

UNIT 2

Number Systems

(5 Hours) — Digital Logic (CACS105)

A number system is a set of rules and symbols used to represent numeric values. Digital electronic circuits — including computers — process information using only two stable states (ON/OFF, HIGH/LOW), which is why the binary number system forms the foundation of digital logic. However, binary numbers are long and hard for humans to read, so octal and hexadecimal systems are commonly used as shorthand notations. This unit covers the major number systems, their inter-conversions, signed number and floating-point representation, binary arithmetic, and the important binary codes (BCD, ASCII, Excess-3, Gray Code) along with error detecting and correcting codes used in digital communication and storage.

2.1 Binary, Octal, and Hexadecimal Number Systems

A number system is characterized by its base (or radix) — the number of distinct digit symbols it uses, including zero. Each digit position in a number carries a positional weight equal to the base raised to the power of its position, counted from right (position 0) to left.

Number System	Base (Radix)	Digits Used	Example
Decimal	10	0 – 9	(245) ₁₀
Binary	2	0, 1	(1101) ₂
Octal	8	0 – 7	(627) ₈
Hexadecimal	16	0–9, A–F	(2F3) ₁₆

Relationship Among Number Systems

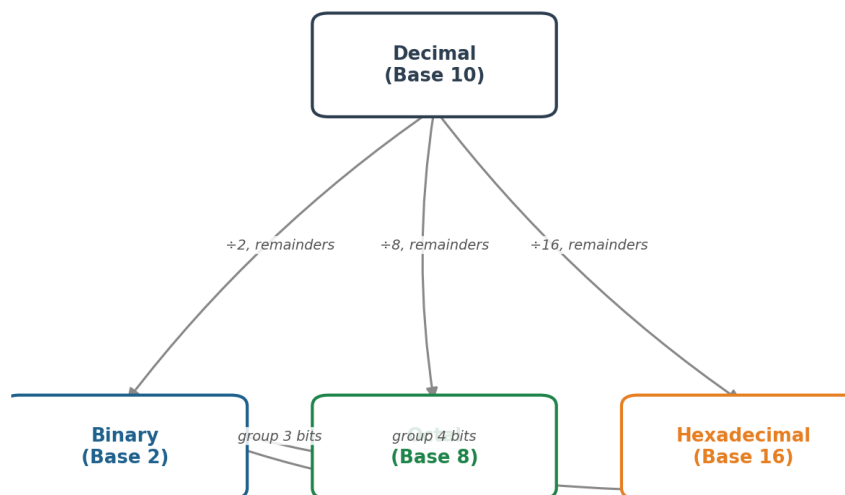


Figure 2.1 — Relationship among the four commonly used number systems.

General Formula: Value of a Positional Number

For a number with digits $d_{n-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots$ in base r , the decimal equivalent is:

$$N_{10} = \sum (d_i \times r^i), \text{ for } i \text{ ranging over all digit positions (integer and fractional)}$$

Note: Positions to the left of the radix point carry positive powers of the base; positions to the right carry negative powers.

(A) Binary ↔ Decimal Conversion

Binary → Decimal: Multiply each bit by its positional weight (power of 2) and sum the results.

Example 2.1 — Convert $(11011)_2$ to decimal

$$(11011)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Solution

$$= 16 + 8 + 0 + 2 + 1 = (27)_{10}$$

Decimal → Binary: Repeatedly divide the decimal number by 2, recording the remainders. The binary equivalent is the sequence of remainders read from bottom (last) to top (first).

Example 2.2 — Convert $(45)_{10}$ to binary

$$45 \div 2 = 22 \text{ remainder } 1$$

$$22 \div 2 = 11 \text{ remainder } 0$$

$$11 \div 2 = 5 \text{ remainder } 1$$

$$5 \div 2 = 2 \text{ remainder } 1$$

$$2 \div 2 = 1 \text{ remainder } 0$$

$$1 \div 2 = 0 \text{ remainder } 1$$

Solution

Reading remainders bottom→top: $(45)_{10} = (101101)_2$

For fractional decimal numbers, repeatedly multiply the fractional part by 2; the integer part generated at each step (0 or 1) forms the binary fraction, read top to bottom.

Example 2.3 — Convert $(0.625)_{10}$ to binary

$$0.625 \times 2 = 1.25 \rightarrow \text{bit} = 1$$

$$0.25 \times 2 = 0.50 \rightarrow \text{bit} = 0$$

$$0.50 \times 2 = 1.00 \rightarrow \text{bit} = 1$$

Solution

Reading top→bottom: $(0.625)_{10} = (0.101)_2$

(B) Octal ↔ Decimal and Octal ↔ Binary

Octal → Decimal: same weighted-sum method using powers of 8.

