Chapter 9.1: Macros

Microprocessors

What a macro is | MACRO / ENDM | Expansion | Parameter pitfalls | Macros vs procedures | Code size | LOCAL | PUTC | POWER | emu8086.inc

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_09_01_Macros.html

Learning Outcomes

Macro definition

Write parameterized macros with `MACRO` / `ENDM`.

Expansion

Trace how the assembler expands a macro call as text.

Macro or procedure?

Compare the two in size, speed and usage, and choose the right one.

LOCAL

Avoid the *Duplicate declaration* error in macros that contain labels.

Practice

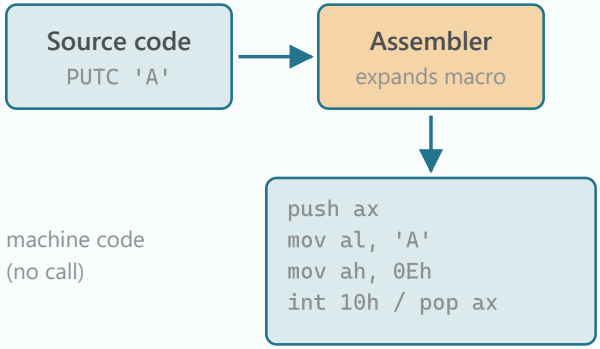
Step through the PUTC (print a character) and POWER (exponentiation) macros.

Pitfalls

Spot parameter side effects, clobbered registers and edge cases.

Macros

- Macros **resemble procedures**: they give a name to a repeated piece of code.
- But they are not called at run time; they are placeholders replaced by real instructions **at assembly time**.
- Macros that are never used are ignored: they add no bytes to the machine code.
- Unlike procedures, macros must be defined **above the code that uses them**; the assembler cannot expand a call to a macro it does not yet know.



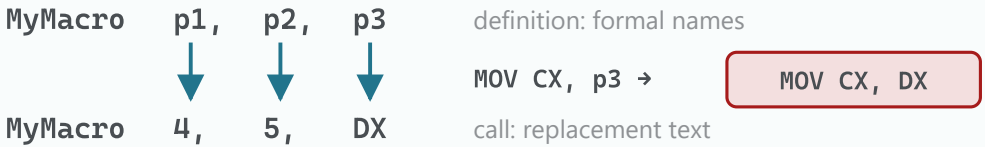
Defining a Macro

```
name    MACRO  [parametreler,...]
        <instructions>
        ENDM
```

name
The macro's name, used like an instruction.

parameters
Comma-separated formal names (optional). The values in the call replace them **as text**.

ENDM
Ends the macro body. Unlike a procedure, the name is not repeated.



Parameters may be registers, memory or immediates: the assembler only substitutes text; whether the resulting instruction is valid is checked **after expansion**.

Macro Expansion Step by Step

```
MyMacro  MACRO  p1, p2, p3
         MOV AX, p1
         MOV BX, p2
         MOV CX, p3
ENDM
```

First the **definition**: the assembler learns MyMacro and its parameters p1, p2, p3; no code is generated yet.

Macro Expansion Step by Step

```
MyMacro 1, 2, 3
```

```
MOV AX, p1
MOV BX, p2
MOV CX, p3
```

`MyMacro 1, 2, 3` : p1 = 1, p2 = 2, p3 = 3.

Macro Expansion Step by Step

```
MyMacro 1, 2, 3
```

```
MOV AX, 1
MOV BX, p2
MOV CX, p3
```

Line 1: `MOV AX, p1` → `MOV AX, 1` .

Macro Expansion Step by Step

```
MyMacro 1, 2, 3
```

```
MOV AX, 1
MOV BX, 2
MOV CX, p3
```

Line 2: `MOV BX, p2` → `MOV BX, 2` .

Macro Expansion Step by Step

```
MyMacro 1, 2, 3
```

```
MOV AX, 1
MOV BX, 2
MOV CX, 3
```

Line 3: `MOV CX, p3` → `MOV CX, 3`.

Macro Expansion Step by Step

```
MyMacro 4, 5, DX
```

```
MOV AX, p1
MOV BX, p2
MOV CX, p3
```

`MyMacro 4, 5, DX` : p1 = 4, p2 = 5, p3 = DX.

Macro Expansion Step by Step

```
MyMacro 4, 5, DX
```

```
MOV AX, 4
MOV BX, p2
MOV CX, p3
```

Line 1: `MOV AX, p1` → `MOV AX, 4`.

Macro Expansion Step by Step

```
MyMacro 4, 5, DX
```

```
MOV AX, 4
MOV BX, 5
MOV CX, p3
```

Line 2: `MOV BX, p2` → `MOV BX, 5`.

Macro Expansion Step by Step

```
MyMacro 4, 5, DX
```

```
MOV AX, 4
MOV BX, 5
MOV CX, DX
```

Line 3: `MOV CX, p3` → `MOV CX, DX` .

Macro Expansion Step by Step

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
RET
```

Result (as emu8086 shows it): the two calls expanded into **6 instructions**. Constants are shown in hex.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h   ; MyMacro 1, 2, 3
3 MOV BX, 00002h
4 MOV CX, 00003h
5 MOV AX, 00004h   ; MyMacro 4, 5, DX
6 MOV BX, 00005h
7 MOV CX, DX
8 RET
```

After loading, CX = **0012h**: the program is 5 × 3 + 2 + 1 = 18 bytes. SP = FFFEh.

Step 1 / 8

VARIABLES

AX = **0000h** BX = **0000h** CX = **0012h**

DX = **0000h** SP = **FFFEh**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
▶ MOV AX, 00001h   ; MyMacro 1, 2, 3
3 MOV BX, 00002h
4 MOV CX, 00003h
5 MOV AX, 00004h   ; MyMacro 4, 5, DX
6 MOV BX, 00005h
7 MOV CX, DX
8 RET
```

AX = 0001h.

Step 2 / 8

VARIABLES

AX = **0001h** | BX = **0000h** | CX = **0012h** | DX = **0000h** | SP = **FFFEh**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h    ; MyMacro 1, 2, 3
▶ MOV BX, 00002h
4 MOV CX, 00003h
5 MOV AX, 00004h    ; MyMacro 4, 5, DX
6 MOV BX, 00005h
7 MOV CX, DX
8 RET
```

BX = 0002h.

Step 3 / 8

VARIABLES

AX = **0001h** | BX = **0002h** | CX = **0012h** | DX = **0000h** | SP = **FFFEh**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h    ; MyMacro 1, 2, 3
3 MOV BX, 00002h
▶ MOV CX, 00003h
5 MOV AX, 00004h    ; MyMacro 4, 5, DX
6 MOV BX, 00005h
7 MOV CX, DX
8 RET
```

CX = 0003h. The first call is done; there is no CALL/RET.

Step 4 / 8

VARIABLES

AX = **0001h** BX = **0002h** CX = **0003h**

DX = **0000h** SP = **FFFEh**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h   ; MyMacro 1, 2, 3
3 MOV BX, 00002h
4 MOV CX, 00003h
▶ MOV AX, 00004h   ; MyMacro 4, 5, DX
6 MOV BX, 00005h
7 MOV CX, DX
8 RET
```

AX = 0004h.

Step 5 / 8

VARIABLES

AX = **0004h** BX = **0002h** CX = **0003h**

DX = **0000h** SP = **FFFEh**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h   ; MyMacro 1, 2, 3
3 MOV BX, 00002h
4 MOV CX, 00003h
5 MOV AX, 00004h   ; MyMacro 4, 5, DX
▶ MOV BX, 00005h
7 MOV CX, DX
8 RET
```

BX = 0005h.

Step 6 / 8

VARIABLES

AX = **0004h** | BX = **0005h** | CX = **0003h** | DX = **0000h** | SP = **FFFEh**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h   ; MyMacro 1, 2, 3
3 MOV BX, 00002h
4 MOV CX, 00003h
5 MOV AX, 00004h   ; MyMacro 4, 5, DX
6 MOV BX, 00005h
▶ MOV CX, DX
8 RET
```

CX ← DX = 0000h (DX starts at zero).

Step 7 / 8

VARIABLES

AX = **0004h**	BX = **0005h**	CX = **0000h**
DX = **0000h**	SP = **FFFEh**	

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Running the Expanded Code

```
1 ORG 100h
2 MOV AX, 00001h   ; MyMacro 1, 2, 3
3 MOV BX, 00002h
4 MOV CX, 00003h
5 MOV AX, 00004h   ; MyMacro 4, 5, DX
6 MOV BX, 00005h
7 MOV CX, DX
▶ RET
```

RET: the program ends.

Step 8 / 8

VARIABLES

AX = **0004h** BX = **0005h** CX = **0000h**

DX = **0000h** SP = **0000h**

SP never changed: a macro does not use the stack (unless its body has PUSH/POP). In a procedure, CALL/RET would change SP.

Pitfall: Parameters Are Text

```
▶ ORG 100h
2 MOV  AX, 1111h
3 MOV  BX, 2222h
4 MOV  AX, BX  ; MyMacro BX,AX,0
5 MOV  BX, AX  ; p2 = AX (bad!)
6 MOV  CX, 0
7 RET
```

The call `MyMacro BX, AX, 0` is meant to do AX ← BX, BX ← AX (a swap).

Step 1 / 7

VARIABLES

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

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Pitfall: Parameters Are Text

```
1 ORG 100h
▶ MOV  AX, 1111h
3 MOV  BX, 2222h
4 MOV  AX, BX  ; MyMacro BX,AX,0
5 MOV  BX, AX  ; p2 = AX (bad!)
6 MOV  CX, 0
7 RET
```

AX = 1111h.

Step 2 / 7

VARIABLES

AX = 1111h | BX = 0000h | CX = 0000h

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Pitfall: Parameters Are Text

```
1 ORG 100h
2 MOV  AX, 1111h
▶ MOV  BX, 2222h
4 MOV  AX, BX  ; MyMacro BX,AX,0
5 MOV  BX, AX  ; p2 = AX (bad!)
6 MOV  CX, 0
7 RET
```

BX = 2222h.

Step 3 / 7

VARIABLES

AX = **1111h** BX = **2222h** CX = **0000h**

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Pitfall: Parameters Are Text

```
1 ORG 100h
2 MOV  AX, 1111h
3 MOV  BX, 2222h
▶ MOV  AX, BX  ; MyMacro BX,AX,0
5 MOV  BX, AX  ; p2 = AX (bad!)
6 MOV  CX, 0
7 RET
```

Line 1 of the macro: `MOV AX, p1` → `MOV AX, BX` → AX = 2222h.

Step 4 / 7

VARIABLES

AX = **2222h** | BX = **2222h** | CX = **0000h**

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Pitfall: Parameters Are Text

```
1 ORG 100h
2 MOV  AX, 1111h
3 MOV  BX, 2222h
4 MOV  AX, BX  ; MyMacro BX,AX,0
▶ MOV  BX, AX  ; p2 = AX (bad!)
6 MOV  CX, 0
7 RET
```

Line 2: `MOV BX, AX`. The programmer expected the "old AX" (1111h), but AX changed on the line before → BX = **2222h**.

Step 5 / 7

VARIABLES

AX = **2222h** BX = **2222h** CX = **0000h**

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Pitfall: Parameters Are Text

```
1 ORG 100h
2 MOV  AX, 1111h
3 MOV  BX, 2222h
4 MOV  AX, BX  ; MyMacro BX,AX,0
5 MOV  BX, AX  ; p2 = AX (bad!)
▶ MOV  CX, 0
7 RET
```

CX = 0.

Step 6 / 7

VARIABLES

AX = **2222h** BX = **2222h** CX = **0000h**

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Pitfall: Parameters Are Text

```
1 ORG 100h
2 MOV  AX, 1111h
3 MOV  BX, 2222h
4 MOV  AX, BX  ; MyMacro BX,AX,0
5 MOV  BX, AX  ; p2 = AX (bad!)
6 MOV  CX, 0
▶ RET
```

Result: AX = BX = 2222h. 1111h is lost.

Step 7 / 7

VARIABLES

AX = **2222h** BX = **2222h** CX = **0000h**

A macro parameter is not a variable but replacement text: its value is read on the line where it is used. Do not pass as parameters registers that the body changes.

Parameter Kinds: Same Macro, Different Expansions

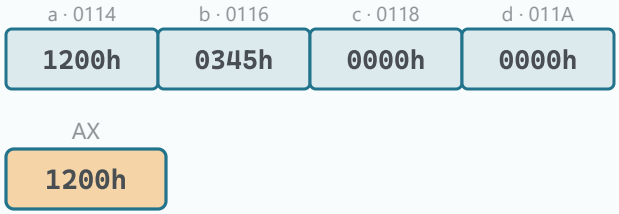
```
ADD3 MACRO x, y, z
  MOV AX, x
  ADD AX, y
  MOV z, AX
ENDM
```

Call	Expansion	Status
`ADD3 a, b, c`	`MOV AX, a` · `ADD AX, b` · `MOV c, AX`	valid
`ADD3 BX, 5, CX`	`MOV AX, BX` · `ADD AX, 5` · `MOV CX, AX`	valid
`ADD3 a, b, 7`	`MOV 7, AX`	dest can't be immediate
`ADD3 k, b, c` (k DB)	`MOV AX, k`	size mismatch
`ADD3 a, b`	`MOV , AX`	missing operand

- A parameter may be a register, a variable, an immediate or an expression; the expanded instruction, not the macro, decides what is valid.
- The error is reported on the line that uses the macro, but its cause is a line in the body: picture the expansion to understand the message.
- A missing parameter is replaced by empty text. In MASM, `z:REQ` makes a parameter required and `y:=<1>` gives it a default value.

Code Trace: Two Expansions of ADD3

```
1  org 100h
▶ mov ax, a       ; ADD3 a, b, c
3  add ax, b
4  mov c, ax
5  mov ax, c      ; ADD3 c, 5, d
6  add ax, 5
7  mov d, ax
8  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



1st expansion `ADD3 a, b, c` : x was replaced by a → AX = **1200h**.

Step 1 / 7

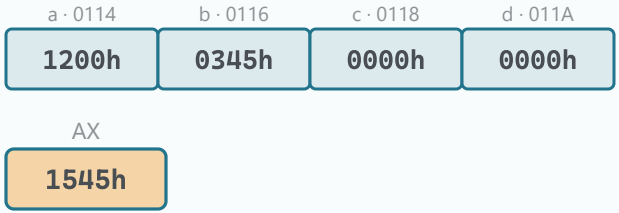
VARIABLES

AX = **1200h** | c = **0000h** | d = **0000h**

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Code Trace: Two Expansions of ADD3

```
1  org 100h
2  mov ax, a       ; ADD3 a, b, c
▶  add ax, b
4  mov c, ax
5  mov ax, c       ; ADD3 c, 5, d
6  add ax, 5
7  mov d, ax
8  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



`ADD AX, b` : 1200h + 0345h = **1545h**.

