ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

COMPUTER ORGANISATION
(TỔ CHỨC MÁY TÍNH)

Number Systems and Codes

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Delete slide QUICK REVIEW QUESTIONS

NUMBER SYSTEMS & CODES

- Information Representations
- Number Systems
- Base Conversion
- Negative Numbers
- Excess Representation
- Floating-Point Numbers
- Decimal codes: BCD, Excess-3, 2421, 84-2-1
- Gray Code
- Alphanumeric Code
- Error Detection and Correction

Read up DLD for details!

INFORMATION REPRESENTATION (1/3)

- Numbers are important to computers
  - Represent information precisely
  - Can be processed
- Examples
  - Represent *yes* or *no*: use 0 and 1
  - Represent the 4 seasons: 0, 1, 2 and 3
- Sometimes, other characters are used
  - Matriculation number: 8 alphanumeric characters (eg: U071234X)

INFORMATION REPRESENTATION (2/3)

- Bit (*B*inary dig*it*)
  - 0 and 1
  - Represent *false* and *true* in logic
  - Represent the *low* and *high* states in electronic devices
- Other units
  - Byte: 8 bits
  - Nibble: 4 bits (seldom used)
  - Word: Multiples of byte (eg: 1 byte, 2 bytes, 4 bytes, 8 bytes, etc.), depending on the architecture of the computer system

INFORMATION REPRESENTATION (3/3)

- N bits can represent up to 2^N values.
  - Examples:
    - 2 bits □ represent up to 4 values (00, 01, 10, 11)
    - 3 bits □ rep. up to 8 values (000, 001, 010, …, 110, 111)
    - 4 bits □ rep. up to 16 values (0000, 0001, 0010, …., 1111)
- To represent M values, $\lceil \log_2 M \rceil$ bits are required.
  - Examples:
    - 32 values □ requires 5 bits
    - 64 values □ requires 6 bits
    - 1024 values □ requires 10 bits
    - 40 values □ how many bits?
    - 100 values □ how many bits?

DECIMAL (BASE 10) SYSTEM (1/2)

- A weighted-positional number system
  - **Base** or **radix** is 10 (the *base* or *radix* of a number system is the total number of symbols/digits allowed in the system)
  - Symbols/digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  - Position is important, as the value of each symbol/digit is dependent on its **type** and its **position** in the number
  - Example, the 9 in the two numbers below has different values:
    - $(75\underline{9}4)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0)$
    - $(\underline{9}12)_{10} = (9 \times 10^2) + (1 \times 10^1) + (2 \times 10^0)$
  - In general,

$$(a_n a_{n-1} \ldots a_0 \,.\, f_1 f_2 \ldots f_m)_{10} = (a_n \times 10^n) + (a_{n-1} \times 10^{n-1}) + \ldots + (a_0 \times 10^0) + (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \ldots + (f_m \times 10^{-m})$$

DECIMAL (BASE 10) SYSTEM (2/2)

- Weighing factors (or weights) are in powers of 10:

  … 10^3 10^2 10^1 10^0 . 10^{-1} 10^{-2} 10^{-3} …

- To evaluate the decimal number 593.68, the digit in each position is multiplied by the corresponding weight:

  $5\times10^2 + 9\times10^1 + 3\times10^0 + 6\times10^{-1} + 8\times10^{-2}$
  $= (593.68)_{10}$

OTHER NUMBER SYSTEMS (1/2)

- Binary (base 2)
  - Weights in powers of 2
  - Binary digits (bits): **0, 1**
- Octal (base 8)
  - Weights in powers of 8
  - Octal digits: **0, 1, 2, 3, 4, 5, 6, 7**.
- Hexadecimal (base 16)
  - Weights in powers of 16
  - Hexadecimal digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**.
- Base/radix R:
  - Weights in powers of R

OTHER NUMBER SYSTEMS (2/2)

- In some programming languages/software, special notations are used to represent numbers in certain bases
  - In programming language C
    - Prefix 0 for octal. Eg: 032 represents the octal number $(32)_8$
    - Prefix 0x for hexadecimal. Eg: 0x32 represents the hexadecimal number $(32)_{16}$
  - In PCSpim (a MIPS simulator)
    - Prefix 0x for hexadecimal. Eg: 0x100 represents the hexadecimal number $(100)_{16}$
  - In Verilog, the following values are the same
    - 8'b11110000: an 8-bit binary value 11110000
    - 8'hF0: an 8-bit binary value represented in hexadecimal F0
    - 8'd240: an 8-bit binary value represented in decimal 240

