Chapter 7: Flow Control

Microprocessors

IP and sequential execution | Unconditional jump (JMP) | Single-flag jumps | Comparing with CMP | Signed / unsigned jumps | The LOOP family | Nested loops | Example programs

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_07_Flow_Control.html

Learning Outcomes

How jumps work

Direct program flow with JMP and labels; explain how IP changes and how the displacement (offset) is computed.

Conditional jumps

State which flag (ZF, CF, SF, OF, PF) each conditional jump tests and name its opposite.

Correct comparisons

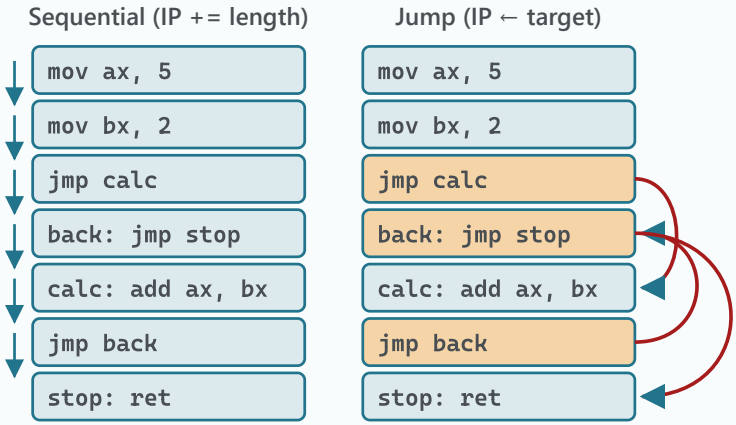
After CMP, choose JA/JB for unsigned numbers and the JG/JL family for signed ones.

Loops

Write counted loops with LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ and JCXZ; trace the example programs step by step.

Program Flow Control

- Normally the CPU executes instructions in memory order: after each instruction, **IP** (Instruction Pointer) advances by the instruction's length.
- **Flow control instructions** load a new value into IP and move execution somewhere else.
- This is how decisions (if/else) and loops (for/while) are built at the machine level.



Unconditional jump

`JMP target`: always jumps.

Conditional jumps

Single-flag tests, plus signed and unsigned comparisons.

Loops

`LOOP`, `LOOPE/LOOPZ`, `LOOPNE/LOOPNZ`, `JCXZ`.

JMP: Unconditional Jump

- `JMP target` transfers control to another point in the program **unconditionally**; it does not change the flags.
- It can jump both **forward** and **backward**.
- A near jump reaches anywhere in the current code segment (64 KB = 65,536 bytes, offsets 0000h–FFFFh); only IP changes.
- A far jump changes CS and IP together and moves to another segment.
- If the target is close, the assembler can pick the 2-byte **short** form automatically.

Form	Code	Bytes	Range
`JMP SHORT`	EB rel8	2	−128… +127
`JMP NEAR`	E9 rel16	3	within segment
`JMP FAR`	EA ofs seg	5	CS:IP
`JMP reg/mem`	FF /4	2–4	indirect

rel8 / rel16: the signed distance of the target from the instruction **after** the jump.

Labels

- To define a label, write its name followed by `:` : `again:`
- It may contain letters, digits and characters such as `_`, but it **cannot start with a digit**.
- It can stand on its own line or in front of an instruction: `back: jmp stop`
- A label takes no memory; the assembler turns it into an **address** (offset).
- Instruction and register names (`mov` , `ax` ...) cannot be labels; each label is defined once.



Valid

+ `again:`
+ `next_char:`
+ `k1:` `calc:`

Invalid

– `1start:` starts with a digit
– `mov:` instruction name
– `two words:` contains a space

Code Trace: Forward and Backward with `JMP`

```
1  org    100h
2  mov    ax, 5          ; ax = 5
3  mov    bx, 2          ; bx = 2
4  jmp    calc           ; go to 'calc'
5  back:  jmp stop       ; go to 'stop'
6  calc:
7  add    ax, bx         ; ax = ax + bx
8  jmp    back           ; back to 'back'
9  stop:
10 ret                   ; return to the OS
```

VARIABLES

`IP = 0100h`

The program starts at address 0100h (`org 100h` , a .COM program). IP = 0100h.

Step 1 / 8

Code Trace: Forward and Backward with `JMP`

```
1  org    100h
▶  mov    ax, 5          ; ax = 5
3  mov    bx, 2          ; bx = 2
4  jmp    calc           ; go to 'calc'
5  back:  jmp stop       ; go to 'stop'
6  calc:
7  add    ax, bx         ; ax = ax + bx
8  jmp    back           ; back to 'back'
9  stop:
10 ret                   ; return to the OS
```

VARIABLES

`AX = 0005h` `IP = 0103h`

`mov ax, 5` (3 bytes) executed: AX = 0005h, IP = 0103h.

Step 2 / 8

Code Trace: Forward and Backward with `JMP`

```
1  org    100h
2  mov    ax, 5          ; ax = 5
▶  mov    bx, 2          ; bx = 2
4  jmp    calc           ; go to 'calc'
5  back:  jmp stop       ; go to 'stop'
6  calc:
7  add    ax, bx         ; ax = ax + bx
8  jmp    back           ; back to 'back'
9  stop:
10 ret                   ; return to the OS
```

VARIABLES

`AX = 0005h` `BX = 0002h` `IP = 0106h`

`mov bx, 2` : BX = 0002h. Next is the `jmp calc` at 0106h.

Step 3 / 8

Code Trace: Forward and Backward with `JMP`

```
1  org   100h
2  mov   ax, 5          ; ax = 5
3  mov   bx, 2          ; bx = 2
▶  jmp   calc           ; go to 'calc'
5  back: jmp stop       ; go to 'stop'
6  calc:
7  add   ax, bx         ; ax = ax + bx
8  jmp   back           ; back to 'back'
9  stop:
10 ret                  ; return to the OS
```

VARIABLES

`AX = 0005h` `BX = 0002h` `IP = 010Ah`

`jmp calc` : IP ← calc = 010Ah. The `back:` line in between is skipped.

Step 4 / 8

Code Trace: Forward and Backward with `JMP`

```
1  org   100h
2  mov   ax, 5          ; ax = 5
3  mov   bx, 2          ; bx = 2
4  jmp   calc           ; go to 'calc'
5  back: jmp stop       ; go to 'stop'
6  calc:
▶  add   ax, bx         ; ax = ax + bx
8  jmp   back           ; back to 'back'
9  stop:
10 ret                  ; return to the OS
```

VARIABLES

`AX = 0007h` `BX = 0002h` `IP = 010Ch`

`add ax, bx` : AX = 5 + 2 = 0007h.

Step 5 / 8

Code Trace: Forward and Backward with `JMP`

```
1  org   100h
2  mov   ax, 5          ; ax = 5
3  mov   bx, 2          ; bx = 2
4  jmp   calc           ; go to 'calc'
5  back: jmp stop       ; go to 'stop'
6  calc:
7  add   ax, bx         ; ax = ax + bx
▶  jmp   back           ; back to 'back'
9  stop:
10 ret                  ; return to the OS
```

VARIABLES

AX = **0007h** BX = **0002h** IP = **0108h**

`jmp back` : a **backward** jump, IP ← 0108h.

Step 6 / 8

Code Trace: Forward and Backward with `JMP`

```
1  org    100h
2  mov    ax, 5          ; ax = 5
3  mov    bx, 2          ; bx = 2
4  jmp    calc           ; go to 'calc'
▶  back:  jmp stop       ; go to 'stop'
6  calc:
7  add    ax, bx         ; ax = ax + bx
8  jmp    back           ; back to 'back'
9  stop:
10 ret                   ; return to the OS
```

VARIABLES

AX = **0007h** BX = **0002h** IP = **010Eh**

`back: jmp stop` : forward again, IP ← 010Eh.

Step 7 / 8

Code Trace: Forward and Backward with `JMP`

```
1 org    100h
2 mov    ax, 5          ; ax = 5
3 mov    bx, 2          ; bx = 2
4 jmp    calc           ; go to 'calc'
5 back:  jmp stop       ; go to 'stop'
6 calc:
7 add    ax, bx         ; ax = ax + bx
8 jmp    back           ; back to 'back'
9 stop:
▶ ret                   ; return to the OS
```

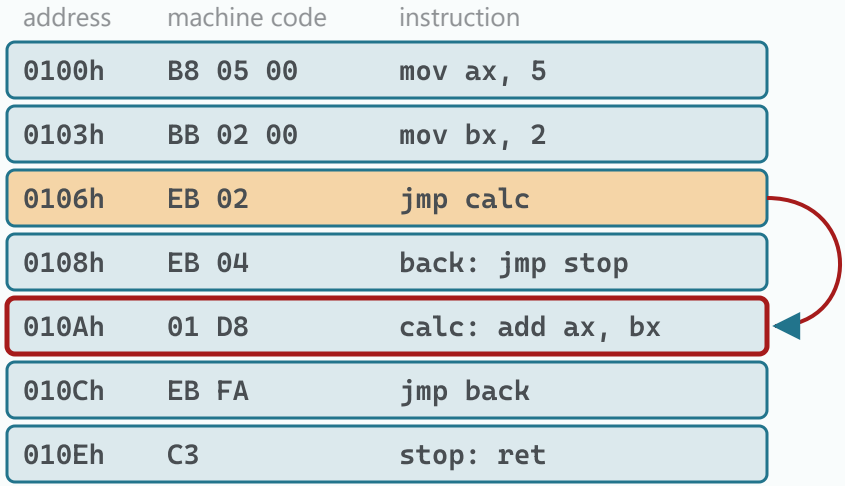
VARIABLES

`AX = 0007h` `BX = 0002h` `IP = 0000h`

`ret` : IP ← 0000h, the program ends. Result AX = 0007h. Flow: 1 → 2 → 3 → calc → back → stop.

Step 8 / 8

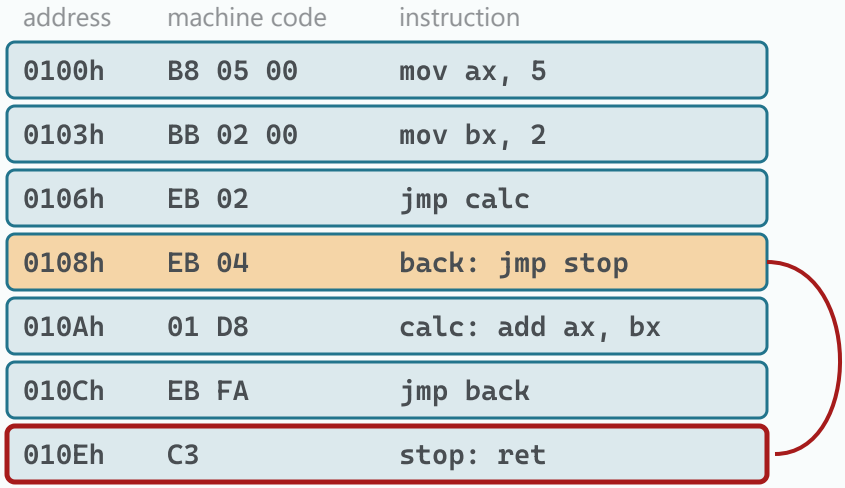
Computing the Displacement




```
jmp calc @ 0106h (2 bytes)
next IP = 0108h
target calc = 010Ah
rel8 = 010A - 0108 = +2
code: EB 02
```

Forward jump: the distance is measured from the address **after** the jump → **+2**.

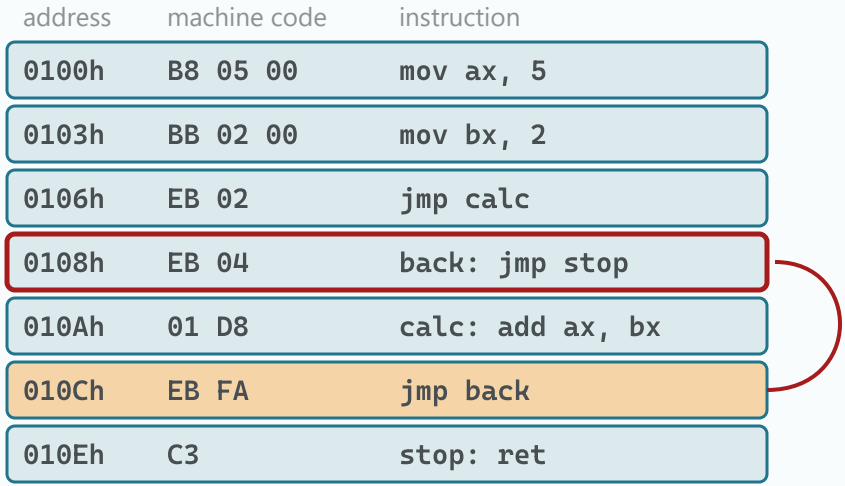
Computing the Displacement




```
jmp stop @ 0108h
next IP = 010Ah
target stop = 010Eh
rel8 = 010E − 010A = +4
code: EB 04
```

At run time the CPU computes IP ← next IP + rel8 = 010Ah + 4 = **010Eh**.

Computing the Displacement

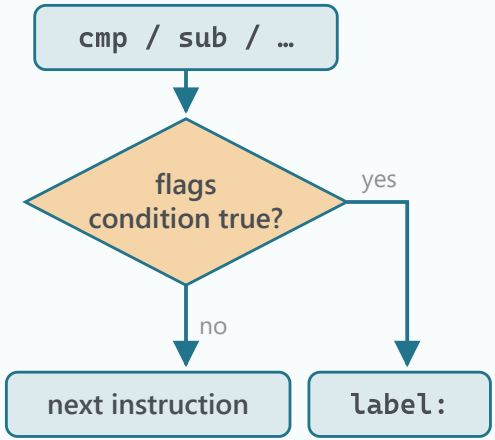



```
jmp back @ 010Ch
next IP = 010Eh
target back = 0108h
rel8 = 0108 - 010E = -6
-6 = FAh (two's complement)
code: EB FA
```

Backward jump: the distance is negative and stored in two's complement: −6 → **FAh**. 010Eh + FFFAh = 0108h (the carry out is dropped).

The Idea of a Conditional Jump

- `Jcc target` : if the condition (cc) holds, IP ← target; otherwise the next instruction runs.
- The condition depends on the **flags** set by an earlier instruction (usually `CMP` , `SUB` , `ADD` , `TEST` , `DEC`).
- Jump instructions themselves **do not change** the flags.



