EIE 2050 Digital Logic and Systems

Chapter 2 : Number Systems

Instructor: Yue ZHENG, Ph.D.

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

Announcements

- NO tutorials and homework for the second week EITHER;

Last Week

- Analog versus Digital
- Bits (Binary digits), Logic Levels and Digital Waveforms
- Basic logic functions: NOT, AND and OR
- Combinational & sequential logic functions:

  comparator, adder, encoder/decoder, (de)multiplexer,

  flip-flops, registers, counter
- Integrated circuit (IC): Programmable versus Fixed-function
  - Package: Surface-mounted and Through-hole
  - Programmable: PLD (SPLD and CPLD) and FPGA
  - Fixed-function : SSI/MSI/VLSI/ULSI

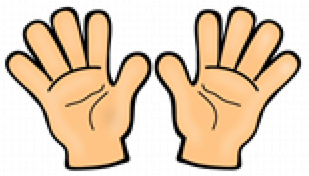
Binary Numbers

Number Systems

- A system of writing to express numbers

- Daily life: 10, 12, 60, …
- Computer systems: 2, 8,16, …

Decimal Numbers



1000's column
100's column
10's column
1's column

$$5374_{10} = 5 \times 10^3 + 3 \times 10^2 + 7 \times 10^1 + 4 \times 10^0$$

five thousands	three hundreds	seven tens	four ones

Base-10 Numbers

Base-R Number System

- Base: R ($R \geq 2$)
- Digit: [0, R-1]
- Weight: R^x

$$(N)_R = a_{n-1}\, a_{n-2} \ldots a_1\, a_0\, a_{-1} \ldots a_{-m}$$

$$= a_{n-1} \times R^{n-1} + a_{n-2} \times R^{n-2} + \ldots + a_1 \times R^1 + a_0 \times R^0 + a_{-1} \times R^{-1} + \ldots + a_{-m} \times R^{-m}$$

$$= \sum_{i=-m}^{n-1} a_i \times Ri$$

- Decimal numbers: base-10
- Binary numbers: base-2
- Octal numbers: base-8
- Hexadecimal numbers: base-16

Decimal Number System

Base: 10

Digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Weight: $\ldots 10^2\ 10^1\ 10^0 . 10^{-1}\ 10^{-2}\ 10^{-3} \ldots$

Fractional numbers

Decimal point

Decimal numbers in digital systems mean any base 10 numbers, not just those with a decimal point.

$$
\begin{aligned}
568.23 &= (5 \times 10^2) + (6 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) + (3 \times 10^{-2}) \\
&= (5 \times 100) + (6 \times 10) + (8 \times 1) + (2 \times 0.1) + (3 \times 0.01) \\
&= 500 + 60 + 8 + 0.2 + 0.03
\end{aligned}
$$

Binary Number System

Base: 2

Digit: 0, 1

Weight: $\ldots 2^2\ 2^1\ 2^0 . 2^{-1}\ 2^{-2}\ 2^{-3} \ldots$

binary point

Binary-to-Decimal Conversion:

$$
\begin{aligned}
(10.101)_2 &= (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) \\
&= (1 \times 2) + (0 \times 1) + (1 \times 0.5) + (0 \times 0.25) + (1 \times 0.125) \\
&= 2 + 0 + 0.5 + 0 + 0.125 \\
&= (2.625)_{10}
\end{aligned}
$$

Powers of Two

- $2^0 =$
- $2^1 =$
- $2^2 =$
- $2^3 =$
- $2^4 =$
- $2^5 =$
- $2^6 =$
- $2^7 =$

Handy to memorize

Counting in Binary

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

Number Conversion

- Binary to decimal conversion:
  - Convert 10011_2 to decimal
  -

- Decimal to binary conversion:
  - Convert 53_{10} to binary

Decimal to Binary Conversion

- Two methods:
  - **Method 1:** Find the largest power of 2 that fits, subtract and repeat
  - **Method 2:** Repeatedly divide by 2, remainder goes in next most significant bit

Decimal to Binary Conversion

- 53_{10}
- **Method 1:** Find the largest power of 2 that fits, subtract and repeat
  - 53_{10} — 32×1
  - $53-32 = 21$ — 16×1
  - $21-16 = 5$ — 4×1
  - $5-4 = 1$ — 1×1 — $\mathbf{= 110101_2}$

