Number Systems and Codes

1. Binary based number system

   1.1 Number Systems

   1.2 Binary Codes for Decimal Digits

   1.3 Parity and Error Correction

1.1 Number Systems

- Decimal (base 10 – 0,1,2,3,4,5,6,7,8,9)
  E.g. 1, 2, 3, 4, 5, … 10, 11, 12, …

- Binary (base 2 - 0,1)
  E.g. 000, 001, 010, 011, …

- Octal (base 8 - 0,1,2,3,4,5,6,7)
  E.g. 00, 01, 02, …, 07, 10, 11, …17, 20, …

- Hexadecimal (base 16 - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)
  E.g. 1, 2, 3, …, 9, A, B, C, D, E, F, 10, 11, …

Number	Decimal	Binary	Octal	Hexadecimal
Zero	0	0	0	0
One	1	1	1	1
Two	2	10	2	2
Three	3	11	3	3
Four	4	100	4	4
Five	5	101	5	5
Six	6	110	6	6
Seven	7	111	7	7
Eight	8	1000	10	8
Nine	9	1001	11	9
Ten	10	1010	12	A
Eleven	11	1011	13	B
Twelve	12	1100	14	C
Thirteen	13	1101	15	D
Fourteen	14	1110	16	E
Fifteen	15	1111	17	F
Sixteen	16	10000	20	10
Seventeen	17	10001	21	11

1.1.1 Radix System Representation

Given a number with radix (or base) r (r is usually a positive integer, but not necessarily)

$$a_n a_{n-1} \ldots a_1 a_0 . a_{-1} \ldots a_{-m+1} a_{-m}$$

where $a_i = x$, x (0,1, … , r).
its decimal value is given by

$$a_{n-1} r^{n-1} + a_{n-2} r^{n-2} + \ldots + a_2 r^2 + a_1 r + a_0$$
$$+ a_{-1} r^{-1} + a_{-2} r^{-2} + \ldots + a_{-m+1} r^{-m+1} + a_{-m} r^{-m}$$

n: number of digits
r: radix (base)
a_n : coefficients (digits)

1.1.2 Polynomial Form

For decimal number, $r = 10$

e.g. $(7392)_{10}$

$= 7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$

For binary number, $r = 2$

e.g. $(1101.011)_2$

$= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$

For octal number, $r = 8$

e.g. $(527)_8$

$= 5 \times 8^2 + 2 \times 8^1 + 7 \times 8^0$

1.1.3 Number Conversion

1.1.3.1 Binary to Decimal

For binary number, $r = 2$

e.g. $(1101.01)_2$

$= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

$= 8 + 4 + 0 + 1 + 0 + 0.25$

$= (13.25)_{10}$

1.1.3.2 Decimal to Binary

- Division method

i. Divide the number by 2
ii. The remainder (either 0 or 1) gives the Least Significant Bit (LSB).
iii. Divide the quotient by 2 repeatedly until the quotient becomes 0.

Example: convert 98_{10} to binary number

98 / 2 = 49	Remainder = **0**	(a_0)	**0 (LSB)**
49 / 2 = 24	Remainder = 1	(a_1)	10
24 / 2 = 12	Remainder = 0	(a_2)	010
12 / 2 = 6	Remainder = 0	(a_3)	0010
6 / 2 = 3	Remainder = 0	(a_4)	00010
3 / 2 = 1	Remainder = 1	(a_5)	100010
1 / 2 = 0	Remainder = **1**	(a_6)	1100010 **(MSB)**

Ans = 1100010_2

$$a_{n-1} r^{n-1} + a_{n-2} r^{n-2} + \dots + a_2 r^2 + a_1 r + a_0$$

1.1.3.3 Decimal to Octal

Example: convert 98_{10} to Octal number

98 / 8 = 12	Remainder = 2	(a_0)	2
12 / 8 = 1	Remainder = 4	(a_1)	42
1 / 8 = 0	Remainder = 1	(a_2)	142