BASE-*R* TO DECIMAL CONVERSION

- Easy!
  - $1101.101_2 = 1\times2^3 + 1\times2^2 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-3}$
  - $572.6_8 =$
  - $2A.8_{16} =$
  - $341.24_5 =$

DECIMAL TO BINARY CONVERSION

- Method 1
  - Sum-of-Weights Method
- Method 2
  - Repeated Division-by-2 Method (for whole numbers)
  - Repeated Multiplication-by-2 Method (for fractions)

SUM-OF-WEIGHTS METHOD

- Determine the set of binary weights whose sum is equal to the decimal number
  - $(9)_{10} = 8 + 1 = 2^3 + 2^0 = (1001)_2$
  - $(18)_{10} = 16 + 2 = 2^4 + 2^1 = (10010)_2$
  - $(58)_{10} = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1 = (111010)_2$
  - $(0.625)_{10} = 0.5 + 0.125 = 2^{-1} + 2^{-3} = (0.101)_2$

REPEATED DIVISION-BY-2

- To convert a whole number to binary, use successive division by 2 until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$(43)_{10} = (101011)_2$

2	43		
2	21	rem 1	← LSB
2	10	rem 1	
2	5	rem 0	
2	2	rem 1	
2	1	rem 0	
	0	rem 1	← MSB

REPEATED MULTIPLICATION-BY-2

- To convert decimal fractions to binary, repeated multiplication by 2 is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$(0.3125)_{10} = (.0101)_2$

	Carry	
$0.3125 \times 2 = 0.625$	0	←MSB
$0.625 \times 2 = 1.25$	1	
$0.25 \times 2 = 0.50$	0	
$0.5 \times 2 = 1.00$	1	←LSB

CONVERSION BETWEEN DECIMAL AND OTHER BASES

- Base-*R* to decimal: multiply digits with their corresponding weights.
- Decimal to binary (base 2)
  - Whole numbers repeated division-by-2
  - Fractions: repeated multiplication-by-2
- Decimal to base-*R*
  - Whole numbers: repeated division-by-*R*
  - Fractions: repeated multiplication-by-*R*

CONVERSION BETWEEN BASES

- In general, conversion between bases can be done via decimal:

$$\begin{tikzcd}
\text{Base-2} \arrow[rdd] & & \text{Base-2} \\
\text{Base-3} \arrow[rd] & & \text{Base-3} \\
\text{Base-4} \arrow[r] & \text{Decimal} \arrow[ruu] \arrow[ru] \arrow[r] \arrow[rdd] & \text{Base-4} \\
\cdots & & \cdots\cdot \\
\text{Base-}R \arrow[ruu] & & \text{Base-}R
\end{tikzcd}$$

- Shortcuts for conversion between bases 2, 4, 8, 16 (see next slide)

BINARY TO OCTAL/HEXADECIMAL CONVERSION

- Binary □ Octal: partition in groups of 3
  - $(10\ 111\ 011\ 001\ .\ 101\ 110)_2$ =
- Octal □ Binary: reverse
  - $(2731.56)_8$ =
- Binary □ Hexadecimal: partition in groups of 4
  - $(101\ 1101\ 1001\ .\ 1011\ 1000)_2$ =
- Hexadecimal □ Binary: reverse
  - $(5D9.B8)_{16}$ =

PEEKING AHEAD (1/2)

- Function simplification (eg: Quine-McCluskey)
- In ‘computer-speak’, units are in powers of 2
- Memory addressing (see next slide)

PEEKING AHEAD (2/2)

- Memory addressing
  - Assume 2^{10} bytes in memory, and each word contains 4 bytes.

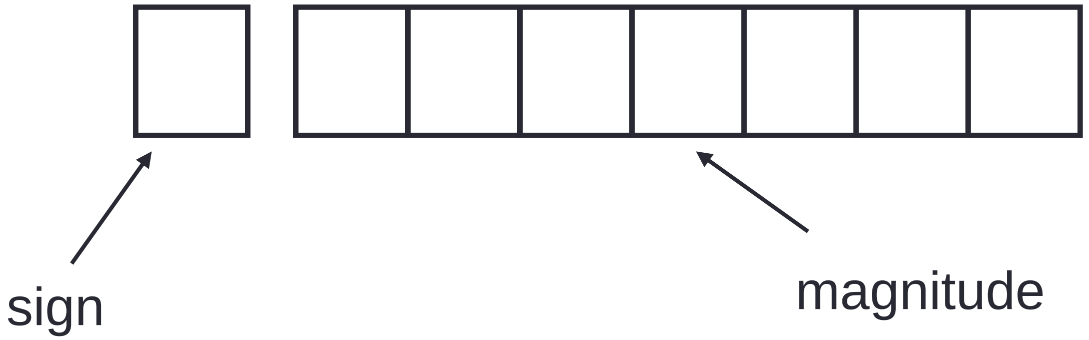
Addresses		Memory
binary	*decimal*	
0000000000	0	0 0 1 0 1 1 0 1
0000000001	1	0 1 0 1 0 1 0 1
0000000010	2	1 0 1 1 1 1 0 0
0000000011	3	0 1 1 1 1 0 0 1
0000000100	4	1 1 0 0 1 1 0 0
0000000101	5	1 0 0 0 0 1 0 1
0000000110	6	1 1 0 1 0 1 1 1
0000000111	7	0 0 0 1 1 0 0 0
0000001000	8	0 1 1 0 1 1 0 1
0000001001	9	1 0 0 1 1 0 1 1
0000001010	10	1 1 0 1 0 1 0 1
0000001011	11	0 1 0 0 0 0 0 1
.	.	●
.	.	●
1111111111	1023	●

