Chapter 6: Arithmetic, Logic and Shift Instructions

Microprocessors

FLAGS register | CF · ZF · SF · OF · PF · AF | Instruction groups | ADD · SUB · CMP | AND · OR · XOR · TEST | MUL · IMUL · DIV · IDIV | INC · DEC · NOT · NEG | SHL · SHR · SAR · ROL · RCR | Array addition | BCD: AAA · AAS | Temperature conversion

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_06_Arithmetic_Logic_and_Shift_Instructions.html

Learning Outcomes

Flags

Compute CF, ZF, SF, OF, PF and AF from the result of an operation; explain the difference between CF and OF.

Instructions

Use addition, subtraction, comparison, logic, multiplication, division, single-operand, shift and rotate instructions with valid operands; trace how shifts affect CF.

Multiply / divide

Know the implicit registers of MUL, IMUL, DIV, IDIV (AL, AX, DX:AX) and the divide error.

Practice

Trace the array addition, BCD adjustment and temperature conversion programs step by step.

Arithmetic and Logic Instructions

- The results of arithmetic and logic instructions affect the CPU's **status flags**.
- The flags live in the 16-bit **FLAGS** register; each bit is a flag that is either 0 or 1.
- The 8086 uses 9 of these 16 bits: 6 **status** flags (describe the result) and 3 **control** flags (steer the CPU's behavior).
- Conditional jumps (`JZ` , `JC` , `JO` ...) make their decisions from these flags.



Dashed bits are unused on the 8086.

The Flags at a Glance

Bit	Flag	Kind	Set to 1 if…	Example / note
0	**CF** · Carry	status	unsigned overflow: carry on addition, borrow on subtraction	255 + 1 → AL = 00h, CF = 1; 7 − 9 → CF = 1; `STC`, `CLC`, `CMC`
2	**PF** · Parity	status	the **low byte** of the result has an even number of 1 bits	96h = 1001 0110b → PF = 1; even for word results, only the low byte
4	**AF** · Auxiliary	status	carry/borrow from bit 3 into bit 4	08h + 08h → AF = 1; used only by AAA, AAS, DAA, DAS
6	**ZF** · Zero	status	result is zero	`SUB AL, AL` → ZF = 1; after `CMP AX, BX`, ZF = 1 means AX = BX
7	**SF** · Sign	status	the most significant bit (MSB) of the result is 1	copy of the MSB: 96h → SF = 1
8	**TF** · Trap	control	INT 1 after every instruction	single-step debugging
9	**IF** · Interrupt	control	INTR interrupts are accepted	`STI` → 1, `CLI` → 0; does not affect NMI
10	**DF** · Direction	control	string instructions run backward	0: SI/DI increment, 1: decrement; `CLD` / `STD`
11	**OF** · Overflow	status	signed overflow: the result does not fit the signed range	100 + 50 → AL = 96h (−106), OF = 1

CF or OF?

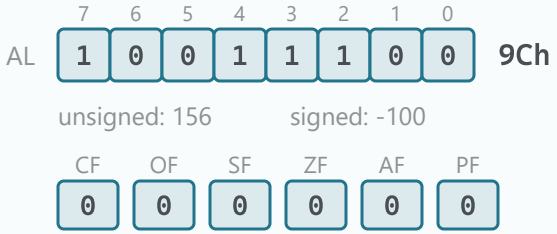
The CPU adds bits and nothing more; **the interpretation is the programmer's**. CF says whether the result is right as unsigned, OF whether it is right as signed.

AL + operand	unsigned view	signed view	AL	CF	OF
`FFh + 01h`	255 + 1 = 256	-1 + 1 = 0	`00h`	**1**	**0**
`64h + 32h`	100 + 50 = 150	100 + 50 = 150	`96h`	**0**	**1**
`C8h + 64h`	200 + 100 = 300	-56 + 100 = 44	`2Ch`	**1**	**0**
`80h + 80h`	128 + 128 = 256	-128 + -128 = -256	`00h`	**1**	**1**
`05h + 0Ah`	5 + 10 = 15	5 + 10 = 15	`0Fh`	**0**	**0**

OF rule: if two numbers of the same sign add up to a result of the other sign, OF = 1. Adding two numbers of different signs never sets OF.

Code Trace: CF, OF and the Other Flags

```
1 ▶ MOV  AL, 9Ch      ; 156 or -100
2   ADD  AL, 70h      ; + 112
3   MOV  AL, 50h      ; 80
4   ADD  AL, 50h      ; + 80
5   ADD  AL, AL       ; A0h + A0h
6   MOV  AL, 0FFh
7   INC  AL           ; CF untouched
8   CMC               ; flip CF
9   RET
```



AL = 9Ch: unsigned 156, signed −100. The flags start at 0; MOV does not change them.

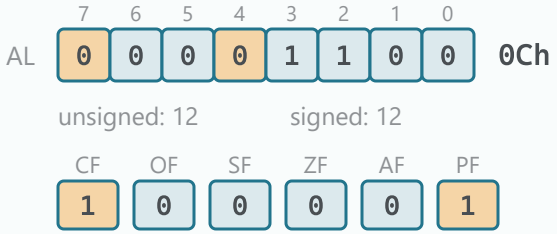
Step 1 / 8

VARIABLES

AL = **9Ch** | CF = **0** | OF = **0** | ZF = **0**

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
▶ ADD  AL, 70h      ; + 112
3 MOV  AL, 50h      ; 80
4 ADD  AL, 50h      ; + 80
5 ADD  AL, AL       ; A0h + A0h
6 MOV  AL, 0FFh
7 INC  AL           ; CF untouched
8 CMC               ; flip CF
9 RET
```



156 + 112 = 268 > 255 → AL = 0Ch, **CF = 1**. Signed −100 + 112 = 12 is correct → OF = 0. 0Ch has two 1s → PF = 1.

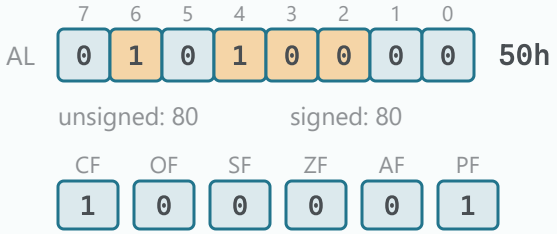
Step 2 / 8

VARIABLES

`AL = 0Ch` `CF = 1` `OF = 0` `ZF = 0`

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
2 ADD  AL, 70h      ; + 112
▶ MOV  AL, 50h      ; 80
4 ADD  AL, 50h      ; + 80
5 ADD  AL, AL       ; A0h + A0h
6 MOV  AL, 0FFh
7 INC  AL           ; CF untouched
8 CMC               ; flip CF
9 RET
```



AL = 50h (80). The flags still reflect the ADD.

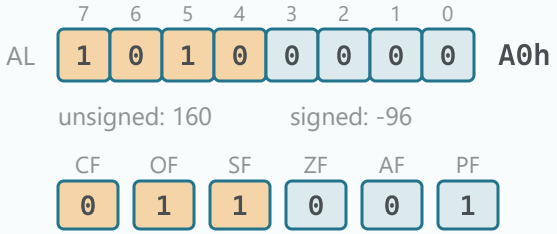
Step 3 / 8

VARIABLES

AL = 50h | CF = 1 | OF = 0 | ZF = 0

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
2 ADD  AL, 70h      ; + 112
3 MOV  AL, 50h      ; 80
▶ ADD  AL, 50h      ; + 80
5 ADD  AL, AL       ; A0h + A0h
6 MOV  AL, 0FFh
7 INC  AL           ; CF untouched
8 CMC               ; flip CF
9 RET
```



80 + 80 = 160 is correct unsigned (CF = 0), but two positives gave a negative sum (A0h = −96) → **OF = 1**, SF = 1.

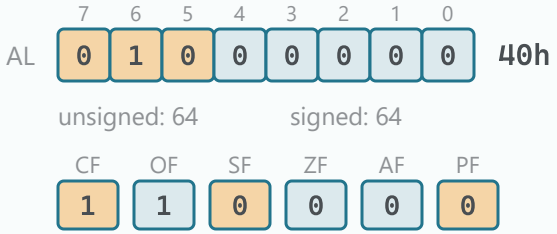
Step 4 / 8

VARIABLES

`AL = A0h` `CF = 0` `OF = 1` `ZF = 0`

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
2 ADD  AL, 70h      ; + 112
3 MOV  AL, 50h      ; 80
4 ADD  AL, 50h      ; + 80
▶ ADD  AL, AL       ; A0h + A0h
6 MOV  AL, 0FFh
7 INC  AL           ; CF untouched
8 CMC               ; flip CF
9 RET
```



A0h + A0h: unsigned 320 (CF = 1), signed −96 + (−96) = −192 (OF = 1). AL = 40h: both views are wrong.

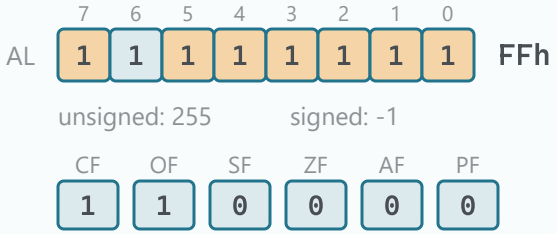
Step 5 / 8

VARIABLES

AL = 40h | CF = 1 | OF = 1 | ZF = 0

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
2 ADD  AL, 70h      ; + 112
3 MOV  AL, 50h      ; 80
4 ADD  AL, 50h      ; + 80
5 ADD  AL, AL       ; A0h + A0h
▶ MOV  AL, 0FFh
7 INC  AL           ; CF untouched
8 CMC               ; flip CF
9 RET
```



AL = FFh. CF = 1 is still there.

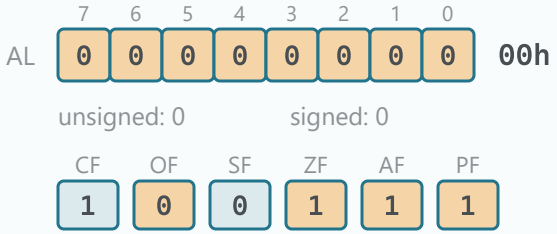
Step 6 / 8

VARIABLES

AL = FFh | CF = 1 | OF = 1 | ZF = 0

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
2 ADD  AL, 70h      ; + 112
3 MOV  AL, 50h      ; 80
4 ADD  AL, 50h      ; + 80
5 ADD  AL, AL       ; A0h + A0h
6 MOV  AL, 0FFh
▶ INC  AL           ; CF untouched
8 CMC               ; flip CF
9 RET
```



`INC AL` : FFh + 1 = 00h → ZF = 1, AF = 1; but INC **leaves CF alone**: CF = 1 is left over from the previous addition.

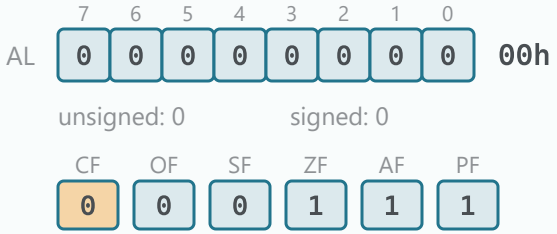
Step 7 / 8

VARIABLES

`AL = 00h` `CF = 1` `OF = 0` `ZF = 1`

Code Trace: CF, OF and the Other Flags

```
1 MOV  AL, 9Ch      ; 156 or -100
2 ADD  AL, 70h      ; + 112
3 MOV  AL, 50h      ; 80
4 ADD  AL, 50h      ; + 80
5 ADD  AL, AL       ; A0h + A0h
6 MOV  AL, 0FFh
7 INC  AL           ; CF untouched
▶ CMC               ; flip CF
9 RET
```



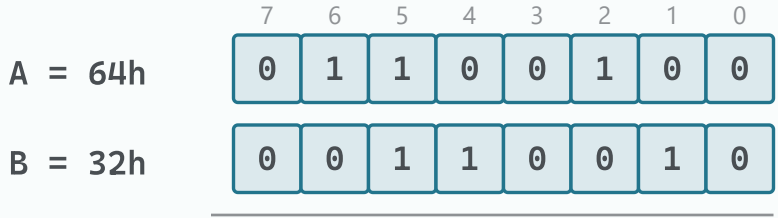
`CMC` complements CF: CF = 0. The other flags do not change.

Step 8 / 8

VARIABLES

`AL = 00h` `CF = 0` `OF = 0` `ZF = 1`

Flags After ADD, Step by Step



`MOV AL, 64h` (100) and `ADD AL, 32h` (50). Bits are added from right to left.

Step 1 / 6

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0.

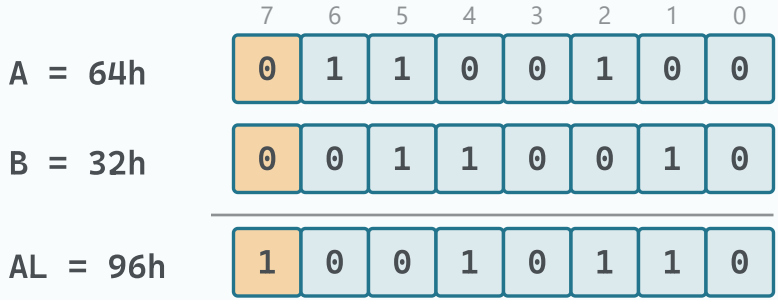
Flags After ADD, Step by Step



The result is **96h** = 1001 0110b. Unsigned it is 150 (right); signed it is −106 (wrong!).

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0.

Flags After ADD, Step by Step



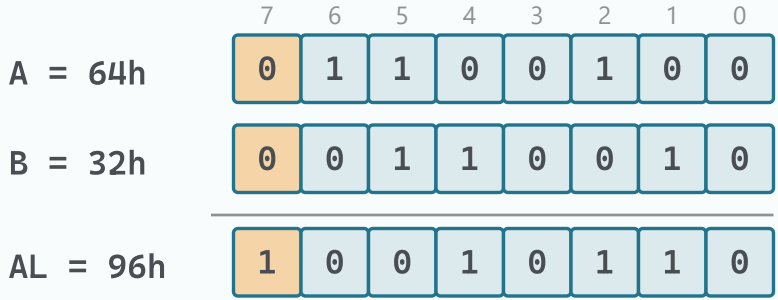
carry out of bit 7: 0

→ CF = 0

CF: did a carry come out of the leftmost bit? No → CF = 0 (150 ≤ 255).

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0.

Flags After ADD, Step by Step



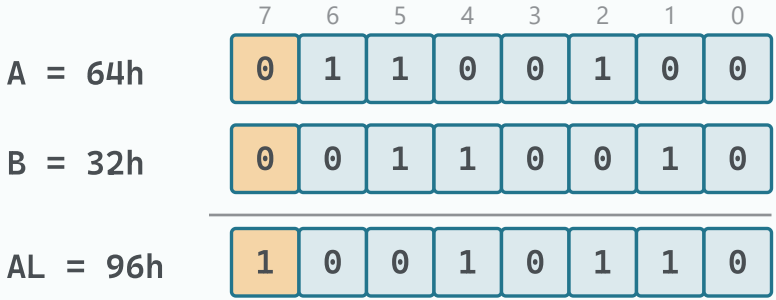
carry into bit 7 is 1, out is 0

→ OF = 1

OF: two positive numbers gave a negative sum (the carries into and out of bit 7 differ) → OF = 1.

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0.

Flags After ADD, Step by Step



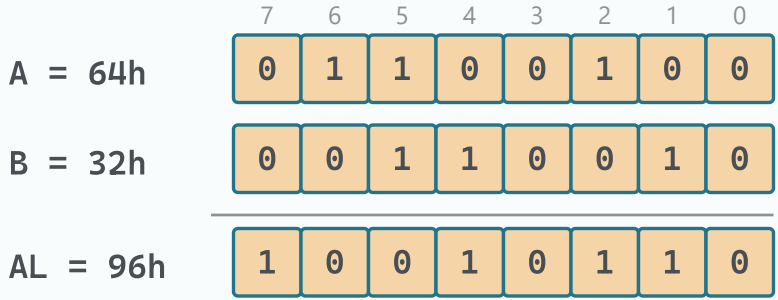
MSB = 1 → SF = 1

result ≠ 0 → ZF = 0

SF = bit 7 of the result = 1. The result is not zero: **ZF = 0**.

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0.

Flags After ADD, Step by Step



four 1s (even) → PF = 1

nibbles 4 + 2: no carry → AF = 0

PF: 1001 0110b has four 1s → even → PF = 1. **AF**: the low nibbles give 4 + 2 = 6, no carry → AF = 0.

Step 6 / 6

Result: AL = 96h; CF = 0, ZF = 0, SF = 1, OF = 1, PF = 1, AF = 0.

Demo: Flag Calculator

Pick two values and an instruction to see the result and the flags. Values may be decimal (`-5`), hex (`3Fh`) or binary (`1010b`).

instruction ADD size 8-bit destination 100 source 50

```
dest   0110 0100  = 64h
src    0011 0010  = 32h
result 1001 0110  = 96h  (unsigned 150, signed -106)
```

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

The result is written to dest.

INSTRUCTION GROUPS

Three Groups of Instructions

Group 1 · two operands

`ADD` , `SUB` , `CMP` , `AND` , `TEST` , `OR` , `XOR`

dest, src

Group 2 · multiply and divide

`MUL` , `IMUL` , `DIV` , `IDIV`

one operand + implicit AL / AX / DX

Group 3 · one operand

`INC` , `DEC` , `NOT` , `NEG`

the operand is both source and destination

Increment (INC) and decrement (DEC) are in **group 3**. The groups are defined by operand form.

dest ← dest ∘ src

```
ADD AX, BX
```

AX ← AL × operand

```
MUL BL
```

operand ← f(operand)

```
INC CX
```

What the Instructions Do

Group 1	What it does
`ADD`	adds two values, writes the result to dest
`SUB`	subtracts src from dest, writes the result to dest
`CMP`	subtracts but **does not store** the result; flags only
`AND`	bitwise AND
`TEST`	bitwise AND; result not stored
`OR`	bitwise OR
`XOR`	bitwise exclusive OR

Group 2	What it does
`MUL`	unsigned multiply
`IMUL`	signed multiply
`DIV`	unsigned divide (quotient + remainder)
`IDIV`	signed divide

Group 3	What it does
`INC`	adds 1
`DEC`	subtracts 1
`NOT`	inverts every bit
`NEG`	changes the sign (two's complement)

GROUP 1: TWO OPERANDS

ADD, SUB, CMP, AND, TEST, OR, XOR: Operands

Form	Example
REG, memory	`ADD AX, [BX]`
memory, REG	`SUB var1, CL`
REG, REG	`XOR AX, AX`
memory, immediate	`AND byte ptr [SI], 0Fh`
REG, immediate	`CMP AL, 'A'`

- **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- **memory**: `[BX]`, `[BX+SI+7]`, a variable ...
- **immediate**: `5`, `-24`, `3Fh`, `10001101b` ...

Invalid: `ADD [BX], [SI]` (two memory), `ADD AL, BX` (size mismatch), `ADD 5, AL` (immediate dest), `ADD DS, AX` (segment register).

Rules and Affected Flags

- After a two-operand instruction the result is always stored in the **first operand** (dest).
- `CMP` and `TEST` only affect the flags; they store no result.
- `ADD` / `SUB` : adds the second operand to / subtracts it from the first.
- `CMP` : subtracts the second operand from the first without storing the result.
- `AND` / `OR` / `XOR` : operate on corresponding bits of the two operands; `TEST` is like AND but only affects the flags.

Instruction	CF	ZF	SF	OF	PF	AF
`ADD` , `SUB` , `CMP`	✓	✓	✓	✓	✓	✓
`AND` , `OR` , `XOR` , `TEST`	0	✓	✓	0	✓	?

