory, so it can change: `ADD p, 2` moves it further into the array.
- To use it, first load it into an address register: `MOV BX, p` · `MOV AL, [BX]`.
- A far pointer defined with `DD` holds offset and segment together; `LDS SI, far_p` or `LES DI, far_p` loads it in one instruction.

Instruction	What goes into the register?	Value
`MOV BX, p`	the address stored in p (the array's address)	010Bh
`LEA SI, p`	p's own address	010Fh
`MOV DI, OFFSET arr`	the array's address, fixed at assembly time	010Bh

Code Trace: A Pointer in Memory

```
1 ORG 100h
▶ MOV BX, p          ; contents of p
3 MOV AL, [BX]       ; arr[0]
4 ADD p, 2           ; advance
5 MOV BX, p
6 MOV AH, [BX]       ; arr[2]
7 RET
8 arr  DB 'KAYA'
9 p    DW arr        ; address
```



`MOV BX, p` reads the **contents** of p. Since p = `DW arr`, BX = **0112h**, the array's address.

Step 1 / 6

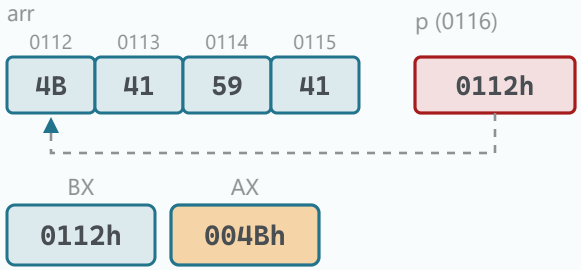
VARIABLES

`BX = 0112h` `AX = 0000h` `p = 0112h`

A pointer variable keeps an address in memory; with so few registers, it lets you track many arrays. Load it into BX, SI or DI before each use.

Code Trace: A Pointer in Memory

```
1 ORG 100h
2 MOV BX, p          ; contents of p
▶ MOV AL, [BX]       ; arr[0]
4 ADD p, 2           ; advance
5 MOV BX, p
6 MOV AH, [BX]       ; arr[2]
7 RET
8 arr  DB 'KAYA'
9 p    DW arr        ; address
```



`MOV AL, [BX]` : the byte the pointer points to: AL = **4Bh** ('K').

Step 2 / 6

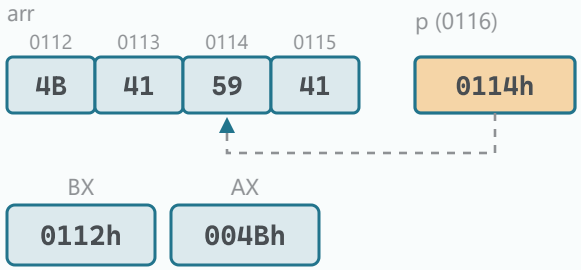
VARIABLES

`BX = 0112h` `AX = 004Bh` `p = 0112h`

A pointer variable keeps an address in memory; with so few registers, it lets you track many arrays. Load it into BX, SI or DI before each use.

Code Trace: A Pointer in Memory

```
1 ORG 100h
2 MOV BX, p          ; contents of p
3 MOV AL, [BX]       ; arr[0]
▶ ADD p, 2           ; advance
5 MOV BX, p
6 MOV AH, [BX]       ; arr[2]
7 RET
8 arr  DB 'KAYA'
9 p    DW arr        ; address
```



pointer moved

`ADD p, 2` : the pointer in memory now points to **0114h**. BX is unchanged; it still holds the old address.

Step 3 / 6

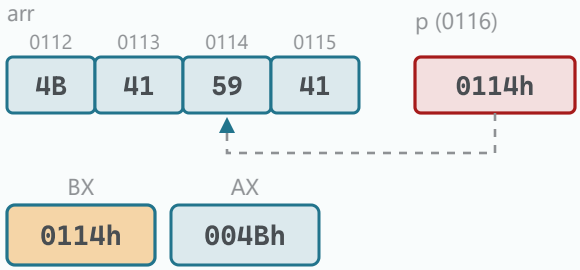
VARIABLES

BX = **0112h** AX = **004Bh** p = **0114h**

A pointer variable keeps an address in memory; with so few registers, it lets you track many arrays. Load it into BX, SI or DI before each use.

Code Trace: A Pointer in Memory

```
1 ORG 100h
2 MOV BX, p          ; contents of p
3 MOV AL, [BX]       ; arr[0]
4 ADD p, 2           ; advance
▶ MOV BX, p
6 MOV AH, [BX]       ; arr[2]
7 RET
8 arr  DB 'KAYA'
9 p    DW arr        ; address
```



The pointer is reloaded: BX = **0114h**.

Step 4 / 6

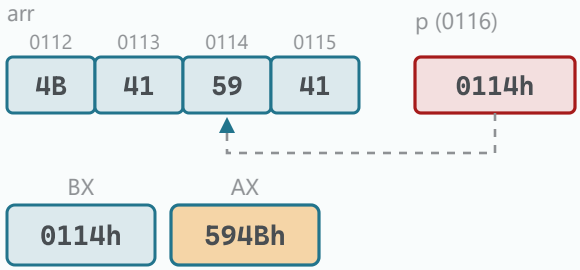
VARIABLES

BX = 0114h AX = 004Bh p = 0114h

A pointer variable keeps an address in memory; with so few registers, it lets you track many arrays. Load it into BX, SI or DI before each use.

Code Trace: A Pointer in Memory

```
1 ORG 100h
2 MOV BX, p          ; contents of p
3 MOV AL, [BX]       ; arr[0]
4 ADD p, 2           ; advance
5 MOV BX, p
▶ MOV AH, [BX]       ; arr[2]
7 RET
8 arr  DB 'KAYA'
9 p    DW arr        ; address
```



`MOV AH, [BX]` : AH = **59h** ('Y'). AX = 594Bh.

Step 5 / 6

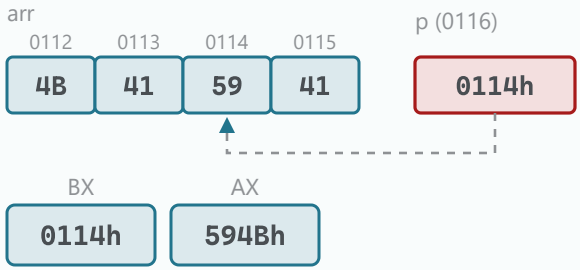
VARIABLES

`BX = 0114h` `AX = 594Bh` `p = 0114h`

A pointer variable keeps an address in memory; with so few registers, it lets you track many arrays. Load it into BX, SI or DI before each use.

Code Trace: A Pointer in Memory

```
1 ORG 100h
2 MOV BX, p          ; contents of p
3 MOV AL, [BX]       ; arr[0]
4 ADD p, 2           ; advance
5 MOV BX, p
6 MOV AH, [BX]       ; arr[2]
▶ RET
8 arr  DB 'KAYA'
9 p    DW arr        ; address
```



RET. The array's address never appears in the code; the pointer in memory decided which element was read.