NEGATIVE NUMBERS

- Unsigned numbers: only non-negative values.
- Signed numbers: include all values (positive and negative)
- There are 3 common representations for signed binary numbers:
  - Sign-and-Magnitude
  - 1s Complement
  - 2s Complement

SIGN-AND-MAGNITUDE (1/3)

- The sign is represented by a 'sign bit'
  - 0 for +
  - 1 for -
- Eg: a 1-bit sign and 7-bit magnitude format.



- 00110100 □ $+110100_2$ = ?
- 10010011 □ -10011_2 = ?

SIGN-AND-MAGNITUDE (2/3)

- Largest value: 01111111 = $+127_{10}$
- Smallest value: 11111111 = -127_{10}
- Zeros: 00000000 = $+0_{10}$
  10000000 = -0_{10}
- Range: -127_{10} to $+127_{10}$
- Question:
  - For an *n*-bit sign-and-magnitude representation, what is the range of values that can be represented?

SIGN-AND-MAGNITUDE (3/3)

- To negate a number, just invert the sign bit.
- Examples:
  - How to negate 00100001_{sm} (decimal 33)?
    Answer: 10100001_{sm} (decimal -33)
  - How to negate 10000101_{sm} (decimal -5)?
    Answer: 00000101_{sm} (decimal +5)

1s COMPLEMENT (1/3)

- Given a number ***x*** which can be expressed as an *n*-bit binary number, its negated value can be obtained in **1s-complement** representation using:

$$-x = 2^n - x - 1$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

$$\begin{aligned} -00001100_2 &= 2^8 - 12 - 1 \text{ (calculation in decimal)} \\ &= 243 \\ &= 11110011_{1s} \end{aligned}$$

(This means that -12_{10} is written as 11110011 in 1s-complement representation.)

1s COMPLEMENT (2/3)

- Essential technique to negate a value: invert all the bits.
- Largest value: 01111111 = $+127_{10}$
- Smallest value: 10000000 = -127_{10}
- Zeros: 00000000 = $+0_{10}$
  11111111 = -0_{10}
- Range: -127_{10} to $+127_{10}$
- The most significant (left-most) bit still represents the sign: 0 for positive; 1 for negative.

1s COMPLEMENT (3/3)

- Examples (assuming 8-bit numbers):

$(14)_{10} = (00001110)_2 = (00001110)_{1s}$

$-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

$-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{1s}$

2s COMPLEMENT (1/3)

- Given a number ***x*** which can be expressed as an *n*-bit binary number, its negated value can be obtained in **2s-complement** representation using:

$$-x = 2^n - x$$

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 2s-complement is:

$$\begin{aligned} -00001100_2 &= 2^8 - 12 \text{ (calculation in decimal)} \\ &= 244 \\ &= 11110100_{2s} \end{aligned}$$

(This means that -12_{10} is written as 11110100 in 2s-complement representation.)

2s COMPLEMENT (2/3)

- Essential technique to negate a value: invert all the bits, then add 1.
- Largest value: $01111111 = +127_{10}$
- Smallest value: $10000000 = -128_{10}$
- Zero: $00000000 = +0_{10}$
- Range: -128_{10} to $+127_{10}$
- The most significant (left-most) bit still represents the sign: 0 for positive; 1 for negative.

2s COMPLEMENT (3/3)

- Examples (assuming 8-bit numbers):

$(14)_{10} = (00001110)_2 = (00001110)_{2s}$

$-(14)_{10} = -(00001110)_2 = (11110010)_{2s}$

$-(80)_{10} = -(\ ?\ )_2 = (\ ?\ )_{2s}$

Compare with slide 30.