✓: set by the result · 0: always cleared · ?: undefined. Logic instructions always clear CF and OF.

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
2 add al, 20h       ; 240 + 32 = 272
3 mov bl, 7
4 sub bl, 9         ; 7 - 9
5 cmp al, 10h       ; AL - 10h
6 cmp bl, 0
7 add bl, 2         ; FEh + 2
```

AL = F0h. MOV does not change the flags (all 0 at the start).

Step 1 / 7

VARIABLES

AL = **F0h** BL = **00h** CF = **0** ZF = **0**

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

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
▶ add al, 20h       ; 240 + 32 = 272
3 mov bl, 7
4 sub bl, 9         ; 7 - 9
5 cmp al, 10h       ; AL - 10h
6 cmp bl, 0
7 add bl, 2         ; FEh + 2
```

F0h + 20h = 110h; AL keeps **10h**; the carry out → **CF = 1**. Signed: −16 + 32 = 16 is correct → OF = 0.

Step 2 / 7

VARIABLES

AL = **10h** | BL = **00h** | CF = **1** | ZF = **0** | SF = **0** | OF = **0** | PF = **0** | AF = **0**

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
2 add al, 20h       ; 240 + 32 = 272
▶ mov bl, 7
4 sub bl, 9         ; 7 - 9
5 cmp al, 10h       ; AL - 10h
6 cmp bl, 0
7 add bl, 2         ; FEh + 2
```

BL = 7. The flags still reflect the ADD.

Step 3 / 7

VARIABLES

AL = **10h** | BL = **07h** | CF = **1** | ZF = **0**

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

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
2 add al, 20h       ; 240 + 32 = 272
3 mov bl, 7
▶ sub bl, 9         ; 7 - 9
5 cmp al, 10h       ; AL - 10h
6 cmp bl, 0
7 add bl, 2         ; FEh + 2
```

7 − 9 = −2 = **FEh**. A borrow occurred → **CF = 1**; MSB = 1 → SF = 1. 7 − 9 borrows in the low nibble → AF = 1.

Step 4 / 7

VARIABLES

AL = **10h** | BL = **FEh** | CF = **1** | ZF = **0**

SF = **1** | OF = **0** | PF = **0** | AF = **1**

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
2 add al, 20h       ; 240 + 32 = 272
3 mov bl, 7
4 sub bl, 9         ; 7 - 9
▶ cmp al, 10h       ; AL - 10h
6 cmp bl, 0
7 add bl, 2         ; FEh + 2
```

`CMP AL, 10h` : 10h − 10h = 0 → **ZF = 1**. AL is **unchanged**: that is how equality is tested.

Step 5 / 7

VARIABLES

AL = **10h** BL = **FEh** CF = **0** ZF = **1**

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

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
2 add al, 20h       ; 240 + 32 = 272
3 mov bl, 7
4 sub bl, 9         ; 7 - 9
5 cmp al, 10h       ; AL - 10h
▶ cmp bl, 0
7 add bl, 2         ; FEh + 2
```

FEh − 0: ZF = 0, SF = 1, CF = 0. Unsigned FEh > 0, signed −2 < 0.

Step 6 / 7

VARIABLES

AL = **10h** | BL = **FEh** | CF = **0** | ZF = **0** | SF = **1** | OF = **0** | PF = **0** | AF = **0**

Code Trace: ADD, SUB, CMP

```
1 mov al, 0F0h      ; 240
2 add al, 20h       ; 240 + 32 = 272
3 mov bl, 7
4 sub bl, 9         ; 7 - 9
5 cmp al, 10h       ; AL - 10h
6 cmp bl, 0
▶ add bl, 2         ; FEh + 2
```

FEh + 2 = 100h → BL = **00h**, CF = 1, ZF = 1. Signed: −2 + 2 = 0 is correct → OF = 0.

Step 7 / 7

VARIABLES

AL = **10h** BL = **00h** CF = **1** ZF = **1**

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

32-Bit Addition with ADC

```
1 ; DX:AX = 0001FFFFh, CX:BX = 00020001h
▶ mov ax, 0FFFFh
3 mov dx, 0001h
4 mov bx, 0001h
5 mov cx, 0002h
6 add ax, bx      ; low words
7 adc dx, cx      ; high + CF
```

The first 32-bit number is loaded into DX:AX: low word FFFFh...

Step 1 / 5

VARIABLES

AX = **FFFFh** DX = **0000h** BX = **0000h**

CX = **0000h** CF = **0** ZF = **0**

SBB (subtract with borrow) does multi-word subtraction the same way: SUB first, then SBB.

32-Bit Addition with ADC

```
1 ; DX:AX = 0001FFFFh, CX:BX = 00020001h
2 mov ax, 0FFFFh
▶ mov dx, 0001h
4 mov bx, 0001h
5 mov cx, 0002h
6 add ax, bx      ; low words
7 adc dx, cx      ; high + CF
```

...high word 0001h → DX:AX = 0001FFFFh (131071).

Step 2 / 5

VARIABLES

AX = **FFFFh** | DX = **0001h** | BX = **0000h**

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

SBB (subtract with borrow) does multi-word subtraction the same way: SUB first, then SBB.

32-Bit Addition with ADC

```
1 ; DX:AX = 0001FFFFh, CX:BX = 00020001h
2 mov ax, 0FFFFh
3 mov dx, 0001h
▶ mov bx, 0001h
▶ mov cx, 0002h
6 add ax, bx      ; low words
7 adc dx, cx      ; high + CF
```

The second number is CX:BX = 00020001h (131073).

Step 3 / 5

VARIABLES

AX = **FFFFh** DX = **0001h** BX = **0001h**

CX = **0002h** CF = **0** ZF = **0**

SBB (subtract with borrow) does multi-word subtraction the same way: SUB first, then SBB.

32-Bit Addition with ADC

```
1 ; DX:AX = 0001FFFFh, CX:BX = 00020001h
2 mov ax, 0FFFFh
3 mov dx, 0001h
4 mov bx, 0001h
5 mov cx, 0002h
▶ add ax, bx       ; low words
7 adc dx, cx       ; high + CF
```

FFFFh + 0001h = 10000h → AX = 0000h, **CF = 1**: a carry for the high word.

Step 4 / 5

VARIABLES

AX = 0000h | DX = 0001h | BX = 0001h | CX = 0002h | CF = 1 | ZF = 1

SBB (subtract with borrow) does multi-word subtraction the same way: SUB first, then SBB.

32-Bit Addition with ADC

```
1 ; DX:AX = 0001FFFFh, CX:BX = 00020001h
2 mov ax, 0FFFFh
3 mov dx, 0001h
4 mov bx, 0001h
5 mov cx, 0002h
6 add ax, bx      ; low words
▶ adc dx, cx      ; high + CF
```

`ADC` = ADD + CF: 0001h + 0002h + 1 = **0004h**.
Result DX:AX = 00040000h (262144) ✓

Step 5 / 5

VARIABLES

AX = **0000h** | DX = **0004h** | BX = **0001h**
CX = **0002h** | CF = **0** | ZF = **0**

SBB (subtract with borrow) does multi-word subtraction the same way: SUB first, then SBB.

32-Bit Subtraction with SBB

```
1 ; DX:AX = 00040000h, CX:BX = 00010001h
▶ mov ax, 0000h
▶ mov dx, 0004h
4 mov bx, 0001h
5 mov cx, 0001h
6 sub ax, bx      ; low words
7 sbb dx, cx      ; high - CF
```

The first number is DX:AX = 00040000h (262144): high word in DX, low word in AX.

Step 1 / 4

VARIABLES

AX = 0000h | DX = 0004h | BX = 0000h | CX = 0000h | CF = 0 | ZF = 0

Rule: `SUB` on the lowest word, `SBB` on every word after it. Instructions in between must not disturb CF (MOV, INC, LOOP preserve it).

32-Bit Subtraction with SBB

```
1 ; DX:AX = 00040000h, CX:BX = 00010001h
2 mov ax, 0000h
3 mov dx, 0004h
▶ mov bx, 0001h
▶ mov cx, 0001h
6 sub ax, bx      ; low words
7 sbb dx, cx      ; high - CF
```

The subtrahend is CX:BX = 00010001h (65537).

Step 2 / 4

VARIABLES

AX = **0000h** DX = **0004h** BX = **0001h**

CX = **0001h** CF = **0** ZF = **0**

Rule: `SUB` on the lowest word, `SBB` on every word after it. Instructions in between must not disturb CF (MOV, INC, LOOP preserve it).

32-Bit Subtraction with SBB

```
1 ; DX:AX = 00040000h, CX:BX = 00010001h
2 mov ax, 0000h
3 mov dx, 0004h
4 mov bx, 0001h
5 mov cx, 0001h
▶ sub ax, bx      ; low words
7 sbb dx, cx      ; high - CF
```

0000h − 0001h: the low word is too small, so a **borrow** occurs → AX = **FFFFh**, **CF = 1**.

Step 3 / 4

VARIABLES

AX = **FFFFh** | DX = **0004h** | BX = **0001h** | CX = **0001h** | CF = **1** | ZF = **0**

Rule: `SUB` on the lowest word, `SBB` on every word after it. Instructions in between must not disturb CF (MOV, INC, LOOP preserve it).

32-Bit Subtraction with SBB

```
1 ; DX:AX = 00040000h, CX:BX = 00010001h
2 mov ax, 0000h
3 mov dx, 0004h
4 mov bx, 0001h
5 mov cx, 0001h
6 sub ax, bx      ; low words
▶ sbb dx, cx      ; high - CF
```

`SBB` = SUB − CF: 0004h − 0001h − 1 = **0002h**.
Result DX:AX = 0002FFFFh (196607) ✓

Step 4 / 4

VARIABLES

AX = **FFFFh** | DX = **0002h** | BX = **0001h**
CX = **0001h** | CF = **0** | ZF = **0**

Rule: `SUB` on the lowest word, `SBB` on every word after it. Instructions in between must not disturb CF (MOV, INC, LOOP preserve it).

Logic Operation Rules

AND

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

1 if both are 1.

OR

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

1 if at least one is 1.

XOR

A	B	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

1 if they differ.

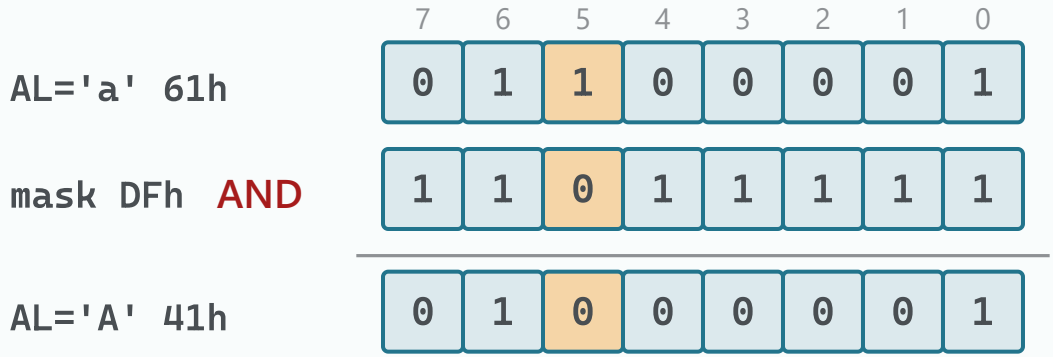
NOT

A	NOT A
1	0
0	1

Inverts. One operand.

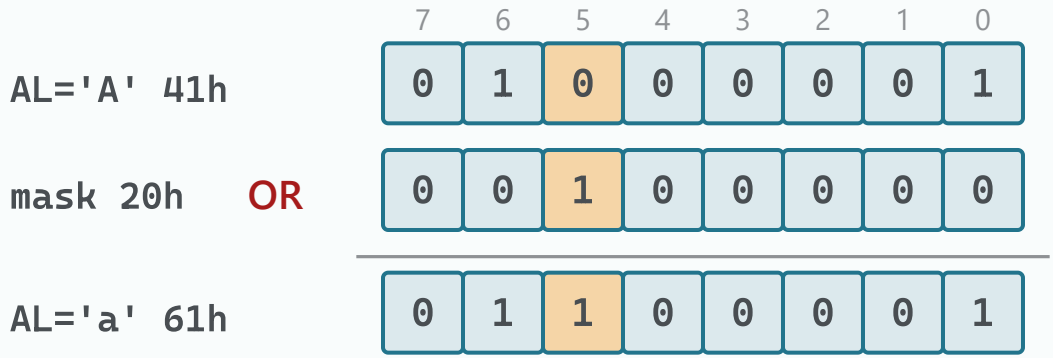
These rules apply to **each** of the 8 or 16 bits separately: `AND AL, BL` is really 8 independent ANDs.

Masking: AND, OR, XOR, TEST



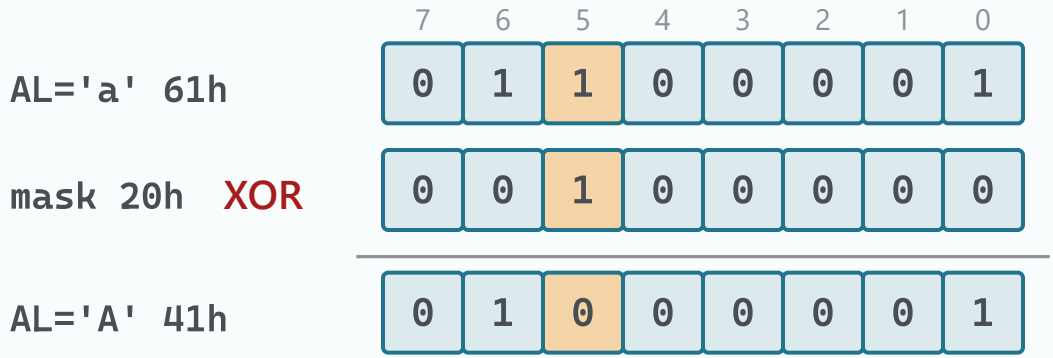
AND clears bits: bit 5, which is 0 in the mask, is cleared and the rest are kept → lowercase becomes uppercase.

Masking: AND, OR, XOR, TEST



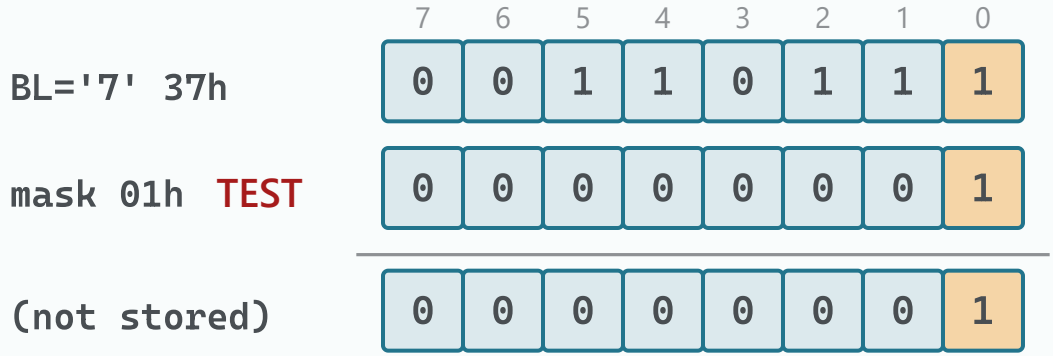
OR sets bits: bit 5, which is 1 in the mask, becomes 1 → uppercase becomes lowercase.

Masking: AND, OR, XOR, TEST



XOR flips bits: bit 5, which is 1 in the mask, changes → the letter switches case.

Masking: AND, OR, XOR, TEST



TEST tests bits: the result 01h ≠ 0 → ZF = 0, so bit 0 = 1 and the number is odd. BL is unchanged.

Common Idioms

Code	Effect	Why?
`XOR AX, AX`	AX = 0, ZF = 1, CF = 0	shorter than `MOV AX, 0` (2 bytes); also clears flags
`AND AL, 0Fh`	'7' (37h) → 7	turns an ASCII digit into a number (high nibble cleared)
`OR AL, 30h`	7 → '7' (37h)	turns a number into an ASCII digit
`AND AL, 0DFh`	'a' → 'A'	clears bit 5: uppercase
`OR AL, 20h`	'A' → 'a'	sets bit 5: lowercase
`TEST AL, 1`	odd if ZF = 0	tests bit 0 without changing AL
`TEST AL, 80h`	negative if ZF = 0	tests the sign bit
`OR AX, AX`	AX unchanged	sets ZF/SF: is AX zero, is it negative?

Code Trace: AND, OR, XOR, TEST

```
▶ mov al, 'a'      ; 61h
2 and al, 0DFh     ; uppercase
3 or  al, 20h      ; lowercase
4 xor al, 20h      ; flip case
5 mov bl, '7'      ; 37h
6 and bl, 0Fh      ; number 7
7 test bl, 1       ; odd?
8 xor bl, bl       ; BL = 0
```

AL = 'a' = 61h = 0110 0001b.

Step 1 / 8

VARIABLES

AL = **61h** | BL = **00h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'       ; 61h
▶ and al, 0DFh      ; uppercase
3 or  al, 20h       ; lowercase
4 xor al, 20h       ; flip case
5 mov bl, '7'       ; 37h
6 and bl, 0Fh       ; number 7
7 test bl, 1        ; odd?
8 xor bl, bl        ; BL = 0
```

61h AND DFh = **41h** = 'A'. 41h has two 1s → PF = 1. CF = OF = 0.

Step 2 / 8

VARIABLES

AL = **41h** | BL = **00h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **1**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'       ; 61h
2 and al, 0DFh      ; uppercase
▶ or  al, 20h       ; lowercase
4 xor al, 20h       ; flip case
5 mov bl, '7'       ; 37h
6 and bl, 0Fh       ; number 7
7 test bl, 1        ; odd?
8 xor bl, bl        ; BL = 0
```

41h OR 20h = **61h** = 'a'. Three 1s → PF = 0.

Step 3 / 8

VARIABLES

AL = **61h** | BL = **00h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'       ; 61h
2 and al, 0DFh      ; uppercase
3 or  al, 20h       ; lowercase
▶ xor al, 20h       ; flip case
5 mov bl, '7'       ; 37h
6 and bl, 0Fh       ; number 7
7 test bl, 1        ; odd?
8 xor bl, bl        ; BL = 0
```

61h XOR 20h = **41h** = 'A' (bit 5 flipped).

Step 4 / 8

VARIABLES

AL = **41h** | BL = **00h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **1**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'      ; 61h
2 and al, 0DFh     ; uppercase
3 or  al, 20h      ; lowercase
4 xor al, 20h      ; flip case
▶ mov bl, '7'      ; 37h
6 and bl, 0Fh      ; number 7
7 test bl, 1       ; odd?
8 xor bl, bl       ; BL = 0
```

BL = '7' = 37h (an ASCII digit).

Step 5 / 8

VARIABLES

AL = **41h** | BL = **37h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **1**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'      ; 61h
2 and al, 0DFh     ; uppercase
3 or  al, 20h      ; lowercase
4 xor al, 20h      ; flip case
5 mov bl, '7'      ; 37h
▶ and bl, 0Fh      ; number 7
7 test bl, 1       ; odd?
8 xor bl, bl       ; BL = 0
```

37h AND 0Fh = **07h**: now the number 7.

Step 6 / 8

VARIABLES

AL = **41h** | BL = **07h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'       ; 61h
2 and al, 0DFh      ; uppercase
3 or  al, 20h       ; lowercase
4 xor al, 20h       ; flip case
5 mov bl, '7'       ; 37h
6 and bl, 0Fh       ; number 7
▶ test bl, 1        ; odd?
8 xor bl, bl        ; BL = 0
```

07h AND 01h = 01h ≠ 0 → **ZF = 0**: the number is odd. BL is unchanged.