Step 6 / 6

VARIABLES

BX = **0114h** AX = **594Bh** p = **0114h**

A pointer variable keeps an address in memory; with so few registers, it lets you track many arrays. Load it into BX, SI or DI before each use.

Constants (EQU)

- Like variables, but they exist only **at assembly time**; they take no memory.
- Once defined, a constant's value **cannot change**.
- They let different parts of a program use the same value, updated in one place.
- Definition: `name EQU <any expression>`
- The assembler replaces the name with its value at every use: `MOV AX, k` → `MOV AX, 5` (`B8 05 00`).

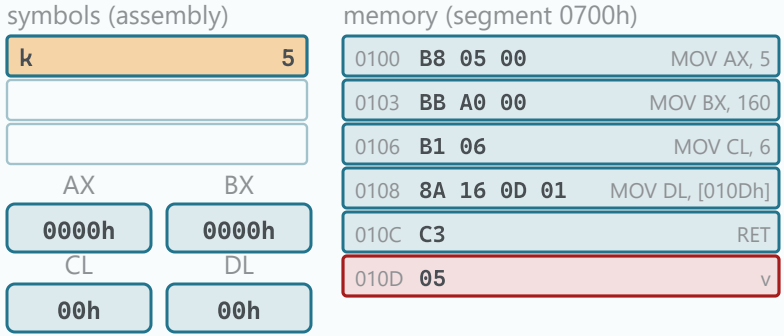
```
k       EQU 5
ROWLEN EQU 80

MOV AX, k          ; AX = 5
MOV BX, ROWLEN*2   ; BX = 160
MOV CL, k+1        ; CL = 6
```

	`k EQU 5`	`k DB 5`
Memory	none	1 byte
`MOV AL, k`	AL ← 5 (immediate)	AL ← [k] (from memory)
Can it change?	no	yes: `MOV k, 7`

Code Trace: EQU Is Built into the Instruction

```
1  k       EQU 5
2  ROWLEN  EQU 80
3  ORG 100h
4  MOV AX, k          ; MOV AX, 5
5  MOV BX, ROWLEN*2   ; MOV BX, 160
6  MOV CL, k+1        ; MOV CL, 6
7  MOV DL, v          ; from memory
8  RET
9  v DB 5             ; takes 1 byte
```



`k EQU 5` : the assembler records **k = 5** in its symbol table. No memory is reserved and no machine code is produced.

Step 1 / 6

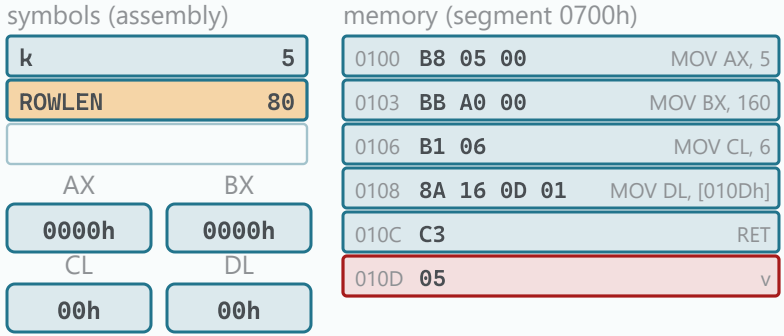
VARIABLES

AX = 0000h | BX = 0000h | CL = 00h | DL = 00h

Machine codes follow the MASM/emu8086 encoding: B8 iw (MOV AX, imm), BB iw (MOV BX, imm), B1 ib (MOV CL, imm), 8A 16 addr (MOV DL, [addr]).

Code Trace: EQU Is Built into the Instruction

```
1 k       EQU 5
2 ROWLEN EQU 80
3 ORG 100h
4 MOV AX, k          ; MOV AX, 5
5 MOV BX, ROWLEN*2   ; MOV BX, 160
6 MOV CL, k+1        ; MOV CL, 6
7 MOV DL, v          ; from memory
8 RET
9 v DB 5             ; takes 1 byte
```



`ROWLEN = 80` also exists only at assembly time. The program starts at 0100h; its first byte is the code of `MOV AX, k`.

Step 2 / 6

VARIABLES

AX = 0000h | BX = 0000h | CL = 00h | DL = 00h

Machine codes follow the MASM/emu8086 encoding: B8 iw (MOV AX, imm), BB iw (MOV BX, imm), B1 ib (MOV CL, imm), 8A 16 addr (MOV DL, [addr]).

Code Trace: EQU Is Built into the Instruction

```
1 k       EQU 5
2 ROWLEN EQU 80
3 ORG 100h
▶ MOV AX, k           ; MOV AX, 5
5 MOV BX, ROWLEN*2    ; MOV BX, 160
6 MOV CL, k+1         ; MOV CL, 6
7 MOV DL, v           ; from memory
8 RET
9 v DB 5              ; takes 1 byte
```

symbols (assembly)

k	5
ROWLEN	80
v	010Dh (address)

AX	BX
0005h	0000h

CL	DL
00h	00h

memory (segment 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`MOV AX, k` → `MOV AX, 5` (B8 05 00): 5 is inside the instruction, an immediate. AX = **0005h**.

Step 3 / 6

VARIABLES

AX = 0005h | BX = 0000h | CL = 00h | DL = 00h

Machine codes follow the MASM/emu8086 encoding: B8 iw (MOV AX, imm), BB iw (MOV BX, imm), B1 ib (MOV CL, imm), 8A 16 addr (MOV DL, [addr]).

Code Trace: EQU Is Built into the Instruction

```
1 k       EQU 5
2 ROWLEN  EQU 80
3 ORG 100h
4 MOV AX, k           ; MOV AX, 5
▶ MOV BX, ROWLEN*2    ; MOV BX, 160
6 MOV CL, k+1         ; MOV CL, 6
7 MOV DL, v           ; from memory
8 RET
9 v DB 5              ; takes 1 byte
```

symbols (assembly)

k	5
ROWLEN	80
v	010Dh (address)

AX	BX
0005h	00A0h

CL	DL
00h	00h

memory (segment 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`ROWLEN*2` was computed at assembly time: 160 = 00A0h → **BX = 00A0h**.

Step 4 / 6

VARIABLES

AX = 0005h BX = 00A0h CL = 00h DL = 00h

Machine codes follow the MASM/emu8086 encoding: B8 iw (MOV AX, imm), BB iw (MOV BX, imm), B1 ib (MOV CL, imm), 8A 16 addr (MOV DL, [addr]).

