Chapter 1.2: Number Systems

Microprocessors

Roman numerals | Decimal, binary, hexadecimal | Base conversion | Signed numbers | Ones' and two's complement | Binary addition/subtraction | IEEE 754 floating point | Flip-flops | BCD | Little-endian / big-endian | Gray code

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_01_02_Number_Systems.html

Learning Outcomes

Positional systems

Compute the value of decimal, binary and hexadecimal numbers from their digit weights.

Conversion

Convert numbers between bases by division and by grouping.

Signed numbers

Write, add and subtract negative numbers in two's complement.

Encodings

Explain how BCD, IEEE 754, Gray code and byte order (endianness) look in memory.

Ways to Represent Numbers

- There are many ways to represent a numeric value.
- Long ago people counted with **sticks**; later they learned to draw the sticks on paper.
- So the number 5 was first written as `| | | | |`.
- In the Roman Empire `| | |` still meant three sticks, while **V** came to stand for five and **X** for ten.
- In the **positional** system we use today, a digit's value also depends on its position.



5 sticks



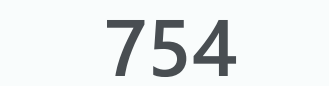
III = 3



V = 5



X = 10



positional: 7·100 + 5·10 + 4

Roman Numerals

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

- A number system used in ancient Rome.
- Built on **addition** and **subtraction**: a smaller symbol after a larger one is added (VI = 6), one before it is subtracted (IV = 4).
- Subtraction is used only in these pairs: IV, IX, XL, XC, CD, CM.
- There is no **zero** and no **place value**, so addition and multiplication are very tedious.

Reading Roman Numerals

MCMLXXXVI

M CM LXXX VI

Split the number into value groups from left to right.

Step 1 / 6

MMVII

M M VII

1000 + 1000 + 7 = 2007

`MMVI` = 2006, **MMVII** = 1000 + 1000 + 7 = 2007.

Reading Roman Numerals

MCMLXXXVI



M = 1000.

Step 2 / 6

MMVII



`MMVI` = 2006, **MMVII** = 1000 + 1000 + 7 = 2007.

Reading Roman Numerals

MCMLXXXVI



CM: C comes before M, so it is subtracted → 1000 − 100 = 900.

MMVII



`MMVI` = 2006, **MMVII** = 1000 + 1000 + 7 = 2007.

Reading Roman Numerals

MCMLXXXVI



LXXX = 50 + 10 + 10 + 10 = 80.

Step 4 / 6

MMVII



`MMVI` = 2006, **MMVII** = 1000 + 1000 + 7 = 2007.

Reading Roman Numerals

MCMLXXXVI



VI = 5 + 1 = 6.

Step 5 / 6

MMVII



`MMVI` = 2006, **MMVII** = 1000 + 1000 + 7 = 2007.

Reading Roman Numerals

MCMLXXXVI



Total 1000 + 900 + 80 + 6 = **1986**.

Step 6 / 6

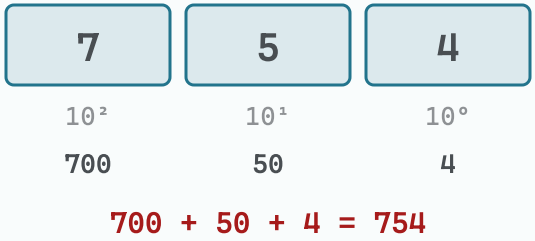
MMVII



`MMVI` = 2006, **MMVII** = 1000 + 1000 + 7 = 2007.

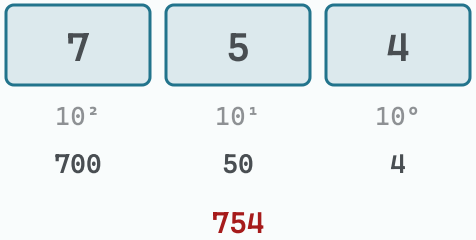
The Decimal System

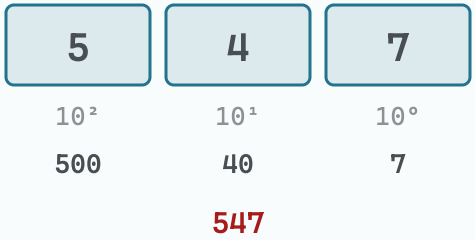
- Today most people count in **decimal** (base 10).
- The decimal system has 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- These digits can represent any value, for example **754**.
- A number's value is the sum of each digit times **the base raised to the digit's position**.



General formula: $d_{n-1}...d_1d_0 = \Sigma\, d_i \times b^i$, where b is the base and i the digit's position (from the right, starting at 0).

Position Matters

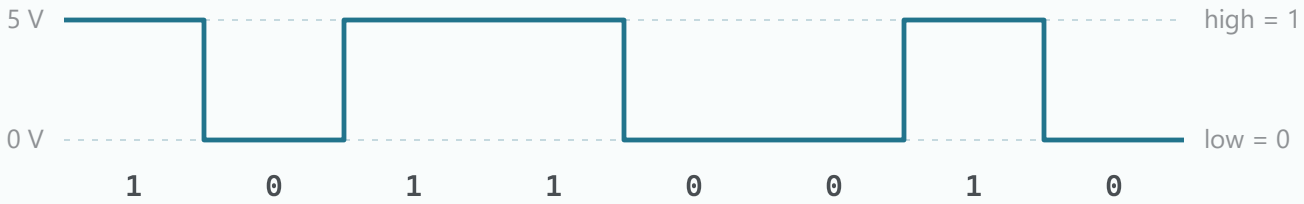




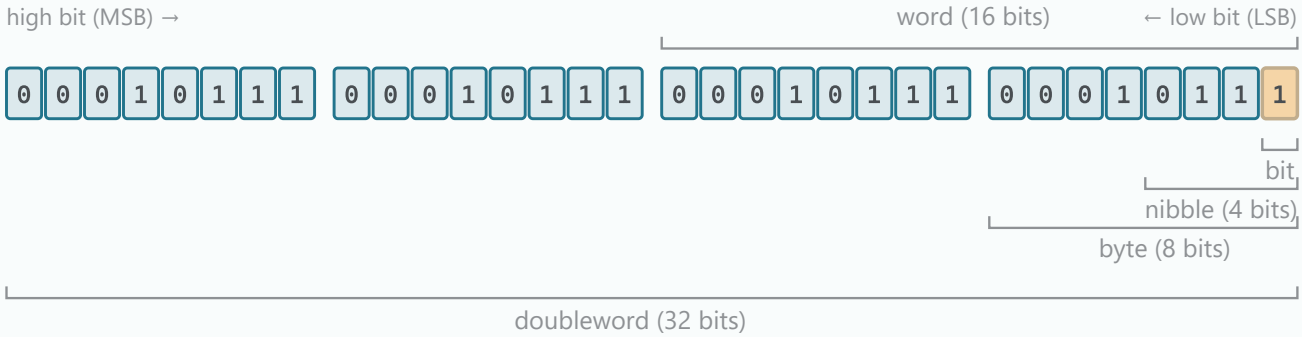
- Each digit's **position matters**: move the 7 to the end and you get 547, a different value.
- Any number to the power zero is 1: $x^0 = 1$. That is why the rightmost digit is the “ones” place in every base.
- 0^0 is undefined in mathematics; in number systems and polynomials it is **taken as 1 by convention**.
- Unlike Roman numerals, the same digit (7) is worth 700, 70 or 7 depending on its place, and an empty place needs a **0**.

The Binary System

- Electronic machines are easiest to build with **two states**: on/off, high/low voltage, 1/0.
- That is why computers use the **binary** system.
- Binary has 2 digits: **0** and **1**; the base is 2.
- The digit weights are powers of 2: 1, 2, 4, 8, 16, 32, 64, 128 …



Bit, Nibble, Byte, Word



Unit	Bit	Unsigned range	On the 8086
bit	1	0 … 1	flag
nibble	4	0 … 15 (0h … Fh)	one hex digit, one BCD digit
byte	8	0 … 255	AL, BL …, `DB`
word	16	0 … 65535	AX, BX …, `DW`
doubleword	32	0 … 4,294,967,295	DX:AX, `DD`

Each digit of a binary number is called a bit (binary digit). n bits can show 2^n different values.

Reading Binary Numbers

- In assembly, binary numbers usually end with the letter **b**: `101b` = 5.
- Example: `10100101b` equals decimal **165**.

1	0	1	0	0	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	0	32	0	0	4	0	1

128 + 32 + 4 + 1 = 165

Shortcut: add up the weights of the digits that are **1**. The 8-bit weights: 128 · 64 · 32 · 16 · 8 · 4 · 2 · 1.

Binary to Decimal: 1111011b



$(1111011)_2$: 7 digits, weights 2^0 … 2^6.

Step 1 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



sum = 1

Digit 0: 1 × 2^0 = 1 → running sum **1**.

Step 2 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



sum = 3

Digit 1: $1 \times 2^1 = 2$ → running sum **3**.

Step 3 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



sum = 3

Digit 2: 0 × 2^2 = 0 → running sum **3**.

Step 4 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



sum = 11

Digit 3: 1 × 2^3 = 8 → running sum **11**.

Step 5 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



sum = 27

Digit 4: 1 × 2^4 = 16 → running sum **27**.

Step 6 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



sum = 59

Digit 5: 1 × 2^5 = 32 → running sum **59**.

Step 7 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

Binary to Decimal: 1111011b



Digit 6: $1 \times 2^6 = 64$ → running sum **123**.

Step 8 / 8

$(1111011)_2 = 1\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6 = 1 + 2 + 0 + 8 + 16 + 32 + 64 =$ **123**

The Hexadecimal System

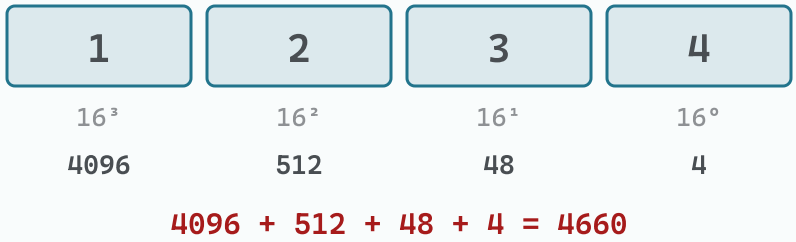
base 10	base 2	base 16
0	`0000`	**0**
1	`0001`	**1**
2	`0010`	**2**
3	`0011`	**3**
4	`0100`	**4**
5	`0101`	**5**
6	`0110`	**6**
7	`0111`	**7**

base 10	base 2	base 16
8	`1000`	**8**
9	`1001`	**9**
10	`1010`	**A**
11	`1011`	**B**
12	`1100`	**C**
13	`1101`	**D**
14	`1110`	**E**
15	`1111`	**F**

- The hexadecimal system uses 16 digits: 0–9 and **A, B, C, D, E, F** (10–15).
- One hex digit corresponds to exactly **4 bits** (a nibble).
- So hex numbers are **shorter** than binary and easier to read: 1010 0101b = A5h.

Writing Hex Numbers

- Hex numbers usually end with **h**: `5Fh` = 5×16 + 15 = **95**.
- Numbers that start with a letter (A…F) get a leading **0**: `0E120h`. Otherwise the assembler takes `E120h` for a variable or label name.
- Example: `1234h` equals decimal **4660**.



Notation	Meaning
`MOV AL, 0FFh`	AL = 255 ✓
`MOV AL, FFh`	error: FFh is an undefined name
`MOV AX, 10`	decimal 10 (= 0Ah) — no suffix means decimal

Hex to Decimal: A12Fh

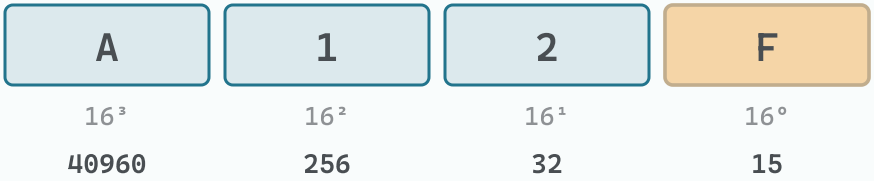


$(A12F)_{16}$: A = 10, F = 15. Weights $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$.

Step 1 / 5

$(A12F)_{16} = 15 + 32 + 256 + 40960 =$ **41263**

Hex to Decimal: A12Fh



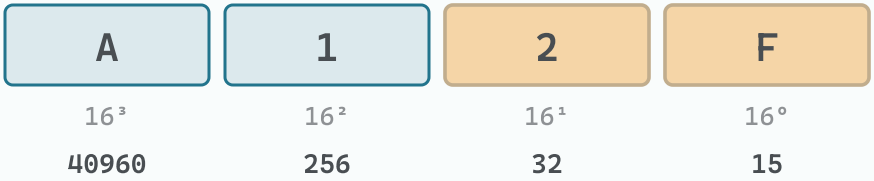
`sum = 15`

F × 16^0 = 15 × 1 = **15**.

Step 2 / 5

$(A12F)_{16}$ = 15 + 32 + 256 + 40960 = **41263**

Hex to Decimal: A12Fh



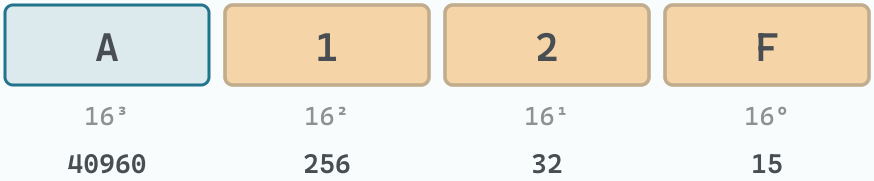
sum = 47

$2 \times 16^1 = 2 \times 16 = \mathbf{32} \rightarrow 47.$

Step 3 / 5

$(A12F)_{16} = 15 + 32 + 256 + 40960 = \mathbf{41263}$

Hex to Decimal: A12Fh



sum = 303

$1 \times 16^2 = 1 \times 256 =$ **256** → 303.

Step 4 / 5

$(A12F)_{16} = 15 + 32 + 256 + 40960 =$ **41263**

Hex to Decimal: A12Fh



sum = 41263

A × 16^3 = 10 × 4096 = **40960** → **41263**.

Step 5 / 5

$(A12F)_{16}$ = 15 + 32 + 256 + 40960 = **41263**

Binary ↔ Hex: Grouping by Four

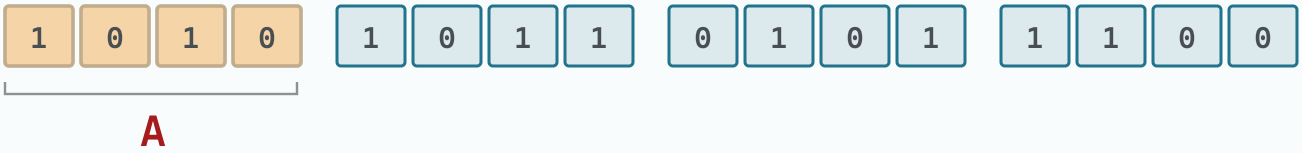


Split the binary number into **groups of 4** starting from the right (pad with 0s on the left if needed).

Step 1 / 5

- The reverse works the same way: each hex digit expands to 4 bits: `3Eh` → 0011 1110b.
- Example: `1101011b` → 0110 1011 → **6Bh** (a 0 was added to the left group).
- That is why programmers always write binary data in hex: one byte = exactly 2 hex digits.