Step 2 / 7

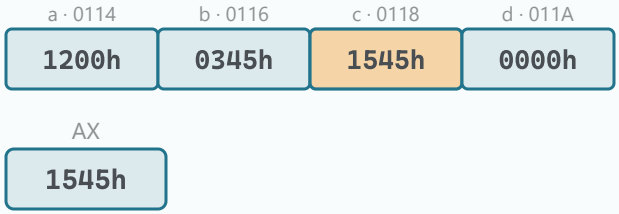
VARIABLES

AX = 1545h | c = 0000h | d = 0000h

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Code Trace: Two Expansions of ADD3

```
1  org 100h
2  mov ax, a       ; ADD3 a, b, c
3  add ax, b
▶  mov c, ax
5  mov ax, c       ; ADD3 c, 5, d
6  add ax, 5
7  mov d, ax
8  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



`MOV c, AX` : c = 1545h. There is no memory-to-memory add, so the macro uses AX as a go-between.

Step 3 / 7

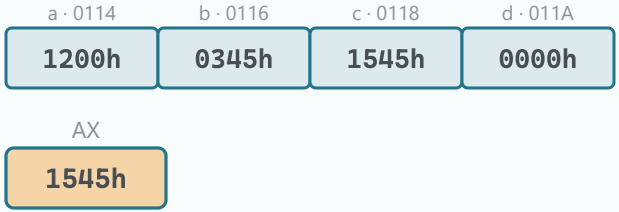
VARIABLES

AX = 1545h | c = 1545h | d = 0000h

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Code Trace: Two Expansions of ADD3

```
1  org 100h
2  mov ax, a       ; ADD3 a, b, c
3  add ax, b
4  mov c, ax
▶  mov ax, c       ; ADD3 c, 5, d
6  add ax, 5
7  mov d, ax
8  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



2nd expansion `ADD3 c, 5, d` : the same body, different text. x = c → AX = 1545h.

Step 4 / 7

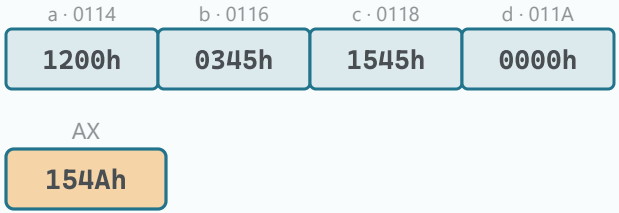
VARIABLES

`AX = 1545h` `c = 1545h` `d = 0000h`

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Code Trace: Two Expansions of ADD3

```
1  org 100h
2  mov ax, a       ; ADD3 a, b, c
3  add ax, b
4  mov c, ax
5  mov ax, c       ; ADD3 c, 5, d
▶  add ax, 5
7  mov d, ax
8  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



This time y = 5 is an immediate: `ADD AX, 5` → AX = **154Ah**.

Step 5 / 7

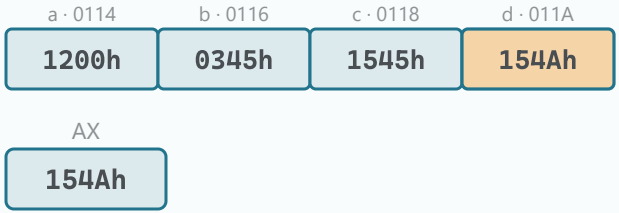
VARIABLES

AX = 154Ah | c = 1545h | d = 0000h

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Code Trace: Two Expansions of ADD3

```
1  org 100h
2  mov ax, a       ; ADD3 a, b, c
3  add ax, b
4  mov c, ax
5  mov ax, c       ; ADD3 c, 5, d
6  add ax, 5
▶  mov d, ax
8  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



`MOV d, AX` : d = 154Ah. The two expansions produced 6 instructions; the macro clobbered AX, and the caller must know it.

Step 6 / 7

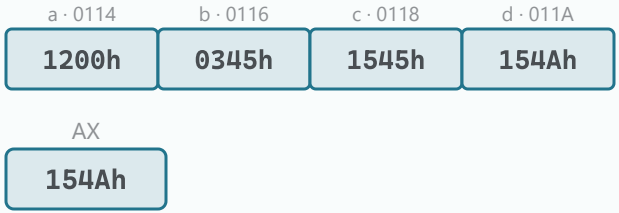
VARIABLES

AX = 154Ah | c = 1545h | d = 154Ah

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Code Trace: Two Expansions of ADD3

```
1  org 100h
2  mov ax, a       ; ADD3 a, b, c
3  add ax, b
4  mov c, ax
5  mov ax, c       ; ADD3 c, 5, d
6  add ax, 5
7  mov d, ax
▶  ret
9  a dw 1200h
10 b dw 0345h
11 c dw ?
12 d dw ?
```



RET. Code (3 + 4 + 3) + (3 + 3 + 3) = 19 bytes + RET = 20 bytes; `ADD AX, 5` is 3 bytes in the short AX form. Every use of ADD3 writes the body again.

Step 7 / 7

VARIABLES

AX = **154Ah** | c = **1545h** | d = **154Ah**

Macro: ADD3 MACRO x, y, z / MOV AX, x / ADD AX, y / MOV z, AX / ENDM. The code is the expanded form of the two calls.

Parameter Expressions and Operator Precedence

```
AREA MACRO wd, ht
  MOV AX, wd*ht
ENDM

DIFF MACRO p, q
  MOV CX, p-q
ENDM
```

- A parameter is text, not a value: operator precedence is applied again on the line where the expression is substituted.
- `AREA 2+3, 4` : the multiplication is done first, so AX = 14; 20 was expected.
- Fix: put the parameters in parentheses in the body: `(wd)*(ht)` , `(p)-(q)` .

Call	Expansion	Result
`AREA 2+3, 4`	`MOV AX, 2+3*4`	AX = 000Eh (14)
fixed `AREA 2+3, 4`	`MOV AX, (2+3)*(4)`	AX = 0014h (20)
`DIFF 10, 4+1`	`MOV CX, 10-4+1`	CX = 0007h
`AREA BX, 2`	`MOV AX, BX*2`	error: * with constants only

The expression is evaluated at assembly time; an expression with a register cannot be multiplied at run time. Multiplying a register needs MUL or SHL.

Interactive Macro Expander

Macro: MyMacro p1, p2, p3 | 1.: 1 | 2.: 2 | 3.: 3 | number of uses: 1

Call

```
MyMacro 1, 2, 3
```

Code generated by the assembler

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
```

One copy ≈ **9 bytes**. 1 uses: macro ≈ **9** bytes, procedure ≈ 10 + 1 × 3 = **13** bytes.

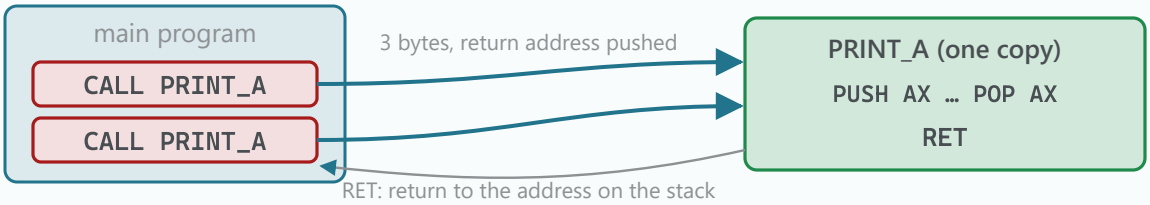
Size comparison: as a procedure, the body + RET would be written once, and each use would be just a 3-byte CALL.

Procedure Features

- A procedure is called with the `CALL` instruction.
- A procedure exists as a **single copy** at a specific address in memory.
- When it is called, the CPU transfers control there; `RET` returns control to the main program.
- The **stack** holds the return address.
- Calling the same procedure 100 times barely grows the file: a near `CALL` takes only **3 bytes**.
- Parameters are passed on the stack or in registers.
- A procedure ends with its name and the `ENDP` directive.

```
PRINT_A PROC
        PUSH AX
        MOV  AL, 'A'
        MOV  AH, 0Eh
        INT  10h
        POP  AX
        RET
PRINT_A ENDP

        CALL PRINT_A  ; 3 bytes
        CALL PRINT_A  ; 3 bytes
```

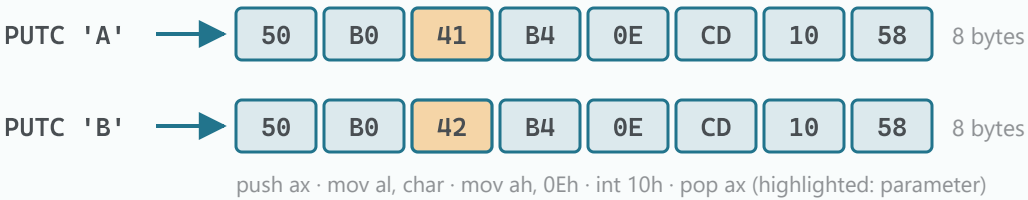


Macro Features

- A macro is used just by **writing its name** (no CALL).
- Macros are **expanded in place** in the program code.
- Use the same macro 100 times and the assembler expands it 100 times; the executable keeps growing.
- Parameters are written after the macro name.
- The `ENDM` directive alone ends a macro.
- If a macro definition contains labels and the macro is used more than once, you can get a **Duplicate declaration** error; use the `LOCAL` directive for variable, label and procedure names.

```
PUTC MACRO char
     PUSH AX
     MOV  AL, char
     MOV  AH, 0Eh
     INT  10h
     POP  AX
ENDM

     PUTC 'A'  ; 8 bytes
     PUTC 'B'  ; 8 bytes
```

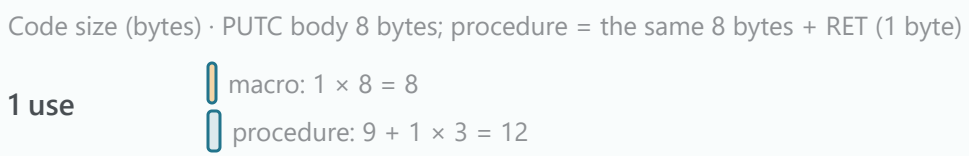


Macro or Procedure?

Criterion	Macro	Procedure
When is it processed?	Assembly time (text substitution)	Run time (CALL/RET)
Usage	`PUTC 'A'`	`CALL PRINT_A`
Code size	Grows by the body at every use	Body once + 3 bytes per call
Speed	Faster (no CALL/RET, no stack access)	CALL + RET cost on every call
Parameters	Values after the name, as text	Registers or the stack
Stack	Not used (unless the body uses it)	Used for the return address
Where defined	**Before** use	Anywhere
End	`ENDM`	`name ENDP`
Labels	`LOCAL` needed	No problem (one copy)

Rule: short, frequently used, speed-critical code → macro; long code used in many places → procedure. The two can be combined: a macro can put its parameters in registers and call a procedure.

Code Size: 1, 10 and 100 Uses

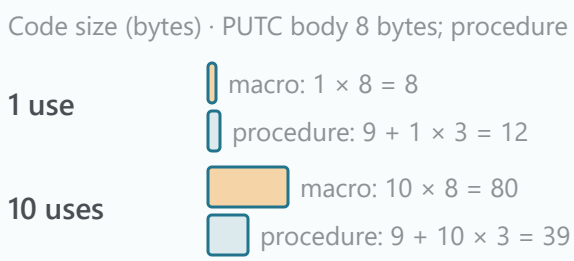


1 use: macro 8 bytes; procedure 8 bytes of body + 1 byte RET + 3 bytes CALL = 12 bytes. Here the macro is **smaller**.

Step 1 / 3

For speed it is the other way round: each CALL + RET pair adds about 19 + 8 = 27 clocks and two stack accesses on the 8086. A macro has no such cost.

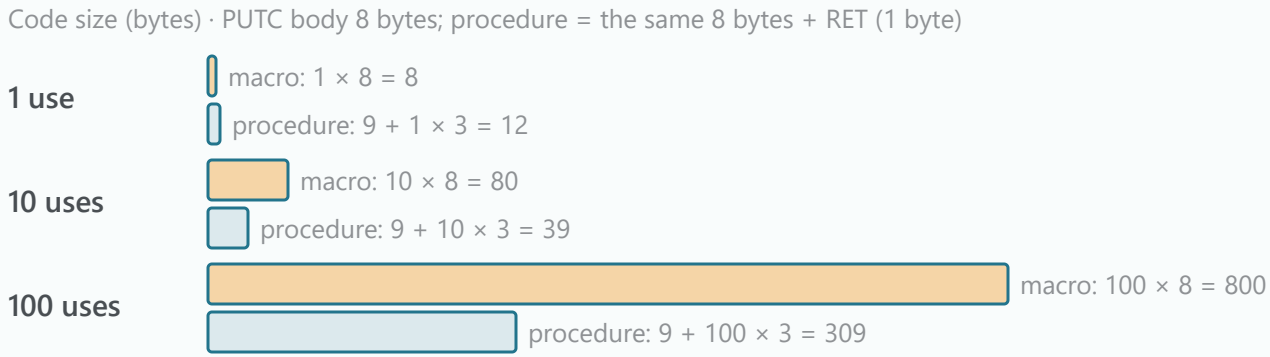
Code Size: 1, 10 and 100 Uses



10 uses: macro 80 bytes; procedure 9 + 30 = 39 bytes.

For speed it is the other way round: each CALL + RET pair adds about 19 + 8 = 27 clocks and two stack accesses on the 8086. A macro has no such cost.

Code Size: 1, 10 and 100 Uses



100 uses: macro 800 bytes; procedure 9 + 300 = 309 bytes. The gap grows with the number of uses.

For speed it is the other way round: each CALL + RET pair adds about 19 + 8 = 27 clocks and two stack accesses on the 8086. A macro has no such cost.

Decision Exercise: Macro or Procedure?

Five cases: in each, weigh code size (how many copies?) against speed (the CALL/RET cost).

Decision Exercise: Macro or Procedure?

Five cases: in each, weigh code size (how many copies?) against speed (the CALL/RET cost).

1. **Print one character, in 200 places** → a short body (8 bytes) but 200 copies = 1600 bytes. A procedure: 9 + 600 = 609 bytes. If size matters, **procedure**; if speed matters, macro.

Decision Exercise: Macro or Procedure?

Five cases: in each, weigh code size (how many copies?) against speed (the CALL/RET cost).

1. **Print one character, in 200 places** → a short body (8 bytes) but 200 copies = 1600 bytes. A procedure: 9 + 600 = 609 bytes. If size matters, **procedure**; if speed matters, macro.
2. **A one-line job swapping two registers** → `XCHG` is already one instruction; a macro is enough for readability, a procedure is needless overhead (**macro**).

Decision Exercise: Macro or Procedure?

Five cases: in each, weigh code size (how many copies?) against speed (the CALL/RET cost).

1. **Print one character, in 200 places** → a short body (8 bytes) but 200 copies = 1600 bytes. A procedure: 9 + 600 = 609 bytes. If size matters, **procedure**; if speed matters, macro.
2. **A one-line job swapping two registers** → `XCHG` is already one instruction; a macro is enough for readability, a procedure is needless overhead (**macro**).
3. **A 300-byte sorting algorithm, in 3 places** → macro 900 bytes, procedure 300 + 9 bytes: clearly a **procedure**.

Decision Exercise: Macro or Procedure?

Five cases: in each, weigh code size (how many copies?) against speed (the CALL/RET cost).

1. **Print one character, in 200 places** → a short body (8 bytes) but 200 copies = 1600 bytes. A procedure: 9 + 600 = 609 bytes. If size matters, **procedure**; if speed matters, macro.
2. **A one-line job swapping two registers** → `XCHG` is already one instruction; a macro is enough for readability, a procedure is needless overhead (**macro**).
3. **A 300-byte sorting algorithm, in 3 places** → macro 900 bytes, procedure 300 + 9 bytes: clearly a **procedure**.
4. **A 2-instruction job done on every pass of a tight loop** → the CALL/RET cost exceeds the job itself: **macro**.

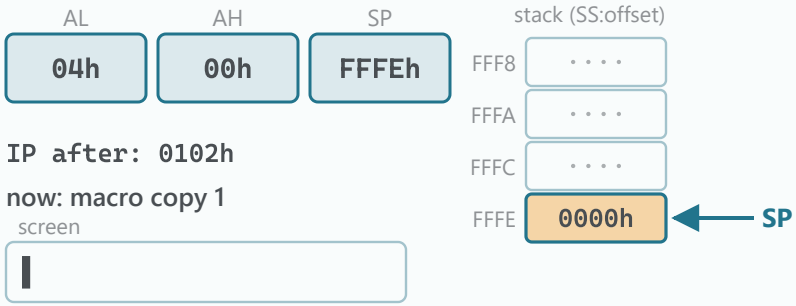
Decision Exercise: Macro or Procedure?

Five cases: in each, weigh code size (how many copies?) against speed (the CALL/RET cost).

1. **Print one character, in 200 places** → a short body (8 bytes) but 200 copies = 1600 bytes. A procedure: 9 + 600 = 609 bytes. If size matters, **procedure**; if speed matters, macro.
2. **A one-line job swapping two registers** → `XCHG` is already one instruction; a macro is enough for readability, a procedure is needless overhead (**macro**).
3. **A 300-byte sorting algorithm, in 3 places** → macro 900 bytes, procedure 300 + 9 bytes: clearly a **procedure**.
4. **A 2-instruction job done on every pass of a tight loop** → the CALL/RET cost exceeds the job itself: **macro**.
5. **Put the parameters in registers and call shared code** → both together: macro `SHOW_NUM x` → `MOV AX, x` + `CALL print_num` (emu8086.inc works this way too).

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
▶  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
11 ret
```



Macro `PUT_DIGIT d` expands at assembly time into `mov al, d` + `call print_digit`. The work is in the procedure, a **single copy**. First copy: AL = **04h**.

