Chapter 8.2: Procedures

Microprocessors

Procedures | PROC / ENDP | CALL and RET | Stack and return address | Parameter passing | Stack frame | Preserving registers | Example programs | Common bugs

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_08_02_Procedures.html

INTRODUCTION

Learning Outcomes

Definition

Define procedures with PROC / ENDP; tell an assembler directive from a CPU instruction.

CALL / RET

Trace, through SP and IP, how the return address is pushed by CALL and popped by RET.

Parameters

Pass data to a procedure and get results back through registers, memory variables and the stack (BP frame).

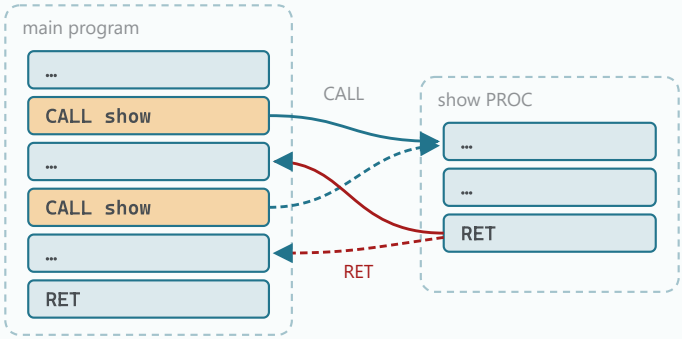
Safe procedures

Preserve registers with PUSH/POP, keep the stack balanced and avoid the typical bugs.

INTRODUCTION

What Is a Procedure?

- A procedure is a piece of code that performs **one specific task** (a function or subroutine in high-level languages).
- The program calls it with `CALL`.
- When the task is done, `RET` returns **right after** the point of the call.
- The code is written once and called from many places.
- Procedures make a program more **structured** and easier to follow.



Reuse
Less duplicated code and less memory.

Readability
The program is split into small, named tasks.

Maintenance
A bug is fixed in one place; each part is tested on its own.

DEFINITION

Defining a Procedure

```
name PROC       ; start (NEAR by default)
    ; procedure body goes here
    RET
name ENDP       ; end
```

- `name` : the procedure's name; it must be **the same at both ends**.
- The name follows the rules for labels (it cannot start with a digit).
- A procedure must contain at least one `RET` .

Where does a procedure go?

```
ORG 100h
    CALL p1
    RET           ; main program ends
p1 PROC           ; after the RET:
    ; ...           ; never fallen into
    RET
p1 ENDP
END
```

Placed after the main program's `RET` , procedures cannot be executed by accidentally "falling into" them. If one comes first, jump over it with `JMP` .

DEFINITION

Directives and Instructions

Item	Kind	Role
`PROC / ENDP`	Assembler directive	Tells the assembler where the procedure starts and its type (NEAR/FAR). **Not translated into machine code**; takes no memory.
`CALL`	CPU instruction	Pushes the return address and jumps to the procedure.
`RET`	CPU instruction	Pops the return address and jumps there. Used to return both from a procedure and from a .COM program.
`END`	Assembler directive	Marks the end of the source file.

CALL and RET are not assembler directives but **instructions** the CPU executes (CALL near: E8, RET: C3). Only PROC, ENDP and END are directives.



Why does RET return to the OS? When DOS starts a .COM program it pushes 0000h. The program's last RET loads it into IP; at CS:0000h (start of the PSP) sits an `INT 20h`, which ends the program.

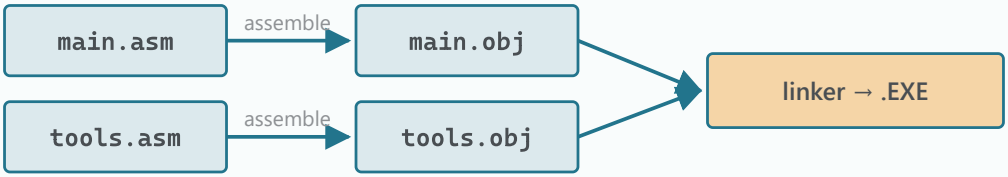
DEFINITION

Modular Programming

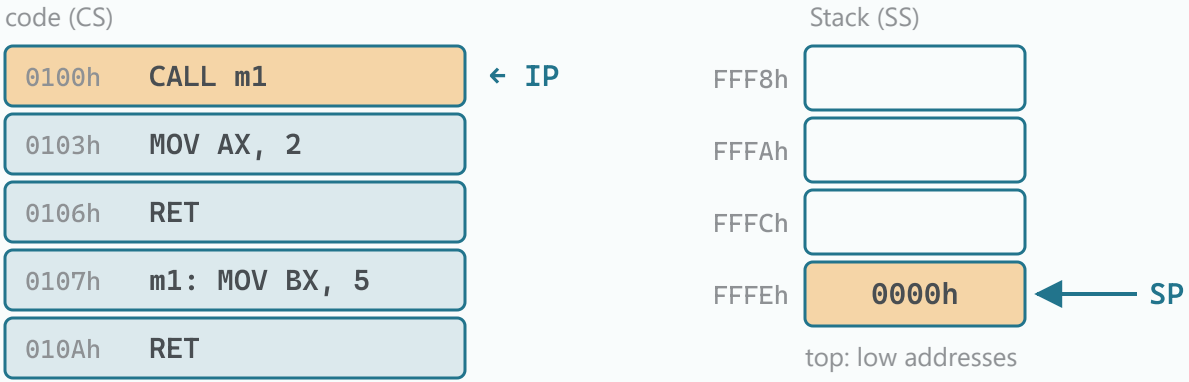
```
; main.asm
EXTRN show:NEAR   ; in another file
  ...
  CALL show

; tools.asm
PUBLIC show       ; exported
show PROC NEAR
  ...
  RET
show ENDP
```

- Large programs are split into procedures, and procedures into separate source files (modules).
- `PUBLIC` makes a procedure name visible to other modules; `EXTRN` declares that a name is defined in another module, along with its type (NEAR/FAR).
- Each module is assembled separately into an object file (.OBJ); the **linker** joins them and fills in the addresses in CALL instructions.
- Frequently used object files are collected in a library (.LIB); the linker pulls in only the procedures the program needs.
- In emu8086, shared code is usually added with `INCLUDE`: the file's text is inserted into the source and assembled as one piece (emu8086.inc).

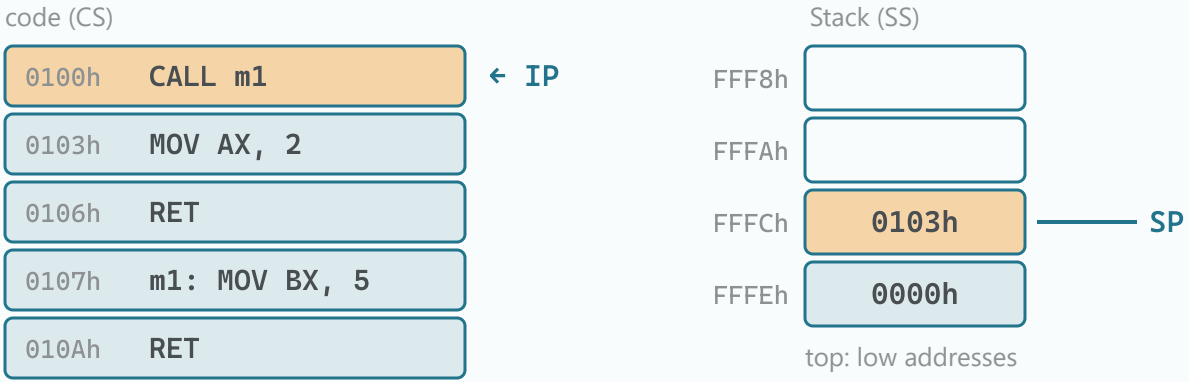


CALL and RET Step by Step



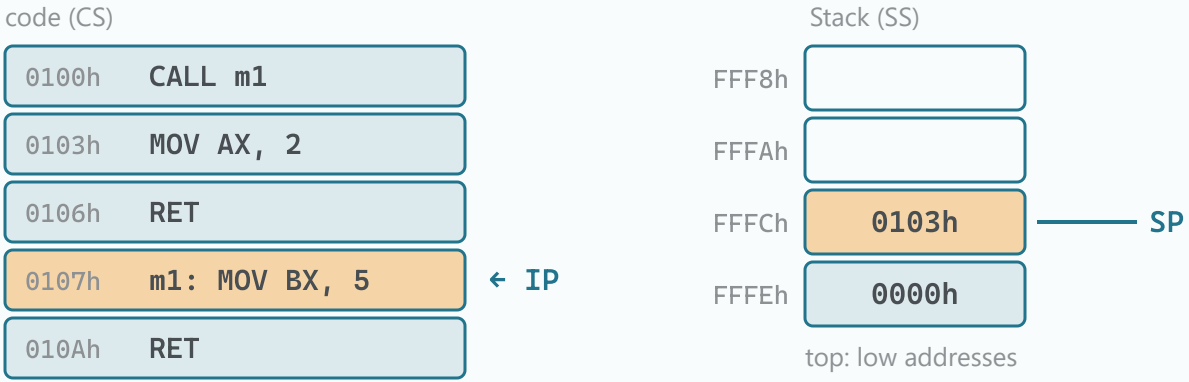
Start: IP = 0100h, SP = FFFEh. Next instruction: `CALL m1` (3 bytes: E8 04 00).

CALL and RET Step by Step



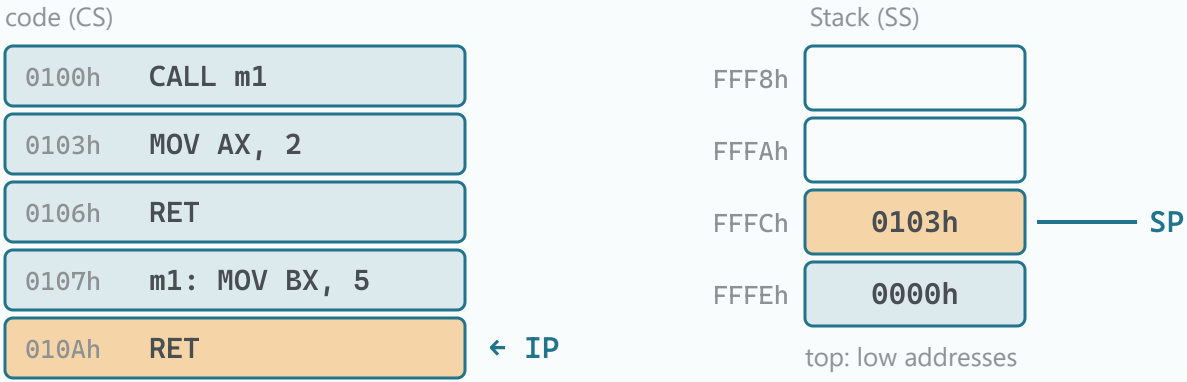
CALL first pushes the **return address** (the next instruction, 0103h): SP = FFFCh, [FFFCh] = 0103h.

CALL and RET Step by Step



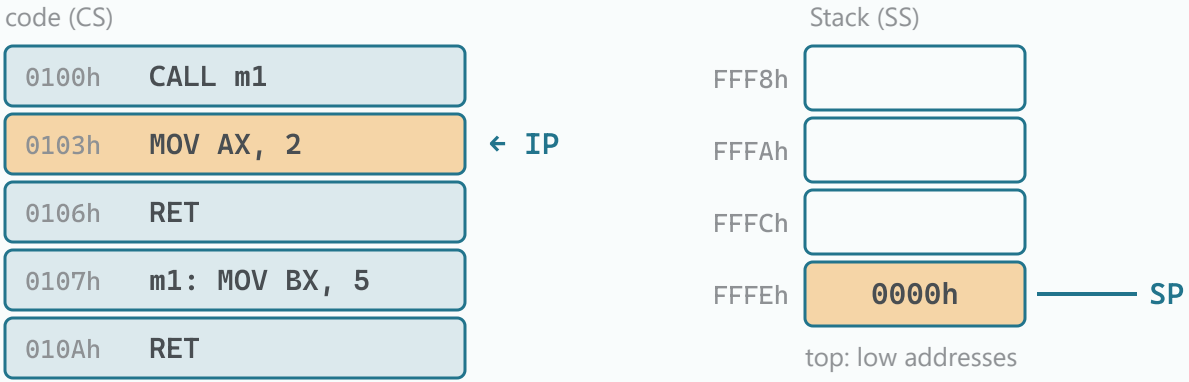
Then IP ← 0103h + 0004h = **0107h**: the first instruction of the procedure.

CALL and RET Step by Step



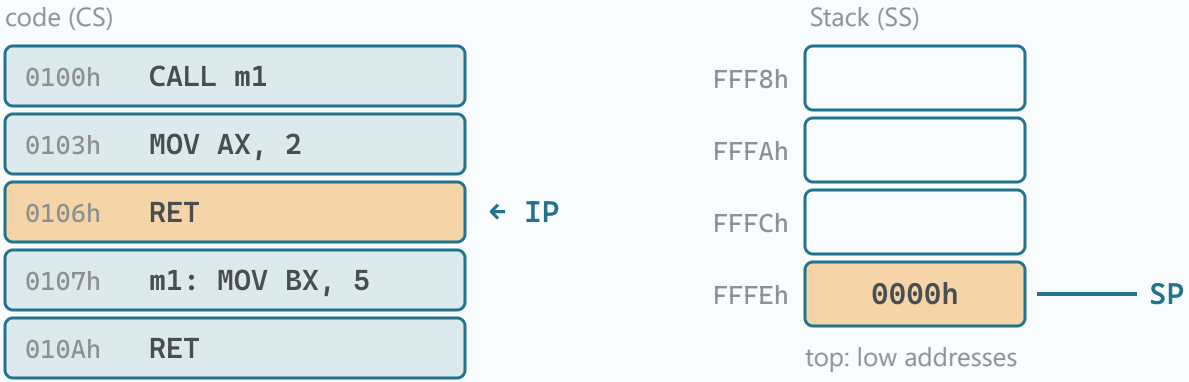
`MOV BX, 5` executed, IP = 010Ah. The stack is unchanged.

CALL and RET Step by Step



RET: IP ← [SP] = **0103h**, SP = FFFEh. Execution continues right after the CALL.

CALL and RET Step by Step

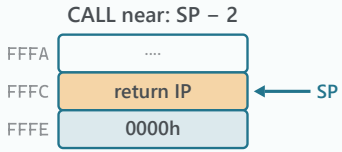


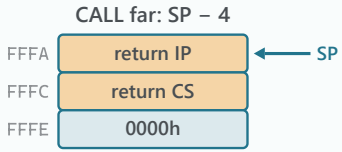
`MOV AX, 2` → IP = 0106h. The RET here pops 0000h and ends the program.

Kinds of CALL / RET

Instruction	Code	Bytes	Pushed / popped	Description
`CALL near`	E8 rel16	3	IP	Direct call within the same segment (default).
`CALL reg/mem`	FF /2	2–4	IP	Indirect near call: `CALL BX`, `CALL [table+SI]`.
`CALL far`	9A ofs seg	5	CS, then IP	Procedure in another segment; CS:IP change together.
`RET`	C3	1	IP	Near return.
`RETF`	CB	1	IP, then CS	Far return (FAR procedure).
`RET n`	C2 n16	3	IP, SP += n	On return, also discards n bytes of parameters from the stack.

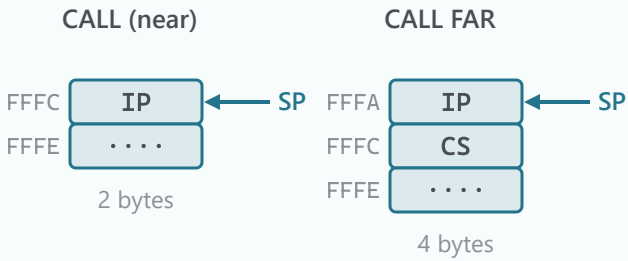
With `name PROC NEAR` (the default) or `name PROC FAR`, the assembler encodes each `RET` inside the procedure accordingly (C3 / CB). In single-segment .COM examples, procedures are NEAR.





Far Procedure: Stack Layout

- A FAR procedure lives in another code segment; reaching it requires changing both CS and IP.
- `CALL FAR` pushes CS first, then IP: 4 bytes on the stack. `RETF` pops IP first, then CS.
- A near `CALL` pushes only IP (2 bytes), and `RET` pops only IP.
- The procedure type and the return instruction must match: with `PROC FAR`, the assembler encodes the RET inside as RETF (CB).
- A .COM program runs in a single segment, so near calls are enough; .EXE programs with several code segments need far calls.



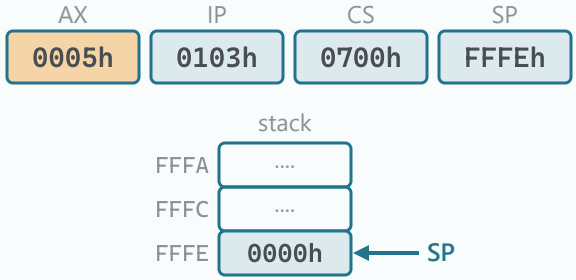
The stack order matches the order RETF pops them: IP, pushed last, is on top and is popped first.

Code Trace: PUSH CS + CALL with RETF

```
1  ORG  100h
▶ MOV  AX, 5
3  PUSH CS          ; first half of a far call
4  CALL farp        ; pushes IP
5  ADD  AX, 1
6  RET
7  farp PROC
8  SHL  AX, 1
9  RETF             ; IP first, then CS
10 farp ENDP
```

VARIABLES

`AX = 0005h` `IP = 0103h` `CS = 0700h` `SP = FFFEh`



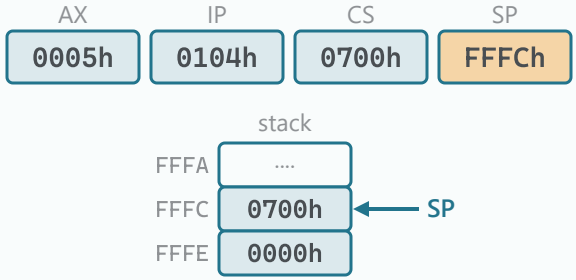
AX = 5. This code imitates the two pushes of a CALL FAR with two separate instructions.

Code Trace: PUSH CS + CALL with RETF

```
1  ORG  100h
2  MOV  AX, 5
▶  PUSH CS          ; first half of a far call
4  CALL farp        ; pushes IP
5  ADD  AX, 1
6  RET
7  farp PROC
8  SHL  AX, 1
9  RETF             ; IP first, then CS
10 farp ENDP
```

VARIABLES

AX = 0005h IP = 0104h CS = 0700h SP = FFFCh



`PUSH CS` : the code segment (0700h in this run) is pushed; the first half of a far call.