- 1s complement:

  $(14)_{10} = (00001110)_2 = (00001110)_{1s}$

  $-(14)_{10} = -(00001110)_2 = (11110001)_{1s}$

READING ASSIGNMENT

- Download from the course website and read the Supplement Notes on Lecture 2: Number Systems.
- Work out the exercises in there and discuss them in the IVLE forum if you have doubts.

COMPARISONS

Important!

4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

COMPLEMENT ON FRACTIONS

- We can extend the idea of complement on fractions.
- Examples:
  - Negate 0101.01 in 1s-complement
    Answer: 1010.10
  - Negate 111000.101 in 1s-complement
    Answer: 000111.010
  - Negate 0101.01 in 2s-complement
    Answer: 1010.11

2s COMPLEMENT ADDITION/SUBTRACTION (1/3)

- Algorithm for addition, A + B:

1. Perform binary addition on the two numbers.
2. Ignore the carry out of the MSB.
3. Check for overflow. Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.

- Algorithm for subtraction, A – B:

  A – B = A + (-B)

1. Take 2s-complement of B.
2. Add the 2s-complement of B to A.

OVERFLOW

- Signed numbers are of a fixed range.
- If the result of addition/subtraction goes beyond this range, an **overflow** occurs.
- Overflow can be easily detected:
  - *positive add positive* □ negative
  - *negative add negative* □ positive
- Example: 4-bit 2s-complement system
  - Range of value: -8_{10} to 7_{10}
  - $0101_{2s} + 0110_{2s} = 1011_{2s}$
    $5_{10} + 6_{10} = -5_{10}$?! (overflow!)
  - $1001_{2s} + 1101_{2s} = \underline{1}0110_{2s}$ (discard end-carry) = 0110_{2s}
    $-7_{10} + -3_{10} = 6_{10}$?! (overflow!)

2s COMPLEMENT ADDITION/SUBTRACTION (2/3)

- Examples: 4-bit system

```
   +3          0011
+  +4       +  0100
 ----       --------
   +7          0111
 ----       --------
```

```
   -2          1110
+  -6       +  1010
 ----       --------
   -8         11000
 ----       --------
```

```
   +6         0110
+  -3       + 1101
 ----       --------
   +3       10011
 ----       --------
```

```
   +4          0100
+  -7       +  1001
 ----       --------
   -3          1101
 ----       --------
```

- Which of the above is/are overflow(s)?

2s COMPLEMENT ADDITION/SUBTRACTION (3/3)

- Examples: 4-bit system

```
   -3           1101
+ -6         + 1010
----         -------
   -9          10111
----         -------
```

```
   +5           0101
+ +6         + 0110
----         -------
  +11           1011
----         -------
```

- Which of the above is/are overflow(s)?

1s COMPLEMENT ADDITION/SUBTRACTION (1/2)

- **Algorithm for addition, A + B:**
  1. Perform binary addition on the two numbers.
  2. If there is a carry out of the MSB, add 1 to the result.
  3. Check for overflow. Overflow occurs if result is opposite sign of A and B.
- **Algorithm for subtraction, A – B:**

  A – B = A + (-B)
  1. Take 1s-complement of B.
  2. Add the 1s-complement of B to A.

1s COMPLEMENT ADDITION/SUBTRACTION (2/2)

- Examples: 4-bit system

Any overflow?

```
   +3          0011
+ +4         + 0100
----         --------
   +7          0111
----         --------
```

```
   +5          0101
+ -5         + 1010
----         --------
   -0          1111
----         --------
```

```
   -2          1101
+ -5     +    1010
----         -------
   -7         10111
----        +     1
             -------
               1000
```

```
   -3          1100
+ -7         + 1000
----         --------
  -10         10100
----         +     1
             --------
               0101
```

QUICK REVIEW QUESTIONS (4)

- DLD pages 42 - 43
  Questions 2-13 to 2-18.

EXCESS REPRESENTATION (1/2)

- Besides sign-and-magnitude and complement schemes, the **excess representation** is another scheme.
- It allows the range of values to be distributed evenly between the positive and negative values, by a simple translation (addition/subtraction).
- Example: Excess-4 representation on 3-bit numbers. See table on the right.

- Questions: What if we use Excess-2 on 3-bit numbers? Excess-7?