ZF result is zero	**CF** carry / borrow	**SF** result is negative
OF signed overflow	**PF** even number of 1s	**AF** not used by jumps

Jumps That Test a Single Flag

Instruction	Meaning	Condition	Opposite
`JZ, JE`	Jump if Zero (Equal)	ZF = 1	JNZ, JNE
`JC, JB, JNAE`	Jump if Carry (Below, Not Above or Equal)	CF = 1	JNC, JNB, JAE
`JS`	Jump if Sign: result negative	SF = 1	JNS
`JO`	Jump if Overflow: signed overflow	OF = 1	JNO
`JPE, JP`	Jump if Parity Even	PF = 1	JPO, JNP
`JNZ, JNE`	Jump if Not Zero (Not Equal)	ZF = 0	JZ, JE
`JNC, JNB, JAE`	Jump if Not Carry (Not Below, Above or Equal)	CF = 0	JC, JB, JNAE
`JNS`	Jump if Not Sign: not negative	SF = 0	JS
`JNO`	Jump if Not Overflow	OF = 0	JO
`JPO, JNP`	Jump if Parity Odd (No Parity)	PF = 0	JPE, JP

The "C" in JC is the carry flag; "overflow" refers to OF. They are different things.

Encoding and Synonyms

- On the 8086, conditional jumps have a fixed length: **2 bytes** = opcode + a 1-byte displacement (rel8).
- rel8 is always **signed**: relative to the next instruction, a jump can go **128 bytes back** or **127 bytes forward**.
- Different names for the same condition (synonyms) assemble to **the same machine code**: JE = JZ, JNE = JNZ, JC = JB = JNAE, JNC = JNB = JAE.
- Conditional jumps with 16-bit displacements (0F 8x) came only with the 80386; the 8086 does not have them.

74 JE / JZ	**75** JNE / JNZ	**72** JB / JC / JNAE	**73** JAE / JNB / JNC
78 JS	**79** JNS	**70** JO	**71** JNO
7A JP / JPE	**7B** JNP / JPO	**77** JA / JNBE	**76** JBE / JNA
7F JG / JNLE	**7D** JGE / JNL	**7C** JL / JNGE	**7E** JLE / JNG
E2 LOOP	**E1** LOOPE / LOOPZ	**E0** LOOPNE / LOOPNZ	**E3** JCXZ

All are 2 bytes: opcode + rel8. The short JMP opcode is EBh.

Code Trace: `JNC` = `JNB` = `JAE`

```
▶ org 100h
2 jnc a          ; 0100h: 73 07
3 jnb a          ; 0102h: 73 05
4 jae a          ; 0104h: 73 03
5 mov ax, 4      ; 0106h: B8 04 00
6 a: mov ax, 5   ; 0109h: B8 05 00
7 ret            ; 010Ch: C3
```

VARIABLES

`CF = 0` `AX = 0000h` `IP = 0100h`

All three test the same condition (CF = 0) and produce the same opcode (**73h**); only rel8 differs, because each sits at a different address.

Step 1 / 4

Code Trace: `JNC` = `JNB` = `JAE`

```
1 org 100h
▶ jnc a          ; 0100h: 73 07
3 jnb a          ; 0102h: 73 05
4 jae a          ; 0104h: 73 03
5 mov ax, 4      ; 0106h: B8 04 00
6 a: mov ax, 5   ; 0109h: B8 05 00
7 ret            ; 010Ch: C3
```

VARIABLES

`CF = 0` `AX = 0000h` `IP = 0109h`

CF = 0, so `jnc a` jumps: IP ← 0102h + 07h = **0109h**. The `jnb`, `jae` and `mov ax, 4` in between do not run.

Step 2 / 4

Code Trace: `JNC` = `JNB` = `JAE`

```
1 org 100h
2 jnc a          ; 0100h: 73 07
3 jnb a          ; 0102h: 73 05
4 jae a          ; 0104h: 73 03
5 mov ax, 4      ; 0106h: B8 04 00
▶ a: mov ax, 5   ; 0109h: B8 05 00
7 ret            ; 010Ch: C3
```

VARIABLES

`CF = 0` `AX = 0005h` `IP = 010Ch`

`a: mov ax, 5` → AX = 0005h.

Step 3 / 4

Code Trace: `JNC` = `JNB` = `JAE`

```
1 org 100h
2 jnc a          ; 0100h: 73 07
3 jnb a          ; 0102h: 73 05
4 jae a          ; 0104h: 73 03
5 mov ax, 4      ; 0106h: B8 04 00
6 a: mov ax, 5   ; 0109h: B8 05 00
▶ ret            ; 010Ch: C3
```

VARIABLES

CF = **0** | AX = **0005h** | IP = **0000h**

`ret` : IP ← 0000h. Had CF been 1, none of the three would jump, and AX would become 4, then 5.

Step 4 / 4

CONDITIONAL JUMPS

Jump Code Calculator

Enter the address of the jump and the target address (hex) to get the rel8 / rel16 value and the machine code.

Instruction	address	target
JMP SHORT (EB)	010C	0108

```
next IP = 010Ch + 2 = 010Eh
displacement = 0108h − 010Eh = −6
rel8 = FAh → machine code EB FA
```

JMP SHORT at 010Ch, target 0108h.

Rule: `displacement = target − (address + instruction length)`. In the short form the result must be within −128…+127; otherwise the assembler reports *"relative jump out of range"*.

COMPARISON

The CMP Instruction

- Use `CMP dest, src` to compare numbers.
- CMP performs a **SUB**: `dest − src`, but it **discards the result**; it only sets the flags (CF, ZF, SF, OF, PF, AF).
- The conditional jump that follows decides based on these flags.
- Operands: register–register, register–memory, memory–register, register/memory–immediate. Two memory operands are **not allowed**.

CMP 05h, 02h

`5 − 2 = 3 → 03h`

ZF = 0, CF = 0, SF = 0, OF = 0

CMP 07h, 07h

`7 − 7 = 0 → 00h`

ZF = 1, CF = 0, SF = 0, OF = 0

Example 1: 5 − 2 = 3 → the result is not zero, **ZF = 0**. Example 2: 7 − 7 = 0 → the result is zero, **ZF = 1**, and `JE` / `JZ` jumps.

COMPARISON

How Does `CMP AL, BL` Set the Flags?

AL	0000 0101	05h	unsigned 5 · signed 5
BL	0000 0010	02h	unsigned 2 · signed 2
AL−BL	0000 0011	03h	result not kept, only the flags

CF=0 ZF=0 SF=0 OF=0 PF=1 AF=0

unsigned: AL > BL

CF=0, ZF=0 → JA jumps

signed: AL > BL

SF=0, OF=0, ZF=0 → JG jumps

Positive difference: no flag is set except PF (PF = 1 because 03h = 0000 0011 has two 1s).

COMPARISON

How Does `CMP AL, BL` Set the Flags?

AL	0000 0111	07h	unsigned 7 · signed 7
BL	0000 0111	07h	unsigned 7 · signed 7
AL−BL	0000 0000	00h	result not kept, only the flags

CF=0 ZF=1 SF=0 OF=0 PF=1 AF=0

unsigned: AL = BL

CF=0, ZF=1 → JE jumps

signed: AL = BL

SF=0, OF=0, ZF=1 → JE jumps

Equal numbers: the result is 0 → **ZF = 1**. Equal whether signed or unsigned.

COMPARISON

How Does `CMP AL, BL` Set the Flags?

AL	0000 0101	05h	unsigned 5 · signed 5
BL	0000 0111	07h	unsigned 7 · signed 7
AL−BL	1111 1110	FEh	result not kept, only the flags

CF=1 ZF=0 SF=1 OF=0 PF=0 AF=1

unsigned: AL < BL

CF=1, ZF=0 → JB jumps

signed: AL < BL

SF=1, OF=0, ZF=0 → JL jumps

Subtracting a larger number needs a borrow → **CF = 1**; the result FEh = −2 → **SF = 1**, no overflow.

COMPARISON

How Does `CMP AL, BL` Set the Flags?

AL	0111 1111	7Fh	unsigned 127 · signed 127
BL	1000 0000	80h	unsigned 128 · signed -128
AL−BL	1111 1111	FFh	result not kept, only the flags

CF=1 ZF=0 SF=1 OF=1 PF=1 AF=0

unsigned: AL < BL

CF=1, ZF=0 → JB jumps

signed: AL > BL

SF=1, OF=1, ZF=0 → JG jumps

127 − (−128) = 255 does not fit in a signed byte → **OF = 1**. Unsigned, 127 < 128 (CF = 1); signed, 127 > −128 (SF = OF).

COMPARISON

How Does `CMP AL, BL` Set the Flags?

AL	1111 1011	FBh	unsigned 251 · signed -5
BL	0000 0011	03h	unsigned 3 · signed 3
AL−BL	1111 1000	F8h	result not kept, only the flags

CF=0 ZF=0 SF=1 OF=0 PF=0 AF=0

unsigned: AL > BL

CF=0, ZF=0 → JA jumps

signed: AL < BL

SF=1, OF=0, ZF=0 → JL jumps

FBh is 251 unsigned and −5 signed. Unsigned: 251 > 3 (CF = 0); signed: −5 < 3 (SF ≠ OF). Same bits, different decision!

COMPARISON

Comparison Calculator

Enter two bytes (decimal −128…255 or hex such as `7Fh`) to see the result of `CMP A, B`, the flags and which jumps are taken.

A: FBh

B: 3

A = 1111 1011 (FBh: 251 / −5)
B = 0000 0011 (03h: 3 / 3)
A − B = 1111 1000 = **F8h**

CF=0 ZF=0 **SF=1** OF=0 PF=0 AF=0

~~JE/JZ~~ JNE/JNZ JA JAE/JNC ~~JB/JC~~ ~~JBE~~ ~~JG~~ ~~JGE~~ JL JLE JS ~~JNS~~ ~~JO~~ JNO ~~JP~~ JNP

unsigned: 251 > 3 · signed: −5 < 3

COMPARISON

Jumps for Signed Numbers

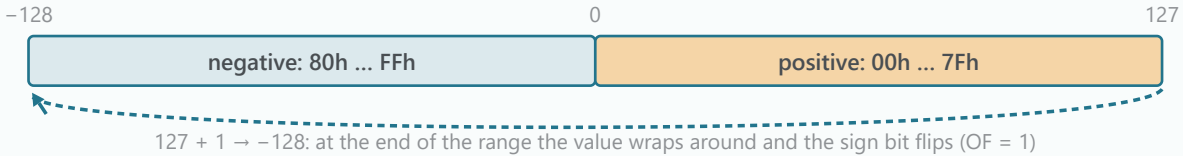
Instruction	Meaning	Condition	Opposite
`JE, JZ`	Jump if Equal (=) · Jump if Zero	ZF = 1	JNE, JNZ
`JNE, JNZ`	Jump if Not Equal (<>) · Jump if Not Zero	ZF = 0	JE, JZ
`JG, JNLE`	Jump if Greater (>) · Not Less or Equal	ZF = 0 and SF = OF	JNG, JLE
`JL, JNGE`	Jump if Less (<) · Not Greater or Equal	SF ≠ OF	JNL, JGE
`JGE, JNL`	Jump if Greater or Equal (>=) · Not Less	SF = OF	JNGE, JL
`JLE, JNG`	Jump if Less or Equal (<=) · Not Greater	ZF = 1 or SF ≠ OF	JNLE, JG

<> means "not equal". **G**reater / **L**ess assume the numbers are signed, in **two's complement**.

COMPARISON

Why SF = OF?

- CMP A, B computes A − B. Without overflow the sign of the result is correct: **SF = 1 ⇔ A < B**.
- With overflow (OF = 1) the result did not fit in 8/16 bits and the **sign bit is flipped**.
- So the true sign is SF XOR OF: **SF ≠ OF ⇔ A < B**, **SF = OF ⇔ A ≥ B**.



A	B	A − B (8 bit)	SF	OF	Result
5	7	FEh (−2)	1	0	SF ≠ OF → 5 < 7
127 (7Fh)	−128 (80h)	FFh (−1) ✗ true: +255	1	1	SF = OF → 127 > −128
−128 (80h)	1	7Fh (+127) ✗ true: −129	0	1	SF ≠ OF → −128 < 1
−5 (FBh)	3	F8h (−8)	1	0	SF ≠ OF → −5 < 3

In the overflow rows, SF alone would give the wrong answer; combined with OF the decision is correct.

COMPARISON

Jumps for Unsigned Numbers

Instruction	Meaning	Condition	Opposite
`JE, JZ`	Jump if Equal (=) · Jump if Zero	ZF = 1	JNE, JNZ
`JNE, JNZ`	Jump if Not Equal (<>) · Jump if Not Zero	ZF = 0	JE, JZ
`JA, JNBE`	Jump if Above (>) · Not Below or Equal	CF = 0 and ZF = 0	JNA, JBE
`JB, JNAE, JC`	Jump if Below (<) · Not Above or Equal · Carry	CF = 1	JNB, JAE, JNC
`JAE, JNB, JNC`	Jump if Above or Equal (>=) · Not Below · Not Carry	CF = 0	JNAE, JB, JC
`JBE, JNA`	Jump if Below or Equal (<=) · Not Above	CF = 1 or ZF = 1	JNBE, JA

In unsigned subtraction, if A < B a borrow occurs and **CF = 1**. So the "below" test is just the carry flag: JB ≡ JC.

COMPARISON

Above/Below or Greater/Less?

Comparison	Unsigned (A/B)	Signed (G/L)
A = B	JE / JZ	JE / JZ
A ≠ B	JNE / JNZ	JNE / JNZ
A > B	JA / JNBE	JG / JNLE
A ≥ B	JAE / JNB	JGE / JNL
A < B	JB / JNAE	JL / JNGE
A ≤ B	JBE / JNA	JLE / JNG

Mnemonic

Above / **B**elow → unsigned (addresses, counters, ASCII codes). **G**reater / **L**ess → signed (temperatures, differences, values that may be negative).

Typical bug

With AL = FBh, BL = 03h, `JA` jumps (251 > 3) but `JG` does not (−5 < 3). Choose the family that matches the data.

Code Trace: Same Bits, JA and JG

```
▶ MOV  AL, 0FBh     ; 251 or -5
▶ MOV  BL, 03h
3  CMP  AL, BL      ; FBh - 03h
4  JA   a1          ; unsigned: 251 > 3 ?
5  MOV  CL, 1       ; if JA failed
6  a1:  JG  g1      ; signed: -5 > 3 ?
7  MOV  CH, 1       ; JG not taken
8  g1:  MOV AL, 7Fh ; 127
9  CMP  AL, 80h     ; 127 - (-128)
10 JG   g2          ; SF = OF ?
11 MOV  DL, 1       ; if JG failed
12 g2:  RET
```

VARIABLES

AL = FBh | CF = 0 | SF = 0 | OF = 0

AL	BL	CH	DL
FBh	03h	00h	00h

CF	ZF	SF	OF
0	0	0	0

AL = 251
signed -5

AL = FBh: unsigned 251, signed −5. BL = 3. CL, CH, DL will mark which jump was not taken.

Step 1 / 9

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
▶  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

`AL = FBh` `CF = 0` `SF = 1` `OF = 0`

AL	BL	CH	DL
FBh	03h	00h	00h

CF	ZF	SF	OF
0	0	1	0

AL = 251
signed -5

`CMP AL, BL`: FBh − 03h = F8h. No borrow → CF = 0; negative result → SF = 1; no overflow → OF = 0.