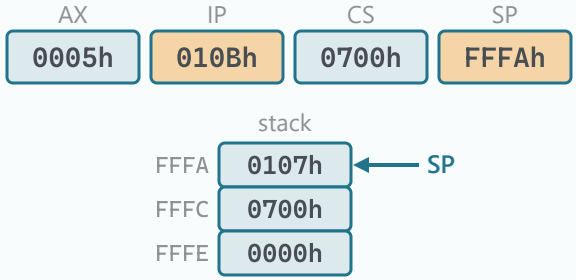
Step 2 / 6

Code Trace: PUSH CS + CALL with RETF

```
1  ORG  100h
2  MOV  AX, 5
3  PUSH CS          ; first half of a far call
▶  CALL farp        ; pushes IP
5  ADD  AX, 1
6  RET
7  farp PROC
8  SHL  AX, 1
9  RETF             ; IP first, then CS
10 farp ENDP
```

VARIABLES

AX = 0005h IP = 010Bh CS = 0700h SP = FFFAh



The near `CALL farp` pushes the return address (0107h) and jumps to 010Bh. The stack now has the far-call layout: IP on top, CS below it.

Step 3 / 6

Code Trace: PUSH CS + CALL with RETF

```
1  ORG  100h
2  MOV  AX, 5
3  PUSH CS            ; first half of a far call
4  CALL farp          ; pushes IP
5  ADD  AX, 1
6  RET
7  farp PROC
▶  SHL  AX, 1
9  RETF               ; IP first, then CS
10 farp ENDP
```

VARIABLES

AX = 000Ah | IP = 010Dh | CS = 0700h | SP = FFFAh

AX	IP	CS	SP
000Ah	010Dh	0700h	FFFAh

stack

FFFA	0107h ← SP
FFFC	0700h
FFFE	0000h

`SHL AX, 1` : AX = 000Ah.

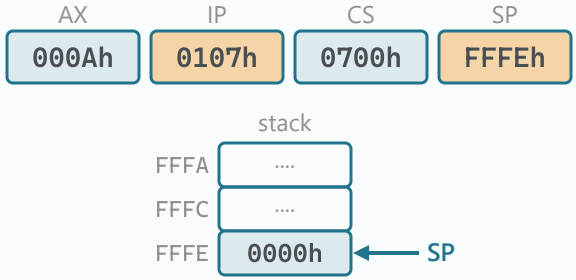
Step 4 / 6

Code Trace: PUSH CS + CALL with RETF

```
1  ORG  100h
2  MOV  AX, 5
3  PUSH CS          ; first half of a far call
4  CALL farp        ; pushes IP
5  ADD  AX, 1
6  RET
7  farp PROC
8  SHL  AX, 1
▶  RETF             ; IP first, then CS
10 farp ENDP
```

VARIABLES

AX = 000Ah IP = 0107h CS = 0700h SP = FFFEh



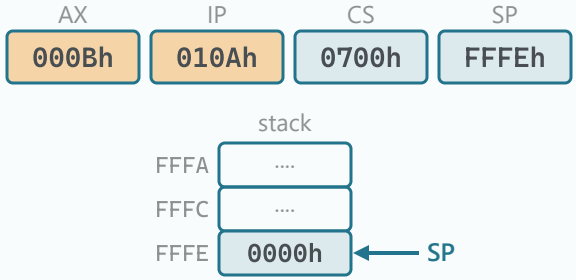
`RETF` pops two words: IP = 0107h, CS = 0700h; SP goes up by 4 to FFFEh.

Code Trace: PUSH CS + CALL with RETF

```
1  ORG  100h
2  MOV  AX, 5
3  PUSH CS          ; first half of a far call
4  CALL farp        ; pushes IP
▶  ADD  AX, 1
6  RET
7  farp PROC
8  SHL  AX, 1
9  RETF             ; IP first, then CS
10 farp ENDP
```

VARIABLES

`AX = 000Bh` `IP = 010Ah` `CS = 0700h` `SP = FFFEh`



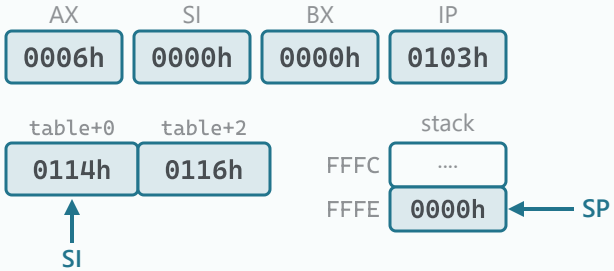
Execution continues at `ADD AX, 1` : AX = 000Bh. A real CALL FAR (9A, 5 bytes) does both pushes in one instruction.

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2          ; 2nd op (× 2)
3  CALL table[SI]      ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX             ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

AX = 6 | IP = 0103h | SP = FFFEh



add_one = 0114h · twice = 0116h

AX = 6. table is an array of words holding the addresses of two procedures (0114h, 0116h).

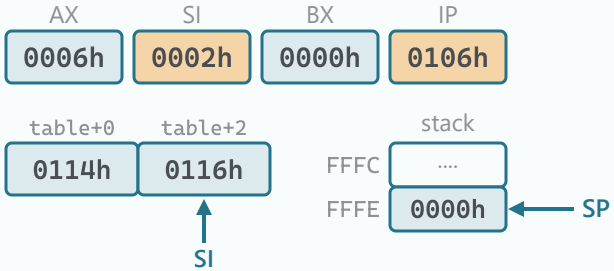
Step 1 / 9

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
▶  MOV  SI, 2          ; 2nd op (× 2)
3  CALL table[SI]      ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX             ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

AX = 6 | IP = 0106h | SP = FFFEh



add_one = 0114h · twice = 0116h

SI = 2: the 2nd entry of the table (each entry is 2 bytes).

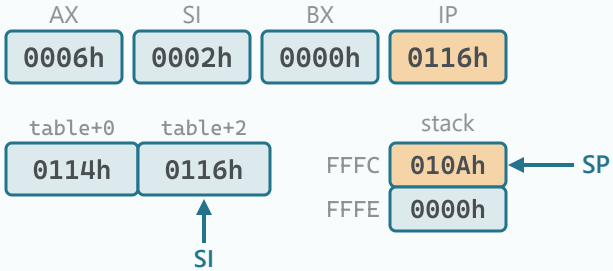
Step 2 / 9

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2          ; 2nd op (× 2)
3  CALL table[SI]      ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX             ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

AX = 6 | IP = 0116h | SP = FFFCh



add_one = 0114h · twice = 0116h

`CALL table[SI]` : the target is read from memory at [table+2] = 0116h. Return address 010Ah pushed, IP = 0116h.

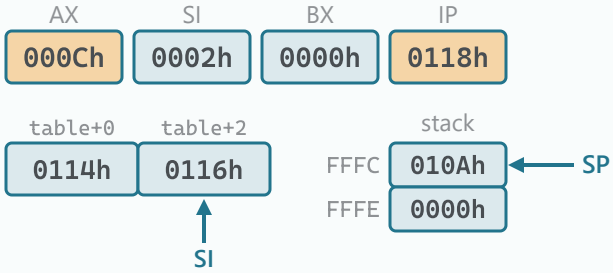
Step 3 / 9

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2        ; 2nd op (× 2)
3  CALL table[SI]    ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX           ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
▶  twice: SHL AX, 1
11 RET
```

VARIABLES

AX = **12** | IP = **0118h** | SP = **FFFCh**



add_one = 0114h · twice = 0116h

twice: `SHL AX, 1` → AX = 12.

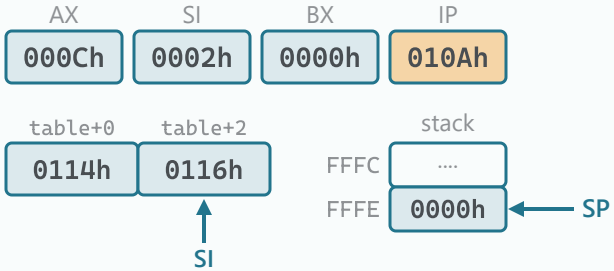
Step 4 / 9

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2        ; 2nd op (× 2)
3  CALL table[SI]    ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX           ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
10 twice: SHL AX, 1
▶  RET
```

VARIABLES

`AX = 12` `IP = 010Ah` `SP = FFFEh`



add_one = 0114h · twice = 0116h

`RET` : IP = 010Ah, back in the main program.

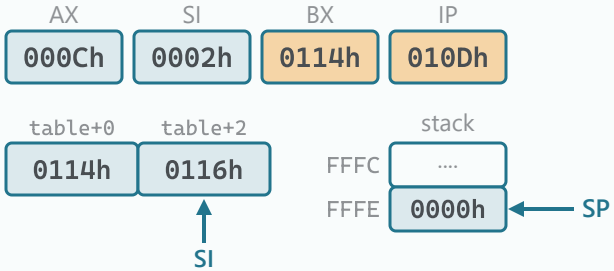
Step 5 / 9

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2          ; 2nd op (× 2)
3  CALL table[SI]      ; indirect, memory
▶  MOV  BX, OFFSET add_one
5  CALL BX             ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

AX = 12 | IP = 010Dh | SP = FFFEh



add_one = 0114h · twice = 0116h

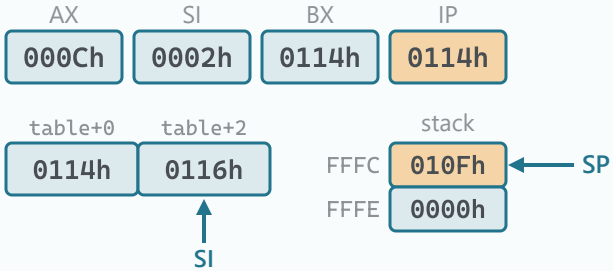
BX = OFFSET add_one = 0114h.

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2          ; 2nd op (× 2)
3  CALL table[SI]      ; indirect, memory
4  MOV  BX, OFFSET add_one
▶  CALL BX             ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
9  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

`AX = 12` `IP = 0114h` `SP = FFFCh`



add_one = 0114h · twice = 0116h

`CALL BX` : the target comes from a register. Return 010Fh pushed, IP = 0114h.

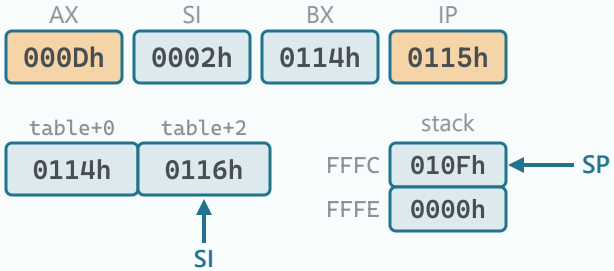
Step 7 / 9

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2        ; 2nd op (× 2)
3  CALL table[SI]    ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX           ; indirect, register
6  RET
7  table DW add_one, twice
   add_one: INC AX
9  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

AX = **13** | IP = **0115h** | SP = **FFFCh**



add_one = 0114h · twice = 0116h

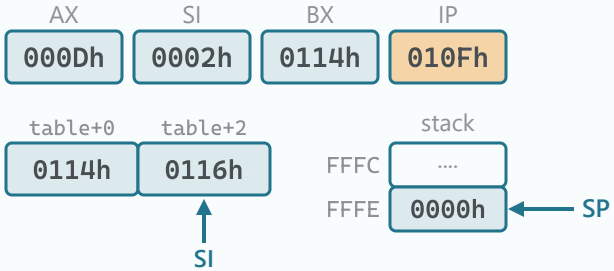
add_one: AX = 13.

Code Trace: Indirect CALL through a Table and a Register

```
1  MOV  AX, 6
2  MOV  SI, 2        ; 2nd op (× 2)
3  CALL table[SI]    ; indirect, memory
4  MOV  BX, OFFSET add_one
5  CALL BX           ; indirect, register
6  RET
7  table DW add_one, twice
8  add_one: INC AX
▶  RET
10 twice: SHL AX, 1
11 RET
```

VARIABLES

AX = 13 | IP = 010Fh | SP = FFFEh



add_one = 0114h · twice = 0116h

RET → 010Fh. Which procedure to call was chosen at run time through SI or BX.

Step 9 / 9

Menu with a Jump Table

```
MOV  AH, 1
INT  21h          ; AL = '1'..'3'
SUB  AL, '1'      ; 0..2
CMP  AL, 2
JA   invalid      ; range check
MOV  BL, AL
MOV  BH, 0
SHL  BX, 1        ; 2-byte entries
CALL ops[BX]
invalid: RET
ops DW sum, diff, prod
```

- Each option is a procedure; their addresses sit in order in `ops`.
- The key becomes a table index: '1' → 0, '2' → 1, '3' → 2. Addresses are 2 bytes, so the index is doubled (`SHL BX, 1`).
- `JA` is unsigned: key '0' gives 0 − 1 = FFh, which is above 2 and rejected too. One check covers both ends.
- Without the check, a bad key would call whatever bytes lie past the table.

Key	BX	Called
1	0	sum
2	2	diff
3	4	prod
0 · 9	—	none (RET)

Code Trace: A Simple Procedure

```
1 ORG     100h
▶ CALL    m1          ; 0100h: E8 04 00
3 MOV     AX, 2       ; 0103h: B8 02 00
4 RET                 ; 0106h: C3 → DOS
5 m1 PROC
6 MOV     BX, 5       ; 0107h: BB 05 00
7 RET                 ; 010Ah: C3 → caller
8 m1 ENDP
9 END
```

VARIABLES

IP = **0100h** SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

The program starts at 0100h. `ORG`, `PROC`, `ENDP`, `END` produce no bytes.

Step 1 / 6

Code Trace: A Simple Procedure

```
1 ORG     100h
▶ CALL    m1            ; 0100h: E8 04 00
3 MOV     AX, 2         ; 0103h: B8 02 00
4 RET                   ; 0106h: C3 → DOS
5 m1 PROC
6 MOV     BX, 5         ; 0107h: BB 05 00
7 RET                   ; 010Ah: C3 → caller
8 m1 ENDP
9 END
```

VARIABLES

`IP = 0107h` `SP = FFFCh`

CALL STACK

`[FFFCh] 0103h`

`[FFFEh] 0000h`

`CALL m1` : 0103h is pushed, IP = 0107h.

Step 2 / 6

Code Trace: A Simple Procedure

```
1 ORG     100h
2 CALL    m1          ; 0100h: E8 04 00
3 MOV     AX, 2       ; 0103h: B8 02 00
4 RET                 ; 0106h: C3 → DOS
5 m1 PROC
▶ MOV     BX, 5       ; 0107h: BB 05 00
7 RET                 ; 010Ah: C3 → caller
8 m1 ENDP
9 END
```

VARIABLES

IP = **010Ah** SP = **FFFCh** BX = **0005h**

CALL STACK

[FFFCh] 0103h

[FFFEh] 0000h

BX = 0005h.

Code Trace: A Simple Procedure

```
1 ORG    100h
2 CALL   m1          ; 0100h: E8 04 00
3 MOV    AX, 2       ; 0103h: B8 02 00
4 RET                ; 0106h: C3 → DOS
5 m1 PROC
6 MOV    BX, 5       ; 0107h: BB 05 00
▶ RET                ; 010Ah: C3 → caller
8 m1 ENDP
9 END
```

VARIABLES

IP = **0103h** SP = **FFFEh** BX = **0005h**

CALL STACK

`[FFFEh] 0000h`

`RET` : 0103h is popped → IP = 0103h, SP = FFFEh.

Code Trace: A Simple Procedure

```
1 ORG     100h
2 CALL    m1          ; 0100h: E8 04 00
▶ MOV     AX, 2       ; 0103h: B8 02 00
4 RET                 ; 0106h: C3 → DOS
5 m1 PROC
6 MOV     BX, 5       ; 0107h: BB 05 00
7 RET                 ; 010Ah: C3 → caller
8 m1 ENDP
9 END
```

VARIABLES

IP = **0106h** SP = **FFFEh** AX = **0002h** BX = **0005h**

CALL STACK

[FFFEh] 0000h

AX = 0002h.

Step 5 / 6

Code Trace: A Simple Procedure

```
1 ORG    100h
2 CALL   m1          ; 0100h: E8 04 00
3 MOV    AX, 2       ; 0103h: B8 02 00
▶ RET                ; 0106h: C3 → DOS
5 m1 PROC
6 MOV    BX, 5       ; 0107h: BB 05 00
7 RET                ; 010Ah: C3 → caller
8 m1 ENDP
9 END
```

VARIABLES

`IP = 0000h` `SP = 0000h` `AX = 0002h` `BX = 0005h`

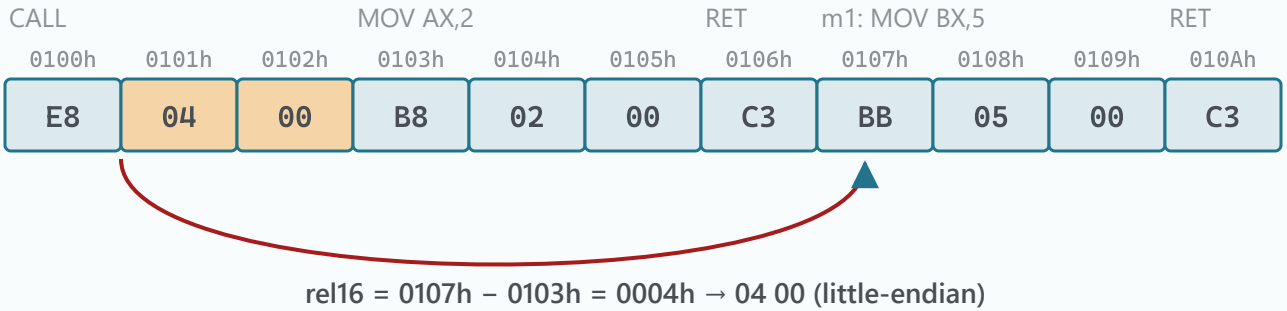
CALL STACK

empty

`RET` : 0000h is popped → IP = 0000h, SP = FFFEh + 2 = 0000h. The `INT 20h` in the PSP ends the program. Result: AX = 2, BX = 5.