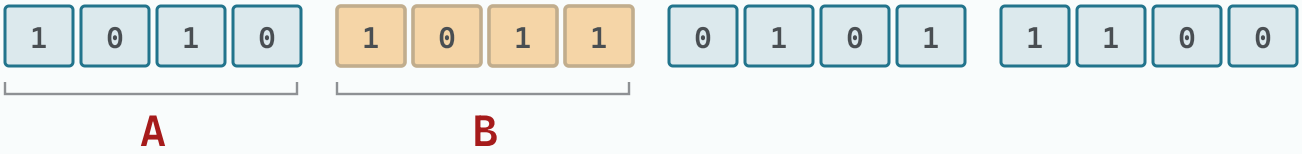
Binary ↔ Hex: Grouping by Four



1010 = 8 + 2 = 10 → **A**.

- The reverse works the same way: each hex digit expands to 4 bits: `3Eh` → 0011 1110b.
- Example: `1101011b` → 0110 1011 → **6Bh** (a 0 was added to the left group).
- That is why programmers always write binary data in hex: one byte = exactly 2 hex digits.

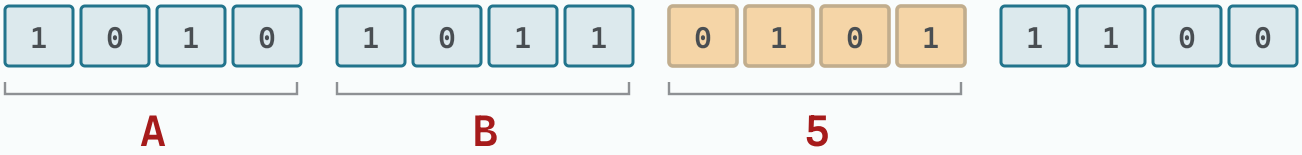
Binary ↔ Hex: Grouping by Four



1011 = 8 + 2 + 1 = 11 → **B**.

- The reverse works the same way: each hex digit expands to 4 bits: `3Eh` → 0011 1110b.
- Example: `1101011b` → 0110 1011 → **6Bh** (a 0 was added to the left group).
- That is why programmers always write binary data in hex: one byte = exactly 2 hex digits.

Binary ↔ Hex: Grouping by Four

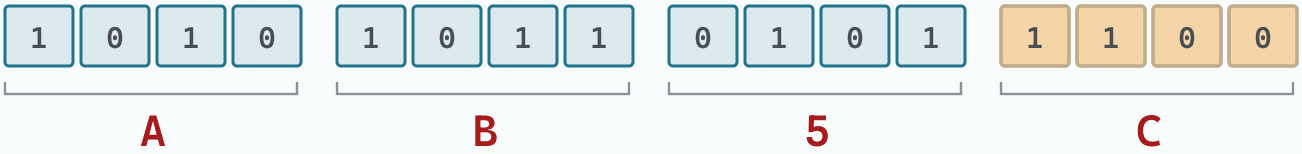


0101 = 4 + 1 = 5 → **5**.

Step 4 / 5

- The reverse works the same way: each hex digit expands to 4 bits: `3Eh` → 0011 1110b.
- Example: `1101011b` → 0110 1011 → **6Bh** (a 0 was added to the left group).
- That is why programmers always write binary data in hex: one byte = exactly 2 hex digits.

Binary ↔ Hex: Grouping by Four



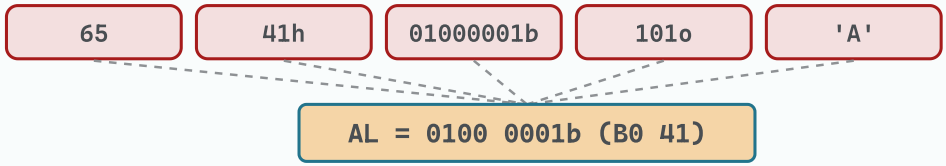
1100 = 8 + 4 = 12 → **C**. Result: **0AB5Ch**.

Step 5 / 5

- The reverse works the same way: each hex digit expands to 4 bits: `3Eh` → 0011 1110b.
- Example: `1101011b` → 0110 1011 → **6Bh** (a 0 was added to the left group).
- That is why programmers always write binary data in hex: one byte = exactly 2 hex digits.

Number Notation in Assembly

Notation	Base	Value
`MOV AL, 65`	decimal (no suffix or `d`)	65
`MOV AL, 41h`	hexadecimal	65
`MOV AL, 01000001b`	binary	65
`MOV AL, 101o`	octal (`o` or `q`)	65
`MOV AL, 'A'`	ASCII character	65
`MOV AL, -1`	decimal, signed	FFh



The first five notations produce the same machine code: `B0 41`. The base is just a notation for **people**; in memory everything is binary.

Decimal to Other Bases

- **Divide** the decimal value by the target base.
- Each time, **keep the quotient** and **take the remainder** (the remainder is a digit, 0 … base−1).
- Keep dividing until the quotient is zero.
- Write the remainders **last to first** (bottom up) to get the result: the first remainder is the lowest digit.

Example: convert 39 to hex.

Decimal to Other Bases

- **Divide** the decimal value by the target base.
- Each time, **keep the quotient** and **take the remainder** (the remainder is a digit, 0 … base−1).
- Keep dividing until the quotient is zero.
- Write the remainders **last to first** (bottom up) to get the result: the first remainder is the lowest digit.



39 ÷ 16 = 2, remainder **7** (the ones digit).

Decimal to Other Bases

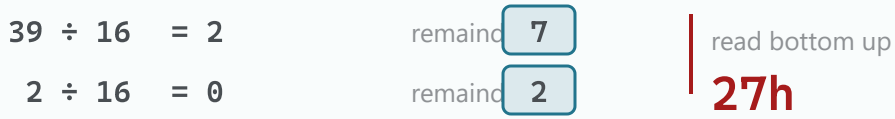
- **Divide** the decimal value by the target base.
- Each time, **keep the quotient** and **take the remainder** (the remainder is a digit, 0 … base−1).
- Keep dividing until the quotient is zero.
- Write the remainders **last to first** (bottom up) to get the result: the first remainder is the lowest digit.



2 ÷ 16 = 0, remainder **2**. Quotient 0 → stop.

Decimal to Other Bases

- **Divide** the decimal value by the target base.
- Each time, **keep the quotient** and **take the remainder** (the remainder is a digit, 0 … base−1).
- Keep dividing until the quotient is zero.
- Write the remainders **last to first** (bottom up) to get the result: the first remainder is the lowest digit.



Remainders bottom up: **27h**. Check: 2 × 16 + 7 = 39 ✓

BASE CONVERSION · STEP BY STEP

43868 → Hex

How is 43868 written in hex?

Step 1 / 6

BASE CONVERSION · STEP BY STEP

43868 → Hex

43868 ÷ 16 = 2741 remainder 12=C

43868 ÷ 16 = 2741, remainder 12 → **C**.

Step 2 / 6

BASE CONVERSION · STEP BY STEP

43868 → Hex

43868 ÷ 16 = 2741	remainder	12=C
2741 ÷ 16 = 171	remainder	5

2741 ÷ 16 = 171, remainder **5**.

Step 3 / 6

43868 → Hex

43868 ÷ 16	= 2741	remainder 12=C
2741 ÷ 16	= 171	remainder 5
171 ÷ 16	= 10	remainder 11=B

171 ÷ 16 = 10, remainder 11 → **B**.

43868 → Hex

43868 ÷ 16	= 2741	remainder 12=C
2741 ÷ 16	= 171	remainder 5
171 ÷ 16	= 10	remainder 11=B
10 ÷ 16	= 0	remainder 10=A

10 ÷ 16 = 0, remainder 10 → **A**.

43868 → Hex

43868 ÷ 16 = 2741	remainder	12=C
2741 ÷ 16 = 171	remainder	5
171 ÷ 16 = 10	remainder	11=B
10 ÷ 16 = 0	remainder	10=A

read bottom up

0AB5Ch

Result **0AB5Ch** (leading 0 because it starts with A). Check: 10·4096 + 11·256 + 5·16 + 12 = 43868 ✓

Code Trace: Hex Digits with DIV

```
1  ORG 100h
▶ MOV AX, 2989    ; number
3  MOV BX, 16     ; base
4  MOV SI, 0      ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX         ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0      ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

AX	0BADh
DX	0000h
SI	0000h

BX = 0000h

rem array

00	00	00	00
+0	+1	+2	+3

AX = 2989 = **0BADh**. The hex digits are found by repeatedly dividing by 16.

Step 1 / 13

VARIABLES

AX = 0BADh DX = 0000h BX = 0000h SI = 0

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
▶  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

AX	0BADh
DX	0000h
SI	0000h

BX = 0010h (16)

rem array

00	00	00	00
+0	+1	+2	+3

BX = 16: the target base. With BX = 2 or 8 the same code would produce that base's digits.

Step 2 / 13

VARIABLES

AX = 0BADh · DX = 0000h · BX = 0010h · SI = 0

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

AX	0BADh
DX	0000h
SI	0000h

BX = 0010h (16)

rem array

00	00	00	00
+0	+1	+2	+3

SI = 0: where the next remainder goes in the `rem` array.

Step 3 / 13

VARIABLES

AX = 0BADh DX = 0000h BX = 0010h SI = 0

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
▶  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

AX	0BADh
DX	0000h
SI	0000h

BX = 0010h (16)

rem array

00	00	00	00
+0	+1	+2	+3

DX = 0. A 16-bit `DIV BX` divides the 32-bit **DX:AX**; without clearing DX the dividend would be wrong.

Step 4 / 13

VARIABLES

`AX = 0BADh` `DX = 0000h` `BX = 0010h` `SI = 0`

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
▶  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

`2989 ÷ 16 = 186  rem 13  D`

AX	00BAh
DX	000Dh
SI	0000h

`BX = 0010h (16)`

rem array

00	00	00	00
+0	+1	+2	+3

2989 ÷ 16 = **186**, remainder **13**: AX = 00BAh (quotient), DX = 000Dh (remainder). The first remainder is the lowest digit: **D**.

Step 5 / 13

VARIABLES

`AX = 00BAh` `DX = 000Dh` `BX = 0010h` `SI = 0`

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
▶  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186 rem 13 **D**

AX	00BAh
DX	000Dh
SI	0000h

BX = 0010h (16)

rem array

0D	00	00	00
+0	+1	+2	+3

`rem[0]` ← DL = **0Dh**.

Step 6 / 13

VARIABLES

`AX = 00BAh` `DX = 000Dh` `BX = 0010h` `SI = 0`

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
▶ JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186 rem 13 **D**

AX	00BAh
DX	000Dh
SI	0001h

BX = 0010h (16)

rem array

+0	+1	+2	+3
0D	00	00	00

SI = 1; quotient 186 ≠ 0 → ZF = 0, `JNE` goes back to again.

Step 7 / 13

VARIABLES

`AX = 00BAh` `DX = 000Dh` `BX = 0010h` `SI = 1`

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
▶  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186	rem 13	D
186 ÷ 16 = 11	rem 10	A

AX	000Bh
DX	000Ah
SI	0001h

BX = 0010h (16)

rem array

0D	00	00	00
+0	+1	+2	+3

Pass 2: DX cleared, 186 ÷ 16 = **11**, remainder **10** → **A**.

Step 8 / 13

VARIABLES

AX = 000Bh · DX = 000Ah · BX = 0010h · SI = 1

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
▶  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186	rem 13	D
186 ÷ 16 = 11	rem 10	A

AX	000Bh
DX	000Ah
SI	0001h

BX = 0010h (16)

rem array

0D	0A	00	00
+0	+1	+2	+3

`rem[1]` ← **0Ah**. Quotient 11 ≠ 0: the loop continues.

Step 9 / 13

VARIABLES

AX = 000Bh · DX = 000Ah · BX = 0010h · SI = 1

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
▶  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186	rem 13	D
186 ÷ 16 = 11	rem 10	A
11 ÷ 16 = 0	rem 11	B

rem array

0D	0A	00	00
+0	+1	+2	+3

AX	0000h
DX	000Bh
SI	0002h

BX = 0010h (16)

Pass 3: 11 ÷ 16 = **0**, remainder **11** → **B**.

Step 10 / 13

VARIABLES

AX = 0000h DX = 000Bh BX = 0010h SI = 2

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
▶  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186	rem 13	D
186 ÷ 16 = 11	rem 10	A
11 ÷ 16 = 0	rem 11	B

AX	0000h
DX	000Bh
SI	0002h

BX = 0010h (16)

rem array

0D	0A	0B	00
+0	+1	+2	+3

`rem[2]` ← **0Bh**.

Step 11 / 13

VARIABLES

AX = 0000h · DX = 000Bh · BX = 0010h · SI = 2

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
▶  JNE again
11 RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186	rem 13	D
186 ÷ 16 = 11	rem 10	A
11 ÷ 16 = 0	rem 11	B

AX	0000h
DX	000Bh
SI	0003h

BX = 0010h (16)

rem array

0D	0A	0B	00
+0	+1	+2	+3

SI = 3. Quotient 0 → ZF = 1, `JNE` does not jump: the loop ends with 3 digits.

Step 12 / 13

VARIABLES

AX = 0000h · DX = 000Bh · BX = 0010h · SI = 3

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

Code Trace: Hex Digits with DIV