Code Trace: EQU Is Built into the Instruction

```
1 k       EQU 5
2 ROWLEN  EQU 80
3 ORG 100h
4 MOV AX, k           ; MOV AX, 5
5 MOV BX, ROWLEN*2    ; MOV BX, 160
▶ MOV CL, k+1         ; MOV CL, 6
7 MOV DL, v           ; from memory
8 RET
9 v DB 5              ; takes 1 byte
```

symbols (assembly)

k	5
ROWLEN	80
v	010Dh (address)

AX	BX
0005h	00A0h

CL	DL
06h	00h

memory (segment 0700h)

0100	B8 05 00	MOV AX, 5
0103	BB A0 00	MOV BX, 160
0106	B1 06	MOV CL, 6
0108	8A 16 0D 01	MOV DL, [010Dh]
010C	C3	RET
010D	05	v

`k+1` = 6 → CL = **06h**. No addition happens at run time.

Step 5 / 6

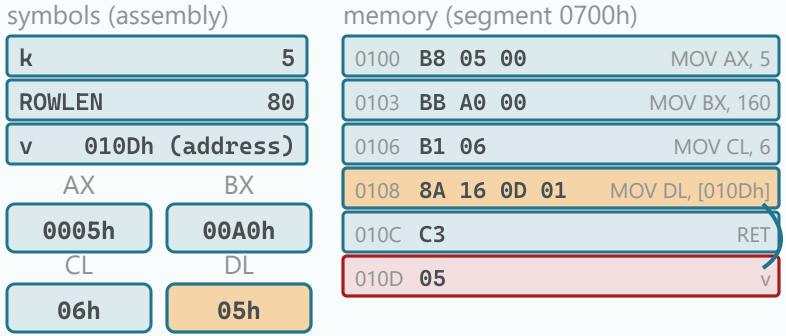
VARIABLES

AX = **0005h** · BX = **00A0h** · CL = **06h** · DL = **00h**

Machine codes follow the MASM/emu8086 encoding: B8 iw (MOV AX, imm), BB iw (MOV BX, imm), B1 ib (MOV CL, imm), 8A 16 addr (MOV DL, [addr]).

Code Trace: EQU Is Built into the Instruction

```
1 k       EQU 5
2 ROWLEN  EQU 80
3 ORG 100h
4 MOV AX, k          ; MOV AX, 5
5 MOV BX, ROWLEN*2   ; MOV BX, 160
6 MOV CL, k+1        ; MOV CL, 6
▶ MOV DL, v          ; from memory
8 RET
9 v DB 5             ; takes 1 byte
```



`v` is a variable: the instruction carries its address (010Dh) and the value is read **from memory**: DL = 05h. v takes 1 byte; k takes none.

Step 6 / 6

VARIABLES

AX = 0005h | BX = 00A0h | CL = 06h | DL = 05h

Machine codes follow the MASM/emu8086 encoding: B8 iw (MOV AX, imm), BB iw (MOV BX, imm), B1 ib (MOV CL, imm), 8A 16 addr (MOV DL, [addr]).

The Location Counter ($) and Size Operators

- The assembler keeps the offset where the next byte will go in a counter: the **location counter**. In source code it is written `$`.
- Right after an array, `$ - arr` is the array's size in bytes. If the array grows or shrinks, the constant stays correct by itself.
- The EQU line must come **immediately after** the array; any definition in between is counted too.
- MASM's `TYPE`, `LENGTHOF` and `SIZEOF` operators read the same information from the variable's definition.

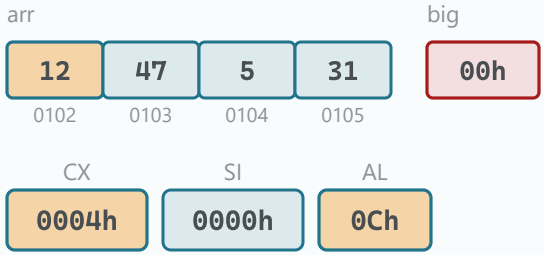
```
s DB 'Welcome', 0
n EQU $ - s        ; 8
w DW 10 DUP(0)
```

Expression	Value	Meaning
`n`	8	size of s in bytes
`TYPE s`	1	element size (DB)
`LENGTHOF s`	8	number of elements
`TYPE w`	2	element size (DW)
`LENGTHOF w`	10	number of elements
`SIZEOF w`	20	LENGTHOF × TYPE

All of these are assembly-time constants: `MOV CX, LENGTHOF w` builds 10 into the instruction and takes no memory. For a word array, take the loop count from LENGTHOF and the byte length from SIZEOF.

Code Trace: Size with $, Maximum with a Loop

```
1  ORG 100h
2  JMP start
3  arr DB 12, 47, 5, 31
4  n EQU $ - arr       ; 4
5  big DB ?
6  start:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, arr[SI]     ; first guess
10 check:
11 CMP AL, arr[SI]
12 JAE skip            ; AL >= element
13 MOV AL, arr[SI]     ; new guess
14 skip:
15 INC SI
16 LOOP check
17 MOV big, AL
18 RET
```



AL = 12

`JMP start` skips the data. `n EQU $ - arr` = 4 was computed at assembly time: CX = **0004h**. The first element is the first guess: AL = 12.

Step 1 / 6

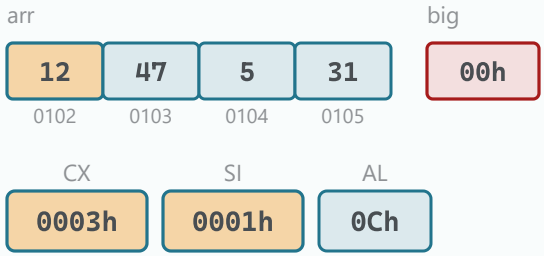
VARIABLES

`CX = 0004h` `SI = 0000h` `AL = 0Ch` `CF = 0`

The loop count is not hard-coded; add an element and n changes by itself. JAE compares unsigned; use JGE for signed arrays.

Code Trace: Size with $, Maximum with a Loop

```
1  ORG 100h
2  JMP start
3  arr DB 12, 47, 5, 31
4  n EQU $ - arr       ; 4
5  big DB ?
6  start:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, arr[SI]     ; first guess
10 check:
▶  CMP AL, arr[SI]
▶  JAE skip            ; AL >= element
13 MOV AL, arr[SI]     ; new guess
14 skip:
▶  INC SI
▶  LOOP check
17 MOV big, AL
18 RET
```



Pass 1: 12 is compared with 12; `JAE` jumps (equal). SI = 1, LOOP: CX = 3.

Step 2 / 6

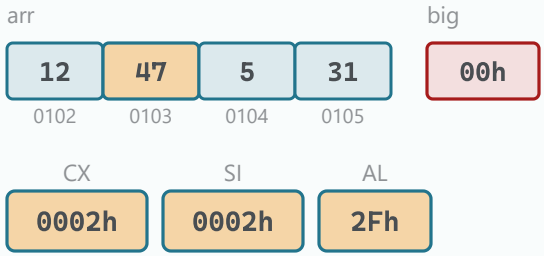
VARIABLES

`CX = 0003h` `SI = 0001h` `AL = 0Ch` `CF = 0`

The loop count is not hard-coded; add an element and n changes by itself. JAE compares unsigned; use JGE for signed arrays.