Step 6 / 6

Machine Code in Memory



rel16 = 0107h − 0103h = 0004h → 04 00 (little-endian)

- The example program takes only **11 bytes** of memory: `E8 04 00 B8 02 00 C3 BB 05 00 C3`.
- `CALL m1` stores the target not as an absolute address but as a **displacement from the next instruction** (rel16). That is why the code still works when moved elsewhere in memory.
- 16-bit values are stored **little-endian**: low byte first (04), then high byte (00).
- The procedure name (`m1`) exists only for the assembler; the machine code keeps just the address 0107h.

Cost of CALL and RET

Instruction (8086)	Clocks
`CALL near`	19
`CALL reg`	16
`CALL FAR`	28
`RET`	8
`RETF`	18
`PUSH reg`	11
`POP reg`	8

Call overhead

- Call overhead of a procedure that saves two registers: 19 + 8 + 2 × (11 + 8) = **65 clocks**.
- On a 5 MHz 8086 a clock is 200 ns: 65 × 200 ns = **13 µs**.
- Called 1000 times, the overhead is 13 ms.
- A macro has no such overhead, but copies its body at every use.

Short, very frequent jobs (printing one character) suit a macro; long or rarely called jobs suit a procedure. The table is for register and direct forms; memory-operand forms add the effective-address time.

Ways to Pass Parameters

1. In registers

The easiest and fastest way. E.g. AL and BL as two parameters, the result in AX. Registers are limited.

2. In memory variables

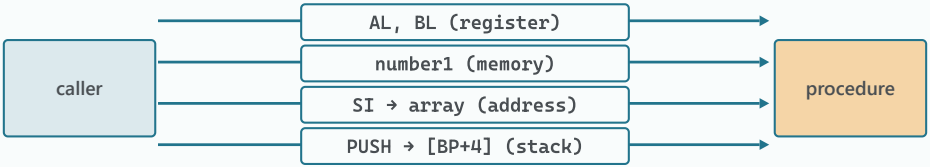
The procedure reads/writes shared variables such as `number1`, `result`. Easy, but it ties the procedure to those variables.

3. By address (pointer)

The **address** of an array goes in SI/DI/BX; long data is processed without copying (print_me).

4. On the stack

Parameters are pushed with `PUSH`, read through BP and removed with `RET n`. The method compilers use.



Whichever way you choose, document the **inputs**, **outputs** and **clobbered registers** in a comment at the top of the procedure.

By Value or by Reference?

Pass by value

The procedure gets a **copy** of the data (AX = 5). Even if it changes the copy, the variable in memory stays the same. Fast and safe for small data such as a single number.

Pass by reference

The procedure gets the data's **address** (SI = OFFSET a). Through `[SI]` it reads and can change the variable itself. For arrays, strings and variables the procedure must update.

```
MOV  AX, a        ; value: AX holds a copy of a
CALL twice        ; AX doubles, a stays the same
LEA  SI, a        ; address: SI = OFFSET a
CALL twice_m      ; [SI] doubles, a changes
```

`LEA SI, a` and `MOV SI, OFFSET a` give the same result: the address of a. `MOV SI, a` instead loads the **value stored in** a, not its address; a common mix-up when passing by reference.

Code Trace: Swap by Reference

```
1  ORG  100h
▶ LEA  SI, a
▶ LEA  DI, b
4  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
7  PUSH AX
8  MOV  AX, [SI]
9  XCHG AX, [DI]
10 MOV  [SI], AX
11 POP  AX
12 RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

`SI = 0113h` `DI = 0115h` `AX = 0000h`
`SP = FFFEh` `a = 1234h` `b = 00FFh`

SI	DI	AX	SP
0113h	0115h	0000h	FFFEh

	a (0113)	b (0115)
memory	1234h	00FFh

SI and DI carry the addresses of a and b (0113h, 0115h): the procedure gets locations, not values.

Step 1 / 7

Code Trace: Swap by Reference

```
1  ORG  100h
2  LEA  SI, a
3  LEA  DI, b
▶  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
7  PUSH AX
8  MOV  AX, [SI]
9  XCHG AX, [DI]
10 MOV  [SI], AX
11 POP  AX
12 RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

SI = **0113h** | DI = **0115h** | AX = **0000h** | SP = **FFFCh** | a = **1234h** | b = **00FFh**

SI	DI	AX	SP
0113h	0115h	0000h	FFFCh

	a (0113)	b (0115)
memory	1234h	00FFh

`CALL swap` : return address 0109h pushed, SP = FFFCh.

Step 2 / 7

Code Trace: Swap by Reference

```
1  ORG  100h
2  LEA  SI, a
3  LEA  DI, b
4  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
▶  PUSH AX
8  MOV  AX, [SI]
9  XCHG AX, [DI]
10 MOV  [SI], AX
11 POP  AX
12 RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

SI = **0113h** | DI = **0115h** | AX = **0000h** | SP = **FFFAh** | a = **1234h** | b = **00FFh**





swap will use AX, so it saves it first (SP = FFFAh).

Step 3 / 7

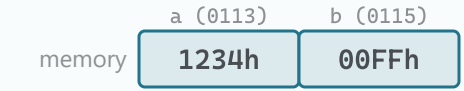
Code Trace: Swap by Reference

```
1  ORG  100h
2  LEA  SI, a
3  LEA  DI, b
4  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
7  PUSH AX
▶  MOV  AX, [SI]
9  XCHG AX, [DI]
10 MOV  [SI], AX
11 POP  AX
12 RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

SI = **0113h** | DI = **0115h** | AX = **1234h** | SP = **FFFAh** | a = **1234h** | b = **00FFh**





`MOV AX, [SI]` : reads the value of a (1234h).

Step 4 / 7

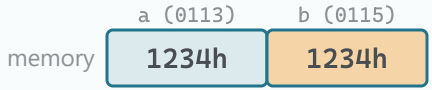
Code Trace: Swap by Reference

```
1  ORG  100h
2  LEA  SI, a
3  LEA  DI, b
4  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
7  PUSH AX
8  MOV  AX, [SI]
▶  XCHG AX, [DI]
10 MOV  [SI], AX
11 POP  AX
12 RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

SI = **0113h** | DI = **0115h** | AX = **00FFh** | SP = **FFFAh** | a = **1234h** | b = **1234h**





`XCHG AX, [DI]` : AX and b trade places in one instruction: b = 1234h, AX = 00FFh.

Step 5 / 7

Code Trace: Swap by Reference

```
1  ORG  100h
2  LEA  SI, a
3  LEA  DI, b
4  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
7  PUSH AX
8  MOV  AX, [SI]
9  XCHG AX, [DI]
▶  MOV  [SI], AX
11 POP  AX
12 RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

`SI = 0113h` `DI = 0115h` `AX = 00FFh`
`SP = FFFAh` `a = 00FFh` `b = 1234h`

SI	DI	AX	SP
0113h	0115h	00FFh	FFFAh

	a (0113)	b (0115)
memory	00FFh	1234h

`MOV [SI], AX` : a = 00FFh. The swap is done in memory.

Step 6 / 7

Code Trace: Swap by Reference

```
1  ORG  100h
2  LEA  SI, a
3  LEA  DI, b
4  CALL swap
5  RET
6  swap PROC          ; In: SI, DI (addrs)
7  PUSH AX
8  MOV  AX, [SI]
9  XCHG AX, [DI]
10 MOV  [SI], AX
▶  POP  AX
▶  RET
13 swap ENDP
14 a    DW 1234h
15 b    DW 00FFh
```

VARIABLES

SI = **0113h** | DI = **0115h** | AX = **0000h** | SP = **FFFEh** | a = **00FFh** | b = **1234h**

SI	DI	AX	SP
0113h	0115h	0000h	FFFEh

	a (0113)	b (0115)
memory	00FFh	1234h

`POP AX` and `RET`: AX gets its old value (0000h) back, SP = FFFEh; a and b stay swapped.

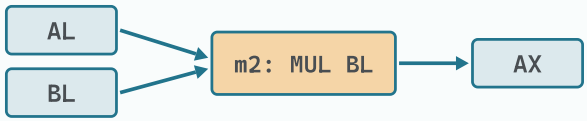
Step 7 / 7

Parameters in Registers

- In the example, the **AL** and **BL** registers hold the two parameters.
- The `m2` procedure **multiplies** AL by BL and leaves the result in **AX**.
- `CALL m2` calls the procedure; `RET` returns to the caller.
- m2 is called twice: in the second call AL carries the **result** of the first, because MUL changes AX (and so AL).

MUL source	Operation	Result
8-bit	AL × source	AX
16-bit	AX × source	DX:AX

MUL is unsigned; if the upper half of the result (AH or DX) is not 0, CF = OF = 1. The source cannot be an immediate: `MUL 2` is invalid.



input registers → procedure → output register

Code Trace: Parameters in Registers

```
1  ORG     100h
▶ MOV     AL, 1          ; 1st parameter
3  MOV     BL, 2          ; 2nd parameter
4  CALL    m2             ; 0104h: E8 04 00
5  CALL    m2             ; 0107h: E8 01 00
6  RET                    ; 010Ah: to DOS
7  m2 PROC                ; 010Bh
8  MUL     BL             ; AX = AL * BL
9  RET                    ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = **0102h** SP = **FFFEh** AL = **01h**

CALL STACK

[FFFEh] 0000h

AL = 1 (MOV AL, 1 = B0 01, 2 bytes).

Step 1 / 9

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1          ; 1st parameter
▶  MOV    BL, 2          ; 2nd parameter
4  CALL   m2             ; 0104h: E8 04 00
5  CALL   m2             ; 0107h: E8 01 00
6  RET                   ; 010Ah: to DOS
7  m2 PROC               ; 010Bh
8  MUL    BL             ; AX = AL * BL
9  RET                   ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = **0104h** SP = **FFFEh** AL = **01h** BL = **02h**

CALL STACK

[FFFEh] 0000h

BL = 2.

Step 2 / 9

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1        ; 1st parameter
3  MOV    BL, 2        ; 2nd parameter
▶  CALL   m2           ; 0104h: E8 04 00
5  CALL   m2           ; 0107h: E8 01 00
6  RET                 ; 010Ah: to DOS
7  m2 PROC             ; 010Bh
8  MUL    BL           ; AX = AL * BL
9  RET                 ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = **010Bh** SP = **FFFCh** AL = **01h** BL = **02h**

CALL STACK

[FFFCh] 0107h

[FFFEh] 0000h

1st call: return address 0107h pushed, IP = 010Bh.

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1          ; 1st parameter
3  MOV    BL, 2          ; 2nd parameter
4  CALL   m2             ; 0104h: E8 04 00
5  CALL   m2             ; 0107h: E8 01 00
6  RET                   ; 010Ah: to DOS
7  m2 PROC               ; 010Bh
▶  MUL    BL             ; AX = AL * BL
9  RET                   ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = 010Dh | SP = FFFCh | AL = 02h | BL = 02h | AX = 0002h | CF = 0

CALL STACK

[FFFCh] 0107h

[FFFEh] 0000h

`MUL BL`: AX = 1 × 2 = 0002h. AH = 0 → CF = OF = 0.

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1        ; 1st parameter
3  MOV    BL, 2        ; 2nd parameter
4  CALL   m2           ; 0104h: E8 04 00
5  CALL   m2           ; 0107h: E8 01 00
6  RET                 ; 010Ah: to DOS
7  m2 PROC             ; 010Bh
8  MUL    BL           ; AX = AL * BL
▶  RET                 ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = **0107h** SP = **FFFEh** AL = **02h**
BL = **02h** AX = **0002h** CF = **0**

CALL STACK

`[FFFEh] 0000h`

`RET` → IP = 0107h: the second CALL.

Step 5 / 9

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1          ; 1st parameter
3  MOV    BL, 2          ; 2nd parameter
4  CALL   m2             ; 0104h: E8 04 00
▶  CALL   m2             ; 0107h: E8 01 00
6  RET                   ; 010Ah: to DOS
7  m2 PROC               ; 010Bh
8  MUL    BL             ; AX = AL * BL
9  RET                   ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = **010Bh** | SP = **FFFCh** | AL = **02h** | BL = **02h** | AX = **0002h** | CF = **0**

CALL STACK

- [FFFCh] 010Ah
- [FFFEh] 0000h

2nd call: this time the return address **010Ah** is pushed. Same procedure, different return point.

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1          ; 1st parameter
3  MOV    BL, 2          ; 2nd parameter
4  CALL   m2             ; 0104h: E8 04 00
5  CALL   m2             ; 0107h: E8 01 00
6  RET                   ; 010Ah: to DOS
7  m2 PROC               ; 010Bh
▶  MUL    BL             ; AX = AL * BL
9  RET                   ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = 010Dh | SP = FFFCh | AL = 04h | BL = 02h | AX = 0004h | CF = 0

CALL STACK

[FFFCh] 010Ah

[FFFEh] 0000h

`MUL BL` : AL is now 2 → AX = 2 × 2 = 0004h.

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1        ; 1st parameter
3  MOV    BL, 2        ; 2nd parameter
4  CALL   m2           ; 0104h: E8 04 00
5  CALL   m2           ; 0107h: E8 01 00
6  RET                 ; 010Ah: to DOS
7  m2 PROC             ; 010Bh
8  MUL    BL           ; AX = AL * BL
▶  RET                 ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = 010Ah | SP = FFFEh | AL = 04h | BL = 02h | AX = 0004h | CF = 0

CALL STACK

[FFFEh] 0000h

`RET` → IP = 010Ah.

Step 8 / 9

Code Trace: Parameters in Registers

```
1  ORG    100h
2  MOV    AL, 1          ; 1st parameter
3  MOV    BL, 2          ; 2nd parameter
4  CALL   m2             ; 0104h: E8 04 00
5  CALL   m2             ; 0107h: E8 01 00
▶  RET                   ; 010Ah: to DOS
7  m2 PROC               ; 010Bh
8  MUL    BL             ; AX = AL * BL
9  RET                   ; to caller
10 m2 ENDP
11 END
```

VARIABLES

IP = **0000h** SP = **0000h** AL = **04h**
BL = **02h** AX = **0004h** CF = **0**

CALL STACK

empty

`RET` : back to DOS. Result AX = 0004h.

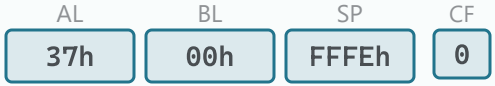
Step 9 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit       ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit       ; CF = 1 → not a digit
6  RET
7  digit PROC       ; In: AL, Out: CF
8  SUB  AL, '0'
9  CMP  AL, 10      ; 0..9 → borrow, CF = 1
10 CMC              ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

AL = **37h** | BL = **00h** | CF = **0**



AL = 55

The digit procedure converts the character in AL to a number and reports success in **CF** (as DOS services do). AL = '7' = 37h.

Step 1 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
▶ CALL digit        ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit       ; CF = 1 → not a digit
6  RET
7  digit PROC       ; In: AL, Out: CF
▶ SUB  AL, '0'
9  CMP  AL, 10      ; 0..9 → borrow, CF = 1
10 CMC              ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

AL = 07h | BL = 00h | CF = 0

AL	BL	SP	CF
07h	00h	FFFCh	0

AL = 7

`CALL digit` ; `SUB AL, '0'` : 37h − 30h = 7.

Step 2 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit       ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit       ; CF = 1 → not a digit
6  RET
7  digit PROC       ; In: AL, Out: CF
8  SUB  AL, '0'
▶  CMP  AL, 10      ; 0..9 → borrow, CF = 1
10 CMC              ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

`AL = 07h` `BL = 00h` `CF = 1`

AL	BL	SP	CF
07h	00h	FFFCh	1

AL = 7

`CMP AL, 10` : 7 − 10 needs a borrow → CF = 1 (value in 0...9).

Step 3 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit       ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit       ; CF = 1 → not a digit
6  RET
7  digit PROC       ; In: AL, Out: CF
8  SUB  AL, '0'
9  CMP  AL, 10      ; 0..9 → borrow, CF = 1
▶  CMC              ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

`AL = 07h` `BL = 00h` `CF = 0`

AL	BL	SP	CF
07h	00h	FFFCh	0

AL = 7

`CMC` : CF = 0 → “a digit”.

Step 4 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit        ; CF = 0 → AL = value
▶  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit        ; CF = 1 → not a digit
6  RET
7  digit PROC        ; In: AL, Out: CF
8  SUB  AL, '0'
9  CMP  AL, 10       ; 0..9 → borrow, CF = 1
10 CMC               ; digit: CF = 0
▶  RET
12 digit ENDP
```

VARIABLES

AL = 07h | BL = 07h | CF = 0

AL	BL	SP	CF
07h	07h	FFFEh	0

AL = 7 result: digit

RET; the caller can test CF (JC/JNC). BL = 7 saved.

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit        ; CF = 0 → AL = value
3  MOV  BL, AL
▶  MOV  AL, 'x'
▶  CALL digit        ; CF = 1 → not a digit
6  RET
7  digit PROC        ; In: AL, Out: CF
▶  SUB  AL, '0'
9  CMP  AL, 10       ; 0..9 → borrow, CF = 1
10 CMC               ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

AL = **48h** | BL = **07h** | CF = **0**

AL	BL	SP	CF
48h	07h	FFFCh	0

AL = 72

AL = 'x' = 78h; second call: 78h − 30h = 48h (72).

Step 6 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit        ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit        ; CF = 1 → not a digit
6  RET
7  digit PROC        ; In: AL, Out: CF
8  SUB  AL, '0'
▶  CMP  AL, 10       ; 0..9 → borrow, CF = 1
10 CMC               ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

`AL = 48h` `BL = 07h` `CF = 0`

AL	BL	SP	CF
48h	07h	FFFCh	0

AL = 72

`CMP AL, 10` : 72 ≥ 10, no borrow → CF = 0.