Step 1 / 9

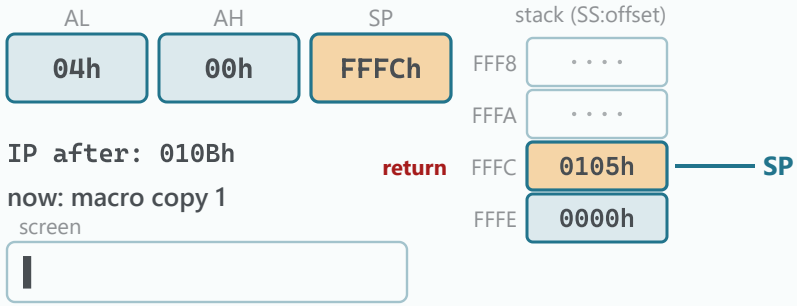
VARIABLES

`AL = 04h` `AH = 00h` `SP = FFFEh` `IP = 0102h`

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
▶  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
11 ret
```



CALL pushes the return address (the instruction after the CALL, **0105h**): SP = FFFCh. IP ← print_digit.

Step 2 / 9

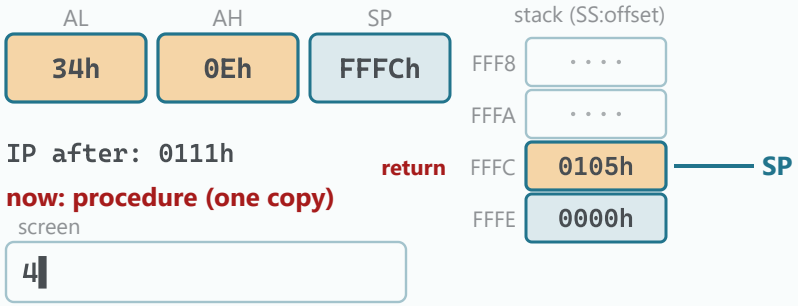
VARIABLES

AL = 04h | AH = 00h | SP = FFFCh | IP = 010Bh

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
▶  add  al, '0'      ; digit → ASCII
▶  mov  ah, 0Eh
▶  int  10h
11 ret
```



Procedure body: AL + 30h = **34h** ('4'), AH = 0Eh, INT 10h prints **4**.

Step 3 / 9

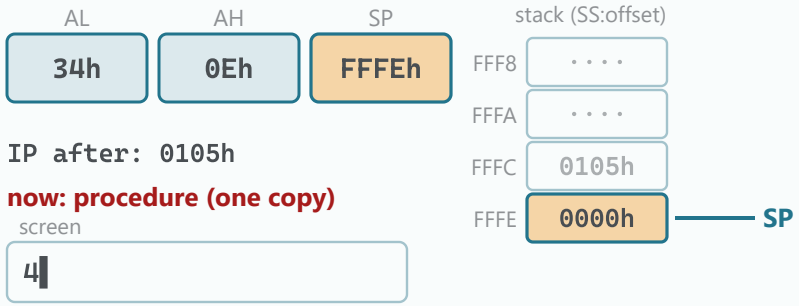
VARIABLES

`AL = 34h` `AH = 0Eh` `SP = FFFCh` `IP = 0111h`

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
▶  ret
```



RET pops 0105h: SP = FFFEh; execution resumes after the macro's first copy.

Step 4 / 9

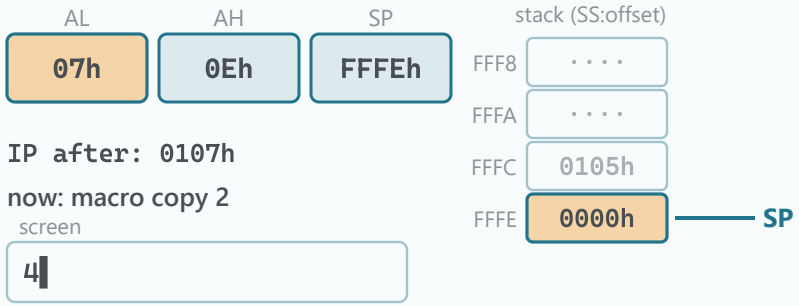
VARIABLES

AL = 34h | AH = 0Eh | SP = FFFEh | IP = 0105h

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
▶  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
11 ret
```



The macro's second copy: AL = **07h**. Each copy is only 5 bytes (2-byte MOV + 3-byte CALL).

Step 5 / 9

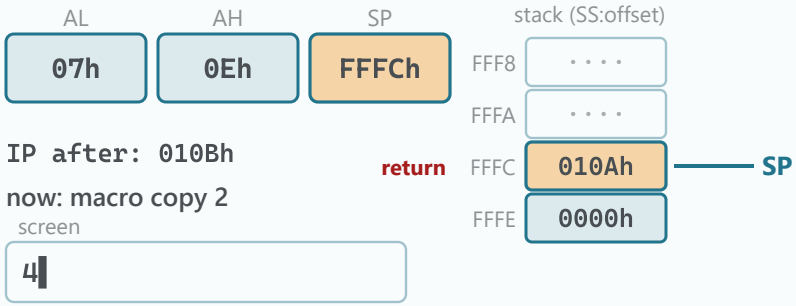
VARIABLES

AL = 07h | AH = 0Eh | SP = FFFEh | IP = 0107h

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
▶  call print_digit
6  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
11 ret
```



CALL again: this time the return address is **010Ah**. Same procedure, different return address.

Step 6 / 9

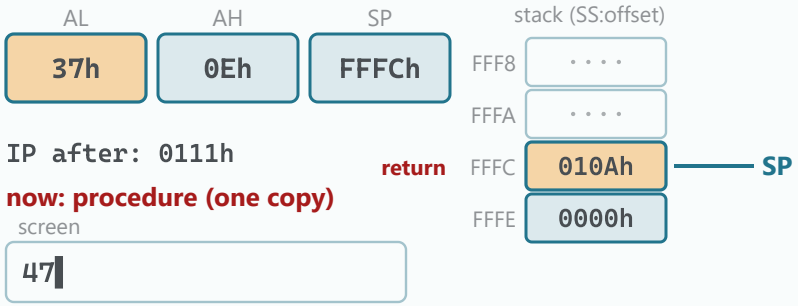
VARIABLES

`AL = 07h` `AH = 0Eh` `SP = FFFCh` `IP = 010Bh`

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
▶  add  al, '0'      ; digit → ASCII
▶  mov  ah, 0Eh
▶  int  10h
11 ret
```



The same single procedure copy runs: AL = 37h ('7'). The screen shows **47**.

Step 7 / 9

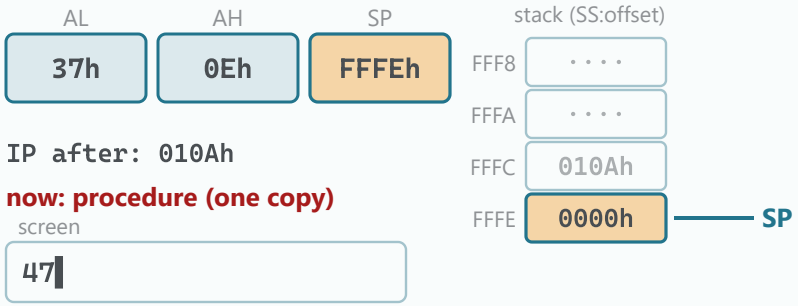
VARIABLES

`AL = 37h` `AH = 0Eh` `SP = FFFCh` `IP = 0111h`

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
6  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
▶  ret
```



RET → 010Ah. The stack is empty again; the caller's SP was preserved.

Step 8 / 9

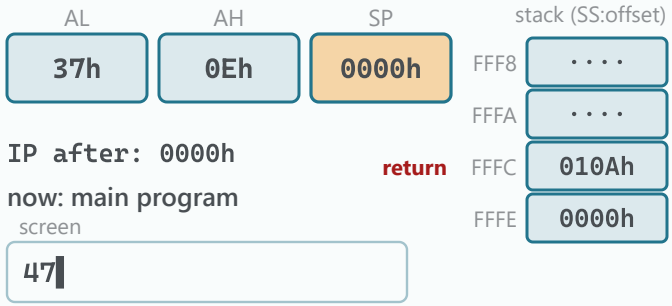
VARIABLES

`AL = 37h` `AH = 0Eh` `SP = FFFEh` `IP = 010Ah`

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Macro Sets the Parameter, Procedure Does the Work

```
1  org 100h
2  mov  al, 4        ; PUT_DIGIT 4
3  call print_digit
4  mov  al, 7        ; PUT_DIGIT 7
5  call print_digit
▶  ret
7  print_digit:
8  add  al, '0'      ; digit → ASCII
9  mov  ah, 0Eh
10 int  10h
11 ret
```



The program ends. The macro loaded the parameter; the shared code is in the procedure, just like emu8086.inc's `DEFINE_...` + macro pairs.

Step 9 / 9

VARIABLES

`AL = 37h` `AH = 0Eh` `SP = 0000h` `IP = 0000h`

Macro: `PUT_DIGIT MACRO d` / `mov al, d` / `call print_digit` / `ENDM`. 10 uses: 10 × 5 + 7 = 57 bytes; with the whole body in the macro, 10 × 8 = 80 bytes.

Code Trace: A RET Inside a Macro

```
1  org 100h
▶  mov al, 'A'     ; EMIT 'A'
3  mov ah, 0Eh
4  int 10h
5  ret             ; the macro's RET
6  mov al, 'B'     ; EMIT 'B'
7  mov ah, 0Eh
8  int 10h
9  ret
10 ret
```

AX	IP	SP	screen
0041h	0102h	FFFEh	

1st expansion `EMIT 'A'` : AL = 41h.

Step 1 / 4

VARIABLES

`AX = 0041h` `IP = 0102h` `SP = FFFEh`

OUTPUT

Macro: EMIT MACRO c / MOV AL, c / MOV AH, 0Eh / INT 10h / RET / ENDM. RET makes sense only in a procedure entered with CALL.

Code Trace: A RET Inside a Macro

```
1  org 100h
2  mov al, 'A'     ; EMIT 'A'
▶  mov ah, 0Eh
▶  int 10h
5  ret             ; the macro's RET
6  mov al, 'B'     ; EMIT 'B'
7  mov ah, 0Eh
8  int 10h
9  ret
10 ret
```

AX	IP	SP	screen
0E41h	0106h	FFFEh	A

AH = 0Eh, `INT 10h` : **A** is printed.

Step 2 / 4

VARIABLES

AX = **0E41h** IP = **0106h** SP = **FFFEh**

OUTPUT

```
A
```

Macro: EMIT MACRO c / MOV AL, c / MOV AH, 0Eh / INT 10h / RET / ENDM. RET makes sense only in a procedure entered with CALL.

Code Trace: A RET Inside a Macro

```
1  org 100h
2  mov al, 'A'     ; EMIT 'A'
3  mov ah, 0Eh
4  int 10h
▶  ret             ; the macro's RET
6  mov al, 'B'     ; EMIT 'B'
7  mov ah, 0Eh
8  int 10h
9  ret
10 ret
```

AX	IP	SP	screen
0E41h	0000h	0000h	A

0000h from the stack

The `RET` in the macro body: a macro is not a procedure; nothing got here by CALL. RET pops the 0000h on top of the stack: IP = **0000h**.

Step 3 / 4

VARIABLES

AX = 0E41h IP = 0000h SP = 0000h

OUTPUT

```
A
```

Macro: EMIT MACRO c / MOV AL, c / MOV AH, 0Eh / INT 10h / RET / ENDM. RET makes sense only in a procedure entered with CALL.

Code Trace: A RET Inside a Macro

```
1  org 100h
2  mov al, 'A'     ; EMIT 'A'
3  mov ah, 0Eh
4  int 10h
5  ret             ; the macro's RET
6  mov al, 'B'     ; EMIT 'B'
7  mov ah, 0Eh
8  int 10h
9  ret
10 ret
```

AX	IP	SP	screen
0E41h	0000h	0000h	A

lines 6–10 never ran: EMIT 'B' lost

The `INT 20h` at PSP:0000 ends the program. `EMIT 'B'` and the main program's own RET never ran; the screen shows only **A**.

Step 4 / 4

VARIABLES

`AX = 0E41h` `IP = 0000h` `SP = 0000h`

OUTPUT