Code Trace: Size with $, Maximum with a Loop

```
1  ORG 100h
2  JMP start
3  arr DB 12, 47, 5, 31
4  n EQU $ - arr       ; 4
5  big DB ?
6  start:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, arr[SI]     ; first guess
10 check:
▶  CMP AL, arr[SI]
▶  JAE skip            ; AL >= element
▶  MOV AL, arr[SI]     ; new guess
14 skip:
▶  INC SI
▶  LOOP check
17 MOV big, AL
18 RET
```



Pass 2: 12 < 47 → CF = 1, `JAE` does not jump; AL = **47** (2Fh). SI = 2, CX = 2.

Step 3 / 6

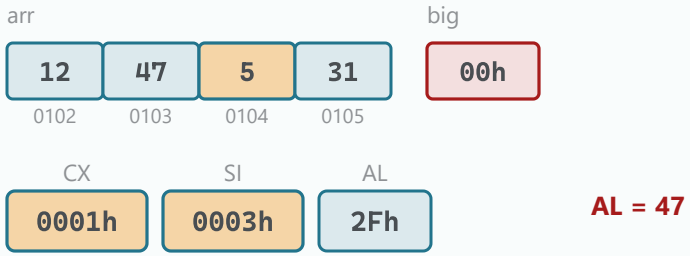
VARIABLES

`CX = 0002h` `SI = 0002h` `AL = 2Fh` `CF = 1`

The loop count is not hard-coded; add an element and n changes by itself. JAE compares unsigned; use JGE for signed arrays.

Code Trace: Size with $, Maximum with a Loop

```
1  ORG 100h
2  JMP start
3  arr DB 12, 47, 5, 31
4  n EQU $ - arr       ; 4
5  big DB ?
6  start:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, arr[SI]     ; first guess
10 check:
▶  CMP AL, arr[SI]
▶  JAE skip            ; AL >= element
13 MOV AL, arr[SI]     ; new guess
14 skip:
▶  INC SI
▶  LOOP check
17 MOV big, AL
18 RET
```



Pass 3: 47 ≥ 5 → skipped, AL unchanged. SI = 3, CX = 1.

Step 4 / 6

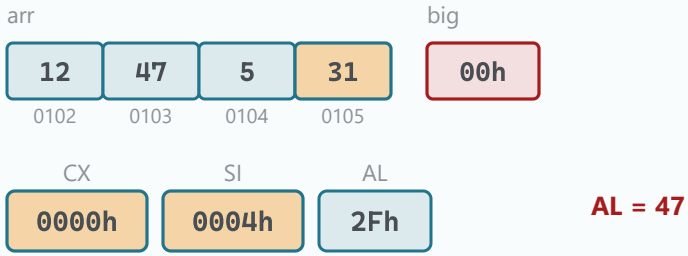
VARIABLES

CX = 0001h SI = 0003h AL = 2Fh CF = 0

The loop count is not hard-coded; add an element and n changes by itself. JAE compares unsigned; use JGE for signed arrays.

Code Trace: Size with $, Maximum with a Loop

```
1  ORG 100h
2  JMP start
3  arr DB 12, 47, 5, 31
4  n EQU $ - arr       ; 4
5  big DB ?
6  start:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, arr[SI]     ; first guess
10 check:
▶  CMP AL, arr[SI]
▶  JAE skip            ; AL >= element
13 MOV AL, arr[SI]     ; new guess
14 skip:
▶  INC SI
▶  LOOP check
17 MOV big, AL
18 RET
```



Pass 4: 47 ≥ 31 → skipped. LOOP: CX = 0, the loop ends.

Step 5 / 6

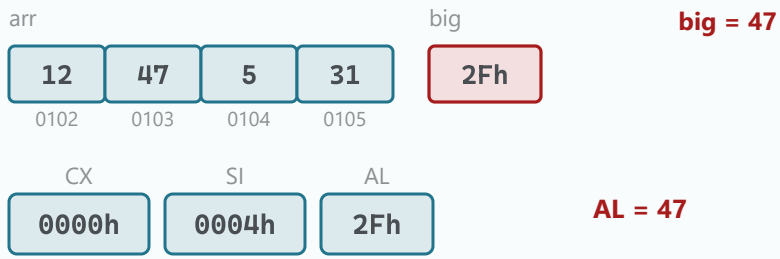
VARIABLES

CX = 0000h SI = 0004h AL = 2Fh CF = 0

The loop count is not hard-coded; add an element and n changes by itself. JAE compares unsigned; use JGE for signed arrays.

Code Trace: Size with $, Maximum with a Loop

```
1  ORG 100h
2  JMP start
3  arr DB 12, 47, 5, 31
4  n EQU $ - arr       ; 4
5  big DB ?
6  start:
7  MOV CX, n           ; CX = 4
8  MOV SI, 0
9  MOV AL, arr[SI]     ; first guess
10 check:
11 CMP AL, arr[SI]
12 JAE skip            ; AL >= element
13 MOV AL, arr[SI]     ; new guess
14 skip:
15 INC SI
16 LOOP check
▶  MOV big, AL
▶  RET
```



`MOV big, AL`: the largest element, **2Fh** = 47, is stored; RET.

Step 6 / 6

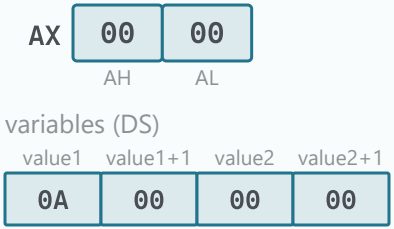
VARIABLES

`CX = 0000h` `SI = 0004h` `AL = 2Fh` `CF = 0`

The loop count is not hard-coded; add an element and n changes by itself. JAE compares unsigned; use JGE for signed arrays.

Code Trace: Moving Data Between Variables

```
1 ORG 100h
▶ jmp start
3 value1 DW 10     ; a 2-byte variable
4 value2 DW 0      ; a second one
5 start:
6 MOV AX, value1   ; AX ← value1
7 MOV value2, AX   ; value2 ← AX
8 RET
9 end start
```



`jmp start` makes the CPU **jump over** the data; otherwise the data bytes would be executed as instructions.

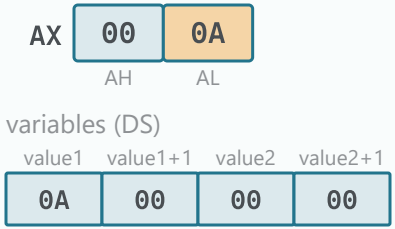
Step 1 / 4

VARIABLES

`AX = 0000h` `value1 = 000Ah` `value2 = 0000h`

Code Trace: Moving Data Between Variables

```
1 ORG 100h
2 jmp start
3 value1 DW 10     ; a 2-byte variable
4 value2 DW 0      ; a second one
5 start:
▶ MOV AX, value1   ; AX ← value1
7 MOV value2, AX   ; value2 ← AX
8 RET
9 end start
```



value1 → AX = **000Ah**. There is no direct memory-to-memory MOV, so a register sits in between.