- **Method 2:** Repeatedly divide by 2, remainder goes in next most significant bit
  - 53_{10} = $53/2 = 26$ R1
  - $26/2 = 13$ R0
  - $13/2 = 6$ R1
  - $6/2 = 3$ R0
  - $3/2 = 1$ R1
  - $1/2 = 0$ R1 — $\mathbf{= 110101_2}$

Binary Values and Range

- ***N*-digit decimal number**
  - How many values? **10^N**
  - Range? **$[0, 10^N - 1]$**
  - Example: 3-digit decimal number:
    - **10^3 = 1000 possible values**
    - **Range: [0, 999]**

- ***N*-bit binary number**
  - How many values? **2^N**
  - Range: **$[0, 2^N - 1]$**
  - Example: 3-digit binary number:
    - **2^3 = 8 possible values**
    - **Range: [0, 7] = [000_2 to 111_2]**

Octal & Hexadecimal Numbers

Octal Number System

Base: 8

Digit: 0, 1, 2, 3, 4, 5, 6, 7

Weight: $\ldots 8^2\ 8^1\ 8^0 . 8^{-1}\ 8^{-2}\ 8^{-3} \ldots$

Octal-to-Decimal Conversion:

$$
\begin{aligned}
(2374)_8 &= (2 \times 8^3) + (3 \times 8^2) + (7 \times 8^1) + (4 \times 8^0) \\
&= (2 \times 512) + (3 \times 64) + (7 \times 8) + (4 \times 1) \\
&= 1024 + 192 + 56 + 4 \\
&= (1276)_{10}
\end{aligned}
$$

Hexadecimal Number System

Base: 16

Digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Weight: $\ldots 16^2\ 16^1\ 16^0 . 16^{-1}\ 16^{-2}\ 16^{-3} \ldots$

Hex-to-Decimal Conversion:

$$
\begin{aligned}
(E5)_{16} &= (E \times 16^1) + (5 \times 16^0) \\
&= (14 \times 16) + (5 \times 1) \\
&= 224 + 5 \\
&= (229)_{10}
\end{aligned}
$$

Decimal-to-Octal Conversion

Repeated division by 8 with the 1st remainder being the least significant digit (LSD)

8	359	reminder	
8	44	7	LSD
8	5	4	
	0	5	MSD

$(359)_{10} = (547)_8$

Decimal-to-Hexadecimal Conversion

Repeated division by 16 with the 1st remainder being the least significant digit (LSD)

		reminder
16	650	
16	40	10 (A)
16	2	8
	0	2

LSD

MSD

$(650)_{10} = (28A)_{16}$

Conversion btw. Binary, Octal, & Hex. Numbers

- Octal/Hex → Binary
  - Replace each octal (hexadecimal) digit with 3 (4) bits

Octal-to-Binary

$$\overbrace{111}^{7\downarrow}\overbrace{101}^{5\downarrow}\overbrace{010}^{2\downarrow}\overbrace{110}^{6\downarrow}$$

$7526_8 = 111101010110_2$

Hexadecimal-to-Binary

$$\overbrace{1100}^{C\downarrow}\overbrace{1111}^{F\downarrow}\overbrace{1000}^{8\downarrow}\overbrace{1110}^{E\downarrow}$$

$CF8E_{16} = 1100111110001110_2$

Conversion btw. Binary, Octal, & Hex. Numbers

- Binary → Octal/Hex
  - Convert each 3-bit (4-bit) group to the equivalent octal (hexadecimal) digit
  - Start from right-most, move from right to left, leading 0s

Binary-to-Octal

$$\underbrace{100}_{4}\,\underbrace{110}_{6}\,\underbrace{011}_{3}\,\underbrace{010}_{2} = 4632_8$$

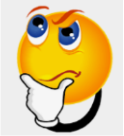
$100110011010_2 = 4632_8$

Binary-to-Hexadecimal

$$\underbrace{1100}_{C}\,\underbrace{1010}_{A}\,\underbrace{0101}_{5}\,\underbrace{0111}_{7} = CA57_{16}$$

$1100101001010111_2 = CA57_{16}$

Why do computers use Binary?



Boolean Logic & Logic Gates: Crash Course Computer Science #3

https://thecrashcourse.com/topic/computerscience/

Why Octal and Hexdecimal?