```
A
```

Macro: EMIT MACRO c / MOV AL, c / MOV AH, 0Eh / INT 10h / RET / ENDM. RET makes sense only in a procedure entered with CALL.

Macro, EQU and Procedure

Criterion	EQU	Macro	Procedure
What is substituted?	a single value or expression	one or more lines	nothing; reached with CALL
Parameters	none	yes (text)	register, stack or memory
When?	assembly	assembly	run time
In memory	takes no space	the body at every use	one copy + 3 bytes per call
Example	`COLS EQU 80`	`PUTC 'A'`	`CALL PRINT_NUM`

In common

EQU and macros are text operations the assembler performs on the source; the CPU never sees either, it only executes the instructions they produce.

Choosing

EQU to name a single number, a macro for a short pattern of a few instructions, a procedure for long or widely used work.

Macros with Labels: The Problem

If a macro whose body contains a label is used twice, the expansion defines the same label **twice**:

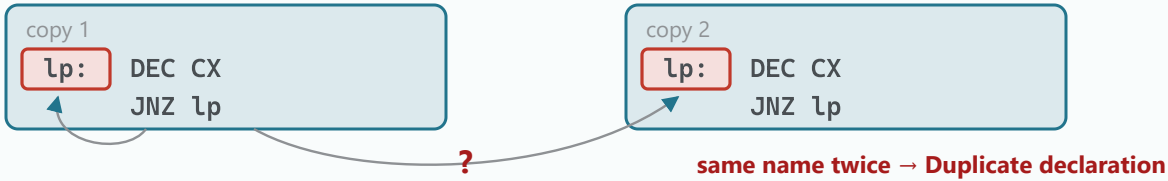
```
DELAY MACRO
lp:  DEC CX
      JNZ lp
ENDM

MOV CX, 5
DELAY
MOV CX, 9
DELAY
```

What the assembler sees:

```
MOV CX, 5
lp:   DEC CX     ; copy 1
      JNZ lp
MOV CX, 9
lp:   DEC CX     ; copy 2 !
      JNZ lp
```

emu8086: a **Duplicate declaration** error. Which `lp:` to jump to is ambiguous.



Fix: The LOCAL Directive

```
MyMacro2 MACRO
    LOCAL label1, label2
    CMP  AX, 2
    JE   label1
    CMP  AX, 3
    JE   label2
label1:
    INC  AX
label2:
    ADD  AX, 2
ENDM
```

- `LOCAL` must be the **first line** of the macro body (right after the MACRO line).
- At each expansion the assembler replaces the listed names with **unique** names (`??0000` , `??0001` , … in MASM).
- It works for labels, variables and procedure names inside the macro.

Usage	label1	label2
1st MyMacro2	`??0000`	`??0001`
2nd MyMacro2	`??0002`	`??0003`

Running MyMacro2 (AX = 2)

```
1 ▶ ORG 100h
2   MOV  AX, 2        ; before MyMacro2
3   CMP  AX, 2        ; MyMacro2 starts
4   JE   ??0000
5   CMP  AX, 3
6   JE   ??0001
7   ??0000:
8   INC  AX
9   ??0001:
10  ADD  AX, 2        ; MyMacro2 ends
11  RET
```

LOCAL has turned the labels into unique names.

Step 1 / 7

VARIABLES

AX = **0000h** | ZF = **0** | CF = **0**

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

Running MyMacro2 (AX = 2)

```
1  ORG 100h
▶  MOV  AX, 2        ; before MyMacro2
3  CMP  AX, 2        ; MyMacro2 starts
4  JE   ??0000
5  CMP  AX, 3
6  JE   ??0001
7  ??0000:
8  INC  AX
9  ??0001:
10 ADD  AX, 2        ; MyMacro2 ends
11 RET
```

AX = 2.

Step 2 / 7

VARIABLES

AX = 0002h | ZF = 0 | CF = 0

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

Running MyMacro2 (AX = 2)

```
1  ORG 100h
2  MOV  AX, 2        ; before MyMacro2
▶  CMP  AX, 2        ; MyMacro2 starts
4  JE   ??0000
5  CMP  AX, 3
6  JE   ??0001
7  ??0000:
8  INC  AX
9  ??0001:
10 ADD  AX, 2        ; MyMacro2 ends
11 RET
```

CMP AX, 2: 2 − 2 = 0 → **ZF = 1**.

Step 3 / 7

VARIABLES

AX = **0002h** | ZF = **1** | CF = **0**

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

Running MyMacro2 (AX = 2)

```
1  ORG 100h
2  MOV  AX, 2        ; before MyMacro2
3  CMP  AX, 2        ; MyMacro2 starts
▶  JE   ??0000
5  CMP  AX, 3
6  JE   ??0001
7  ??0000:
8  INC  AX
9  ??0001:
10 ADD  AX, 2        ; MyMacro2 ends
11 RET
```

JE: ZF = 1, so it jumps to **??0000** (label1).

Step 4 / 7

VARIABLES

AX = **0002h** | ZF = **1** | CF = **0**

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

Running MyMacro2 (AX = 2)

```
1  ORG 100h
2  MOV  AX, 2        ; before MyMacro2
3  CMP  AX, 2        ; MyMacro2 starts
4  JE   ??0000
5  CMP  AX, 3
6  JE   ??0001
7  ??0000:
▶  INC  AX
9  ??0001:
10 ADD  AX, 2        ; MyMacro2 ends
11 RET
```

INC AX → AX = 3.

Step 5 / 7

VARIABLES

AX = 0003h | ZF = 0 | CF = 0

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

Running MyMacro2 (AX = 2)

```
1  ORG 100h
2  MOV  AX, 2        ; before MyMacro2
3  CMP  AX, 2        ; MyMacro2 starts
4  JE   ??0000
5  CMP  AX, 3
6  JE   ??0001
7  ??0000:
8  INC  AX
9  ??0001:
▶  ADD  AX, 2        ; MyMacro2 ends
11 RET
```

ADD AX, 2 → AX = **5**.

Step 6 / 7

VARIABLES

AX = **0005h** ZF = **0** CF = **0**

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

Running MyMacro2 (AX = 2)

```
1  ORG 100h
2  MOV  AX, 2        ; before MyMacro2
3  CMP  AX, 2        ; MyMacro2 starts
4  JE   ??0000
5  CMP  AX, 3
6  JE   ??0001
7  ??0000:
8  INC  AX
9  ??0001:
10 ADD  AX, 2        ; MyMacro2 ends
▶  RET
```

The program ends.

Step 7 / 7

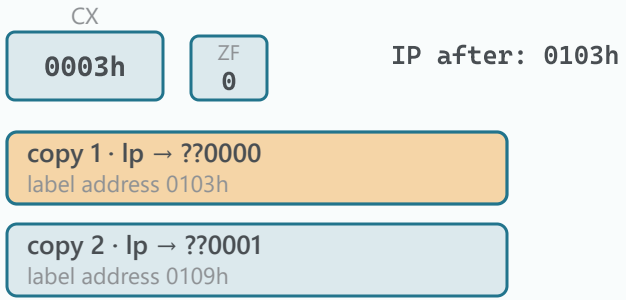
VARIABLES

AX = **0005h** ZF = **0** CF = **0**

With AX = 3 it would jump to label2 and AX = 5. With AX = 7 there is no jump and it falls into label1: INC + ADD → AX = 10.

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
▶ mov cx, 3
3 ??0000: dec cx    ; DELAY (copy 1)
4 jnz ??0000
5 mov cx, 2
6 ??0001: dec cx    ; DELAY (copy 2)
7 jnz ??0001
8 ret
```



The macro has `LOCAL lp`. At each expansion the assembler renamed `lp`: **??0000** (0103h) in copy 1, **??0001** (0109h) in copy 2. CX = 3.

Step 1 / 8

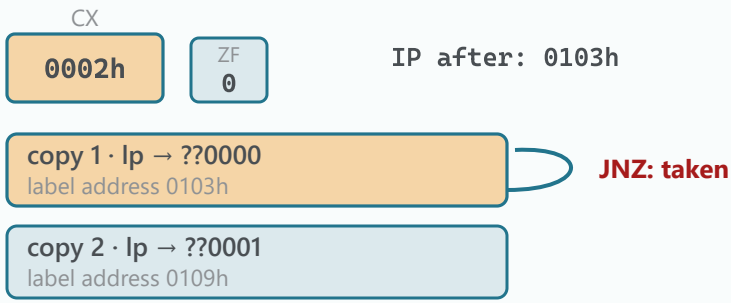
VARIABLES

`CX = 0003h` `ZF = 0` `IP = 0103h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM`; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY`.

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
▶ ??0000: dec cx   ; DELAY (copy 1)
▶ jnz ??0000
5 mov cx, 2
6 ??0001: dec cx   ; DELAY (copy 2)
7 jnz ??0001
8 ret
```



Pass 1: DEC CX → CX = 2, ZF = 0; JNZ goes back to **??0000** (0103h).

Step 2 / 8

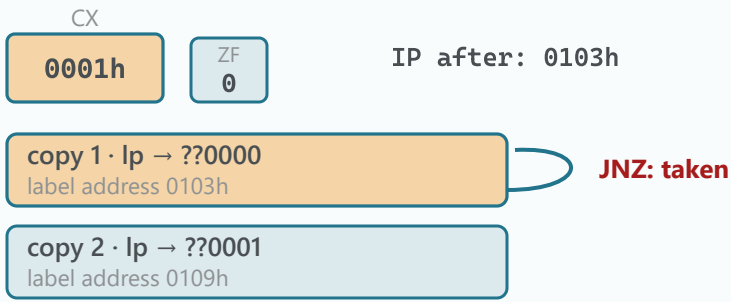
VARIABLES

`CX = 0002h` `ZF = 0` `IP = 0103h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
▶ ??0000: dec cx   ; DELAY (copy 1)
▶ jnz ??0000
5 mov cx, 2
6 ??0001: dec cx   ; DELAY (copy 2)
7 jnz ??0001
8 ret
```



Pass 2: CX = 1, ZF = 0 → again 0103h.

Step 3 / 8

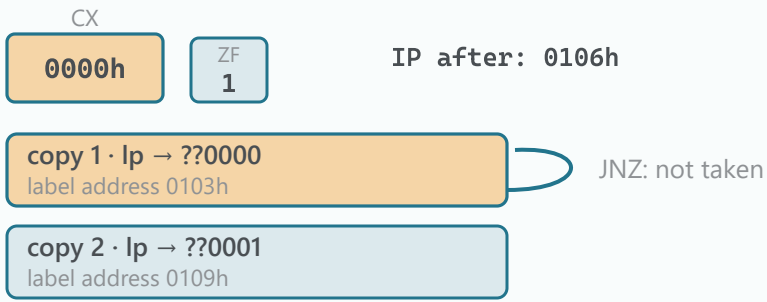
VARIABLES

`CX = 0001h` `ZF = 0` `IP = 0103h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
▶ ??0000: dec cx   ; DELAY (copy 1)
▶ jnz ??0000
5 mov cx, 2
6 ??0001: dec cx   ; DELAY (copy 2)
7 jnz ??0001
8 ret
```



Pass 3: CX = 0 → **ZF = 1**; JNZ does not jump, copy 1 is done.

Step 4 / 8

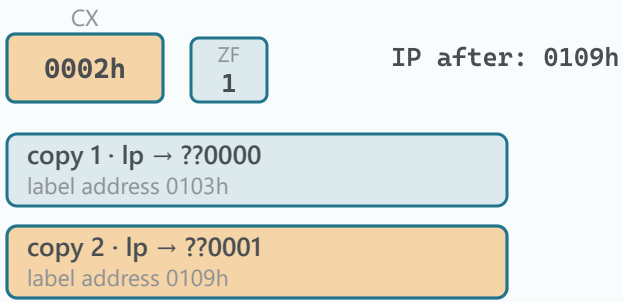
VARIABLES

`CX = 0000h` `ZF = 1` `IP = 0106h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
3 ??0000: dec cx   ; DELAY (copy 1)
4 jnz ??0000
▶ mov cx, 2
6 ??0001: dec cx   ; DELAY (copy 2)
7 jnz ??0001
8 ret
```



CX = 2: the counter for the macro's second use.

Step 5 / 8

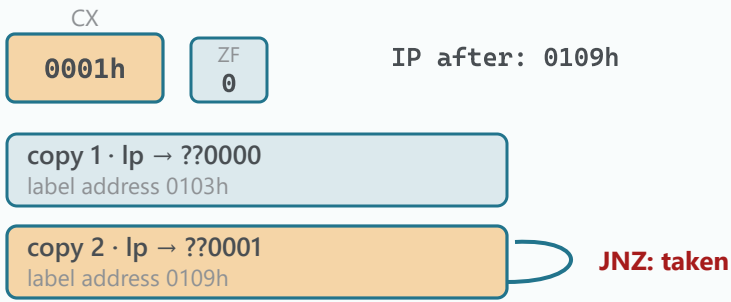
VARIABLES

`CX = 0002h` `ZF = 1` `IP = 0109h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
3 ??0000: dec cx    ; DELAY (copy 1)
4 jnz ??0000
5 mov cx, 2
▶ ??0001: dec cx    ; DELAY (copy 2)
▶ jnz ??0001
8 ret
```



Copy 2: DEC CX → CX = 1; JNZ goes back to **??0001** (0109h), not to copy 1's label.

Step 6 / 8

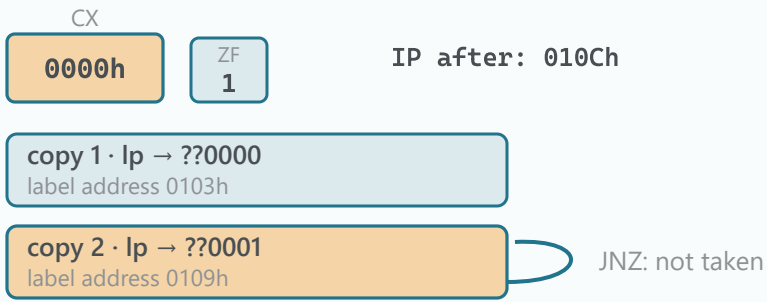
VARIABLES

`CX = 0001h` `ZF = 0` `IP = 0109h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
3 ??0000: dec cx    ; DELAY (copy 1)
4 jnz ??0000
5 mov cx, 2
▶ ??0001: dec cx    ; DELAY (copy 2)
▶ jnz ??0001
8 ret
```



CX = 0, ZF = 1: the loop ends. No label name is defined twice, so there is no *Duplicate declaration* error.

Step 7 / 8

VARIABLES

`CX = 0000h` `ZF = 1` `IP = 010Ch`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

LOCAL: Two Copies, Two Separate Labels

```
1 org 100h
2 mov cx, 3
3 ??0000: dec cx    ; DELAY (copy 1)
4 jnz ??0000
5 mov cx, 2
6 ??0001: dec cx    ; DELAY (copy 2)
7 jnz ??0001
▶ ret
```

CX

0000h

ZF

1

IP after: 0000h

copy 1 · lp → ??0000
label address 0103h

copy 2 · lp → ??0001
label address 0109h

RET: the program ends. 3 + 2 = 5 DECs ran in total.

Step 8 / 8

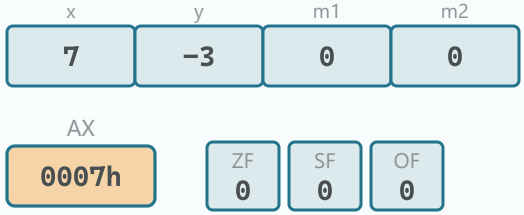
VARIABLES

`CX = 0000h` `ZF = 1` `IP = 0000h`

Macro: `DELAY MACRO` / `LOCAL lp` / `lp: DEC CX` / `JNZ lp` / `ENDM` ; main program `mov cx, 3` / `DELAY` / `mov cx, 2` / `DELAY` .