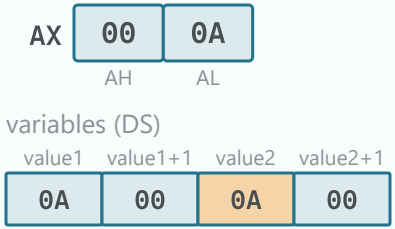
Step 2 / 4

VARIABLES

AX = 000Ah | value1 = 000Ah | value2 = 0000h

Code Trace: Moving Data Between Variables

```
1 ORG 100h
2 jmp start
3 value1 DW 10     ; a 2-byte variable
4 value2 DW 0      ; a second one
5 start:
6 MOV AX, value1   ; AX ← value1
▶ MOV value2, AX   ; value2 ← AX
8 RET
9 end start
```



AX → value2: **0A 00** in memory. Now value1 = value2 = 10.

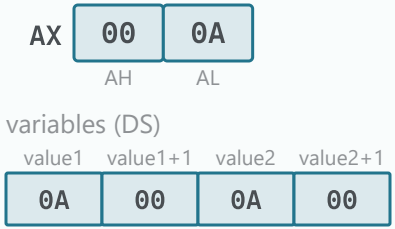
Step 3 / 4

VARIABLES

`AX = 000Ah` `value1 = 000Ah` `value2 = 000Ah`

Code Trace: Moving Data Between Variables

```
1 ORG 100h
2 jmp start
3 value1 DW 10     ; a 2-byte variable
4 value2 DW 0      ; a second one
5 start:
6 MOV AX, value1   ; AX ← value1
7 MOV value2, AX   ; value2 ← AX
▶ RET
9 end start
```



`end start` : the end of the source; **start** is the program's entry point.

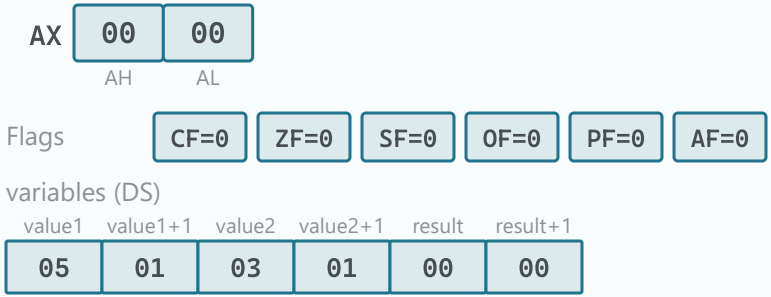
Step 4 / 4

VARIABLES

`AX = 000Ah` `value1 = 000Ah` `value2 = 000Ah`

Code Trace: Adding Variables

```
1  ORG 100h
▶  jmp start
3  value1 DW 105h   ; initially 105h
4  value2 DW 103h   ; initially 103h
5  result DW ?      ; undefined
6  start:
7  MOV AX, value1   ; AX ← value1
8  ADD AX, value2   ; AX ← AX + value2
9  MOV result, AX   ; result ← AX
10 RET
11 end start
```



Start: result is defined with `?` ; emu8086 stores 00 00, but do not rely on that value.

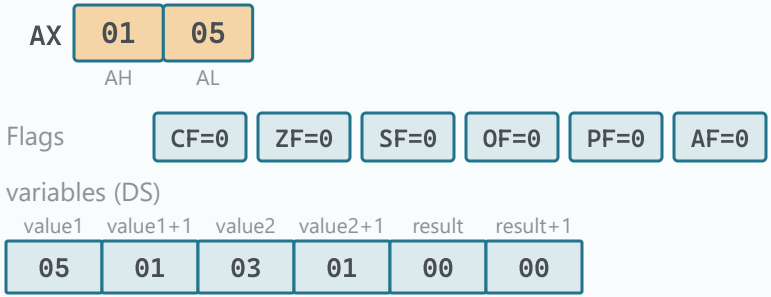
Step 1 / 5

VARIABLES

`AX = 0000h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`
`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`
`result = 0000h`

Code Trace: Adding Variables

```
1  ORG 100h
2  jmp start
3  value1 DW 105h   ; initially 105h
4  value2 DW 103h   ; initially 103h
5  result DW ?      ; undefined
6  start:
▶  MOV AX, value1   ; AX ← value1
8  ADD AX, value2   ; AX ← AX + value2
9  MOV result, AX   ; result ← AX
10 RET
11 end start
```



AX = value1 = **0105h**.

Step 2 / 5

VARIABLES

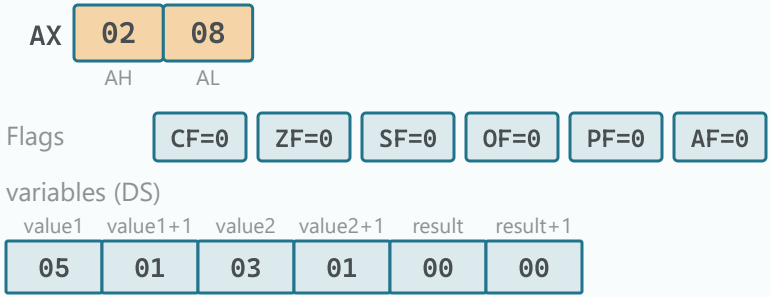
`AX = 0105h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`

`result = 0000h`

Code Trace: Adding Variables

```
1  ORG 100h
2  jmp start
3  value1 DW 105h   ; initially 105h
4  value2 DW 103h   ; initially 103h
5  result DW ?      ; undefined
6  start:
7  MOV AX, value1   ; AX ← value1
▶  ADD AX, value2   ; AX ← AX + value2
9  MOV result, AX   ; result ← AX
10 RET
11 end start
```



ADD can take a memory operand directly: 0105h + 0103h = **0208h**. Flags: CF = 0, ZF = 0, PF = 0 (08h has a single 1 bit).

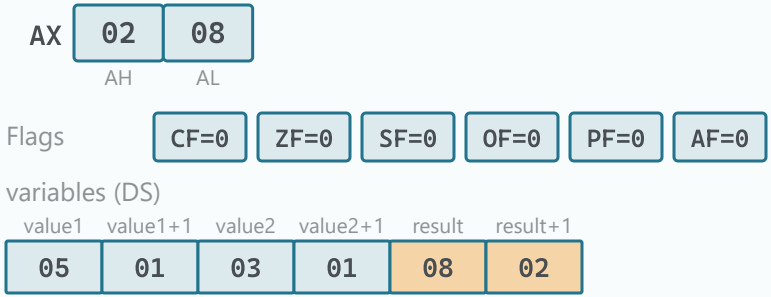
Step 3 / 5

VARIABLES

`AX = 0208h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`
`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`
`result = 0000h`

Code Trace: Adding Variables

```
1  ORG 100h
2  jmp start
3  value1 DW 105h   ; initially 105h
4  value2 DW 103h   ; initially 103h
5  result DW ?      ; undefined
6  start:
7  MOV AX, value1   ; AX ← value1
8  ADD AX, value2   ; AX ← AX + value2
▶  MOV result, AX   ; result ← AX
10 RET
11 end start
```



result ← AX: **08 02** in memory (little-endian).