Ans = 142_8

2.1.3.4 Octal to Decimal

Example: convert 237_8 to Decimal number

$= 2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0$

$= 159_{10}$

1.1.3.5 Decimal to Hexadecimal

Example: convert 675_{10} to Hexadecimal number

675 / 16 = 42	Remainder = **3**	(a_0)	3
42 / 16 = 2	Remainder = A	(a_1)	A3
2/ 16 = 0	Remainder = 2	(a_2)	2A3

Ans = $2A3_{16}$

2.1.3.6 Hexadecimal to Decimal

Example: convert $E4C_{16}$ to Decimal number

$= E \times 16^2 + 4 \times 16^1 + C \times 16^0$

$= 3660_{10}$

1.1.3.7 Octal to Hexadecimal

Method 1:

Convert the octal number to decimal number, then convert decimal number to hexadecimal number.

Method 2 – bit regrouping:

Convert the octal number to binary number, then to hexadecimal number (bit re-grouping)

Example:

27534_8 = 010 111 101 011 100 (noted that 3-bit

2 7 5 3 4 = 1 octal digit)

0010 1111 0101 1100 (noted that 4-bit

= $2F5C_{16}$ = 1 hexadecimal digit)

1.1.3.8 Hexadecimal to Octal

Likewise

Convert the hexadecimal number to binary number, then to octal number

Example:

$6A3C9_{16}$ = 0110 1010 0011 1100 1001

6 A 3 C 9

001 101 010 001 111 001 001

= 1521711_8

1.1.3.9 Decimal Fractions to Binary

Example: converting the decimal value .625 to a binary representation

- Multiply the decimal fraction by 2. The integer part of the result is the first binary digit.

  .625 x 2 = 1.25

  i.e. .625 = .1???....(base 2)

- Take the fraction and multiply by 2. The integer part is the secondary binary digit.

  .25 x 2 = 0.50

  i.e. .625 = .10???....(base 2)

- Multiply the decimal fraction by 2. The integer part of the result is the first binary digit to the right of the point.

  .5 x 2 = 1.00

  i.e. .625 = .101???....(base 2)

- Fraction is 0, stop (Since .00 x 2 = 0.00)

  i.e. .625 x 2 = .101 (base 2)

- If the process cannot be converged, it requires infinite long binary digits to represent the decimal point number.
- Physical digital system is designed with fixed number binary bit length, truncation is required to *approximate* the real value.

1.2 Binary Codes for Decimal Digits

- In our daily life, we comprehend decimal number easily.
- In digital system design, hardware circuit only comprehends binary number.
- Decimal digits (0-9) can be coded with different binary representations.
- Weighted codes: 8421, 5421, 2421
- Non-weight codes: Excess-3 code, Gray code
- Alphanumeric code: ASCII code

1.2.1 8421 code (Binary Coded Decimal – BCD code)

- 8,4,2,1 are the weights for the digits
- Use 4-bit to represent a decimal digit
  - Code: $a_3a_2a_1a_0$
  - Value: $(8 \times a_3) + (4 \times a_2) + (2 \times a_1) + (1 \times a_0)$
- Example
  - Code: 0110
  - Value: $(8 \times 0) + (4 \times 1) + (2 \times 1) + (1 \times 0)$
  - $= 4 + 2 = 6$

Decimal Digit	8421 Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Unsued	1010
	1011
	1100
	1101
	1110
	1111

IMPORTANT: BCD ≠ Binary weighted code !!!

- For decimal system:
  - 739 read as " seven hundred and thirty nine"
- For BCD representation:
  - 739 would be stored as
  - 0111 0011 1001 (12 bits)
  - 739 read as seven-three-nine
- For binary representation:
  - $739_{10} = 1011100011_2$ (10 bits)

1.2.2 Summary of weighted codes for decimal digits

Value (decimal digit)	8421 code	5421 code	2421 code
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111
unused	1010	0101	0101
	1011	0110	0110
	1100	0111	0111
	1101	1101	1000
	1110	1110	1001
	1111	1111	1010