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
▶ mov ax, x       ; MAX2 x, y, m1
3  cmp ax, y
4  jge ??0000
5  mov ax, y
6  ??0000: mov m1, ax
7  mov ax, y      ; MAX2 y, 4, m2
8  cmp ax, 4
9  jge ??0001
10 mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



1st expansion `MAX2 x, y, m1` : AX = x = **7**.

Step 1 / 8

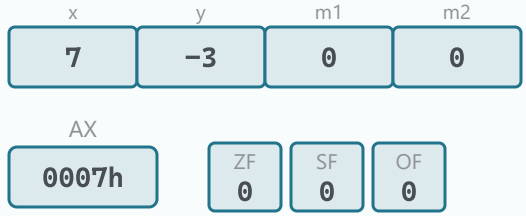
VARIABLES

`AX = 0007h` `m1 = 0000h` `m2 = 0000h`

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
2  mov ax, x       ; MAX2 x, y, m1
▶  cmp ax, y
4  jge ??0000
5  mov ax, y
6  ??0000: mov m1, ax
7  mov ax, y       ; MAX2 y, 4, m2
8  cmp ax, 4
9  jge ??0001
10 mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



`CMP AX, y` : 7 − (−3) = 10 → SF = 0, OF = 0.

Step 2 / 8

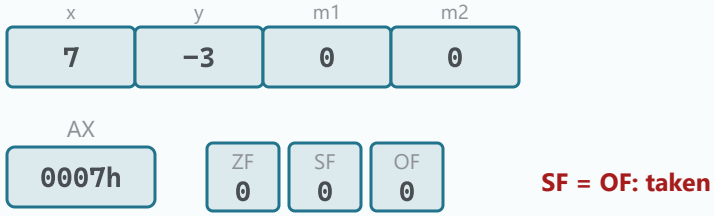
VARIABLES

`AX = 0007h` `m1 = 0000h` `m2 = 0000h`

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
2  mov ax, x       ; MAX2 x, y, m1
3  cmp ax, y
▶  jge ??0000
5  mov ax, y
6  ??0000: mov m1, ax
7  mov ax, y       ; MAX2 y, 4, m2
8  cmp ax, 4
9  jge ??0001
10 mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



JGE compares signed: SF = OF → jumps to the `??0000` label. LOCAL turned this copy's `ok` label into ??0000.

Step 3 / 8

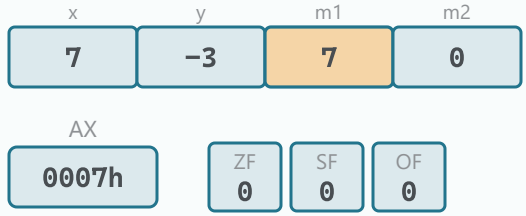
VARIABLES

`AX = 0007h` `m1 = 0000h` `m2 = 0000h`

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
2  mov ax, x       ; MAX2 x, y, m1
3  cmp ax, y
4  jge ??0000
5  mov ax, y
▶  ??0000: mov m1, ax
7  mov ax, y       ; MAX2 y, 4, m2
8  cmp ax, 4
9  jge ??0001
10 mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



m1 = **7**.

Step 4 / 8

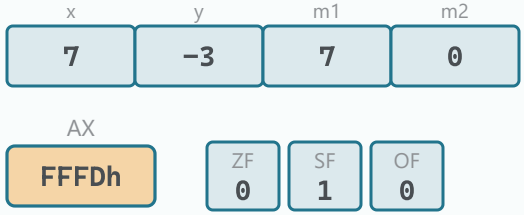
VARIABLES

AX = 0007h | m1 = 0007h | m2 = 0000h

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
2  mov ax, x       ; MAX2 x, y, m1
3  cmp ax, y
4  jge ??0000
5  mov ax, y
6  ??0000: mov m1, ax
▶  mov ax, y       ; MAX2 y, 4, m2
▶  cmp ax, 4
9  jge ??0001
10 mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



2nd expansion `MAX2 y, 4, m2` : AX = y = −3 (FFFDh); `CMP AX, 4` : −3 − 4 = −7 → SF = 1, OF = 0.

Step 5 / 8

VARIABLES

`AX = FFFDh` `m1 = 0007h` `m2 = 0000h`

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
 1 org 100h
 2 mov ax, x       ; MAX2 x, y, m1
 3 cmp ax, y
 4 jge ??0000
 5 mov ax, y
 6 ??0000: mov m1, ax
 7 mov ax, y       ; MAX2 y, 4, m2
 8 cmp ax, 4
 9 jge ??0001
10 mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



SF ≠ OF: `JGE ??0001` does not jump. The second copy's label is ??0001; no clash with the first.

Step 6 / 8

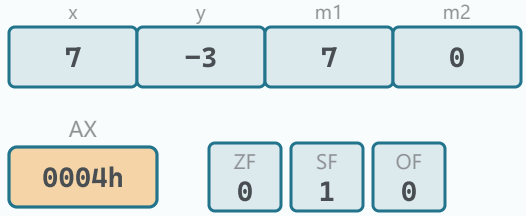
VARIABLES

`AX = FFFDh` `m1 = 0007h` `m2 = 0000h`

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
2  mov ax, x       ; MAX2 x, y, m1
3  cmp ax, y
4  jge ??0000
5  mov ax, y
6  ??0000: mov m1, ax
7  mov ax, y       ; MAX2 y, 4, m2
8  cmp ax, 4
9  jge ??0001
▶  mov ax, 4
11 ??0001: mov m2, ax
12 ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



The larger value is the second parameter: AX = **4**.

Step 7 / 8

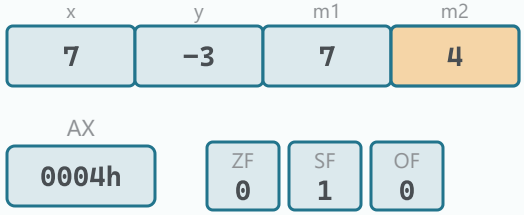
VARIABLES

AX = 0004h | m1 = 0007h | m2 = 0000h

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

Code Trace: Two Expansions of MAX2 p, q, r

```
1  org 100h
2  mov ax, x       ; MAX2 x, y, m1
3  cmp ax, y
4  jge ??0000
5  mov ax, y
6  ??0000: mov m1, ax
7  mov ax, y       ; MAX2 y, 4, m2
8  cmp ax, 4
9  jge ??0001
10 mov ax, 4
▶  ??0001: mov m2, ax
▶  ret
13 x  dw 7
14 y  dw -3
15 m1 dw ?
16 m2 dw ?
```



m2 = **4**; RET. With JAE, −3 (FFFDh) would count as the unsigned 65533, and m2 would be −3.

Step 8 / 8

VARIABLES

AX = 0004h | m1 = 0007h | m2 = 0004h

Macro: MAX2 MACRO p, q, r / LOCAL ok / MOV AX, p / CMP AX, q / JGE ok / MOV AX, q / ok: MOV r, AX / ENDM. In the first call the parameters are variables; in the second, one is an immediate.

PUTC: Printing a Character

```
; prints a char, moves the cursor
PUTC  MACRO  char
      push   ax
      mov    al, char
      mov    ah, 0eh
      int    10h
      pop    ax
ENDM

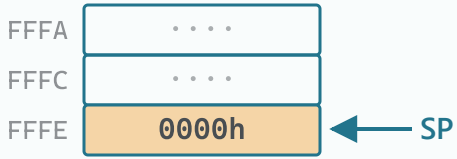
ORG  100h
START:
      PUTC  'A'
      PUTC  'B'
      RET
```

- The macro prints the character in its `char` parameter and moves the cursor one place right.
- `INT 10h` , `AH = 0Eh` : BIOS **teletype** output. The character in AL is written at the cursor; control characters such as 0Dh, 0Ah, 07h, 08h are interpreted.
- `push ax` / `pop ax` : the macro uses AX, so it preserves the caller's AX. No “invisible” side effect remains.
- This macro is identical to the PUTC macro in the `emu8086.inc` library.

The `RET` after `PUTC 'B'` ends the program; without it the CPU would go on executing the random bytes after the program.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

After loading, CX = **0011h**: 2 copies × 8 bytes + RET = 17 bytes. The screen is empty.

Step 1 / 12

VARIABLES

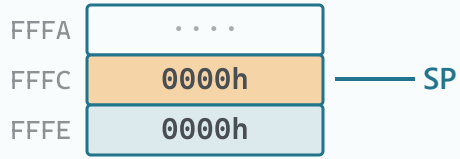
AX = **0000h** SP = **FFFEh**

OUTPUT

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
▶ push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

AX (0000h) is saved on the stack; SP = FFFCh.

Step 2 / 12

VARIABLES

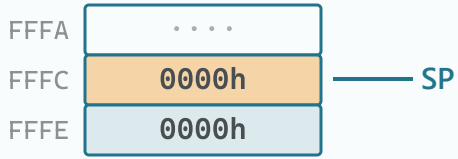
AX = 0000h SP = FFFCh

OUTPUT

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
▶  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

AL = 41h ('A').

Step 3 / 12

VARIABLES

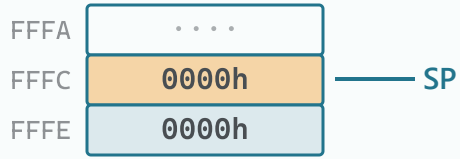
AX = 0041h SP = FFFCh

OUTPUT

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
▶  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

AH = 0Eh → AX = 0E41h.

Step 4 / 12

VARIABLES

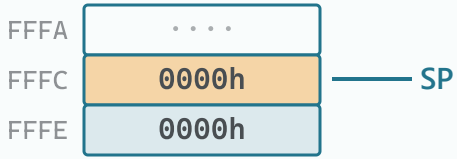
AX = 0E41h SP = FFFCh

OUTPUT

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
▶  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

INT 10h: **A** is printed and the cursor advances.

Step 5 / 12

VARIABLES

`AX = 0E41h` `SP = FFFCh`

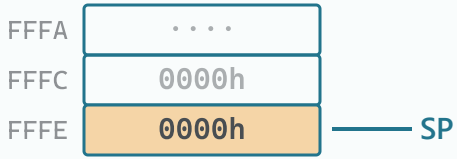
OUTPUT

```
A
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
▶  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

AX gets its old value (0000h) back; SP = FFFEh.

Step 6 / 12

VARIABLES

AX = 0000h SP = FFFEh

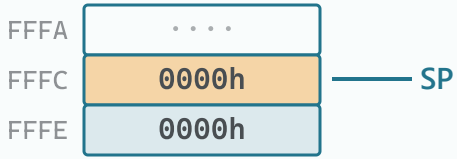
OUTPUT

```
A
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
▶  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

Second copy: AX is saved again.

Step 7 / 12

VARIABLES

AX = 0000h SP = FFFCh

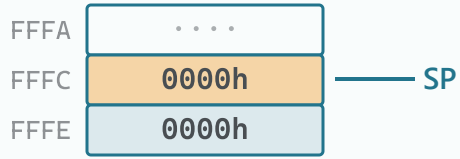
OUTPUT

```
A
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
▶  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

AL = 42h ('B').

Step 8 / 12

VARIABLES

AX = 0042h SP = FFFCh

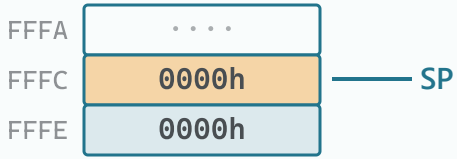
OUTPUT

```
A
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
▶  mov  ah, 0eh
10 int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

AX = 0E42h.

Step 9 / 12

VARIABLES

AX = 0E42h SP = FFFCh

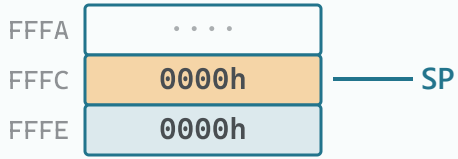
OUTPUT

```
A
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
▶  int  10h
11 pop  ax
12 RET
```



SS:offset · low address at top

INT 10h: the screen shows **AB**.

Step 10 / 12

VARIABLES

`AX = 0E42h` `SP = FFFCh`

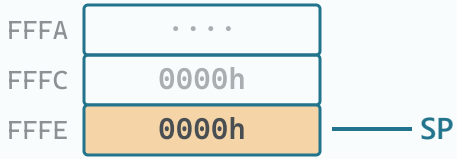
OUTPUT

```
AB
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
▶  pop  ax
12 RET
```



SS:offset · low address at top

AX = 0000h; the macro did not clobber the caller's AX.

Step 11 / 12

VARIABLES

AX = 0000h SP = FFFEh

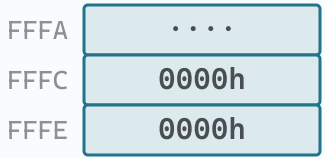
OUTPUT

```
AB
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

PUTC 'A' and PUTC 'B' at Run Time

```
1  ORG  100h
2  push ax          ; PUTC 'A'
3  mov  al, 'A'
4  mov  ah, 0eh
5  int  10h
6  pop  ax
7  push ax          ; PUTC 'B'
8  mov  al, 'B'
9  mov  ah, 0eh
10 int  10h
11 pop  ax
▶  RET
```



SS:offset · low address at top

RET: the program ends.

Step 12 / 12

VARIABLES

AX = 0000h SP = 0000h

OUTPUT

```
AB
```

Output in the emu8086 screen window (80×25): AB. Each PUTC copy is 8 bytes: 50 B0 41 B4 0E CD 10 58.

Register-Saving Macros

```
SAVE MACRO
  PUSH AX
  PUSH BX
  PUSH CX
  PUSH DX
ENDM

RESTORE MACRO
  POP DX
  POP CX
  POP BX
  POP AX
ENDM
```

- PUTC's `push ax` / `pop ax` pattern grows to four registers in larger macros; the pattern itself can be a macro.
- RESTORE pops the registers in the **reverse** of the order SAVE pushed them: the stack is last in, first out. In the same order, AX and DX, BX and CX would swap.
- Each SAVE + RESTORE pair adds 8 bytes and 8 stack accesses: 4 × 11 (PUSH) + 4 × 8 (POP) = 76 clocks on the 8086.
- Use the two macros together in one body; a conditional exit in between leaves the stack unbalanced.

On the 80186 and later, single-instruction `PUSHA` / `POPA` save all general-purpose registers; the 8086 lacks them, so a macro does the job.

Exponentiation: The POWER Macro

```
POWER macro b, e, r
    mov ax, 1          ; result so far = 1
    mov cx, e          ; exponent → CX
    mov bx, b          ; base → BX
powerloop:
    mul bx             ; DX:AX = AX * BX
    loop powerloop     ; repeat e times
    mov r, ax          ; store result
endm
```

```
ORG 100h
start:
    POWER base, exponent, result
    ret
base     dw 2      ; base
exponent dw 3      ; exponent
result   dw ?      ; result
```

- Result $r = b^e$: start AX at 1 and multiply by the base e times.
- `MUL BX` : DX:AX ← AX × BX (unsigned). **DX changes too!**
- `LOOP` : decrements CX and jumps to the label if CX ≠ 0.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

The data comes right after the code: base = 0113h, exponent = 0115h, result = 0117h (code is 19 bytes).

Step 1 / 12

VARIABLES

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

DX = **0000h** [0117h] = **0000h** CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
▶  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2     ; 0113h
11 exponent dw 3     ; 0115h
12 result   dw ?     ; 0117h
```

AX = 1 (result so far).