Step 7 / 8

VARIABLES

AL = **41h** | BL = **07h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0**

Code Trace: AND, OR, XOR, TEST

```
1 mov al, 'a'       ; 61h
2 and al, 0DFh      ; uppercase
3 or  al, 20h       ; lowercase
4 xor al, 20h       ; flip case
5 mov bl, '7'       ; 37h
6 and bl, 0Fh       ; number 7
7 test bl, 1        ; odd?
▶ xor bl, bl        ; BL = 0
```

XOR with itself → **00h**, ZF = 1, PF = 1 (zero 1s, even).

Step 8 / 8

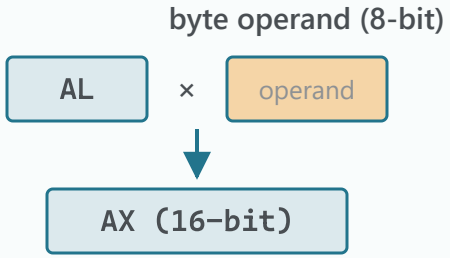
VARIABLES

AL = **41h** BL = **00h** CF = **0** ZF = **1**

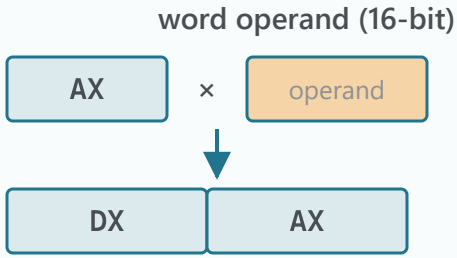
SF = **0** OF = **0** PF = **1**

GROUP 2: MULTIPLY AND DIVIDE

MUL and IMUL



MUL / IMUL



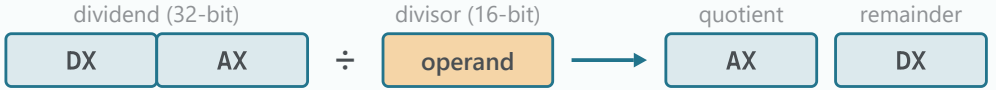
32-bit result DX:AX (DX high)

Instruction	Byte operand	Word operand
`MUL` (unsigned)	AX = AL × operand	(DX AX) = AX × operand
`IMUL` (signed)	AX = AL × operand	(DX AX) = AX × operand

Operand: REG or memory (`MUL BL` , `IMUL word ptr [SI]`). The 8086 has **no immediate operand** here: `MUL 5` is invalid.

DIV and IDIV

Instruction	Byte operand	Word operand
`DIV` (unsigned)	AL = AX / operand AH = remainder (modulus)	AX = (DX AX) / operand DX = remainder (modulus)
`IDIV` (signed)	AL = AX / operand AH = remainder	AX = (DX AX) / operand DX = remainder



Flags

MUL / IMUL set only CF and OF meaningfully: if the result exceeds the operand size (the upper half is needed), CF = OF = 1. The rest are undefined.

After **DIV / IDIV** all flags are undefined.

Divide error

If the divisor is 0, or the quotient does not fit in its destination (AL / AX), the CPU raises **INT 0** (divide error).

IDIV truncates the quotient toward zero; the remainder has the sign of the dividend.

Multiply and Divide Step by Step

AL C8h BL 03h

`MOV AL, 200`, `MOV BL, 3`. 200 = C8h, which is −56 signed.

Multiply and Divide Step by Step

AX	CF	OF
0258h	1	1

`MUL BL` : 200 × 3 = 600 = **0258h** → AX. The result did not fit in AL (AH ≠ 0) → **CF = OF = 1**.

Multiply and Divide Step by Step

AX	CF	OF
FF58h	1	1

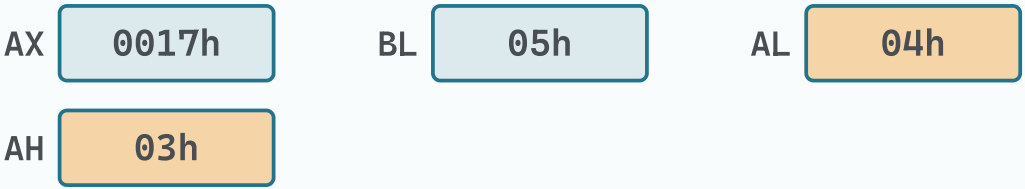
Same bits with `IMUL BL`: −56 × 3 = −168 = **FF58h**. A different result! −168 does not fit in a byte → CF = OF = 1.

Multiply and Divide Step by Step

AX	CF	OF
FFF6h	0	0

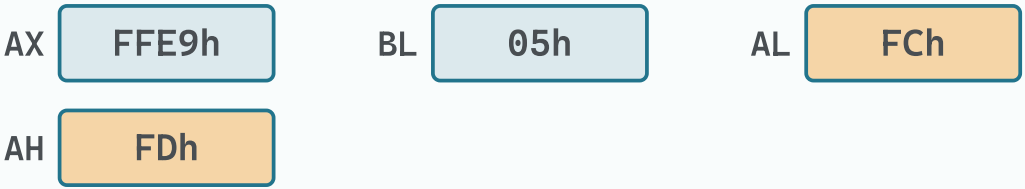
AL = FEh (−2), BL = 5: `IMUL BL` → −10 = **FFF6h**. AH is just the sign extension of AL → **CF = OF = 0**: the result fit in AL.

Multiply and Divide Step by Step



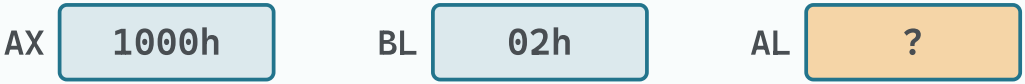
`DIV BL` : AX = 23, BL = 5 → quotient **AL = 4**, remainder **AH = 3**. New AX = 0304h.

Multiply and Divide Step by Step



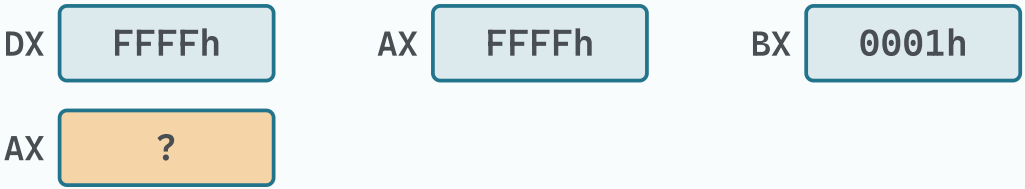
`IDIV BL`: −23 / 5 → quotient **AL = FCh (−4)**, remainder **AH = FDh (−3)**. Check: −4 × 5 + (−3) = −23.

Multiply and Divide Step by Step



AX = 1000h (4096), `DIV BL` (2): the quotient 2048 does not fit in AL → **divide error, INT 0**. A word division (DX:AX / BX) was needed.

Multiply and Divide Step by Step



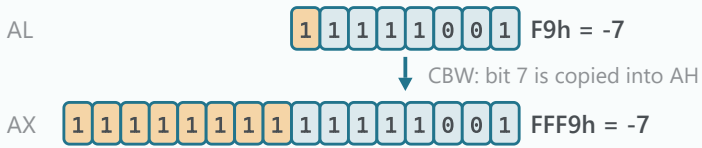
DX:AX = FFFFFFFFh, `DIV BX` (BX = 1): the divisor is not zero, but the quotient FFFFFFFFh does not fit in AX → **INT 0** again. Rule: in word division, if DX ≥ divisor the quotient cannot fit (in byte division, AH ≥ divisor).

Preparing to Divide: CBW, CWD

- Division needs a dividend **twice** as wide as the operand: AX for byte division, DX:AX for word division.
- Before **unsigned** division, clear the upper half: `MOV AH, 0` or `XOR DX, DX`.
- Before **signed** division, sign-extend: `CBW` (AL → AX), `CWD` (AX → DX:AX).
- Forgetting DX is a classic bug: an old value left in DX enlarges the dividend and can cause a divide error.

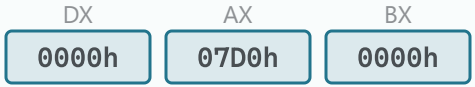
```
; unsigned: 1000 / 7
mov ax, 1000
xor dx, dx       ; DX:AX = 1000
mov bx, 7
div bx           ; AX = 142, DX = 6

; signed: -1000 / 7
mov ax, -1000
cwd              ; DX = FFFFh
idiv bx          ; AX = -142, DX = -6
```



Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX           ; unsigned div
5  MOV  AX, -2000
6  CWD                ; sign-extend
7  IDIV BX           ; signed div
8  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```



DX:AX = 000007D0h

unsigned 2000 · signed 2000

AX = 2000 = 07D0h. In word division the dividend is DX:AX; what DX holds matters too.

Step 1 / 9

VARIABLES

AX = **07D0h** DX = **0000h** BX = **0000h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
▶ XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX          ; unsigned div
5  MOV  AX, -2000
6  CWD              ; sign-extend
7  IDIV BX          ; signed div
8  MOV  AX, 2000    ; DX not cleared!
9  DIV  BX
10 RET
```



`XOR DX, DX`: the upper half is zero for unsigned division → DX:AX = 2000.

Step 2 / 9

VARIABLES

AX = **07D0h** DX = **0000h** BX = **0000h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
▶  MOV  BX, 9
4  DIV  BX           ; unsigned div
5  MOV  AX, -2000
6  CWD                ; sign-extend
7  IDIV BX           ; signed div
8  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```

DX	AX	BX
0000h	07D0h	0009h

DX:AX = 000007D0h

unsigned 2000 · signed 2000

Divisor BX = 9.

Step 3 / 9

VARIABLES

AX = **07D0h** DX = **0000h** BX = **0009h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
▶  DIV  BX           ; unsigned div
5  MOV  AX, -2000
6  CWD               ; sign-extend
7  IDIV BX           ; signed div
8  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```

DX	AX	BX
0002h	00DEh	0009h

DX:AX = 000200DEh

unsigned 131294 · signed 131294

quotient AX = 222, remainder DX = 2

`DIV BX` : 2000 / 9 → AX = 222 (00DEh), DX = 2. The flags are undefined.

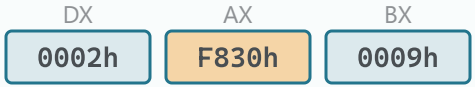
Step 4 / 9

VARIABLES

AX = 00DEh DX = 0002h BX = 0009h

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX           ; unsigned div
▶  MOV  AX, -2000
6  CWD               ; sign-extend
7  IDIV BX           ; signed div
8  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```



DX:AX = 0002F830h

unsigned 194608 · signed 194608

AX = −2000 = F830h. DX still holds the previous remainder (2): DX:AX is now a wrong dividend.

Step 5 / 9

VARIABLES

AX = **F830h** DX = **0002h** BX = **0009h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX           ; unsigned div
5  MOV  AX, -2000
▶  CWD               ; sign-extend
7  IDIV BX           ; signed div
8  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```

DX	AX	BX
FFFFh	F830h	0009h

DX:AX = FFFFF830h

unsigned 4294965296 · signed -2000

`CWD` : the sign bit of AX is 1 → DX = FFFFh. DX:AX = FFFFF830h, −2000 in 32 bits.

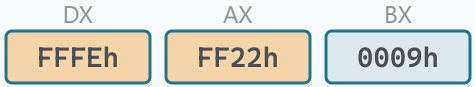
Step 6 / 9

VARIABLES

AX = **F830h** DX = **FFFFh** BX = **0009h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX           ; unsigned div
5  MOV  AX, -2000
6  CWD                ; sign-extend
▶ IDIV BX            ; signed div
8  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```



DX:AX = FFFEFF22h

unsigned 4294901538 · signed -65758

quotient AX = -222, remainder DX = -2

`IDIV BX` : −2000 / 9 → AX = FF22h (−222), DX = FFFEh (−2). The remainder has the sign of the dividend.

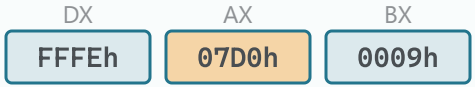
Step 7 / 9

VARIABLES

AX = **FF22h** DX = **FFFEh** BX = **0009h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX           ; unsigned div
5  MOV  AX, -2000
6  CWD               ; sign-extend
7  IDIV BX           ; signed div
▶  MOV  AX, 2000     ; DX not cleared!
9  DIV  BX
10 RET
```



DX:AX = FFFE07D0h

unsigned 4294838224 · signed -129072

AX = 2000, but DX = FFFEh was not cleared: DX:AX = FFFE07D0h, about 4.29 billion.

Step 8 / 9

VARIABLES

AX = **07D0h** DX = **FFFEh** BX = **0009h**

Code Trace: DIV, CWD and IDIV

```
1  MOV  AX, 2000
2  XOR  DX, DX       ; DX:AX = 2000
3  MOV  BX, 9
4  DIV  BX           ; unsigned div
5  MOV  AX, -2000
6  CWD                ; sign-extend
7  IDIV BX           ; signed div
8  MOV  AX, 2000     ; DX not cleared!
▶  DIV  BX
10 RET
```

DX	AX	BX
FFFEh	07D0h	0009h

DX:AX = FFFE07D0h

unsigned 4294838224 · signed -129072

quotient doesn't fit in AX → INT 0

`DIV BX`: the quotient does not fit in 16 bits (DX ≥ divisor) → the CPU raises **INT 0** and the program stops. Fix: `XOR DX, DX` before dividing.

Step 9 / 9

VARIABLES

AX = **07D0h** DX = **FFFEh** BX = **0009h**

Signed Overflow: JO and CBW

```
▶ mov al, -92       ; A4h
▶ mov bl, -92
3  add al, bl       ; 8-bit add
4  jo  widen        ; if OF = 1
5  jmp done
6  widen:
7  mov al, -92      ; first number
8  cbw              ; AL -> AX
9  mov dx, ax
10 mov al, bl       ; second number
11 cbw
12 add ax, dx       ; 16-bit add
13 done:
14 ret
```

AL = BL = A4h: −92 (256 − 92 = 164 = A4h).

Step 1 / 6

VARIABLES

AL = **A4h** AX = **00A4h** DX = **0000h** CF = **0**
OF = **0** SF = **0** ZF = **0**

`JO` (jump if overflow) jumps if OF = 1, `JNO` if OF = 0. If a signed byte sum can overflow, first widen the operands to words with `CBW`.

Signed Overflow: JO and CBW

```
1  mov al, -92      ; A4h
2  mov bl, -92
▶  add al, bl       ; 8-bit add
4  jo  widen        ; if OF = 1
5  jmp done
6  widen:
7  mov al, -92      ; first number
8  cbw              ; AL -> AX
9  mov dx, ax
10 mov al, bl       ; second number
11 cbw
12 add ax, dx       ; 16-bit add
13 done:
14 ret
```

A4h + A4h = 148h → AL = **48h** = +72. Two negatives gave a positive sum → **OF = 1** (CF = 1 too, but it does not matter in the signed view).

Step 2 / 6

VARIABLES

AL = **48h** AX = **0048h** DX = **0000h** CF = **1**
OF = **1** SF = **0** ZF = **0**

`JO` (jump if overflow) jumps if OF = 1, `JNO` if OF = 0. If a signed byte sum can overflow, first widen the operands to words with `CBW`.

Signed Overflow: JO and CBW

```
1  mov al, -92      ; A4h
2  mov bl, -92
3  add al, bl       ; 8-bit add
▶  jo  widen        ; if OF = 1
5  jmp done
6  widen:
7  mov al, -92      ; first number
8  cbw              ; AL -> AX
9  mov dx, ax
10 mov al, bl       ; second number
11 cbw
12 add ax, dx       ; 16-bit add
13 done:
14 ret
```

OF = 1, so `JO` jumps to the `widen` label. The true result −184 is outside −128 … 127.

Step 3 / 6

VARIABLES

AL = **48h** | AX = **0048h** | DX = **0000h** | CF = **1** | OF = **1** | SF = **0** | ZF = **0**

`JO` (jump if overflow) jumps if OF = 1, `JNO` if OF = 0. If a signed byte sum can overflow, first widen the operands to words with `CBW`.

Signed Overflow: JO and CBW

```
1  mov al, -92      ; A4h
2  mov bl, -92
3  add al, bl       ; 8-bit add
4  jo  widen        ; if OF = 1
5  jmp done
6  widen:
▶  mov al, -92      ; first number
▶  cbw              ; AL -> AX
9  mov dx, ax
10 mov al, bl       ; second number
11 cbw
12 add ax, dx       ; 16-bit add
13 done:
14 ret
```

AL = A4h, `CBW`: MSB = 1 → AH = FFh, AX = **FFA4h** (−92 in 16 bits).

Step 4 / 6

VARIABLES

AL = **A4h** | AX = **FFA4h** | DX = **0000h** | CF = **1** | OF = **1** | SF = **0** | ZF = **0**

`JO` (jump if overflow) jumps if OF = 1, `JNO` if OF = 0. If a signed byte sum can overflow, first widen the operands to words with `CBW`.

Signed Overflow: JO and CBW

```
 1 mov al, -92      ; A4h
 2 mov bl, -92
 3 add al, bl       ; 8-bit add
 4 jo  widen        ; if OF = 1
 5 jmp done
 6 widen:
 7 mov al, -92      ; first number
 8 cbw              ; AL -> AX
▶ mov dx, ax
▶ mov al, bl       ; second number
▶ cbw
12 add ax, dx       ; 16-bit add
13 done:
14 ret
```

The first number is saved in DX; the second (BL) is moved into AL and extended too: AX = FFA4h.

Step 5 / 6

VARIABLES

AL = **A4h** | AX = **FFA4h** | DX = **FFA4h** | CF = **1** | OF = **1** | SF = **0** | ZF = **0**

`JO` (jump if overflow) jumps if OF = 1, `JNO` if OF = 0. If a signed byte sum can overflow, first widen the operands to words with `CBW`.

Signed Overflow: JO and CBW

```
1  mov al, -92      ; A4h
2  mov bl, -92
3  add al, bl       ; 8-bit add
4  jo  widen        ; if OF = 1
5  jmp done
6  widen:
7  mov al, -92      ; first number
8  cbw              ; AL -> AX
9  mov dx, ax
10 mov al, bl       ; second number
11 cbw
▶ 12 add ax, dx     ; 16-bit add
13 done:
14 ret
```

FFA4h + FFA4h = 1FF48h → AX = **FF48h** = −184 ✓. The result fits in 16 bits: **OF = 0**.

Step 6 / 6

VARIABLES

AL = **48h** AX = **FF48h** DX = **FFA4h** CF = **1** OF = **0** SF = **1** ZF = **0**

`JO` (jump if overflow) jumps if OF = 1, `JNO` if OF = 0. If a signed byte sum can overflow, first widen the operands to words with `CBW`.

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
▶  lea si, arr
▶  mov cx, 4
▶  xor bx, bx       ; BX = 16-bit sum
9  again:
10 mov al, [si]
11 cbw              ; AL -> AX (signed)
12 add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

SI points to the start of the array, CX = 4 elements, BX = 0 (sum cleared).

Step 1 / 10

VARIABLES

AL = **00h** AX = **0000h** BX = **0000h** CX = **0004h**

total = **0000h**

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
6  lea si, arr
7  mov cx, 4
8  xor bx, bx       ; BX = 16-bit sum
9  again:
▶ 10 mov al, [si]
▶ 11 cbw             ; AL -> AX (signed)
12 add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

AL = 64h (+100). MSB = 0 → `CBW` AH = 00h, AX = 0064h.