Step 7 / 9

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit       ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit       ; CF = 1 → not a digit
6  RET
7  digit PROC       ; In: AL, Out: CF
8  SUB  AL, '0'
9  CMP  AL, 10      ; 0..9 → borrow, CF = 1
▶  CMC              ; digit: CF = 0
11 RET
12 digit ENDP
```

VARIABLES

`AL = 48h` `BL = 07h` `CF = 1`

AL	BL	SP	CF
48h	07h	FFFCh	1

AL = 72

`CMC` : CF = 1 → “not a digit”. AL is now garbage (48h) and must not be used.

Code Trace: Returning the Result in CF

```
1  MOV  AL, '7'
2  CALL digit       ; CF = 0 → AL = value
3  MOV  BL, AL
4  MOV  AL, 'x'
5  CALL digit       ; CF = 1 → not a digit
6  RET
7  digit PROC       ; In: AL, Out: CF
8  SUB  AL, '0'
9  CMP  AL, 10      ; 0..9 → borrow, CF = 1
10 CMC              ; digit: CF = 0
▶  RET
12 digit ENDP
```

VARIABLES

`AL = 48h` `BL = 07h` `CF = 1`

AL	BL	SP	CF
48h	07h	FFFEh	1

AL = 72 result: not a digit

RET: the result is in the flag. The caller can branch with `JC error`.

Step 9 / 9

Returning Several Results

```
divmod PROC      ; In: AX, BL (divisor)
CMP  BL, 0       ; Out: AL = quotient
JE   error       ;      AH = remainder
DIV  BL          ;      CF = 1: BL is 0
CLC
RET
error: STC
RET
divmod ENDP
```

- **Register pair**: a 32-bit result in DX:AX (like 16-bit MUL), quotient and remainder in AL and AH.
- **Flag + register**: CF reports success, the register holds the value; the caller tests `JC` first.
- **Memory**: any number of results are written to a buffer at a given address.

Input	Result
AX = 100, BL = 7	AL = 0Eh (14), AH = 02h, CF = 0
AX = 100, BL = 0	CF = 1, AX unchanged (0064h)
AX = 1000, BL = 2	quotient 500 does not fit in AL: divide error (INT 0)

This procedure has two RETs; that is safe because nothing was pushed. In procedures that push, all paths meet at a single exit label.

Preserving Registers

Problem

If a procedure quietly changes a register the caller relies on (e.g. the loop counter CX), the caller breaks.

Documentation

```
; m2: computes AL * BL
; In      : AL, BL (unsigned)
; Out     : AX = AL * BL
; Clobber: CF, OF
```

Solution: PUSH / POP

```
show PROC
    PUSH AX        ; save what
    PUSH SI        ; you use
    ; ... AX and SI are used
    POP  SI        ; in REVERSE
    POP  AX        ; order
    RET
show ENDP
```

The register that returns the result (e.g. AX) is not preserved; otherwise the result would be lost.

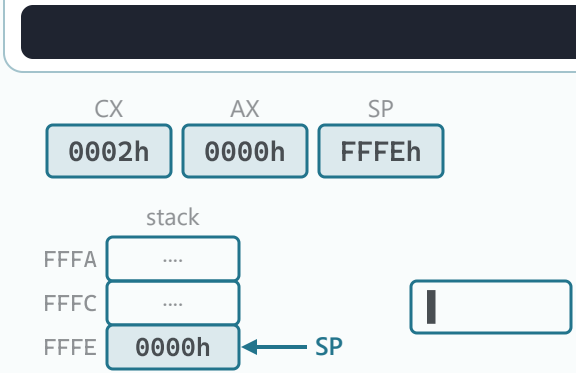
Code Trace: Saving the Counter with PUSH CX

```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = **2** SP = **FFFEh**

OUTPUT



The main loop prints 2 rows: counter CX = 2. The stars procedure also uses CX in its inner loop.

Step 1 / 10

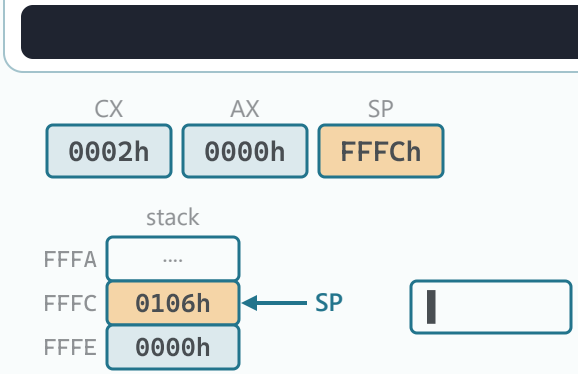
Code Trace: Saving the Counter with PUSH CX

```
1  MOV  CX, 2        ; main loop: 2 rows
▶  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = **2** SP = **FFFCh**

OUTPUT



`CALL stars` : return address 0106h pushed, SP = FFFCh.

Step 2 / 10

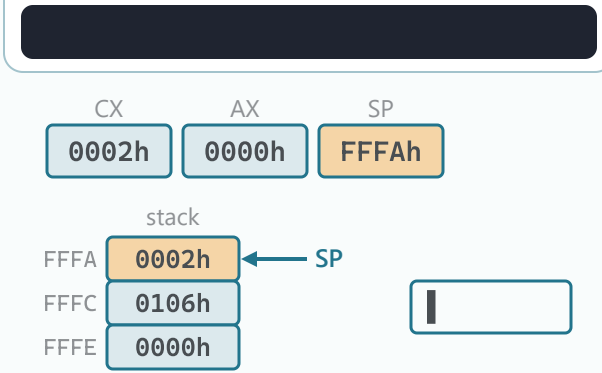
Code Trace: Saving the Counter with PUSH CX

```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
▶  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

`CX = 2` `SP = FFFAh`

OUTPUT



`PUSH CX` : the caller's counter (2) is saved on the stack.

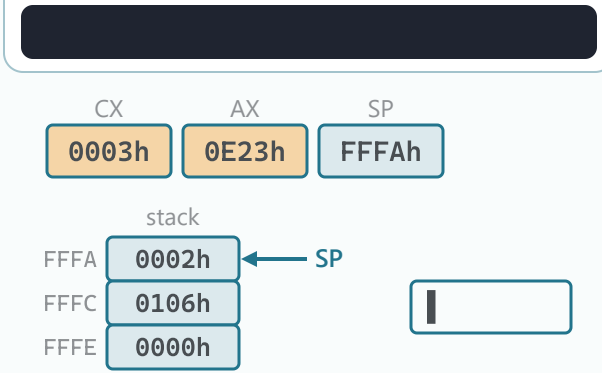
Code Trace: Saving the Counter with PUSH CX

```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
▶  MOV  CX, 3        ; inner counter
▶  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = 3 | SP = FFFAh

OUTPUT



Inner counter CX = 3; AX = 0E23h (AH = 0Eh teletype, AL = '#').

Code Trace: Saving the Counter with PUSH CX

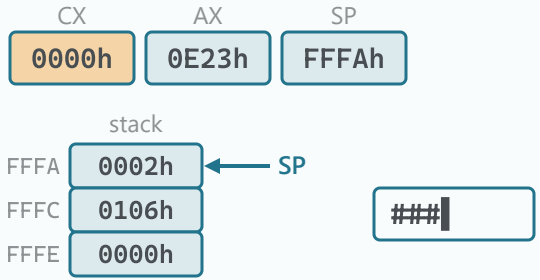
```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
▶  inner:  INT 10h
▶  LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = 0 | SP = FFFAh

OUTPUT

```
###
```



The inner loop runs `INT 10h` three times: ###. After LOOP, CX = 0.

Step 5 / 10

Code Trace: Saving the Counter with PUSH CX

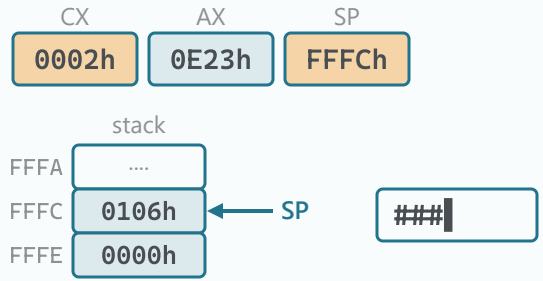
```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
▶ POP  CX            ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = 2 SP = FFFCh

OUTPUT

```
###
```



`POP CX` : the caller's counter is back, CX = 2.

Step 6 / 10

Code Trace: Saving the Counter with PUSH CX

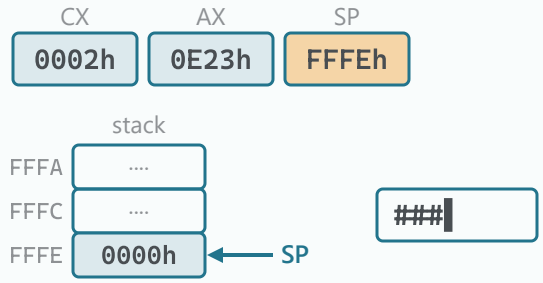
```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
3  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
▶  RET
13 stars ENDP
```

VARIABLES

CX = **2** SP = **FFFEh**

OUTPUT

```
###
```



`RET` : returns to 0106h, SP = FFFEh.

Step 7 / 10

Code Trace: Saving the Counter with PUSH CX

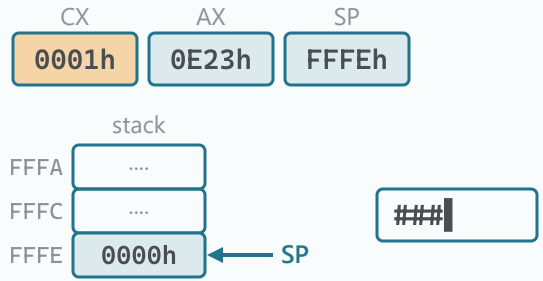
```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
▶  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = 1 | SP = FFFEh

OUTPUT

```
###
```



Main `LOOP row` : CX = 1 → call again for the second row.

Code Trace: Saving the Counter with PUSH CX

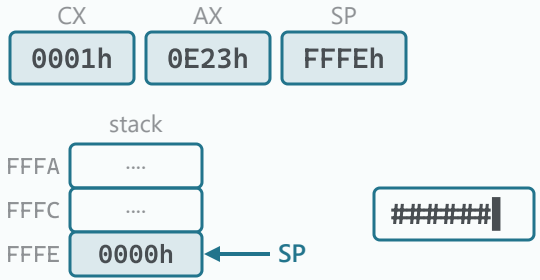
```
1  MOV  CX, 2        ; main loop: 2 rows
   row: CALL stars
3  LOOP row
4  RET
5  stars PROC
   PUSH CX           ; caller's counter
   MOV  CX, 3        ; inner counter
   MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
   inner:  INT 10h
   LOOP inner
   POP  CX           ; counter back
   RET
13 stars ENDP
```

VARIABLES

CX = **1** SP = **FFFEh**

OUTPUT

```
######
```



The second call works the same way: ### is printed, CX = 1 restored.

Code Trace: Saving the Counter with PUSH CX

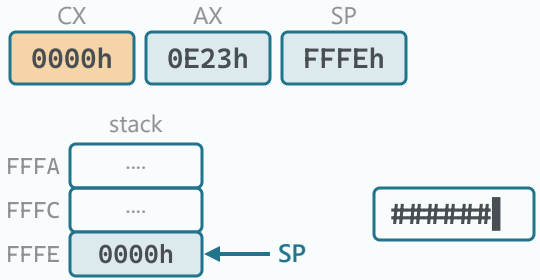
```
1  MOV  CX, 2        ; main loop: 2 rows
2  row: CALL stars
▶  LOOP row
4  RET
5  stars PROC
6  PUSH CX           ; caller's counter
7  MOV  CX, 3        ; inner counter
8  MOV  AX, 0E23h    ; AH = 0Eh, AL = '#'
9  inner:  INT 10h
10 LOOP inner
11 POP  CX           ; counter back
12 RET
13 stars ENDP
```

VARIABLES

CX = 0 | SP = FFFEh

OUTPUT

```
######
```



LOOP: CX = 0 → the loop ends. 6 characters on screen. Without PUSH/POP, CX would be 0 on return, and LOOP would make it FFFFh and loop almost forever.

Step 10 / 10

Caller-Saved or Callee-Saved?

Criterion	Callee saves (PUSH/POP in the procedure)	Caller saves (around the CALL)
Where is the code?	once, in the procedure	at every call site
Needless saving	saved even if the caller doesn't use it	only the registers really needed
Risk of forgetting	low: written in one place	high: needed again at every call
Best for	library procedures	when the old value of an output register is needed

```
again: PUSH CX          ; caller: counter
       CALL read_num    ; result in CX
       ADD  total, CX
       POP  CX          ; counter back
       LOOP again
```

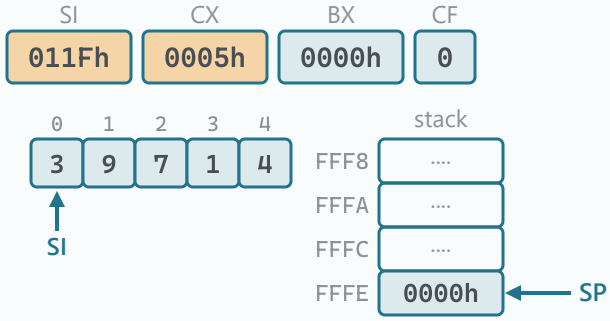
The rule in practice: a procedure preserves the registers it uses but not its output registers. If the old value of an output register is needed, the caller saves it.

Code Trace: Single Exit Point

```
1  ORG  100h
▶ LEA  SI, arr
▶ MOV  CX, 5
▶ MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
11 scan: CMP [SI], AL
12 JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

SI = 011Fh | CX = 0005h | BX = 0000h | CF = 0 | SP = FFFEh



The find procedure searches the CX-element array at SI for AL (7); it returns the result in CF and BX.

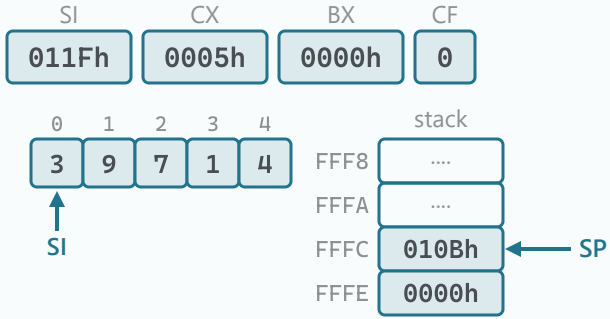
Step 1 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
▶  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
11 scan: CMP [SI], AL
12 JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

`SI = 011Fh` `CX = 0005h` `BX = 0000h` `CF = 0` `SP = FFFCh`



`CALL find` : return address 010Bh pushed.

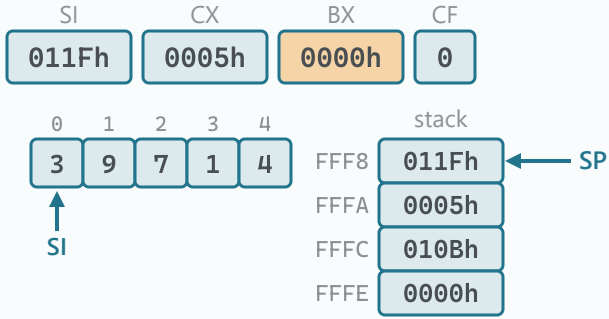
Step 2 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
▶  PUSH CX
▶  PUSH SI
▶  XOR  BX, BX      ; Out: BX = index
11 scan: CMP [SI], AL
12 JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

SI = 011Fh | CX = 0005h | BX = 0000h | CF = 0 | SP = FFF8h



The procedure will change CX and SI, so it saves both; SP = FFF8h. BX = 0 (index).

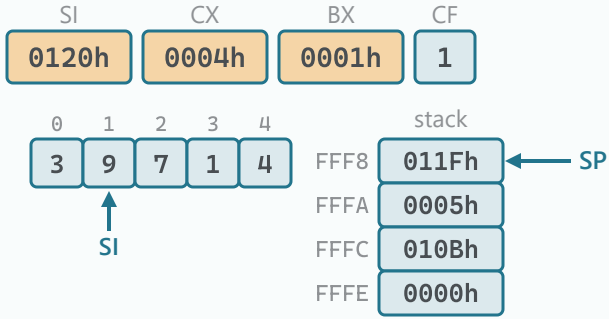
Step 3 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
▶ scan: CMP [SI], AL
▶ JE   found
▶ INC  SI
▶ INC  BX
▶ LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

`SI = 0120h` `CX = 0004h` `BX = 0001h` `CF = 1` `SP = FFF8h`



arr[0] = 3 ≠ 7: SI and BX advance, LOOP brings CX down to 4.

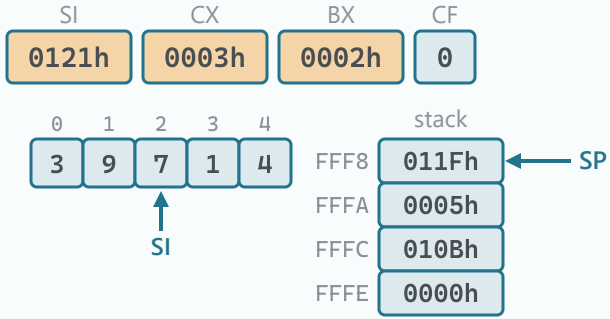
Step 4 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
▶ scan: CMP [SI], AL
▶ JE   found
▶ INC  SI
▶ INC  BX
▶ LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

SI = 0121h | CX = 0003h | BX = 0002h | CF = 0 | SP = FFF8h



arr[1] = 9 ≠ 7: BX = 2, CX = 3.

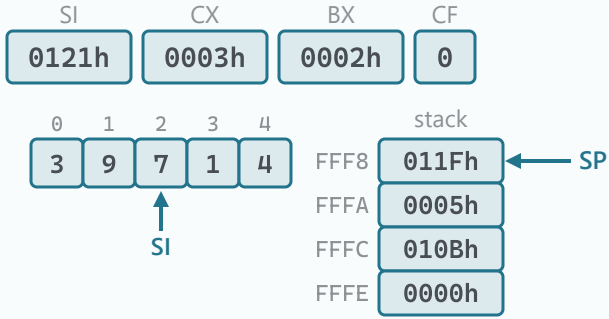
Step 5 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
▶ scan: CMP [SI], AL
▶ JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

`SI = 0121h` `CX = 0003h` `BX = 0002h` `CF = 0` `SP = FFF8h`



arr[2] = 7: ZF = 1, `JE found` jumps. The loop ends early, but two saved values are still on the stack.

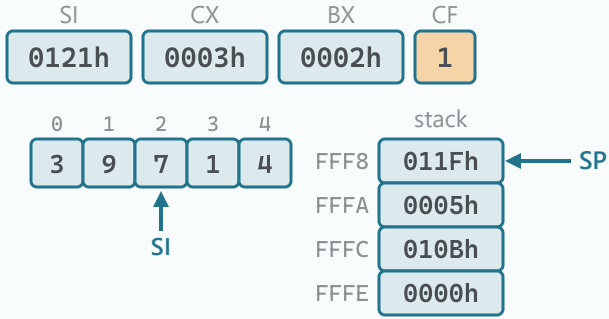
Step 6 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
11 scan: CMP [SI], AL
12 JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
▶  found: STC
19 done: POP SI     ; single exit point
20 POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

`SI = 0121h` `CX = 0003h` `BX = 0002h` `CF = 1` `SP = FFF8h`



`STC` : CF = 1 → “found”. If not found, CLC and `JMP done` would arrive at the same place.

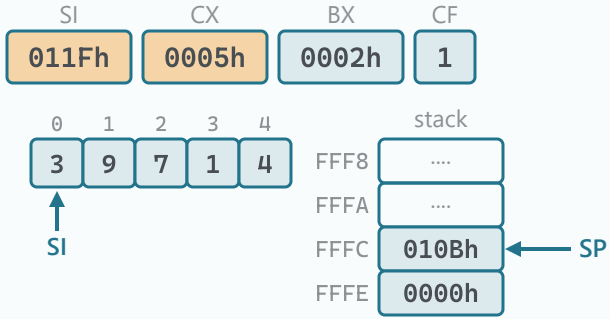
Step 7 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
11 scan: CMP [SI], AL
12 JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
▶  done: POP SI     ; single exit point
▶  POP  CX
21 RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

SI = 011Fh | CX = 0005h | BX = 0002h | CF = 1 | SP = FFFCh



Both paths meet at the `done` label: the POPs always run, and SI and CX get their old values back.

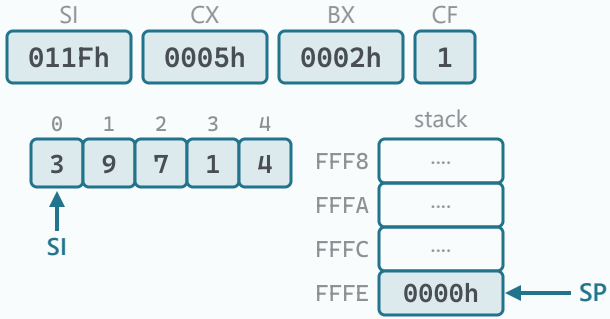
Step 8 / 9

Code Trace: Single Exit Point