Step 2 / 12

VARIABLES

AX = **0001h** | BX = **0000h** | CX = **0000h** | DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
▶  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2     ; 0113h
11 exponent dw 3     ; 0115h
12 result   dw ?     ; 0117h
```

CX ← [0115h] = **3** (exponent).

Step 3 / 12

VARIABLES

AX = **0001h** | BX = **0000h** | CX = **0003h**

DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
▶  mov bx, base
5  powerloop:
6  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

BX ← [0113h] = **2** (base).

Step 4 / 12

VARIABLES

AX = **0001h** | BX = **0002h** | CX = **0003h**

DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
▶  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

Pass 1: DX:AX = 1 × 2 = **0002h**.

Step 5 / 12

VARIABLES

AX = **0002h** | BX = **0002h** | CX = **0003h**

DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
▶  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

LOOP: CX = 2 ≠ 0 → powerloop.

Step 6 / 12

VARIABLES

AX = **0002h** BX = **0002h** CX = **0002h**

DX = **0000h** [0117h] = **0000h** CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
▶  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

Pass 2: AX = 2 × 2 = **0004h**.

Step 7 / 12

VARIABLES

AX = **0004h** | BX = **0002h** | CX = **0002h** | DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
▶  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

LOOP: CX = 1 → powerloop.

Step 8 / 12

VARIABLES

AX = **0004h** | BX = **0002h** | CX = **0001h** | DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
▶  mul bx
7  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

Pass 3: AX = 4 × 2 = **0008h**.

Step 9 / 12

VARIABLES

AX = **0008h** | BX = **0002h** | CX = **0001h**

DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
▶  loop powerloop
8  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

LOOP: CX = 0 → exit the loop.

Step 10 / 12

VARIABLES

AX = **0008h** | BX = **0002h** | CX = **0000h**

DX = **0000h** | [0117h] = **0000h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
7  loop powerloop
▶  mov result, ax
9  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

result ← AX: [0117h] = **0008h** = 2^3.

Step 11 / 12

VARIABLES

AX = **0008h** | BX = **0002h** | CX = **0000h** | DX = **0000h** | [0117h] = **0008h** | CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

POWER 2, 3 Step by Step

```
1  ORG 100h
2  mov ax, 1          ; POWER expansion
3  mov cx, exponent
4  mov bx, base
5  powerloop:
6  mul bx
7  loop powerloop
8  mov result, ax
▶  ret
10 base     dw 2      ; 0113h
11 exponent dw 3      ; 0115h
12 result   dw ?      ; 0117h
```

RET: the program ends.

Step 12 / 12

VARIABLES

AX = **0008h** BX = **0002h** CX = **0000h**

DX = **0000h** [0117h] = **0008h** CF = **0**

After MUL, DX = 0, so CF = OF = 0. Had the result exceeded 16 bits (e.g. 2^{16}), DX ≠ 0 and CF = 1; since the macro stores only AX, the result would be wrong.

Pitfalls of the POWER Macro

1. Exponent = 0

With CX = 0, `LOOP` decrements CX first: FFFFh → the loop runs **65,536 times**; b^0 should be 1. Fix: `JCXZ`.

2. Label

`powerloop` is not LOCAL: a *Duplicate declaration* on the second use.

3. Clobbered registers

AX, BX, CX and (because of MUL) DX change.

4. Overflow

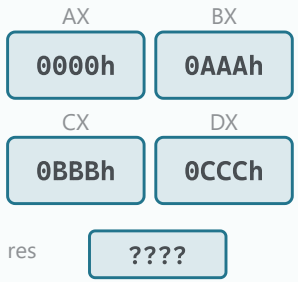
If the result exceeds 65,535, the high part stays in DX and is not stored.

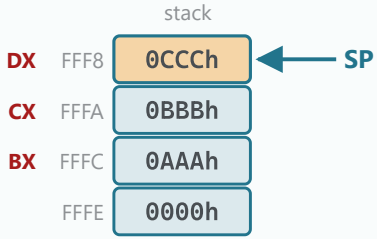
Fixed version:

```
POWER MACRO b, e, r
      LOCAL p_loop, p_end
      PUSH BX
      PUSH CX
      PUSH DX
      MOV  AX, 1
      MOV  CX, e
      MOV  BX, b
      JCXZ p_end    ; e = 0 → 1
p_loop: MUL BX
      LOOP p_loop
p_end: MOV r, AX
      POP  DX
      POP  CX
      POP  BX
ENDM
```

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





The caller's registers are BX = 0AAAh, CX = 0BBBh, DX = 0CCCh. The fixed macro first saves all three on the stack: SP = FFF8h.

Step 1 / 10

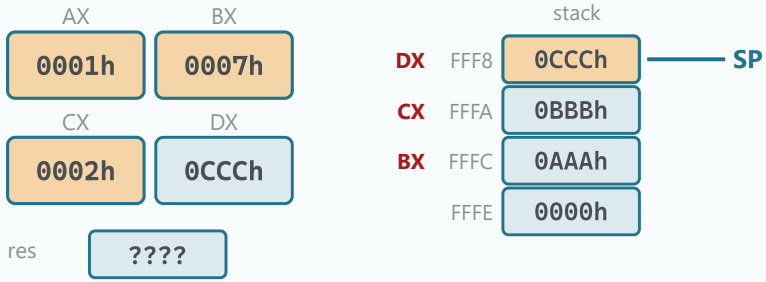
VARIABLES

AX = **0000h** · BX = **0AAAh** · CX = **0BBBh** · DX = **0CCCh** · SP = **FFF8h** · res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 ▶ mov  ax, 1
 ▶ mov  cx, expn
 ▶ mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```



AX = 1, CX ← expn = **2**, BX ← base = **7**. The registers now belong to the macro.

Step 2 / 10

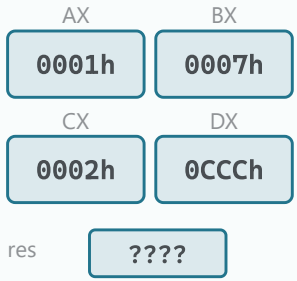
VARIABLES

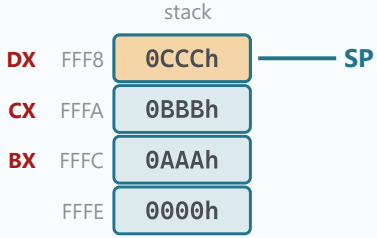
AX = **0001h** BX = **0007h** CX = **0002h** DX = **0CCCh** SP = **FFF8h** res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
▶ jcxz ??0001       ; expn = 0 → 1
 9 ??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





JCXZ: CX = 2 ≠ 0, no jump. Had expn been 0, it would go straight to ??0001 and the result would be 1 (no 65,536-pass LOOP trap).

Step 3 / 10

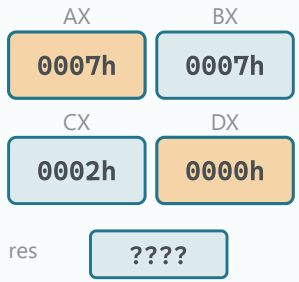
VARIABLES

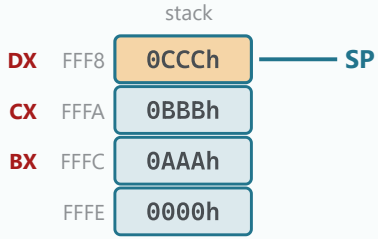
AX = **0001h** · BX = **0007h** · CX = **0002h** · DX = **0CCCh** · SP = **FFF8h** · res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
1  org 100h
2  push bx          ; POWER expansion
3  push cx
4  push dx
5  mov  ax, 1
6  mov  cx, expn
7  mov  bx, base
8  jcxz ??0001      ; expn = 0 → 1
▶  ??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





1st MUL: DX:AX = 1 × 7 = **0007h**. MUL writes DX too (DX = 0).

Step 4 / 10

VARIABLES

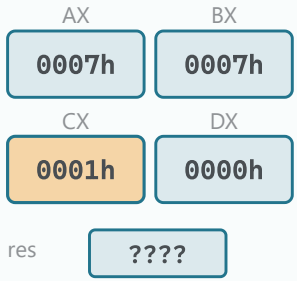
AX = **0007h** BX = **0007h** CX = **0002h** DX = **0000h**

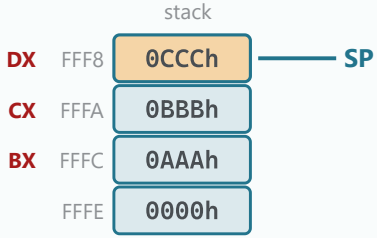
SP = **FFF8h** res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
▶10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





LOOP: CX = 1 ≠ 0 → ??0000.

Step 5 / 10

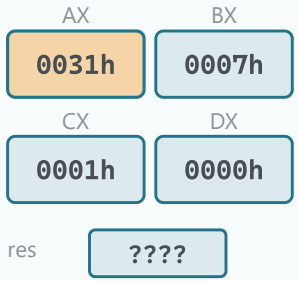
VARIABLES

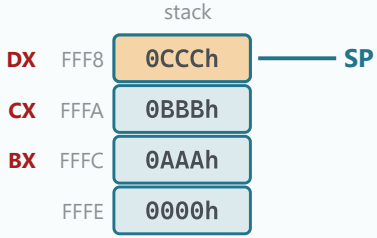
AX = **0007h** · BX = **0007h** · CX = **0001h** · DX = **0000h** · SP = **FFF8h** · res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
▶??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





2nd MUL: DX:AX = 7 × 7 = **0031h** = 49.

Step 6 / 10

VARIABLES

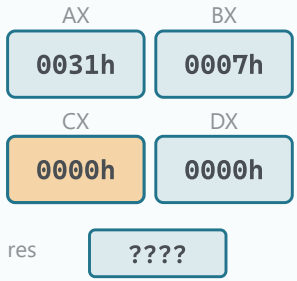
AX = **0031h** BX = **0007h** CX = **0001h** DX = **0000h**

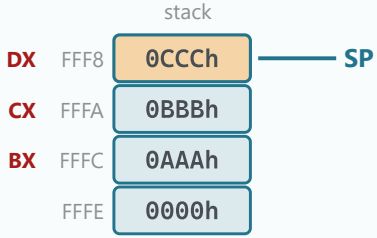
SP = **FFF8h** res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
▶10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





LOOP: CX = 0 → exit the loop.

Step 7 / 10

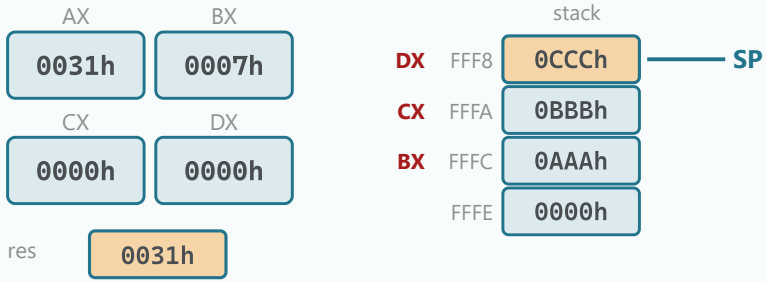
VARIABLES

AX = **0031h** · BX = **0007h** · CX = **0000h** · DX = **0000h** · SP = **FFF8h** · res = **?**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
10 loop ??0000
▶ ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```



res ← AX: [011Fh] = **0031h** = 7^2. LOCAL renamed the labels ??0000 / ??0001.

Step 8 / 10

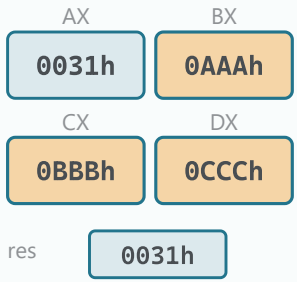
VARIABLES

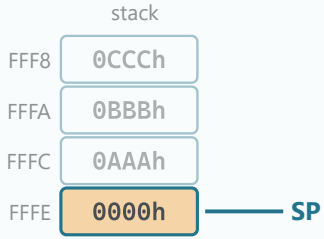
AX = **0031h** BX = **0007h** CX = **0000h** DX = **0000h** SP = **FFF8h** res = **0031h**

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
▶ pop  dx
▶ pop  cx
▶ pop  bx
15 ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





Three POPs, in reverse order of the PUSHes: DX = 0CCCh, CX = 0BBBh, BX = 0AAAh are back; SP = FFFEh. The macro changed only AX.

Step 9 / 10

VARIABLES

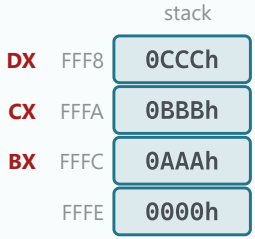
AX = 0031h · BX = 0AAAh · CX = 0BBBh · DX = 0CCCh · SP = FFFEh · res = 0031h

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Fixed POWER 7, 2 Step by Step

```
 1 org 100h
 2 push bx          ; POWER expansion
 3 push cx
 4 push dx
 5 mov  ax, 1
 6 mov  cx, expn
 7 mov  bx, base
 8 jcxz ??0001      ; expn = 0 → 1
 9 ??0000: mul bx
10 loop ??0000
11 ??0001: mov res, ax
12 pop  dx
13 pop  cx
14 pop  bx
▶ ret
16 base dw 7
17 expn    dw 2
18 res dw ?
```





RET: the program ends.

Step 10 / 10

VARIABLES

AX = 0031h · BX = 0AAAh · CX = 0BBBh · DX = 0CCCh · SP = 0000h · res = 0031h

The code is the expansion `POWER base, expn, res` of the fixed macro; LOCAL names are shown as the assembler generates them.

Nested Macros

```
PRINT2 MACRO c1, c2
       PUTC c1
       PUTC c2
ENDM

PRINT2 'O', 'K'
```

```
; level 1: PRINT2 expands
PUTC 'O'
PUTC 'K'
```

The outer macro expands first: both lines in its body are themselves **macro calls**.

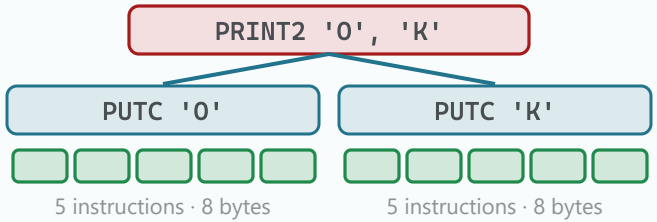
Step 1 / 2

PUTC must be defined before PRINT2. If a macro calls itself (recursion), the expansion never ends without conditional assembly.

Nested Macros

```
PRINT2 MACRO c1, c2
       PUTC c1
       PUTC c2
ENDM

PRINT2 'O', 'K'
```




```
; level 2: the PUTCs expand
push ax
mov  al, 'O'
mov  ah, 0eh
int  10h
pop  ax
push ax
mov  al, 'K'
mov  ah, 0eh
int  10h
pop  ax
```

The assembler expands the inner macros too: a one-line call produces **10 instructions** (16 bytes). The screen shows: OK.

PUTC must be defined before PRINT2. If a macro calls itself (recursion), the expansion never ends without conditional assembly.

Macro Libraries: emu8086.inc

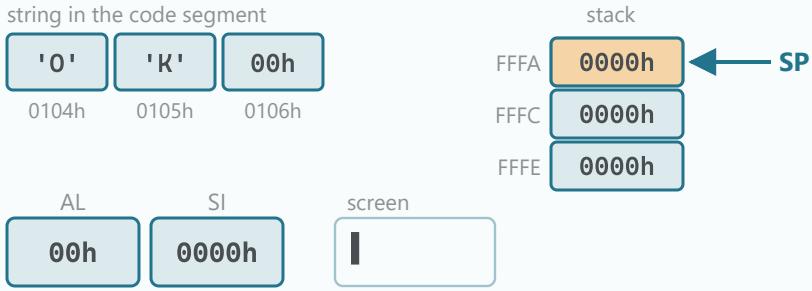
- Frequently used macros are collected in a separate file and added to a program with `INCLUDE`.
- Unused macros generate no code, so including a large library does not grow the file.
- Macros can call other macros (nested expansion).
- Besides macros, the library offers procedures through the `DEFINE_...` macros (e.g. `DEFINE_SCAN_NUM`): the two are used together.

Macro	Function
`PUTC ch`	Prints one character
`PRINT 'text'`	Prints text
`PRINTN 'text'`	Text + new line
`GOTOXY x, y`	Moves the cursor
`CURSOROFF`	Hides the cursor

```
include 'emu8086.inc'
ORG 100h
    GOTOXY 10, 5
    PRINT 'Hello'
    PUTC  '!'
    RET
```

emu8086.inc: Expanding PRINT 'OK'

```
 1 org 100h
 2 push ax          ; PRINT 'OK'
 3 push si
 4 jmp  ??0003      ; skip the data
 5 ??0001 db 'OK', 0
 6 ??0003: lea si, ??0001
 7 ??0000: mov al, cs:[si]
 8 cmp  al, 0
 9 jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



PRINT 'OK' is expanded. The macro will use AX and SI, so it saves both on the stack first.