```
1  ORG 100h
2  MOV AX, 2989    ; number
3  MOV BX, 16      ; base
4  MOV SI, 0       ; counter
5  again: MOV DX, 0 ; DX:AX
6  DIV BX          ; quot, rem
7  MOV rem[SI], DL
8  INC SI
9  CMP AX, 0       ; quotient 0?
10 JNE again
▶  RET
12 rem DB 4 DUP(0)
```

dividing with DIV BX (base 16)

2989 ÷ 16 = 186	rem 13	D
186 ÷ 16 = 11	rem 10	A
11 ÷ 16 = 0	rem 11	B

read last to first

AX	0000h
DX	000Bh
SI	0003h

BX = 0010h (16)

rem array

0D	0A	0B	00
+0	+1	+2	+3

Memory holds the remainders **0D 0A 0B**; read last to first they give **0BADh**. Check: 11·256 + 10·16 + 13 = 2989 ✓

Step 13 / 13

VARIABLES

AX = 0000h DX = 000Bh BX = 0010h SI = 3

The same loop works in any base: the remainder is always between 0 and base−1, and it is one digit.

45 → Binary (Dividing by 2)

The same method works for binary: divide 45 by 2.

45 → Binary (Dividing by 2)

45 ÷ 2 = 22 remainder 1

45 ÷ 2 = 22, remainder 1.

45 → Binary (Dividing by 2)

45 ÷ 2 = 22	remainder 1
22 ÷ 2 = 11	remainder 0

22 ÷ 2 = 11, remainder 0.

45 → Binary (Dividing by 2)

45 ÷ 2	= 22	remainder 1
22 ÷ 2	= 11	remainder 0
11 ÷ 2	= 5	remainder 1

11 ÷ 2 = 5, remainder 1.

45 → Binary (Dividing by 2)

45 ÷ 2 = 22	remainder	1
22 ÷ 2 = 11	remainder	0
11 ÷ 2 = 5	remainder	1
5 ÷ 2 = 2	remainder	1

5 ÷ 2 = 2, remainder 1.

45 → Binary (Dividing by 2)

45 ÷ 2 = 22	remainder	1
22 ÷ 2 = 11	remainder	0
11 ÷ 2 = 5	remainder	1
5 ÷ 2 = 2	remainder	1
2 ÷ 2 = 1	remainder	0

2 ÷ 2 = 1, remainder 0.

45 → Binary (Dividing by 2)

45 ÷ 2	= 22	remainder 1
22 ÷ 2	= 11	remainder 0
11 ÷ 2	= 5	remainder 1
5 ÷ 2	= 2	remainder 1
2 ÷ 2	= 1	remainder 0
1 ÷ 2	= 0	remainder 1

1 ÷ 2 = 0, remainder 1.

45 → Binary (Dividing by 2)

45 ÷ 2 = 22	remainder	1
22 ÷ 2 = 11	remainder	0
11 ÷ 2 = 5	remainder	1
5 ÷ 2 = 2	remainder	1
2 ÷ 2 = 1	remainder	0
1 ÷ 2 = 0	remainder	1

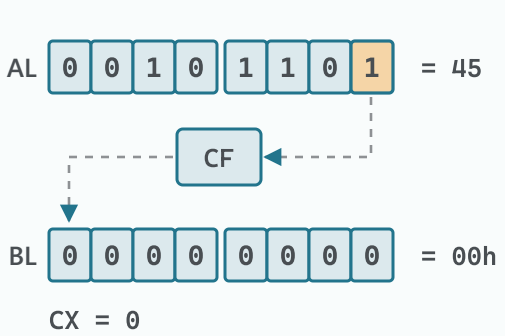
read bottom up

101101b

Result **101101b**. Check: 32 + 8 + 4 + 1 = 45 ✓ Alternative: for a large number, convert to hex first (2Dh) and expand each digit to 4 bits; it is faster.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
▶ MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1  ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



AL = 45 = **0010 1101b**. Shifting right one bit divides by 2; the bit that falls out is the **remainder**.

Step 1 / 13

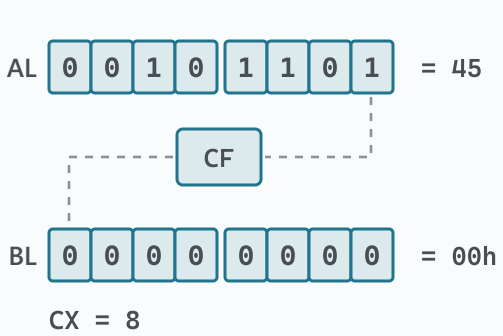
VARIABLES

`AL = 2Dh` `BL = 00h` `CF = 0` `CX = 0`

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1  ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



CX = 8: one pass for each of the 8 bits.

Step 2 / 13

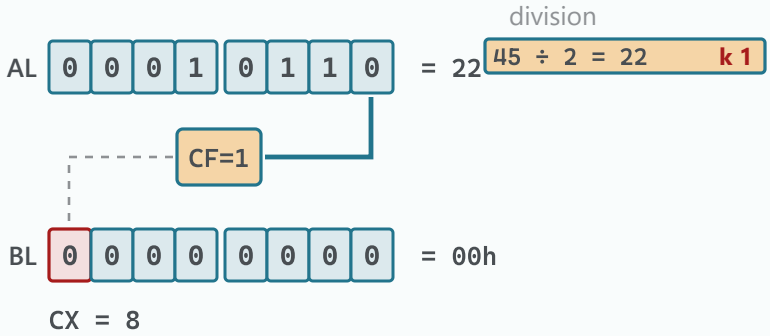
VARIABLES

AL = 2Dh | BL = 00h | CF = 0 | CX = 8

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
▶ bit: SHR AL, 1  ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



`SHR AL, 1` : 45 ÷ 2 = **22** (AL = 16h); the bit that drops out, the remainder **1** → CF = 1.

Step 3 / 13

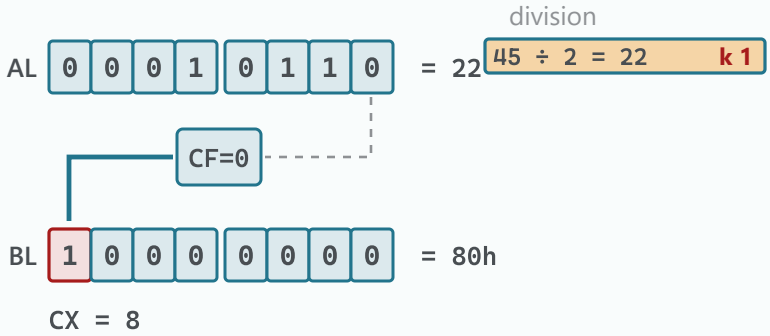
VARIABLES

`AL = 16h` `BL = 00h` `CF = 1` `CX = 8`

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
▶ RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



`RCR BL, 1` : CF enters BL from the left (bit 7); the remainder is saved.

Step 4 / 13

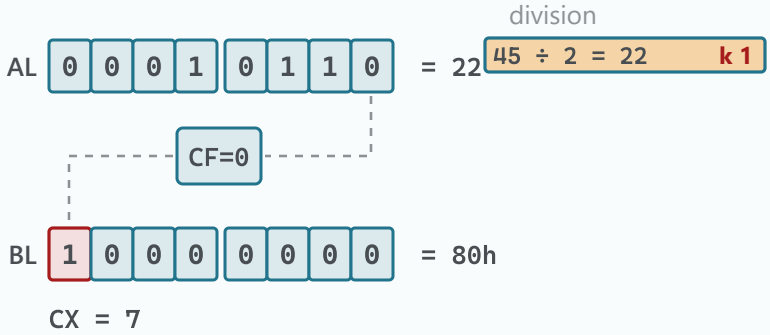
VARIABLES

`AL = 16h` `BL = 80h` `CF = 0` `CX = 8`

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



`LOOP` : CX = 7 ≠ 0 → back to the `bit` label.

Step 5 / 13

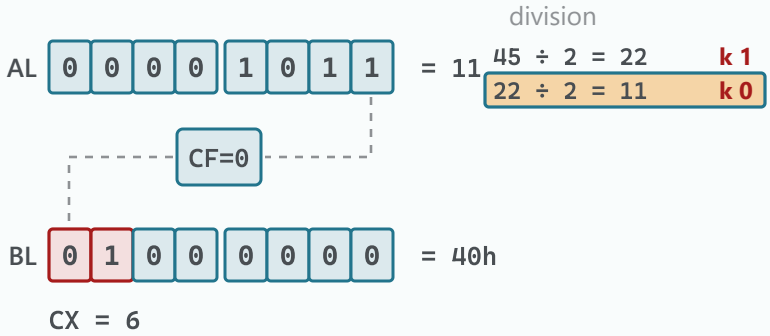
VARIABLES

`AL = 16h` `BL = 80h` `CF = 0` `CX = 7`

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 2 (summary): 22 ÷ 2 = 11, remainder **0**. The earlier remainders shift one place right in BL; CX = 6.

Step 6 / 13

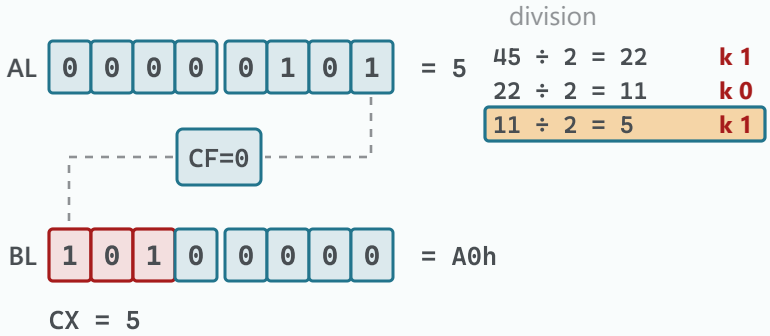
VARIABLES

AL = 0Bh | BL = 40h | CF = 0 | CX = 6

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 3 (summary): 11 ÷ 2 = 5, remainder **1**. The earlier remainders shift one place right in BL; CX = 5.

Step 7 / 13

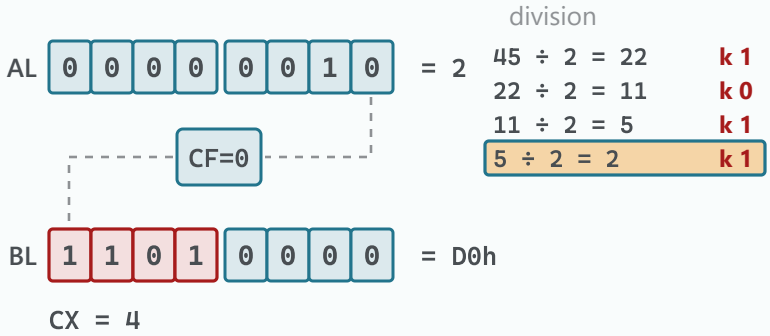
VARIABLES

AL = **05h** BL = **A0h** CF = **0** CX = **5**

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 4 (summary): 5 ÷ 2 = 2, remainder **1**. The earlier remainders shift one place right in BL; CX = 4.

Step 8 / 13

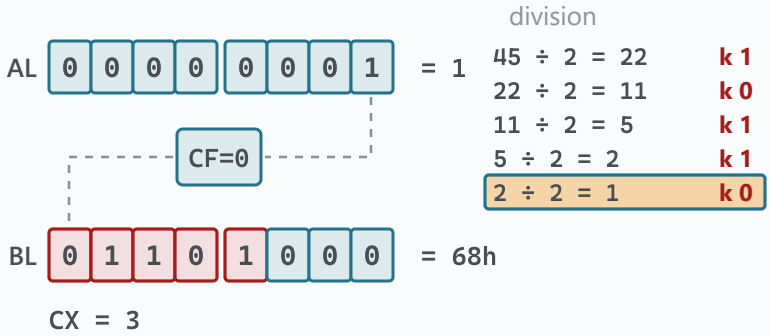
VARIABLES

AL = 02h | BL = D0h | CF = 0 | CX = 4

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 5 (summary): 2 ÷ 2 = 1, remainder **0**. The earlier remainders shift one place right in BL; CX = 3.

Step 9 / 13

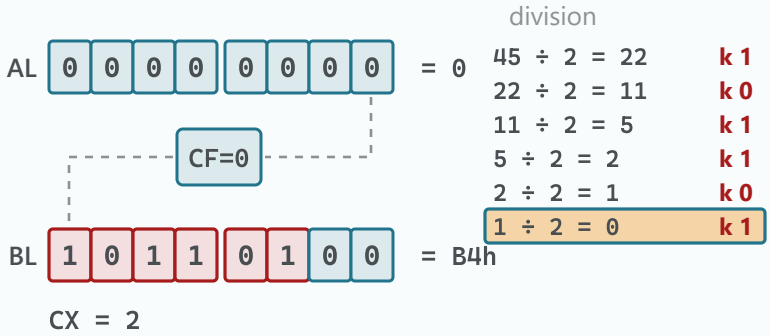
VARIABLES

AL = 01h | BL = 68h | CF = 0 | CX = 3

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 6 (summary): 1 ÷ 2 = 0, remainder **1**. The earlier remainders shift one place right in BL; CX = 2.

Step 10 / 13

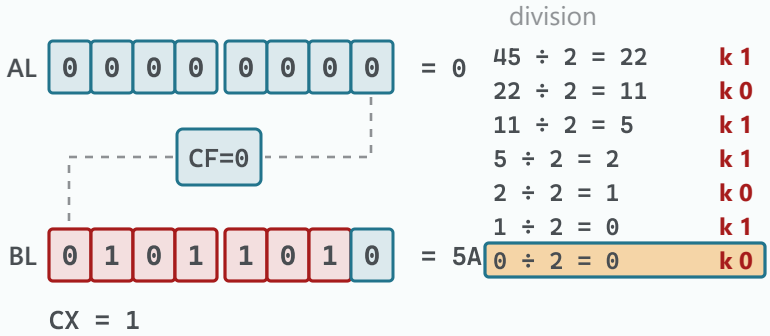
VARIABLES

AL = **00h** BL = **B4h** CF = **0** CX = **2**

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 7 (summary): 0 ÷ 2 = 0, remainder **0**. The earlier remainders shift one place right in BL; CX = 1. Leading zeros.

Step 11 / 13

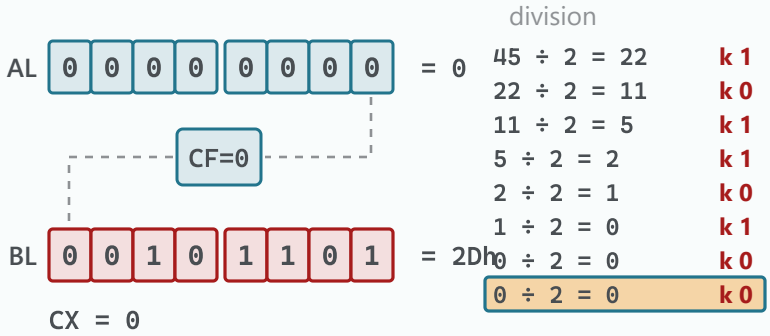
VARIABLES

AL = **00h** BL = **5Ah** CF = **0** CX = **1**

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
7 RET
```



Pass 8 (summary): 0 ÷ 2 = 0, remainder **0**. The earlier remainders shift one place right in BL; CX = 0. Leading zeros.

Step 12 / 13

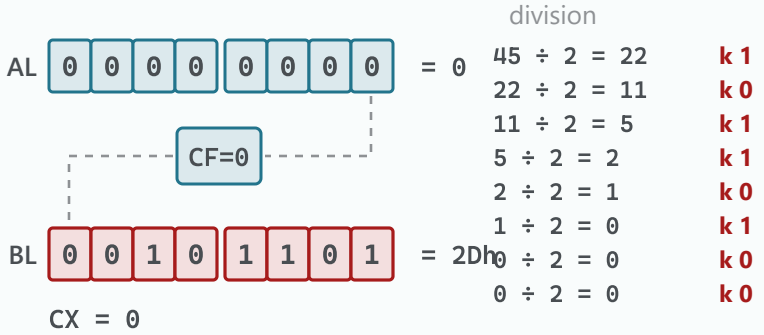
VARIABLES

AL = **00h** BL = **2Dh** CF = **0** CX = **0**

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

Code Trace: The Remainders Come from CF

```
1 ORG 100h
2 MOV AL, 45      ; dividend
3 MOV CX, 8       ; 8 bits
4 bit: SHR AL, 1 ; rem → CF
5 RCR BL, 1       ; CF → BL
6 LOOP bit
▶ RET
```



The first remainder reached bit 0 of BL, the last one bit 7: BL = **0010 1101b** = 2Dh = 45. Reading the remainders last to first gives the binary form.

Step 13 / 13

VARIABLES

AL = **00h** BL = **2Dh** CF = **0** CX = **0**

To divide by a power of 2, a shift is enough: `SHR` gives the quotient, CF the remainder. That is why binary digits are the remainders of dividing by 2.

The Fractional Part: Multiplying by 2

- **Multiply** the fractional part **by 2**; the integer part of the product (0 or 1) is one **bit**.
- Continue with the remaining fraction; stop when the fraction is **0** or you have enough bits.
- Convert the integer and fractional parts separately and join them: 25.625 = 11001b + 0.101b = **11001.101b**.

Example: 0.625 → binary. Multiply the fractional part by 2 again and again.

The Fractional Part: Multiplying by 2

- **Multiply** the fractional part **by 2**; the integer part of the product (0 or 1) is one **bit**.
- Continue with the remaining fraction; stop when the fraction is **0** or you have enough bits.
- Convert the integer and fractional parts separately and join them: 25.625 = 11001b + 0.101b = **11001.101b**.

`0.625 × 2  = 1.25` bit **1** fraction left 0.25

0.625 × 2 = 1.25: integer part **1** → the first fraction bit (2^{-1}). Continue with 0.25.

The Fractional Part: Multiplying by 2

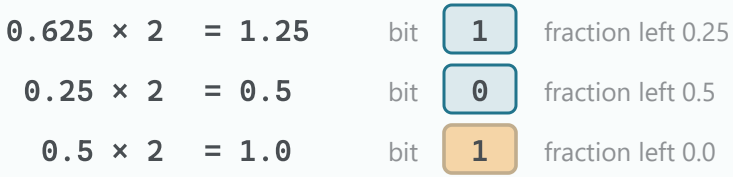
- **Multiply** the fractional part **by 2**; the integer part of the product (0 or 1) is one **bit**.
- Continue with the remaining fraction; stop when the fraction is **0** or you have enough bits.
- Convert the integer and fractional parts separately and join them: 25.625 = 11001b + 0.101b = **11001.101b**.