- A more convenient/compact way to represent large binary numbers
- Easy Conversion from/to Binary

- Examples:
  - Octal: file permission in Linux
  - Hex: memory address

Bytes, Nibbles, & All that Jazz

Bits, Bytes, Nibbles …

- Byte: 8 bits
  - Represents one of _____ values
  - [__, ___]
- Nibble: 4 bits
  - Represents one of _____ values
  - [__, ___]

One binary digit is __ bit
One hex digit is ___ bits or ___ nibble
Two hex digits make ___ byte

Most significant on left
Least significant on right

10010110

most significant bit

least significant bit

byte

10010110

nibble

CEBF9AD7

most significant byte

least significant byte

Large Powers of Two

- 2^{10} = 1 kilo ≈ 10^3 (1024)
- 2^{20} = 1 mega ≈ 10^6 (1,048,576)
- 2^{30} = 1 giga ≈ 10^9 (1,073,741,824)
- 2^{40} = 1 tera ≈ 10^{12}
- 2^{50} = 1 peta ≈ 10^{15}
- 2^{60} = 1 exa ≈ 10^{18}

Estimating Powers of Two

- What is the value of 2^{24}?

- How large of a value can a 32-bit integer variable represent?

Binary Arithmetic: + − × ÷

Addition

- Decimal

$$
\begin{array}{r}
{\color{blue}11} \quad \leftarrow \text{carries} \\
3734 \\
+\ \ 5168 \\
\hline
8902
\end{array}
$$

- Binary

$$
\begin{array}{r}
{\color{blue}11} \quad \leftarrow \text{carries} \\
1011 \\
+\ \ 0011 \\
\hline
1110
\end{array}
$$

Addition rules

$0 + 0 = 0$	Sum of 0 with a carry of 0
$0 + 1 = 1$	Sum of 1 with a carry of 0
$1 + 0 = 1$	Sum of 1 with a carry of 0
$1 + 1 = 10$	Sum of 0 with a carry of 1

Binary Addition Examples

- Add the following 4-bit binary numbers

$$\begin{array}{r} 1001 \\ +\ \ 0101 \\ \hline \end{array}$$

- Add the following 4-bit binary numbers

$$\begin{array}{r} 1011 \\ +\ \ 0110 \\ \hline \end{array}$$

Overflow

- Digital systems operate on a **fixed number of bits**
- Overflow: when result is too big to fit in the available number of bits
- See previous example of 11 + 6

Subtraction

$$\begin{array}{r} 101 \\ -011 \\ \hline \mathbf{010} \end{array}$$

Subtraction rules

$0 - 0 = 0$	
$1 - 1 = 0$	
$1 - 0 = 1$	
$10 - 1 = 1$	$0 - 1$ with a borrow of 1

Multiplication

$$
\begin{array}{r}
111 \\
\times\ 101 \\
\hline
111 \\
000 \\
+111 \\
\hline
100011
\end{array}
$$

Partial products

Multiplication Rules

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Division

- Decimal

$$\begin{array}{r} 25 \\ 5\,\overline{)125} \\ \underline{10} \\ 25 \\ \underline{25} \\ 0 \end{array}$$

- Binary

$$\begin{array}{r} 11 \\ 10\,\overline{)110} \\ \underline{10} \\ 10 \\ \underline{10} \\ 00 \end{array}
\qquad\qquad
\begin{array}{r} 11 \\ 11\,\overline{)1001} \\ \underline{11} \\ 11 \\ \underline{11} \\ 00 \end{array}$$

Signed Numbers

Signed Binary Number

- Sign/Magnitude Numbers
- Two’s Complement Numbers

Sign/Magnitude Numbers

- 1 sign bit, *N*-1 magnitude bits
- Sign bit is the most significant (left-most) bit
  - Positive number: sign bit = 0
  - Negative number: sign bit = 1

$$A: \{a_{N-1}, a_{N-2}, \ldots a_2, a_1, a_0\}$$

$$A = (-1)^{a_{N-1}} \sum_{i=0}^{N-2} a_i \, 2^i$$

- Example, 4-bit sign/mag representations of ± 6:

  +6 =

  \- 6

- Range of an *N*-bit sign/magnitude number:

Sign/Magnitude Numbers

Problems:

- Addition doesn't work, for example -6 + 6:

$$\begin{array}{r} 1110 \\ +\ 0110 \\ \hline 10100 \end{array}$$ (wrong!)