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
▶  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

AL = **FBh** | CF = **0** | SF = **1** | OF = **0**

AL	BL	CH	DL
FBh	03h	00h	00h

CF	ZF	SF	OF
0	0	1	0

AL = 251
signed -5

JA → jumps

`JA`: CF = 0 and ZF = 0 → jumps (251 > 3). The CL = 1 line is skipped.

Step 3 / 9

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
▶  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

AL = **FBh** | CF = **0** | SF = **1** | OF = **0**

AL	BL	CH	DL
FBh	03h	00h	00h

CF	ZF	SF	OF
0	0	1	0

AL = 251
signed -5

JG → no jump

`JG`: ZF = 0 but SF ≠ OF → no jump (−5 < 3).

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
▶  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

AL = **FBh** | CF = **0** | SF = **1** | OF = **0**

AL	BL	CH	DL
FBh	03h	01h	00h

CF	ZF	SF	OF
0	0	1	0

AL = 251
signed -5

CH = 1: in the signed comparison AL is not greater. One CMP, two different decisions.

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
▶  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

AL = 7Fh | CF = 0 | SF = 1 | OF = 0

AL	BL	CH	DL
7Fh	03h	01h	00h

CF	ZF	SF	OF
0	0	1	0

AL = 127
signed 127

AL = 7Fh = 127. Now compare 127 with 80h (−128).

Step 6 / 9

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
▶  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

AL = **7Fh** | CF = **1** | SF = **1** | OF = **1**

AL	BL	CH	DL
7Fh	03h	01h	00h

CF	ZF	SF	OF
1	0	1	1

AL = 127
signed 127

7Fh − 80h = FFh: signed 127 − (−128) = 255 does not fit → OF = 1, the result's sign bit SF = 1. Borrow → CF = 1.

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
▶  JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
12 g2:  RET
```

VARIABLES

AL = **7Fh** | CF = **1** | SF = **1** | OF = **1**

AL	BL	CH	DL
7Fh	03h	01h	00h

CF	ZF	SF	OF
1	0	1	1

AL = 127
signed 127

JG → jumps

`JG`: ZF = 0 and SF = OF (1 = 1) → jumps: 127 > −128 is true. SF alone would have said “less”.

Code Trace: Same Bits, JA and JG

```
1  MOV  AL, 0FBh     ; 251 or -5
2  MOV  BL, 03h
3  CMP  AL, BL       ; FBh - 03h
4  JA   a1           ; unsigned: 251 > 3 ?
5  MOV  CL, 1        ; if JA failed
6  a1:  JG  g1       ; signed: -5 > 3 ?
7  MOV  CH, 1        ; JG not taken
8  g1:  MOV AL, 7Fh  ; 127
9  CMP  AL, 80h      ; 127 - (-128)
10 JG   g2           ; SF = OF ?
11 MOV  DL, 1        ; if JG failed
▶  g2:  RET
```

VARIABLES

AL = **7Fh** | CF = **1** | SF = **1** | OF = **1**

AL	BL	CH	DL
7Fh	03h	01h	00h

CF	ZF	SF	OF
1	0	1	1

AL = 127
signed 127

DL stays 0. Result: CH = 1 (−5 > 3 is false), DL = 0 (127 > −128).

Step 9 / 9

COMPARISON · STEP THROUGH THE CODE

Code Trace: `CMP` + `JE`

```
1 ▶ mov ax, 5
2   mov bx, 5
3   cmp ax, bx
4   je  equal_message
5   jmp not_equal_message
6   equal_message:        ; if equal
7   jmp end_program
8   not_equal_message:    ; if not equal
9   end_program:
```

VARIABLES

AX = **0005h**

AX = 0005h.

Step 1 / 7

COMPARISON · STEP THROUGH THE CODE

Code Trace: `CMP` + `JE`

```
1 mov ax, 5
▶ mov bx, 5
3 cmp ax, bx
4 je  equal_message
5 jmp not_equal_message
6 equal_message:        ; if equal
7 jmp end_program
8 not_equal_message:    ; if not equal
9 end_program:
```

VARIABLES

AX = 0005h BX = 0005h

BX = 0005h.

Step 2 / 7

COMPARISON · STEP THROUGH THE CODE

Code Trace: `CMP` + `JE`

```
1 mov ax, 5
2 mov bx, 5
▶ cmp ax, bx
4 je  equal_message
5 jmp not_equal_message
6 equal_message:        ; if equal
7 jmp end_program
8 not_equal_message:    ; if not equal
9 end_program:
```

VARIABLES

AX = **0005h** BX = **0005h** ZF = **1** CF = **0** SF = **0** OF = **0**

`cmp ax, bx` : 5 − 5 = 0 → **ZF = 1**. AX is unchanged.

Step 3 / 7

COMPARISON · STEP THROUGH THE CODE

Code Trace: `CMP` + `JE`

```
1 mov ax, 5
2 mov bx, 5
3 cmp ax, bx
▶ je  equal_message
5 jmp not_equal_message
6 equal_message:        ; if equal
7 jmp end_program
8 not_equal_message:    ; if not equal
9 end_program:
```

VARIABLES

AX = **0005h** BX = **0005h** ZF = **1**

CF = **0** SF = **0** OF = **0**

ZF = 1, so `je` jumps.

Step 4 / 7

Code Trace: `CMP` + `JE`

```
1 mov ax, 5
2 mov bx, 5
3 cmp ax, bx
4 je  equal_message
5 jmp not_equal_message
▶ equal_message:        ; if equal
7 jmp end_program
8 not_equal_message:    ; if not equal
9 end_program:
```

VARIABLES

`AX = 0005h` `BX = 0005h` `ZF = 1`
`CF = 0` `SF = 0` `OF = 0`

`equal_message:` the work for the equal case goes here.

Step 5 / 7

Code Trace: `CMP` + `JE`

```
1 mov ax, 5
2 mov bx, 5
3 cmp ax, bx
4 je  equal_message
5 jmp not_equal_message
6 equal_message:        ; if equal
▶ jmp end_program
8 not_equal_message:    ; if not equal
9 end_program:
```

VARIABLES

`AX = 0005h` `BX = 0005h` `ZF = 1`
`CF = 0` `SF = 0` `OF = 0`

`jmp end_program` : skips the `not_equal_message` block. Without this JMP both blocks would run.

Step 6 / 7

Code Trace: `CMP` + `JE`

```
1 mov ax, 5
2 mov bx, 5
3 cmp ax, bx
4 je  equal_message
5 jmp not_equal_message
6 equal_message:        ; if equal
7 jmp end_program
8 not_equal_message:    ; if not equal
▶ end_program:
```

VARIABLES

AX = **0005h** | BX = **0005h** | ZF = **1** | CF = **0** | SF = **0** | OF = **0**

End of program. If they were not equal, `je` would not jump and the `jmp` on line 5 would go to line 8.

Step 7 / 7

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
2 mov bx, 7
3 cmp ax, bx
4 ja  jump_above
5 jmp not_jump_above
6 jump_above:       ; if ax is greater
7 jmp end_program
8 not_jump_above:   ; if ax is less/equal
9 end_program:
```

VARIABLES

`AX = 0005h`

AX = 0005h.

Step 1 / 7

COMPARISON · STEP THROUGH THE CODE

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
▶ mov bx, 7
3 cmp ax, bx
4 ja  jump_above
5 jmp not_jump_above
6 jump_above:       ; if ax is greater
7 jmp end_program
8 not_jump_above:   ; if ax is less/equal
9 end_program:
```

VARIABLES

AX = 0005h BX = 0007h

BX = 0007h.

Step 2 / 7

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
2 mov bx, 7
▶ cmp ax, bx
4 ja  jump_above
5 jmp not_jump_above
6 jump_above:        ; if ax is greater
7 jmp end_program
8 not_jump_above:    ; if ax is less/equal
9 end_program:
```

VARIABLES

`AX = 0005h` `BX = 0007h` `ZF = 0` `CF = 1` `SF = 1` `OF = 0`

`cmp ax, bx` : 5 − 7 = FFFEh (−2). Borrow → **CF = 1**, ZF = 0.

Step 3 / 7

COMPARISON · STEP THROUGH THE CODE

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
2 mov bx, 7
3 cmp ax, bx
▶ ja  jump_above
5 jmp not_jump_above
6 jump_above:        ; if ax is greater
7 jmp end_program
8 not_jump_above:    ; if ax is less/equal
9 end_program:
```

VARIABLES

AX = **0005h** BX = **0007h** ZF = **0**

CF = **1** SF = **1** OF = **0**

`ja` needs CF = 0 **and** ZF = 0. CF = 1 → no jump.

Step 4 / 7

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
2 mov bx, 7
3 cmp ax, bx
4 ja  jump_above
▶ jmp not_jump_above
6 jump_above:       ; if ax is greater
7 jmp end_program
8 not_jump_above:   ; if ax is less/equal
9 end_program:
```

VARIABLES

`AX = 0005h` `BX = 0007h` `ZF = 0`
`CF = 1` `SF = 1` `OF = 0`

`jmp not_jump_above` goes to the "less or equal" block.

Step 5 / 7

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
2 mov bx, 7
3 cmp ax, bx
4 ja  jump_above
5 jmp not_jump_above
6 jump_above:       ; if ax is greater
7 jmp end_program
▶ not_jump_above:   ; if ax is less/equal
9 end_program:
```

VARIABLES

`AX = 0005h` `BX = 0007h` `ZF = 0`
`CF = 1` `SF = 1` `OF = 0`

`not_jump_above:` the work for AX ≤ BX.

Step 6 / 7

Code Trace: `CMP` + `JA`

```
1 mov ax, 5
2 mov bx, 7
3 cmp ax, bx
4 ja  jump_above
5 jmp not_jump_above
6 jump_above:       ; if ax is greater
7 jmp end_program
8 not_jump_above:   ; if ax is less/equal
▶ end_program:
```

VARIABLES

AX = **0005h** | BX = **0007h** | ZF = **0** | CF = **1** | SF = **1** | OF = **0**

End of program. Using `jg` instead of `ja` would not jump either (SF ≠ OF); with positive values both families agree.

Step 7 / 7

COMPARISON · STEP THROUGH THE CODE

Code Trace: Compare and Print

```
1  org   100h
2  mov   al, 25      ; al = 25
3  mov   bl, 10      ; bl = 10
4  cmp   al, bl      ; compare al - bl
5  je    equal       ; jump if al = bl (zf = 1)
6  mov   dl, 'n'     ; al <> bl: print 'n'
7  jmp   prt         ; go to the print line
8  equal:
9  mov   dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov   ah, 2       ; DOS: print char in DL
12 int   21h
13 ret               ; return to DOS
```

VARIABLES

—

OUTPUT

The character is printed with a DOS service: `INT 21h` with AH = 02h prints the character in DL.

Step 1 / 10

COMPARISON · STEP THROUGH THE CODE

Code Trace: Compare and Print

```
1  org    100h
▶  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
12 int    21h
13 ret                ; return to DOS
```

VARIABLES

AL = **19h**

OUTPUT



AL = 25 = **19h**.

Step 2 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
▶  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
12 int    21h
13 ret                ; return to DOS
```

VARIABLES

AL = **19h** BL = **0Ah**

OUTPUT



BL = 10 = **0Ah**.

Step 3 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
▶  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
12 int    21h
13 ret                ; return to DOS
```

VARIABLES

AL = **19h** | BL = **0Ah** | ZF = **0** | CF = **0** | SF = **0** | OF = **0**

OUTPUT

`cmp al, bl` : 25 − 10 = 15 (0Fh) → ZF = 0.

Step 4 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
▶  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
12 int    21h
13 ret                ; return to DOS
```

VARIABLES

`AL = 19h` `BL = 0Ah` `ZF = 0` `CF = 0` `SF = 0` `OF = 0`

OUTPUT

ZF = 0 → `je` does not jump.

Step 5 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
▶  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
12 int    21h
13 ret                ; return to DOS
```

VARIABLES

AL = **19h** | BL = **0Ah** | DL = **6Eh** | ZF = **0** | CF = **0** | SF = **0** | OF = **0**

OUTPUT

`mov dl, 'n'` : the character to print is in DL (6Eh).

Step 6 / 10

Code Trace: Compare and Print

```
1  org     100h
2  mov     al, 25      ; al = 25
3  mov     bl, 10      ; bl = 10
4  cmp     al, bl      ; compare al - bl
5  je      equal       ; jump if al = bl (zf = 1)
6  mov     dl, 'n'     ; al <> bl: print 'n'
▶  jmp     prt         ; go to the print line
8  equal:
9  mov     dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov     ah, 2       ; DOS: print char in DL
12 int     21h
13 ret                 ; return to DOS
```

VARIABLES

`AL = 19h` `BL = 0Ah` `DL = 6Eh`

`ZF = 0` `CF = 0` `SF = 0` `OF = 0`

OUTPUT

`jmp prt` : skips the 'y' line; both branches meet at the `prt` label.

Step 7 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
▶  mov    ah, 2       ; DOS: print char in DL
12 int    21h
13 ret                ; return to DOS
```

VARIABLES

AL = **19h** | BL = **0Ah** | DL = **6Eh** | AH = **02h** | ZF = **0** | CF = **0** | SF = **0** | OF = **0**

OUTPUT

AH = 02h: DOS character output service.