Excess-4 Representation	*Value*
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

EXCESS REPRESENTATION (2/2)

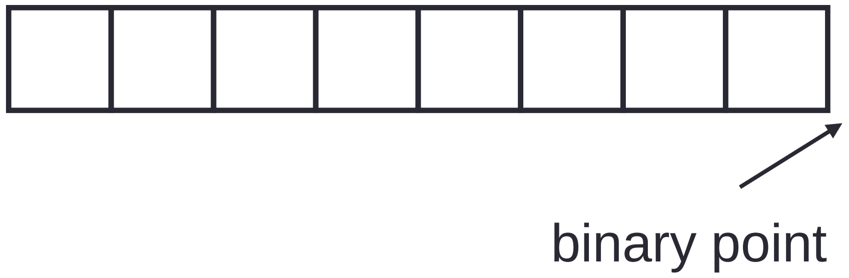
- Example: For 4-bit numbers, we may use excess-7 or excess-8. Excess-8 is shown below. Fill in the values.

Excess-8 Representation	*Value*
0000	-8
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Excess-8 Representation	*Value*
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

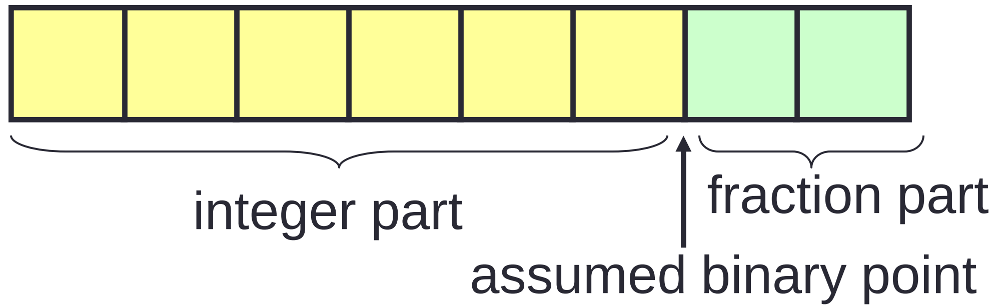
FIXED POINT NUMBERS (1/2)

- In fixed point representation, the binary point is assumed to be at a fixed location.
  - For example, if the binary point is at the end of an 8-bit representation as shown below, it can represent integers from -128 to +127.



FIXED POINT NUMBERS (2/2)

- In general, the binary point may be assumed to be at any pre-fixed location.
  - Example: Two fractional bits are assumed as shown below.



- If 2s complement is used, we can represent values like:

  $011010.11_{2s} = 26.75_{10}$

  $111110.11_{2s} = -000001.01_{2} = -1.25_{10}$

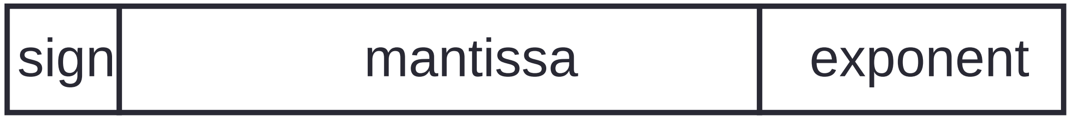
FLOATING POINT NUMBERS (1/4)

- Fixed point numbers have limited range.
- Floating point numbers allow us to represent very large or very small numbers.
- Examples:

  0.23×10^{23} (very large positive number)
  0.5×10^{-37} (very small positive number)
  -0.2397×10^{-18} (very small negative number)

FLOATING POINT NUMBERS (2/4)

- 3 parts: **sign**, **mantissa** and **exponent**
- The base (radix) is assumed to be 2.
- Sign bit: 0 for positive, 1 for negative.



- Mantissa is usually in **normalised** form (the integer part is zero and the fraction part must not begin with zero)
  - 0.01101×2^4 □ normalised □
  - $101011.0110 \times 2^{-4}$ □ normalised □
- Trade-off:
  - More bits in mantissa □ better precision
  - More bits in exponent □ larger range of values

FLOATING POINT NUMBERS (3/4)

- Exponent is usually expressed in complement or excess format.
- Example: Express -6.5_{10} in base-2 normalised form

  $-6.5_{10} = -110.1_2 = -0.1101_2 \times 2^3$
- Assuming that the floating-point representation contains 1-bit, 5-bit normalised mantissa, and 4-bit exponent. The above example will be stored as if the exponent is in 1s or 2s complement.

1	11010	0011

FLOATING POINT NUMBERS (4/4)

- Example: Express 0.1875_{10} in base-2 normalised form

  $0.1875_{10} = 0.0011_2 = 0.11 \times 2^{-2}$

- Assume this floating-point representation:1-bit sign, 5-bit normalised mantissa, and 4-bit exponent.

- The above example will be represented as

0	11000	1101	If exponent is in 1's complement.
0	11000	1110	If exponent is in 2's complement.
0	11000	0110	If exponent is in excess-8.