```
1  ORG  100h
2  LEA  SI, arr
3  MOV  CX, 5
4  MOV  AL, 7
5  CALL find        ; CF = 1: found
6  RET
7  find PROC        ; In: SI, CX, AL
8  PUSH CX
9  PUSH SI
10 XOR  BX, BX      ; Out: BX = index
11 scan: CMP [SI], AL
12 JE   found
13 INC  SI
14 INC  BX
15 LOOP scan
16 CLC              ; not found
17 JMP  done
18 found: STC
19 done: POP SI     ; single exit point
20 POP  CX
▶  RET
22 find ENDP
23 arr DB 3, 9, 7, 1, 4
```

VARIABLES

`SI = 011Fh` `CX = 0005h` `BX = 0002h` `CF = 1` `SP = FFFEh`

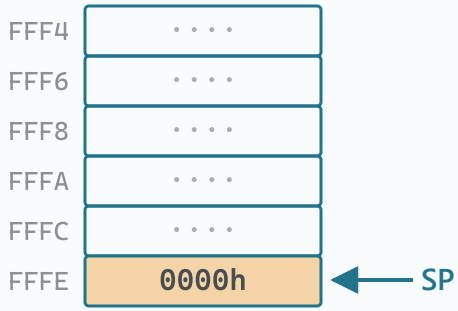


`RET` returns to the right address (010Bh): CF = 1, BX = 2. Had JE been replaced by a direct RET, the SI value on top (011Fh) would have been taken as the return address.

Step 9 / 9

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

The caller puts the parameters on the stack; the procedure reads them through BP.

Step 1 / 14

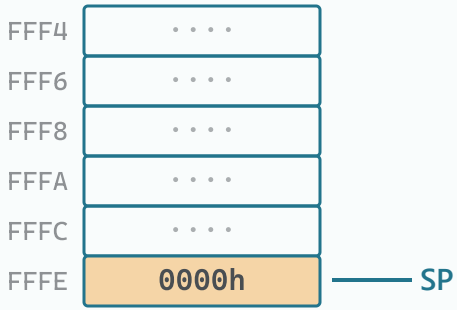
VARIABLES

AX = 0000h BP = 0000h SP = FFFEh IP = 0100h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
▶  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

AX = 7.

Step 2 / 14

VARIABLES

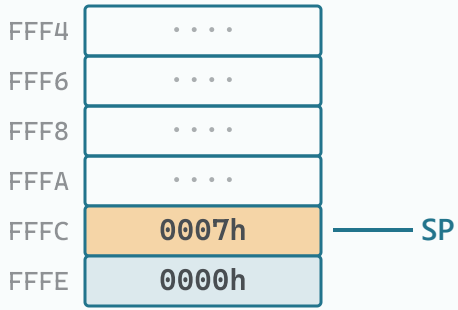
AX = **0007h** BP = **0000h** SP = **FFFEh**

IP = **0103h**

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
▶  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

1st parameter on the stack: [FFFCh] = 0007h.

Step 3 / 14

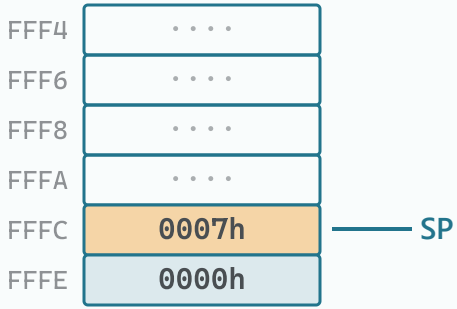
VARIABLES

AX = 0007h BP = 0000h SP = FFFCh IP = 0104h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
▶  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

AX = 3.

Step 4 / 14

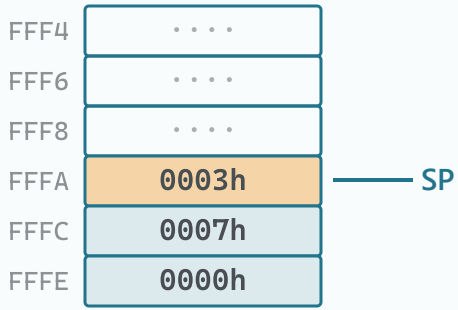
VARIABLES

AX = 0003h BP = 0000h SP = FFFCh IP = 0107h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
▶  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

2nd parameter: [FFFAh] = 0003h.

Step 5 / 14

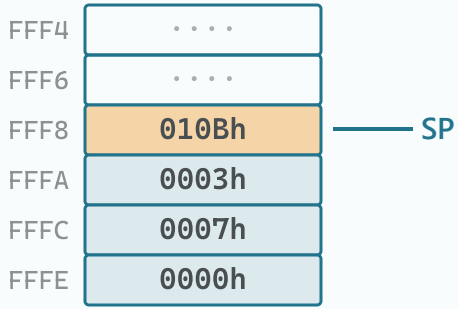
VARIABLES

AX = 0003h BP = 0000h SP = FFFAh IP = 0108h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

CALL: return address **010Bh** → [FFF8h]. IP = 010Fh.

Step 6 / 14

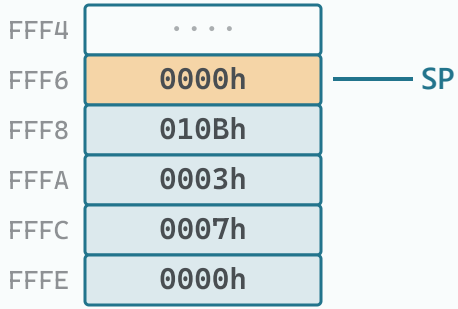
VARIABLES

AX = **0003h** BP = **0000h** SP = **FFF8h** IP = **010Fh**

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
▶  PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

The old BP is saved: [FFF6h] = 0000h.

Step 7 / 14

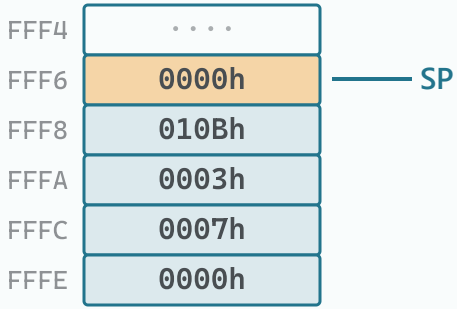
VARIABLES

AX = 0003h BP = 0000h SP = FFF6h IP = 0110h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
▶  MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

`MOV BP, SP` : BP = **FFF6h** becomes a fixed frame pointer.

Step 8 / 14

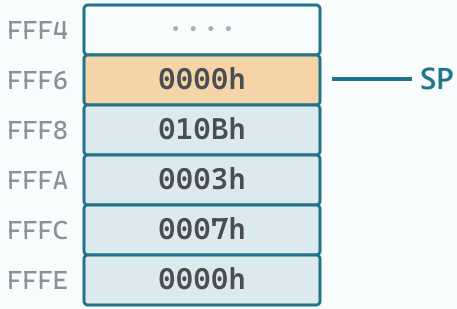
VARIABLES

`AX = 0003h` `BP = FFF6h` `SP = FFF6h` `IP = 0112h`

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
▶ MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

[BP+6] = [FFFCh] = 0007h → AX = 7.

Step 9 / 14

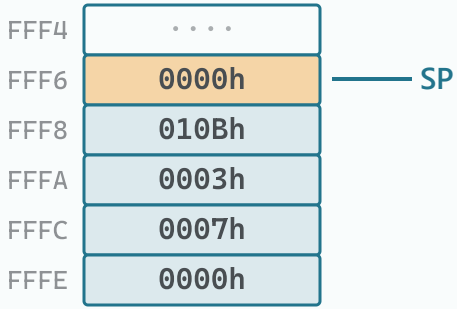
VARIABLES

AX = **0007h** BP = **FFF6h** SP = **FFF6h** IP = **0115h**

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
▶ ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

[BP+4] = [FFFAh] = 0003h → AX = **000Ah**.

Step 10 / 14

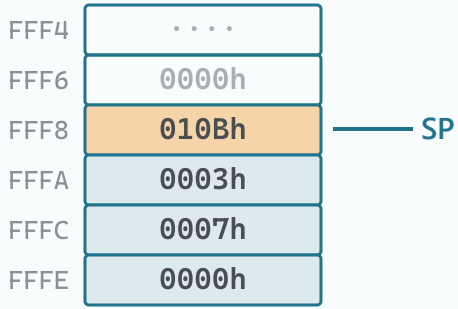
VARIABLES

AX = **000Ah** BP = **FFF6h** SP = **FFF6h** IP = **0118h**

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
▶ POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

BP gets its old value back; SP = FFF8h (return address on top).

Step 11 / 14

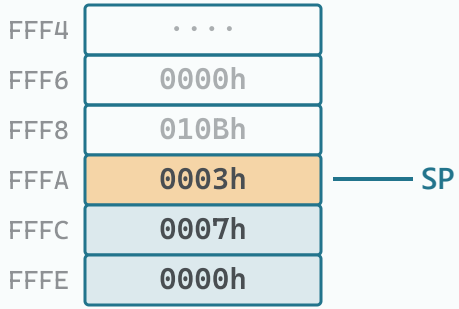
VARIABLES

AX = 000Ah BP = 0000h SP = FFF8h IP = 0119h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
▶  RET
16 SUM2 ENDP
```



SS:offset · low address at top

RET: IP = 010Bh, SP = FFFAh.

Step 12 / 14

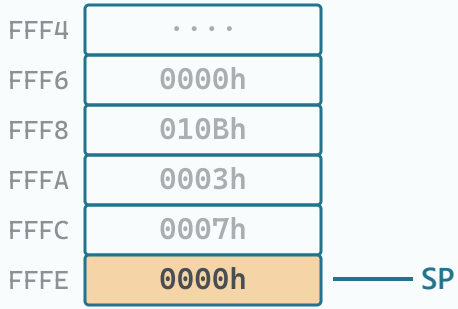
VARIABLES

AX = 000Ah | BP = 0000h | SP = FFFAh | IP = 010Bh

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
▶  ADD  SP, 4       ; drop parameters
8  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

`ADD SP, 4` : both parameters are discarded, SP = FFFEh. The stack is balanced.

Step 13 / 14

VARIABLES

AX = **000Ah** BP = **0000h** SP = **FFFEh** IP = **010Eh**

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Passing Parameters on the Stack (with BP)

```
1  ORG  100h
2  MOV  AX, 7
3  PUSH AX          ; 1st parameter
4  MOV  AX, 3
5  PUSH AX          ; 2nd parameter
6  CALL SUM2
7  ADD  SP, 4       ; drop parameters
▶  RET
9  SUM2 PROC
10 PUSH BP
11 MOV  BP, SP
12 MOV  AX, [BP+6]  ; 1st parameter
13 ADD  AX, [BP+4]  ; 2nd parameter
14 POP  BP
15 RET
16 SUM2 ENDP
```



SS:offset · low address at top

The program ends.

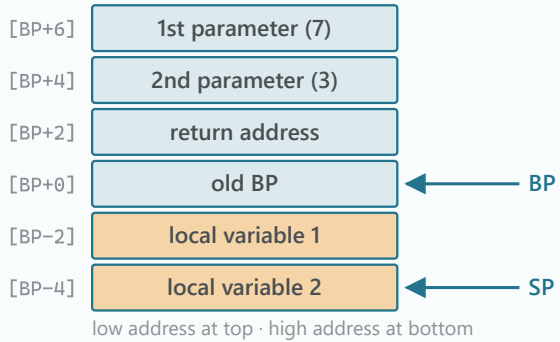
Step 14 / 14

VARIABLES

AX = 000Ah BP = 0000h SP = 0000h IP = 0000h

[BP+0] = old BP, [BP+2] = return address, [BP+4] = the last parameter pushed. Instead of ADD SP, 4, the procedure could also end with RET 4.

Stack Frame and Local Variables

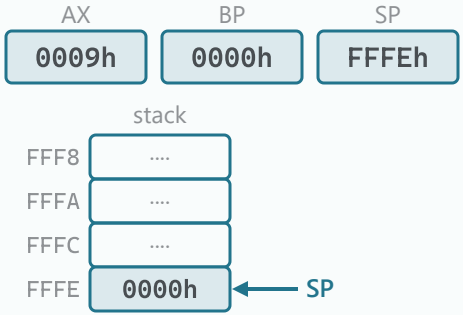



```
PUSH BP
MOV  BP, SP
SUB  SP, 4          ; local space
MOV  [BP-2], AX     ; local 1
...
MOV  SP, BP         ; drop locals
POP  BP
RET  4              ; drop 2 params
```

- `SUB SP, 4` reserves two **local** words on the stack: [BP−2], [BP−4].
- `MOV SP, BP` releases the local space in one instruction.
- Parameters sit at positive BP offsets, locals at negative ones.

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



Parameter x = 9 is in AX. The procedure will keep x in a local variable and compute x·x + x.

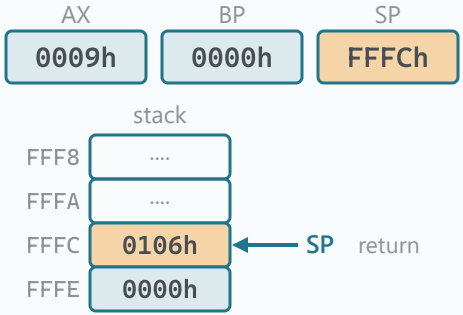
Step 1 / 10

VARIABLES

AX = **9** BP = **0000h** SP = **FFFEh**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
▶ CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2       ; local [BP-2]
8  MOV  [BP-2], AX  ; local = x
9  MUL  AX          ; AX = x·x
10 ADD  AX, [BP-2]  ; + x
11 MOV  SP, BP      ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`CALL` : return address 0106h → [FFFCh].

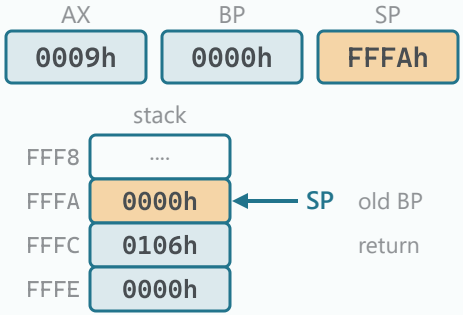
Step 2 / 10

VARIABLES

AX = **9** | BP = **0000h** | SP = **FFFCh**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
▶  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`PUSH BP` : the caller's BP (0000h) is saved.

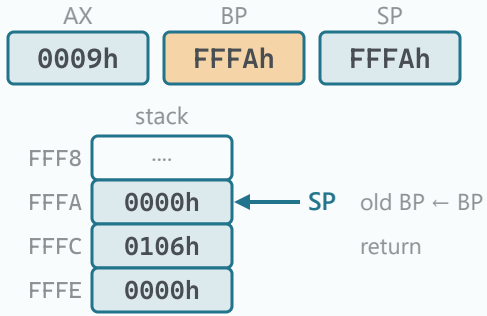
Step 3 / 10

VARIABLES

AX = **9** | BP = **0000h** | SP = **FFFAh**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
▶  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`MOV BP, SP` : BP = FFFAh, the fixed point of the frame.

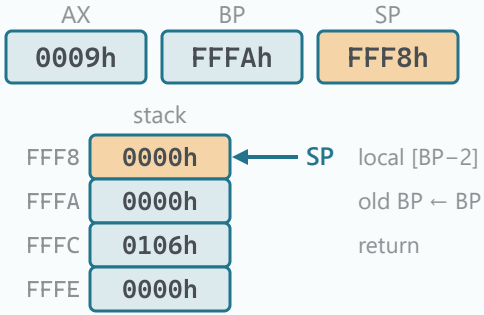
Step 4 / 10

VARIABLES

AX = **9** | BP = **FFFAh** | SP = **FFFAh**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
▶  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`SUB SP, 2` : one word of local space; [BP−2] = FFF8h.

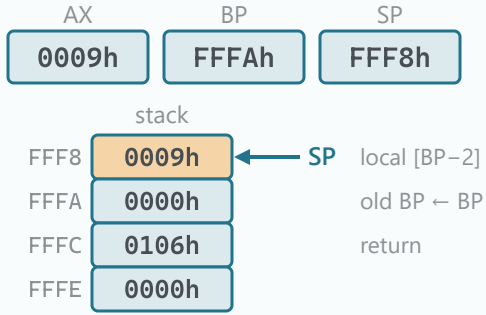
Step 5 / 10

VARIABLES

AX = **9** BP = **FFFAh** SP = **FFF8h**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
▶  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`MOV [BP−2], AX` : local = 9. [BP...] addresses use the SS segment by default.

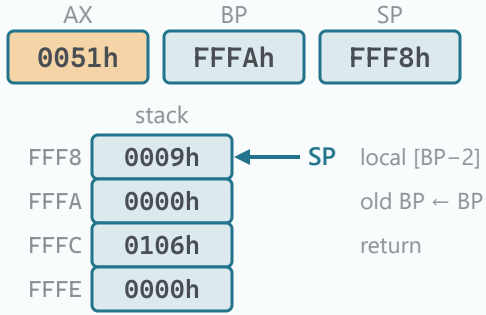
Step 6 / 10

VARIABLES

`AX = 9` `BP = FFFAh` `SP = FFF8h`

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
▶ MUL  AX            ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`MUL AX` : AX = 81 (DX = 0). AX changed, but x is still in the local.