Step 8 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
▶  int    21h
13 ret                ; return to DOS
```

VARIABLES

AL = **6Eh** | BL = **0Ah** | DL = **6Eh** | AH = **02h** | ZF = **0** | CF = **0** | SF = **0** | OF = **0**

OUTPUT

```
n
```

`int 21h` : prints **n**. DOS also leaves the printed character in AL: AL = 6Eh.

Step 9 / 10

Code Trace: Compare and Print

```
1  org    100h
2  mov    al, 25      ; al = 25
3  mov    bl, 10      ; bl = 10
4  cmp    al, bl      ; compare al - bl
5  je     equal       ; jump if al = bl (zf = 1)
6  mov    dl, 'n'     ; al <> bl: print 'n'
7  jmp    prt         ; go to the print line
8  equal:
9  mov    dl, 'y'     ; al = bl: print 'y'
10 prt:
11 mov    ah, 2       ; DOS: print char in DL
12 int    21h
▶  ret                ; return to DOS
```

VARIABLES

AL = **6Eh** | BL = **0Ah** | DL = **6Eh** | AH = **02h** | ZF = **0** | CF = **0** | SF = **0** | OF = **0**

OUTPUT

```
n
```

A program must end with `ret`; otherwise the CPU would execute the random bytes that follow as instructions.

Step 10 / 10

COMPARISON

The if / else Pattern

High Level

```
if (ax == bx) {
    ; block A
} else {
    ; block B
}
while (cx > 0) {      ; unsigned
    ; body
}
```

Assembly

```
    cmp ax, bx
    jne not_eq      ; the OPPOSITE test
    ; block A
    jmp after
not_eq:
    ; block B
after:
top: cmp cx, 0
    jbe done        ; cx <= 0 → exit
    ; body
    jmp top
done:
```

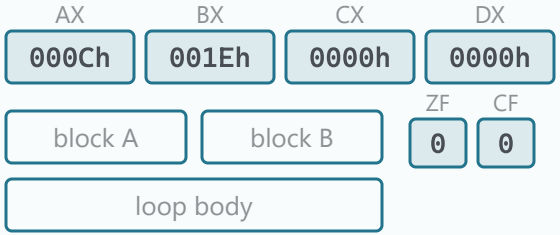
Skipping a block with the **opposite** condition keeps the code in the same order as the high-level version. Forget the `jmp after` at the end of block A, and block B runs too.

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

AX = 12 | BX = 30 | CX = 0 | DX = 0



if (AX == BX) DX = 1; else DX = 2; then, with CX = 2, while (CX > 0) DX += CX. AX = 12, BX = 30.

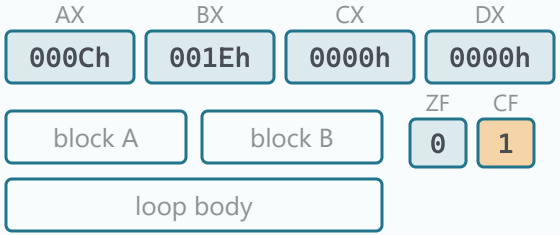
Step 1 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

`AX = 12` `BX = 30` `CX = 0` `DX = 0`



`CMP AX, BX` : 12 − 30 → ZF = 0 (not equal), borrow → CF = 1.

Step 2 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
▶ 4  JNE  not_eq      ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

AX = **12** | BX = **30** | CX = **0** | DX = **0**

AX	BX	CX	DX
000Ch	001Eh	0000h	0000h

block A | block B | loop body

ZF = 0, CF = 1

JNE → jumps

`JNE not_eq` : the opposite condition holds → block A is skipped.

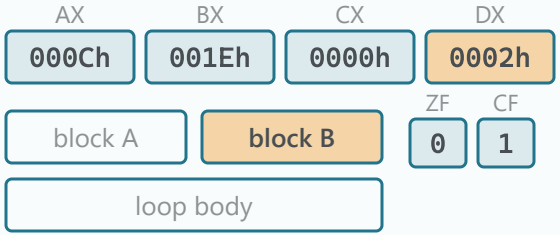
Step 3 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq      ; OPPOSITE test
5  MOV  DX, 1       ; block A: AX = BX
6  JMP  after
▶ not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2 ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done        ; CX ≤ 0 → exit
11 ADD  DX, CX      ; body
12 DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

AX = **12** | BX = **30** | CX = **0** | DX = **2**



Block B: DX = 2. The next line is `after`; the `JMP after` in block A was not needed on this path.

Step 4 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
▶  after: MOV CX, 2  ; while (CX > 0)
▶  top: CMP CX, 0
▶  JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

AX = 12 | BX = 30 | CX = 2 | DX = 2

AX	BX	CX	DX
000Ch	001Eh	0002h	0002h

block A | block B | loop body

ZF	CF
0	0

JBE → no jump

CX = 2. `CMP CX, 0` : 2 > 0 → CF = 0, ZF = 0; `JBE` does not jump, the body is entered.

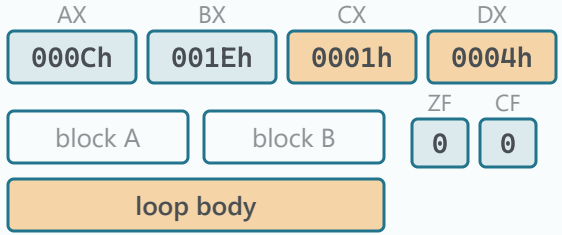
Step 5 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done         ; CX ≤ 0 → exit
▶  ADD  DX, CX       ; body
▶  DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

AX = 12 | BX = 30 | CX = 1 | DX = 4



Body: DX = 2 + 2 = 4; CX = 1.

Step 6 / 10

Code Trace: if / else and while

```
 1 MOV  AX, 12
 2 MOV  BX, 30
 3 CMP  AX, BX
 4 JNE  not_eq       ; OPPOSITE test
 5 MOV  DX, 1        ; block A: AX = BX
 6 JMP  after
 7 not_eq: MOV DX, 2 ; block B: AX ≠ BX
 8 after: MOV CX, 2 ; while (CX > 0)
▶ top: CMP CX, 0
▶ JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
▶ JMP  top
14 done: RET
```

VARIABLES

AX = **12** | BX = **30** | CX = **1** | DX = **4**

AX	BX	CX	DX
000Ch	001Eh	0001h	0004h

block A | block B | loop body

ZF = 0, CF = 0

JBE → no jump

`JMP top`: the condition is tested again; 1 > 0 → JBE does not jump.

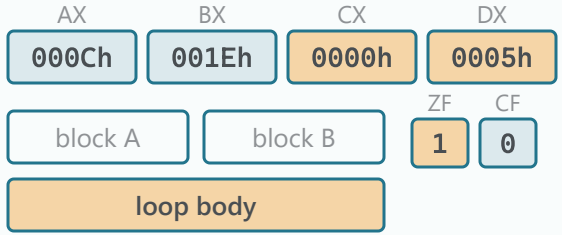
Step 7 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done         ; CX ≤ 0 → exit
▶  ADD  DX, CX       ; body
▶  DEC  CX
13 JMP  top
14 done: RET
```

VARIABLES

AX = 12 | BX = 30 | CX = 0 | DX = 5



DX = 4 + 1 = 5; CX = 0 → DEC sets ZF = 1.

Step 8 / 10

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
▶  top: CMP CX, 0
▶  JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
▶  JMP  top
14 done: RET
```

VARIABLES

`AX = 12` `BX = 30` `CX = 0` `DX = 5`

AX	BX	CX	DX
000Ch	001Eh	0000h	0005h

block A | block B | loop body

ZF = 1, CF = 0

JBE → jumps

Back at the top: CX = 0 → ZF = 1, `JBE done` jumps. Because the condition is tested first, CX = 0 would never enter the loop.

Code Trace: if / else and while

```
1  MOV  AX, 12
2  MOV  BX, 30
3  CMP  AX, BX
4  JNE  not_eq       ; OPPOSITE test
5  MOV  DX, 1        ; block A: AX = BX
6  JMP  after
7  not_eq: MOV DX, 2 ; block B: AX ≠ BX
8  after: MOV CX, 2  ; while (CX > 0)
9  top: CMP CX, 0
10 JBE  done         ; CX ≤ 0 → exit
11 ADD  DX, CX       ; body
12 DEC  CX
13 JMP  top
▶  done: RET
```

VARIABLES

AX = **12** | BX = **30** | CX = **0** | DX = **5**

AX	BX	CX	DX
000Ch	001Eh	0000h	0005h

ZF	CF
1	0

block A | block B

loop body

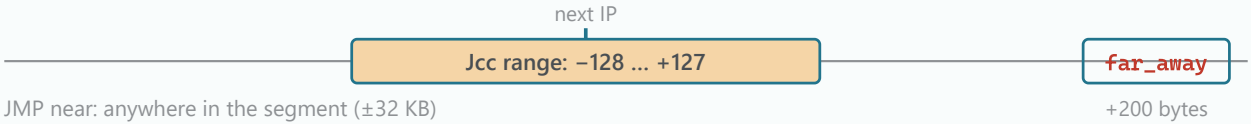
Result DX = 5: block B (2) + 2 + 1.

Step 10 / 10

COMPARISON

Out-of-Range Conditional Jump

- On the 8086 a conditional jump reaches only −128…+127 bytes.
- If the target is farther, the assembler reports an error. Fix: **invert** the condition, short-jump over the next instruction, and reach the far target with a **JMP** (near, ±32 KB).
- emu8086 does this rewrite itself, turning an out-of-range Jcc into an inverted short jump plus a JMP; assemblers such as MASM report an error.



Wrong (target too far)

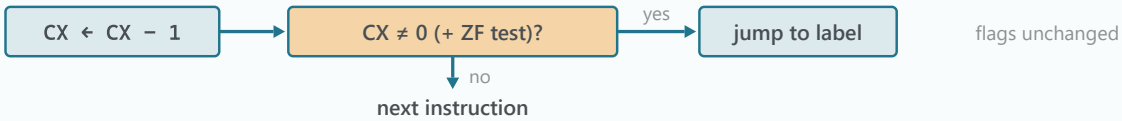
```
cmp al, 0
je  far_away    ; out of range!
; ... 200 bytes of code ...
far_away:
```

Right

```
cmp al, 0
jne cont        ; opposite test, short
jmp far_away    ; near JMP (E9)
cont:
; ... 200 bytes of code ...
far_away:
```

Loop Instructions

Instruction	Operation and jump condition
LOOP	Decrement CX; jump if CX ≠ 0.
LOOPE	Decrement CX; jump if CX ≠ 0 **and** equal (ZF = 1).
LOOPZ	Same as LOOPE: jump if CX ≠ 0 and ZF = 1.
LOOPNE	Decrement CX; jump if CX ≠ 0 **and** not equal (ZF = 0).
LOOPNZ	Same as LOOPNE: jump if CX ≠ 0 and ZF = 0.
JCXZ	Jump if CX = 0 (CX is not decremented).



Counter

The counter is always the **CX** register.

Flags

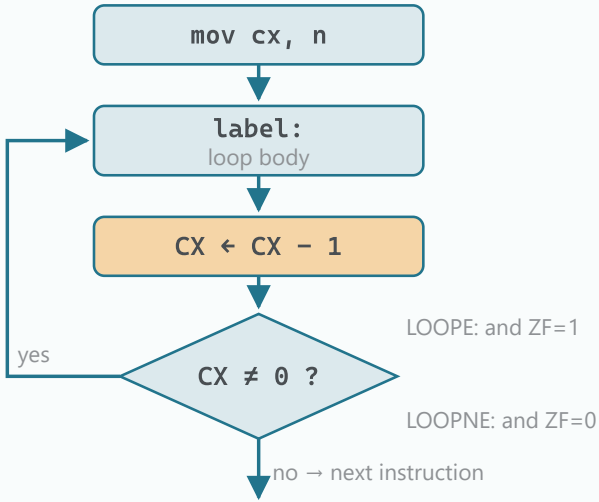
The LOOP family decrements CX **without changing** the flags.

Range

All are 2 bytes, rel8: at most 128 bytes back.

How Does LOOP Work?

- Loops **repeat** a block of code depending on a condition.
- `LOOP target` : CX is decremented first; if it is not zero, jump to the label. Same effect as `DEC CX` + `JNZ` , but the flags are preserved.
- `LOOPE / LOOPZ` jumps while CX ≠ 0 and ZF = 1; `LOOPNE / LOOPNZ` while CX ≠ 0 and ZF = 0.
- `JCXZ` : jumps to the label when CX is zero.
- With CX = 0 on entry, the first decrement makes CX FFFFh → the body runs **65,536** times.



Code Trace: LOOP

```
1  org 100h
▶  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
5  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

CX = 0005h pass = 0

CX = 0005h: the body will run 5 times.

Step 1 / 13

Code Trace: LOOP

```
1 org 100h
2 mov cx, 5          ; CX = repeat count
3 again:
▶ ; loop body
5 loop again         ; if CX ≠ 0 go to again
6 jmp quit           ; loop done: go to quit
7 quit:
8 ret
```

VARIABLES

`CX = 0005h` `pass = 1`

Pass 1: the loop body runs.

Step 2 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
▶  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0004h` `pass = 1`

`loop` : CX = 0004h ≠ 0 → jump back to `again` .

Step 3 / 13

Code Trace: LOOP

```
1 org 100h
2 mov cx, 5          ; CX = repeat count
3 again:
▶ ; loop body
5 loop again         ; if CX ≠ 0 go to again
6 jmp quit           ; loop done: go to quit
7 quit:
8 ret
```

VARIABLES

`CX = 0004h` `pass = 2`

Pass 2: the loop body runs.

Step 4 / 13

Code Trace: `LOOP`

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
▶  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0003h` `pass = 2`

`loop` : CX = 0003h ≠ 0 → jump back to `again` .

Step 5 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
▶  ; loop body
5  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0003h` `pass = 3`

Pass 3: the loop body runs.

Step 6 / 13

Code Trace: `LOOP`

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
▶  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0002h` `pass = 3`

`loop` : CX = 0002h ≠ 0 → jump back to `again` .

Step 7 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
▶  ; loop body
5  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0002h` `pass = 4`

Pass 4: the loop body runs.

Step 8 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
▶  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0001h` `pass = 4`

`loop` : CX = 0001h ≠ 0 → jump back to `again` .

Step 9 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
▶  ; loop body
5  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0001h` `pass = 5`

Pass 5: the loop body runs.

Step 10 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
▶  loop again         ; if CX ≠ 0 go to again
6  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0000h` `pass = 5`

`loop` : CX = 0000h → no jump, the loop exits.

Step 11 / 13

Code Trace: LOOP

```
1  org 100h
2  mov cx, 5          ; CX = repeat count
3  again:
4  ; loop body
5  loop again         ; if CX ≠ 0 go to again
▶  jmp quit           ; loop done: go to quit
7  quit:
8  ret
```

VARIABLES

`CX = 0000h` `pass = 5`

`jmp quit` : `quit` is already the next line, so this jump is unnecessary but harmless.

Step 12 / 13

Code Trace: `LOOP`

```
1 org 100h
2 mov cx, 5          ; CX = repeat count
3 again:
4 ; loop body
5 loop again         ; if CX ≠ 0 go to again
6 jmp quit           ; loop done: go to quit
7 quit:
▶ ret
```

VARIABLES

`CX = 0000h` `pass = 5`

`ret` : the body ran 5 times in total.

Step 13 / 13

Code Trace: Nested `LOOP`s

```
1  org 100h
▶  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h        ; print '1'
9   push cx        ; save outer
10  mov cx, 5      ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
16   loop k2       ; inner loop
17  pop cx         ; restore outer
18 loop k1         ; outer loop
19 ret
```

VARIABLES

BX = **0000h** SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

BX counts the total steps: 0. In a .COM program SP = FFFEh.

Step 1 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0        ; total steps
▶  mov cx, 5        ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h        ; print '2'
16   loop k2        ; inner loop
17  pop cx          ; restore outer
18 loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0000h** | CX = **0005h** | SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

The outer loop will run 5 times: CX = 0005h.