The code are the same for the first 5 digits

(5 x 1) + (4 x 0) + (2 x 1) + (1 x 1)
= 5 + 2 + 1
= 8

(2 x 1) + (4 x 1) + (2 x 1) + (1 x 0)
= 2 + 4 + 2
= 8

4-bit codes have 16 combinations, but we just use 10 of them. The remaining 6 codes are unused and undefined

1.2.3 Properties of 2421 code

- A self-complementing code
  - The complement of 0 is 9 (0000 ⇔ 1111)
  - The complement of 1 is 8 (0001 ⇔ 1110)
  - The complement of 2 is 7 (0010 ⇔ 1101)
  - The complement of 3 is 6 (0011 ⇔ 1100)
  - The complement of 4 is 5 (0100 ⇔ 1011)

Complementing:

0 turns to 1

and 1 turns to 0 circui

1.2.4 Excess-3 code (XS3 code)

Value (decimal digit)	8421code	Excess 3 code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100
unused	1010	0000
	1011	0001
	1100	0010
	1101	1101
	1110	1110
	1111	1111

Excess-3 code is a shifted 8421 code:

XS3'0 = 8421's 0 + 3 = 8421's 3
XS3'1 = 8421's 1 + 3 = 8421's 4
:
:
XS3'6 = 8421's 6 + 3 = 8421's 9
XS3'7 = 8421's 7 + 3
XS3'8 = 8421's 8 + 3
XS3'9 = 8421's 9 + 3

XS3 code is a self complementing code:
Complement of 0 is 9 (0011 ↔ 1100)
Complement of 1 is 8 (0100 ↔ 1011)
:
:
Complement of 4 is 5 (0111 ↔ 1000)

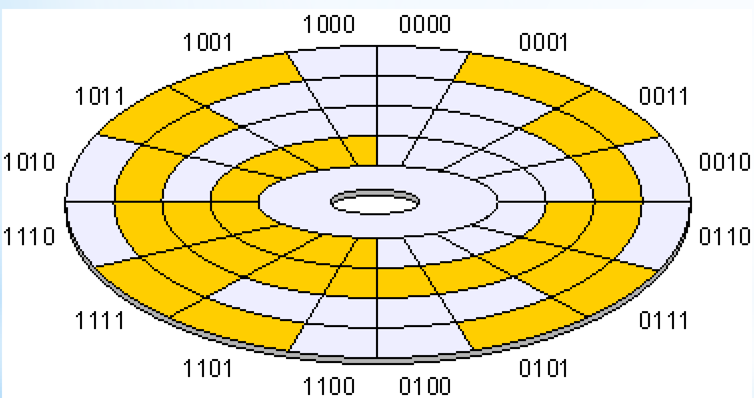
1.2.5 Gray code

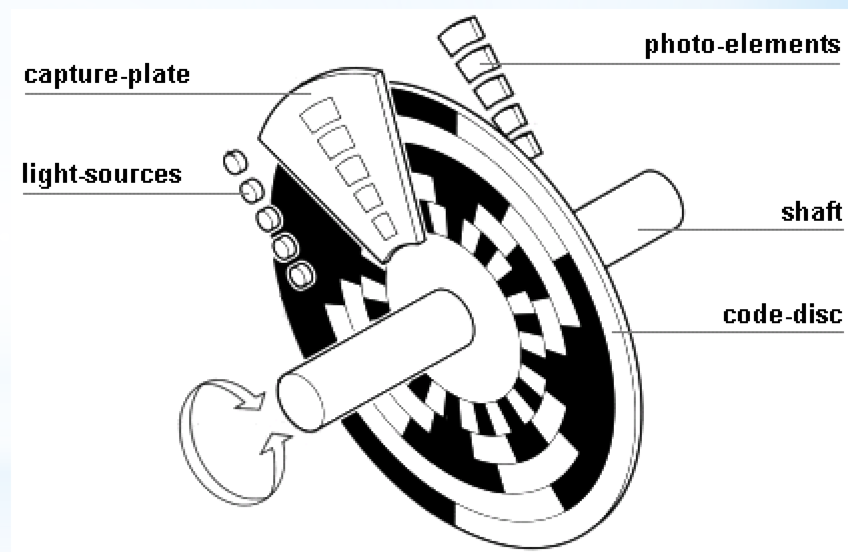
2-bit gray code	3-bit gray code	4-bit gray code	decimal
00	000	0000	0
01	001	0001	1
11	011	0011	2
10	010	0010	3
	110	0110	4
	111	0111	5
	101	0101	6
	100	0100	7
		1100	8
		1101	9
		1111	10
		1110	11
		1010	12
		1011	13
		1001	14
		1000	15