Step 1 / 10

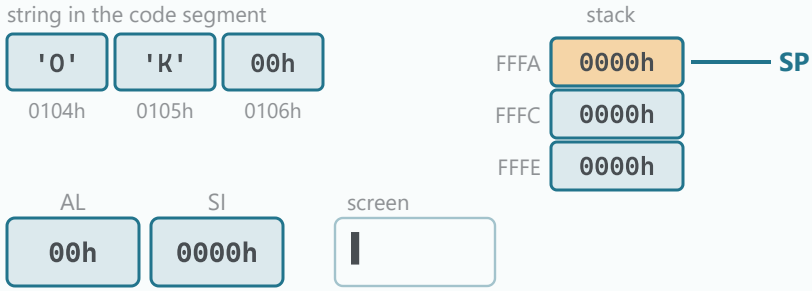
VARIABLES

AL = **00h** SI = **0000h** SP = **FFFAh** ZF = **0**

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
▶  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
6  ??0003: lea si, ??0001
7  ??0000: mov al, cs:[si]
8  cmp  al, 0
9  jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



The string is embedded **between instructions** inside the macro (0104h: 'O', 'K', 0). The JMP jumps over these bytes so the CPU does not take them for instructions.

Step 2 / 10

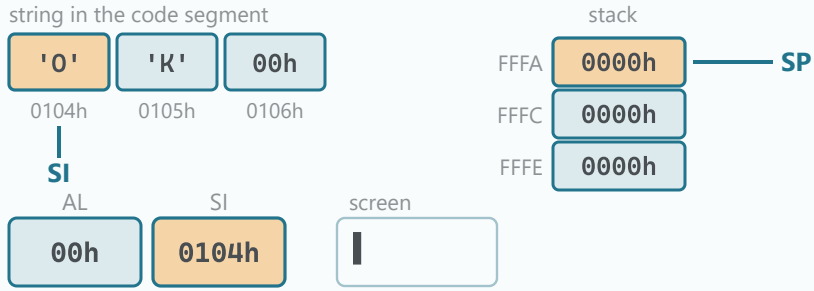
VARIABLES

`AL = 00h` `SI = 0000h` `SP = FFFAh` `ZF = 0`

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
4  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
▶  ??0003: lea si, ??0001
7  ??0000: mov al, cs:[si]
8  cmp  al, 0
9  jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



SI ← the string address 0104h. LOCAL names (??0000…) are new for each PRINT.

Step 3 / 10

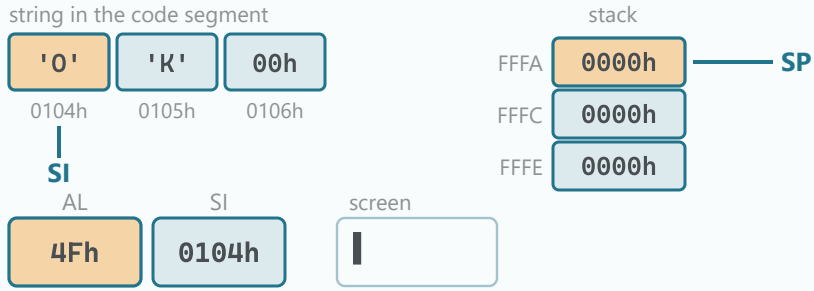
VARIABLES

AL = **00h** | SI = **0104h** | SP = **FFFAh** | ZF = **0**

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
 1 org 100h
 2 push ax          ; PRINT 'OK'
 3 push si
 4 jmp  ??0003      ; skip the data
 5 ??0001 db 'OK', 0
 6 ??0003: lea si, ??0001
 ▶ ??0000: mov al, cs:[si]
 ▶ cmp  al, 0
 ▶ jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



AL ← CS:[SI] = 4Fh ('O'). The string is in the code segment, so the **CS:** prefix is used. Not 0 → continue.

Step 4 / 10

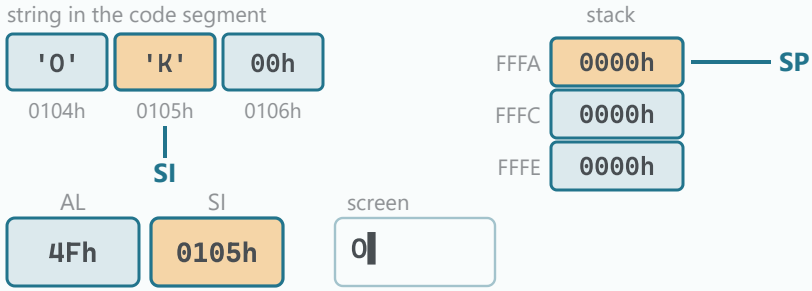
VARIABLES

AL = **4Fh** SI = **0104h** SP = **FFFAh** ZF = **0**

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
4  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
6  ??0003: lea si, ??0001
7  ??0000: mov al, cs:[si]
8  cmp  al, 0
9  jz   ??0002
▶  inc  si
▶  mov  ah, 0Eh
▶  int  10h
▶  jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



SI advances by one, INT 10h/0Eh prints **O**, JMP goes back to the top of the loop.

Step 5 / 10

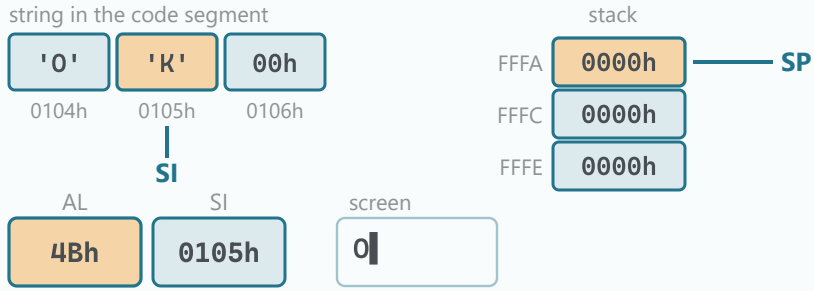
VARIABLES

AL = **4Fh** | SI = **0105h** | SP = **FFFAh** | ZF = **0**

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
4  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
6  ??0003: lea si, ??0001
▶  ??0000: mov al, cs:[si]
▶  cmp  al, 0
▶  jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



AL = 4Bh ('K'); not the terminator.

Step 6 / 10

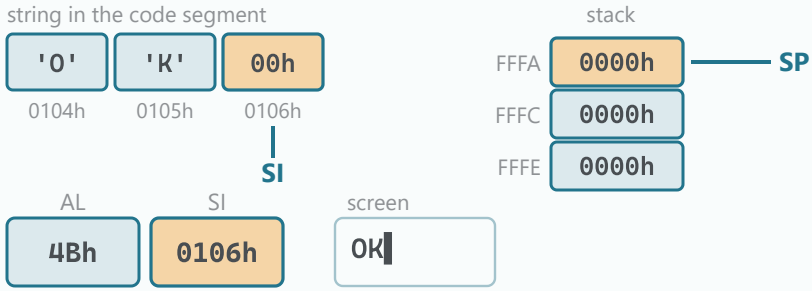
VARIABLES

AL = 4Bh | SI = 0105h | SP = FFFAh | ZF = 0

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
4  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
6  ??0003: lea si, ??0001
7  ??0000: mov al, cs:[si]
8  cmp  al, 0
9  jz   ??0002
▶  inc  si
▶  mov  ah, 0Eh
▶  int  10h
▶  jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



The screen shows **OK**.

Step 7 / 10

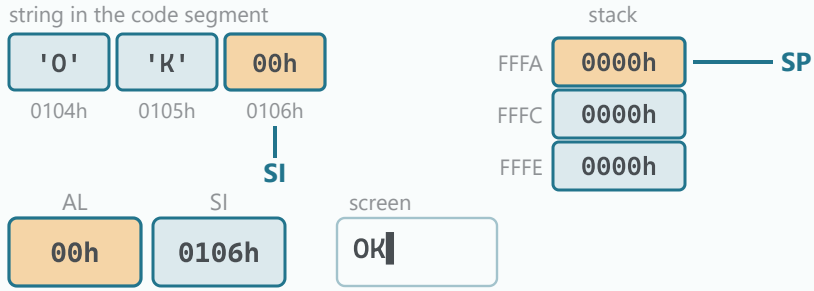
VARIABLES

AL = **4Bh** | SI = **0106h** | SP = **FFFAh** | ZF = **0**

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
4  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
6  ??0003: lea si, ??0001
▶  ??0000: mov al, cs:[si]
▶  cmp  al, 0
▶  jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
16 ret
```



AL = 00h: CMP → ZF = 1, JZ jumps to the exit.

Step 8 / 10

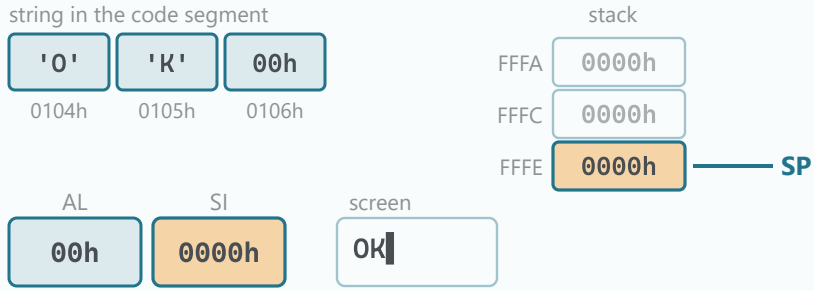
VARIABLES

AL = 00h | SI = 0106h | SP = FFFAh | ZF = 1

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
 1 org 100h
 2 push ax          ; PRINT 'OK'
 3 push si
 4 jmp  ??0003      ; skip the data
 5 ??0001 db 'OK', 0
 6 ??0003: lea si, ??0001
 7 ??0000: mov al, cs:[si]
 8 cmp  al, 0
 9 jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
▶ ??0002: pop si
▶ pop  ax
16 ret
```



SI and AX are popped back: the macro did not clobber the caller's registers.

Step 9 / 10

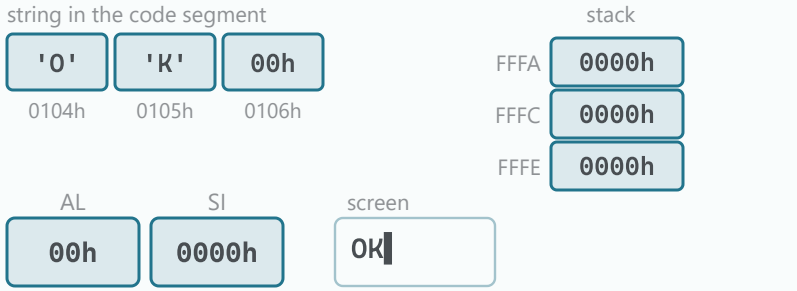
VARIABLES

AL = 00h | SI = 0000h | SP = FFFEh | ZF = 1

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

emu8086.inc: Expanding PRINT 'OK'

```
1  org 100h
2  push ax          ; PRINT 'OK'
3  push si
4  jmp  ??0003      ; skip the data
5  ??0001 db 'OK', 0
6  ??0003: lea si, ??0001
7  ??0000: mov al, cs:[si]
8  cmp  al, 0
9  jz   ??0002
10 inc  si
11 mov  ah, 0Eh
12 int  10h
13 jmp  ??0000
14 ??0002: pop si
15 pop  ax
▶  ret
```



RET: the program ends. The one-line PRINT 'OK' produced 26 bytes of code + data.

Step 10 / 10

VARIABLES

AL = 00h | SI = 0000h | SP = 0000h | ZF = 1

Expansion of the PRINT macro from emu8086.inc (LOCAL names as the assembler generates them). PRINTN does the same and appends CR LF.

Macro Operators: &, <>, !, %, ;;

Operator	Role	Example and expansion
`&`	joins a parameter to adjacent text	`CLEAR MACRO r` · `XOR r&X, r&X` → `CLEAR B`: `XOR BX, BX`
`< >`	makes text containing commas or spaces a single parameter	`EMIT <'A', 'B'>` → one parameter: `'A', 'B'`
`!`	makes the next character an ordinary character	`<5 !> 3>` → text: `5 > 3`
`%`	evaluates an expression and substitutes the result as text	`SHOW %(4*8)` → parameter `32`
`;;`	macro comment; not copied into the expansion	`;; visible only in the definition`

- These operators are defined in MASM and TASM; they are used to split and join text when writing a macro definition.
- Without `&`, `rX` counts as a single name and is not substituted.
- A comment written with a single `;` is repeated in every expansion; `;;` keeps the listing short.