- Two representations of 0 (± 0):

1000

0000

Two’s Complement Numbers

- Don’t have same problems as sign/magnitude numbers:
  - **Addition works**
  - **Single representation for 0**

Two's Complement Numbers

- MSB has weight of -2^{N-1}

$$A = a_{N-1}(-2^{N-1}) + \sum_{i=0}^{N-2} a_i \, 2^i$$

- Most positive 4-bit number:
- Most negative 4-bit number:

Decimal	2's Complement
-1	11111111
-2	11111110
-3	11111101
-4	11111100
…	…
-128	10000000

- The most significant bit still indicates the sign (1 = negative, 0 = positive)
- Range of an *N*-bit two's complement number:

Reverse the Sign

- **Reverse the sign** of a two's complement number
- **Method:**
    1. Invert the bits
    2. Add 1
- **Example:** Reverse the sign of $3_{10} = 0011_2$
    1. **1100**
    2.

Historically, this reversing the sign method has been called: "Taking the Two's complement".
But this terminology can be confusing, so we instead we call it "reversing the sign".

Two's Complement Examples

- Reverse the sign of $6_{10} = 0110_2$
    1.
    2.

- What is the decimal value of the two's complement number 1001_2?
    1.
    2.

Two's Complement Addition

- 3-step procedures:

1. Convert the decimal numbers into the binary form with negative numbers expressed in the 2's complement form
2. Perform binary addition
3. Sign: If both numbers positive (negative) → the result positive (negative).
   If the sign is incorrect, **overflow** has occurred!

Both numbers positive

$$\begin{array}{r} 00000111 \\ +\ 00000100 \\ \hline 00001011 \end{array} \qquad \begin{array}{r} 7 \\ +\ 4 \\ \hline 11 \end{array}$$

Both numbers negative

$5_{10} = 00000101_2$

2's complement

$9_{10} = 00001001_2$

Discard carry

$$\begin{array}{r} 11111011 \\ +\ 11110111 \\ \hline 1\ \ 11110010 \end{array} \qquad \begin{array}{r} -5 \\ +\ -9 \\ \hline -14 \end{array}$$

$00001110_2 = 14_{10}$

Two's Complement Addition

❑ 3-step procedures:

1. Convert the decimal numbers into the binary form with negative numbers expressed in the 2's complement form
2. Perform binary addition
3. Sign: If both numbers positive (negative) → the result positive (negative).
   If the sign is incorrect, **overflow** has occurred!

8-bit numbers [-128, +127]

01111101	125
+ 00111010	+ 58
10110111	183

incorrect

10000011	-125
+ 11000110	+ -58
1 01001001	-183

incorrect

Two's Complement Addition: one negative number

$$\begin{array}{r} 00010000 \\ +\ 11101000 \\ \hline 11111000 \end{array} \qquad \boxed{\begin{array}{r} 16 \\ +\ -24 \\ \hline -8 \end{array}}$$

1. $16_{10} = 00010000_2$; $24_{10} = 00011000_2$ → $-24_{10} = 11101000_2$
2. $+\ 11101000$

2's complement

3. $00001000_2 = 8_{10}$ → -8

No carry indicates the result is negative

$$\begin{array}{rr} & 00001111 \\ & +\ 11111010 \\ \hline \text{Carry} \rightarrow 1 & 00001001 \end{array} \qquad \boxed{\begin{array}{r} 15 \\ +\ -6 \\ \hline 9 \end{array}}$$

1. $15_{10}=00001111_2$; $6_{10}=00000110_2$ → $-6_{10}=11111010_2$
2. $+\ 11111010$
3. $00001001_2 = 9_{10}$

Carry indicates the result is positive

Subtraction

- Subtract a 2’s complement number by reversing the sign and adding.
- Reverse sign by taking 2’s complement

- Ex: 3 – 5 = 3 + (-5)

	Binary	Decimal
	0011	3
+	1011	-5
	1110	-2

- Ex: 8 – 3 = 8 + (-3)

	Binary	Decimal
	1000	8
+	1101	-3
	10101	5

Discard the carry

Multiplication

- **Direct addition :**
  - Very lengthy if the multiplier is a large number
  - both numbers must be in true (uncomplemented) form.
- **Partial product**
  - Same sign → Positive;
  - Different signs → Negative

A Decimal Example

$$\begin{array}{r} 239 \\ \times\ 123 \\ \hline 717 \\ 478 \\ +\ 239 \\ \hline 29{,}397 \end{array}$$

EXAMPLE 2–22

01010011 (multiplicand) and $\boxed{11000101}$ (multiplier).