DECIMAL CODES

- Decimal numbers are favoured by humans. Binary numbers are natural to computers. Hence, conversion is required.
- If little calculation is required, we can use some **coding schemes** to store decimal numbers, for data transmission purposes.
- Examples: BCD (or 8421), Excess-3, 84-2-1, 2421, etc.
- Each decimal digit is represented as a 4-bit code.
- The number of digits in a code is also called the length of the code.

BINARY CODE DECIMAL (BCD) (1/2)

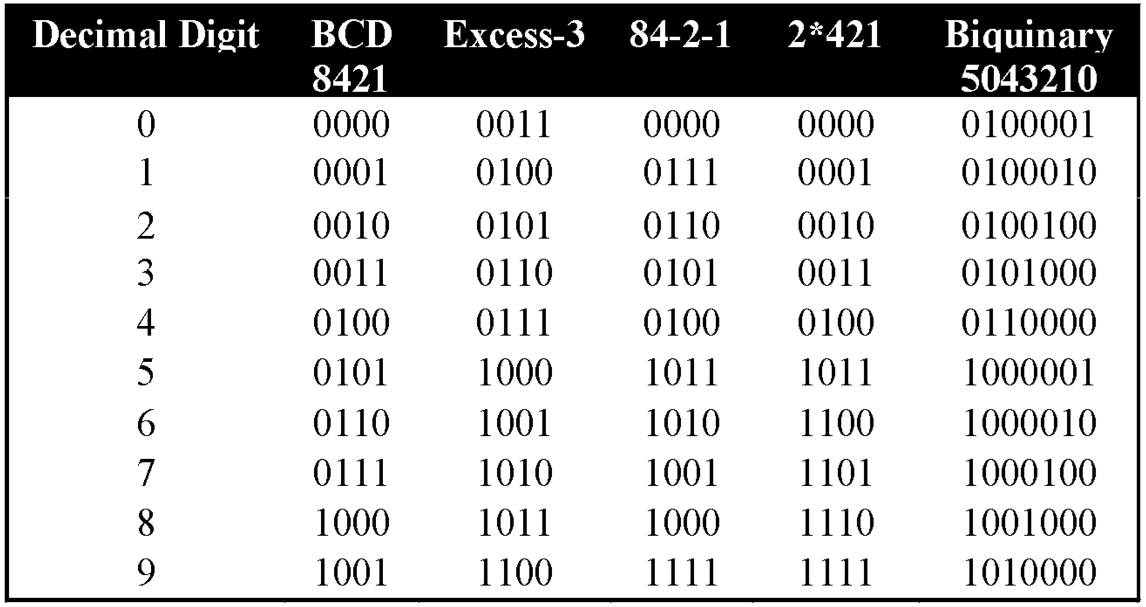
- Some codes are unused, like 1010_{BCD}, 1011_{BCD}, … 1111_{BCD}. These codes are considered as errors.
- Easy to convert, but arithmetic operations are more complicated.
- Suitable for interfaces such as keypad inputs.

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

BINARY CODE DECIMAL (BCD) (2/2)

- Examples of conversion between BCD values and decimal values:
  - $(234)_{10} = (0010\ 0011\ 0100)_{BCD}$
  - $(7093)_{10} = (0111\ 0000\ 1001\ 0011)_{BCD}$
  - $(1000\ 0110)_{BCD} = (86)_{10}$
  - $(1001\ 0100\ 0111\ 0010)_{BCD} = (9472)_{10}$
- Note that BCD is <u>not equivalent</u> to binary
  - Example: $(234)_{10} = (11101010)_2$

OTHER DECIMAL CODES



Decimal Digit	BCD 8421	Excess-3	84-2-1	2*421	Biquinary 5043210
0	0000	0011	0000	0000	0100001
1	0001	0100	0111	0001	0100010
2	0010	0101	0110	0010	0100100
3	0011	0110	0101	0011	0101000
4	0100	0111	0100	0100	0110000
5	0101	1000	1011	1011	1000001
6	0110	1001	1010	1100	1000010
7	0111	1010	1001	1101	1000100
8	1000	1011	1000	1110	1001000
9	1001	1100	1111	1111	1010000

- Self-complementing code: codes for complementary digits are also complementary to each other.
- Error-detecting code: biquinary code (*bi*=two, *quinary*=five).