`0.625 × 2 = 1.25`	bit	`1`	fraction left 0.25
`0.25 × 2 = 0.5`	bit	`0`	fraction left 0.5

0.25 × 2 = 0.5: integer part **0** → the 2^{-2} bit.

The Fractional Part: Multiplying by 2

- **Multiply** the fractional part **by 2**; the integer part of the product (0 or 1) is one **bit**.
- Continue with the remaining fraction; stop when the fraction is **0** or you have enough bits.
- Convert the integer and fractional parts separately and join them: 25.625 = 11001b + 0.101b = **11001.101b**.



0.5 × 2 = 1.0: integer part **1**, fraction left 0 → stop.

The Fractional Part: Multiplying by 2

- **Multiply** the fractional part **by 2**; the integer part of the product (0 or 1) is one **bit**.
- Continue with the remaining fraction; stop when the fraction is **0** or you have enough bits.
- Convert the integer and fractional parts separately and join them: 25.625 = 11001b + 0.101b = **11001.101b**.

`0.625 × 2 = 1.25`	bit	`1`	fraction left 0.25
`0.25 × 2 = 0.5`	bit	`0`	fraction left 0.5
`0.5 × 2 = 1.0`	bit	`1`	fraction left 0.0

read top down

`0.101b`

Result **0.101b**. Check: 1/2 + 1/8 = 0.5 + 0.125 = 0.625 ✓ The bits are read **top down** (the opposite of the division method).

0.1 → Binary: A Repeating Fraction

0.1 → binary: the same method.

Step 1 / 8

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

BASE CONVERSION · STEP BY STEP

0.1 → Binary: A Repeating Fraction

`0.1 × 2  = 0.2` bit **0** fraction left 0.2

$$0.1 \times 2 = 0.2 \rightarrow \mathbf{0}.$$

Step 2 / 8

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

0.1 → Binary: A Repeating Fraction

`0.1 × 2 = 0.2`	bit	`0`	fraction left 0.2
`0.2 × 2 = 0.4`	bit	`0`	fraction left 0.4

$$0.2 \times 2 = 0.4 \rightarrow \mathbf{0}.$$

Step 3 / 8

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

0.1 → Binary: A Repeating Fraction

`0.1 × 2 = 0.2`	bit	`0`	fraction left 0.2
`0.2 × 2 = 0.4`	bit	`0`	fraction left 0.4
`0.4 × 2 = 0.8`	bit	`0`	fraction left 0.8

$0.4 \times 2 = 0.8 \rightarrow \mathbf{0}.$

Step 4 / 8

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

0.1 → Binary: A Repeating Fraction

`0.1 × 2 = 0.2`	bit	`0`	fraction left 0.2
`0.2 × 2 = 0.4`	bit	`0`	fraction left 0.4
`0.4 × 2 = 0.8`	bit	`0`	fraction left 0.8
`0.8 × 2 = 1.6`	bit	`1`	fraction left 0.6

0.8 × 2 = 1.6 → **1**, fraction 0.6.

Step 5 / 8

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

0.1 → Binary: A Repeating Fraction

`0.1 × 2 = 0.2`	bit	`0`	fraction left 0.2
`0.2 × 2 = 0.4`	bit	`0`	fraction left 0.4
`0.4 × 2 = 0.8`	bit	`0`	fraction left 0.8
`0.8 × 2 = 1.6`	bit	`1`	fraction left 0.6
`0.6 × 2 = 1.2`	bit	`1`	fraction left 0.2

0.6 × 2 = 1.2 → **1**, fraction **0.2**: we have seen this fraction before (row 2)!

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

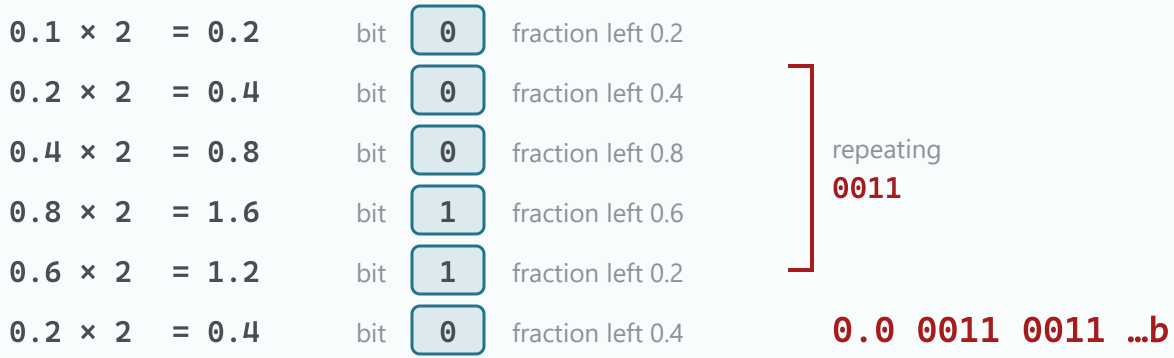
0.1 → Binary: A Repeating Fraction

0.1 × 2 = 0.2	bit	0	fraction left 0.2
0.2 × 2 = 0.4	bit	0	fraction left 0.4
0.4 × 2 = 0.8	bit	0	fraction left 0.8
0.8 × 2 = 1.6	bit	1	fraction left 0.6
0.6 × 2 = 1.2	bit	1	fraction left 0.2
0.2 × 2 = 0.4	bit	0	fraction left 0.4

0.2 is multiplied again: rows 2–5 repeat forever.

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

0.1 → Binary: A Repeating Fraction



Result **0.1 = 0.0 0011 0011 0011…b**: it **does not terminate** in binary. With 8 fraction bits, 0.00011001b = 0.09765625 is stored.

Fractions whose denominator is not a power of 2 (0.1 = 1/10) cannot be written exactly in binary; with a limited number of bits they are rounded. IEEE 754 also stores 0.1 approximately, which is why 0.1 + 0.2 ≠ 0.3 in many languages.

BASE CONVERSION · INTERACTIVE

Base Converter

number 43868 input base 10 (decimal) division steps divide by 16

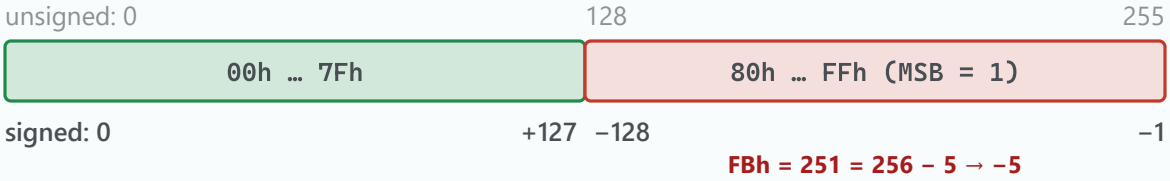
decimal	hex	octal	binary
43868	**0AB5Ch**	**125534o**	**1010 1011 0101 1100b**

```
0000043868 ÷ 16 = 0000002741   rem 12 = C
0000002741 ÷ 16 = 0000000171   rem 5
0000000171 ÷ 16 = 0000000010   rem 11 = B
0000000010 ÷ 16 = 0000000000   rem 10 = A
```

Read the remainders bottom-up → **AB5C** (base 16)

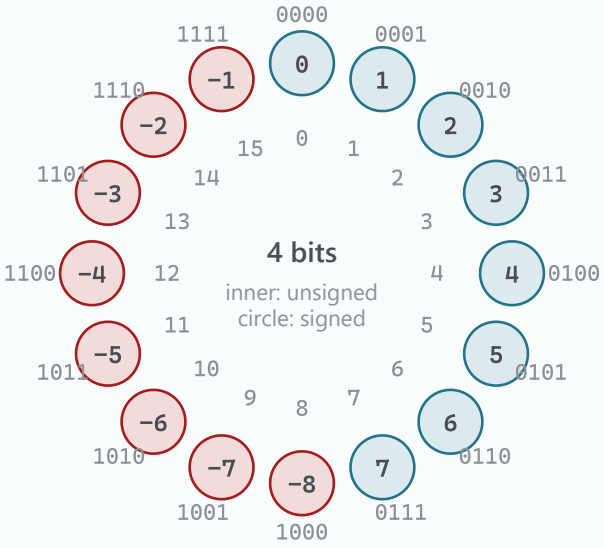
Signed Numbers

- Is a number like `0FFh` positive or negative? The bit pattern alone **cannot tell you**: it could be **255** or **−1**.
- 8 bits give 256 different combinations (including zero).
- The first 128 combinations (**0 … 127**) stand for positive numbers, the next 128 (**128 … 255**) for negative numbers.
- To get −5, subtract 5 from the number of combinations (256): 256 − 5 = **251** = FBh.
- General rule (n bits): $-x \leftrightarrow 2^n - x$. This representation is called **two's complement**.



The largest 8-bit combination is **255** (0 … 255 = 256 values). In 16 bits the negative range is likewise 32768 … **65535**.

The Number Circle



- All 4-bit patterns sit on a circle: moving clockwise means **+1**.
- Between 0111 (7) and 1000 the signed reading jumps **from +7 to −8** → signed overflow (OF).
- Between 1111 and 0000 the unsigned reading jumps **from 15 to 0** → carry (CF).
- Every negative number has its top bit (MSB) set to **1**: the MSB is read as the sign bit.
- The add and subtract circuit is **the same** for both readings; only the flags tell the readings apart.

4-Bit Representations

Binary	Un-signed	Sign-mag.	ones' compl.	two's compl.
`0000`	0	0	0	**0**
`0001`	1	1	1	**1**
`0010`	2	2	2	**2**
`0011`	3	3	3	**3**
`0100`	4	4	4	**4**
`0101`	5	5	5	**5**
`0110`	6	6	6	**6**
`0111`	7	7	7	**7**

Binary	Un-signed	Sign-mag.	ones' compl.	two's compl.
`1000`	8	−0	−7	**−8**
`1001`	9	−1	−6	**−7**
`1010`	10	−2	−5	**−6**
`1011`	11	−3	−4	**−5**
`1100`	12	−4	−3	**−4**
`1101`	13	−5	−2	**−3**
`1110`	14	−6	−1	**−2**
`1111`	15	−7	−0	**−1**

The middle column of the table is **sign-magnitude** (1000 = −0, 1111 = −7). The 8086 and all modern processors use **two's complement**: there is a single zero and the range is −8 … +7.

The Ones' Complement Method

- **Ones' complement**: to negate a number, **flip all its bits** (1 → 0, 0 → 1).
- 8-bit example: 00101101 → 11010010.
- The catch: there are **two zeros** (00000000 = +0, 11111111 = −0), and after an addition the carry must be added back (end-around carry).
- On the 8086 the `NOT` instruction produces the ones' complement.

Start.

The Ones' Complement Method

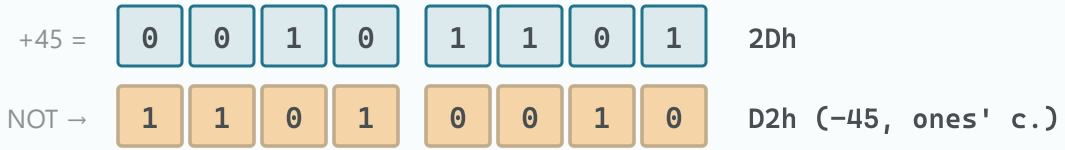
- **Ones' complement**: to negate a number, **flip all its bits** (1 → 0, 0 → 1).
- 8-bit example: 00101101 → 11010010.
- The catch: there are **two zeros** (00000000 = +0, 11111111 = −0), and after an addition the carry must be added back (end-around carry).
- On the 8086 the `NOT` instruction produces the ones' complement.



+45 = 0010 1101b.

The Ones' Complement Method

- **Ones' complement**: to negate a number, **flip all its bits** (1 → 0, 0 → 1).
- 8-bit example: 00101101 → 11010010.
- The catch: there are **two zeros** (00000000 = +0, 11111111 = −0), and after an addition the carry must be added back (end-around carry).
- On the 8086 the `NOT` instruction produces the ones' complement.



All bits are flipped: 1101 0010b = D2h. In ones' complement this is −45; in two's complement it is −46!

The Two's Complement Method

- The standard way to represent negative numbers: **flip all the bits, then add 1**.
- On the 8086 the `NEG` instruction does this in one step (0 − x).
- Shortcut: keep the bits up to and including the first **1** from the right; flip every bit to its left.

Start.

Step 1 / 4

Check: D3h + 2Dh = 100h → 00h in 8 bits. The two's complement of a two's complement is the number itself.

SIGNED NUMBERS

The Two's Complement Method

- The standard way to represent negative numbers: **flip all the bits, then add 1**.
- On the 8086 the `NEG` instruction does this in one step (0 − x).
- Shortcut: keep the bits up to and including the first **1** from the right; flip every bit to its left.



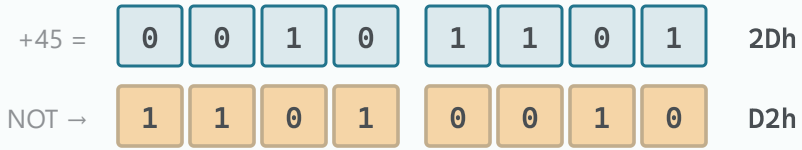
+45 = 0010 1101b.

Step 2 / 4

Check: D3h + 2Dh = 100h → 00h in 8 bits. The two's complement of a two's complement is the number itself.

The Two's Complement Method

- The standard way to represent negative numbers: **flip all the bits, then add 1**.
- On the 8086 the `NEG` instruction does this in one step (0 − x).
- Shortcut: keep the bits up to and including the first **1** from the right; flip every bit to its left.

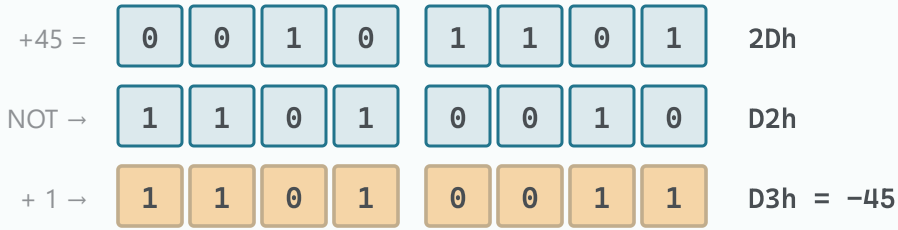


All bits are flipped (NOT): 1101 0010b.

Check: D3h + 2Dh = 100h → 00h in 8 bits. The two's complement of a two's complement is the number itself.

The Two's Complement Method

- The standard way to represent negative numbers: **flip all the bits, then add 1**.
- On the 8086 the `NEG` instruction does this in one step (0 − x).
- Shortcut: keep the bits up to and including the first **1** from the right; flip every bit to its left.



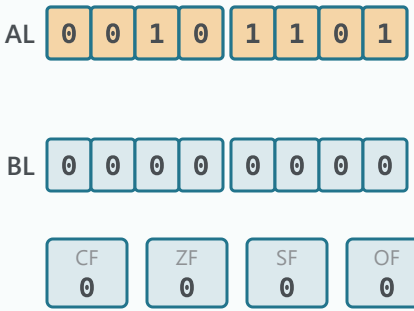
Add 1: 1101 0011b = **D3h** = 211 = 256 − 45 ✓ This is the 8-bit form of −45.

Step 4 / 4

Check: D3h + 2Dh = 100h → 00h in 8 bits. The two's complement of a two's complement is the number itself.

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
▶  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



AL = +45 = **0010 1101b** (2Dh).

Step 1 / 9

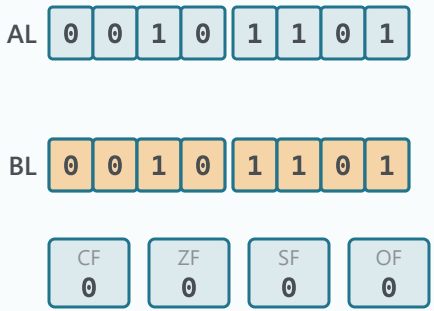
VARIABLES

AX = **002Dh** · BL = **00h** · CF = **0** · ZF = **0** · SF = **0**

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
▶  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



BL = AL: a copy of the same value; the complement will be taken on BL in two steps.