- Not limited to represent decimal number.
- Non-weighted code.
- One bit difference between adjacent codes

Interesting properties of Gray code

- Error detection
- Rotational Position detection





1.2.6 Binary code to Gray code conversion

i. Place a leading 0 before the MSB.

ii. Perform XOR operation to the adjacent bits starting from the left hand side of this number will result in gray code. (Add an leading “0” to the left if its begins with “1”)

e.g. 82_{10} ---> 1010010_2

```
0 1 0 1 0 0 1 0
 v v v v v v v
 1 1 1 1 0 1 1   ←—— Gray code
```

XOR

$00 \rightarrow 0$

$01 \rightarrow 1$

$10 \rightarrow 1$

$11 \rightarrow 0$

1.2.7 Gray code to Binary code conversion

i. Copy the first 1 starting from left hand side and continue with 1 until the next 1 appears in the gray code.
ii. Change to 0 and continue with 0 until the next 1 appears in the gray code.
iii. Change to 1 and follow steps (i) & (ii).

10001011 ⟵ gray code

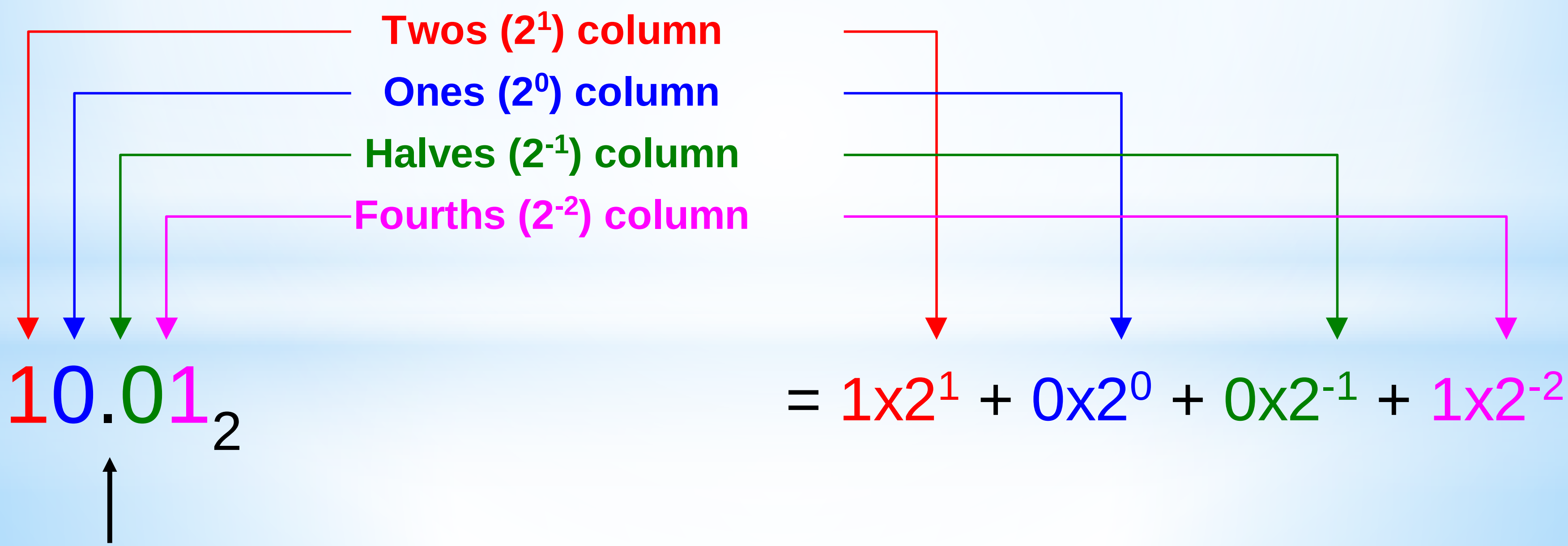
11110010 ⟵ binary code

1.2.8 Floating point number

- Integer number has limitations.
- Example: 8-bit from 0 to 255.
- Real number requires decimal point.
- Fixed point position is not flexible.
- Floating point representation allows large value number with precision.