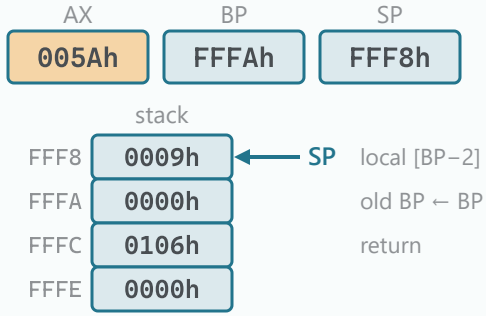
Step 7 / 10

VARIABLES

AX = **81** | BP = **FFFAh** | SP = **FFF8h**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
▶ ADD  AX, [BP-2]    ; + x
11 MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`ADD AX, [BP−2]` : 81 + 9 = 90 = 005Ah.

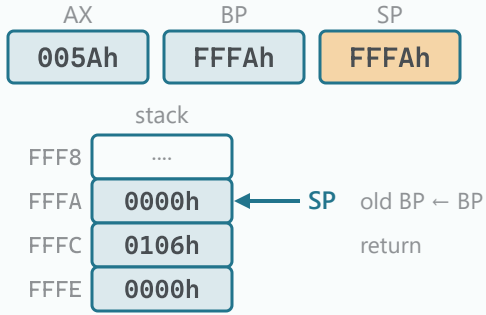
Step 8 / 10

VARIABLES

AX = **90** | BP = **FFFAh** | SP = **FFF8h**

Code Trace: A Local Variable on the Stack

```
1  MOV  AX, 9
2  CALL sq_plus      ; AX = x·x + x
3  RET
4  sq_plus PROC
5  PUSH BP
6  MOV  BP, SP
7  SUB  SP, 2        ; local [BP-2]
8  MOV  [BP-2], AX   ; local = x
9  MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
▶  MOV  SP, BP       ; drop local
12 POP  BP
13 RET
14 sq_plus ENDP
```



`MOV SP, BP` : the local space is released in one instruction, SP = FFFAh.

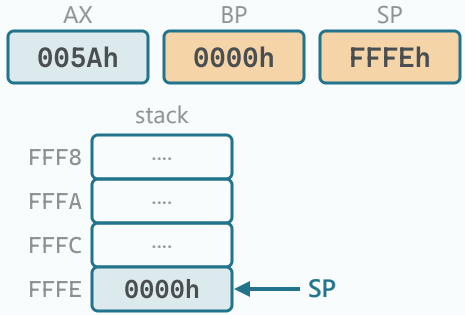
Step 9 / 10

VARIABLES

AX = **90** BP = **FFFAh** SP = **FFFAh**

Code Trace: A Local Variable on the Stack

```
 1 MOV  AX, 9
 2 CALL sq_plus      ; AX = x·x + x
 3 RET
 4 sq_plus PROC
 5 PUSH BP
 6 MOV  BP, SP
 7 SUB  SP, 2        ; local [BP-2]
 8 MOV  [BP-2], AX   ; local = x
 9 MUL  AX           ; AX = x·x
10 ADD  AX, [BP-2]   ; + x
11 MOV  SP, BP       ; drop local
   POP  BP
   RET
14 sq_plus ENDP
```



`POP BP`, `RET`: BP = 0, SP = FFFEh, back in the caller. Result AX = 90.

Step 10 / 10

VARIABLES

AX = **90** BP = **0000h** SP = **FFFEh**

Hello World Message

- Printing a message with a procedure.
- `LEA SI, msg` : loads the **address** of the string msg into SI (parameter = address).
- `CALL print_me` : calls the print_me procedure.
- `RET` : returns to the operating system.
- `print_me` prints a **null-terminated (0)** string character by character.
- `b.[SI]` is emu8086 shorthand for `BYTE PTR [SI]` : `CMP [SI], 0` does not give the operand size, so it is stated explicitly.

```
ORG    100h
LEA    SI, msg     ; address of msg
CALL   print_me
RET                ; to DOS

print_me PROC
  ; ............
print_me ENDP

msg DB 'Hello World!', 0
END
```

Code Trace: `print_me`

```
 1 ORG    100h
▶3 LEA    SI, msg        ; SI = address of msg
 3 CALL   print_me
 4 RET                   ; to DOS
 5 print_me PROC
 6 next_char:
 7 CMP  b.[SI], 0        ; zero?
 8 JE   stop             ; yes: done
 9 MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = **msg+0** SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

SI points to the first character of the string ('H').

Step 1 / 12

Code Trace: `print_me`

```
1  ORG    100h
2  LEA    SI, msg        ; SI = address of msg
▶  CALL   print_me
4  RET                   ; to DOS
5  print_me PROC
6  next_char:
7  CMP  b.[SI], 0        ; zero?
8  JE   stop             ; yes: done
9  MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = **msg+0** SP = **FFFCh**

CALL STACK

[FFFCh] ret → line 4

[FFFEh] 0000h

OUTPUT

`CALL print_me` : the return address (the RET on line 4) is pushed.

Step 2 / 12

Code Trace: `print_me`

```
 1 ORG    100h
 2 LEA    SI, msg        ; SI = address of msg
 3 CALL   print_me
 4 RET                   ; to DOS
 5 print_me PROC
 6 next_char:
▶ CMP  b.[SI], 0         ; zero?
▶ JE   stop              ; yes: done
 9 MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

`SI = msg+0` `ZF = 0` `SP = FFFCh`

CALL STACK

`[FFFCh] ret → line 4`

`[FFFEh] 0000h`

OUTPUT

[SI] = 'H' (48h) ≠ 0 → ZF = 0, JE does not jump.

Step 3 / 12

Code Trace: `print_me`

```
1  ORG    100h
2  LEA    SI, msg        ; SI = address of msg
3  CALL   print_me
4  RET                   ; to DOS
5  print_me PROC
6  next_char:
7  CMP  b.[SI], 0        ; zero?
8  JE   stop             ; yes: done
▶  MOV  AL, [SI]         ; next ASCII char
▶  MOV  AH, 0Eh          ; teletype function
▶  INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

`SI = msg+0` `AL = 48h` `AH = 0Eh` `ZF = 0` `SP = FFFCh`

CALL STACK

`[FFFCh] ret → line 4`

`[FFFEh] 0000h`

OUTPUT

```
H
```

AL = 48h, AH = 0Eh, INT 10h → **H**.

Step 4 / 12

Code Trace: `print_me`

```
1  ORG    100h
2  LEA    SI, msg        ; SI = address of msg
3  CALL   print_me
4  RET                   ; to DOS
5  print_me PROC
6  next_char:
7  CMP  b.[SI], 0        ; zero?
8  JE   stop             ; yes: done
9  MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
▶ ADD  SI, 1             ; next character
▶ JMP  next_char         ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = msg+1 | AL = 48h | AH = 0Eh | ZF = 0 | SP = FFFCh

CALL STACK

[FFFCh] ret → line 4

[FFFEh] 0000h

OUTPUT

```
H
```

SI = msg+1, back to `next_char`.

Step 5 / 12

Code Trace: `print_me`

```
 1 ORG    100h
 2 LEA    SI, msg        ; SI = address of msg
 3 CALL   print_me
 4 RET                   ; to DOS
 5 print_me PROC
 6 next_char:
▶ CMP  b.[SI], 0         ; zero?
▶ JE   stop              ; yes: done
▶ MOV  AL, [SI]          ; next ASCII char
▶ MOV  AH, 0Eh           ; teletype function
▶ INT  10h               ; print AL
▶ ADD  SI, 1             ; next character
▶ JMP  next_char         ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = msg+2 | AL = 65h | AH = 0Eh | ZF = 0 | SP = FFFCh

CALL STACK

[FFFCh] ret → line 4

[FFFEh] 0000h

OUTPUT

```
He
```

Next pass: 'e' is printed. AL = 65h, SI = msg+2.

Step 6 / 12

Code Trace: `print_me`

```
1  ORG    100h
2  LEA    SI, msg        ; SI = address of msg
3  CALL   print_me
4  RET                   ; to DOS
5  print_me PROC
6  next_char:
▶  CMP  b.[SI], 0        ; zero?
▶  JE   stop             ; yes: done
▶  MOV  AL, [SI]         ; next ASCII char
▶  MOV  AH, 0Eh          ; teletype function
▶  INT  10h              ; print AL
▶  ADD  SI, 1            ; next character
▶  JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = **msg+5** | AL = **6Fh** | AH = **0Eh** | ZF = **0** | SP = **FFFCh**

CALL STACK

- **[FFFCh] ret → line 4**
- [FFFEh] 0000h

OUTPUT

```
Hello
```

3 more iterations: "llo" is printed. AL = 6Fh, SI = msg+5.

Step 7 / 12

Code Trace: `print_me`

```
1  ORG    100h
2  LEA    SI, msg        ; SI = address of msg
3  CALL   print_me
4  RET                   ; to DOS
5  print_me PROC
6  next_char:
▶  CMP  b.[SI], 0        ; zero?
▶  JE   stop             ; yes: done
▶  MOV  AL, [SI]         ; next ASCII char
▶  MOV  AH, 0Eh          ; teletype function
▶  INT  10h              ; print AL
▶  ADD  SI, 1            ; next character
▶  JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = **msg+11** | AL = **64h** | AH = **0Eh** | ZF = **0** | SP = **FFFCh**

CALL STACK

[FFFCh] ret → line 4

[FFFEh] 0000h

OUTPUT

```
Hello World
```

6 more iterations: " World" is printed (the space is a character too: 20h). AL = 64h, SI = msg+11.

Code Trace: `print_me`

```
 1 ORG    100h
 2 LEA    SI, msg        ; SI = address of msg
 3 CALL   print_me
 4 RET                   ; to DOS
 5 print_me PROC
 6 next_char:
▶ CMP  b.[SI], 0         ; zero?
▶ JE   stop              ; yes: done
▶ MOV  AL, [SI]          ; next ASCII char
▶ MOV  AH, 0Eh           ; teletype function
▶ INT  10h               ; print AL
▶ ADD  SI, 1             ; next character
▶ JMP  next_char         ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

SI = msg+12 | AL = 21h | AH = 0Eh | ZF = 0 | SP = FFFCh

CALL STACK

[FFFCh] ret → line 4

[FFFEh] 0000h

OUTPUT

```
Hello World!
```

Next pass: '!' is printed. AL = 21h, SI = msg+12.

Step 9 / 12

Code Trace: `print_me`

```
 1 ORG    100h
 2 LEA    SI, msg        ; SI = address of msg
 3 CALL   print_me
 4 RET                   ; to DOS
 5 print_me PROC
 6 next_char:
▶ CMP  b.[SI], 0         ; zero?
▶ JE   stop              ; yes: done
 9 MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

`SI = msg+12` `AL = 21h` `AH = 0Eh` `ZF = 1` `SP = FFFCh`

CALL STACK

`[FFFCh] ret → line 4`

`[FFFEh] 0000h`

OUTPUT

```
Hello World!
```

[SI] = 0 → ZF = 1, `JE stop` jumps.

Step 10 / 12

Code Trace: `print_me`

```
 1 ORG    100h
 2 LEA    SI, msg        ; SI = address of msg
 3 CALL   print_me
 4 RET                   ; to DOS
 5 print_me PROC
 6 next_char:
 7 CMP  b.[SI], 0        ; zero?
 8 JE   stop             ; yes: done
 9 MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
▶ RET                    ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

`SI = msg+12` `AL = 21h` `AH = 0Eh` `ZF = 1` `SP = FFFEh`

CALL STACK

`[FFFEh] 0000h`

OUTPUT

```
Hello World!
```

The procedure's `RET` pops the return address → back to line 4.

Step 11 / 12

Code Trace: `print_me`

```
1  ORG    100h
2  LEA    SI, msg        ; SI = address of msg
3  CALL   print_me
▶  RET                   ; to DOS
5  print_me PROC
6  next_char:
7  CMP  b.[SI], 0        ; zero?
8  JE   stop             ; yes: done
9  MOV  AL, [SI]         ; next ASCII char
10 MOV  AH, 0Eh          ; teletype function
11 INT  10h              ; print AL
12 ADD  SI, 1            ; next character
13 JMP  next_char        ; loop back
14 stop:
15 RET                   ; to caller
16 print_me ENDP
17 msg DB 'Hello World!', 0
18 END
```

VARIABLES

`SI = msg+12` `AL = 21h` `AH = 0Eh` `ZF = 1` `SP = 0000h`

CALL STACK

empty

OUTPUT

```
Hello World!
```

The main program's `RET` returns to DOS. On screen: **Hello World!**

Step 12 / 12

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
▶  CALL AddNumbers     ; call it
4  RET                 ; to DOS
5  AddNumbers PROC
6  MOV AX, number1     ; AX = first number
7  ADD AX, number2     ; AX += second number
8  MOV result, AX      ; store the result
9  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

SP = **FFFEh** result = **?**

CALL STACK

[FFFEh] 0000h

The parameters are in memory variables: number1 = 5, number2 = 3.

Step 1 / 7

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
▶  CALL AddNumbers     ; call it
4  RET                 ; to DOS
5  AddNumbers PROC
6  MOV AX, number1     ; AX = first number
7  ADD AX, number2     ; AX += second number
8  MOV result, AX      ; store the result
9  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

SP = **FFFCh** | result = **?**

CALL STACK

[FFFCh] 0103h

[FFFEh] 0000h

`CALL AddNumbers` (0100h, 3 bytes): return address **0103h** is pushed.

Step 2 / 7

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
3  CALL AddNumbers     ; call it
4  RET                 ; to DOS
5  AddNumbers PROC
▶  MOV AX, number1     ; AX = first number
7  ADD AX, number2     ; AX += second number
8  MOV result, AX      ; store the result
9  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

SP = **FFFCh** | AX = **0005h** | result = **?**

CALL STACK

[FFFCh] 0103h

[FFFEh] 0000h

AX = number1 = 0005h.

Step 3 / 7

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
3  CALL AddNumbers     ; call it
4  RET                 ; to DOS
5  AddNumbers PROC
6  MOV AX, number1     ; AX = first number
▶  ADD AX, number2     ; AX += second number
8  MOV result, AX      ; store the result
9  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

SP = FFFCh | AX = 0008h | result = ? | ZF = 0 | CF = 0

CALL STACK

[FFFCh] 0103h

[FFFEh] 0000h

AX = 5 + 3 = 0008h; the result is nonzero, no carry.

Step 4 / 7

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
3  CALL AddNumbers     ; call it
4  RET                 ; to DOS
5  AddNumbers PROC
6  MOV AX, number1     ; AX = first number
7  ADD AX, number2     ; AX += second number
▶  MOV result, AX      ; store the result
9  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

SP = FFFCh | AX = 0008h | result = 0008h | ZF = 0 | CF = 0

CALL STACK

[FFFCh] 0103h

[FFFEh] 0000h

result = 0008h (08 00 in memory).

Step 5 / 7

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
3  CALL AddNumbers     ; call it
4  RET                 ; to DOS
5  AddNumbers PROC
6  MOV AX, number1     ; AX = first number
7  ADD AX, number2     ; AX += second number
8  MOV result, AX      ; store the result
▶  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

SP = FFFEh | AX = 0008h | result = 0008h | ZF = 0 | CF = 0

CALL STACK

[FFFEh] 0000h

`RET` → IP = 0103h.

Step 6 / 7

Code Trace: Adding Two Variables

```
1  ORG 100h
2  start:
3  CALL AddNumbers     ; call it
▶  RET                 ; to DOS
5  AddNumbers PROC
6  MOV AX, number1     ; AX = first number
7  ADD AX, number2     ; AX += second number
8  MOV result, AX      ; store the result
9  RET                 ; end of procedure
10 AddNumbers ENDP
11 number1 DW 5        ; first number 5
12 number2 DW 3        ; second number 3
13 result  DW ?        ; holds the result
```

VARIABLES

`SP = 0000h` `AX = 0008h` `result = 0008h` `ZF = 0` `CF = 0`

CALL STACK

empty

`RET` → DOS. `DW ?` means the initial value is undefined (emu8086 uses 0).

Step 7 / 7

Code Trace: Address and Count as Parameters

```
1  ORG  100h
▶ 2  LEA  SI, vector     ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+0** | SP = **FFFEh** | m = **00h**

CALL STACK

[FFFEh] 0000h

The array sum as a reusable procedure: SI = address of the array.

Step 1 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
▶  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+0** CX = **0005h**

SP = **FFFEh** m = **00h**

CALL STACK

[FFFEh] 0000h

CX = 5 elements.

Step 2 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
▶  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+0** | CX = **0005h** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

`CALL sum`: return address pushed.

Step 3 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
▶  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+0** | CX = **0005h** | AL = **00h** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

AL = 0.

Step 4 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
▶  ADD  AL, [SI]
▶  INC  SI
▶  LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+1** | CX = **0004h** | AL = **05h** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

+5 → AL = 5; SI advances, CX = 4.

Step 5 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
▶  ADD  AL, [SI]
▶  INC  SI
▶  LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+2** | CX = **0003h** | AL = **09h** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

+4 → AL = 9; SI advances, CX = 3.

Step 6 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
▶  ADD  AL, [SI]
▶  INC  SI
▶  LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+3** | CX = **0002h** | AL = **0Eh** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

+5 → AL = 14; SI advances, CX = 2.

Step 7 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
▶  ADD  AL, [SI]
▶  INC  SI
▶  LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+4** | CX = **0001h** | AL = **10h** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

+2 → AL = 16; SI advances, CX = 1.

Step 8 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
▶  ADD  AL, [SI]
▶  INC  SI
▶  LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+5** | CX = **0000h** | AL = **11h** | SP = **FFFCh** | m = **00h**

CALL STACK

[FFFCh] ret → line 5

[FFFEh] 0000h

+1 → AL = 17; SI advances, CX = 0 → the loop ends.

Step 9 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
▶  RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+5** | CX = **0000h** | AL = **11h** | SP = **FFFEh** | m = **00h**

CALL STACK

`[FFFEh] 0000h`

`RET` : back to the caller, the result in AL.

Step 10 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
▶  MOV  m, AL           ; result is in AL
6  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+5** | CX = **0000h** | AL = **11h** | SP = **FFFEh** | m = **11h**

CALL STACK

[FFFEh] 0000h

m = 11h = 17.