Step 2 / 9

VARIABLES

AX = 002Dh | BL = 2Dh | CF = 0 | ZF = 0 | SF = 0

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
▶  NOT BL            ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



`NOT BL` : every bit flips → **1101 0010b** = D2h (ones' complement). NOT does not change the flags.

Step 3 / 9

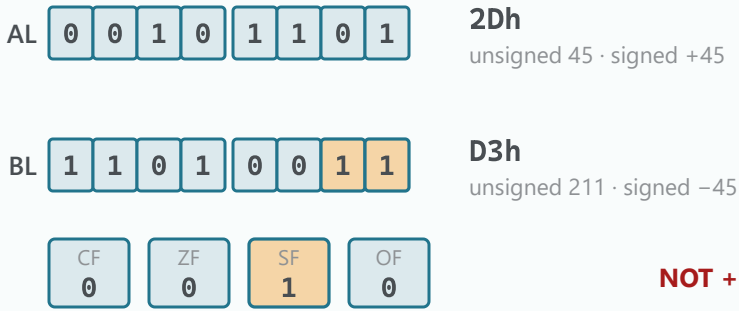
VARIABLES

`AX = 002Dh` `BL = D2h` `CF = 0` `ZF = 0` `SF = 0`

$-x$ = NOT x + 1 = $2^n - x$. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL            ; ones' comp.
▶  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



`INC BL` : + 1 → **1101 0011b** = D3h. Unsigned 211 = 256 − 45; signed **−45**.

Step 4 / 9

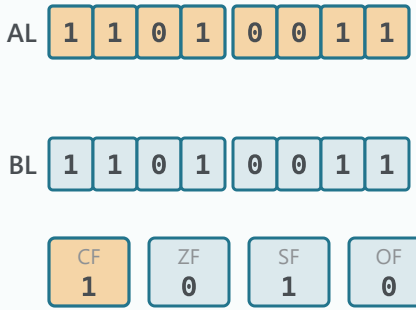
VARIABLES

AX = 002Dh | BL = D3h | CF = 0 | ZF = 0 | SF = 1

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
▶  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



`NEG AL` (0 − AL) gives the same result in one instruction: AL = **D3h**. The operand ≠ 0 → CF = 1.

Step 5 / 9

VARIABLES

AX = 00D3h | BL = D3h | CF = 1 | ZF = 0 | SF = 1

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
▶  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



`CMP AL, BL`: D3h − D3h = 0 → **ZF = 1**. Both routes gave the same bit pattern.

Step 6 / 9

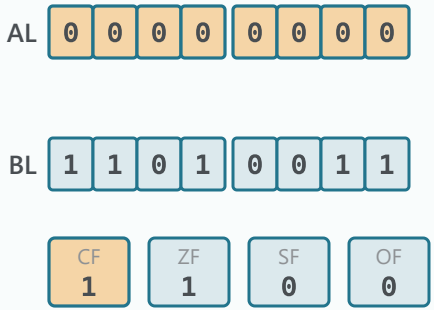
VARIABLES

AX = **00D3h** | BL = **D3h** | CF = **0** | ZF = **1** | SF = **0**

$-x = \text{NOT } x + 1 = 2^n - x$. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
▶  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



`ADD AL, 45`: D3h + 2Dh = 100h. Bit 9 falls out (CF = 1), AL = **00h**, ZF = 1: −45 + 45 = 0 ✓

Step 7 / 9

VARIABLES

AX = **0000h** · BL = **D3h** · CF = **1** · ZF = **1** · SF = **0**

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
▶  MOV AX, 45
10 NEG AX            ; -45 in 16 bits
11 RET
```



AX = 45 = **002Dh**: the same number in 16 bits.

Step 8 / 9

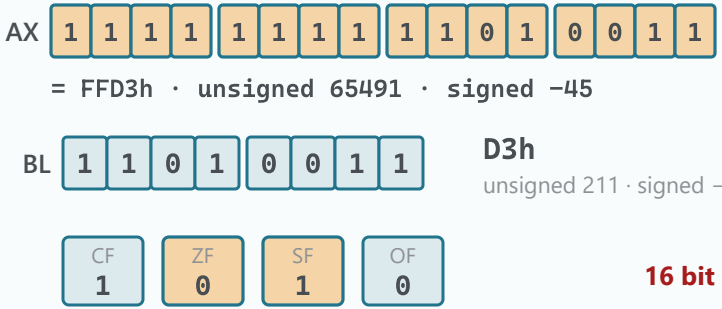
VARIABLES

AX = **002Dh** | BL = **D3h** | CF = **1** | ZF = **1** | SF = **0**

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

Code Trace: NOT + 1 and NEG