Step 2 / 10

VARIABLES

AL = **64h** | AX = **0064h** | BX = **0000h** | CX = **0004h** | total = **0000h**

Signed Byte Sum with CBW

```
 1 org 100h
 2 jmp start
 3 arr db 100, -30, 90, -7
 4 total dw 0
 5 start:
 6 lea si, arr
 7 mov cx, 4
 8 xor bx, bx       ; BX = 16-bit sum
 9 again:
10 mov al, [si]
11 cbw              ; AL -> AX (signed)
▶ add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

BX = BX + AX = **0064h** (100).

Step 3 / 10

VARIABLES

AL = **64h** | AX = **0064h** | BX = **0064h** | CX = **0004h** | total = **0000h**

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
6  lea si, arr
7  mov cx, 4
8  xor bx, bx       ; BX = 16-bit sum
9  again:
▶  mov al, [si]
▶  cbw              ; AL -> AX (signed)
12 add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

AL = E2h (−30). MSB = 1 → `CBW` AH = FFh, AX = **FFE2h**.

Step 4 / 10

VARIABLES

AL = **E2h** AX = **FFE2h** BX = **0064h** CX = **0003h**

total = **0000h**

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
6  lea si, arr
7  mov cx, 4
8  xor bx, bx       ; BX = 16-bit sum
9  again:
10 mov al, [si]
11 cbw              ; AL -> AX (signed)
▶ 12 add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

BX = BX + AX = **0046h** (70).

Step 5 / 10

VARIABLES

AL = **E2h** | AX = **FFE2h** | BX = **0046h** | CX = **0003h** | total = **0000h**

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
6  lea si, arr
7  mov cx, 4
8  xor bx, bx       ; BX = 16-bit sum
9  again:
▶  mov al, [si]
▶  cbw              ; AL -> AX (signed)
12 add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

AL = 5Ah (+90). MSB = 0 → `CBW` AH = 00h, AX = 005Ah.

Step 6 / 10

VARIABLES

AL = **5Ah** AX = **005Ah** BX = **0046h** CX = **0002h**

total = **0000h**

Signed Byte Sum with CBW

```
 1 org 100h
 2 jmp start
 3 arr db 100, -30, 90, -7
 4 total dw 0
 5 start:
 6 lea si, arr
 7 mov cx, 4
 8 xor bx, bx       ; BX = 16-bit sum
 9 again:
10 mov al, [si]
11 cbw              ; AL -> AX (signed)
▶ add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

BX = **00A0h** (160). Adding bytes, 70 + 90 = 160 > 127 → AL = A0h (−96), OF = 1.

Step 7 / 10

VARIABLES

AL = **5Ah** | AX = **005Ah** | BX = **00A0h** | CX = **0002h** | total = **0000h**

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
6  lea si, arr
7  mov cx, 4
8  xor bx, bx       ; BX = 16-bit sum
9  again:
▶  mov al, [si]
▶  cbw              ; AL -> AX (signed)
12 add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

AL = F9h (−7). MSB = 1 → `CBW` AH = FFh, AX = **FFF9h**.

Step 8 / 10

VARIABLES

AL = **F9h** | AX = **FFF9h** | BX = **00A0h** | CX = **0001h** | total = **0000h**

Signed Byte Sum with CBW

```
 1 org 100h
 2 jmp start
 3 arr db 100, -30, 90, -7
 4 total dw 0
 5 start:
 6 lea si, arr
 7 mov cx, 4
 8 xor bx, bx       ; BX = 16-bit sum
 9 again:
10 mov al, [si]
11 cbw              ; AL -> AX (signed)
▶ add bx, ax
13 inc si
14 loop again
15 mov total, bx    ; 153 = 0099h
16 ret
```

BX = BX + AX = **0099h** (153).

Step 9 / 10

VARIABLES

AL = **F9h** | AX = **FFF9h** | BX = **0099h** | CX = **0001h** | total = **0000h**

Signed Byte Sum with CBW

```
1  org 100h
2  jmp start
3  arr db 100, -30, 90, -7
4  total dw 0
5  start:
6  lea si, arr
7  mov cx, 4
8  xor bx, bx       ; BX = 16-bit sum
9  again:
10 mov al, [si]
11 cbw              ; AL -> AX (signed)
12 add bx, ax
13 inc si
14 loop again
▶ 15 mov total, bx   ; 153 = 0099h
16 ret
```

The loop is done; total = **0099h** = 153 ✓ is written to memory. With `MOV AH, 0` instead of CBW, −30 = E2h would be added as 226.

Step 10 / 10

VARIABLES

AL = **F9h** | AX = **FFF9h** | BX = **0099h** | CX = **0000h** | total = **0099h**

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
▶  lea si, arr
▶  mov cx, 3
8  again:
9  mov ax, [si]
10 cwd                    ; AX -> DX:AX
11 add [total], ax        ; low word
12 adc [total+2], dx      ; high + CF
13 add si, 2
14 loop again
15 ret
```

SI = arr, CX = 3 words. The (32-bit) total in memory is 0000:0000.

Step 1 / 10

VARIABLES

`AX = 0000h` `DX = 0000h` `CF = 0`

`total = 0000h` `total+2 = 0000h`

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
▶  mov ax, [si]
▶  cwd                    ; AX -> DX:AX
11 add [total], ax        ; low word
12 adc [total+2], dx      ; high + CF
13 add si, 2
14 loop again
15 ret
```

AX = 7530h (30000). Positive → `CWD` gives DX = 0000h.

Step 2 / 10

VARIABLES

`AX = 7530h` `DX = 0000h` `CF = 0`

`total = 0000h` `total+2 = 0000h`

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
 1 org 100h
 2 jmp start
 3 arr dw 30000, 25000, -4000
 4 total dw 0, 0          ; 32-bit total
 5 start:
 6 lea si, arr
 7 mov cx, 3
 8 again:
 9 mov ax, [si]
10 cwd                    ; AX -> DX:AX
▶ add [total], ax         ; low word
12 adc [total+2], dx      ; high + CF
13 add si, 2
14 loop again
15 ret
```

Low word: 0000h + 7530h = 7530h, CF = 0.

Step 3 / 10

VARIABLES

AX = **7530h** | DX = **0000h** | CF = **0**

total = **7530h** | total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
9  mov ax, [si]
10 cwd                    ; AX -> DX:AX
11 add [total], ax        ; low word
▶ 12 adc [total+2], dx    ; high + CF
13 add si, 2
14 loop again
15 ret
```

High word: 0000h + 0000h + CF(0) = 0000h. Total 00007530h.

Step 4 / 10

VARIABLES

AX = **7530h** DX = **0000h** CF = **0**

total = **7530h** total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0        ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
▶  mov ax, [si]
▶  cwd                  ; AX -> DX:AX
11 add [total], ax      ; low word
12 adc [total+2], dx    ; high + CF
13 add si, 2
14 loop again
15 ret
```

AX = 61A8h (25000), DX = 0000h.

Step 5 / 10

VARIABLES

AX = **61A8h** | DX = **0000h** | CF = **0** | total = **7530h** | total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
9  mov ax, [si]
10 cwd                    ; AX -> DX:AX
▶ add [total], ax         ; low word
12 adc [total+2], dx      ; high + CF
13 add si, 2
14 loop again
15 ret
```

7530h + 61A8h = **D6D8h** (55000). Negative in a signed 16-bit view (OF = 1), but no problem since the total is kept in 32 bits.

Step 6 / 10

VARIABLES

AX = **61A8h** DX = **0000h** CF = **0**

total = **D6D8h** total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
 1 org 100h
 2 jmp start
 3 arr dw 30000, 25000, -4000
 4 total dw 0, 0          ; 32-bit total
 5 start:
 6 lea si, arr
 7 mov cx, 3
 8 again:
 9 mov ax, [si]
10 cwd                    ; AX -> DX:AX
11 add [total], ax        ; low word
▶ adc [total+2], dx       ; high + CF
13 add si, 2
14 loop again
15 ret
```

The high word stays 0000h: total 0000D6D8h = 55000.

Step 7 / 10

VARIABLES

AX = **61A8h** DX = **0000h** CF = **0**

total = **D6D8h** total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
▶  mov ax, [si]
▶  cwd                    ; AX -> DX:AX
11 add [total], ax        ; low word
12 adc [total+2], dx      ; high + CF
13 add si, 2
14 loop again
15 ret
```

AX = F060h (−4000). MSB = 1 → `CWD` gives **DX = FFFFh**: DX:AX = FFFFF060h (−4000 in 32 bits).

Step 8 / 10

VARIABLES

AX = **F060h** DX = **FFFFh** CF = **0**

total = **D6D8h** total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
9  mov ax, [si]
10 cwd                    ; AX -> DX:AX
▶ add [total], ax         ; low word
12 adc [total+2], dx      ; high + CF
13 add si, 2
14 loop again
15 ret
```

D6D8h + F060h = 1C738h → low word **C738h**, **CF = 1**.

Step 9 / 10

VARIABLES

AX = **F060h** DX = **FFFFh** CF = **1**

total = **C738h** total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

32-Bit Sum with CWD and ADC

```
1  org 100h
2  jmp start
3  arr dw 30000, 25000, -4000
4  total dw 0, 0          ; 32-bit total
5  start:
6  lea si, arr
7  mov cx, 3
8  again:
9  mov ax, [si]
10 cwd                    ; AX -> DX:AX
11 add [total], ax        ; low word
▶ adc [total+2], dx       ; high + CF
13 add si, 2
14 loop again
15 ret
```

FFFFh + 0000h + CF(1) = 10000h → high word **0000h**.
Total 0000C738h = **51000** ✓

Step 10 / 10

VARIABLES

AX = **F060h** DX = **FFFFh** CF = **1**

total = **C738h** total+2 = **0000h**

Signed word array: each element is widened to 32 bits with `CWD`; the low word is added with `ADD`, the high word with `ADC`.

Demo: Multiply / Divide Calculator

Pick an instruction, a size and values to see which registers receive the result. For division the dividend is AX for bytes and DX:AX for words.

instruction MUL size byte AL 200 operand 3

200 × 3 = **600**

AX = 0258h AH = 02h AL = 58h **CF = OF = 1**

The result spills into the upper half → CF = OF = 1.

32 × 32-Bit Multiplication: Partial Products

The 8086 multiplies at most 16 × 16 bits. 32-bit numbers are split into words: A × B = A1·B1·2^{32} + (A1·B0 + A0·B1)·2^{16} + A0·B0. Four `MUL` s; the results are added by word position with `ADD` / `ADC` ; the result is 64 bits.

	z3 · bit 48–63	z2 · bit 32–47	z1 · bit 16–31	z0 · bit 0–15
A =			9ABC (A1)	DEF0 (A0)
× B =			EDCB (B1)	FEDC (B0)

A = 9ABCDEF0h and B = EDCBFEDCh are split into 16-bit halves: A = A1:A0, B = B1:B0.

32 × 32-Bit Multiplication: Partial Products

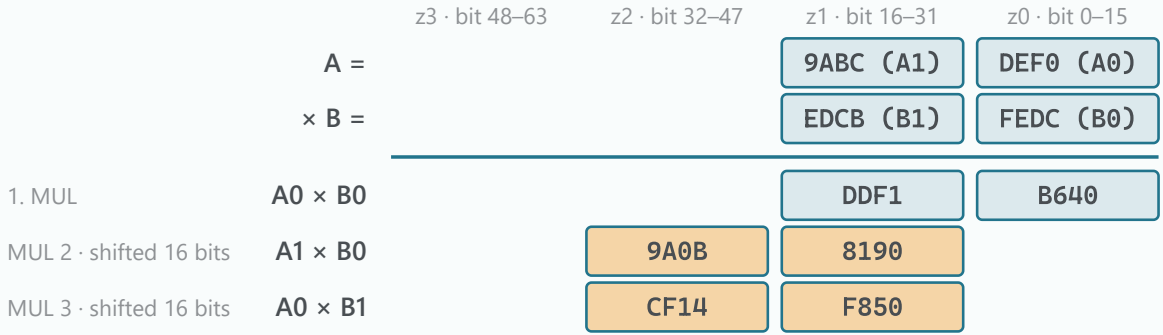
The 8086 multiplies at most 16 × 16 bits. 32-bit numbers are split into words: $A \times B = A1 \cdot B1 \cdot 2^{32} + (A1 \cdot B0 + A0 \cdot B1) \cdot 2^{16} + A0 \cdot B0$. Four `MUL` s; the results are added by word position with `ADD` / `ADC` ; the result is 64 bits.



A0 × B0 is not shifted: it lands in the two lowest words of the result (z1:z0).

32 × 32-Bit Multiplication: Partial Products

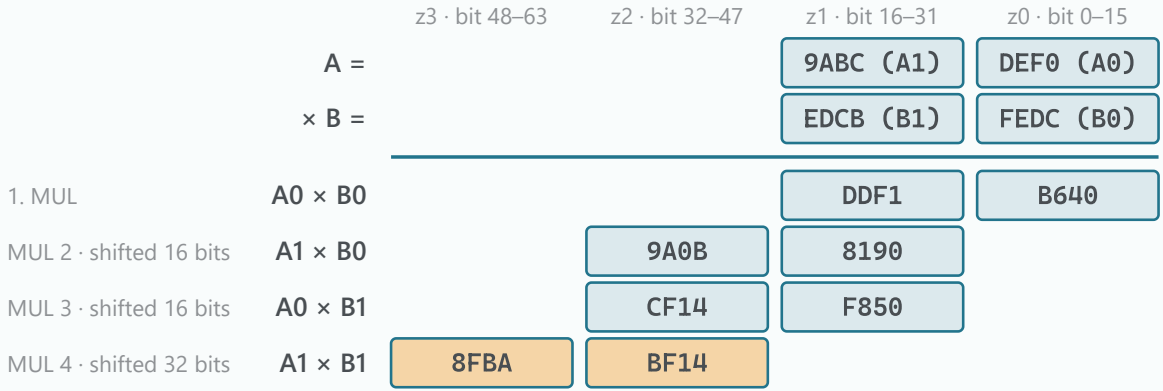
The 8086 multiplies at most 16 × 16 bits. 32-bit numbers are split into words: A × B = A1·B1·2^{32} + (A1·B0 + A0·B1)·2^{16} + A0·B0. Four `MUL` s; the results are added by word position with `ADD` / `ADC` ; the result is 64 bits.



A1 × B0 and A0 × B1 are both multiplied by 2^{16}: they go into columns z2:z1.

32 × 32-Bit Multiplication: Partial Products

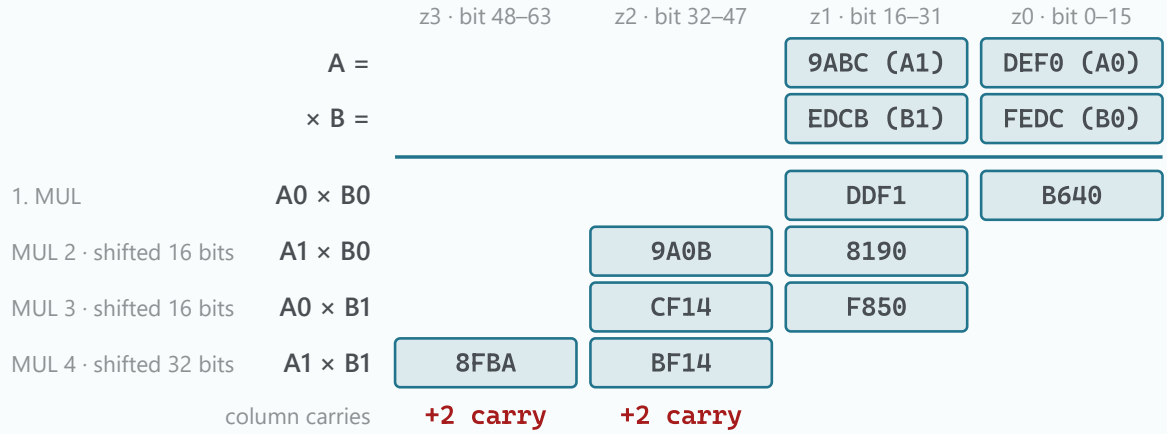
The 8086 multiplies at most 16 × 16 bits. 32-bit numbers are split into words: A × B = A1·B1·2^{32} + (A1·B0 + A0·B1)·2^{16} + A0·B0. Four `MUL` s; the results are added by word position with `ADD` / `ADC` ; the result is 64 bits.



A1 × B1 is multiplied by 2^{32}: it goes into columns z3:z2.

32 × 32-Bit Multiplication: Partial Products

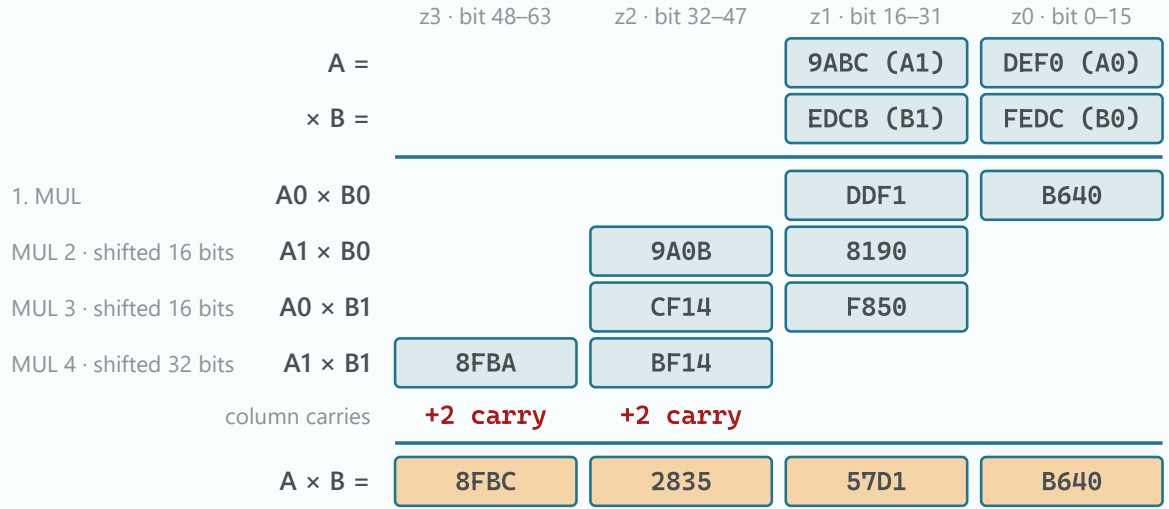
The 8086 multiplies at most 16 × 16 bits. 32-bit numbers are split into words: $A \times B = A1 \cdot B1 \cdot 2^{32} + (A1 \cdot B0 + A0 \cdot B1) \cdot 2^{16} + A0 \cdot B0$. Four `MUL` s; the results are added by word position with `ADD` / `ADC` ; the result is 64 bits.



Column sums: z1 = 257D1h → write 57D1h, **carry 2**; z2 = 22835h → write 2835h, **carry 2**. Each ADD gives at most one carry: hence the ADC chain in the code.

32 × 32-Bit Multiplication: Partial Products

The 8086 multiplies at most 16 × 16 bits. 32-bit numbers are split into words: A × B = A1·B1·2^{32} + (A1·B0 + A0·B1)·2^{16} + A0·B0. Four `MUL` s; the results are added by word position with `ADD` / `ADC` ; the result is 64 bits.



Result: **8FBC 2835 57D1 B640h**. Check (decimal): 2596069104 × 3989569244 = 10357197452617037376 ✓

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
▶ mov ax, [x]        ; A0
▶ mul [y]            ; A0*B0
4  mov [z], ax
5  mov [z+2], dx
6  mov ax, [x+2]     ; A1
7  mul [y]           ; A1*B0
8  add [z+2], ax
9  adc [z+4], dx
10 mov ax, [x]       ; A0
11 mul [y+2]         ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]     ; A1
16 mul [y+2]         ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

AX = A0 = DEF0h, `MUL [y]`: A0 × B0 = DEF0h × FEDCh = **DDF1:B640**h (DX:AX). DX ≠ 0, so MUL sets CF = 1.

Step 1 / 12

VARIABLES

AX = **B640h** | DX = **DDF1h** | CF = **1** | z3 = **0000h** | z2 = **0000h** | z1 = **0000h** | z0 = **0000h**

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
2  mov ax, [x]        ; A0
3  mul [y]            ; A0*B0
▶  mov [z], ax
▶  mov [z+2], dx
6  mov ax, [x+2]      ; A1
7  mul [y]            ; A1*B0
8  add [z+2], ax
9  adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

The first partial product is written to words z0 (bits 0–15) and z1 (bits 16–31).

Step 2 / 12

VARIABLES

AX = **B640h**	DX = **DDF1h**	CF = **1**	z3 = **0000h**
z2 = **0000h**	z1 = **DDF1h**	z0 = **B640h**	

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
2  mov ax, [x]        ; A0
3  mul [y]            ; A0*B0
4  mov [z], ax
5  mov [z+2], dx
▶  mov ax, [x+2]      ; A1
▶  mul [y]            ; A1*B0
8  add [z+2], ax
9  adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

A1 × B0 = 9ABCh × FEDCh = **9A0B:8190**h. Shifted 16 bits left: it is added into z1 and z2.

Step 3 / 12

VARIABLES

AX = **8190h** DX = **9A0Bh** CF = **1** z3 = **0000h**

z2 = **0000h** z1 = **DDF1h** z0 = **B640h**

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
2  mov ax, [x]        ; A0
3  mul [y]            ; A0*B0
4  mov [z], ax
5  mov [z+2], dx
6  mov ax, [x+2]      ; A1
7  mul [y]            ; A1*B0
▶  add [z+2], ax
9  adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

z1 + AX = DDF1h + 8190h → z1 = **5F81h**, CF = 1.

Step 4 / 12

VARIABLES

AX = **8190h**	DX = **9A0Bh**	CF = **1**	z3 = **0000h**
z2 = **0000h**	z1 = **5F81h**	z0 = **B640h**	

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
2  mov ax, [x]        ; A0
3  mul [y]            ; A0*B0
4  mov [z], ax
5  mov [z+2], dx
6  mov ax, [x+2]      ; A1
7  mul [y]            ; A1*B0
8  add [z+2], ax
▶  adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

z2 = 0 + DX + CF = **9A0Ch**. z2 was 0 before, so no new carry arises here.

Step 5 / 12

VARIABLES

AX = **8190h**	DX = **9A0Bh**	CF = **0**	z3 = **0000h**
z2 = **9A0Ch**	z1 = **5F81h**	z0 = **B640h**	

32 × 32-Bit Multiplication: Code Trace