Step 2 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h        ; print '1'
9   push cx        ; save outer
10  mov cx, 5      ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
16   loop k2       ; inner loop
17  pop cx         ; restore outer
18 loop k1         ; outer loop
19 ret
```

VARIABLES

BX = 0001h | CX = 0005h | SP = FFFEh

CALL STACK

[FFFEh] 0000h

OUTPUT

Outer pass 1: BX = 1.

Step 3 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
▶   mov al, '1'
▶   mov ah, 0eh
▶   int 10h        ; print '1'
9   push cx        ; save outer
10  mov cx, 5      ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
16   loop k2       ; inner loop
17  pop cx         ; restore outer
18 loop k1         ; outer loop
19 ret
```

VARIABLES

BX = **0001h** CX = **0005h** AL = **31h** AH = **0Eh** SP = **FFFEh**

CALL STACK

```
[FFFEh] 0000h
```

OUTPUT

```
1
```

AL = '1' (31h), AH = 0Eh → INT 10h teletype prints **1**. (Function 0Eh uses BH as the page number; BX < 256, so BH = 0.)

Step 4 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
▶   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h        ; print '2'
16   loop k2        ; inner loop
17  pop cx          ; restore outer
18 loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0001h** | CX = **0005h** | AL = **31h** | AH = **0Eh** | SP = **FFFCh**

CALL STACK

- `[FFFCh] 0005h`
- `[FFFEh] 0000h`

OUTPUT

```
1
```

`push cx`: the inner loop also uses CX, so the outer counter (5) is saved on the stack; SP = FFFCh.

Code Trace: Nested `LOOP`s

```
 1 org 100h
 2 mov bx, 0       ; total steps
 3 mov cx, 5       ; outer counter
 4 k1:
 5  add bx, 1
 6  mov al, '1'
 7  mov ah, 0eh
 8  int 10h        ; print '1'
 9  push cx        ; save outer
▶10  mov cx, 5     ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
16   loop k2       ; inner loop
17  pop cx         ; restore outer
18 loop k1         ; outer loop
19 ret
```

VARIABLES

BX = **0001h** CX = **0005h** AL = **31h**
AH = **0Eh** SP = **FFFCh**

CALL STACK

[FFFCh] 0005h
[FFFEh] 0000h

OUTPUT

```
1
```

Inner loop counter: CX = 5.

Step 6 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
▶    add bx, 1
▶    mov al, '2'
▶    mov ah, 0eh
▶    int 10h        ; print '2'
16   loop k2        ; inner loop
17  pop cx          ; restore outer
18 loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0002h** CX = **0005h** AL = **32h** AH = **0Eh** SP = **FFFCh**

CALL STACK

```
[FFFCh] 0005h
[FFFEh] 0000h
```

OUTPUT

```
12
```

Inner pass 1: BX = 2, prints **2**.

Step 7 / 16

Code Trace: Nested `LOOP`s

```
 1 org 100h
 2 mov bx, 0       ; total steps
 3 mov cx, 5       ; outer counter
 4 k1:
 5  add bx, 1
 6  mov al, '1'
 7  mov ah, 0eh
 8  int 10h        ; print '1'
 9  push cx        ; save outer
10  mov cx, 5      ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
▶    loop k2       ; inner loop
17  pop cx         ; restore outer
18 loop k1         ; outer loop
19 ret
```

VARIABLES

BX = **0002h** | CX = **0004h** | AL = **32h** | AH = **0Eh** | SP = **FFFCh**

CALL STACK

- **[FFFCh] 0005h**
- [FFFEh] 0000h

OUTPUT

```
12
```

`loop k2` : CX = 4 → back to k2.

Step 8 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h         ; print '2'
▶    loop k2         ; inner loop
17  pop cx          ; restore outer
18 loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0006h** CX = **0000h** AL = **32h** AH = **0Eh** SP = **FFFCh**

CALL STACK

[FFFCh] 0005h

[FFFEh] 0000h

OUTPUT

```
122222
```

4 more inner passes: 5 **2**s printed in total, BX = 6, CX = 0 → the inner loop ends.

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h         ; print '2'
16   loop k2         ; inner loop
▶   pop cx          ; restore outer
18 loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0006h** CX = **0005h** AL = **32h**
AH = **0Eh** SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

```
122222
```

`pop cx` : the outer counter is back: CX = 5, SP = FFFEh.

Step 10 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h         ; print '2'
16   loop k2         ; inner loop
17  pop cx          ; restore outer
▶ loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0006h** | CX = **0004h** | AL = **32h** | AH = **0Eh** | SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

```
122222
```

`loop k1` : CX = 4 → back to k1 (outer pass 2).

Step 11 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h         ; print '2'
16   loop k2         ; inner loop
17  pop cx          ; restore outer
▶ loop k1          ; outer loop
19 ret
```

VARIABLES

BX = 000Ch CX = 0003h AL = 32h AH = 0Eh SP = FFFEh

CALL STACK

[FFFEh] 0000h

OUTPUT

```
122222122222
```

Outer pass 2 done: BX = 12, CX = 3.

Step 12 / 16

Code Trace: Nested `LOOP`s

```
 1 org 100h
 2 mov bx, 0       ; total steps
 3 mov cx, 5       ; outer counter
 4 k1:
 5  add bx, 1
 6  mov al, '1'
 7  mov ah, 0eh
 8  int 10h        ; print '1'
 9  push cx        ; save outer
10  mov cx, 5      ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
16   loop k2       ; inner loop
17  pop cx         ; restore outer
▶ loop k1          ; outer loop
19 ret
```

VARIABLES

BX = 0012h CX = 0002h AL = 32h AH = 0Eh SP = FFFEh

CALL STACK

[FFFEh] 0000h

OUTPUT

```
122222122222122222
```

Outer pass 3 done: BX = 18, CX = 2.

Step 13 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h         ; print '2'
16   loop k2         ; inner loop
17  pop cx          ; restore outer
▶ loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **0018h** CX = **0001h** AL = **32h**
AH = **0Eh** SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

```
122222122222122222122222
```

Outer pass 4 done: BX = 24, CX = 1.

Step 14 / 16

Code Trace: Nested `LOOP`s

```
 1 org 100h
 2 mov bx, 0       ; total steps
 3 mov cx, 5       ; outer counter
 4 k1:
 5  add bx, 1
 6  mov al, '1'
 7  mov ah, 0eh
 8  int 10h        ; print '1'
 9  push cx        ; save outer
10  mov cx, 5      ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h       ; print '2'
16   loop k2       ; inner loop
17  pop cx         ; restore outer
▶ loop k1          ; outer loop
19 ret
```

VARIABLES

BX = **001Eh** CX = **0000h** AL = **32h** AH = **0Eh** SP = **FFFEh**

CALL STACK

`[FFFEh] 0000h`

OUTPUT

```
1222221222221222221222222122222
```

Outer pass 5 done: CX = 0 → the outer loop exits. BX = 30 = **001Eh** (5 + 5×5 steps).

Step 15 / 16

Code Trace: Nested `LOOP`s

```
1  org 100h
2  mov bx, 0       ; total steps
3  mov cx, 5       ; outer counter
4  k1:
5   add bx, 1
6   mov al, '1'
7   mov ah, 0eh
8   int 10h         ; print '1'
9   push cx         ; save outer
10  mov cx, 5       ; inner counter
11 k2:
12   add bx, 1
13   mov al, '2'
14   mov ah, 0eh
15   int 10h         ; print '2'
16   loop k2         ; inner loop
17  pop cx          ; restore outer
18 loop k1          ; outer loop
▶  ret
```

VARIABLES

BX = 001Eh | CX = 0000h | AL = 32h | AH = 0Eh | SP = 0000h

CALL STACK

empty

OUTPUT

```
12222212222212222212222212222
```

`ret`. The screen shows "122222" 5 times. Because push/pop are balanced, ret returns to the right address (0000h); SP = 0000h.

Step 16 / 16

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
▶ 2  mov cx, 5          ; element count
3  mov si, -1           ; first inc makes it 0
4  scan:
5  inc si               ; next element
6  cmp arr[si], 3       ; is it 3?
7  loopne scan          ; go on if CX≠0, ZF=0
8  jne notfound         ; ZF=0 → not found
9  mov al, 1            ; found, position in SI
10 ret
11 notfound:
12 mov al, 0            ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

CX = **0005h**



At most 5 elements will be examined.

Step 1 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5            ; element count
▶  mov si, -1           ; first inc makes it 0
4  scan:
5  inc si               ; next element
6  cmp arr[si], 3       ; is it 3?
7  loopne scan          ; go on if CX≠0, ZF=0
8  jne notfound         ; ZF=0 → not found
9  mov al, 1            ; found, position in SI
10 ret
11 notfound:
12 mov al, 0            ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0005h` `SI = FFFFh`



SI = −1 (FFFFh); the first `inc` in the loop makes it 0.

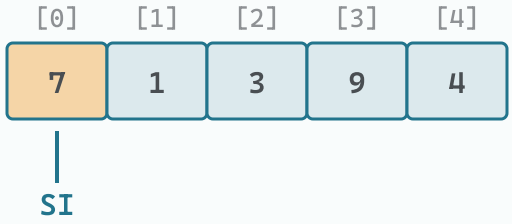
Step 2 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5            ; element count
3  mov si, -1           ; first inc makes it 0
4  scan:
▶  inc si               ; next element
6  cmp arr[si], 3       ; is it 3?
7  loopne scan          ; go on if CX≠0, ZF=0
8  jne notfound         ; ZF=0 → not found
9  mov al, 1            ; found, position in SI
10 ret
11 notfound:
12 mov al, 0            ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0005h` `SI = 0000h`



SI = 0.

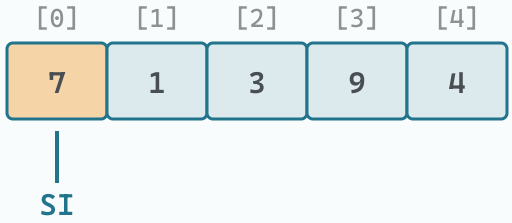
Step 3 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
▶  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

CX = **0005h** | SI = **0000h** | ZF = **0**



arr[0] = 7; 7 − 3 → ZF = 0.

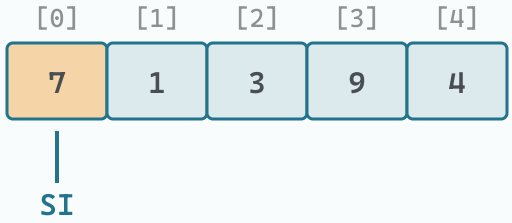
Step 4 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
▶  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0004h` `SI = 0000h` `ZF = 0`



`loopne` : CX = 4 ≠ 0 and ZF = 0 → keep searching.

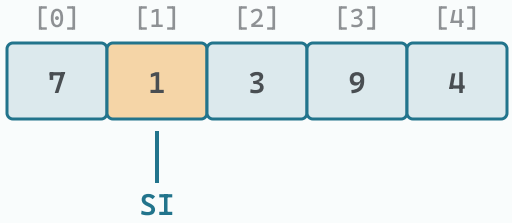
Step 5 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
▶  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0004h` `SI = 0001h` `ZF = 0`



SI = 1.

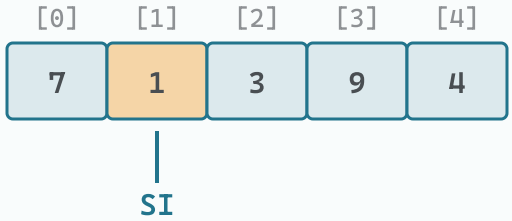
Step 6 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
▶  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0004h` `SI = 0001h` `ZF = 0`



arr[1] = 1; 1 − 3 → ZF = 0.

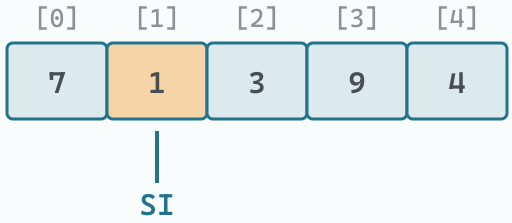
Step 7 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
▶  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0003h` `SI = 0001h` `ZF = 0`



`loopne` : CX = 3 ≠ 0 and ZF = 0 → keep searching.

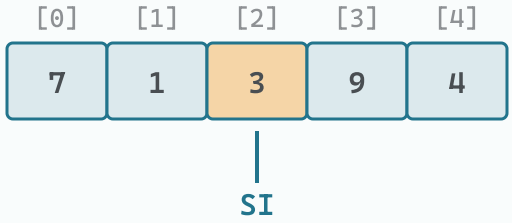
Step 8 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0003h` `SI = 0002h` `ZF = 0`



SI = 2.

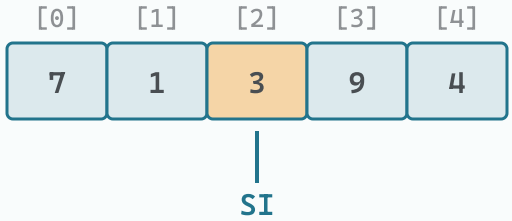
Step 9 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
▶  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

CX = **0003h** | SI = **0002h** | ZF = **1**



arr[2] = 3; 3 − 3 → ZF = 1.

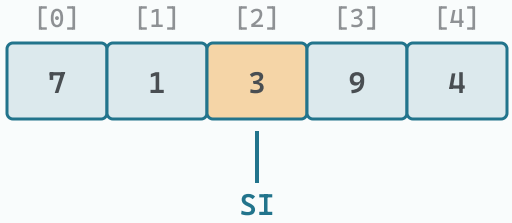
Step 10 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
▶  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0002h` `SI = 0002h` `ZF = 1`



`loopne` : CX = 2 but **ZF = 1** → the loop exits. LOOPNE does not change the flags, so ZF = 1 survives.

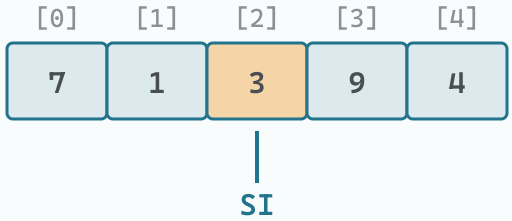
Step 11 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
▶  jne notfound       ; ZF=0 → not found
9  mov al, 1          ; found, position in SI
10 ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0002h` `SI = 0002h` `ZF = 1`



ZF = 1 → `jne notfound` does not jump: the value was found.

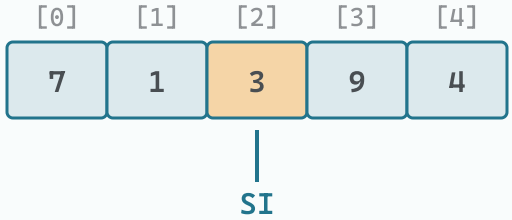
Step 12 / 13

Code Trace: Searching an Array with `LOOPNE`

```
1  org 100h
2  mov cx, 5          ; element count
3  mov si, -1         ; first inc makes it 0
4  scan:
5  inc si             ; next element
6  cmp arr[si], 3     ; is it 3?
7  loopne scan        ; go on if CX≠0, ZF=0
8  jne notfound       ; ZF=0 → not found
▶  mov al, 1          ; found, position in SI
▶  ret
11 notfound:
12 mov al, 0          ; not found
13 ret
14 arr db 7, 1, 3, 9, 4
```

VARIABLES

`CX = 0002h` `SI = 0002h` `ZF = 1` `AL = 01h`



AL = 1 (found), SI = 2 gives the position. The loop finished in 3 passes instead of 5.

Step 13 / 13

JCXZ and LOOP / DEC–JNZ

Guarding against a zero-pass loop

```
mov cx, n          ; n known at run time
jcxz skip          ; skip if CX = 0
again:
    ; body
    loop again
skip:
```

Without JCXZ, LOOP with CX = 0 first makes CX FFFFh and the body runs 65,536 times.

LOOP or DEC + JNZ?

	LOOP	DEC CX + JNZ
Size	2 bytes	1 + 2 = 3 bytes
Flags	unchanged	ZF, SF, OF, PF, AF change (CF kept)
Counter	CX only	any register
Extra condition	LOOPE / LOOPNE	any Jcc can be chosen

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
8  again2: INC AX
9  DEC  BX           ; CF is kept
10 JNZ  again2
11 RET
```

VARIABLES

AX = 0 | BX = 0 | CX = 0 | CF = 0

AX	BX	CX
0000h	0000h	0000h

ZF	CF
0	0

The pass count n = 0 is in CX. An unguarded LOOP would first make CX FFFFh and run the body 65,536 times.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
▶  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
8  again2: INC AX
9  DEC  BX           ; CF is kept
10 JNZ  again2
11 RET
```

VARIABLES

`AX = 0` `BX = 0` `CX = 0` `CF = 0`

AX	BX	CX
0000h	0000h	0000h

ZF	CF
0	0

JCXZ → jumps

`JCXZ skip`: CX = 0 → the loop is never entered. AX stays 0.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
▶  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
8  again2: INC AX
9  DEC  BX           ; CF is kept
10 JNZ  again2
11 RET
```

VARIABLES

AX = 0 | BX = 3 | CX = 0 | CF = 0

AX	BX	CX
0000h	0003h	0000h

ZF	CF
0	0

Second loop: counter BX = 3 (DEC–JNZ can use any register).