```
1  ORG 100h
2  MOV AL, 45        ; +45
3  MOV BL, AL
4  NOT BL          ; ones' comp.
5  INC BL            ; + 1
6  NEG AL            ; one instr.
7  CMP AL, BL        ; equal?
8  ADD AL, 45        ; -45 + 45
9  MOV AX, 45
▶  NEG AX            ; -45 in 16 bits
11 RET
```



`NEG AX` : **FFD3h** = 65536 − 45. The low byte is D3h again; the high byte is FFh, MSB = 1 → negative (SF = 1).

Step 9 / 9

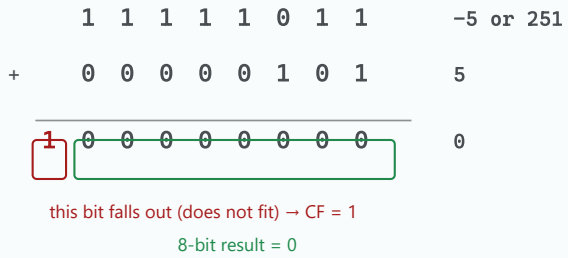
VARIABLES

AX = **FFD3h** · BL = **D3h** · CF = **1** · ZF = **0** · SF = **1**

−x = NOT x + 1 = 2^n − x. NEG does it in one instruction; the same rule works at every width (8, 16 bits).

−5 + 5 = 0

- The method must agree with the arithmetic rule **−5 + 5 = 0**.
- Adding the bytes 5 and 251 gives more than 255 (256).
- The ninth bit does not fit in the byte and **falls out**; the remaining 8 bits are **zero**.
- So the same adder gives the right answer for both unsigned and signed numbers.



Signed Ranges

Width	Combinations	Unsigned	Signed (two's complement)	Negatives
8 bits	256	0 … 255	−128 … +127	80h … FFh
16 bits	65536	0 … 65535	−32768 … +32767	8000h … FFFFh
32 bits	4,294,967,296	0 … 4,294,967,295	−2,147,483,648 … +2,147,483,647	80000000h …

- In the upper half of the combinations the **top bit is always 1**; this bit gives the sign of the number (the SF flag).
- 16 bits give 65536 combinations: the first 32768 (0 … 32767) are positive, the other 32768 (32768 … 65535) negative.
- The negative range has one more value than the positive: the positive of −128 **does not fit** in 8 bits (NEG 80h = 80h, OF = 1).
- **Sign extension**: when an 8-bit signed number moves to 16 bits, the MSB is copied to the left: FBh (−5) → FFFBh. 8086: `CBW` (AL → AX), `CWD` (AX → DX:AX).

Fixed-Point Signed Fractions

Let the last 3 of 8 bits be fraction bits: the **Q5.3** format (5 integer bits including the sign + 3 fraction bits). Example: **−6.75**.



Digit weights: the leftmost bit is **−16** (the sign bit); after the point ½, ¼, ⅛. Range −16 … +15.875, step 1/8 = 0.125.

Step 1 / 5

To the processor CAh is an ordinary byte; where the point sits is the programmer's convention. Addition works unchanged: −6.75 + 2.5 → CAh + 14h = **DEh** = 11011.110b = −16 + 11.75 = **−4.25** ✓

Fixed-Point Signed Fractions

Let the last 3 of 8 bits be fraction bits: the **Q5.3** format (5 integer bits including the sign + 3 fraction bits). Example: **−6.75**.



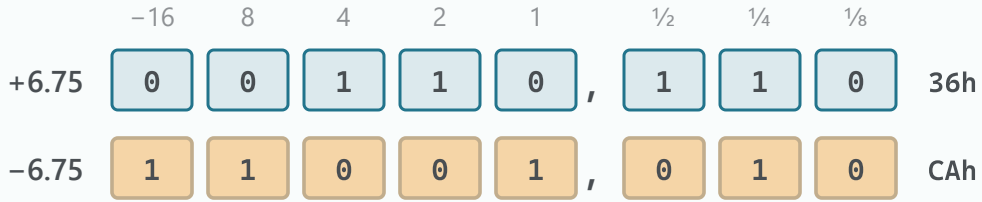
+6.75 = 4 + 2 + ½ + ¼ → **00110.110**.

Step 2 / 5

To the processor CAh is an ordinary byte; where the point sits is the programmer's convention. Addition works unchanged: −6.75 + 2.5 → CAh + 14h = **DEh** = 11011.110b = −16 + 11.75 = **−4.25** ✓

Fixed-Point Signed Fractions

Let the last 3 of 8 bits be fraction bits: the **Q5.3** format (5 integer bits including the sign + 3 fraction bits). Example: **−6.75**.

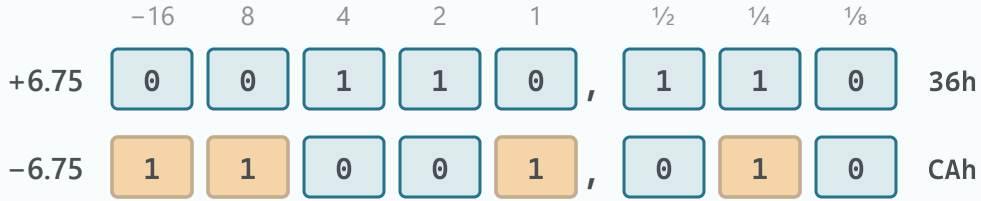


Two's complement (flip + 1; ignore the point): 00110110 → 11001001 + 1 = **11001.010**.

To the processor CAh is an ordinary byte; where the point sits is the programmer's convention. Addition works unchanged: −6.75 + 2.5 → CAh + 14h = **DEh** = 11011.110b = −16 + 11.75 = **−4.25** ✓

Fixed-Point Signed Fractions

Let the last 3 of 8 bits be fraction bits: the **Q5.3** format (5 integer bits including the sign + 3 fraction bits). Example: **−6.75**.



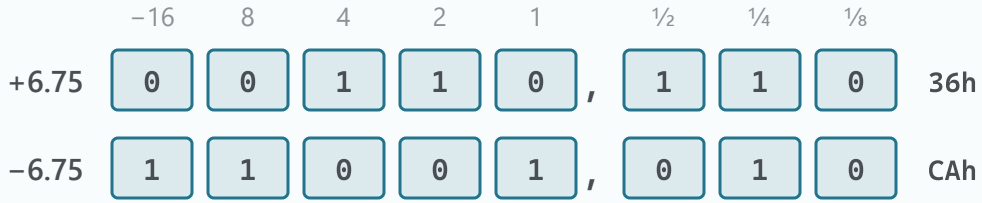
−16 + 8 + 1 + ¼ = −6.75 ✓

Check: the sign bit is worth −16: −16 + 8 + 1 + 0.25 = **−6.75** ✓

To the processor CAh is an ordinary byte; where the point sits is the programmer's convention. Addition works unchanged: −6.75 + 2.5 → CAh + 14h = **DEh** = 11011.110b = −16 + 11.75 = **−4.25** ✓

Fixed-Point Signed Fractions

Let the last 3 of 8 bits be fraction bits: the **Q5.3** format (5 integer bits including the sign + 3 fraction bits). Example: **−6.75**.



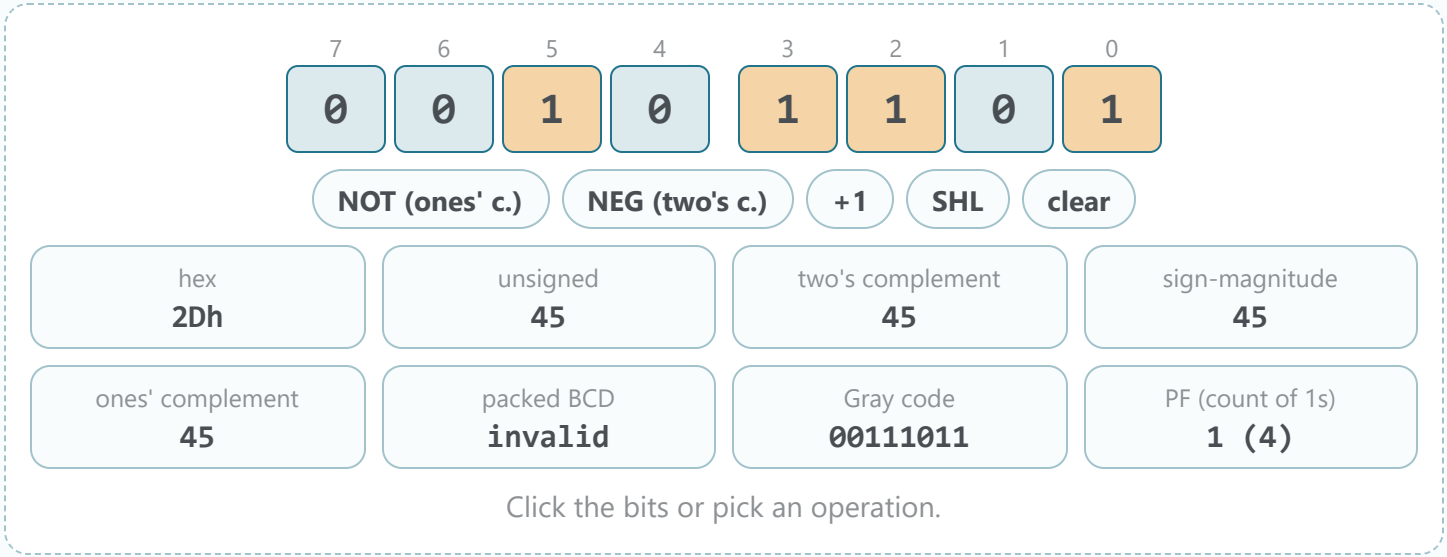
$-6.75 \times 2^3 = -54 \rightarrow 256 - 54 = 202 = \text{CAh}$

Shortcut: multiply by $2^3 = 8$ and take the two's complement as for an integer: −54 → **CAh**, then put the point 3 bits from the right.

To the processor CAh is an ordinary byte; where the point sits is the programmer's convention. Addition works unchanged: −6.75 + 2.5 → CAh + 14h = **DEh** = 11011.110b = −16 + 11.75 = **−4.25** ✓

SIGNED NUMBERS · INTERACTIVE

Bit Explorer



Click a bit to flip it; the unsigned, signed, BCD and Gray code readings of the same 8 bits appear together.

Binary Addition

- Each digit pair has four cases: 0+0, 0+1, 1+0 and 1+1.
- Starting from the lowest digit, add the matching digits of the two numbers.
- A digit sum of 0 or 1 is written directly.
- A sum of 2 (10b) produces a **carry**: write 0 and add 1 to the next digit.
- With the previous carry the sum can be 3 (11b): write 1, carry 1.

a	b	$carry_{in}$	sum	$carry_{out}$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	**1**
1	1	1	1	**1**

`0 + 0 = 0`	carry 0, write 0
`0 + 1 = 1`	carry 0, write 1
`1 + 0 = 1`	carry 0, write 1
`1 + 1 = 10`	**carry 1**, write 0

This table is the truth table of a **full adder**; chain 8 of them and you get an 8-bit adder.

Addition: 110001101 + 100001011

carry									
397		1	1	0	0	0	1	1	0	1
+ 267		1	0	0	0	0	1	0	1	1

110001101b (397) + 100001011b (267). Add column by column, right to left.

Step 1 / 11

Addition: 110001101 + 100001011

carry									
397		1	1	0	0	0	1	1	0	1
+ 267		1	0	0	0	0	1	0	1	1
										0

Digit 0: 1 + 1 = 2 → write **0**, carry 1.

Addition: 110001101 + 100001011

carry								1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
								0	0

Digit 1: 0 + 1 + 1 (carry) = 2 → write **0**, carry 1.

Addition: 110001101 + 100001011

carry							1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
							0	0	0

Digit 2: 1 + 0 + 1 (carry) = 2 → write **0**, carry 1.

Addition: 110001101 + 100001011

carry						1	1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
						1	0	0	0

Digit 3: 1 + 1 + 1 (carry) = 3 → write **1**, carry 1.

Addition: 110001101 + 100001011

carry					1	1	1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
					1	1	0	0	0

Digit 4: 0 + 0 + 1 (carry) = 1 → write **1**.

Addition: 110001101 + 100001011

carry					1	1	1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
				0	1	1	0	0	0

Digit 5: 0 + 0 = 0 → write **0**.

Addition: 110001101 + 100001011

carry					1	1	1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
			0	0	1	1	0	0	0

Digit 6: 0 + 0 = 0 → write **0**.

Addition: 110001101 + 100001011

carry					1	1	1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
		1	0	0	1	1	0	0	0

Digit 7: 1 + 0 = 1 → write **1**.

Addition: 110001101 + 100001011

carry					1	1	1	1	
397	1	1	0	0	0	1	1	0	1
+ 267	1	0	0	0	0	1	0	1	1
	0	1	0	0	1	1	0	0	0

Digit 8: 1 + 1 = 2 → write **0**, carry 1.

Addition: 110001101 + 100001011

carry	1					1	1	1	1	
397		1	1	0	0	0	1	1	0	1
+ 267		1	0	0	0	0	1	0	1	1
= 664	1	0	1	0	0	1	1	0	0	0

Digit 9: 0 + 0 + 1 (carry) = 1 → write **1**. Result **1010011000b** = 664 ✓ (the 10th bit did not fit the 9-bit addition → carry).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.

The first number is 1011 (11), the second 0110 (6).

Step 1 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.



A = 1011.

Step 2 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.

A =	1 0 1 1	11
B =	0 1 1 0	6

B = 0110.

Step 3 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.

A =	1 0 1 1	11
B =	0 1 1 0	6
NOT B =	1 0 0 1	ones' complement

Flip the bits of B: 1001.

Step 4 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.

A =	1 0 1 1	11
B =	0 1 1 0	6
NOT B =	1 0 0 1	ones' complement
+1 →	1 0 1 0	−B (two's c.)

Add 1: **1010** = −6 (4 bits).

Step 5 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.

A =		1	0	1	1	11
B =		0	1	1	0	6
NOT B =		1	0	0	1	ones' complement
+1 →		1	0	1	0	−B (two's c.)
A + (−B) =	~~1~~	0	1	0	1	5 bits: the leftmost carry is dropped

1011 + 1010 = **10101** (5 bits).

Step 6 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Binary Subtraction

- Subtraction becomes **addition of the complement**: A − B = A + (−B).
- Flip every bit of the number being subtracted.
- Add 1 to the complemented number (two's complement).
- Add the result to the first number; a carry beyond the bit width is **discarded**.
- So the processor needs no separate subtractor circuit.

A =		1	0	1	1	11
B =		0	1	1	0	6
NOT B =		1	0	0	1	ones' complement
+1 →		1	0	1	0	−B (two's c.)
A + (−B) =	~~1~~	0	1	0	1	5 bits: the leftmost carry is dropped
result =		0	1	0	1	5 ✓

The leftmost 1 that does not fit in 4 bits is dropped: **0101** = 5 = 11 − 6 ✓

Step 7 / 7

On the 8086, SUB writes this dropped carry to CF inverted: a carry of 1 means no borrow (CF = 0).

Hex Addition: 9A7Ch + 5B96h

carry					
	9	A	7	C	9A7Ch = 39548
+	5	B	9	6	5B96h = 23446

Hex addition: digit by digit, right to left. If a digit sum exceeds 15, subtract 16 and **carry 1** to the left.

Step 1 / 6

Think of digit values in decimal (A = 10 … F = 15). If a sum is 16 or more, write sum − 16 in that digit.

Hex Addition: 9A7Ch + 5B96h

carry			1		
	9	A	7	C	C + 6 = 12 + 6 = 18
+	5	B	9	6	18 − 16 = 2, carry 1
				2	

C + 6 = 18 ≥ 16 → write **2**, carry 1.

Step 2 / 6

Think of digit values in decimal (A = 10 … F = 15). If a sum is 16 or more, write sum − 16 in that digit.

Hex Addition: 9A7Ch + 5B96h

carry		1	1		
	9	A	7	C	7 + 9 + 1 = 17
+	5	B	9	6	17 − 16 = 1, carry 1
			1	2	

7 + 9 + 1 = 17 → write **1**, carry 1.

Step 3 / 6

Think of digit values in decimal (A = 10 … F = 15). If a sum is 16 or more, write sum − 16 in that digit.

Hex Addition: 9A7Ch + 5B96h

carry	1	1	1		
	9	A	7	C	A + B + 1 = 10 + 11 + 1 = 22
+	5	B	9	6	22 − 16 = 6, carry 1
		6	1	2	

A + B + 1 = 22 → write **6**, carry 1.

Step 4 / 6

Think of digit values in decimal (A = 10 … F = 15). If a sum is 16 or more, write sum − 16 in that digit.

Hex Addition: 9A7Ch + 5B96h

carry	1	1	1		
	9	A	7	C	9 + 5 + 1 = 15 = F
+	5	B	9	6	below 16, no carry
	F	6	1	2	

9 + 5 + 1 = 15 → write **F**, no carry.

Step 5 / 6

Think of digit values in decimal (A = 10 … F = 15). If a sum is 16 or more, write sum − 16 in that digit.

Hex Addition: 9A7Ch + 5B96h

carry	1	1	1		
	9	A	7	C	Check: 39548 + 23446
+	5	B	9	6	= 62994 ✓
	F	6	1	2	= F612h

Result **F612h**. On the 8086, `ADD AX, BX` gives the same result; no carry leaves the last digit, so CF = 0.

Step 6 / 6

Think of digit values in decimal (A = 10 … F = 15). If a sum is 16 or more, write sum − 16 in that digit.

Hex Subtraction: 9A7Ch − 5B96h

borrow					
	9	A	7	C	If the top digit is smaller,
−	5	B	9	6	borrow 16 from the left

Hex subtraction: if the top digit is too small, **borrow 1** from the digit to its left; that adds 16 to this digit.

Step 1 / 6

The "−1" marks show the digits that lent. The same operation can also be done as addition of the complement.

Hex Subtraction: 9A7Ch − 5B96h

borrow

	9	A	7	C	C − 6 = 12 − 6 = 6
−	5	B	9	6	no borrow
				6	

C − 6 = **6**.

Step 2 / 6

The "−1" marks show the digits that lent. The same operation can also be done as addition of the complement.

Hex Subtraction: 9A7Ch − 5B96h

borrow		−1			
	9	A	7	C	7 − 9 fails → borrow 16
−	5	B	9	6	7 + 16 − 9 = 14 = E
			E	6	

7 < 9 → borrow from the left: 7 + 16 − 9 = 14 → **E**; the A on the left becomes 9.

The "−1" marks show the digits that lent. The same operation can also be done as addition of the complement.

Hex Subtraction: 9A7Ch − 5B96h

borrow	−1	−1			
	9	A	7	C	A − 1 = 9, 9 − B fails
−	5	B	9	6	9 + 16 − 11 = 14 = E
		E	E	6	

9 < B → borrow again: 9 + 16 − 11 = 14 → **E**; the 9 on the left becomes 8.

Step 4 / 6

The "−1" marks show the digits that lent. The same operation can also be done as addition of the complement.

Hex Subtraction: 9A7Ch − 5B96h

borrow	−1	−1			
	9	A	7	C	9 − 1 = 8
−	5	B	9	6	8 − 5 = 3
	3	E	E	6	

8 − 5 = **3**.

Step 5 / 6

The "−1" marks show the digits that lent. The same operation can also be done as addition of the complement.

Hex Subtraction: 9A7Ch − 5B96h

borrow	−1	−1			
	9	A	7	C	Check: 39548 − 23446
−	5	B	9	6	= 16102 ✓
	3	E	E	6	= 3EE6h

Result **3EE6h**. No borrow is needed from the leftmost digit, so after `SUB AX, BX` on the 8086, CF = 0.

Step 6 / 6

The "−1" marks show the digits that lent. The same operation can also be done as addition of the complement.

15's and 16's Complement

15's complement **FFFFh − 5B96h** **= A469h** subtract each digit from F

The **15's complement** of 5B96h: subtract each digit from F (F−5, F−B, F−9, F−6) → A469h. In binary this is NOT.

Step 1 / 5

These are the hex counterparts of the binary ones' and two's complement. On the 8086, `SUB AX, BX` (AX = 5B96h, BX = 9A7Ch) gives **C11Ah** directly, with **CF = 1** because of the borrow; read as signed, C11Ah is −16102.

15's and 16's Complement

15's complement	`FFFFh − 5B96h`	`= A469h`	subtract each digit from F
16's complement	`A469h + 1`	`= A46Ah`	= 10000h − 5B96h

16's complement = 15's complement + 1 = A46Ah. This is −5B96h in 4 digits (NEG).

These are the hex counterparts of the binary ones' and two's complement. On the 8086, `SUB AX, BX` (AX = 5B96h, BX = 9A7Ch) gives **C11Ah** directly, with **CF = 1** because of the borrow; read as signed, C11Ah is −16102.

15's and 16's Complement

15's complement	**FFFFh − 5B96h**	**= A469h**	subtract each digit from F
16's complement	**A469h + 1**	**= A46Ah**	= 10000h − 5B96h
9A7C − 5B96	**9A7Ch + A46Ah**	**= 1 3EE6h**	the overflowing 1 is dropped → 3EE6h ✓

9A7Ch − 5B96h = 9A7Ch + A46Ah = 13EE6h. The 1 beyond 4 digits is dropped: **3EE6h**, the same as direct subtraction.

These are the hex counterparts of the binary ones' and two's complement. On the 8086, `SUB AX, BX` (AX = 5B96h, BX = 9A7Ch) gives **C11Ah** directly, with **CF = 1** because of the borrow; read as signed, C11Ah is −16102.

15's and 16's Complement

15's complement	`FFFFh − 5B96h`	`= A469h`	subtract each digit from F
16's complement	`A469h + 1`	`= A46Ah`	= 10000h − 5B96h
9A7C − 5B96	`9A7Ch + A46Ah`	`= 1 3EE6h`	the overflowing 1 is dropped → 3EE6h ✓
5B96 − 9A7C	`5B96h + 6584h`	`= C11Ah`	no overflow → the result is negative

The other way round: 5B96h − 9A7Ch = 5B96h + 6584h (the 16's complement of 9A7Ch) = C11Ah. There is **no** overflowing digit → the result is negative.

These are the hex counterparts of the binary ones' and two's complement. On the 8086, `SUB AX, BX` (AX = 5B96h, BX = 9A7Ch) gives **C11Ah** directly, with **CF = 1** because of the borrow; read as signed, C11Ah is −16102.

15's and 16's Complement

15's complement	FFFFh − 5B96h	= A469h	subtract each digit from F
16's complement	A469h + 1	= A46Ah	= 10000h − 5B96h
9A7C − 5B96	9A7Ch + A46Ah	= 1 3EE6h	the overflowing 1 is dropped → 3EE6h ✓
5B96 − 9A7C	5B96h + 6584h	= C11Ah	no overflow → the result is negative
	−(16's c. of C11Ah)	= −3EE6h	10000h − C11Ah = 3EE6h

To get the magnitude, take the 16's complement again: C11Ah → 3EE6h, so the result is **−3EE6h** (−16102).

Step 5 / 5

These are the hex counterparts of the binary ones' and two's complement. On the 8086, `SUB AX, BX` (AX = 5B96h, BX = 9A7Ch) gives **C11Ah** directly, with **CF = 1** because of the borrow; read as signed, C11Ah is −16102.

The Signed Overflow Rule

Operation	Signs	Example (8 bits)	Signed reading	OF
A + B	+ with +	`50h + 40h = 90h`	80 + 64 → −112 ✗	**1**
A + B	− with −	`9Ch + CEh = 6Ah`	−100 + (−50) → +106 ✗	**1**
A + B	different	`64h + CEh = 32h`	100 + (−50) = 50 ✓	0 (never)
A − B	+ minus −	`64h - CEh = 96h`	100 − (−50) → −106 ✗	**1**
A − B	− minus +	`9Ch - 32h = 6Ah`	−100 − 50 → +106 ✗	**1**
A − B	same	`9Ch - CEh = CEh`	−100 − (−50) = −50 ✓	0 (never)

- Overflow can happen only when **adding two numbers of the same sign** or **subtracting two numbers of different signs**; the symptom is a result whose sign is the opposite of what you expect.
- Hardware rule: OF = 1 when the **carry into** the sign bit differs from the **carry out of** it. CF means nothing in the signed reading (row 3 has CF = 1 but the result is correct).
- Before adding signed numbers of different widths, **sign-extend** the shorter one: FBh (−5) + 0100h → FFFBh + 0100h = 00FBh = 251 ✓; zero-extending gives 01FBh = 507 ✗.

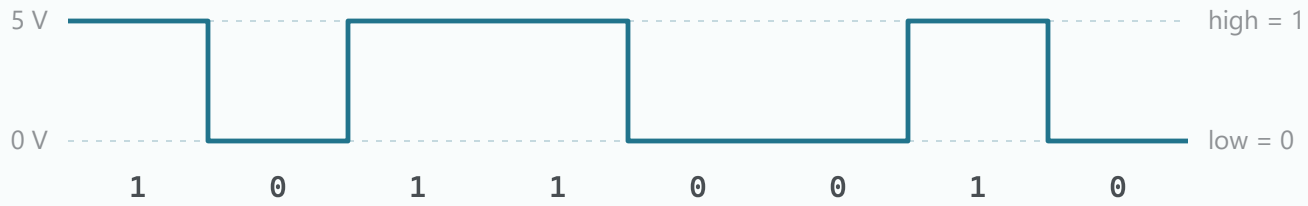
Supported Operators

Op.	Meaning	Example → value
`~`	not (flips all bits)	`~0Fh` → F0h (8 bit)
`*`	multiply	`3*4` → 12
`/`	divide (integer)	`17/5` → 3
`%`	mod (remainder)	`17%5` → 2
`+`	add	`40h+2` → 42h
`-`	subtract (unary −)	`-5` → FBh
`<<`	shift left	`1<<4` → 16
`>>`	shift right	`80h>>3` → 10h
`&`	bitwise AND	`0F0h&3Ch` → 30h
`^`	bitwise XOR	`0Fh^05h` → 0Ah
`|`	bitwise OR	`0F0h|0Fh` → FFh

- Operators the emu8086 assembler supports in **constant expressions**.
- Evaluated **at assembly time**: `MOV AX, 3*4+1` → `MOV AX, 13`.
- The CPU does not run these; their run-time counterparts are the `NOT`, `MUL`, `DIV`, `ADD`, `SUB`, `SHL`, `SHR`, `AND`, `XOR` and `OR` instructions.
- Constants only: `MOV AX, BX*2` is invalid.

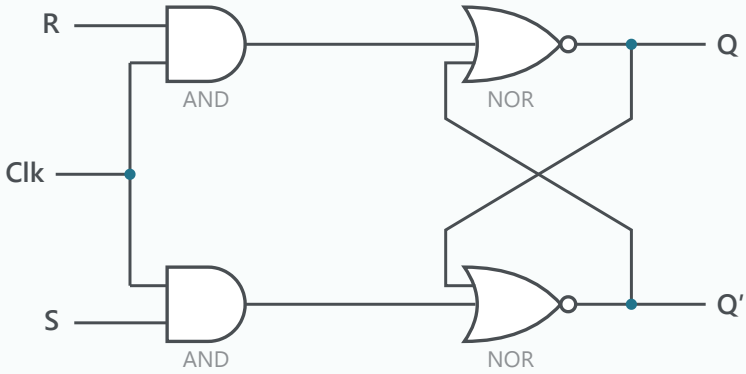
Digital Systems and Binary Signals

- In a digital system, software is a **collection of digits** grouped into data and instructions.
- These digits are physically realized in circuits as **voltage and current**.
- Digital computers use two kinds of signal, **high** and **low**: 1 (high), 0 (low).
- The voltage need not be exactly 5 V or 0 V; there are threshold ranges (TTL: ≥ 2.0 V → 1, ≤ 0.8 V → 0). This tolerance makes digital circuits **robust** against noise.
- Binary signals are the basic language of computers; a “bit” is one digit of a binary number.



Flip-Flop Circuits

- The **basic memory element** of digital electronics: it stores 1 bit.
- It holds a value until the next **clock pulse**.
- It takes an input and a clock signal and produces an output (Q and its inverse Q′).
- The clock signal marks a moment in time; the input is **loaded** into the flip-flop during that clock pulse.
- Registers (AX, BX …) are made of 16 flip-flops side by side.

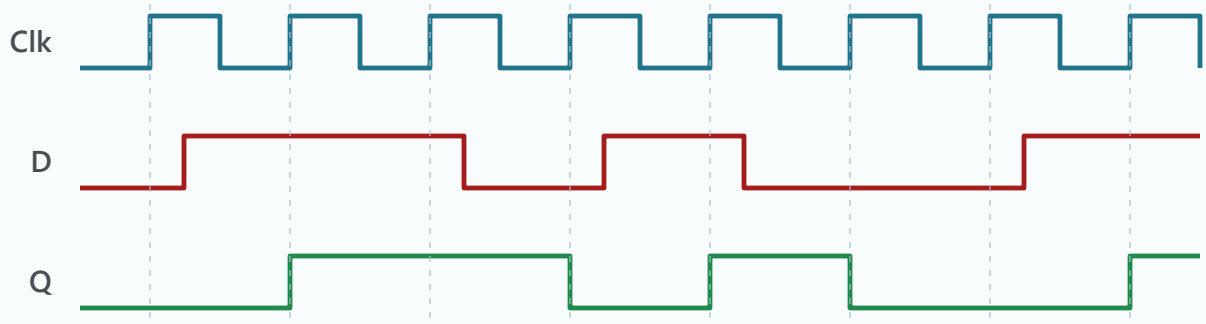


Flip-Flop Types

Type	Inputs	Behavior (on the clock pulse)	Characteristic
SR (Set-Reset)	S, R	S = 1 → Q = 1 (set); R = 1 → Q = 0 (reset); S = R = 0 → no change	S = R = 1 **forbidden**
D (Data)	D	Loads its single data input to the output on the clock: Q ← D	register/memory cell
JK	J, K	J = 1 → set, K = 1 → reset, J = K = 0 → no change, **J = K = 1 → toggle**	no forbidden state, unlike SR
T (Toggle)	T	With T = 1 the output flips on every pulse; T = 0 → no change	counters, frequency dividers

Besides set and reset, the JK **toggles** when J = K = 1; that toggle is what makes it flexible.

D Flip-Flop Timing



Q takes D's current value only on the rising clock edge (dashed lines)

- The D input may change any number of times between clock pulses; it **does not affect** the output.
- On the rising edge D is sampled and **held** in Q until the next edge: 1 bit of memory.
- All internal operations of the 8086 are divided into clock cycles this way (e.g. 5 MHz → a 200 ns cycle).

Floating Point: IEEE 754

- Fractional numbers use the **IEEE 754** standard; a single-precision number is **32 bits**.
- The 32 bits form three fields: the **sign** (1 bit), the **exponent** (8 bits) and the **mantissa** (fraction, 23 bits).
- The first bit is the sign bit: 0 positive, 1 negative.
- The exponent is stored with a **bias of 127**: true exponent = e − 127.
- The mantissa is the **fractional part** of the normalized number; the leading 1 is not stored (the hidden bit): value = $(-1)^s \times 1.f \times 2^{e-127}$.



The 8086 has no floating-point instructions; these operations are done by the 8087 math coprocessor or in software. 64-bit double: 1 + 11 + 52 bits, bias 1023.

Decoding IEEE 754



Example: `0 01111100 01000000000000000000000` = 3E200000h.

Step 1 / 5

Result (+1) × (2^{-3}) × 1.25; its decimal value is **0.15625**.

Decoding IEEE 754

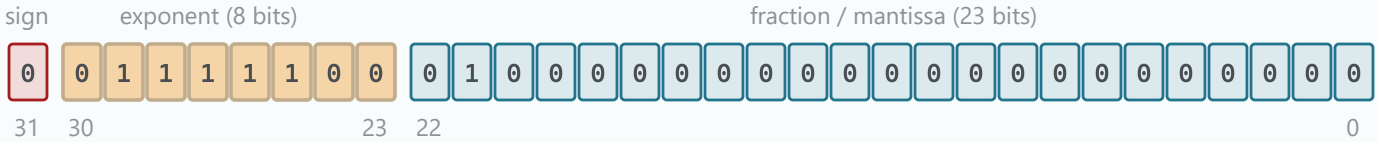


Sign bit 0 → the number is **positive**: (+1).

Step 2 / 5

Result (+1) × (2^{-3}) × 1.25; its decimal value is **0.15625**.

Decoding IEEE 754

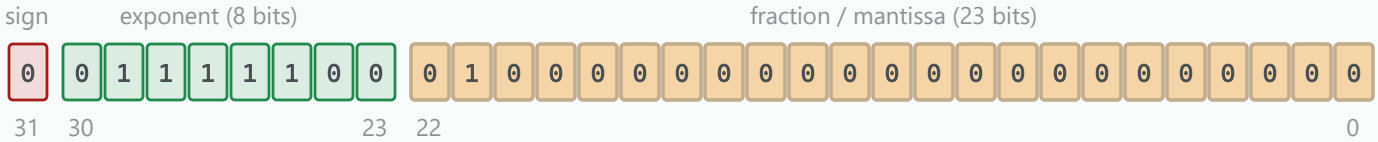


Exponent 01111100b = 124 → true exponent 124 − 127 = **−3** → 2^{-3}.

Step 3 / 5

Result (+1) × (2^{-3}) × 1.25; its decimal value is **0.15625**.

Decoding IEEE 754

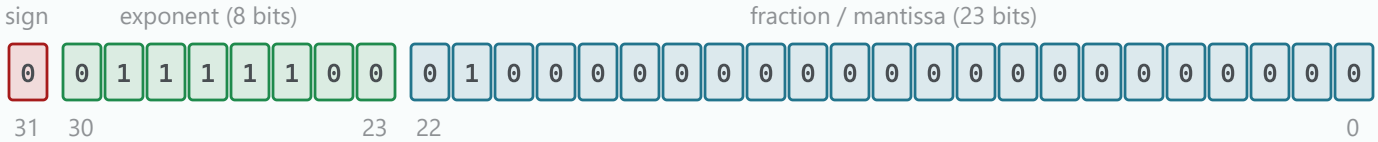


Mantissa 0100…0 = $0\times2^{-1} + 1\times2^{-2} = 0.25$ → with the hidden bit **1.25**.

Step 4 / 5

Result $(+1) \times (2^{-3}) \times 1.25$; its decimal value is **0.15625**.

Decoding IEEE 754



Result: $(+1) \times 2^{-3} \times 1.25 = 1.25 / 8 =$ **0.15625**.

Step 5 / 5

Result $(+1) \times (2^{-3}) \times 1.25$; its decimal value is **0.15625**.

Encoding IEEE 754: −6.5



Let us convert −6.5 to 32-bit IEEE 754.

Step 1 / 6

Pattern	Meaning
e = 0, f = 0	±0
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN (not a number, e.g. 0/0)
e = 0, f ≠ 0	denormalized (very small) number

Encoding IEEE 754: −6.5

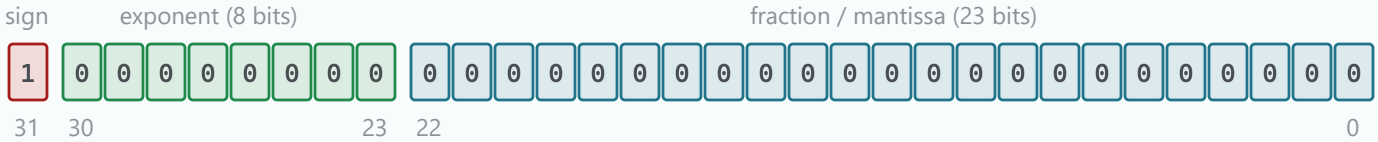


Negative → sign bit **1**.

Step 2 / 6

Pattern	Meaning
e = 0, f = 0	±0
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN (not a number, e.g. 0/0)
e = 0, f ≠ 0	denormalized (very small) number

Encoding IEEE 754: −6.5

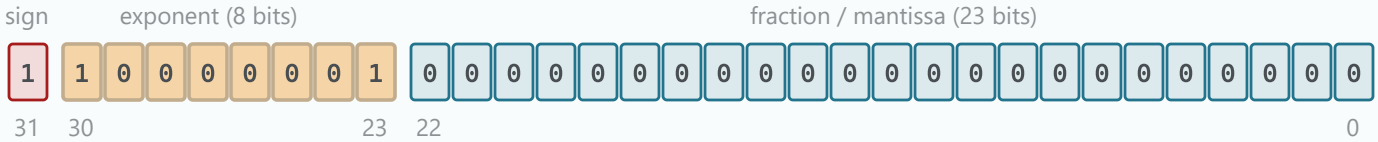


6.5 = 110.1b (4 + 2 + 0.5). Normalize: **1.101b × 2^2**.

Step 3 / 6

Pattern	Meaning
e = 0, f = 0	±0
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN (not a number, e.g. 0/0)
e = 0, f ≠ 0	denormalized (very small) number

Encoding IEEE 754: −6.5

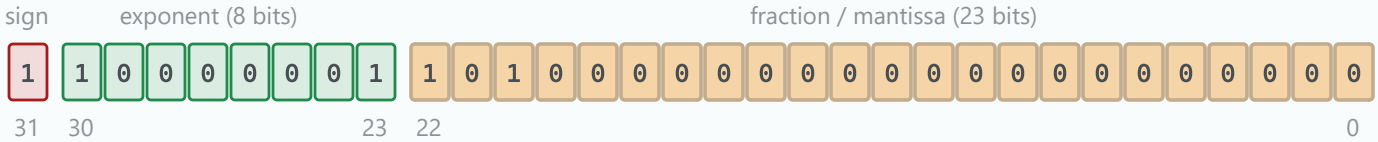


Exponent: 2 + 127 = 129 = **10000001b**.

Step 4 / 6

Pattern	Meaning
e = 0, f = 0	±0
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN (not a number, e.g. 0/0)
e = 0, f ≠ 0	denormalized (very small) number

Encoding IEEE 754: −6.5

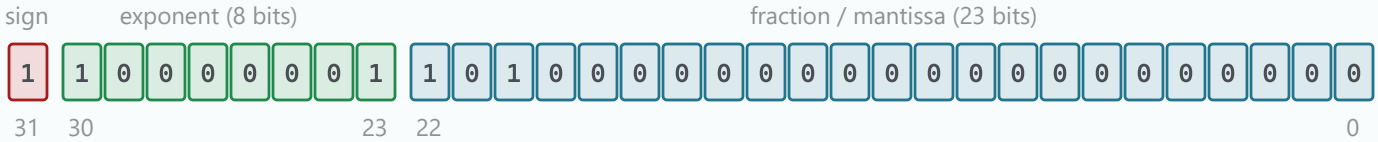


Mantissa: drop the hidden 1 and pad the remaining **101** with 0s on the right.

Step 5 / 6

Pattern	Meaning
e = 0, f = 0	±0
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN (not a number, e.g. 0/0)
e = 0, f ≠ 0	denormalized (very small) number

Encoding IEEE 754: −6.5



Result: 1100 0000 1101 0000 0…0b = **C0D00000h**.

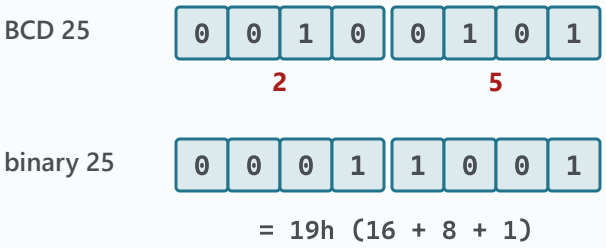
Step 6 / 6

Pattern	Meaning
e = 0, f = 0	±0
e = 255, f = 0	±∞ (e.g. 1/0)
e = 255, f ≠ 0	NaN (not a number, e.g. 0/0)
e = 0, f ≠ 0	denormalized (very small) number

BCD: Binary-Coded Decimal

- BCD (binary-coded decimal) represents **each decimal digit with four bits**.
- Each digit from 0 to 9 takes 4 bits; the patterns 1010 … 1111 are **invalid**.
- Example: 25 → `0010 0101` (BCD), while in binary it is `0001 1001` (19h).
- **Packed** BCD: two digits per byte (25h). **Unpacked**: one digit per byte (02h 05h).
- Uses: clock chips (RTC), 7-segment displays, calculators, financial arithmetic.

Decimal	BCD
0	`0000`
1	`0001`
2	`0010`
3	`0011`
4	`0100`
5	`0101`
6	`0110`
7	`0111`
8	`1000`
9	`1001`



BCD Addition and Adjustment

Let us add BCD 38 and BCD 45.

Step 1 / 6

A carry out of the low nibble (e.g. 29 + 18 = 41h, AF = 1) also adds 6 → 47. If the high digit exceeds 9, 60h is added to it as well and CF = 1.

BCD Addition and Adjustment