83 -59

1010011		Multiplicand
$\times$ $\boxed{0111011}$	59	Multiplier
1010011		1st partial product
+ 1010011		2nd partial product
11111001		Sum of 1st and 2nd
+ 0000000		3rd partial product
011111001		Sum
+ 1010011		4th partial product
1110010001		Sum
+ 1010011		5th partial product
100011000001		Sum
+ 1010011		6th partial product
1001100100001		Sum
+ 0000000		7th partial product
4897 01001100100001		Final product
10110011011111		2's complement

-4897

Division

- Division operation in computers is accomplished using subtraction
- Quotient and Remainder
  - When to stop the subtraction: the remainder is a zero or a negative number
  - Quotient: # of times of subtraction performed
- Sign of the quotient
  - Same sign → Positive;
  - Different signs → Negative

What about dividing 100 by -25?

EXAMPLE 2–23 Divide 01100100 by 00011001

100 25

Step 1: 00011001 → 11100111

2's complement

Step 2

$$\begin{array}{r} 01100100 \\ +\ 11100111 \\ \hline 01001011 \end{array}$$

Add 1 to quotient

Step 3:

$$\begin{array}{r} 01001011 \\ +\ 11100111 \\ \hline 00110010 \end{array}$$

Add 1 to quotient

Step 4:

$$\begin{array}{r} 00110010 \\ +\ 11100111 \\ \hline 00011001 \end{array}$$

Add 1 to quotient

Step 5:

$$\begin{array}{r} 00011001 \\ +\ 11100111 \\ \hline 00000000 \end{array}$$

Add 1 to quotient

Quotient=4_{10}=0100

Number System Comparison

Number System	Range
Unsigned	$[0, 2^N-1]$
Sign/Magnitude	$[-(2^{N-1}-1), 2^{N-1}-1]$
Two's Complement	$[-2^{N-1}, 2^{N-1}-1]$

For example, 4-bit representation:

	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Unsigned									0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Two's Complement	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111								
Sign/Magnitude		1111	1110	1101	1100	1011	1010	1001	0000 1000	0001	0010	0011	0100	0101	0110	0111								

Fixed-Point Numbers

Number Systems

- Numbers we can represent using binary representations
  - **Positive numbers**
    - Unsigned binary
  - **Negative numbers**
    - Two's complement
    - Sign/magnitude numbers

- What about **fractions**?

Numbers with Fractions

- Two common notations:
- **Fixed-point:** binary point fixed
- **Floating-point:** binary point floats to the right of the most significant 1

Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Binary point is implied
- The number of integer and fraction bits must be agreed upon beforehand

Unsigned Fixed Point Formats

- **Ua.b:** unsigned number with
  - **a** integer bits
  - **b** fractional bits.
- **Example:** **6.75** is
  - **U4.4:** 01101100
  - **U3.5:** 11011000
  - **U6.2:** 00011011
- 8, 16, and 32-bit fixed point numbers are common
  - **U8.8** often represents sensor data, audio, pixels
  - **U16.16** used for higher precision signal processing

Signed Fixed Point Formats

- **Qa.b:** signed 2's complement number with
  - **a** integer bits (including the sign bit)
  - **b** fractional bits
- **To negate a Q fixed point number:**
  - Invert the bits
  - Add one to the LSB
- **Example:** write **-6.75** in Q4.4
  - 6.75 = 01101100
  - Invert: 10010011
  - Add 1 LSB: 10010100
- **Q1.15** (aka Q15) is common for signal processing (1, -1]

Saturating Arithmetic

- Fixed point **overflow** is usually bad
  - Produces undesired artifacts:
    - Video: dark pixel in middle of bright pixels
    - Audio: clicking sounds
- **Saturating arithmetic**
  - Instead of overflowing, use largest value
  - In U4.4: 11000000 + 01111000 = 11111111
    - 12 + 7.5 = 15.9375

Floating-Point Numbers

Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write 273_{10} in scientific notation:
  - $$\mathbf{273 = 2.73 \times 10^2}$$
- In general, a number is written in scientific notation as:

$$\pm \mathbf{M} \times \mathbf{B}^{\mathbf{E}}$$

  - **M** = mantissa
  - **B** = base
  - **E** = exponent
  - In the example, M = 2.73, B = 10, and E = 2