SELF-COMPLEMENTING CODES

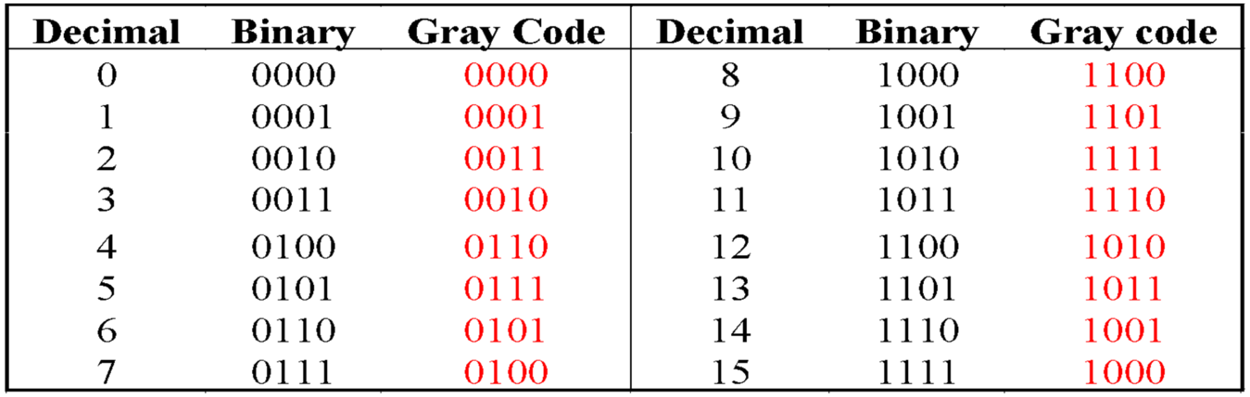
- The codes representing the pair of complementary digits are also complementary to each other.
- Example: Excess-3 code

```
0: 0011
1: 0100
2: 0101
3: 0110
4: 0111
5: 1000
6: 1001
7: 1010
8: 1011
9: 1100
```

- Question: What are the other self-complementing codes?

GRAY CODE (1/3)

- Unweighted (not an arithmetic code)
- Only a single bit change from one code value to the next.
- Not restricted to decimal digits: *n* bits □ 2^n values.
- Good for error detection.
- Example: 4-bit standard 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

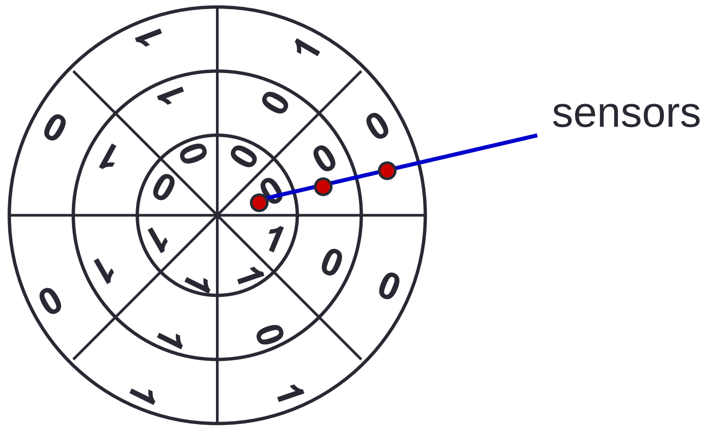
GRAY CODE (2/3)

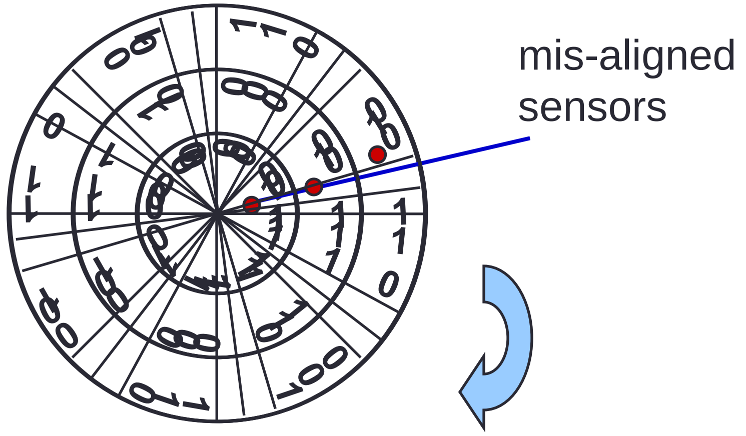
- Generating a 4-bit standard Gray code sequence.