```
38    0011 1000        38h
```

38 → 0011 1000.

Step 2 / 6

A carry out of the low nibble (e.g. 29 + 18 = 41h, AF = 1) also adds 6 → 47. If the high digit exceeds 9, 60h is added to it as well and CF = 1.

BCD Addition and Adjustment

38	0011 1000	38h
+ 45	0100 0101	45h

45 → 0100 0101.

Step 3 / 6

A carry out of the low nibble (e.g. 29 + 18 = 41h, AF = 1) also adds 6 → 47. If the high digit exceeds 9, 60h is added to it as well and CF = 1.

BCD Addition and Adjustment

38	0011 1000	38h	
+ 45	0100 0101	45h	
= ?	0111 1101	7Dh	1101 = 13 > 9 → invalid digit

Add in binary: 0111 1101 = 7Dh. The low digit 1101 (13) is **invalid** in BCD.

Step 4 / 6

A carry out of the low nibble (e.g. 29 + 18 = 41h, AF = 1) also adds 6 → 47. If the high digit exceeds 9, 60h is added to it as well and CF = 1.

BCD Addition and Adjustment

38	0011 1000	38h	
+ 45	0100 0101	45h	
= ?	0111 1101	7Dh	1101 = 13 > 9 → invalid digit
+ 6	0000 0110	adjustment	

Add **6** to an invalid digit (or when AF = 1): this skips the 16 − 10 = 6 unused patterns.

Step 5 / 6

A carry out of the low nibble (e.g. 29 + 18 = 41h, AF = 1) also adds 6 → 47. If the high digit exceeds 9, 60h is added to it as well and CF = 1.

BCD Addition and Adjustment

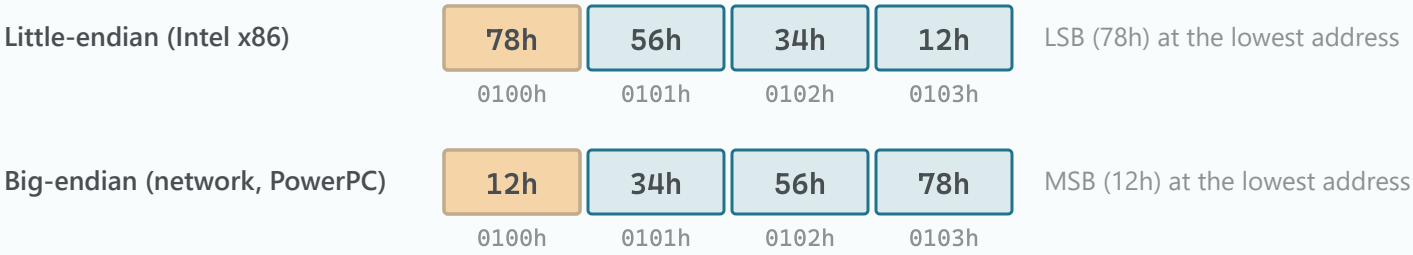
38	0011 1000	38h	
+ 45	0100 0101	45h	
= ?	0111 1101	7Dh	1101 = 13 > 9 → invalid digit
+ 6	0000 0110	adjustment	
= 83	1000 0011	83h	BCD 83 ✓

Result 1000 0011 = **83** ✓. On the 8086, `ADD AL, BL` followed by `DAA` does this adjustment.

A carry out of the low nibble (e.g. 29 + 18 = 41h, AF = 1) also adds 6 → 47. If the high digit exceeds 9, 60h is added to it as well and CF = 1.

Little-Endian and Big-Endian

The 32-bit value 12345678h, starting at address 0100h:



- **Little-endian**: the least significant byte (LSB) is at the lowest memory address. Intel x86 (the 8086 included) is little-endian.
- **Big-endian**: the most significant byte (MSB) is at the lowest address. PowerPC (mostly), the Motorola 68000 and **network communication** (“network byte order”) are big-endian.

For 01020304h the little-endian memory layout is `04 03 02 01`; the big-endian layout is `01 02 03 04`.

Gray Code

Decimal	Binary	changed bit	Gray
0	`000`	–	`000`
1	`001`	1	`00`**`1`**
2	`010`	2	`0`**`1`**`1`
3	`011`	1	`01`**`0`**
4	`100`	3	**`1`**`10`
5	`101`	1	`11`**`1`**
6	`110`	2	**`1`**`0`**`1`**
7	`111`	1	`10`**`0`**

- In Gray code two consecutive numbers differ **in only one bit**.
- In binary, going from 3 to 4 (011 → 100) changes **three bits** at once; if they do not change at the same instant, intermediate values (e.g. 111) can be read.
- In Gray code that step is 010 → 110: one bit, no false intermediate value.
- Used in rotary encoders, analog-to-digital converters and Karnaugh maps; robust against **electrical noise**.
- The 3-bit sequence: 000, 001, 011, 010, 110, 111, 101, 100.

Binary to Gray Code

Conversion formula: **G = B XOR (B >> 1)**.

Step 1 / 4

- Rule: the top Gray bit equals the binary one; each following bit is the XOR of the binary bit and the bit to its left.
- On the 8086 in three instructions: `MOV BL, AL` / `SHR BL, 1` / `XOR AL, BL`.
- Back again: $B_{top} = G_{top}$, then $B_i = B_{i+1}$ XOR G_i.

Binary to Gray Code

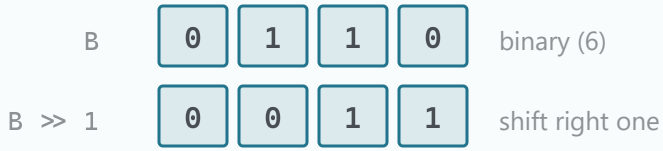


B = 0110 (6).

Step 2 / 4

- Rule: the top Gray bit equals the binary one; each following bit is the XOR of the binary bit and the bit to its left.
- On the 8086 in three instructions: `MOV BL, AL` / `SHR BL, 1` / `XOR AL, BL`.
- Back again: $B_{top} = G_{top}$, then $B_i = B_{i+1}$ XOR G_i.

Binary to Gray Code

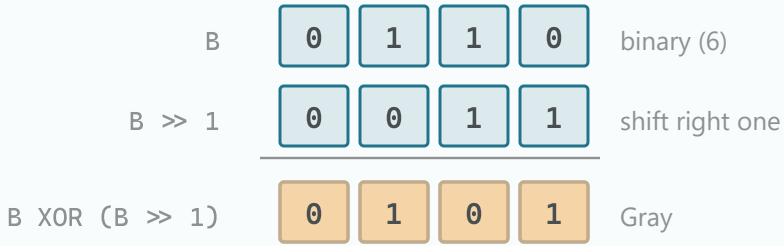


B is shifted right one bit: 0011.

Step 3 / 4

- Rule: the top Gray bit equals the binary one; each following bit is the XOR of the binary bit and the bit to its left.
- On the 8086 in three instructions: `MOV BL, AL` / `SHR BL, 1` / `XOR AL, BL`.
- Back again: $B_{top} = G_{top}$, then $B_i = B_{i+1}$ XOR G_i.

Binary to Gray Code

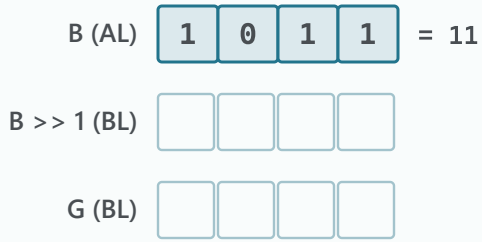


Bitwise XOR: 0110 ⊕ 0011 = **0101** → the Gray code of 6.

- Rule: the top Gray bit equals the binary one; each following bit is the XOR of the binary bit and the bit to its left.
- On the 8086 in three instructions: `MOV BL, AL` / `SHR BL, 1` / `XOR AL, BL`.
- Back again: $B_{top} = G_{top}$, then $B_i = B_{i+1}$ XOR G_i.

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
▶ MOV AL, 1011b  ; 11
3  MOV BL, AL
4  SHR BL, 1       ; B >> 1
5  XOR BL, AL      ; G
6  MOV AL, 1100b   ; 12
7  MOV BH, AL
8  SHR BH, 1
9  XOR BH, AL      ; G
10 RET
```