1.2.8.1 Fixed-point representation

- Decimal point is placed in an appropriate position.
- Every column represents a power of 2.



$$10.01_2 = 1\text{x}2^1 + 0\text{x}2^0 + 0\text{x}2^{-1} + 1\text{x}2^{-2}$$

1.2.8.2 Fixed-point arithmetic

- For system with fixed bit length, numbers with different decimal point location require alignment before arithmetic operation takes place.

Example for 8-bit system:

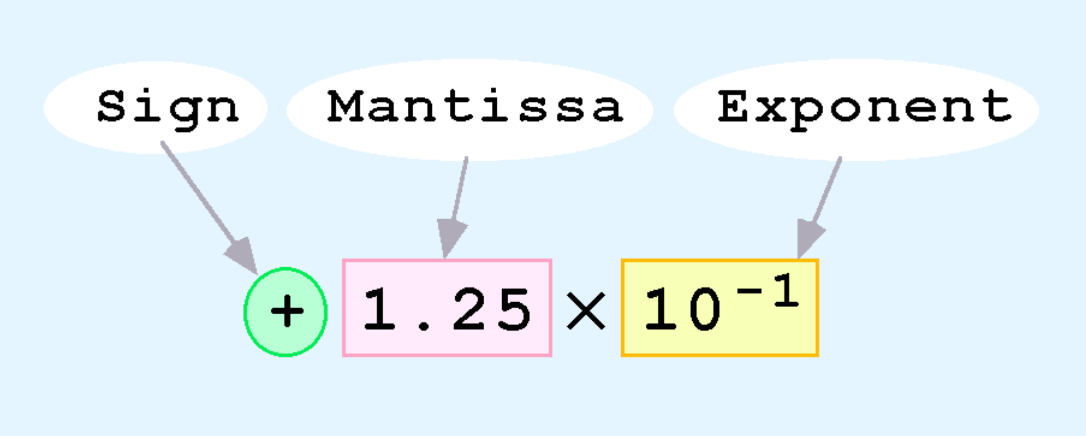
```
1101.1011 + 101.10011
```

Problem occurs:

```
 1101.1011       or     101.10110
+0101.1001             +101.10011
```

1.2.8.3 Floating point number representation

- A floating point number consists of 3 components
  - Sign bit
  - Exponent
  - Mantissa



- It is a scientific notation for floating-point representation.

1.2.8.4 IEEE floating point number format

- Computer representation of a floating-point number consists of three fixed-size fields.

S	Exponent	Mantissa

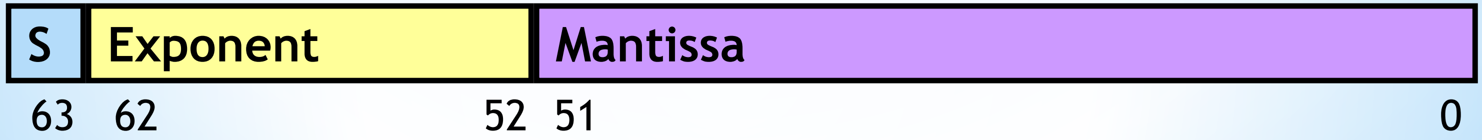
  - MSB is sign bit: *S* - determines whether number is negative or positive
  - Exponent field: *E* - weights value by power of two
  - Mantissa field: *M* - normally a fractional value in range [1.0,2.0)
- Numeral form:

$$-1^{s} \times 2^{Bias+E} \times M$$

1.2.8.5 Floating point precisions

- Sizes
  - Single precision: 8 `exp` bits, 23 `Mant` bits, 1 sign bit
    - Bias = -127
    - 32 bits total