Step 4 / 5

VARIABLES

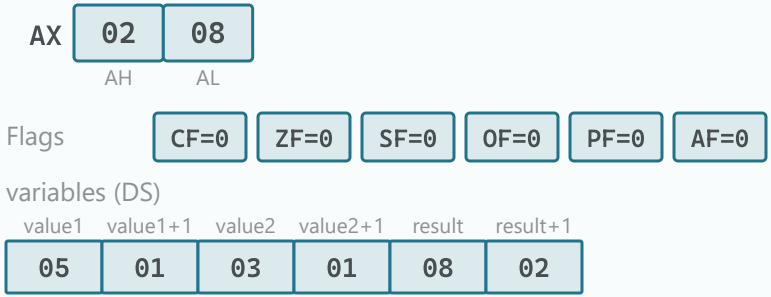
`AX = 0208h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`

`result = 0208h`

Code Trace: Adding Variables

```
1  ORG 100h
2  jmp start
3  value1 DW 105h  ; initially 105h
4  value2 DW 103h  ; initially 103h
5  result DW ?     ; undefined
6  start:
7  MOV AX, value1  ; AX ← value1
8  ADD AX, value2  ; AX ← AX + value2
9  MOV result, AX  ; result ← AX
▶  RET
11 end start
```



Result: result = 0208h = 520 (261 + 259).

Step 5 / 5

VARIABLES

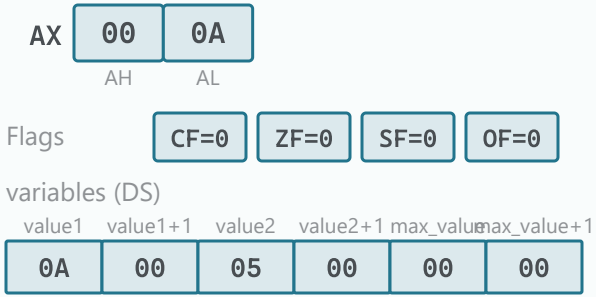
`AX = 0208h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`PF = 0` `AF = 0` `value1 = 0105h` `value2 = 0103h`

`result = 0208h`

Code Trace: Find the Larger (10, 5)

```
1  ORG 100h
▶ MOV AX, value1      ; AX ← value1
3  CMP AX, value2     ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2     ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX  ; store result
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



AX = value1 = **000Ah**.

Step 1 / 6

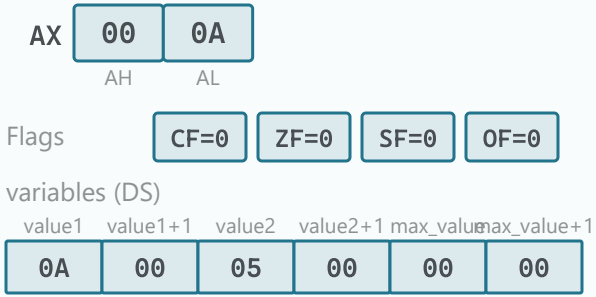
VARIABLES

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Code Trace: Find the Larger (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
▶  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



CMP subtracts but **does not store** the result; it only sets the flags: 10 − 5.

Step 2 / 6

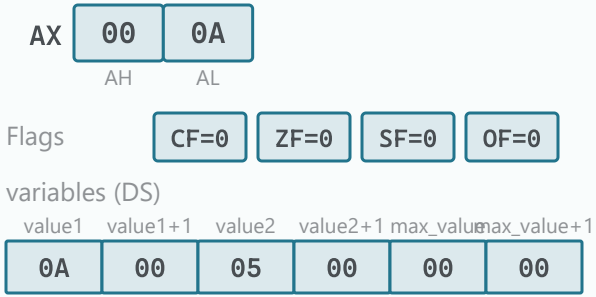
VARIABLES

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Code Trace: Find the Larger (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
▶  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



JGE (signed ≥): SF = OF, so it **jumps** → greater_or_equal.

Step 3 / 6

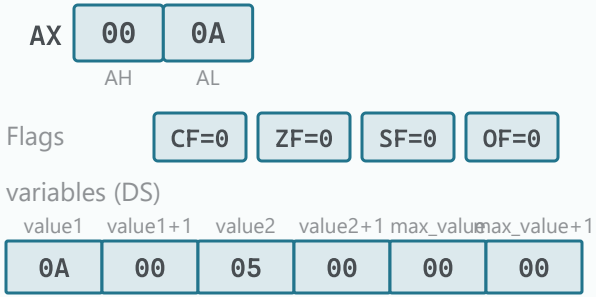
VARIABLES

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Code Trace: Find the Larger (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
▶  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



AX = value1 (it already was; the line is redundant but harmless).

Step 4 / 6

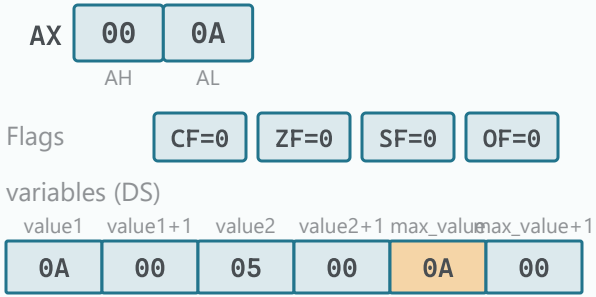
VARIABLES

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 0000h`

Code Trace: Find the Larger (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
▶  MOV max_value, AX   ; store result
11 RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



max_value ← AX = **000Ah**.

Step 5 / 6

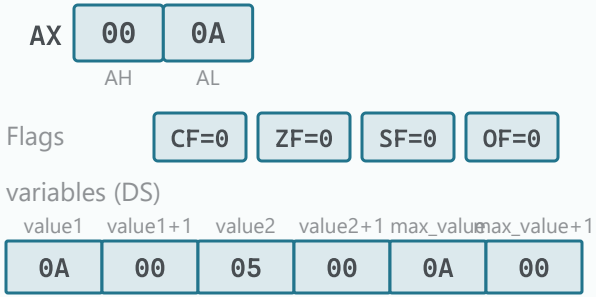
VARIABLES

AX = **000Ah** | CF = **0** | ZF = **0** | SF = **0** | OF = **0**

value1 = **000Ah** | value2 = **0005h** | max_value = **000Ah**

Code Trace: Find the Larger (10, 5)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
▶  RET
12 value1 DW 10
13 value2 DW 5
14 max_value DW ?
```



RET: back to DOS. Without RET the CPU would execute the bytes of value1 as instructions.