AL = **1011b** = 11. The Gray code formula: G = B xor (B >> 1).

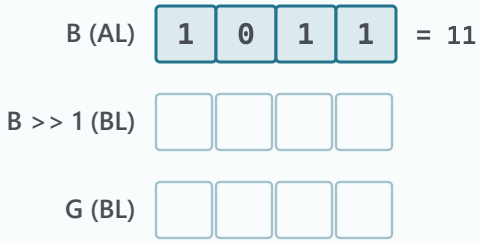
Step 1 / 8

VARIABLES

AL = **0Bh** BL = **00h** BH = **00h**

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b   ; 11
▶  MOV BL, AL
4  SHR BL, 1       ; B >> 1
5  XOR BL, AL      ; G
6  MOV AL, 1100b   ; 12
7  MOV BH, AL
8  SHR BH, 1
9  XOR BH, AL      ; G
10 RET
```



BL = AL: the shift is done on a copy, so B (AL) is kept.

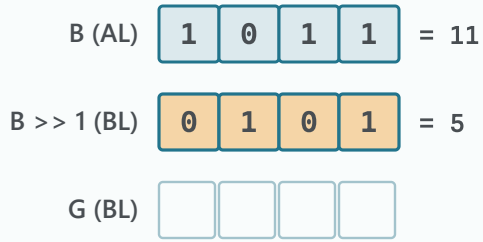
Step 2 / 8

VARIABLES

AL = 0Bh | BL = 0Bh | BH = 00h

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b   ; 11
3  MOV BL, AL
4  SHR BL, 1       ; B >> 1
5  XOR BL, AL      ; G
6  MOV AL, 1100b   ; 12
7  MOV BH, AL
8  SHR BH, 1
9  XOR BH, AL      ; G
10 RET
```



`SHR BL, 1` : **0101b**. The rightmost bit falls out (CF = 1) and a 0 enters on the left.

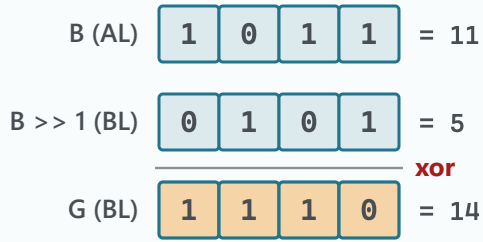
Step 3 / 8

VARIABLES

AL = **0Bh** BL = **05h** BH = **00h**

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b   ; 11
3  MOV BL, AL
4  SHR BL, 1       ; B >> 1
▶  XOR BL, AL      ; G
6  MOV AL, 1100b   ; 12
7  MOV BH, AL
8  SHR BH, 1
9  XOR BH, AL      ; G
10 RET
```



`Gray(11) = 1110`

`XOR BL, AL`: 1011 xor 0101 = **1110b**: the Gray code of 11. The top bit stayed the same as in binary.

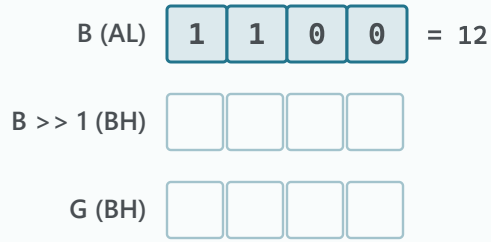
Step 4 / 8

VARIABLES

AL = 0Bh BL = 0Eh BH = 00h

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b   ; 11
3  MOV BL, AL
4  SHR BL, 1       ; B >> 1
5  XOR BL, AL      ; G
▶  MOV AL, 1100b   ; 12
7  MOV BH, AL
8  SHR BH, 1
9  XOR BH, AL      ; G
10 RET
```



`Gray(11) = 1110`

AL = **1100b** = 12, the next number.

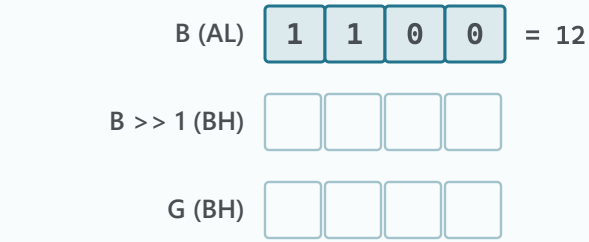
Step 5 / 8

VARIABLES

AL = 0Ch BL = 0Eh BH = 00h

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b   ; 11
3  MOV BL, AL
4  SHR BL, 1       ; B >> 1
5  XOR BL, AL      ; G
6  MOV AL, 1100b   ; 12
▶  MOV BH, AL
8  SHR BH, 1
9  XOR BH, AL      ; G
10 RET
```



`Gray(11) = 1110`

BH = AL.

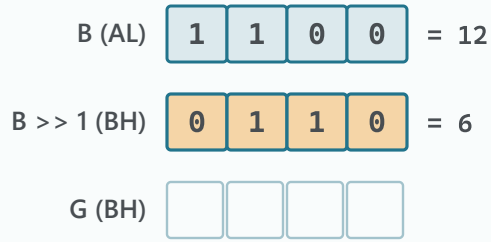
Step 6 / 8

VARIABLES

AL = 0Ch BL = 0Eh BH = 0Ch

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b  ; 11
3  MOV BL, AL
4  SHR BL, 1      ; B >> 1
5  XOR BL, AL     ; G
6  MOV AL, 1100b  ; 12
7  MOV BH, AL
▶  SHR BH, 1
9  XOR BH, AL     ; G
10 RET
```



`Gray(11) = 1110`

`SHR BH, 1` : **0110b**.

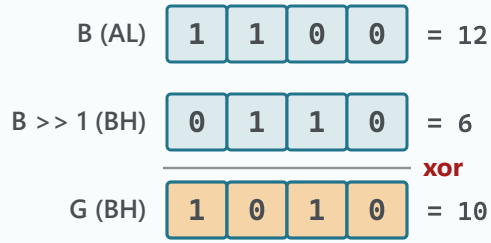
Step 7 / 8

VARIABLES

`AL = 0Ch` `BL = 0Eh` `BH = 06h`

Code Trace: Gray Code with SHR and XOR

```
1  ORG 100h
2  MOV AL, 1011b  ; 11
3  MOV BL, AL
4  SHR BL, 1      ; B >> 1
5  XOR BL, AL     ; G
6  MOV AL, 1100b  ; 12
7  MOV BH, AL
8  SHR BH, 1
▶  XOR BH, AL     ; G
10 RET
```



`Gray(11) = 1110` `Gray(12) = 1010` **only bit 2 differs**

XOR → **1010b**. From 11 to 12, 3 bits change in binary (1011 → 1100); in Gray code only **1 bit** (1110 → 1010).

Step 8 / 8

VARIABLES

AL = 0Ch BL = 0Eh BH = 0Ah

Summary

Positional systems

Value = Σ digit × base^{position}. Binary (b), hex (h), decimal; 1 hex digit = 4 bits.

Conversion

From decimal: divide by the base, read the remainders bottom up. Binary ↔ hex: groups of 4.

Signed numbers

Two's complement: flip + 1 (NEG). The MSB is the sign bit; 8 bits −128…127, 16 bits −32768…32767.

Encodings

IEEE 754: sign/exponent(127)/mantissa. BCD: 4 bits per digit, +6 adjustment. x86 is little-endian. Gray: one bit changes.

SUMMARY

Exercises

1. Convert MCMXCIV and MMXXVI to decimal.
2. Convert 11001010b to decimal and hex.
3. Write 2024 in hex and binary.
4. Write −100 as an 8-bit two's complement number. What does it look like in 16 bits?
5. What are the 8-bit result, CF and signed reading of 0F0h + 20h?
6. Do the BCD addition 47 + 36 with adjustment.
7. What is the value of the IEEE 754 number 0 10000010 01100000000000000000000?
8. What is the 3-bit Gray code of 5? If 0ABCDh is written as a word at address 0200h, what is [0200h]?

▶ **Answers**

SUMMARY

Exercises (2)

9. Convert 0.375 and 0.2 to binary (8 fraction bits for 0.2).
10. Write −3.625 in 8-bit Q5.3 format (binary and hex).
11. Compute 7C5Eh + 3BA9h; compute 3BA9h − 7C5Eh with the 16's complement.
12. In 8 bits, what are the results and OF for 70h + 30h, 90h − 20h and 90h − F0h?

▶ **Answers**

References

- M. Morris Mano, Michael D. Ciletti, *Digital Design*, 5th ed., Pearson, 2013.
- Barry B. Brey, *The Intel Microprocessors*, 8th ed., Pearson, 2009 — Chapter 1: number systems.
- IEEE Std 754-2019, *IEEE Standard for Floating-Point Arithmetic*.
- emu8086 documentation: *Numbering systems tutorial*.
- Mehmet Bodur, *Mikroişlemcilere Giriş: Assembler ile Yazılım ve Arayüz*, EMO, 2014 (in Turkish).
- Şadi Çağatay Öztürk, Oğuz Ergin, *Intel 8086 ile Mikroişlemci Programlamaya Giriş*, 2009 (in Turkish).

The End

Chapter 1.2: Number Systems

Previous: Chapter 1.1: Introduction · Next: Chapter 2.1: 8086 Architecture