Floating vs. Fixed Point Numbers

- Floating point numbers are like **scientific notation**
  - Allow a **greater dynamic range** of smallest to largest
  - **Arithmetic is harder**
    - Mantissa must be aligned before adding
    - This costs performance and power
- Fixed point numbers are **harder for the programmer**
  - **Smaller dynamic range**
  - Take care of **overflow**

- **Floating Point** is preferred for general-purpose computing where programming time is most important
- **Fixed Point** is preferred for signal processing performance, power, and hardware cost matter most
  - Machine learning, video

Floating-Point Numbers

1 bit	8 bits	23 bits
Sign	**Exponent**	**Mantissa**

- **Example:** represent the value 228_{10} using a 32-bit floating point representation

IEEE 754 floating-point standard

- First bit of the mantissa is always 1: $228_{10} = 11100100_2 = \mathbf{1.11001} \times 2^7$
  - So, no need to store it: *implicit leading 1*
- Store just **fraction bits** in 23-bit field

- *Biased exponent*: bias = 127 (01111111_2)
  – Biased exponent = bias + exponent
  – Exponent of 7 is stored as: $127 + 7 = 134 = 0x10000110_2$

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**

in hexadecimal: **0x43640000**

Why Biased Exponent?

- The range of the biased exponent: [-127, +128]
  - Extremely large or Small numbers can be represented in this way
-

Floating-Point Example

- Write **-58.25$_{10}$** in floating point (IEEE 754)

1. Convert magnitude of decimal to binary:
    - **58.25$_{10}$** = **111010.01$_2$**
2. Write in binary scientific notation:

- **1.1101001×2^5**

3. Fill in fields:
    - **Sign bit:** **1** (negative)
    - **8 exponent bits:** (127 + 5) = 132 = **10000100$_2$**
    - **23 fraction bits:** **110 1001 0000 0000 0000 0000**

1 bit	8 bits	23 bits
Sign	**Exponent**	**Fraction**

-
- in hexadecimal: **0xC2690000**

Floating-Point Special Cases

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
-∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Non-positional/Non-weighted Number Systems

Positional/Weighted Number System

Positional/Weighted number system

The value of a symbol is determined by its position

Examples: decimal / binary / octal / hexadecimal number systems …

Non-positional/Non-weighted number system

The value of a symbol is NOT determined by its position

Examples: gray code, cyclic code …

Binary Coded Decimal (BCD)

- Express each of the decimal digits with a binary code.
- The 8421 code

TABLE 2–5

Decimal/BCD conversion.

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

EXAMPLE

170_{10}

(c) 1 7 0

↓ ↓ ↓

000101110000

2469_{10}

(d) 2 4 6 9

↓ ↓ ↓ ↓

0010010001101001

- BCD-to-Decimal Conversion
  - Starting from the right-most bit and divide the code into groups of four bits

BCD Addition

1. Add the two BCD numbers, using the rules for binary addition
2. If a 4-bit sum ≤ 9 → Valid BCD number
3. Otherwise, add 6 (0110) to the 4-bit sum

Decimal/BCD conversion.

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

EXAMPLE

$$
\begin{array}{rrl}
 & 1001 & \qquad 9 \\
 & +\ 0100 & \qquad +4 \\
\hline
 & \boxed{1101} & \rightarrow \text{Invalid BCD number } (>9) \qquad 13 \\
 & +\ 0110 & \text{Add 6} \\
\hline
\underbrace{0001}_{\downarrow\ 1} & \underbrace{0011}_{\downarrow\ 3} & \text{Valid BCD number}
\end{array}
$$

Why adding 6?

The Gray Code (I)

- Only a single bit change from one code word to the next in sequence

Four-bit Gray code.

Decimal	Binary	Gray Code	Decimal	Binary	Gray Code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Gray Code: How to Transfer (II)

EXAMPLE

(a) Convert the binary number 11000110 to Gray code.

(b) Convert the Gray code 10101111 to binary.