```
 1 org 100h
 2 mov ax, [x]        ; A0
 3 mul [y]            ; A0*B0
 4 mov [z], ax
 5 mov [z+2], dx
 6 mov ax, [x+2]      ; A1
 7 mul [y]            ; A1*B0
 8 add [z+2], ax
 9 adc [z+4], dx
▶ mov ax, [x]        ; A0
▶ mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]     ; A1
16 mul [y+2]         ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

A0 × B1 = DEF0h × EDCBh = **CF14:F850**h. This too is added at the z1–z2 position.

Step 6 / 12

VARIABLES

AX = **F850h** DX = **CF14h** CF = **1** z3 = **0000h**

z2 = **9A0Ch** z1 = **5F81h** z0 = **B640h**

32 × 32-Bit Multiplication: Code Trace

```
 1 org 100h
 2 mov ax, [x]        ; A0
 3 mul [y]            ; A0*B0
 4 mov [z], ax
 5 mov [z+2], dx
 6 mov ax, [x+2]      ; A1
 7 mul [y]            ; A1*B0
 8 add [z+2], ax
 9 adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
▶ add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

z1 = 5F81h + F850h → **57D1h**, CF = 1.

Step 7 / 12

VARIABLES

AX = **F850h**	DX = **CF14h**	CF = **1**	z3 = **0000h**
z2 = **9A0Ch**	z1 = **57D1h**	z0 = **B640h**	

32 × 32-Bit Multiplication: Code Trace

```
 1 org 100h
 2 mov ax, [x]        ; A0
 3 mul [y]            ; A0*B0
 4 mov [z], ax
 5 mov [z+2], dx
 6 mov ax, [x+2]      ; A1
 7 mul [y]            ; A1*B0
 8 add [z+2], ax
 9 adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
▶ adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

z2 = 9A0Ch + CF14h + CF → **6921h**, CF = 1: the carry must go into z3!

Step 8 / 12

VARIABLES

AX = **F850h**	DX = **CF14h**	CF = **1**	z3 = **0000h**
z2 = **6921h**	z1 = **57D1h**	z0 = **B640h**	

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
2  mov ax, [x]        ; A0
3  mul [y]            ; A0*B0
4  mov [z], ax
5  mov [z+2], dx
6  mov ax, [x+2]      ; A1
7  mul [y]            ; A1*B0
8  add [z+2], ax
9  adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
▶ adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

`ADC [z+6], 0` : adds only the carry → z3 = **0001h**.

Step 9 / 12

VARIABLES

AX = **F850h** DX = **CF14h** CF = **0** z3 = **0001h**

z2 = **6921h** z1 = **57D1h** z0 = **B640h**

32 × 32-Bit Multiplication: Code Trace

```
1  org 100h
2  mov ax, [x]        ; A0
3  mul [y]            ; A0*B0
4  mov [z], ax
5  mov [z+2], dx
6  mov ax, [x+2]      ; A1
7  mul [y]            ; A1*B0
8  add [z+2], ax
9  adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
▶ mov ax, [x+2]      ; A1
▶ mul [y+2]          ; A1*B1
17 add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

A1 × B1 = 9ABCh × EDCBh = **8FBA:BF14**h, shifted 32 bits left: z2–z3.

Step 10 / 12

VARIABLES

AX = **BF14h** DX = **8FBAh** CF = **1** z3 = **0001h**

z2 = **6921h** z1 = **57D1h** z0 = **B640h**

32 × 32-Bit Multiplication: Code Trace

```
 1 org 100h
 2 mov ax, [x]        ; A0
 3 mul [y]            ; A0*B0
 4 mov [z], ax
 5 mov [z+2], dx
 6 mov ax, [x+2]      ; A1
 7 mul [y]            ; A1*B0
 8 add [z+2], ax
 9 adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
▶ add [z+4], ax
18 adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

z2 = 6921h + BF14h → **2835h**, CF = 1.

Step 11 / 12

VARIABLES

AX = **BF14h**	DX = **8FBAh**	CF = **1**	z3 = **0001h**
z2 = **2835h**	z1 = **57D1h**	z0 = **B640h**	

32 × 32-Bit Multiplication: Code Trace

```
 1 org 100h
 2 mov ax, [x]        ; A0
 3 mul [y]            ; A0*B0
 4 mov [z], ax
 5 mov [z+2], dx
 6 mov ax, [x+2]      ; A1
 7 mul [y]            ; A1*B0
 8 add [z+2], ax
 9 adc [z+4], dx
10 mov ax, [x]        ; A0
11 mul [y+2]          ; A0*B1
12 add [z+2], ax
13 adc [z+4], dx
14 adc [z+6], 0
15 mov ax, [x+2]      ; A1
16 mul [y+2]          ; A1*B1
17 add [z+4], ax
▶ adc [z+6], dx
19 ret
20 x dw 0DEF0h, 9ABCh  ; A = 9ABCDEF0h
21 y dw 0FEDCh, 0EDCBh ; B = EDCBFEDCh
22 z dw 4 dup(0)       ; 64-bit result
```

z3 = 0001h + 8FBAh + CF = **8FBCh**. Result z3:z2:z1:z0 = **8FBC 2835 57D1 B640h** ✓

Step 12 / 12

VARIABLES

AX = **BF14h**	DX = **8FBAh**	CF = **0**	z3 = **8FBCh**	z2 = **2835h**	z1 = **57D1h**	z0 = **B640h**

GROUP 3: ONE OPERAND

INC, DEC, NOT, NEG

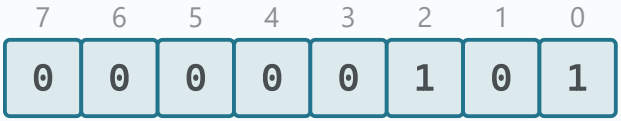
- Operands: **REG** (AX, BX, … SP) or **memory** (`[BX]` , `[BX+SI+7]` , a variable). No immediates.
- `INC` and `DEC` affect only **ZF, SF, OF, PF, AF**; they **leave CF alone**.
- `NOT` affects no flags at all! It inverts every bit of the operand (ones' complement).
- `NEG` affects CF, ZF, SF, OF, PF, AF. It negates the operand (**two's complement**): invert every bit, then add 1. E.g. 5 → −5, −2 → 2.

Instruction	CF	ZF	SF	OF	PF	AF
`INC` , `DEC`	—	✓	✓	✓	✓	✓
`NOT`	—	—	—	—	—	—
`NEG`	✓ (1 if operand ≠ 0)	✓	✓	✓	✓	✓

—: unchanged. Because INC/DEC preserve CF, a loop can bump its counter while doing multi-byte addition with ADC.

NEG: Two's Complement Step by Step

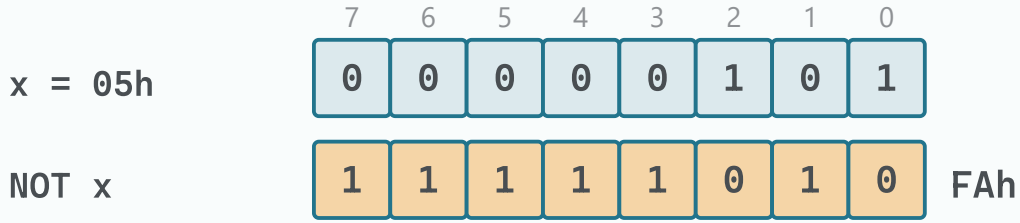
`x = 05h`



x = 5 = 0000 0101b.

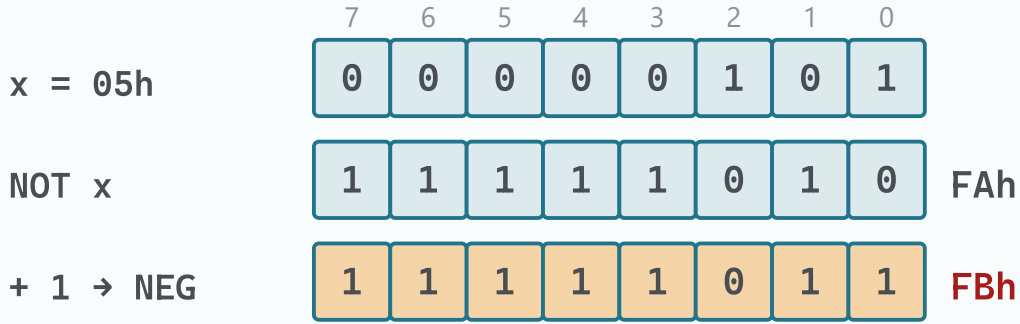
Step 1 / 6

NEG: Two's Complement Step by Step



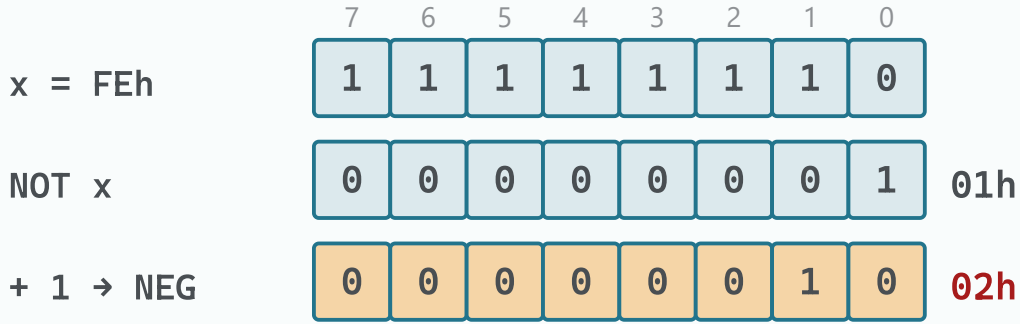
NOT: every bit is inverted → 1111 1010b = FAh (−6). Ones' complement.

NEG: Two's Complement Step by Step



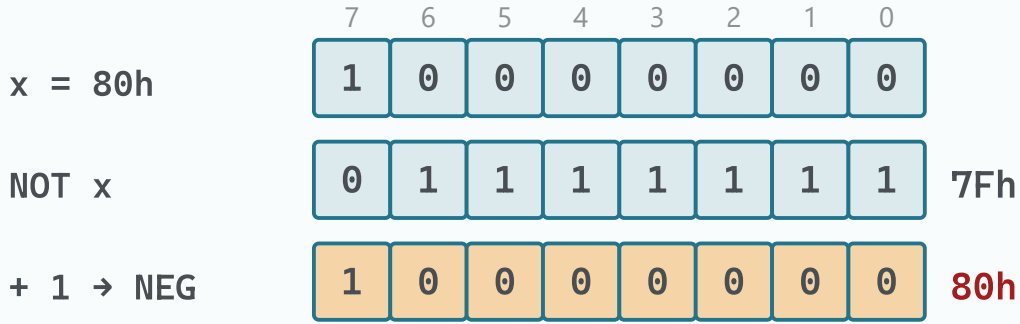
+1: FBh = 1111 1011b = **−5**. `NEG` does both steps in one instruction; CF = 1 (x ≠ 0).

NEG: Two's Complement Step by Step



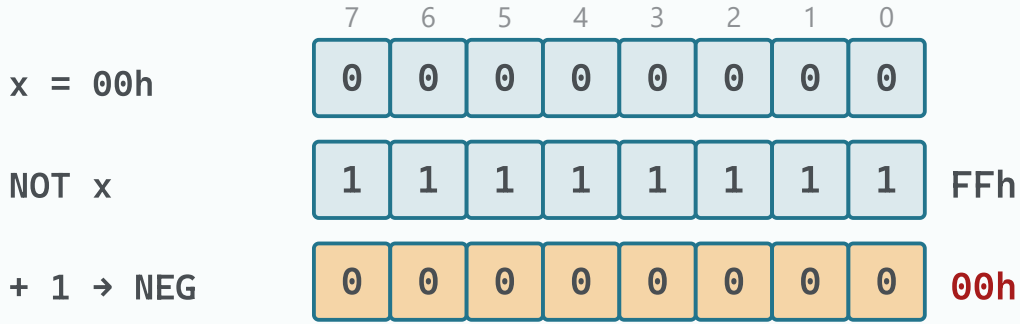
x = FEh (−2): NOT → 01h, +1 → **02h**. The negative of a negative is positive.

NEG: Two's Complement Step by Step



Special case: x = 80h (−128). NOT → 7Fh, +1 → **80h**: +128 does not fit in a byte → **OF = 1**.

NEG: Two's Complement Step by Step



x = 0: the result is 0, **CF = 0** (the only case where NEG leaves CF = 0), ZF = 1.

Code Trace: INC, DEC, NOT, NEG

```
▶ mov al, 0FFh
2 inc al          ; FFh + 1
3 dec al          ; 00h - 1
4 mov bl, 7Fh     ; +127
5 inc bl          ; +128 ?
6 not bl
7 neg bl
```

AL = FFh (255 / −1).

Step 1 / 7

VARIABLES

AL = **FFh** | BL = **00h** | CF = **0** | ZF = **0** | SF = **0** | OF = **0** | PF = **0** | AF = **0**

Code Trace: INC, DEC, NOT, NEG

```
1 mov al, 0FFh
▶ inc al          ; FFh + 1
3 dec al          ; 00h - 1
4 mov bl, 7Fh     ; +127
5 inc bl          ; +128 ?
6 not bl
7 neg bl
```

FFh + 1 = **00h**, ZF = 1. ADD would have set CF = 1; **INC leaves CF alone** → CF stays 0.

Step 2 / 7

VARIABLES

AL = 00h | BL = 00h | CF = 0 | ZF = 1 | SF = 0 | OF = 0 | PF = 1 | AF = 1

Code Trace: INC, DEC, NOT, NEG

```
1 mov al, 0FFh
2 inc al          ; FFh + 1
▶ dec al          ; 00h - 1
4 mov bl, 7Fh     ; +127
5 inc bl          ; +128 ?
6 not bl
7 neg bl
```

00h − 1 = **FFh**, SF = 1. There was a borrow, but CF still did not change.

Step 3 / 7

VARIABLES

AL = **FFh** | BL = **00h** | CF = **0** | ZF = **0**

SF = **1** | OF = **0** | PF = **1** | AF = **1**

Code Trace: INC, DEC, NOT, NEG

```
1 mov al, 0FFh
2 inc al          ; FFh + 1
3 dec al          ; 00h - 1
▶ mov bl, 7Fh     ; +127
5 inc bl          ; +128 ?
6 not bl
7 neg bl
```

BL = 7Fh = +127, the largest positive byte.

Step 4 / 7

VARIABLES

AL = **FFh** | BL = **7Fh** | CF = **0** | ZF = **0**

SF = **1** | OF = **0** | PF = **1** | AF = **1**

Code Trace: INC, DEC, NOT, NEG

```
1 mov al, 0FFh
2 inc al          ; FFh + 1
3 dec al          ; 00h - 1
4 mov bl, 7Fh     ; +127
▶ inc bl          ; +128 ?
6 not bl
7 neg bl
```

7Fh + 1 = **80h** = −128: signed overflow → **OF = 1**, SF = 1.

Step 5 / 7

VARIABLES

AL = **FFh** | BL = **80h** | CF = **0** | ZF = **0** | SF = **1** | OF = **1** | PF = **0** | AF = **1**

Code Trace: INC, DEC, NOT, NEG

```
1 mov al, 0FFh
2 inc al          ; FFh + 1
3 dec al          ; 00h - 1
4 mov bl, 7Fh     ; +127
5 inc bl          ; +128 ?
▶ not bl
7 neg bl
```

`NOT` : 80h → **7Fh**. The flags **did not change at all** (SF still shows 1).

Step 6 / 7

VARIABLES

AL = **FFh** | BL = **7Fh** | CF = **0** | ZF = **0** | SF = **1** | OF = **1** | PF = **0** | AF = **1**

Code Trace: INC, DEC, NOT, NEG

```
1 mov al, 0FFh
2 inc al          ; FFh + 1
3 dec al          ; 00h - 1
4 mov bl, 7Fh     ; +127
5 inc bl          ; +128 ?
6 not bl
▶ neg bl
```

`NEG` : −127 = **81h**. CF = 1 (operand ≠ 0), SF = 1, OF = 0.

Step 7 / 7

VARIABLES

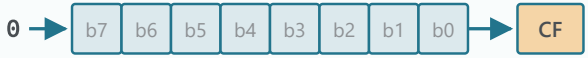
AL = **FFh** | BL = **81h** | CF = **1** | ZF = **0**

SF = **1** | OF = **0** | PF = **1** | AF = **1**

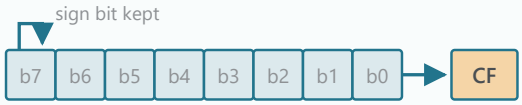
Shifts: SHL/SAL, SHR, SAR



SHL/SAL: shift left, 0 enters from the right, the bit shifted out goes to CF. Each step is × **2**.