Example 2.4 — Convert $(247)_8$ to decimal

$$(247)_8 = 2 \times 8^2 + 4 \times 8^1 + 7 \times 8^0$$

Solution

$$= 128 + 32 + 7 = (167)_{10}$$

Octal ↔ Binary: Because $8 = 2^3$, each octal digit corresponds exactly to a group of 3 binary bits — no arithmetic conversion is required.

Example 2.5 — Convert $(101110)_2$ to octal

Group into 3-bit sets from the right: 101 | 110

$$101 = 5, 110 = 6$$

Solution

$$(101110)_2 = (56)_8$$

(C) Hexadecimal ↔ Decimal and Hexadecimal ↔ Binary

Hexadecimal uses 16 symbols: 0–9 followed by A, B, C, D, E, F representing decimal values 10 through 15. Since $16 = 2^4$, each hex digit maps directly to a 4-bit binary group.

Hex Digit	Decimal Value	4-bit Binary
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Example 2.6 — Convert $(1011\ 1100)_2$ to hexadecimal

Group into 4-bit sets from the right: 1011 | 1100

1011 = B, 1100 = C

Solution

$$(10111100)_2 = (BC)_{16}$$

Example 2.7 — Convert $(2A)_{16}$ to decimal

$$(2A)_{16} = 2 \times 16^1 + 10 \times 16^0$$

Solution

$$= 32 + 10 = (42)_{10}$$

Key Takeaways — Section 2.1

- ✓ Every positional number system is defined by its base/radix and digit set.
- ✓ Binary↔Decimal uses division/multiplication by 2 with remainders.
- ✓ Octal and Hexadecimal convert to/from Binary by simple bit grouping (3 bits and 4 bits respectively) — no arithmetic needed.
- ✓ Hexadecimal is widely used in computing to represent memory addresses and machine code compactly.

bcalink.com

2.1.1 Representation of Signed Numbers & Floating-Point Numbers

Digital systems must represent both positive and negative numbers using only bits (0s and 1s). Three common signed-number representations are used:

(A) Sign-Magnitude Representation

The most significant bit (MSB) is reserved as the sign bit (0 = positive, 1 = negative); the remaining bits represent the magnitude.

Example 2.8 — Represent +25 and –25 in 8-bit sign-magnitude form

25 in binary (7 bits) = 0011001

Solution

+25 = 0 0011001 (sign bit 0)

–25 = 1 0011001 (sign bit 1)

Note: Sign-magnitude has two representations of zero (+0 and –0), which complicates arithmetic circuit design.

(B) 1's Complement Representation

The negative of a number is obtained by inverting every bit (0→1, 1→0) of its positive binary form.

Example 2.9 — Represent –18 in 8-bit 1's complement

+18 = 00010010

Invert all bits:

Solution

–18 = 11101101 (1's complement)

(C) 2's Complement Representation (most widely used)

Obtained by taking the 1's complement and adding 1. This system has a single representation for zero and simplifies subtraction to addition, which is why virtually all modern processors use it.

Example 2.10 — Represent -18 in 8-bit 2's complement

+18 = 00010010

1's complement = 11101101

Add 1:

Solution

11101101 + 1 = 11101110 = -18 in 2's complement

Representation	Range (n-bit)	Zero Representations	Arithmetic Ease
Sign-Magnitude	$-(2^{n-1}-1)$ to $+(2^{n-1}-1)$	Two (+0, -0)	Complex
1's Complement	$-(2^{n-1}-1)$ to $+(2^{n-1}-1)$	Two (+0, -0)	Moderate
2's Complement	-2^{n-1} to $+(2^{n-1}-1)$	One	Simple

(D) Floating-Point Number Representation

Floating-point representation stores very large or very small real numbers using scientific notation adapted to binary: $N = \pm M \times 2^E$, where M is the mantissa (significand) and E is the exponent. The widely adopted IEEE-754 standard defines the format bit-layout so that different machines interpret floating-point values consistently.



Figure 2.2 — IEEE-754 single precision (32-bit) floating-point layout.

In the 32-bit single-precision format: 1 bit is used for the sign, 8 bits for a biased exponent (bias = 127), and 23 bits for the mantissa (with an implied leading 1). Double precision uses 64 bits (1 sign, 11 exponent, 52 mantissa bits) for greater range and precision.