Solution

(a) Binary to Gray code:

Start from the MSB

1 −+→ 1 −+→ 0 −+→ 0 −+→ 0 −+→ 1 −+→ 1 −+→ 0

1 0 1 0 0 1 0 1

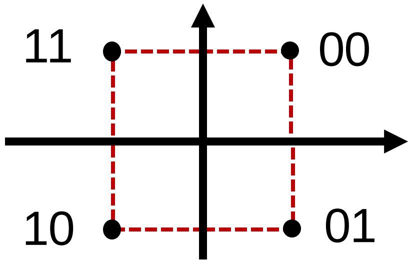
(b) Gray code to binary:

Start from the MSB

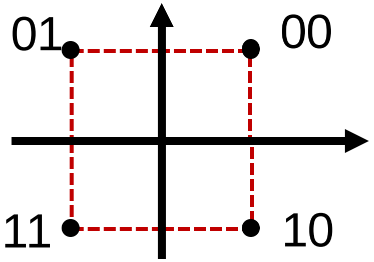
1 0 1 0 1 1 1 1

1 1 0 0 1 0 1 0

Without gray code



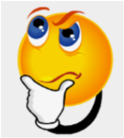
With gray code



011->100,
intermediate states:
010,001,101,110,111

Encoding transmit symbols with the Gray Code

- Transmitter sends one of the four symbols;
- Transmitted symbols are distorted by noise;
- The receiver tries to decode the distorted symbols;
- The probability of making a symbol detection error is ***inversely*** proportional to the symbol distance



Why Gray Code has an advantage?

Alphanumeric Codes

- ASCII :

  American

  Standard

  Code for

  Information

  Interchange

Control Characters				Graphic Symbols											
Name	Dec	Binary	Hex	Symbol	Dec	Binary	Hex	Symbol	Dec	Binary	Hex	Symbol	Dec	Binary	Hex
NUL	0	0000000	00	space	32	0100000	20	@	64	1000000	40	'	96	1100000	60
SOH	1	0000001	01	!	33	0100001	21	A	65	1000001	41	a	97	1100001	61
STX	2	0000010	02	”	34	0100010	22	B	66	1000010	42	b	98	1100010	62
ETX	3	0000011	03	#	35	0100011	23	C	67	1000011	43	c	99	1100011	63
EOT	4	0000100	04	$	36	0100100	24	D	68	1000100	44	d	100	1100100	64
ENQ	5	0000101	05	%	37	0100101	25	E	69	1000101	45	e	101	1100101	65
ACK	6	0000110	06	&	38	0100110	26	F	70	1000110	46	f	102	1100110	66
BEL	7	0000111	07	’	39	0100111	27	G	71	1000111	47	g	103	1100111	67
BS	8	0001000	08	(	40	0101000	28	H	72	1001000	48	h	104	1101000	68
HT	9	0001001	09	)	41	0101001	29	I	73	1001001	49	i	105	1101001	69
LF	10	0001010	0A	*	42	0101010	2A	J	74	1001010	4A	j	106	1101010	6A
VT	11	0001011	0B	+	43	0101011	2B	K	75	1001011	4B	k	107	1101011	6B
FF	12	0001100	0C	,	44	0101100	2C	L	76	1001100	4C	l	108	1101100	6C
CR	13	0001101	0D	–	45	0101101	2D	M	77	1001101	4D	m	109	1101101	6D
SO	14	0001110	0E	.	46	0101110	2E	N	78	1001110	4E	n	110	1101110	6E
SI	15	0001111	0F	/	47	0101111	2F	O	79	1001111	4F	o	111	1101111	6F
DLE	16	0010000	10	0	48	0110000	30	P	80	1010000	50	p	112	1110000	70
DC1	17	0010001	11	1	49	0110001	31	Q	81	1010001	51	q	113	1110001	71
DC2	18	0010010	12	2	50	0110010	32	R	82	1010010	52	r	114	1110010	72
DC3	19	0010011	13	3	51	0110011	33	S	83	1010011	53	s	115	1110011	73
DC4	20	0010100	14	4	52	0110100	34	T	84	1010100	54	t	116	1110100	74
NAK	21	0010101	15	5	53	0110101	35	U	85	1010101	55	u	117	1110101	75
SYN	22	0010110	16	6	54	0110110	36	V	86	1010110	56	v	118	1110110	76
ETB	23	0010111	17	7	55	0110111	37	W	87	1010111	57	w	119	1110111	77
CAN	24	0011000	18	8	56	0111000	38	X	88	1011000	58	x	120	1111000	78
EM	25	0011001	19	9	57	0111001	39	Y	89	1011001	59	y	121	1111001	79
SUB	26	0011010	1A	:	58	0111010	3A	Z	90	1011010	5A	z	122	1111010	7A
ESC	27	0011011	1B	;	59	0111011	3B	[	91	1011011	5B	{	123	1111011	7B
FS	28	0011100	1C	<	60	0111100	3C	\	92	1011100	5C	|	124	1111100	7C
GS	29	0011101	1D	=	61	0111101	3D	]	93	1011101	5D	}	125	1111101	7D
RS	30	0011110	1E	>	62	0111110	3E	^	94	1011110	5E	~	126	1111110	7E
US	31	0011111	1F	?	63	0111111	3F	_	95	1011111	5F	Del	127	1111111	7F