SHR: logical shift right, 0 enters from the left. Unsigned ÷ **2**.



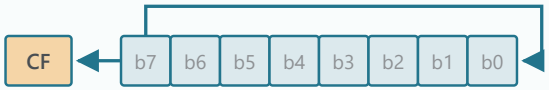
SAR: arithmetic shift right, the sign bit is kept. Signed ÷ **2** (rounds toward −∞).

Shift count

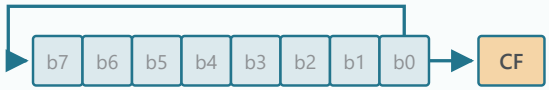
On the 8086 only `1` or `CL`: `shl ax, 1`, `shr al, cl`. An immediate count such as `shl al, 4` is valid from the 80186 on. OF is defined only when the count is 1.

SAR and IDIV round differently: −7 (F9h) `sar` 1 → FCh = **−4**; `idiv` 2 → **−3**.

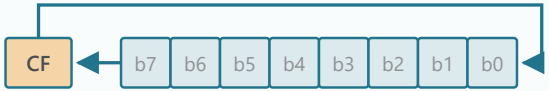
Rotates: ROL, ROR, RCL, RCR



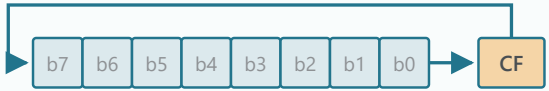
ROL: rotate left; b7 goes to both b0 and CF.



ROR: rotate right; b0 goes to both b7 and CF.



RCL: rotate left through CF (a 9-bit ring).



RCR: rotate right through CF.

No bit is lost in a rotate. Only **CF** and (when the count is 1) **OF** change; SF, ZF, PF are not affected. RCL/RCR are used for multi-word shifts: `shl ax, 1` + `rcl dx, 1` → DX:AX × 2.

Shift and Rotate Explorer

AL = B4 **SHL** **SHR** **SAR** **ROL** **ROR** **RCL** **RCR** **flip CF**

AL **1 0 1 1 0 1 0 0** `B4h = 180 / -76`

`CF = 0`

AL = **B4h**. Pick an instruction button to see the result here.

Each button applies the instruction as `op al, 1`. Press SHL repeatedly and the number doubles at each step; SAR keeps the sign bit.

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
▶  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

The multiplicand BL = 13 = **0Dh** (0000 1101b).

Step 1 / 22

VARIABLES

BL = **0Dh** DL = **00h** CX = **0000h** CF = **0**

result = **00h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
▶  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

The multiplier DL = 5 = **0101b**: as in long multiplication on paper, its bits are examined one by one.

Step 2 / 22

VARIABLES

BL = **0Dh** DL = **05h** CX = **0000h** CF = **0**

result = **00h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
▶  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

4 passes for a 4-bit multiplier.

Step 3 / 22

VARIABLES

BL = **0Dh** DL = **05h** CX = **0004h** CF = **0**

result = **00h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
▶  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

`shr dl, 1` : the bit shifted out is 1 → **CF = 1**.

Step 4 / 22

VARIABLES

BL = **0Dh** DL = **02h** CX = **0004h** CF = **1**

result = **00h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
▶  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

CF = 1 → no jump; BL is added.

Step 5 / 22

VARIABLES

BL = **0Dh** DL = **02h** CX = **0004h** CF = **1**

result = **00h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
▶  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

result = 0 + 0Dh = **0Dh** (13 × 1).

Step 6 / 22

VARIABLES

BL = **0Dh** DL = **02h** CX = **0004h** CF = **0**

result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
▶  shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

The multiplicand moves one place left: BL = **1Ah** (26).

Step 7 / 22

VARIABLES

BL = **1Ah** DL = **02h** CX = **0004h** CF = **0**

result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
▶ loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

`loop` : CX = 0003h.

Step 8 / 22

VARIABLES

BL = **1Ah** | DL = **02h** | CX = **0003h** | CF = **0** | result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
▶  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

The bit shifted out is 0 → **CF = 0**.

Step 9 / 22

VARIABLES

BL = **1Ah** | DL = **01h** | CX = **0003h** | CF = **0** | result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
▶  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

CF = 0 → jumps to `skip`.

Step 10 / 22

VARIABLES

BL = **1Ah** DL = **01h** CX = **0003h** CF = **0**

result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
 1 org 100h
 2 mov bl, mcand      ; 13
 3 mov dl, mplier     ; 5 = 0101b
 4 mov cx, 4          ; 4 bits
 5 check:
 6 shr dl, 1          ; low bit → CF
 7 jnc skip
 8 add result, bl     ; add if bit is 1
 9 skip:
▶ shl bl, 1           ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

BL = **34h** (52).

Step 11 / 22

VARIABLES

BL = **34h** DL = **01h** CX = **0003h** CF = **0**

result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
▶ loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

`loop` : CX = 0002h.

Step 12 / 22

VARIABLES

BL = **34h** | DL = **01h** | CX = **0002h** | CF = **0** | result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
▶  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

The bit shifted out is 1 → **CF = 1**.

Step 13 / 22

VARIABLES

BL = **34h** DL = **00h** CX = **0002h** CF = **1**

result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
▶  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

CF = 1 → BL is added.

Step 14 / 22

VARIABLES

BL = **34h** DL = **00h** CX = **0002h** CF = **1**

result = **0Dh**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
▶  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

result = 0Dh + 34h = **41h = 65** (13 × 4 added).

Step 15 / 22

VARIABLES

BL = **34h** DL = **00h** CX = **0002h** CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
 1 org 100h
 2 mov bl, mcand      ; 13
 3 mov dl, mplier     ; 5 = 0101b
 4 mov cx, 4          ; 4 bits
 5 check:
 6 shr dl, 1          ; low bit → CF
 7 jnc skip
 8 add result, bl     ; add if bit is 1
 9 skip:
▶10 shl bl, 1         ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

BL = **68h** (104).

Step 16 / 22

VARIABLES

BL = **68h** DL = **00h** CX = **0002h** CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
 1 org 100h
 2 mov bl, mcand      ; 13
 3 mov dl, mplier     ; 5 = 0101b
 4 mov cx, 4          ; 4 bits
 5 check:
 6 shr dl, 1          ; low bit → CF
 7 jnc skip
 8 add result, bl     ; add if bit is 1
 9 skip:
10 shl bl, 1          ; mcand × 2
▶ loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

`loop` : CX = 0001h.

Step 17 / 22

VARIABLES

BL = **68h** DL = **00h** CX = **0001h** CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
▶  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

The bit shifted out is 0 → CF = 0.

Step 18 / 22

VARIABLES

BL = **68h** DL = **00h** CX = **0001h** CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
▶  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

CF = 0 → skip.

Step 19 / 22

VARIABLES

BL = **68h** DL = **00h** CX = **0001h** CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
 1 org 100h
 2 mov bl, mcand      ; 13
 3 mov dl, mplier     ; 5 = 0101b
 4 mov cx, 4          ; 4 bits
 5 check:
 6 shr dl, 1          ; low bit → CF
 7 jnc skip
 8 add result, bl     ; add if bit is 1
 9 skip:
▶10 shl bl, 1         ; mcand × 2
11 loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

BL = **D0h**.

Step 20 / 22

VARIABLES

BL = **D0h** DL = **00h** CX = **0001h** CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
▶ loop check
12 ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

`loop` : CX = 0000h.

Step 21 / 22

VARIABLES

BL = **D0h** | DL = **00h** | CX = **0000h** | CF = **0**

result = **41h**

Code Trace: Multiplying by Shifting (13 × 5)

```
1  org 100h
2  mov bl, mcand      ; 13
3  mov dl, mplier     ; 5 = 0101b
4  mov cx, 4          ; 4 bits
5  check:
6  shr dl, 1          ; low bit → CF
7  jnc skip
8  add result, bl     ; add if bit is 1
9  skip:
10 shl bl, 1          ; mcand × 2
11 loop check
▶ ret
13 mcand db 13
14 mplier   db 5
15 result    db 0
```

Result: 13 × 5 = **65 = 41h**. This is how CPUs without MUL multiply.

Step 22 / 22

VARIABLES

`BL = D0h` `DL = 00h` `CX = 0000h` `CF = 0`

`result = 41h`

Fast Multiplication and Division by Shifting

```
; AX ← AX × 3 = AX×2 + AX
mov bx, ax      ; save
shl ax, 1       ; AX × 2
add ax, bx      ; ×2 + ×1

; unsigned ÷ 8 and remainder
mov dx, ax
and dx, 0007h   ; rem = low 3 bits
mov cl, 3
shr ax, cl      ; quotient

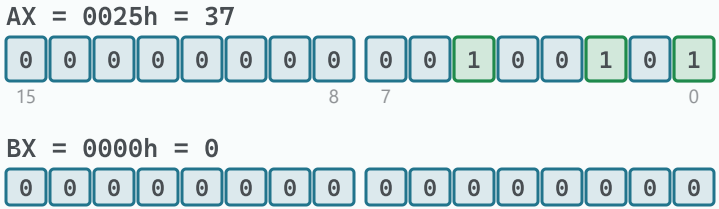
; signed ÷ 4
mov cl, 2
sar ax, cl      ; −7 → −2
```

- On the 8086 `mul` takes 70–133 and `div` 80–162 clocks; a `shl reg, 1` takes just **2**.
- Multiplying/dividing by powers of 2 becomes a shift; other constants become shifts plus additions.
- Remainder of division by 2^n: the low n bits, via `and`.
- Signed division uses SAR; for a negative dividend SAR rounds toward $-\infty$ while IDIV rounds toward zero: for −7, SAR 2 → **−2**, IDIV 4 → **−1**.

Example: AX = 25 → BX = 25, AX = 50 → 50 + 25 = **75**.

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
▶ mov ax, 37
3  shl ax, 1        ; AX × 2
4  mov bx, ax       ; save
5  shl ax, 1        ; AX × 4
6  shl ax, 1        ; AX × 8
7  add ax, bx       ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh    ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl       ; quotient: ÷ 16
12 ret
```



AX = 37 = **0025h** = 0000 0000 0010 0101b.

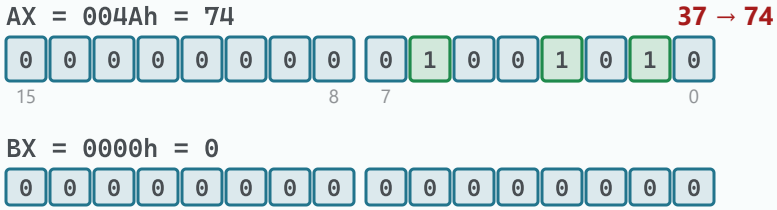
Step 1 / 11

VARIABLES

AX = **0025h (37)** BX = **0000h** CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
▶  shl ax, 1        ; AX × 2
4  mov bx, ax       ; save
5  shl ax, 1        ; AX × 4
6  shl ax, 1        ; AX × 8
7  add ax, bx       ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh    ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl       ; quotient: ÷ 16
12 ret
```



SHL AX, 1: bits move one place left, 0 enters from the right → AX = **74** (004Ah). 2 clocks; MUL takes more than 70.

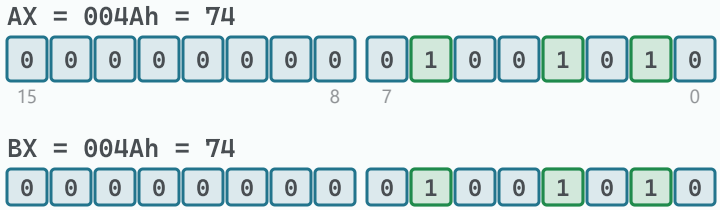
Step 2 / 11

VARIABLES

AX = **004Ah (74)** BX = **0000h** CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1       ; AX × 2
4  mov bx, ax      ; save
5  shl ax, 1       ; AX × 4
6  shl ax, 1       ; AX × 8
7  add ax, bx      ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh   ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl      ; quotient: ÷ 16
12 ret
```



BX = 74: ×2 is saved.

Step 3 / 11

VARIABLES

AX = 004Ah (74) | BX = 004Ah | CL = 00h

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1        ; AX × 2
4  mov bx, ax       ; save
▶  shl ax, 1        ; AX × 4
6  shl ax, 1        ; AX × 8
7  add ax, bx       ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh    ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl       ; quotient: ÷ 16
12 ret
```

AX = 0094h = 148 — 74 → 148



BX = 004Ah = 74



SHL → AX = **148** (×4).

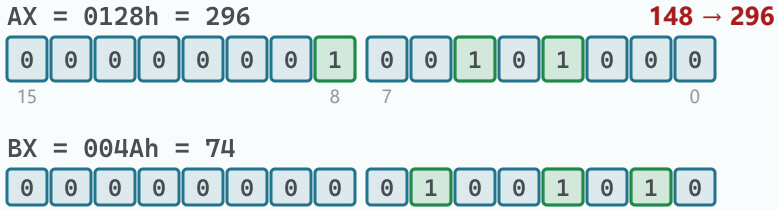
Step 4 / 11

VARIABLES

AX = **0094h (148)** | BX = **004Ah** | CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1        ; AX × 2
4  mov bx, ax       ; save
5  shl ax, 1        ; AX × 4
▶  shl ax, 1        ; AX × 8
7  add ax, bx       ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh    ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl       ; quotient: ÷ 16
12 ret
```



SHL → AX = **296** (×8).

Step 5 / 11

VARIABLES

AX = **0128h (296)** BX = **004Ah** CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1       ; AX × 2
4  mov bx, ax      ; save
5  shl ax, 1       ; AX × 4
6  shl ax, 1       ; AX × 8
▶  add ax, bx      ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh   ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl      ; quotient: ÷ 16
12 ret
```

AX = 0172h = 370 — 296 → 370



BX = 004Ah = 74



ADD AX, BX: 296 + 74 = **370** = 0172h = 37 × 10.

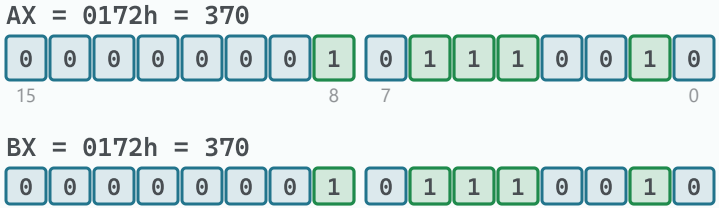
Step 6 / 11

VARIABLES

AX = **0172h (370)** | BX = **004Ah** | CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1       ; AX × 2
4  mov bx, ax      ; save
5  shl ax, 1       ; AX × 4
6  shl ax, 1       ; AX × 8
7  add ax, bx      ; ×8 + ×2
▶  mov bx, ax
9  and bx, 000Fh   ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl      ; quotient: ÷ 16
12 ret
```



BX = 370: a copy for the division.

Step 7 / 11

VARIABLES

AX = **0172h (370)** BX = **0172h** CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1       ; AX × 2
4  mov bx, ax      ; save
5  shl ax, 1       ; AX × 4
6  shl ax, 1       ; AX × 8
7  add ax, bx      ; ×8 + ×2
8  mov bx, ax
▶  and bx, 000Fh   ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl      ; quotient: ÷ 16
12 ret
```

AX = 0172h = 370



BX = 0002h = 2



AND BX, 000Fh: low 4 bits = 370 mod 16 = **2**.

Step 8 / 11

VARIABLES

AX = **0172h (370)** BX = **0002h** CL = **00h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1        ; AX × 2
4  mov bx, ax       ; save
5  shl ax, 1        ; AX × 4
6  shl ax, 1        ; AX × 8
7  add ax, bx       ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh    ; rem: low 4 bits
▶  mov cl, 4
11 shr ax, cl       ; quotient: ÷ 16
12 ret
```

AX = 0172h = 370



BX = 0002h = 2



CL = 4: on the 8086 a shift count above 1 goes in CL.

Step 9 / 11

VARIABLES

AX = **0172h (370)** BX = **0002h** CL = **04h**

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1       ; AX × 2
4  mov bx, ax      ; save
5  shl ax, 1       ; AX × 4
6  shl ax, 1       ; AX × 8
7  add ax, bx      ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh   ; rem: low 4 bits
10 mov cl, 4
▶ shr ax, cl       ; quotient: ÷ 16
12 ret
```

AX = 0017h = 23 — 370 → 23



BX = 0002h = 2



SHR AX, CL: 4 bits right = ÷ 16 → AX = **23** (0017h). 23 × 16 + 2 = 370.

Step 10 / 11

VARIABLES

AX = 0017h (23) | BX = 0002h | CL = 04h

37 × 10 and 370 ÷ 16 by Shifting

```
1  org 100h
2  mov ax, 37
3  shl ax, 1       ; AX × 2
4  mov bx, ax      ; save
5  shl ax, 1       ; AX × 4
6  shl ax, 1       ; AX × 8
7  add ax, bx      ; ×8 + ×2
8  mov bx, ax
9  and bx, 000Fh   ; rem: low 4 bits
10 mov cl, 4
11 shr ax, cl      ; quotient: ÷ 16
▶  ret
```

AX = 0017h = 23



BX = 0002h = 2



RET: the program ends.

Step 11 / 11

VARIABLES

AX = **0017h (23)** BX = **0002h** CL = **04h**

EXAMPLE: ADDING ARRAYS

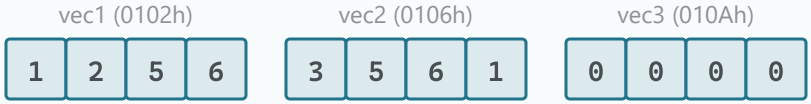
Adding Arrays

```
org 100h
jmp start
vec1 db 1, 2, 5, 6
vec2 db 3, 5, 6, 1
vec3 db ?, ?, ?, ?
start:
lea si, vec1
lea bx, vec2
lea di, vec3
mov cx, 4
sum:
mov al, [si]
add al, [bx]
mov [di], al
inc si
inc bx
inc di
loop sum
ret
```

- `vec3[i] = vec1[i] + vec2[i]`, i = 0…3.
- `JMP start` jumps over the data definitions; otherwise the CPU would take the data bytes for instructions.
- SI, BX, DI are **pointers** to the three arrays; `LEA` loads the address (offset).
- `[SI]`: the memory byte SI points to (indirect addressing).
- `LOOP sum`: decrements CX and goes back to `sum` while CX ≠ 0 → 4 passes.
- Expected result: vec3 = 4, 7, 11, 7.

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



`JMP start` : the data (0102h–010Dh) is skipped; execution goes to 010Eh.