Step 11 / 12

Code Trace: Address and Count as Parameters

```
1  ORG  100h
2  LEA  SI, vector      ; param 1: address
3  MOV  CX, 5           ; param 2: count
4  CALL sum
5  MOV  m, AL           ; result is in AL
▶  RET
7  sum PROC             ; In: SI, CX
8  MOV  AL, 0           ; Out: AL = sum
9  again:
10 ADD  AL, [SI]
11 INC  SI
12 LOOP again
13 RET
14 sum ENDP
15 vector DB 5, 4, 5, 2, 1
16 m      DB 0
```

VARIABLES

SI = **vector+5** | CX = **0000h** | AL = **11h** | SP = **0000h** | m = **11h**

CALL STACK

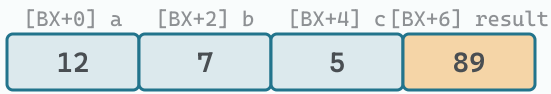
empty

The program ends. The same procedure can be called for another array just by changing SI and CX.

Step 12 / 12

Parameter Block

```
LEA  BX, block
CALL calc          ; result = a × b + c
RET
calc PROC          ; In: BX = block address
PUSH AX
PUSH DX            ; MUL writes DX too
MOV  AX, [BX]      ; a
MUL  WORD PTR [BX+2] ; × b
ADD  AX, [BX+4]    ; + c
MOV  [BX+6], AX    ; result
POP  DX
POP  AX
RET
calc ENDP
block DW 12, 7, 5, ?
```



BX → block

- A single register (BX) carries any number of parameters: the block is read at fixed offsets ([BX+2], [BX+4]).
- The result goes into the block too: 12 × 7 + 5 = 89 (0059h).
- The same procedure is called with another block just by changing BX.
- Document the block layout (what sits at which offset) in the procedure's header comment.

Testing a Procedure

Try a procedure not only with “normal” input but also with edge cases. Results for the `find` procedure (array: 3, 9, 7, 1, 4):

Case	AL	CX	CF	BX	Verdict
first element	3	5	1	0	correct
last element	4	5	1	4	correct
not in the array	8	5	0	5	correct; BX is meaningless, look only at CF
one-element search	9	1	0	1	correct: 9 is the 2nd element, outside the range searched
empty array	7	0	1	2	**wrong**: with CX = 0, LOOP runs 65,536 times

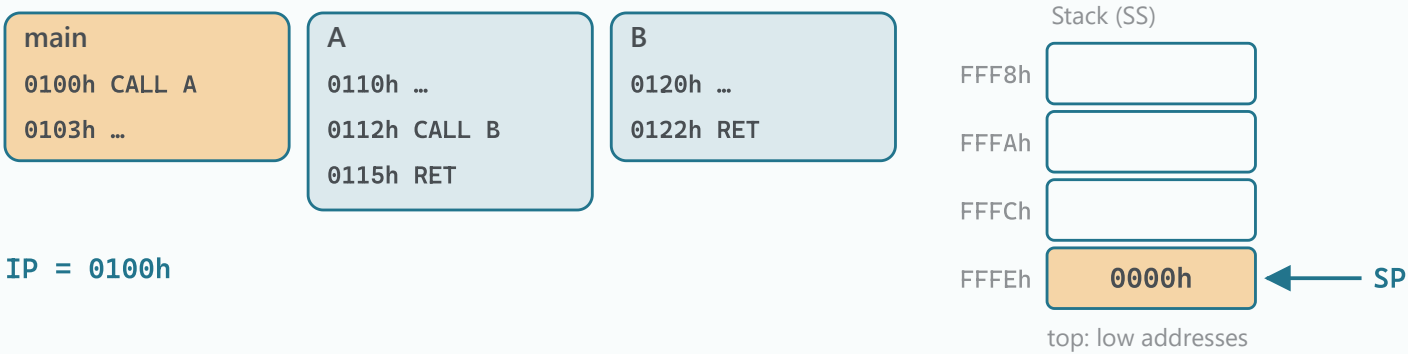
Cause of the bug

LOOP decrements CX first and then tests for zero: 0 − 1 = FFFFh. The search runs 65,536 iterations and “finds” the 7 in memory (BX = 2); with a truly empty array it would read past the end.

Fix

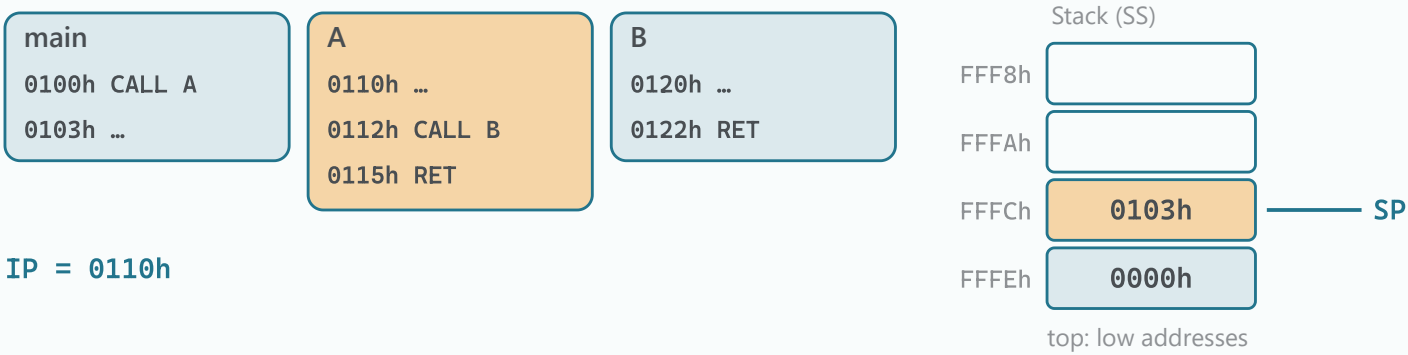
Handle the empty array with `JCXZ` before the loop: if CX = 0, do CLC and go straight to the `done` label. Also check in every test that CX and SI are preserved.

Nested Calls



main is about to call A. SP = FFFEh.

Nested Calls



`CALL A` : 0103h pushed, SP = FFFCh.

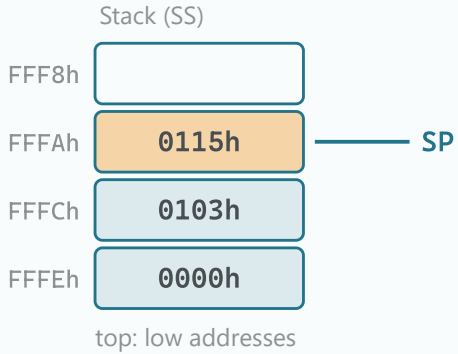
Nested Calls

```
main
0100h CALL A
0103h …
```

```
A
0110h …
0112h CALL B
0115h RET
```

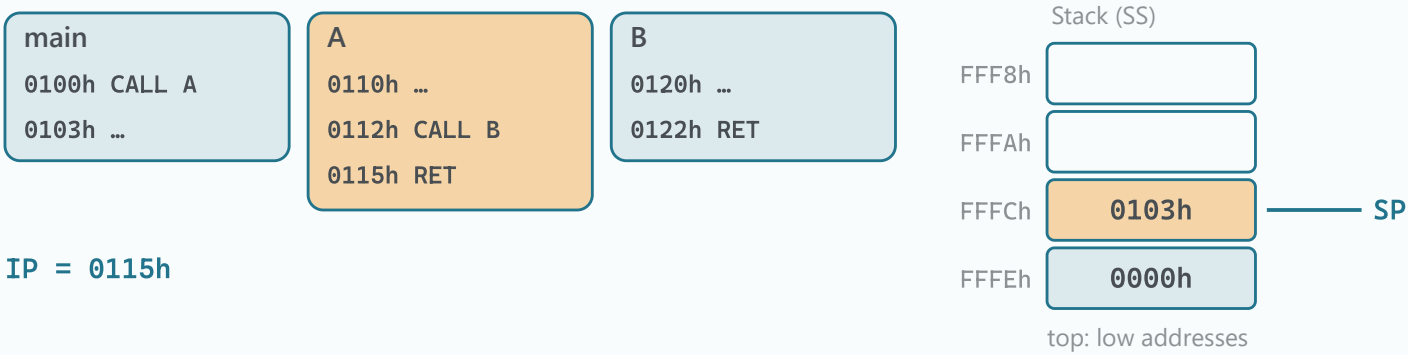
```
B
0120h …
0122h RET
```

IP = 0120h



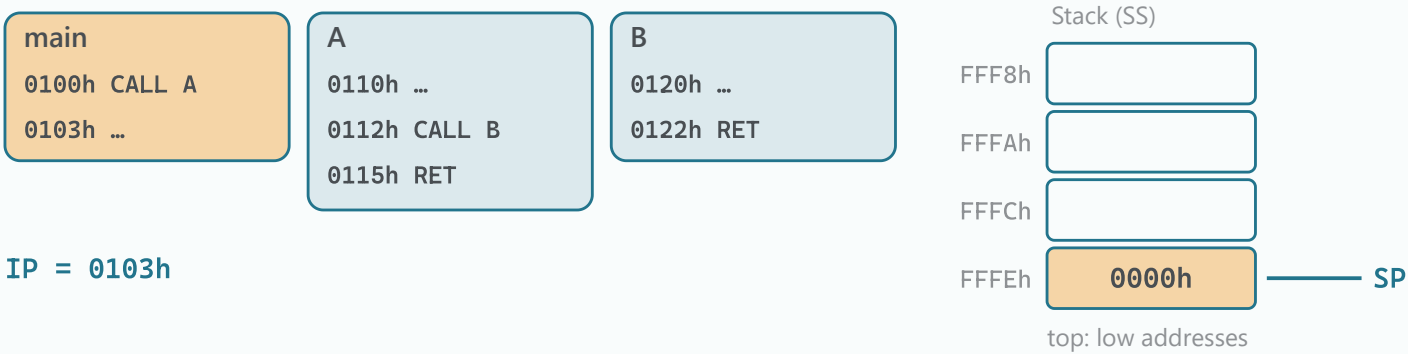
`CALL B` from inside A: 0115h pushed, SP = FFFAh. Two return addresses are on the stack.

Nested Calls



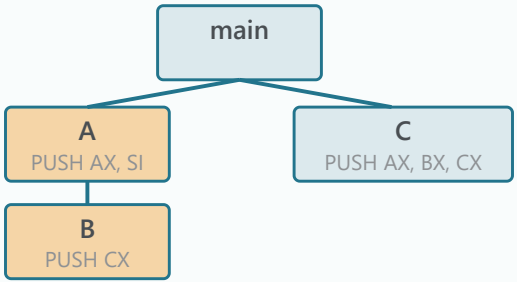
B's `RET` pops 0115h from the top: back to A (LIFO).

Nested Calls



A's `RET` pops 0103h: back to main. Last called, first to return.

How Much Stack?



- A program's stack requirement is the sum of the bytes accumulated along the **deepest path** in its call tree.
- Each near CALL adds a 2-byte return address, each PUSH 2 bytes.

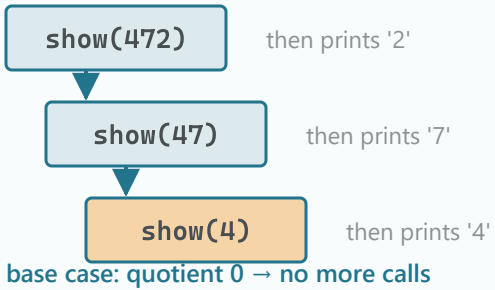
Path	Calculation	Bytes
main → A → B	2 + 4 + 2 + 2	**10**
main → C	2 + 6	8

- In a program that starts with SP = FFFEh, SP goes no lower than FFF4h.
- A and C are never on the stack at the same time: by the time C is called, A's bytes have already been popped.

Interrupts use the stack too (6 bytes for flags, CS and IP): an interrupt may arrive at the stack's deepest moment, so leave a margin above the computed value.

Recursion

- **Recursion**: a procedure calling itself (directly or through another procedure).
- Each call pushes a new return address and its own saved values: the values of each level stay separate.
- A **base case** (stopping condition) is essential; without it every call eats 2 more bytes and the stack overflows.
- The problem shrinks at each step into a smaller copy of itself: printing 472 = printing 47 + printing '2'.
- Typical examples: printing numbers, factorial, walking a tree or a directory.



Code Trace: Printing a Number Recursively

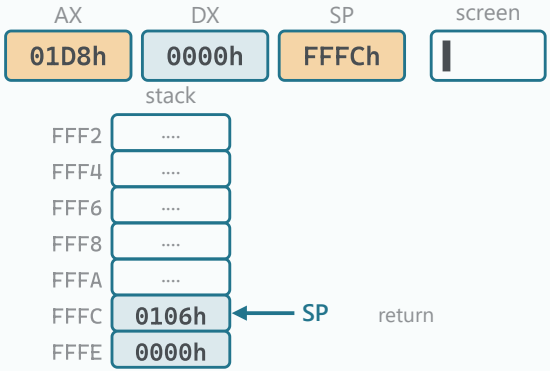
```
1  ORG  100h
▶ MOV  AX, 472
▶ CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
6  XOR  DX, DX
7  MOV  BX, 10
8  DIV  BX          ; AX = quot., DX = digit
9  PUSH DX
10 OR   AX, AX      ; more digits?
11 JZ   done
12 CALL show        ; upper digits first
13 done: POP AX
14 ADD  AL, '0'
15 MOV  AH, 0Eh
16 INT  10h
17 RET
18 show ENDP
```

VARIABLES

AX = 01D8h DX = 0000h SP = FFFCh

OUTPUT





AX = 472 (01D8h); `CALL show` : return address 0106h pushed.

Step 1 / 9

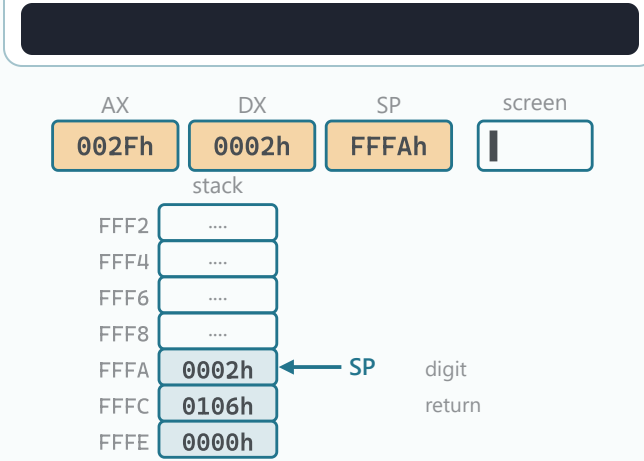
Code Trace: Printing a Number Recursively

```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
▶  XOR  DX, DX
▶  MOV  BX, 10
▶  DIV  BX          ; AX = quot., DX = digit
▶  PUSH DX
10 OR   AX, AX      ; more digits?
11 JZ   done
12 CALL show        ; upper digits first
13 done: POP AX
14 ADD  AL, '0'
15 MOV  AH, 0Eh
16 INT  10h
17 RET
18 show ENDP
```

VARIABLES

AX = 002Fh DX = 0002h SP = FFFAh

OUTPUT



472 / 10: AX = 47 (2Fh), DX = 2. The last digit (2) is pushed.

Step 2 / 9

Code Trace: Printing a Number Recursively

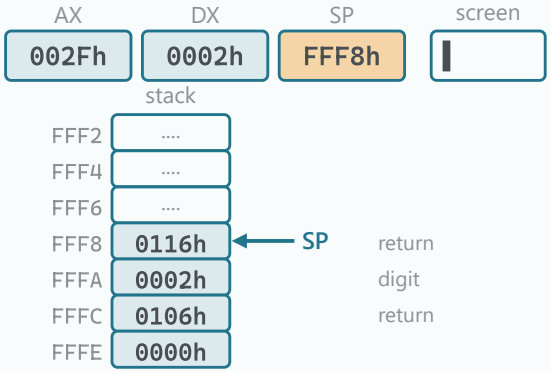
```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
6  XOR  DX, DX
7  MOV  BX, 10
8  DIV  BX          ; AX = quot., DX = digit
9  PUSH DX
▶  OR   AX, AX      ; more digits?
▶  JZ   done
▶  CALL show        ; upper digits first
13 done: POP AX
14 ADD  AL, '0'
15 MOV  AH, 0Eh
16 INT  10h
17 RET
18 show ENDP
```

VARIABLES

AX = 002Fh DX = 0002h SP = FFF8h

OUTPUT





AX ≠ 0: show calls itself; return address 0116h (the `done` label) pushed.

Step 3 / 9

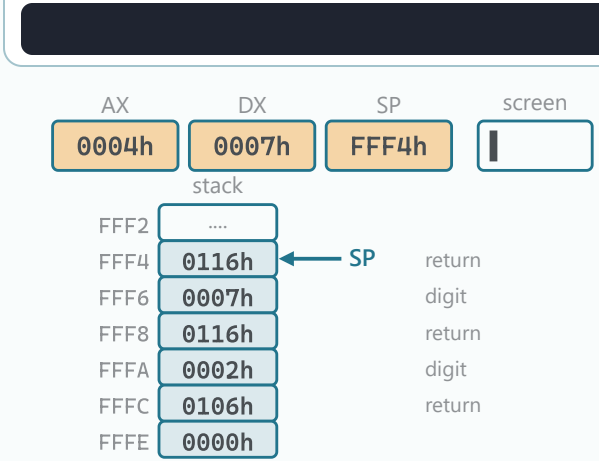
Code Trace: Printing a Number Recursively

```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
▶  XOR  DX, DX
▶  MOV  BX, 10
▶  DIV  BX          ; AX = quot., DX = digit
▶  PUSH DX
▶  OR   AX, AX      ; more digits?
▶  JZ   done
▶  CALL show        ; upper digits first
13 done: POP AX
14 ADD  AL, '0'
15 MOV  AH, 0Eh
16 INT  10h
17 RET
18 show ENDP
```

VARIABLES

`AX = 0004h` `DX = 0007h` `SP = FFF4h`

OUTPUT



Second level: 47 / 10 → AX = 4, digit 7 pushed; CALL again.