Repeat Blocks: REPT, IRP, IRPC

REPT n

```
REPT 3
  SHL AX, 1
ENDM
```

expansion: `SHL AX, 1` three times (AX × 8)

IRP p, <list>

```
IRP r, <AX,BX,CX>
  PUSH r
ENDM
```

expansion: `PUSH AX` · `PUSH BX` · `PUSH CX`

IRPC p, text

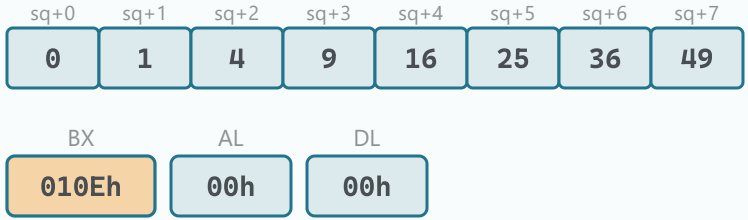
```
IRPC c, OK
  PUTC '&c'
ENDM
```

expansion: `PUTC 'O'` · `PUTC 'K'`

- Repeat blocks are nameless macros expanded right where they are defined; they end with ENDM.
- REPT writes the body n times, IRP once for each item in the list, IRPC once for each character of the text. In MASM 6, IRP and IRPC can also be written as FOR and FORC.
- No loop counter is needed, but the code grows: `REPT 100` copies the body 100 times. A counted LOOP works with a single copy.
- The most common use is generating tables at assembly time: number tables, masks, squares or sine values.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
▶ lea  bx, sq
3  mov  al, 5
4  xlat            ; AL = sq[5]
5  mov  dl, al
6  mov  al, 7
7  xlat            ; AL = sq[7]
8  add  al, dl
9  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



REPT generated the table at assembly time; memory holds just 8 bytes. `LEA BX, sq` : BX = **010Eh**.

Step 1 / 7

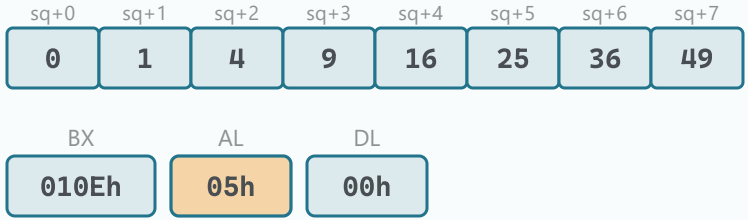
VARIABLES

`BX = 010Eh` `AL = 00h` `DL = 00h`

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
2  lea  bx, sq
▶  mov  al, 5
4  xlat            ; AL = sq[5]
5  mov  dl, al
6  mov  al, 7
7  xlat            ; AL = sq[7]
8  add  al, dl
9  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



AL = 5: the number to square, which is also the table index.

Step 2 / 7

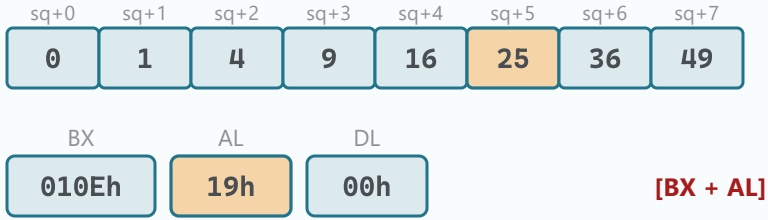
VARIABLES

BX = 010Eh AL = 05h DL = 00h

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
2  lea  bx, sq
3  mov  al, 5
▶ xlat           ; AL = sq[5]
5  mov  dl, al
6  mov  al, 7
7  xlat           ; AL = sq[7]
8  add  al, dl
9  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



`XLAT` : AL ← [BX + AL] = sq+5 = **25** (19h). No multiplication, just a table read.

Step 3 / 7

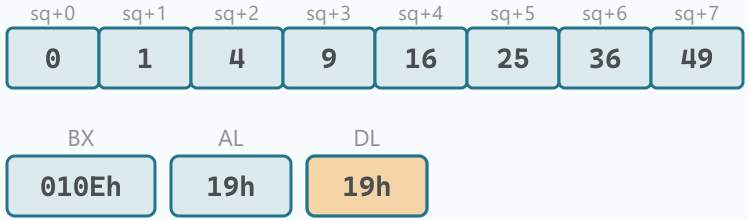
VARIABLES

BX = **010Eh** AL = **19h** DL = **00h**

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
2  lea  bx, sq
3  mov  al, 5
4  xlat            ; AL = sq[5]
▶  mov  dl, al
6  mov  al, 7
7  xlat            ; AL = sq[7]
8  add  al, dl
9  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



DL = 19h: the first result is saved.

Step 4 / 7

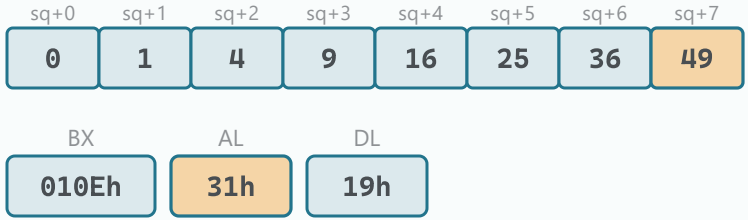
VARIABLES

BX = 010Eh AL = 19h DL = 19h

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
2  lea  bx, sq
3  mov  al, 5
4  xlat            ; AL = sq[5]
5  mov  dl, al
6  mov  al, 7
7  xlat            ; AL = sq[7]
8  add  al, dl
9  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



AL = 7, `XLAT` → sq+7 = **49** (31h).

Step 5 / 7

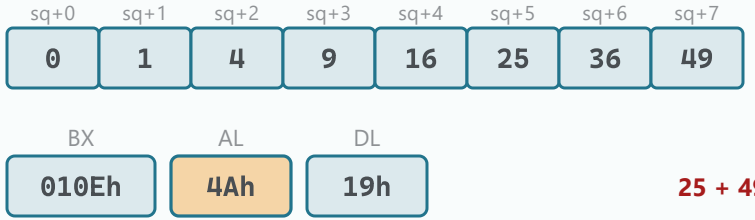
VARIABLES

BX = **010Eh** AL = **31h** DL = **19h**

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
2  lea  bx, sq
3  mov  al, 5
4  xlat            ; AL = sq[5]
5  mov  dl, al
6  mov  al, 7
7  xlat            ; AL = sq[7]
▶ add  al, dl
9  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



`ADD AL, DL` : 49 + 25 = **74** (4Ah) = $5^2 + 7^2$.

Step 6 / 7

VARIABLES

`BX = 010Eh` `AL = 4Ah` `DL = 19h`

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Code Trace: A Table of Squares Generated by REPT

```
1  org 100h
2  lea  bx, sq
3  mov  al, 5
4  xlat            ; AL = sq[5]
5  mov  dl, al
6  mov  al, 7
7  xlat            ; AL = sq[7]
8  add  al, dl
▶  ret
10 sq db 0, 1, 4, 9, 16, 25, 36, 49
```



RET. XLAT takes 11 clocks; an 8-bit MUL on the 8086 takes 70–77.

Step 7 / 7

VARIABLES

BX = **010Eh** AL = **4Ah** DL = **19h**

Source of the table: n = 0 / sq LABEL BYTE / REPT 8 / DB n*n / n = n + 1 / ENDM. The code ran in the expanded form of this block. An EQU constant cannot change; a symbol defined with = takes a new value on each pass.

Conditional Assembly: IF, IFB, IFIDN

```
SHIFTN MACRO r, n
  IFB <n>          ;; n not given?
    SHL r, 1
  ELSE
    MOV CL, n
    SHL r, CL
  ENDIF
ENDM

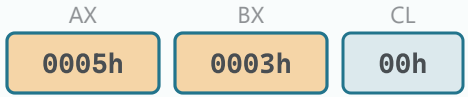
DEBUG = 1
IF DEBUG
  PUTC '*'         ; debug build only
ENDIF
```

Directive	Condition
`IF expr`	expr ≠ 0
`IFE expr`	expr = 0
`IFB <p>`	parameter blank
`IFNB <p>`	parameter not blank
`IFIDN <a>,<b>`	two texts identical
`IFDIF <a>,<b>`	two texts different

- The condition is evaluated at assembly time: the branch not chosen never enters the machine code; there is no flag or jump at run time.
- `EXITM` ends the expansion at that point; `IFIDN <r>, <CX>` can catch a clobbered register passed as a parameter.

Code Trace: SHIFTN AX and SHIFTN BX, 3

```
1 org 100h
▶ mov ax, 5
▶ mov bx, 3
4 shl ax, 1       ; SHIFTN AX
5 mov cl, 3       ; SHIFTN BX, 3
6 shl bx, cl
7 ret
```



AL = 00000101 CF = 0
BL = 00000011

AX = 5, BX = 3: the values to shift.

Step 1 / 5

VARIABLES

AX = 0005h | BX = 0003h | CL = 00h | CF = 0

The code is the expanded form of two SHIFTN calls. The assembler chose which branch to emit; no comparison happened at run time.

Code Trace: SHIFTN AX and SHIFTN BX, 3

```
1 org 100h
2 mov ax, 5
3 mov bx, 3
▶ shl ax, 1       ; SHIFTN AX
5 mov cl, 3       ; SHIFTN BX, 3
6 shl bx, cl
7 ret
```

AX	BX	CL
000Ah	0003h	00h

AL = 00001010 CF = 0
BL = 00000011

IFB branch

`SHIFTN AX` : n is blank, the IFB branch is chosen → a single `SHL AX, 1` . AX = **000Ah** (5 × 2).

Step 2 / 5

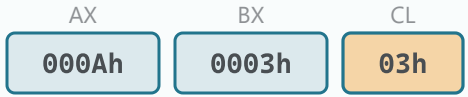
VARIABLES

AX = 000Ah | BX = 0003h | CL = 00h | CF = 0

The code is the expanded form of two SHIFTN calls. The assembler chose which branch to emit; no comparison happened at run time.

Code Trace: SHIFTN AX and SHIFTN BX, 3

```
1 org 100h
2 mov ax, 5
3 mov bx, 3
4 shl ax, 1       ; SHIFTN AX
▶ mov cl, 3       ; SHIFTN BX, 3
6 shl bx, cl
7 ret
```



AL = 00001010 CF = 0
BL = 00000011

ELSE branch

`SHIFTN BX, 3` : n = 3, ELSE branch → `MOV CL, 3` . On the 8086 a shift count above 1 goes in CL.

Step 3 / 5

VARIABLES

AX = 000Ah | BX = 0003h | CL = 03h | CF = 0

The code is the expanded form of two SHIFTN calls. The assembler chose which branch to emit; no comparison happened at run time.

Code Trace: SHIFTN AX and SHIFTN BX, 3

```
1 org 100h
2 mov ax, 5
3 mov bx, 3
4 shl ax, 1       ; SHIFTN AX
5 mov cl, 3       ; SHIFTN BX, 3
▶ shl bx, cl
7 ret
```

AX	BX	CL
000Ah	0018h	03h

AL = 00001010 CF = 0
BL = 00011000

`SHL BX, CL` : 3 × 2^3 = **0018h** (24). The same macro produced code of two different lengths: 2 bytes and 4 bytes.

Step 4 / 5

VARIABLES

AX = 000Ah | BX = 0018h | CL = 03h | CF = 0

The code is the expanded form of two SHIFTN calls. The assembler chose which branch to emit; no comparison happened at run time.

Code Trace: SHIFTN AX and SHIFTN BX, 3

```
1 org 100h
2 mov ax, 5
3 mov bx, 3
4 shl ax, 1       ; SHIFTN AX
5 mov cl, 3       ; SHIFTN BX, 3
6 shl bx, cl
▶ ret
```

AX	BX	CL
000Ah	0018h	03h

AL = 00001010 CF = 0
BL = 00011000

RET. The SHIFTN AX expansion did not touch CL; only the second expansion changed it.

Step 5 / 5

VARIABLES

AX = 000Ah | BX = 0018h | CL = 03h | CF = 0

The code is the expanded form of two SHIFTN calls. The assembler chose which branch to emit; no comparison happened at run time.

Debugging and Listing Macros

Seeing the expansion

The instruction list in emu8086's emulator window shows the expanded instructions, not macro names; single-stepping moves through these instructions.

Listing file

In MASM, `.LISTMACROALL` lists every line of the expansion, `.LISTMACRO` only the lines that generate code, and `.NOLISTMACRO` hides the expansion.

Error line

An error in the body is reported separately at every line that uses the macro. If the same error shows up several times, the cause is usually in the body.

Expand by hand first

Develop a new macro by writing and testing its body as plain code first; once it works, turn it into a parameterized macro.

Because LOCAL gives macro labels names like `??0000`, jump targets in the emulator don't show the source names; these numbers tell you which copy is running.

Common Macro Mistakes

Defining after use

The assembler cannot expand a call to a macro it does not know; put definitions at the top of the file or in an INCLUDE file.

Forgetting ENDM

The rest of the file is taken as the macro body; the error shows up much further down.

Labels not made LOCAL

Duplicate declaration on the second use.

RET in the body

A macro is not entered with CALL; RET takes the caller's return address.

Passing a clobbered register

In a call such as `ADD3 BX, AX, CX`, the body may change a parameter before reading it.

Unparenthesized expressions

`(wd)*(ht)` instead of `wd*ht`; otherwise precedence changes.

Mistaking a macro for a loop

`REPT 1000` produces 1000 copies; repeating at run time needs LOOP.

Using a long body many times

Code size grows with the number of uses; move long work into a procedure.

Lab: Macros

MyMacro: three parameters

CODE · **18 bytes** Expansion

```
 1 ORG 100h
 2 MyMacro  MACRO  p1, p2, p3
 3          MOV AX, p1
 4          MOV BX, p2
 5          MOV CX, p3
 6 ENDM
 7 MyMacro 1, 2, 3
+ MOV AX, 1
+ MOV BX, 2
+ MOV CX, 3
 8 MyMacro 4, 5, DX
+ MOV AX, 4
+ MOV BX, 5
+ MOV CX, DX
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	0012	CH	00	CL	12
DX	0000	DH	00	DL	00

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

FLAGS

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

CS:IP = 0700:0100 · step 0

MEMORY **DS = 0700h**

```
0100 B8 01 00 BB 02 00 B9 03
0108 00 B8 04 00 BB 05 00 8B
0110 CA C3 00 00 00 00 00 00
0118 00 00 00 00 00 00 00 00
0120 00 00 00 00 00 00 00 00
```

STACK **SS:SP**

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

OUTPUT

Program loaded: 18 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (0012h). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

Topic	Key point
Definition	`name MACRO p1, p2 … ENDM`; defined before use.
Expansion	Text substitution at assembly time; no CALL at run time.
Parameters	Substituted as text; their values are read on the line where they are used.
Size / speed	Macro: grows by the body at every use, faster. Procedure: a 3-byte CALL, CALL/RET cost.
LOCAL	The first line of the body, so labels are unique at each expansion.
Side effects	Preserve the registers a macro changes with PUSH/POP (as PUTC does).
Edge cases	Check CX = 0 before LOOP (JCXZ).

Exercises

1. Write the expanded form of the call `MyMacro AX, BX, CX`. Which lines are redundant?
2. Write an `ADD3 a, b, c` macro that adds two word variables and stores the sum in the third.
3. What are the code sizes if the PUTC macro is used 50 times and an equivalent procedure is called 50 times?
4. What error do you get when the POWER macro is used twice as `POWER base, exponent, result`? How do you fix it?
5. With exponent = 0, how many times does the original POWER macro execute MUL?
6. Which instructions must be added so that a macro clobbers no register other than AX?

▶ **Answers**

References

- emu8086 help: *8086 Assembler Tutorial, Part 10 (Macros)* and `emu8086.inc`.
- Microsoft, *MASM Programmer's Guide* — the MACRO, ENDM, LOCAL directives.
- K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014 — Chapter 10 (Macros).
- B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009.

The End

Chapter 9.1: Macros

Previous: Chapter 8.2: Procedures · Next: Chapter 9.2: The emu8086.inc Library