Step 1 / 17

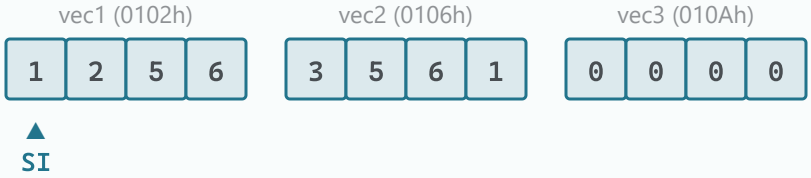
VARIABLES

SI = 0000h | BX = 0000h | DI = 0000h | CX = 0000h

AL = 00h | CF = 0 | ZF = 0

Code Trace: Adding Arrays

```
1  org 100h
2  jmp start
3  vec1 db 1, 2, 5, 6
4  vec2 db 3, 5, 6, 1
5  vec3 db ?, ?, ?, ?
6  start:
▶  lea si, vec1
8  lea bx, vec2
9  lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



SI = address of vec1 = **0102h**.

Step 2 / 17

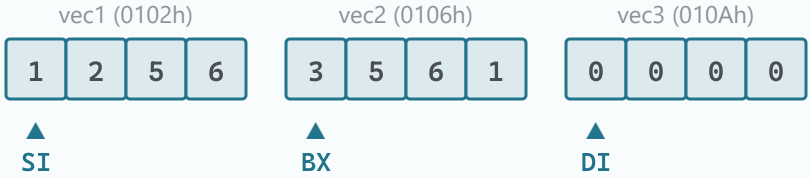
VARIABLES

SI = **0102h** | BX = **0000h** | DI = **0000h** | CX = **0000h**

AL = **00h** | CF = **0** | ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
▶ lea bx, vec2
▶ lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



BX = 0106h (vec2), DI = 010Ah (vec3).

Step 3 / 17

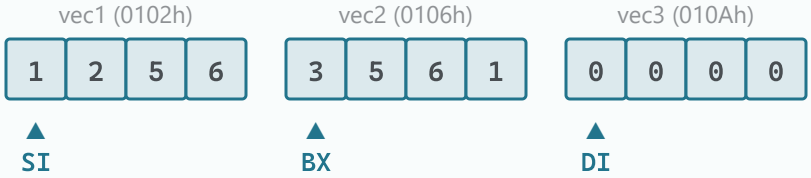
VARIABLES

SI = **0102h** BX = **0106h** DI = **010Ah** CX = **0000h**

AL = **00h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
▶10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



CX = 4 passes.

Step 4 / 17

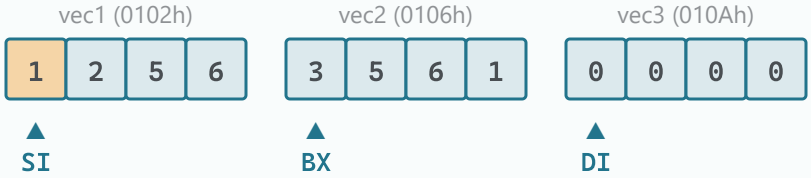
VARIABLES

SI = 0102h | BX = 0106h | DI = 010Ah | CX = 0004h

AL = 00h | CF = 0 | ZF = 0

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
▶ mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



Pass 1: AL ← [SI] = [0102h] = **1**.

Step 5 / 17

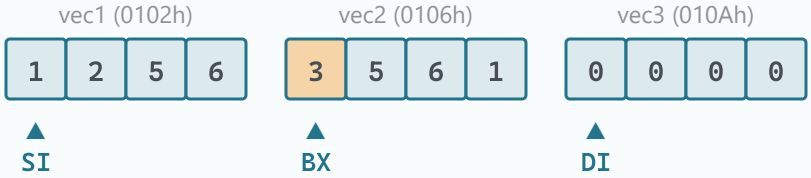
VARIABLES

SI = **0102h** BX = **0106h** DI = **010Ah** CX = **0004h**

AL = **01h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
▶ add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



AL ← AL + [BX] = 1 + 3 = **4** (04h).

Step 6 / 17

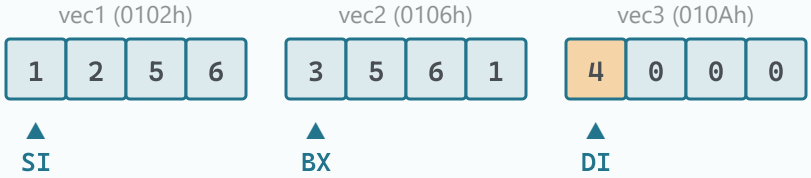
VARIABLES

SI = **0102h** BX = **0106h** DI = **010Ah** CX = **0004h**

AL = **04h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
▶ mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



[DI] ← AL: vec3[0] = **4**.

Step 7 / 17

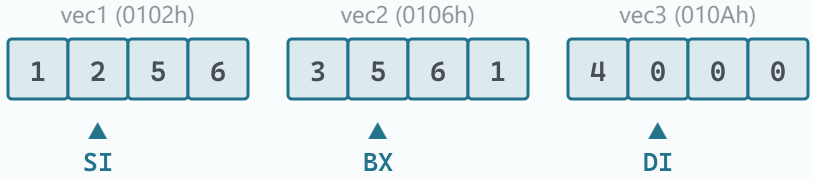
VARIABLES

SI = **0102h** BX = **0106h** DI = **010Ah** CX = **0004h**

AL = **04h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
1  org 100h
2  jmp start
3  vec1 db 1, 2, 5, 6
4  vec2 db 3, 5, 6, 1
5  vec3 db ?, ?, ?, ?
6  start:
7  lea si, vec1
8  lea bx, vec2
9  lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
   inc si
   inc bx
   inc di
18 loop sum
19 ret
```



All three pointers advance one byte (these are byte arrays; for word arrays, add 2).

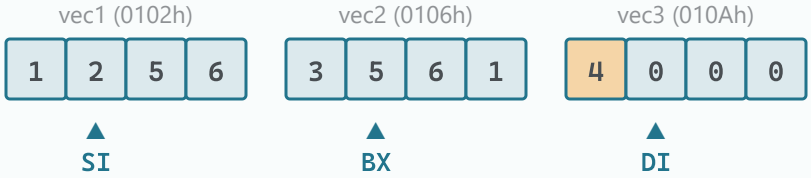
Step 8 / 17

VARIABLES

SI = 0103h | BX = 0107h | DI = 010Bh | CX = 0004h | AL = 04h | CF = 0 | ZF = 0

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
▶ loop sum
19 ret
```



`LOOP` : CX = 3 ≠ 0 → back to `sum` .

Step 9 / 17

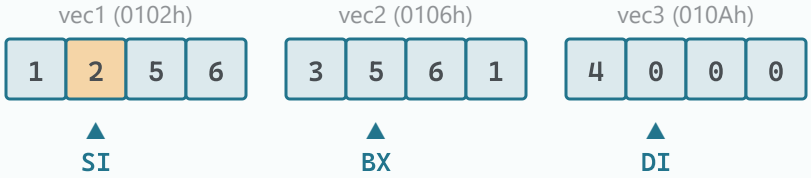
VARIABLES

`SI = 0103h` `BX = 0107h` `DI = 010Bh` `CX = 0003h`

`AL = 04h` `CF = 0` `ZF = 0`

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
▶ mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



Pass 2: AL ← [SI] = [0103h] = **2**.

Step 10 / 17

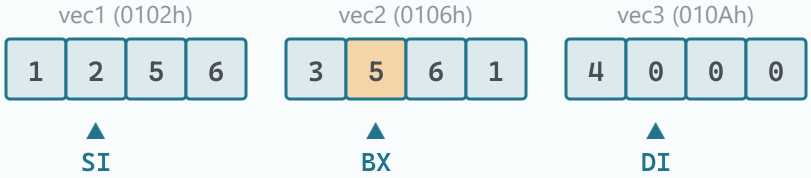
VARIABLES

SI = **0103h** BX = **0107h** DI = **010Bh** CX = **0003h**

AL = **02h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
▶ add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



AL ← AL + [BX] = 2 + 5 = **7** (07h).

Step 11 / 17

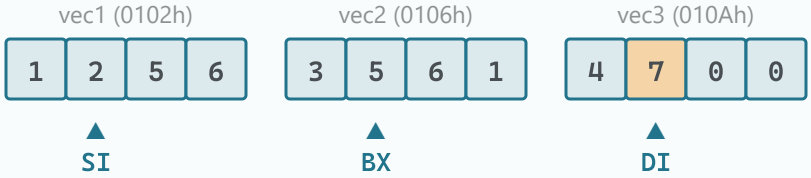
VARIABLES

SI = **0103h** BX = **0107h** DI = **010Bh** CX = **0003h**

AL = **07h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
▶ mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



[DI] ← AL: vec3[1] = **7**.

Step 12 / 17

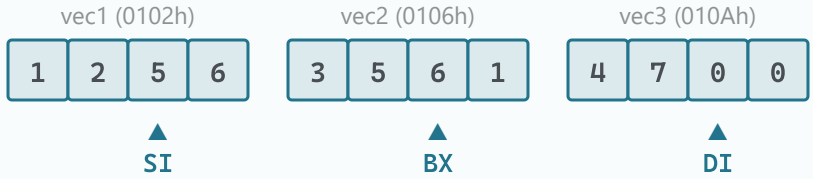
VARIABLES

SI = **0103h** BX = **0107h** DI = **010Bh** CX = **0003h**

AL = **07h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
   inc si
   inc bx
   inc di
18 loop sum
19 ret
```



All three pointers advance one byte (these are byte arrays; for word arrays, add 2).

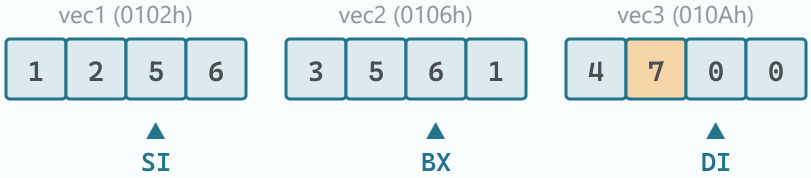
Step 13 / 17

VARIABLES

SI = 0104h | BX = 0108h | DI = 010Ch | CX = 0003h | AL = 07h | CF = 0 | ZF = 0

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
19 ret
```



`LOOP` : CX = 2 ≠ 0 → back to `sum`.

Step 14 / 17

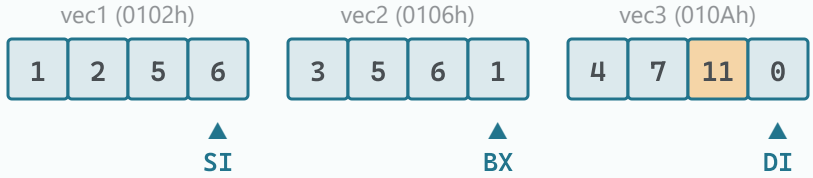
VARIABLES

SI = **0104h** BX = **0108h** DI = **010Ch** CX = **0002h**

AL = **07h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
▶ loop sum
19 ret
```



Pass 3 (summary): 5 + 6 = **11** → vec3[2]. `LOOP` : CX = 1 ≠ 0 → back to `sum` .

Step 15 / 17

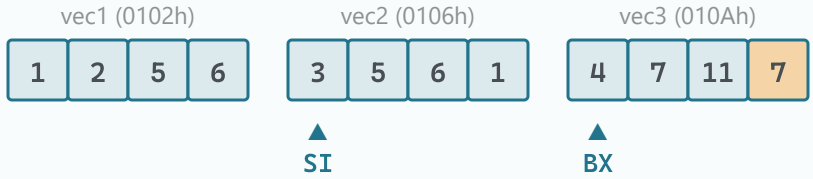
VARIABLES

`SI = 0105h` `BX = 0109h` `DI = 010Dh` `CX = 0001h`

`AL = 0Bh` `CF = 0` `ZF = 0`

Code Trace: Adding Arrays

```
 1 org 100h
 2 jmp start
 3 vec1 db 1, 2, 5, 6
 4 vec2 db 3, 5, 6, 1
 5 vec3 db ?, ?, ?, ?
 6 start:
 7 lea si, vec1
 8 lea bx, vec2
 9 lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
▶ loop sum
19 ret
```



Pass 4 (summary): 6 + 1 = **7** → vec3[3]. `LOOP` : CX = 0 → the loop ends.

Step 16 / 17

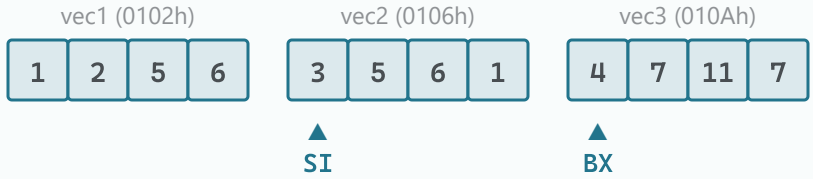
VARIABLES

SI = **0106h** BX = **010Ah** DI = **010Eh** CX = **0000h**

AL = **07h** CF = **0** ZF = **0**

Code Trace: Adding Arrays

```
1  org 100h
2  jmp start
3  vec1 db 1, 2, 5, 6
4  vec2 db 3, 5, 6, 1
5  vec3 db ?, ?, ?, ?
6  start:
7  lea si, vec1
8  lea bx, vec2
9  lea di, vec3
10 mov cx, 4
11 sum:
12 mov al, [si]
13 add al, [bx]
14 mov [di], al
15 inc si
16 inc bx
17 inc di
18 loop sum
▶  ret
```



`RET` : vec3 = **4, 7, 11, 7** (04h 07h 0Bh 07h).

Step 17 / 17

VARIABLES

`SI = 0106h` `BX = 010Ah` `DI = 010Eh` `CX = 0000h`

`AL = 07h` `CF = 0` `ZF = 0`

EXAMPLE: ADDING ARRAYS

Program and Data in Memory

Address	Bytes	Meaning
`0100`	`EB 0C`	`JMP 010Eh` +0Ch: 0102h + 0Ch
`0102`	`01 02 05 06`	`vec1` data
`0106`	`03 05 06 01`	`vec2` data
`010A`	`04 07 0B 07`	`vec3` after the run
`010E`	`BE 02 01`	`MOV SI, 0102h` LEA → MOV
`0111`	`BB 06 01`	`MOV BX, 0106h`
`0114`	`BF 0A 01`	`MOV DI, 010Ah`
`0117`	`B9 04 00`	`MOV CX, 0004h` little-endian
`011A`	`8A 04`	`MOV AL, [SI]` sum:
`011C`	`02 07`	`ADD AL, [BX]`
`011E`	`88 05`	`MOV [DI], AL`
`0120`	`46 / 43 / 47`	`INC SI / BX / DI` 1 byte each
`0123`	`E2 F5`	`LOOP 011Ah` F5h = −11
`0125`	`C3`	`RET`

emu8086 memory window (CS = DS = 0700h). Code and data share one segment; the assembler encoded LEA as the shorter `MOV reg, imm`.

Binary Coded Decimal (BCD)

- **BCD** (Binary Coded Decimal): each decimal digit is kept in its own 4-bit group (0000–1001).
- **Unpacked** BCD: one digit per byte. **Packed** BCD: two digits per byte.
- The CPU adds in binary; special instructions adjust the result to BCD.

Representation	The number 14	Byte(s)
Binary	0000 1110b	`0Eh`
Unpacked BCD	0000 0001 · 0000 0100	`01h 04h` (AH, AL)
Packed BCD	0001 0100b	`14h`
ASCII	'1' '4'	`31h 34h`

BCD Adjust Instructions

Instruction	When?	Effect
`AAA`	after unpacked addition	adjust AL to BCD, carry into AH
`AAS`	after unpacked subtraction	adjust AL, borrow from AH
`AAM`	after unpacked multiplication	AH = AL / 10, AL = AL mod 10
`AAD`	**before** unpacked division	AL = AH × 10 + AL, AH = 0
`DAA` / `DAS`	after packed addition / subtraction	adjust the two digits in AL

The adjust instructions work only on **AL** (and AH), so BCD addition/subtraction is done byte by byte. Example: 83h − 38h = 4Bh → `DAS` → **45h** (83 − 38 = 45).

Code Trace: 9 + 5 with AAA

```
 1 ; first number '9':
 2 mov     ah, 09h
 3 ; second number '5':
 4 mov     al, 05h
 5 ; al = al + ah = 09h + 05h = 0eh
 6 add     al, ah
 7 ; clear tens byte of bcd
 8 xor     ah, ah
 9 ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

AH = 09h: the first digit, 9.

Step 1 / 12

VARIABLES

AX = **0900h** BL = **00h** CF = **0**

AF = **0** ZF = **0**

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

AL = 05h: the second digit, 5. AX = 0905h.

Step 2 / 12

VARIABLES

AX = 0905h | BL = 00h | CF = 0 | AF = 0 | ZF = 0

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
▶  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

09h + 05h = **0Eh** (14): correct in binary but not BCD (E > 9). No nibble carry → AF = 0.

Step 3 / 12

VARIABLES

AX = 090Eh | BL = 00h | CF = 0 | AF = 0 | ZF = 0

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
▶ xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

AH = 0: the tens digit is cleared. AX = 000Eh.

Step 4 / 12

VARIABLES

AX = 000Eh | BL = 00h | CF = 0 | AF = 0 | ZF = 1

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
▶  aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h    ; to ASCII
13 mov     bl, al       ; save '4'
14 mov     al, ah       ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

`AAA`: the low nibble of AL is E > 9 → AL + 6 = 14h, **AH + 1**, AF = CF = 1; the high nibble of AL is cleared → **AX = 0104h** → “14”.

Step 5 / 12

VARIABLES

AX = **0104h** BL = **00h** CF = **1** AF = **1** ZF = **1**

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
▶ or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

30h is added to each digit: AX = **3134h** = '1','4'.

Step 6 / 12

VARIABLES

AX = **3134h** BL = **00h** CF = **0** AF = **1** ZF = **0**

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
▶ mov     bl, al      ; save '4'
▶ mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

BL = '4' is saved, AL = '1'.

Step 7 / 12

VARIABLES

AX = **3131h** BL = **34h** CF = **0**

AF = **1** ZF = **0**

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
▶ mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
19 ret
```

AH = 0Eh (teletype).

Step 8 / 12

VARIABLES

AX = **0E31h** | BL = **34h** | CF = **0** | AF = **1** | ZF = **0**

OUTPUT

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
▶ int     10h
17 mov     al, bl
18 int     10h
19 ret
```

'1' is printed.

Step 9 / 12

VARIABLES

AX = **0E31h** | BL = **34h** | CF = **0** | AF = **1** | ZF = **0**

OUTPUT

```
1
```

Code Trace: 9 + 5 with AAA

```
 1 ; first number '9':
 2 mov     ah, 09h
 3 ; second number '5':
 4 mov     al, 05h
 5 ; al = al + ah = 09h + 05h = 0eh
 6 add     al, ah
 7 ; clear tens byte of bcd
 8 xor     ah, ah
 9 ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
▶ mov     al, bl
18 int     10h
19 ret
```

AL = '4'.

Step 10 / 12

VARIABLES

AX = **0E34h** | BL = **34h** | CF = **0** | AF = **1** | ZF = **0**

OUTPUT

```
1
```

Code Trace: 9 + 5 with AAA

```
1  ; first number '9':
2  mov     ah, 09h
3  ; second number '5':
4  mov     al, 05h
5  ; al = al + ah = 09h + 05h = 0eh
6  add     al, ah
7  ; clear tens byte of bcd
8  xor     ah, ah
9  ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
▶  int     10h
19 ret
```

'4' is printed. The screen shows **14**.

Step 11 / 12

VARIABLES

AX = **0E34h** BL = **34h** CF = **0**

AF = **1** ZF = **0**

OUTPUT

```
14
```

Code Trace: 9 + 5 with AAA

```
 1 ; first number '9':
 2 mov     ah, 09h
 3 ; second number '5':
 4 mov     al, 05h
 5 ; al = al + ah = 09h + 05h = 0eh
 6 add     al, ah
 7 ; clear tens byte of bcd
 8 xor     ah, ah
 9 ; adjust result to bcd form, ah = 1, al = 4 -> '14'
10 aaa
11 ; --- print the result (extra) ---
12 or      ax, 3030h   ; to ASCII
13 mov     bl, al      ; save '4'
14 mov     al, ah      ; '1'
15 mov     ah, 0Eh
16 int     10h
17 mov     al, bl
18 int     10h
▶ ret
```

`RET` .

Step 12 / 12

VARIABLES

AX = **0E34h** BL = **34h** CF = **0**

AF = **1** ZF = **0**

OUTPUT

```
14
```

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

AL = 05h. emu8086 starts the program with AX = 0000h.

Step 1 / 9

VARIABLES

AX = **0005h** BL = **00h** CF = **0** AF = **0** SF = **0**

OUTPUT

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
▶  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

BL = 08h.

Step 2 / 9

VARIABLES

AX = **0005h** | BL = **08h** | CF = **0** | AF = **0** | SF = **0**

OUTPUT

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
▶  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

5 − 8 = −3 = **FDh**. Borrow: CF = 1; the low nibble borrows too: AF = 1.

Step 3 / 9

VARIABLES

AX = **00FDh** BL = **08h** CF = **1** AF = **1** SF = **1**

OUTPUT

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
▶  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

`AAS`: AF = 1 → AL − 6 = F7h, **AH − 1 = FFh**, CF = AF = 1; the high nibble is cleared → **AX = FF07h**.

Step 4 / 9

VARIABLES

AX = **FF07h** | BL = **08h** | CF = **1** | AF = **1** | SF = **1**

OUTPUT

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
▶ or      al, 30h
11 mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

AL = 37h = '7'.

Step 5 / 9

VARIABLES

AX = **FF37h** BL = **08h** CF = **0** AF = **1** SF = **0**

OUTPUT

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
▶  mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

AH = 0Eh.

Step 6 / 9

VARIABLES

AX = **0E37h** | BL = **08h** | CF = **0** | AF = **1** | SF = **0**

OUTPUT

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
▶ int     10h
13 mov     ah, 0
14 int     16h
15 ret
```

'7' is printed.