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
▶  STC               ; CF = 1
8  again2: INC AX
9  DEC  BX           ; CF is kept
10 JNZ  again2
11 RET
```

VARIABLES

`AX = 0` `BX = 3` `CX = 0` `CF = 1`

AX	BX	CX
0000h	0003h	0000h

ZF	CF
0	1

`STC` : CF = 1. Think of it as information that must survive the loop.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
▶  again2: INC AX
▶  DEC  BX           ; CF is kept
▶  JNZ  again2
11 RET
```

VARIABLES

AX = 1 | BX = 2 | CX = 0 | CF = 1

AX	BX	CX
0001h	0002h	0000h

ZF	CF
0	1

JNZ → jumps

Pass 1: AX = 1. `DEC BX` → 2, ZF = 0 → `JNZ` jumps back. CF is still 1.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
▶  again2: INC AX
▶  DEC  BX           ; CF is kept
▶  JNZ  again2
11 RET
```

VARIABLES

AX = 2 | BX = 1 | CX = 0 | CF = 1

AX	BX	CX
0002h	0001h	0000h

ZF	CF
0	1

JNZ → jumps

Pass 2: AX = 2, BX = 1.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
▶  again2: INC AX
▶  DEC  BX           ; CF is kept
10 JNZ  again2
11 RET
```

VARIABLES

AX = 3 | BX = 0 | CX = 0 | CF = 1

AX	BX	CX
0003h	0000h	0000h

ZF	CF
1	1

Pass 3: AX = 3. `DEC BX` → 0, ZF = 1.
INC/DEC did not touch CF: CF = 1.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
8  again2: INC AX
9  DEC  BX           ; CF is kept
▶  JNZ  again2
11 RET
```

VARIABLES

`AX = 3` `BX = 0` `CX = 0` `CF = 1`

AX	BX	CX
0003h	0000h	0000h

ZF	CF
1	1

JNZ → no jump

`JNZ` : ZF = 1 → no jump, the loop ends.

Code Trace: JCXZ Guard and DEC–JNZ

```
1  MOV  CX, 0        ; n = 0
2  MOV  AX, 0
3  JCXZ skip         ; CX = 0 → skip
4  again: INC AX
5  LOOP again
6  skip: MOV BX, 3   ; n = 3, counter BX
7  STC               ; CF = 1
8  again2: INC AX
9  DEC  BX           ; CF is kept
10 JNZ  again2
▶  RET
```

VARIABLES

AX = 3 | BX = 0 | CX = 0 | CF = 1

AX	BX	CX
0003h	0000h	0000h

ZF	CF
1	1

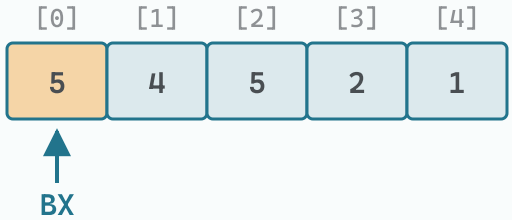
Result AX = 3, CF = 1 preserved. LOOP does not change flags either, but works only with CX.

Code Trace: Sum of Array Elements

```
1  org 100h
▶  mov cx, 5            ; element count
▶  mov al, 0            ; sum (AL) = 0
▶  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = 0005h | BX = 0000h | AL = 00h | m = 00h



CX = 5 elements, sum AL = 0, index BX = 0.

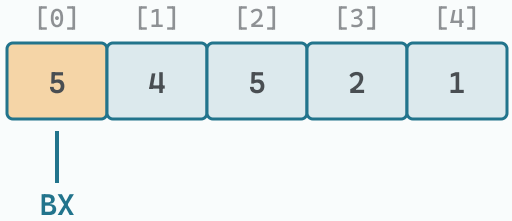
Step 1 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
▶  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0005h** | BX = **0000h** | AL = **05h** | m = **00h**



`add al, vector[0]` : AL = 0 + 5 = 5 (05h).

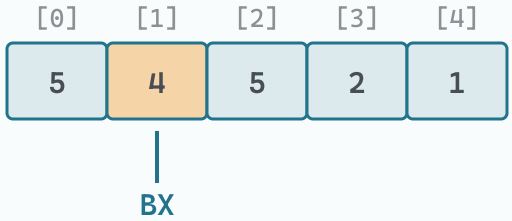
Step 2 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
▶  inc bx               ; next element
▶  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = 0004h | BX = 0001h | AL = 05h | m = 00h



BX = 1, CX = 4 → back to next.

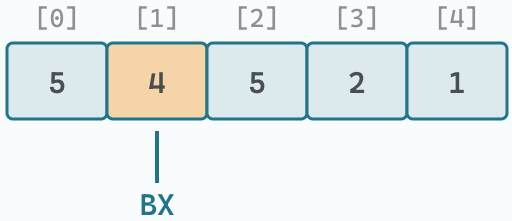
Step 3 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0004h** | BX = **0001h** | AL = **09h** | m = **00h**



`add al, vector[1]` : AL = 5 + 4 = 9 (09h).

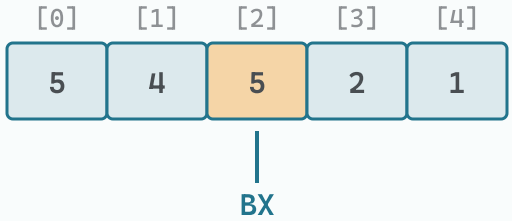
Step 4 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
   inc bx               ; next element
   loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0003h** | BX = **0002h** | AL = **09h** | m = **00h**



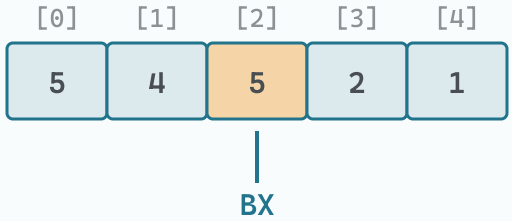
BX = 2, CX = 3 → back to next.

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
▶  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0003h** | BX = **0002h** | AL = **0Eh** | m = **00h**



`add al, vector[2]` : AL = 9 + 5 = 14 (0Eh).

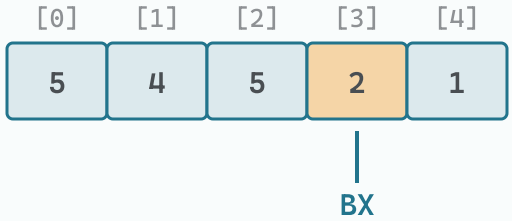
Step 6 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
   inc bx               ; next element
   loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = 0002h | BX = 0003h | AL = 0Eh | m = 00h



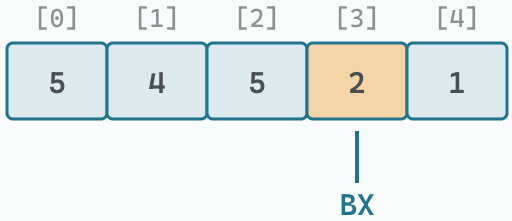
BX = 3, CX = 2 → back to next.

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
▶  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

`CX = 0002h` `BX = 0003h` `AL = 10h` `m = 00h`



`add al, vector[3]` : AL = 14 + 2 = 16 (10h).

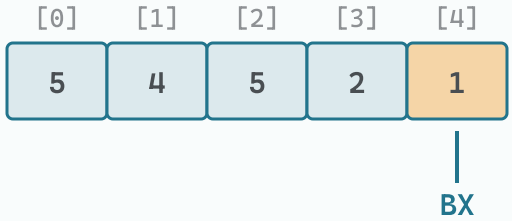
Step 8 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
   inc bx               ; next element
   loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = 0001h | BX = 0004h | AL = 10h | m = 00h



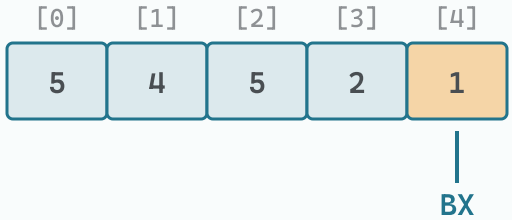
BX = 4, CX = 1 → back to next.

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
▶  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0001h** | BX = **0004h** | AL = **11h** | m = **00h**



`add al, vector[4]` : AL = 16 + 1 = 17 (11h).

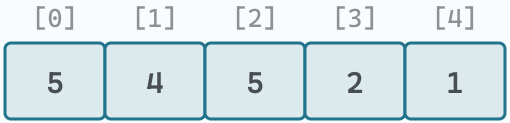
Step 10 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
   inc bx               ; next element
   loop next            ; until CX = 0
9  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0000h** BX = **0005h** AL = **11h** m = **00h**



BX = 5, CX = 0 → the loop ends.

Step 11 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
▶  mov m, al            ; store in m
10 ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0000h** | BX = **0005h** | AL = **11h** | m = **11h**



`mov m, al` : m = 11h = **17** = 5 + 4 + 5 + 2 + 1.

Step 12 / 13

Code Trace: Sum of Array Elements

```
1  org 100h
2  mov cx, 5            ; element count
3  mov al, 0            ; sum (AL) = 0
4  mov bx, 0            ; BX = index
5  next:
6  add al, vector[bx]   ; add element
7  inc bx               ; next element
8  loop next            ; until CX = 0
9  mov m, al            ; store in m
▶  ret
11 vector db 5, 4, 5, 2, 1
12 m db 0
```

VARIABLES

CX = **0000h** BX = **0005h** AL = **11h**

m = **11h**

[0]	[1]	[2]	[3]	[4]
5	4	5	2	1

`ret`. The data is defined after `ret`, so it is never executed as code.

Code Trace: Printing Hello on the Screen

```
1  org     100h
▶  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

—

OUTPUT

48h	65h	6Ch	6Ch	6Fh	00h
'H'	'e'	'l'	'l'	'o'	0

`jmp start`: the data (msg) sits between instructions, so it is jumped over; otherwise bytes such as 'H' (48h) would be taken as instructions.

Step 1 / 13

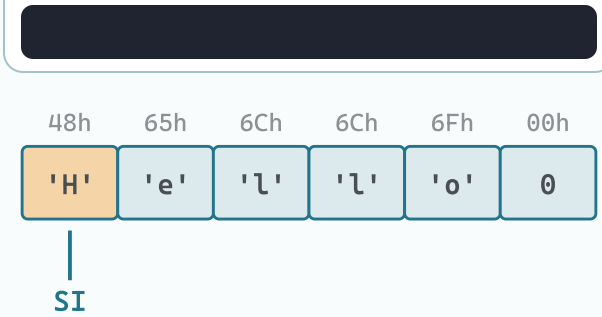
Code Trace: Printing Hello on the Screen

```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
▶  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0000h

OUTPUT



SI = 0: first character.

Step 2 / 13

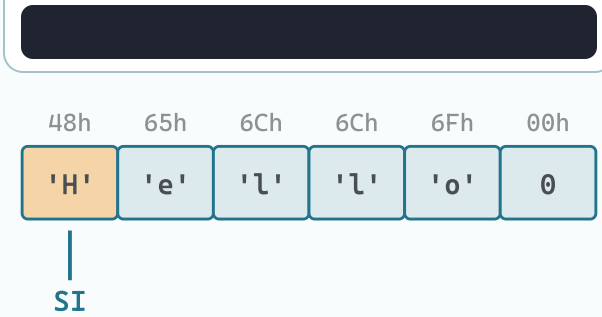
Code Trace: Printing Hello on the Screen

```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = **0000h** AL = **48h**

OUTPUT



AL = msg[0] = 'H' = 48h.

Step 3 / 13

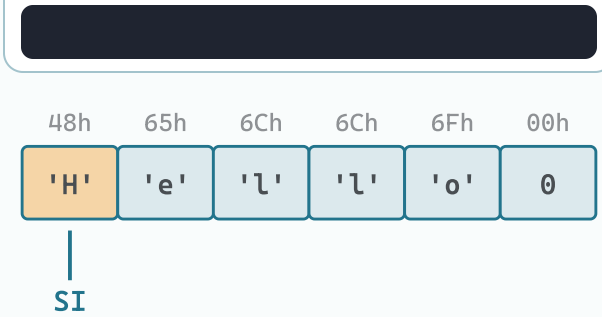
Code Trace: Printing Hello on the Screen

```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
▶  cmp     al, 0           ; compare with 0
▶  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0000h | AL = 48h | ZF = 0

OUTPUT



48h − 0 ≠ 0 → ZF = 0, `je` does not jump.