Error Codes: Parity Bit for Error Detection

- A parity bit is attached to a group of bits to make the total number of 1's in a group always even or always odd.

TABLE 2–8

The BCD code with parity bits.

Even Parity		Odd Parity	
P	BCD	*P*	BCD
0	0000	1	0000
1	0001	0	0001
1	0010	0	0010
0	0011	1	0011
1	0100	0	0100
0	0101	1	0101
0	0110	1	0110
1	0111	0	0111
1	1000	0	1000
0	1001	1	1001

EXAMPLE

An odd parity system receives the following code groups: 10110, 11010, 110011, 110101110100, and 1100010101010. Determine which groups, if any, are in error.

	# of 1's
10110	3
11010	3
110011	4
110101110100	7
1100010101010	6

Since odd parity is required, any group with an even number of 1s is incorrect. The following groups are in error: 110011 and 1100010101010.

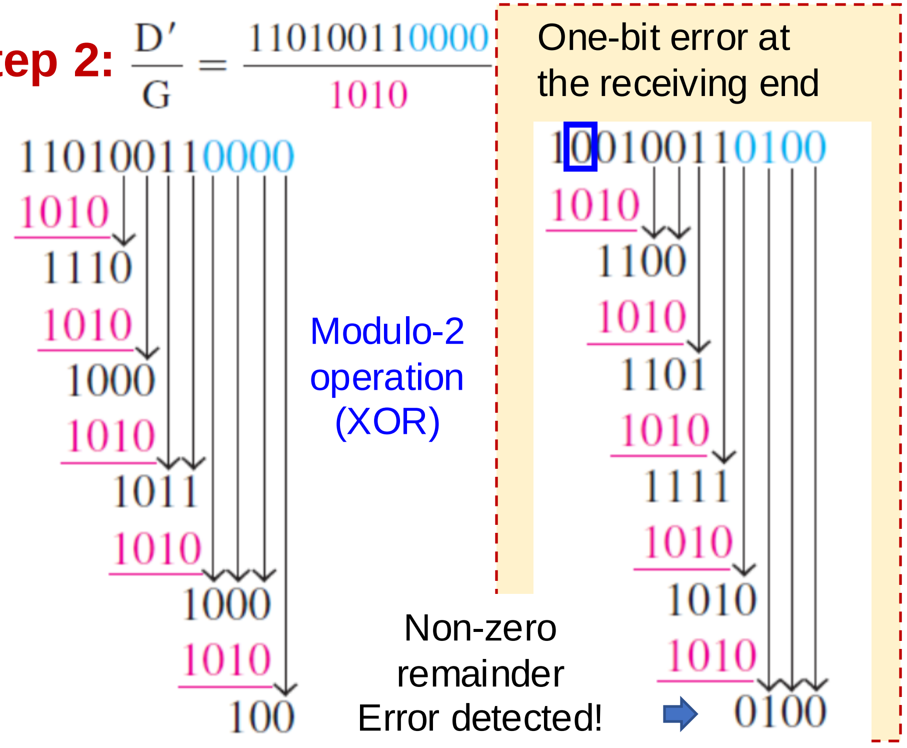
Error Codes: Cyclic Redundancy Check

- Cyclic Redundancy Check (CRC) : an error detection method that can detect multiple errors in data blocks
- At the sending end, a checksum is appended to a block of data.
- At the receiving end, the check sum is generated and compared to the sent checksum. If the check sums are zeros, no error is detected.

D: 11010011 Data
G: 1010 Generator code

Step 1: Append 0000 to the data
D' = 110100110000

Step 2: $\frac{D'}{G} = \frac{110100110000}{1010}$



Remainder (Checksum) : 0100

Step 3: The transmitted CRC is 110100110100

Chapter Review