Step 6 / 6

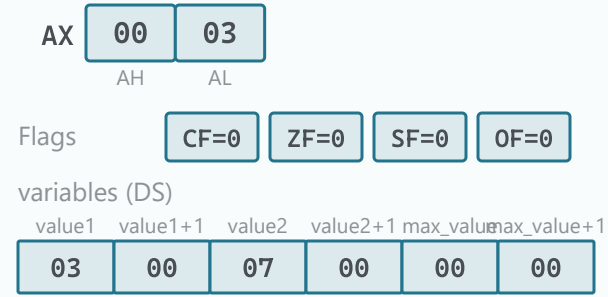
VARIABLES

`AX = 000Ah` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 000Ah` `value2 = 0005h` `max_value = 000Ah`

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
▶ MOV AX, value1      ; AX ← value1
3  CMP AX, value2     ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2     ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX  ; store result
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



AX = value1 = **0003h**.

Step 1 / 7

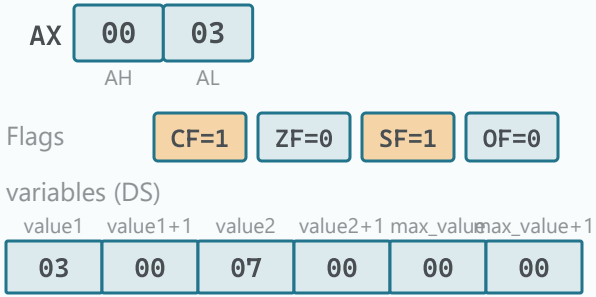
VARIABLES

`AX = 0003h` `CF = 0` `ZF = 0` `SF = 0` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
▶  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



CMP subtracts but **does not store** the result; it only sets the flags: 3 − 7.

Step 2 / 7

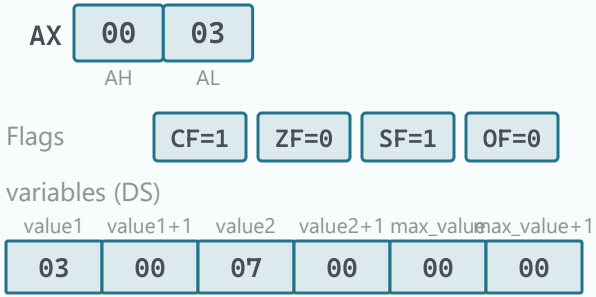
VARIABLES

`AX = 0003h` `CF = 1` `ZF = 0` `SF = 1` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
▶  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



SF ≠ OF → the condition fails, **no jump**; execution continues with the next instruction.

Step 3 / 7

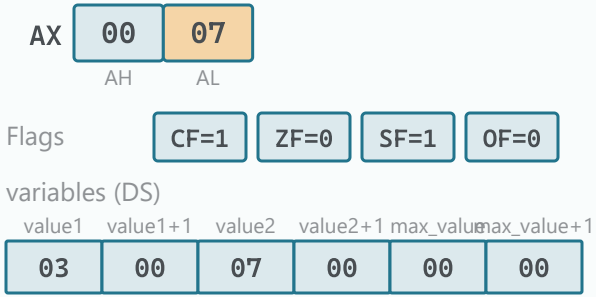
VARIABLES

`AX = 0003h` `CF = 1` `ZF = 0` `SF = 1` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
▶  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



AX = value2 = **0007h** (the larger one).

Step 4 / 7

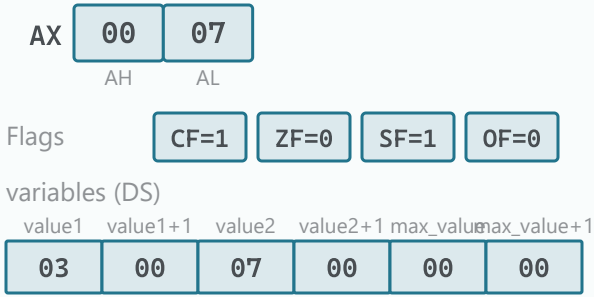
VARIABLES

AX = 0007h | CF = 1 | ZF = 0 | SF = 1 | OF = 0

value1 = 0003h | value2 = 0007h | max_value = 0000h

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
▶  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



JMP jumps unconditionally → store_max.

Step 5 / 7

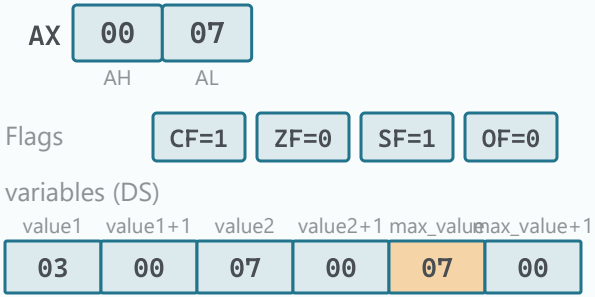
VARIABLES

`AX = 0007h` `CF = 1` `ZF = 0` `SF = 1` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0000h`

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
▶  MOV max_value, AX   ; store result
11 RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



max_value ← AX = **0007h**.

Step 6 / 7

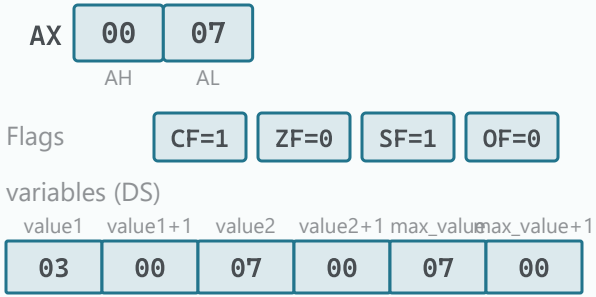
VARIABLES

AX = **0007h** | CF = **1** | ZF = **0** | SF = **1** | OF = **0**

value1 = **0003h** | value2 = **0007h** | max_value = **0007h**

Code Trace: Find the Larger (3, 7)

```
1  ORG 100h
2  MOV AX, value1      ; AX ← value1
3  CMP AX, value2      ; AX − value2
4  JGE greater_or_equal
5  MOV AX, value2      ; if less,
   value2
6  JMP store_max
7  greater_or_equal:
8  MOV AX, value1
9  store_max:
10 MOV max_value, AX   ; store result
▶  RET
12 value1 DW 3
13 value2 DW 7
14 max_value DW ?
```



RET: back to DOS. Without RET the CPU would execute the bytes of value1 as instructions.

Step 7 / 7

VARIABLES

`AX = 0007h` `CF = 1` `ZF = 0` `SF = 1` `OF = 0`

`value1 = 0003h` `value2 = 0007h` `max_value = 0007h`

Notes on Comparison

CMP

`CMP a, b` computes `a − b` and discards the result; only the flags (CF, ZF, SF, OF, PF, AF) are updated.