Step 4 / 13

Code Trace: Printing Hello on the Screen

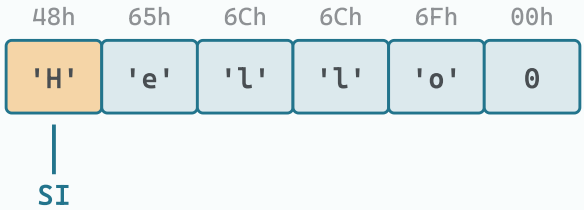
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
▶  mov     ah, 0eh         ; teletype function
▶  int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = **0000h** | AL = **48h** | AH = **0Eh** | ZF = **0**

OUTPUT

```
H
```



AH = 0Eh, INT 10h → prints **H**.

Step 5 / 13

Code Trace: Printing Hello on the Screen

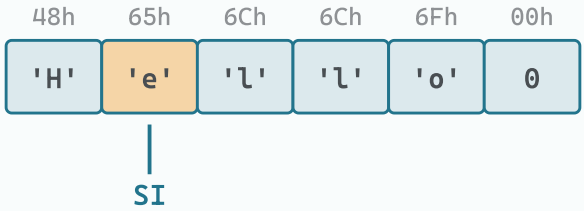
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
▶ inc     si              ; next character
▶ jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0001h | AL = 48h | AH = 0Eh | ZF = 0

OUTPUT

```
H
```



SI = 1, `jmp next_char` goes back to the top.

Code Trace: Printing Hello on the Screen

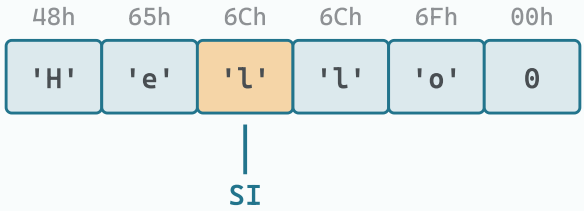
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
   mov     al, msg[si]     ; get the char
   cmp     al, 0           ; compare with 0
   je      stop            ; stop if 0
   mov     ah, 0eh         ; teletype function
   int     10h             ; print char in AL
   inc     si              ; next character
   jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0002h AL = 65h AH = 0Eh ZF = 0

OUTPUT

```
He
```



Next pass: AL = 'e' (65h), ZF = 0 → prints **e**, SI = 2.

Code Trace: Printing Hello on the Screen

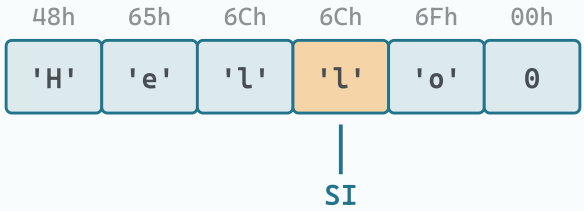
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
   mov     al, msg[si]     ; get the char
   cmp     al, 0           ; compare with 0
   je      stop            ; stop if 0
   mov     ah, 0eh         ; teletype function
   int     10h             ; print char in AL
   inc     si              ; next character
   jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0003h | AL = 6Ch | AH = 0Eh | ZF = 0

OUTPUT

```
Hel
```



Next pass: AL = 'l' (6Ch), ZF = 0 → prints **l**, SI = 3.

Code Trace: Printing Hello on the Screen

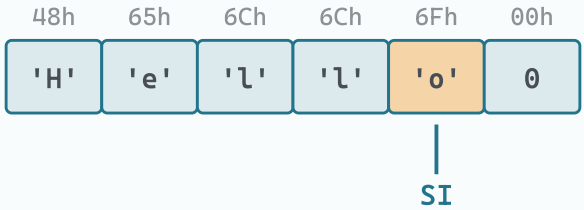
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
   mov     al, msg[si]     ; get the char
   cmp     al, 0           ; compare with 0
   je      stop            ; stop if 0
   mov     ah, 0eh         ; teletype function
   int     10h             ; print char in AL
   inc     si              ; next character
   jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0004h | AL = 6Ch | AH = 0Eh | ZF = 0

OUTPUT

```
Hell
```



Next pass: AL = 'l' (6Ch), ZF = 0 → prints **l**, SI = 4.

Code Trace: Printing Hello on the Screen

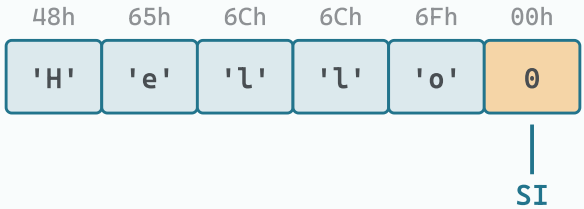
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
   mov     al, msg[si]     ; get the char
   cmp     al, 0           ; compare with 0
   je      stop            ; stop if 0
   mov     ah, 0eh         ; teletype function
   int     10h             ; print char in AL
   inc     si              ; next character
   jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = 0005h AL = 6Fh AH = 0Eh ZF = 0

OUTPUT

```
Hello
```



Next pass: AL = 'o' (6Fh), ZF = 0 → prints **o**, SI = 5.

Code Trace: Printing Hello on the Screen

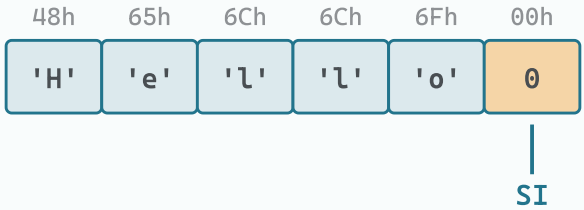
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
▶  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = **0005h** AL = **00h** AH = **0Eh** ZF = **0**

OUTPUT

```
Hello
```



AL = msg[5] = 0: end of the string (null terminator).

Code Trace: Printing Hello on the Screen

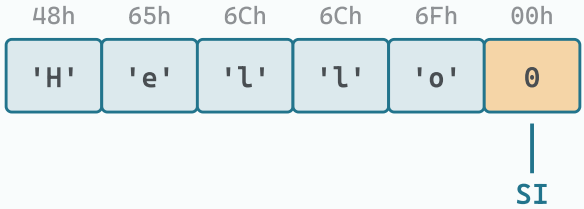
```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
▶  cmp     al, 0           ; compare with 0
▶  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
15 ret
```

VARIABLES

SI = **0005h** AL = **00h** AH = **0Eh** ZF = **1**

OUTPUT

```
Hello
```



0 − 0 = 0 → ZF = 1, `je stop` jumps.

Code Trace: Printing Hello on the Screen

```
1  org     100h
2  jmp     start
3  msg db 'Hello', 0
4  start:
5  mov     si, 0           ; source index = 0
6  next_char:
7  mov     al, msg[si]     ; get the char
8  cmp     al, 0           ; compare with 0
9  je      stop            ; stop if 0
10 mov     ah, 0eh         ; teletype function
11 int     10h             ; print char in AL
12 inc     si              ; next character
13 jmp     next_char       ; loop back
14 stop:
▶  ret
```

VARIABLES

`SI = 0005h` `AL = 00h` `AH = 0Eh` `ZF = 1`

OUTPUT

```
Hello
```

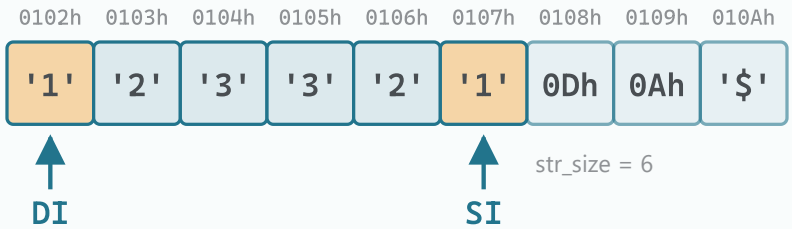
48h	65h	6Ch	6Ch	6Fh	00h
'H'	'e'	'l'	'l'	'o'	0

`ret` : the screen shows **Hello**.

Step 13 / 13

Palindrome Check: The Idea

- Palindrome: a string that reads the same forward and backward (`123321` , `kayak`).
- **DI** points to the start of the string and **SI** to its end; the characters are compared, DI moves forward and SI backward.
- Comparing **half** the length is enough (`shr cx, 1` → CX / 2). With an odd length the middle character is not compared.
- `str_size = $ - msg` : `$` is the current address; `msg` and `str` are at the same address, so the result is the constant **6**, the string length (it takes no memory).
- INT 21h / AH = 09h prints the string at DS:DX up to the `'$'` character; 0Dh, 0Ah are carriage return and line feed.



Addresses assume `org 100h` and a 2-byte short `jmp start` (0100h): str starts at 0102h.

Code Trace: Palindrome (1/2) Setup

```
1  org 100h
▶  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

—

OUTPUT

0102h	0103h	0104h	0105h	0106h	0107h	0108h	0109h	010Ah
'1'	'2'	'3'	'3'	'2'	'1'	0Dh	0Ah	'$'

str_size = 6

`jmp start` : jumps over the data section.

Step 1 / 11

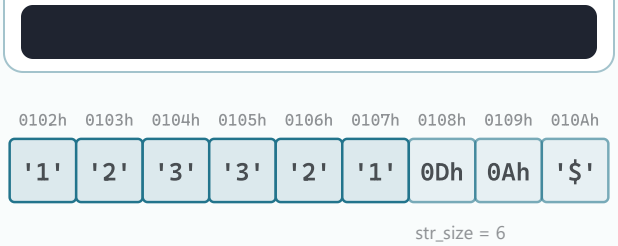
Code Trace: Palindrome (1/2) Setup

```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
▶  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = 09h

OUTPUT



AH = 09h: DOS string output function.

Step 2 / 11

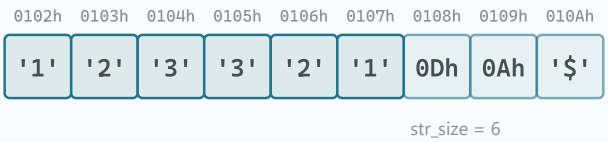
Code Trace: Palindrome (1/2) Setup

```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
▶  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h**

OUTPUT



DX = offset str = 0102h.

Step 3 / 11

Code Trace: Palindrome (1/2) Setup

```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
▶  int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h**

OUTPUT

```
123321
```



INT 21h: prints **123321** and a line break ('$' is not printed).

Step 4 / 11

Code Trace: Palindrome (1/2) Setup

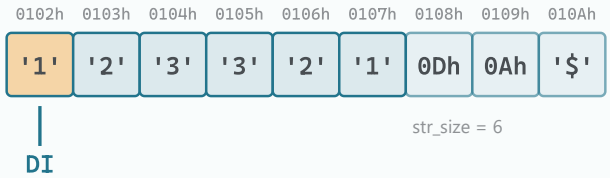
```
 1 org 100h
 2 jmp start
 3 msg:
 4 str db '123321'
 5 str_size = $ - msg      ; = 6
 6 db 0Dh,0Ah,'$'
 7 start:
 8 mov ah, 9               ; print up to '$'
 9 mov dx, offset str
10 int 21h
▶ lea di, str              ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h** DI = **0102h**

OUTPUT

```
123321
```



`lea di, str` : DI = 0102h (first character).

Step 5 / 11

Code Trace: Palindrome (1/2) Setup

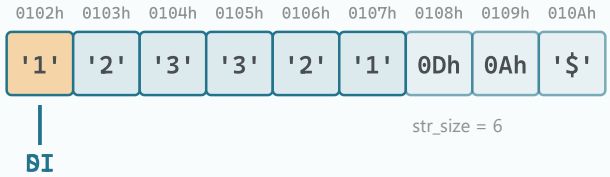
```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
▶ mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h** DI = **0102h** SI = **0102h**

OUTPUT

```
123321
```



SI = DI = 0102h.

Step 6 / 11

Code Trace: Palindrome (1/2) Setup

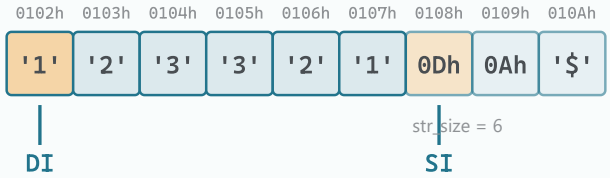
```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
▶ add si, str_size         ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h** DI = **0102h** SI = **0108h**

OUTPUT

```
123321
```



SI = 0102h + 6 = 0108h: one past the string (0Dh).

Step 7 / 11

Code Trace: Palindrome (1/2) Setup

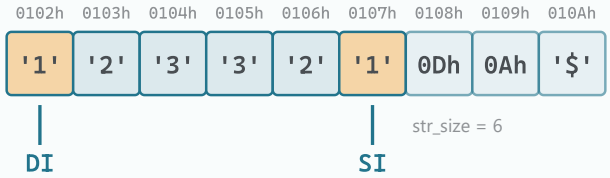
```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
▶ dec si                   ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h** DI = **0102h** SI = **0107h**

OUTPUT

```
123321
```



`dec si` : SI = 0107h, the last character '1'.

Step 8 / 11

Code Trace: Palindrome (1/2) Setup

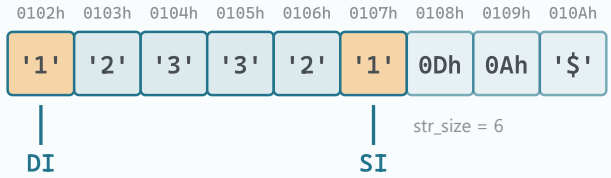
```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
▶ mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** DX = **0102h** DI = **0102h**
SI = **0107h** CX = **0006h**

OUTPUT

```
123321
```



CX = str_size = 6.

Step 9 / 11

Code Trace: Palindrome (1/2) Setup

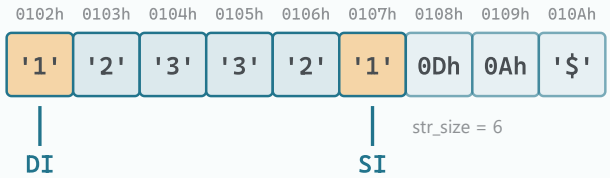
```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
▶  cmp cx, 1               ; one character?
▶  je is_palindrome        ; one char is a palindrome
18 shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** | DX = **0102h** | DI = **0102h** | SI = **0107h** | CX = **0006h** | ZF = **0** | CF = **0**

OUTPUT

```
123321
```



6 − 1 ≠ 0 → ZF = 0, `je` does not jump.

Code Trace: Palindrome (1/2) Setup

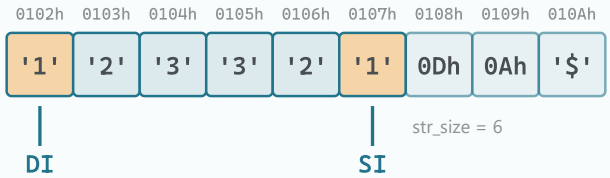
```
1  org 100h
2  jmp start
3  msg:
4  str db '123321'
5  str_size = $ - msg      ; = 6
6  db 0Dh,0Ah,'$'
7  start:
8  mov ah, 9               ; print up to '$'
9  mov dx, offset str
10 int 21h
11 lea di, str             ; DI = address of str
12 mov si, di
13 add si, str_size        ; SI = last + 1
14 dec si                  ; SI = last char
15 mov cx, str_size
16 cmp cx, 1               ; one character?
17 je is_palindrome        ; one char is a palindrome
▶  shr cx, 1               ; CX = length / 2
```

VARIABLES

AH = **09h** | DX = **0102h** | DI = **0102h** | SI = **0107h** | CX = **0003h** | ZF = **0** | CF = **0**

OUTPUT

```
123321
```



`shr cx, 1` : 0110b → 0011b, CX = 3. The bit shifted out is 0 → CF = 0.