Example 2.11 — Represent $(10.5)_{10}$ in IEEE-754 single precision

Step 1: Convert to binary $\rightarrow 1010.1 = 1.0101 \times 2^3$

Step 2: Sign = 0 (positive)

Step 3: Exponent = 3 + 127 (bias) = 130 = 10000010

Step 4: Mantissa (23 bits, drop implied leading 1) = 01010000000000000000000

Solution

IEEE-754 $(10.5)_{10} = 0 \mid 10000010 \mid 010100000000000000000000$

Key Takeaways — Section 2.1.1

- ✓ Sign-magnitude, 1's complement, and 2's complement are the three main signed-number systems.
- ✓ 2's complement is preferred because it has a single zero and reduces subtraction to addition.
- ✓ Floating-point numbers use a sign, exponent, and mantissa to represent a wide range of real values.
- ✓ IEEE-754 is the international standard ensuring portability of floating-point data across systems.

bcalink.com

2.1.2 Binary Arithmetic

Binary arithmetic follows the same principles as decimal arithmetic but uses only the digits 0 and 1. The four basic operations — addition, subtraction, multiplication, and division — all reduce to simple bit-level rules.

(A) Binary Addition

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Example 2.12 — Add $(1011)_2$ and $(1101)_2$

```
1011
+ 1101
-----
```

Solution

```
1011
+ 1101
-----
11000 →  $(1011)_2 + (1101)_2 = (11000)_2$  [verify:  $11+13=24=(11000)_2$  ✓]
```

(B) Binary Subtraction

Direct subtraction uses borrow rules (0–1 borrows 1, making it 10–1=1), but digital circuits normally perform subtraction using 2's complement addition: $A - B = A + (2\text{'s complement of } B)$.

Example 2.13 — Subtract $(0101)_2$ from $(1100)_2$ using 2's complement

$1100 - 0101 \rightarrow$ Take 2's complement of 0101:
1's complement of 0101 = 1010; add 1 $\rightarrow 1011$
Now add: $1100 + 1011$

Solution

$1100 + 1011 = 10111 \rightarrow$ discard the extra carry bit (5th bit)

Result = $0111 = (12 - 5 = 7 \checkmark)$

(C) Binary Multiplication

Binary multiplication follows the same shift-and-add process as decimal long multiplication, but each partial product is either the multiplicand itself (if the multiplier bit is 1) or zero (if it is 0).

Example 2.14 — Multiply $(1011)_2 \times (101)_2$

1011 (11)

$\times 101$ (5)

$1011 \leftarrow 1011 \times 1$

$0000 \leftarrow 1011 \times 0$, shifted 1

$1011 \leftarrow 1011 \times 1$, shifted 2

Solution

Sum of partial products = $110111 = (55)_{10}$ [$11 \times 5 = 55 \checkmark$]

(D) Binary Division

Binary division follows the long-division method, repeatedly comparing, subtracting, and bringing down bits — analogous to decimal division but using only 0s and 1s as quotient digits.

Example 2.15 — Divide $(1010)_2$ by $(10)_2$

$1010 \div 10 \rightarrow$ quotient built bit by bit using binary long division

Solution

$(1010)_2 \div (10)_2 = (101)_2$ [$10 \div 2 = 5 \checkmark$]

Key Takeaways — Section 2.1.2

- ✓ Binary addition/subtraction follow simple bit rules; carry and borrow propagate leftward.
- ✓ Subtraction in hardware is usually implemented via 2's complement addition.
- ✓ Multiplication uses the shift-and-add technique; division uses shift-and-subtract (long division).
- ✓ All arithmetic circuits in a CPU (adders, ALUs) are built from these basic binary rules.

bcalink.com

2.2 Codes: BCD, ASCII, Excess-3, Gray Code, and Error-Detecting/Correcting Codes

A code is a symbolic representation that assigns a unique bit pattern to each element of a set of information (digits, characters, or instructions). Digital systems use standardized codes for interoperability, data storage, and reliable transmission.

(A) Binary Coded Decimal (BCD)

BCD represents each decimal digit (0–9) individually using its 4-bit binary equivalent, rather than converting the entire decimal number to pure binary. This makes conversion to/from decimal simple, at the cost of using more bits than pure binary for the same value.

Example 2.16 — Represent $(529)_{10}$ in BCD

$5 \rightarrow 0101$, $2 \rightarrow 0010$, $9 \rightarrow 1001$

Solution

$(529)_{10} = 0101\ 0010\ 1001$ (BCD)

Note: BCD wastes code combinations: the 4-bit patterns 1010–1111 (10–15) are invalid and never used in BCD.

(B) ASCII Code

ASCII (American Standard Code for Information Interchange) is a 7-bit character encoding scheme (128 possible characters) used to represent letters, digits, punctuation, and control characters in computers and communication equipment. An 8th bit is often added as a parity bit for simple error checking.

Character	ASCII (Decimal)	ASCII (Binary, 7-bit)
A	65	1000001
a	97	1100001
0	48	0110000
Space	32	0100000

(C) Excess-3 Code

Excess-3 (XS-3) is a self-complementing BCD variant obtained by adding 3 (0011) to the 4-bit binary value of each decimal digit before encoding. Being self-complementing means the 9's complement of a digit can be found simply by inverting all bits — this property simplifies

decimal subtraction circuits.