  - Double precision: 11 `exp` bits, 52 `Mant` bits, 1 sign bit
    - Bias = -1023
    - 64 bits total



- In the single-precision format, the number is presented by the sign S, biased exponent E and the fraction F in the form:

$$(-1)^S \times 2^{E-127} \times 1.F \qquad (F = \text{fraction} \equiv \text{mantissa})$$

- The 1 is not stored physically.
- Example 1:

$$C25A8000 = \underbrace{1}_{\text{Sign}} \quad \underbrace{10000100}_{\text{biased exponent}} \quad \underbrace{101101010\cdots0}_{\text{fraction}}$$

- True exponent is 132 – 127 = 5, and the floating-point number being represented is

$$\begin{aligned} -1.10110101\times 2^5 &= -110110.101 \\ &= -(2^5+2^4+2^2+2^1+2^{-1}+2^{-3}) \\ &= -54.625 \end{aligned}$$

- Example 2:

$$313.125_{10} = 100111001_2 + .001_2$$
$$= 100111001.001_2$$
$$= 1.00111001001 \times 2^8$$

i.e.

S = 0, E = 127 + 8 = 135 = 10000111, and F = 00111001001

The single-precision form of 313.125_{10} is:

$$\underbrace{0}_{\text{sign}}\ \underbrace{10000111}_{\text{biased exponent}}\ \underbrace{001110010010\cdots0}_{\text{fraction}} = 439C9000$$

For double-precision format, there are 11 exponent bits and 52 fraction bits. The range of number represented is much larger and it improves the precision of the real number.

1.2.9 Alphanumeric code

- Binary code is easy for digital hardware to manipulate, but difficult for human to perceive.
- For user friendly purpose, characters & numeric are encoded using a fixed length to facilitate human computer interface.
- ASCII - American Standard Code for Information Interchange
  - 7-bit to provide 128 characters
  - including some control codes

ASCII code

ASCII Code	Value
000 0000	NULL
...	...
010 0000	Space
010 0001	! (exclamation mark)
010 0010	" (double quote)
...	...
011 0000	0
011 0001	1
...	...
011 1010	: (colon)
...	...
100 0001	A
...	...
101 1010	Z
...	...
110 0001	a
...	...
111 1010	z
...	...

control signals
symbols
numeric characters
symbols
capital letters
symbols
small letters
symbols

The word *Logic* would be coded as:

100 1100 110 1111
L o

110 0111 110 1001 110 0011
g i c

739 would be coded as:

011 0111 011 0011 011 1001
7 3 9

(Please refer to the complete ASCII table in your book)

ASCII Code

- 7-bit to provide 128 characters (characters, numeric, control)

e.g. A : 100 0001 9: 011 1001 ~ : 111 1110

$: 010 0100 ESC : 001 1011 BS : 000 1000

	$c_6c_5c_4$							
$c_3c_2c_1c_0$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EQT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	S0	RS	.	>	N	^	n	~
1111	S1	US	/	?	O	_	o	DEL

ASCII code used in computer applications:

COM Port
ASCII code

Keyboard
ASCII code

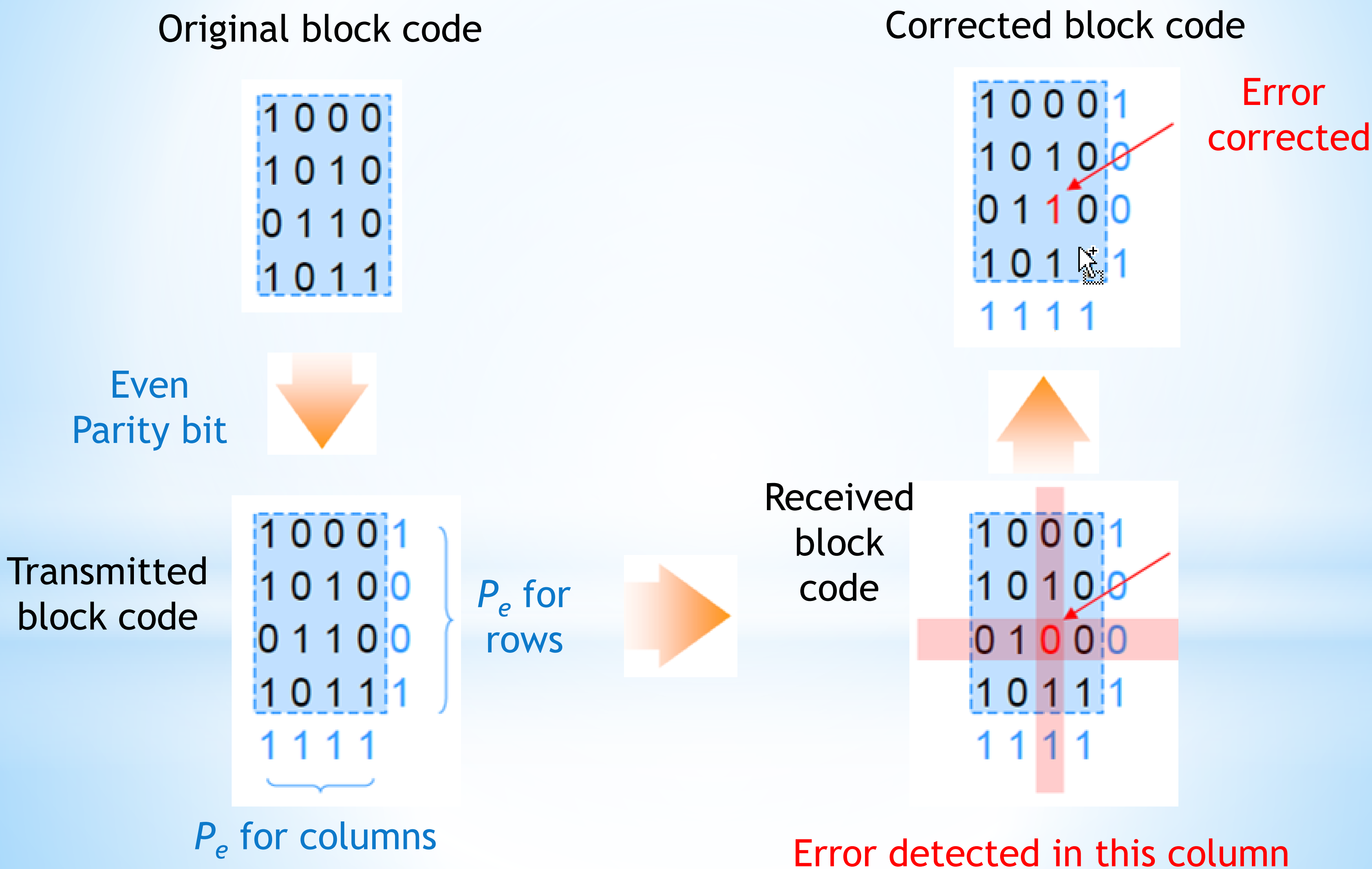
1.3 Parity and Error Correction

- The simplest method for error detection is using parity bit
- An additional bit attaches to the original code
- Two kinds of party bit (even parity or odd parity)
- The value of parity bit is defined by the total no. of “1” in the resulting codeword either even or odd

1.3.1 Parity bit generation

- Original Code: $a1\ a2\ a3\ a4$ (4-bit)
- For even parity bit P_e $a5 = a1 \oplus a2 \oplus a3 \oplus a4$
- For odd parity bit P_o $a5 = a1 \oplus a2 \oplus a3 \oplus a4 \oplus 1$
- even parity = $\overline{\text{odd parity}}$
- Final codeword $a1\ a2\ a3\ a4\ a5$
- Example: code: 10001101
  - Final codeword with P_e : 100011010
  - Final codeword with P_o : 100011011

1.3.2 Error Correction



Error Correction Realization:

Received block code

↓

Generate the local parity code
from the original block code

↓

Compare the local parity code
with the received parity code