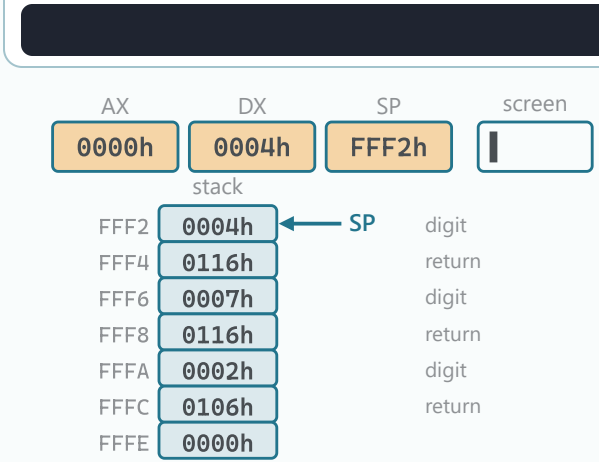
Code Trace: Printing a Number Recursively

```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
▶  XOR  DX, DX
▶  MOV  BX, 10
▶  DIV  BX          ; AX = quot., DX = digit
▶  PUSH DX
▶  OR   AX, AX      ; more digits?
▶  JZ   done
12 CALL show        ; upper digits first
13 done: POP AX
14 ADD  AL, '0'
15 MOV  AH, 0Eh
16 INT  10h
17 RET
18 show ENDP
```

VARIABLES

`AX = 0000h` `DX = 0004h` `SP = FFF2h`

OUTPUT



Third level: 4 / 10 → AX = 0, digit 4 pushed. AX = 0: the base case, `JZ done`. SP at its deepest: FFF2h.

Code Trace: Printing a Number Recursively

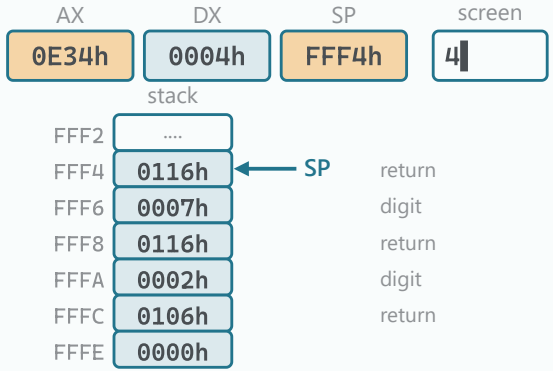
```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
6  XOR  DX, DX
7  MOV  BX, 10
8  DIV  BX          ; AX = quot., DX = digit
9  PUSH DX
10 OR   AX, AX      ; more digits?
11 JZ   done
12 CALL show        ; upper digits first
▶  done: POP AX
▶  ADD  AL, '0'
▶  MOV  AH, 0Eh
▶  INT  10h
17 RET
18 show ENDP
```

VARIABLES

AX = 0E34h DX = 0004h SP = FFF4h

OUTPUT

```
4
```



`POP AX` = 4 → '4' is printed. The digit pushed last comes out first.

Code Trace: Printing a Number Recursively

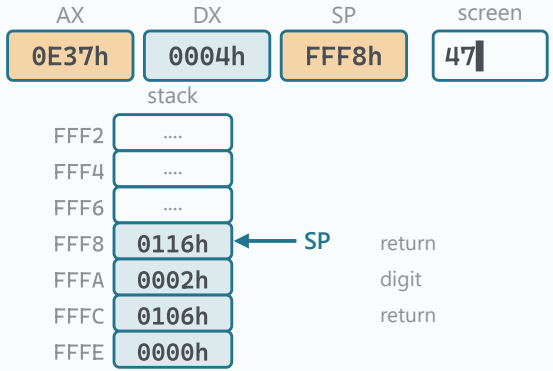
```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
6  XOR  DX, DX
7  MOV  BX, 10
8  DIV  BX          ; AX = quot., DX = digit
9  PUSH DX
10 OR   AX, AX      ; more digits?
11 JZ   done
12 CALL show        ; upper digits first
▶ done: POP AX
▶ ADD  AL, '0'
▶ MOV  AH, 0Eh
▶ INT  10h
▶ RET
18 show ENDP
```

VARIABLES

AX = 0E37h DX = 0004h SP = FFF8h

OUTPUT

```
47
```



RET goes to 0116h, the `done` line of the second level: POP AX = 7 → '7'.

Step 7 / 9

Code Trace: Printing a Number Recursively

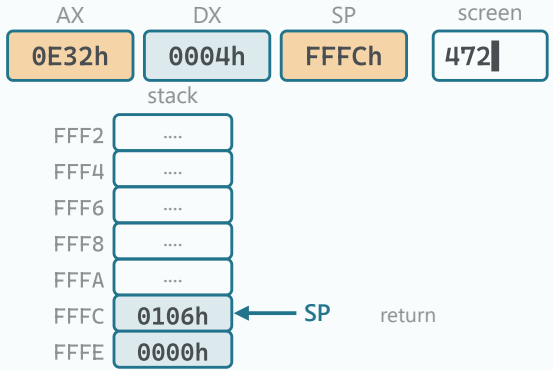
```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
6  XOR  DX, DX
7  MOV  BX, 10
8  DIV  BX          ; AX = quot., DX = digit
9  PUSH DX
10 OR   AX, AX      ; more digits?
11 JZ   done
12 CALL show        ; upper digits first
▶ done: POP AX
▶ ADD  AL, '0'
▶ MOV  AH, 0Eh
▶ INT  10h
▶ RET
18 show ENDP
```

VARIABLES

AX = 0E32h | DX = 0004h | SP = FFFCh

OUTPUT

```
472
```



First level: POP AX = 2 → '2'. The screen shows 472.

Code Trace: Printing a Number Recursively

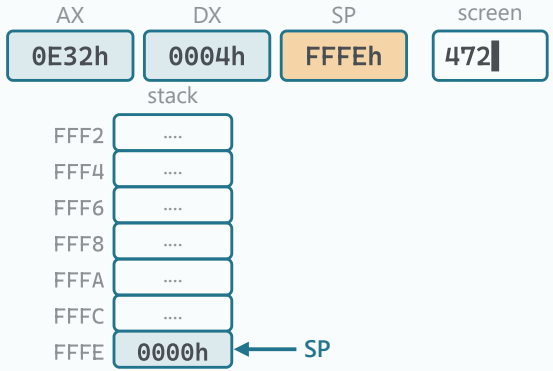
```
1  ORG  100h
2  MOV  AX, 472
3  CALL show
4  RET
5  show PROC        ; In: AX (unsigned)
6  XOR  DX, DX
7  MOV  BX, 10
8  DIV  BX          ; AX = quot., DX = digit
9  PUSH DX
10 OR   AX, AX      ; more digits?
11 JZ   done
12 CALL show        ; upper digits first
13 done: POP AX
14 ADD  AL, '0'
15 MOV  AH, 0Eh
16 INT  10h
▶ RET
18 show ENDP
```

VARIABLES

AX = 0E32h DX = 0004h SP = FFFEh

OUTPUT

```
472
```



The last RET returns to 0106h; SP = FFFEh. Each level kept its digit on the stack, so no extra variable was needed.

Recursion or Loop?

Criterion	Recursion (show)	Loop (divide + LOOP)
Code	short; close to the problem's definition	needs a counter and a separate print loop
Stack	return address + digit per level: 4 bytes	2 bytes per digit
Time	CALL + RET at every level (19 + 8 clocks)	no call overhead
Risk	unbounded depth overflows the stack	bounded stack use

Depth

For show, the number of levels = the number of digits. 65,535 → 5 levels: 5 × 2 (return) + 5 × 2 (digit) = 20 bytes; SP goes no lower than FFEAh.

Where the base case goes

The test must come before the CALL. For AX = 0 one digit (0) is still printed: the number 0 comes out as '0' with no extra code.

Code Trace: Unbalanced PUSH / POP

```
1 ORG 100h
▶ MOV  AX, 1234h      ; 0100h: B8 34 12
3 CALL bad            ; 0103h: ret = 0106h
4 RET                 ; 0106h
5 bad PROC
6 PUSH AX             ; AX saved...
7 ; ... work ...
8 RET                 ; forgot POP AX!
9 bad ENDP
```

VARIABLES

AX = **1234h** | IP = **0103h** | SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

AX = 1234h.

Step 1 / 4

Code Trace: Unbalanced PUSH / POP

```
1 ORG 100h
2 MOV  AX, 1234h      ; 0100h: B8 34 12
▶ CALL bad            ; 0103h: ret = 0106h
4 RET                 ; 0106h
5 bad PROC
6 PUSH AX             ; AX saved...
7 ; ... work ...
8 RET                 ; forgot POP AX!
9 bad ENDP
```

VARIABLES

AX = **1234h** | IP = **bad** | SP = **FFFCh**

CALL STACK

- **[FFFCh] 0106h**
- [FFFEh] 0000h

CALL: return address 0106h pushed.

Code Trace: Unbalanced PUSH / POP

```
1 ORG 100h
2 MOV  AX, 1234h      ; 0100h: B8 34 12
3 CALL bad            ; 0103h: ret = 0106h
4 RET                 ; 0106h
5 bad PROC
▶ PUSH AX             ; AX saved...
7 ; ... work ...
8 RET                 ; forgot POP AX!
9 bad ENDP
```

VARIABLES

`AX = 1234h` `IP = bad` `SP = FFFAh`

CALL STACK

- `[FFFAh] 1234h`
- `[FFFCh] 0106h`
- `[FFFEh] 0000h`

`PUSH AX` : 1234h is pushed **on top of** the return address.

Step 3 / 4

Code Trace: Unbalanced PUSH / POP

```
1 ORG 100h
2 MOV  AX, 1234h      ; 0100h: B8 34 12
3 CALL bad            ; 0103h: ret = 0106h
4 RET                 ; 0106h
5 bad PROC
6 PUSH AX             ; AX saved...
7 ; ... work ...
▶ RET                 ; forgot POP AX!
9 bad ENDP
```

VARIABLES

AX = **1234h** IP = **1234h** SP = **FFFCh**

CALL STACK

[FFFCh] 0106h

[FFFEh] 0000h

`RET` pops whatever is on top: IP = **1234h**! The CPU jumps to a random address; the program crashes or behaves unpredictably.

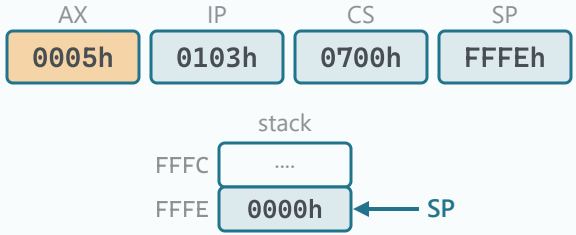
Step 4 / 4

Code Trace: Near CALL, Far Return

```
1 ORG  100h
▶ MOV  AX, 5
3 CALL farp          ; near CALL: IP only
4 ADD  AX, 1
5 RET
6 farp PROC
7 SHL  AX, 1
8 RETF               ; pops IP and CS!
9 farp ENDP
```

VARIABLES

`AX = 0005h` `IP = 0103h` `CS = 0700h` `SP = FFFEh`



AX = 5. The farp procedure ends with RETF, but the call is near.

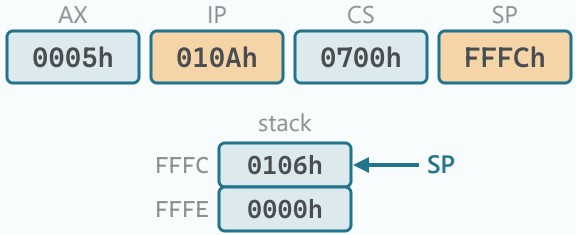
Step 1 / 5

Code Trace: Near CALL, Far Return

```
1 ORG  100h
2 MOV  AX, 5
▶ CALL farp          ; near CALL: IP only
4 ADD  AX, 1
5 RET
6 farp PROC
7 SHL  AX, 1
8 RETF               ; pops IP and CS!
9 farp ENDP
```

VARIABLES

AX = **0005h** IP = **010Ah** CS = **0700h** SP = **FFFCh**



A near `CALL` pushes only IP (0106h): 2 bytes on the stack.

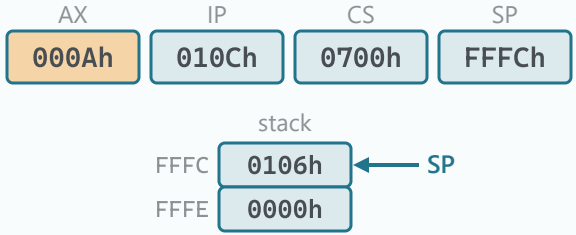
Step 2 / 5

Code Trace: Near CALL, Far Return

```
1 ORG  100h
2 MOV  AX, 5
3 CALL farp          ; near CALL: IP only
4 ADD  AX, 1
5 RET
6 farp PROC
▶ SHL  AX, 1
8 RETF               ; pops IP and CS!
9 farp ENDP
```

VARIABLES

AX = 000Ah | IP = 010Ch | CS = 0700h | SP = FFFCh



`SHL AX, 1` : AX = 000Ah.

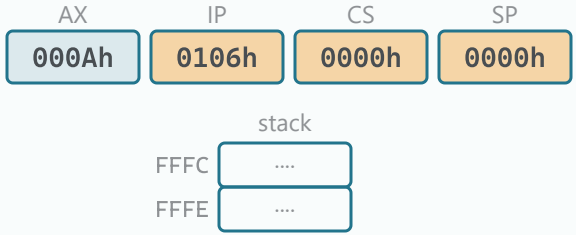
Step 3 / 5

Code Trace: Near CALL, Far Return

```
1 ORG  100h
2 MOV  AX, 5
3 CALL farp          ; near CALL: IP only
4 ADD  AX, 1
5 RET
6 farp PROC
7 SHL  AX, 1
▶ RETF               ; pops IP and CS!
9 farp ENDP
```

VARIABLES

`AX = 000Ah` `IP = 0106h` `CS = 0000h` `SP = 0000h`



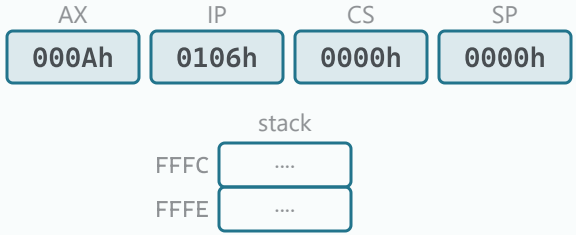
`RETF` pops two words: IP = 0106h and, as CS, whatever it finds on the stack, here the 0000h DOS pushed for the .COM program. SP = 0000h.

Code Trace: Near CALL, Far Return

```
1 ORG  100h
2 MOV  AX, 5
3 CALL farp          ; near CALL: IP only
4 ADD  AX, 1
5 RET
6 farp PROC
7 SHL  AX, 1
▶ RETF               ; pops IP and CS!
9 farp ENDP
```

VARIABLES

`AX = 000Ah` `IP = 0106h` `CS = 0000h` `SP = 0000h`



The CPU jumps to 0000:0106h: that is the interrupt vector table, not program code. The program crashes; `ADD AX, 1` never runs.

Step 5 / 5

Common Bugs

Forgetting RET

The CPU never sees ENDP; whatever code or data follows the procedure is executed.

Unbalanced PUSH / POP

RET loads the wrong value into IP. Every PUSH needs a POP, in reverse order.

"Falling into" a procedure

A procedure placed in the middle of the main code and not jumped over runs without a CALL; its RET ends the program.

JMP instead of CALL

JMP pushes no return address; the procedure's RET returns to some unrelated value on the stack.

Name mismatch

`p1 PROC` … `p2 ENDP` → assembler error.

Infinite recursion

Without a stopping condition, every call to a self-calling procedure eats 2 bytes until the stack overflows.

COMMON BUGS

More Procedure Bugs

Early RET

A conditional RET after a PUSH takes the saved value as the return address. Route all paths to one exit label with the POPs.

MOV instead of LEA

`MOV SI, arr` loads the array's first word, not its address; the procedure reads from an unrelated address.

DX silently clobbered

16-bit MUL and DIV also write DX. A procedure whose header lacks “Clobbers: DX” wipes out the caller's DX.

Type mismatch

Near CALL + RETF, or CALL FAR + RET: the stack is off by 2 bytes and the return goes to the wrong address.

Clobbering the flag

In a procedure that returns its result in CF, an instruction such as ADD or CMP after STC/CLC changes CF. POP and RET leave the flags alone.

Testing the base case too late

If a recursive procedure tests its stopping condition after the CALL, the calls never stop and the stack overflows.

Lab: Procedures

CALL m1

CODE · **11 bytes**

```
1 ORG    100h
2 CALL   m1          ; 0100h:…
3 MOV    AX, 2       ; 0103h:…
4 RET                ; 0106h:…
5 m1 PROC
6 MOV    BX, 5       ; 0107h:…
7 RET                ; 010Ah:…
8 m1 ENDP
9 END
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	000B	CH	00	CL	0B
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 E8 04 00 B8 02 00 C3 BB
0108 05 00 C3 00 00 00 00 00
0110 00 00 00 00 00 00 00 00
0118 00 00 00 00 00 00 00 00
0120 00 00 00 00 00 00 00 00
```

STACK **SS:SP**

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

OUTPUT

Program loaded: 11 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (000Bh). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

Procedure
A piece of code for one task, called with CALL and returning with RET.

PROC / ENDP
Directives; no machine code. CALL and RET are CPU instructions.

CALL
Pushes the address of the next instruction (SP −= 2), IP ← target.

RET
IP ← [SP], SP += 2. In .COM the last RET goes to 0000h → INT 20h.

Parameters
Register, memory variable, address (SI/DI/BX) or stack: read with [BP+4], [BP+6]..., removed with RET n.

Safety
Preserve registers with PUSH/POP, keep the stack balanced, document inputs/outputs.

SUMMARY

Exercises

1. The `CALL p` at address 0120h calls p at 0100h. What is the machine code? What is pushed?
2. In the m2 example, what would the final AX be if `MOV AL, 3` were added before the second `CALL m2`?
3. With SP = FFFEh, 3 CALLs run in a row (no RET). What is SP?
4. Why is it wrong to call a procedure that clobbers CX inside a LOOP? Suggest two fixes.
5. Write an `upper` procedure that converts a lowercase letter in AL to uppercase (input/output AL).
6. Change print_me so that it returns the number of characters printed in CX.

▶ **Hints / answers**

References

- emu8086 documentation: *8086 Assembler Tutorial for Beginners* (Part 8: Procedures).
- Intel, *The 8086 Family User's Manual*, 1979: CALL, RET and stack operations.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009: Section 6.3 "Procedures".
- Kip R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014: Chapter 5 "Procedures".

The End

Chapter 8.2: Procedures

Previous: Chapter 8.1: The Stack · Next: Chapter 9.1: Macros