Step 7 / 9

VARIABLES

AX = **0E37h** | BL = **08h** | CF = **0** | AF = **1** | SF = **0**

OUTPUT

```
7
```

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
12 int     10h
▶  mov     ah, 0
▶  int     16h
15 ret
```

Waits for a key.

Step 8 / 9

VARIABLES

AX = **0037h** BL = **08h** CF = **0** AF = **1** SF = **0**

OUTPUT

```
7
```

Code Trace: 5 − 8 with AAS

```
1  ; make 5 - 8, al = 0fdh (not in bcd form)
2  mov     al, 05h
3  mov     bl, 08h
4  sub     al, bl
5  ; convert to bcd, al = 7
6  ; and 1 is borrowed from ah,
7  ; like calculating 15 - 8:
8  aas
9  ; --- print ---
10 or      al, 30h
11 mov     ah, 0Eh
12 int     10h
13 mov     ah, 0
14 int     16h
▶  ret
```

The result is “7, borrow 1”, as in 15 − 8 = 7. The true difference is −3; CF = 1 says the result is negative and 7 is the tens' complement of 3.

Step 9 / 9

VARIABLES

AX = **0037h** BL = **08h** CF = **0** AF = **1** SF = **0**

OUTPUT

```
7
```

AAM and AAD

AAM · after multiplication

```
mov al, 7
mov bl, 8
mul bl      ; AX = 0038h (56)
aam         ; AH = 05h, AL = 06h
```

Divides AL by 10: quotient to AH, remainder to AL. 56 → “5”, “6”.

AAD · before division

```
mov ax, 0504h  ; '54' unpacked
aad            ; AX = 0036h (54)
mov bl, 6
div bl         ; AL = 09h, AH = 00h
```

AL = AH × 10 + AL, AH = 0: the two digits become a binary number, then a normal DIV.

For packed BCD use `DAA`: `MOV AL,38h / ADD AL,45h` → AL = 7Dh, `DAA` → AL = **83h** (38 + 45 = 83).

Code Trace: AAM, AAD and DAA

```
▶ MOV  AL, 6
▶ MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h     ; packed 19
10 ADD  AL, 28h     ; + 28
11 DAA              ; adjust BCD
12 RET
```

AH	AL	BL	AF	CF
00h	06h	09h	0	0

AX = 0006h (binary 6)

AL nibbles: 0 | 6

Two single-digit numbers: AL = 6, BL = 9.

Step 1 / 9

VARIABLES

AX = **0006h** BL = **09h**

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
▶  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h     ; packed 19
10 ADD  AL, 28h     ; + 28
11 DAA              ; adjust BCD
12 RET
```

AH	AL	BL	AF	CF
00h	36h	09h	?	0

AX = 0036h (binary 54)

AL nibbles: 3 | 6

`MUL BL` : AX = 6 × 9 = 54 = 0036h. A binary result; to print it, split it into digits.

Step 2 / 9

VARIABLES

AX = 0036h BL = 09h

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
▶  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h     ; packed 19
10 ADD  AL, 28h     ; + 28
11 DAA              ; adjust BCD
12 RET
```

AH	AL	BL	AF	CF
05h	04h	09h	?	?

AX = 0504h (binary 1284)

AL nibbles: 0 | 4

`AAM` : AL / 10 → AH = 5 (quotient), AL = 4 (remainder). AX = 0504h: unpacked “54”.

Step 3 / 9

VARIABLES

AX = **0504h** BL = **09h**

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL            ; AX = 6 × 9
4  AAM                ; split digits
▶  MOV  AX, 0702h     ; '72' unpacked
6  AAD                ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h       ; packed 19
10 ADD  AL, 28h       ; + 28
11 DAA                ; adjust BCD
12 RET
```



AX = 0702h (binary 1794)

AL nibbles: 0 | 2

AX = 0702h: unpacked BCD “72” (AH = 7, AL = 2).

Step 4 / 9

VARIABLES

AX = **0702h** BL = **09h**

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
▶  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h     ; packed 19
10 ADD  AL, 28h     ; + 28
11 DAA              ; adjust BCD
12 RET
```



AX = 0048h (binary 72)

AL nibbles: 4 | 8

`AAD` : AL = 7 × 10 + 2 = 72 = 48h, AH = 0. Converted to binary before the division.

Step 5 / 9

VARIABLES

AX = 0048h BL = 09h

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
▶  MOV  BL, 8
▶  DIV  BL
9  MOV  AL, 19h     ; packed 19
10 ADD  AL, 28h     ; + 28
11 DAA              ; adjust BCD
12 RET
```

AH	AL	BL	AF	CF
00h	09h	08h	?	?

AX = 0009h (binary 9)

AL nibbles: 0 | 9

`DIV BL` : 72 / 8 → AL = 9, AH = 0 (remainder). Without AAD, 0702h = 1794 would be divided.

Step 6 / 9

VARIABLES

AX = 0009h BL = 08h

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
▶ MOV  AL, 19h      ; packed 19
10 ADD  AL, 28h     ; + 28
11 DAA              ; adjust BCD
12 RET
```

AH	AL	BL	AF	CF
00h	19h	08h	?	?

AX = 0019h (binary 25)

AL nibbles: 1 | 9

AL = 19h: packed BCD “19” (two digits in one byte).

Step 7 / 9

VARIABLES

AX = 0019h BL = 08h

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h     ; packed 19
▶ ADD  AL, 28h      ; + 28
11 DAA              ; adjust BCD
12 RET
```



AX = 0041h (binary 65)

AL nibbles: 4 | 1

19h + 28h = 41h: a binary addition. 9 + 8 = 17 overflowed the low nibble → **AF = 1**; 41h is wrong as BCD.

Step 8 / 9

VARIABLES

AX = **0041h** BL = **08h**

Code Trace: AAM, AAD and DAA

```
1  MOV  AL, 6
2  MOV  BL, 9
3  MUL  BL          ; AX = 6 × 9
4  AAM              ; split digits
5  MOV  AX, 0702h   ; '72' unpacked
6  AAD              ; to binary
7  MOV  BL, 8
8  DIV  BL
9  MOV  AL, 19h     ; packed 19
10 ADD  AL, 28h     ; + 28
▶  DAA              ; adjust BCD
12 RET
```

AH	AL	BL	AF	CF
00h	47h	08h	1	0

AX = 0047h (binary 71)

AL nibbles: 4 | 7

`DAA` : AF = 1 → AL + 06h = 47h. The result is “47” = 19 + 28 ✓

Step 9 / 9

VARIABLES

AX = **0047h** BL = **08h**

Code Trace: °C → °F

```
1  org 100h
2  start:
3  ;celsius to fahrenheit according to f = c * 9 / 5 + 32
▶  mov cl, 15
5  mov al, 9
6  imul cl
7  mov cl, 5
8  idiv cl
9  add al, 32
10 ret
```

CL = 15 (°C) = 0Fh.

Step 1 / 7

VARIABLES

AX = **0000h** CL = **0Fh**

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

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

EXAMPLE: TEMPERATURE CONVERSION · STEP THROUGH THE CODE

Code Trace: °C → °F

```
1  org 100h
2  start:
3  ;celsius to fahrenheit according to f = c * 9 / 5 + 32
4  mov cl, 15
▶  mov al, 9
6  imul cl
7  mov cl, 5
8  idiv cl
9  add al, 32
10 ret
```

AL = 9.

Step 2 / 7

VARIABLES

AX = **0009h** CL = **0Fh**

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

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

EXAMPLE: TEMPERATURE CONVERSION · STEP THROUGH THE CODE

Code Trace: °C → °F

```
1  org 100h
2  start:
3  ;celsius to fahrenheit according to f = c * 9 / 5 + 32
4  mov cl, 15
5  mov al, 9
▶  imul cl
7  mov cl, 5
8  idiv cl
9  add al, 32
10 ret
```

`IMUL CL` : AX = 9 × 15 = 135 = **0087h**. 135 does not fit in a signed byte (>127) → **CF = OF = 1**; but the full result is in AX.

Step 3 / 7

VARIABLES

AX = **0087h** CL = **0Fh**

CF = **1** OF = **1**

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

Code Trace: °C → °F

```
1  org 100h
2  start:
3  ;celsius to fahrenheit according to f = c * 9 / 5 + 32
4  mov cl, 15
5  mov al, 9
6  imul cl
▶  mov cl, 5
8  idiv cl
9  add al, 32
10 ret
```

CL = 5 (the divisor).

Step 4 / 7

VARIABLES

AX = **0087h** CL = **05h**

CF = **1** OF = **1**

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

Code Trace: °C → °F

```
1  org 100h
2  start:
3  ;celsius to fahrenheit according to f = c * 9 / 5 + 32
4  mov cl, 15
5  mov al, 9
6  imul cl
7  mov cl, 5
▶  idiv cl
9  add al, 32
10 ret
```

`IDIV CL` : AX / 5 = 135 / 5 → **AL = 1Bh (27)**, AH = 0 (remainder). The flags are undefined.

Step 5 / 7

VARIABLES

AX = **001Bh** CL = **05h**

CF = **1** OF = **1**

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

Code Trace: °C → °F

```
1  org 100h
2  start:
3  ;celsius to fahrenheit according to f = c * 9 / 5 + 32
4  mov cl, 15
5  mov al, 9
6  imul cl
7  mov cl, 5
8  idiv cl
▶  add al, 32
10 ret
```

27 + 32 = **59 = 3Bh** → AX = 003Bh. 15 °C = 59 °F ✓

Step 6 / 7

VARIABLES

AX = **003Bh** CL = **05h**

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

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

Code Trace: °C → °F

```
1 org 100h
2 start:
3 ;celsius to fahrenheit according to f = c * 9 / 5 + 32
4 mov cl, 15
5 mov al, 9
6 imul cl
7 mov cl, 5
8 idiv cl
9 add al, 32
▶ ret
```

`RET`. Final state in emu8086: AX = 003Bh, CX = 0005h.

Step 7 / 7

VARIABLES

AX = **003Bh** CL = **05h**

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

f = c × 9 / 5 + 32. Multiply first, then divide: in integer division 9/5 would be 1 and the result would be wrong.

EXAMPLE: TEMPERATURE CONVERSION · STEP THROUGH THE CODE

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
▶  mov cl, 59
5  sub cl, 32
6  mov al, 5
7  imul cl
8  mov cl, 9
9  idiv cl
10 ret
```

CL = 59 (°F) = 3Bh.

Step 1 / 7

VARIABLES

AX = **0000h** CL = **3Bh**

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

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
4  mov cl, 59
▶  sub cl, 32
6  mov al, 5
7  imul cl
8  mov cl, 9
9  idiv cl
10 ret
```

59 − 32 = **27 = 1Bh**.

Step 2 / 7

VARIABLES

AX = **0000h** CL = **1Bh**

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

EXAMPLE: TEMPERATURE CONVERSION · STEP THROUGH THE CODE

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
4  mov cl, 59
5  sub cl, 32
▶  mov al, 5
7  imul cl
8  mov cl, 9
9  idiv cl
10 ret
```

AL = 5.

Step 3 / 7

VARIABLES

AX = **0005h** CL = **1Bh**

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

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
4  mov cl, 59
5  sub cl, 32
6  mov al, 5
▶  imul cl
8  mov cl, 9
9  idiv cl
10 ret
```

AX = 5 × 27 = 135 = **0087h**; CF = OF = 1 (it did not fit in AL).

Step 4 / 7

VARIABLES

AX = **0087h** CL = **1Bh**

CF = **1** OF = **1**

EXAMPLE: TEMPERATURE CONVERSION · STEP THROUGH THE CODE

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
4  mov cl, 59
5  sub cl, 32
6  mov al, 5
7  imul cl
▶  mov cl, 9
9  idiv cl
10 ret
```

CL = 9.

Step 5 / 7

VARIABLES

AX = **0087h** CL = **09h**

CF = **1** OF = **1**

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
4  mov cl, 59
5  sub cl, 32
6  mov al, 5
7  imul cl
8  mov cl, 9
▶  idiv cl
10 ret
```

135 / 9 = **15 = 0Fh** → AL, remainder AH = 0. 59 °F = 15 °C ✓

Step 6 / 7

VARIABLES

AX = **000Fh** CL = **09h**

CF = **1** OF = **1**

EXAMPLE: TEMPERATURE CONVERSION · STEP THROUGH THE CODE

Code Trace: °F → °C

```
1  org 100h
2  start:
3  ;fahrenheit to celsius according to: c = (f - 32) * 5 / 9
4  mov cl, 59
5  sub cl, 32
6  mov al, 5
7  imul cl
8  mov cl, 9
9  idiv cl
▶  ret
```

`RET` .

Step 7 / 7

VARIABLES

`AX = 000Fh` `CL = 09h`

`CF = 1` `OF = 1`

Why IMUL / IDIV? Limits

Negative temperatures

c = −10: `IMUL` → AX = FFA6h (−90), `IDIV` → AL = EEh (−18), +32 → **0Eh = 14** °F ✓

With MUL/DIV, F6h would be taken as 246 and the result would be meaningless.

Integer division

IDIV drops the fraction (toward zero): c = 1 → 9/5 = 1 → 33 °F (really 33.8). The remainder (AH) can be checked for rounding.

Valid range for °C → °F

The result is in the signed byte AL: **c** ∈ **[−71, 53]**. At c = 54, AL = 81h (OF = 1, wrong); at c = 72 the quotient does not fit in AL → **divide error**.

Valid range for °F → °C

f − 32 must fit in a byte: **f** ∈ **[−96, 127]**. For a wider range use word operations (AX, CWD, IDIV BX).

Lab: Arithmetic and Logic

CF and OF

CODE · 16 bytes

```
 1 ORG 100h
 2 MOV  AL, 9Ch      ; 156 or …
 3 ADD  AL, 70h      ; + 112
 4 MOV  AL, 50h      ; 80
 5 ADD  AL, 50h      ; + 80
 6 ADD  AL, AL       ; A0h + A…
 7 MOV  AL, 0FFh
 8 INC  AL           ; CF unto…
 9 CMC               ; flip CF
10 RET
```

REGISTERS

AX	0000	AH	00	AL	00
BX	0000	BH	00	BL	00
CX	0010	CH	00	CL	10
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 B0 9C 04 70 B0 50 04 50
0108 02 C0 B0 FF FE C0 F5 C3
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: 16 bytes, CS:IP = 0700:0100h. As in emu8086, CX initially holds the program size (0010h). The Step button executes instructions one at a time; the highlighted line is the next instruction.

Summary: Instructions and Their Flags

Instruction	CF	ZF	SF	OF	PF	AF
`ADD` `SUB` `CMP` `ADC` `SBB`	✓	✓	✓	✓	✓	✓
`NEG`	✓	✓	✓	✓	✓	✓
`INC` `DEC`	—	✓	✓	✓	✓	✓
`AND` `OR` `XOR` `TEST`	0	✓	✓	0	✓	?
`NOT`	—	—	—	—	—	—
`MUL` `IMUL`	✓	?	?	✓	?	?
`DIV` `IDIV`	?	?	?	?	?	?
`SHL` `SHR` `SAR`	✓	✓	✓	✓[1]	✓	?
`ROL` `ROR` `RCL` `RCR`	✓	—	—	✓[1]	—	—
`AAA` `AAS`	✓	?	?	?	?	✓
`MOV` `LEA` `LOOP`	—	—	—	—	—	—

✓: set by result · 0: cleared · —: unchanged · ?: undefined · [1]: defined only for a count of 1.

Summary

Flags

CF unsigned, OF signed overflow; ZF zero; SF the MSB; PF an even number of 1s in the low byte; AF the nibble carry.

Group 1

Result in the first operand; CMP/TEST flags only. Logic instructions set CF = OF = 0.

Group 2

Implicit AL/AX/DX. Byte: AX = AL × op, AL = AX / op (AH remainder). Word: DX:AX.

Group 3

INC/DEC preserve CF; NOT changes no flags; NEG is two's complement.

Shifts and rotates

Count 1 or CL. SHL ×2, SHR unsigned ÷2, SAR signed ÷2 (rounds toward −∞); rotates lose no bits, RCL/RCR form a 9 (17)-bit ring with CF.

BCD

AAA/AAS after addition/subtraction, AAM after multiplication, AAD before division.

Pitfalls

Prepare AH/DX before dividing; use IMUL/IDIV for signed data; detect overflow with the flags.

SUMMARY

Exercises

1. What are AL, CF, OF, SF, ZF, PF after `MOV AL, 0C0h` / `ADD AL, 0C0h` ?
2. After `CMP AL, BL` , which flag tells unsigned “AL < BL”, and which flags tell signed “AL < BL”?
3. With AX = 0FFFFh and BX = 0FFFFh, what are DX, AX, CF after `MUL BX` ?
4. What happens on `DIV BL` with AX = 0100h and BL = 0?
5. Change the array addition program for word (DW) arrays.
6. Write a program that counts the 1 bits in a byte using SHL/TEST and a loop. Compare with PF.

▶ **Hints / answers**

SUMMARY

Exercises (2)

7. What happens on `DIV BX` with DX = 0005h, AX = 0000h, BX = 0005h?
8. What are AL and OF after `MOV AL, 70h` / `ADD AL, 30h`? How do you get the correct sum in 16 bits?
9. Compute 00012345h − 0000ABCDh with SUB/SBB. What is CF after SUB?
10. In a 32 × 32-bit multiplication, which words of the result do the four partial products add into?
11. With DX = 0000h, AX = 0FFFh, CF = 0, what is DX:AX after `SHL AX, 1` and `RCL DX, 1`?

▶ **Hints / answers**

References

- Intel, *The 8086 Family User's Manual*, 1979 — Instruction Set chapter.
- B. B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — Chapter 5: arithmetic, logic, shift and rotate instructions.
- emu8086 help: the “8086 instructions” and “Arithmetic and logic instructions” pages.
- K. R. Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson, 2014 — Chapter 7: shift and rotate instructions.
- A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Pearson, 2008.
- M. Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014 (in Turkish).
- Ş. Ç. Öztürk, O. Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009 (in Turkish).

The End

Chapter 6: Arithmetic, Logic and Shift Instructions

Previous: Chapter 5.2: Interrupt Functions · Next: Chapter 7: Flow Control