Step 11 / 11

Code Trace: Palindrome (2/2) Comparison

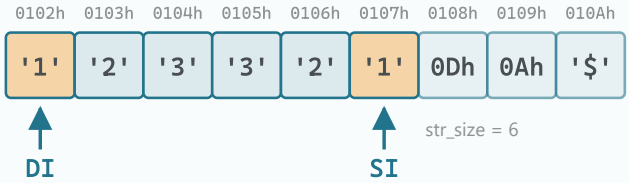
```
next_char:
mov al, [di]          ; char at the start
mov bl, [si]          ; char at the end
cmp al, bl
jne not_palindrome    ; differ: not one
inc di                ; start: one forward
dec si                ; end: one back
loop next_char
is_palindrome:
mov ah, 9             ; print "palindrome!"
mov dx, offset msg1
int 21h
jmp stop
not_palindrome:
mov ah, 9             ; "not palindrome!"
mov dx, offset msg2
int 21h
stop:
ret
msg1 db "palindrome!$"
msg2 db "not palindrome!$"
```

VARIABLES

DI = **0102h** SI = **0107h** CX = **0003h**

OUTPUT

```
123321
```



Loop start: DI = 0102h, SI = 0107h, CX = 3.

Step 1 / 14

Code Trace: Palindrome (2/2) Comparison

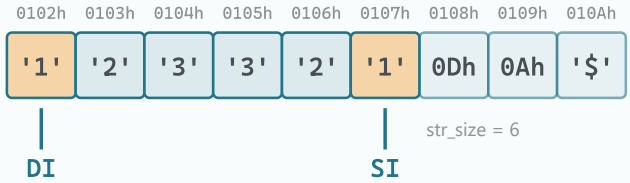
```
1  next_char:
▶  mov al, [di]          ; char at the start
▶  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 31h | BL = 31h | DI = 0102h | SI = 0107h | CX = 0003h

OUTPUT

```
123321
```



AL = [DI] = '1', BL = [SI] = '1'.

Step 2 / 14

Code Trace: Palindrome (2/2) Comparison

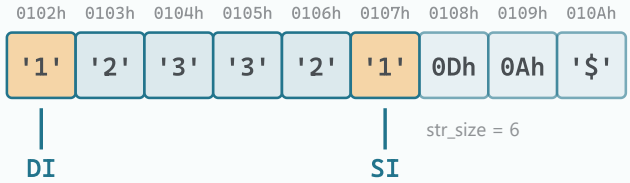
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
▶  cmp al, bl
▶  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = **31h** | BL = **31h** | DI = **0102h** | SI = **0107h** | CX = **0003h** | ZF = **1**

OUTPUT

```
123321
```



Equal → ZF = 1, `jne` does not jump.

Step 3 / 14

Code Trace: Palindrome (2/2) Comparison

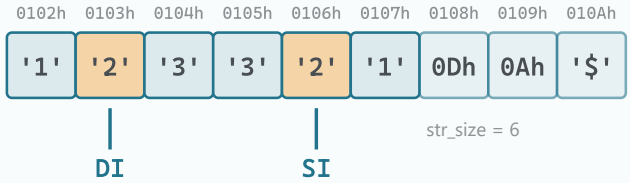
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
▶  inc di                ; start: one forward
▶  dec si                ; end: one back
▶  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 31h | BL = 31h | DI = 0103h | SI = 0106h | CX = 0002h | ZF = 0

OUTPUT

```
123321
```



DI = 0103h, SI = 0106h (the INC/DEC result is nonzero → ZF = 0), CX = 2 → next_char.

Step 4 / 14

Code Trace: Palindrome (2/2) Comparison

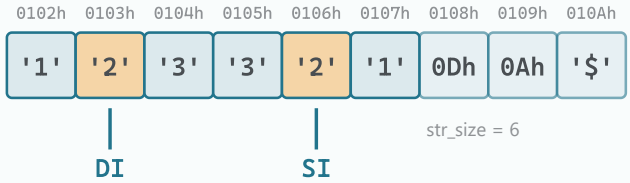
```
1  next_char:
▶  mov al, [di]          ; char at the start
▶  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 32h | BL = 32h | DI = 0103h | SI = 0106h | CX = 0002h | ZF = 0

OUTPUT

```
123321
```



AL = [DI] = '2', BL = [SI] = '2'.

Step 5 / 14

Code Trace: Palindrome (2/2) Comparison

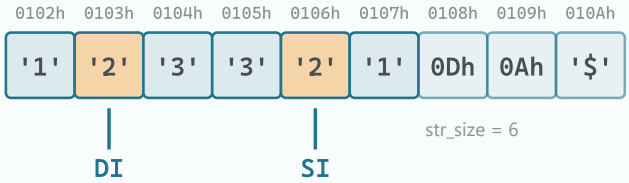
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
▶  cmp al, bl
▶  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 32h | BL = 32h | DI = 0103h | SI = 0106h | CX = 0002h | ZF = 1

OUTPUT

```
123321
```



Equal → ZF = 1, `jne` does not jump.

Step 6 / 14

Code Trace: Palindrome (2/2) Comparison

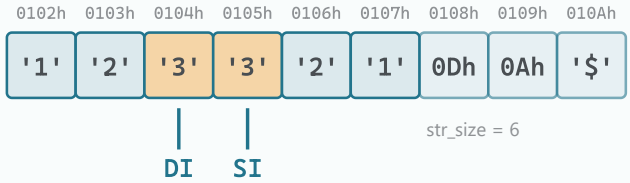
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
▶  inc di                ; start: one forward
▶  dec si                ; end: one back
▶  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 32h | BL = 32h | DI = 0104h | SI = 0105h | CX = 0001h | ZF = 0

OUTPUT

```
123321
```



DI = 0104h, SI = 0105h, CX = 1 → next_char.

Step 7 / 14

Code Trace: Palindrome (2/2) Comparison

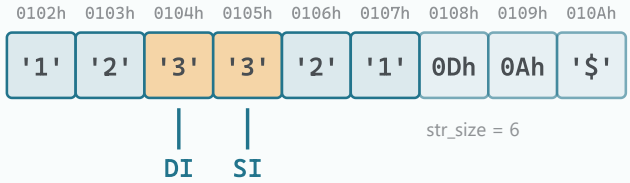
```
1  next_char:
▶  mov al, [di]          ; char at the start
▶  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 33h | BL = 33h | DI = 0104h | SI = 0105h | CX = 0001h | ZF = 0

OUTPUT

```
123321
```



AL = [DI] = '3', BL = [SI] = '3'.

Step 8 / 14

Code Trace: Palindrome (2/2) Comparison

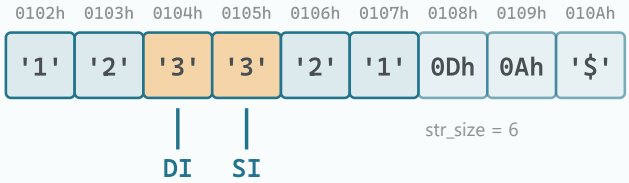
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
▶  cmp al, bl
▶  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 33h | BL = 33h | DI = 0104h | SI = 0105h | CX = 0001h | ZF = 1

OUTPUT

```
123321
```



Equal → ZF = 1, `jne` does not jump.

Step 9 / 14

Code Trace: Palindrome (2/2) Comparison

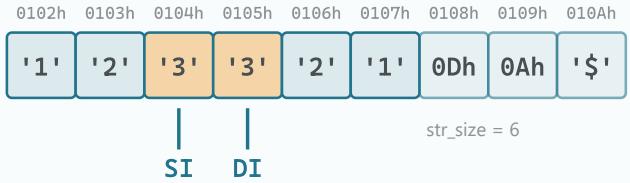
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
▶  inc di                ; start: one forward
▶  dec si                ; end: one back
▶  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 33h | BL = 33h | DI = 0105h | SI = 0104h | CX = 0000h | ZF = 0

OUTPUT

```
123321
```



CX = 0 → the loop ends: every pair matched.

Step 10 / 14

Code Trace: Palindrome (2/2) Comparison

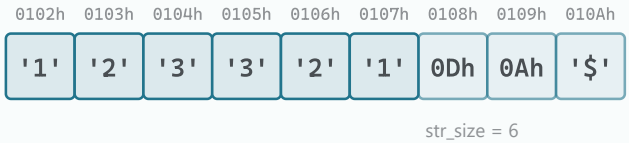
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
▶  mov ah, 9             ; print "palindrome!"
▶  mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 33h | BL = 33h | DI = 0105h | SI = 0104h | CX = 0000h | ZF = 0 | AH = 09h | DX = 0142h

OUTPUT

```
123321
```



AH = 09h, DX = offset msg1 = 0142h.

Step 11 / 14

Code Trace: Palindrome (2/2) Comparison

```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
▶ int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = 24h | BL = 33h | DI = 0105h | SI = 0104h | CX = 0000h | ZF = 0 | AH = 09h | DX = 0142h

OUTPUT

```
123321
palindrome!
```

0102h	0103h	0104h	0105h	0106h	0107h	0108h	0109h	010Ah
'1'	'2'	'3'	'3'	'2'	'1'	0Dh	0Ah	'$'

str_size = 6

INT 21h: prints **palindrome!** (DOS returns from this function with AL = 24h, i.e. '$'.)

Code Trace: Palindrome (2/2) Comparison

```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
▶  jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
19 ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = **24h** | BL = **33h** | DI = **0105h**
SI = **0104h** | CX = **0000h** | ZF = **0**
AH = **09h** | DX = **0142h**

OUTPUT

```
123321
palindrome!
```

0102h	0103h	0104h	0105h	0106h	0107h	0108h	0109h	010Ah
'1'	'2'	'3'	'3'	'2'	'1'	0Dh	0Ah	'$'

str_size = 6

`jmp stop` : skips the "not palindrome!" block.

Step 13 / 14

Code Trace: Palindrome (2/2) Comparison

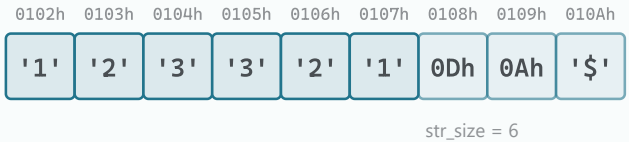
```
1  next_char:
2  mov al, [di]          ; char at the start
3  mov bl, [si]          ; char at the end
4  cmp al, bl
5  jne not_palindrome    ; differ: not one
6  inc di                ; start: one forward
7  dec si                ; end: one back
8  loop next_char
9  is_palindrome:
10 mov ah, 9             ; print "palindrome!"
11 mov dx, offset msg1
12 int 21h
13 jmp stop
14 not_palindrome:
15 mov ah, 9             ; "not palindrome!"
16 mov dx, offset msg2
17 int 21h
18 stop:
▶  ret
20 msg1 db "palindrome!$"
21 msg2 db "not palindrome!$"
```

VARIABLES

AL = **24h** | BL = **33h** | DI = **0105h** | SI = **0104h** | CX = **0000h** | ZF = **0** | AH = **09h** | DX = **0142h**

OUTPUT

```
123321
palindrome!
```



`ret`. The data comes right after the `stop:` label, so without `ret` the CPU would execute the bytes of "palindrome!$" as instructions.

Step 14 / 14

Palindrome: Different Inputs

str	Length	CX (after shr)	Comparisons	Output
`'123321'`	6	3	1=1, 2=2, 3=3	palindrome!
`'123421'`	6	3	1=1, 2=2, 3≠4 → jne	not palindrome!
`'12321'`	5	2	1=1, 2=2 (the middle 3 is skipped)	palindrome!
`'7'`	1	—	cmp cx,1 → je is_palindrome	palindrome!
`'ab'`	2	1	a≠b → jne	not palindrome!

Why `cmp cx, 1`? With length 1, `shr cx, 1` makes CX 0; entering a LOOP-based loop with CX = 0 would run the comparison 65,536 times. Length 0 has the same danger: guard it with `jcxz`.

Lab: Jumps and Loops

JMP example

CODE · **15 bytes**

```
 1 org    100h
 2 mov    ax, 5              ; ax…
 3 mov    bx, 2              ; bx…
 4 jmp    calc               ; go…
 5 back:  jmp stop           ; go…
 6 calc:
 7 add    ax, bx             ; ax…
 8 jmp    back               ; ba…
 9 stop:
10 ret                      ; re…
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	000F	CH	00	CL	0F
DX	0000	DH	00	DL	00

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

FLAGS

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

CS:IP = 0700:0100 · step 0

MEMORY **DS = 0700h**

```
0100 B8 05 00 BB 02 00 EB 02
0108 EB 04 03 C3 EB FA C3 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: 15 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (000Fh). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary

JMP
Unconditional; IP ← target. Short (EB, −128…+127), near (E9, within segment), far (EA, CS:IP).

Displacement
target − address of the next instruction; stored in two's complement if negative.

Jcc
Tests the flags without changing them; on the 8086 short only (2 bytes).

CMP
A SUB that keeps no result; the Jcc that follows makes the decision.

Unsigned
JA, JAE, JB, JBE → CF and ZF.

Signed
JG, JGE, JL, JLE → ZF, SF and OF.

LOOP family
CX is decremented, flags are preserved; LOOPE/LOOPNE also test ZF.

Pitfalls
LOOP with CX = 0, not saving CX in nested loops, executing data as code.

SUMMARY

Exercises

1. The `JNE` at address 0120h must jump to address 0100h. What is its machine code?
2. After `mov al, 80h` / `cmp al, 1`, do `JA` and `JG` jump?
3. How many times does `mov cx, 0` / `d: loop d` run? How do you prevent it?
4. Write code that sets BX = 1 if the unsigned number in AX is greater than 100, and BX = 0 otherwise.
5. In the array sum example, what would AL and CF be if the elements were 200 and 100? Suggest a fix.
6. What happens in the Hello example if the `jmp start` line is deleted?

▶ **Hints / answers**

References

- emu8086 documentation: *8086 Assembler Tutorial for Beginners* (Part 7: Program Flow Control).
- Intel, *The 8086 Family User's Manual*, 1979: instruction set and encoding tables.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009: Chapter 6 "Program Control Instructions".
- Kip R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014: conditional processing and loops.

The End

Chapter 7: Flow Control

Previous: Chapter 6: Arithmetic, Logic and Shift Instructions · Next: Chapter 8.1: The Stack