Example 2.17 — Represent decimal 6 in Excess-3

$6 + 3 = 9 \rightarrow$ binary of 9 = 1001

Solution

Excess-3 code of 6 = 1001

(D) Gray Code

Gray code is a binary numeral system where two successive values differ in only one bit position (unit distance code). This property eliminates ambiguous transitional states in rotary encoders, position sensors, and Karnaugh-map adjacency analysis, making it valuable wherever multiple bits could otherwise change simultaneously and cause glitches.

Binary $\rightarrow$ Gray conversion: The MSB of Gray code equals the MSB of binary; each subsequent Gray bit is the XOR of the current and previous binary bits.

Example 2.18 — Convert binary (1011) to Gray code

$$g_3 = b_3 = 1$$

$$g_2 = b_3 \oplus b_2 = 1 \oplus 0 = 1$$

$$g_1 = b_2 \oplus b_1 = 0 \oplus 1 = 1$$

$$g_0 = b_1 \oplus b_0 = 1 \oplus 1 = 0$$

Solution

Gray code of $(1011)_2 = 1110$

Code Comparison: Decimal 0-9 in Different Codes

Decimal	Binary	Gray Code	BCD	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0010	0011	0010	0101
3	0011	0010	0011	0110
4	0100	0110	0100	0111
5	0101	0111	0101	1000
6	0110	0101	0110	1001
7	0111	0100	0111	1010
8	1000	1100	1000	1011
9	1001	1101	1001	1100

Figure 2.3 — Binary, Gray, BCD, and Excess-3 codes for decimal 0–9.

(E) Error Detecting and Correcting Codes

During transmission or storage, noise can flip bits, corrupting data. Error detecting/correcting codes add redundant bits so that receivers can detect — and, in some schemes, even correct — such errors without retransmission.

1. Parity Bit (Error Detection Only)

A single extra bit is appended to a data word so that the total number of 1-bits is either always even (even parity) or always odd (odd parity). The receiver recomputes the parity; a mismatch signals that a bit was altered in transit. Simple parity can detect a single-bit error but cannot correct it, and it fails to detect an even number of simultaneous bit errors.

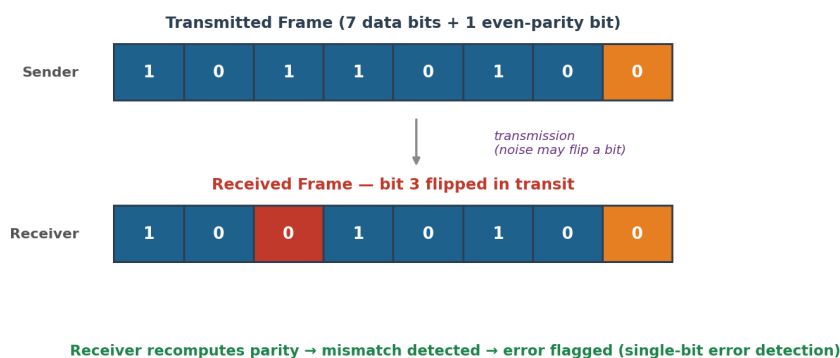


Figure 2.4 — Even-parity error detection: a flipped data bit is caught by the mismatch.

Example 2.19 — Generate even parity for data 1001101

Count of 1's in 1001101 = 4 (already even)

Solution

Even parity bit = 0 → transmitted frame = 10011010

2. Hamming Code (Error Detection & Correction)

Hamming code inserts multiple parity bits at specific positions (powers of 2: 1, 2, 4, 8, ...) within the data, each checking a different overlapping subset of bits. On reception, the parity checks combine to point directly at the position of a single-bit error (if any), allowing the receiver to locate and correct it automatically — a capability plain parity does not offer.

Code	Detects	Corrects	Typical Use
Simple Parity	Odd number of bit errors	No	Serial links, memory checks
Checksum	Burst / multiple errors	No	Network packets, file transfer
Hamming Code	Single & some double-bit errors	Single-bit errors	RAM (ECC memory), satellite links
CRC	Burst errors reliably	No (detection only)	Ethernet, storage devices

Note: The choice between BCD, Excess-3, Gray code, or error-correcting schemes depends on the application — arithmetic ease, self-complementing property, glitch-free transitions, or reliability of data transmission.

Key Takeaways — Section 2.2

- ✓ BCD encodes each decimal digit separately in 4 bits, simplifying decimal I/O at the cost of efficiency.
- ✓ ASCII is the standard 7-bit code for representing text characters in computing systems.
- ✓ Excess-3 is self-complementing, simplifying decimal subtraction circuits.
- ✓ Gray code changes only one bit between successive values, avoiding transition glitches.
- ✓ Parity bits detect single-bit errors; Hamming code can both detect and correct them.

— End of Unit 2 —