Signed / unsigned

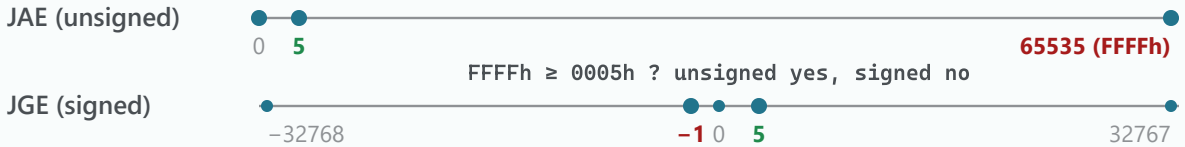
`JGE`, `JG`, `JL`: signed (SF, OF). `JAE`, `JA`, `JB`: unsigned (CF). With value1 = −1 (FFFFh) and value2 = 5, JGE does not jump but JAE does.

Why RET?

The data definitions come right after the code; without `RET` the CPU would execute the bytes `0A 00 05 00…` as instructions. Fix: `RET` (or a `jmp start` at the top).

Improvement

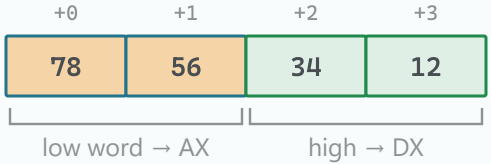
The `MOV AX, value1` under greater_or_equal is redundant; AX already holds value1. The jump can go straight to store_max.



Doubleword Variables

```
org 100h
jmp start
mydouble dd 12345678h  ; double word
mywords  dw 5678h      ; 2 separate words
         dw 1234h
mybytes  db 78h        ; 4 separate bytes
         db 56h
         db 34h
         db 12h
data dd 00010010000110100010101100111000b
start:  ; load data into DX:AX
mov ax, word ptr data
mov dx, word ptr [data+2]
ret
```

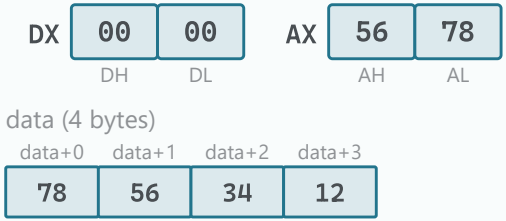
All four definitions produce **the same** bytes in memory:



- `DD` is 32 bits = 4 bytes; the lowest byte is at the lowest address.
- The 8086 has no 32-bit register: the value is held in the pair **DX:AX** (DX the high word, AX the low word).

Code Trace: A 32-Bit Load into DX:AX

```
▶ mov ax, word ptr data       ; data+0
2 mov dx, word ptr [data+2]   ; data+2
3 ret
4 data dd 12345678h
```



The first two bytes of data (78 56) → AX = **5678h**. `word ptr` states the size explicitly, since data is a DD.

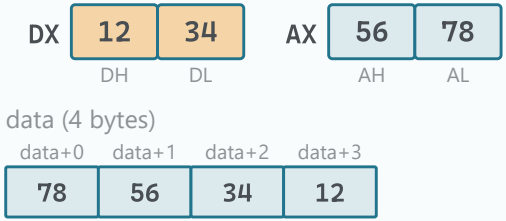
Step 1 / 2

VARIABLES

`DX = 0000h` `AX = 5678h`

Code Trace: A 32-Bit Load into DX:AX

```
1 mov ax, word ptr data       ; data+0
▶ mov dx, word ptr [data+2]   ; data+2
3 ret
4 data dd 12345678h
```



The two bytes at data+2 (34 12) → DX = **1234h**. DX:AX = **1234:5678h** = 12345678h.

Step 2 / 2

VARIABLES

DX = **1234h** AX = **5678h**

Common Mistakes with Variables

Confusing value and address

`MOV SI, arr` loads the first word; for the address use `LEA SI, arr`.

Memory-to-memory moves

There is no `MOV v1, v2`; the value goes through a register.

Indexing a word array

Element 3 is at `w[4]`: the index counts bytes.

Trusting ?

`total DW ?` may not be zero; start with `MOV total, 0`.

Forgetting the suffix

`DW 10` is decimal 10 (000Ah); for hex write `10h`.

Name clashes

`COL EQU 4` and `col DB 1` are the same name: “duplicate definition”.

Data in the path of the code

Without RET or JMP, data bytes are taken for instructions.

Forgetting the terminator

Without 0 or '$', printing runs on into the bytes that follow.

Data Definition Summary

Syntax	Meaning	Memory
`x DB 7`	a byte	`07`
`y DW 1234h`	a word	`34 12`
`z DD 12345678h`	a doubleword	`78 56 34 12`
`s DB 'Hi', 0`	string + terminator	`48 69 00`
`t DB 3 DUP(0)`	repetition	`00 00 00`
`u DW ?`	uninitialized	`?? ??` (emu8086: 00 00)
`k EQU 10`	constant	takes no space

Address: `LEA BX, x` / `MOV BX, OFFSET x`. Value: `MOV AL, x`. Changing the size: `word ptr`, `byte ptr`.

Summary

Variable

A name for a memory location; the assembler replaces the name with an offset.

DB / DW / DD

1, 2, 4 bytes; little-endian; `?` means uninitialized.

ORG 100h

.COM code starts at 0100h; the first 256 bytes are the PSP. Data goes after RET or is skipped with jmp.

Arrays

Consecutive bytes; `a[i]` is a byte offset; DUP repeats; in a DW array the index is 2 × i.

LEA / OFFSET

Get an address; LEA also computes indexed addresses at run time.

EQU

An assembly-time constant; takes no memory and cannot change.

SUMMARY

Exercises

1. If `x DB 5, 6` · `y DW 0ABCDh` · `z DB 'OK'` start at 0200h, give the address and value of every byte.
2. How many bytes does `table DW 10 DUP(?)` take? What is the offset of the 7th element (index 6)?
3. For `y` above, what is AL after `MOV AL, BYTE PTR y` and after `MOV AL, BYTE PTR y+1`?
4. What is the difference between `LEA SI, a[BX]` and `MOV SI, OFFSET a`?
5. Write code that stores the sum of two DW variables in a third and sets `ovf DB 1` if the sum overflows (CF = 1).
6. Which pair of instructions adds 1 to `dd 0FFFFFFFFh` held in DX:AX?

▶ **Answers**

References

- emu8086 documentation, *8086 assembler tutorial for beginners*, part 3: variables (DB, DW, DUP, LEA, OFFSET, EQU).
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson.
- Kip R. Irvine, *Assembly Language for x86 Processors*, Pearson (data definition directives).
- Microsoft MASM 6.1 Programmer's Guide: DB/DW/DD, DUP, EQU, OFFSET.

The End

Chapter 4.2: Variables

Previous: Chapter 4.1: Memory Access · Next: Chapter 5.1: Interrupts