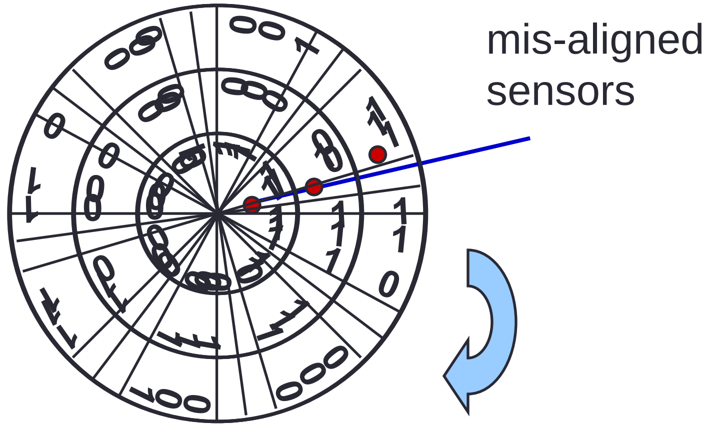
0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

- Questions: How to generate 5-bit standard Gray code sequence? 6-bit standard Gray code sequence?

GRAY CODE (3/3)







Binary coded: 111 → 110 → 000

Gray coded: 111 → 101

ALPHANUMERIC CODES (1/3)

- Computers also handle textual data.
- Character set frequently used:

alphabets:	‘A’ … ‘Z’, ‘a’ … ‘z’
digits:	‘0’ … ‘9’
special symbols:	‘$’, ‘.’, ‘@’, ‘*’, etc.
non-printable:	NULL, BELL, CR, etc.

- Examples
  - ASCII (8 bits), Unicode

ALPHANUMERIC CODES (2/3)

- ASCII
  - American Standard Code for Information Interchange
  - 7 bits, plus a parity bit for error detection
  - Odd or even parity

Character	ASCII Code
0	0110000
1	0110001
. . .	. . .
9	0111001
:	0111010
A	1000001
B	1000010
. . .	. . .
Z	1011010
[	1011011
\	1011100

ALPHANUMERIC CODES (3/3)

- ASCII table

A: 1000001

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC$_1$	!	1	A	Q	a	q
0010	STX	DC$_2$	“	2	B	R	b	r
0011	ETX	DC$_3$	#	3	C	S	c	s
0100	EOT	DC$_4$	$	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	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

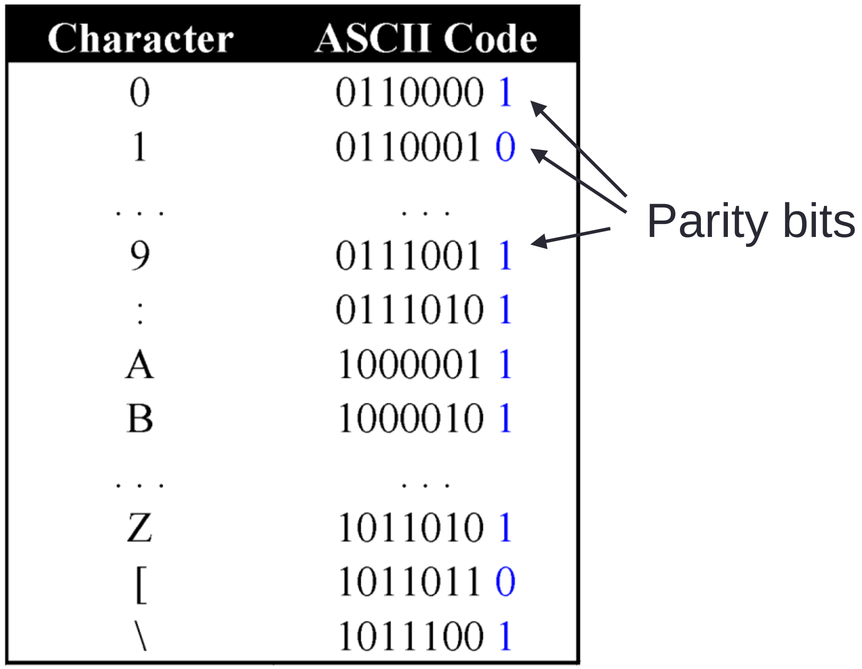
ERROR DETECTION (1/4)

- Errors can occur during data transmission. They should be detected, so that re-transmission can be requested.
- With binary numbers, usually single-bit errors occur.
  - Example: 0010 erroneously transmitted as 001<u>1</u> or 00<u>0</u>0 or 0<u>1</u>10 or <u>1</u>010.
- Biquinary code has length 7; it uses 3 additional bits for error-detection.

Decimal digit	Biquinary 5043210
0	0100001
1	0100010
2	0100100
3	0101000
4	0110000
5	1000001
6	1000010
7	1000100
8	1001000
9	1010000

ERROR DETECTION (2/4)

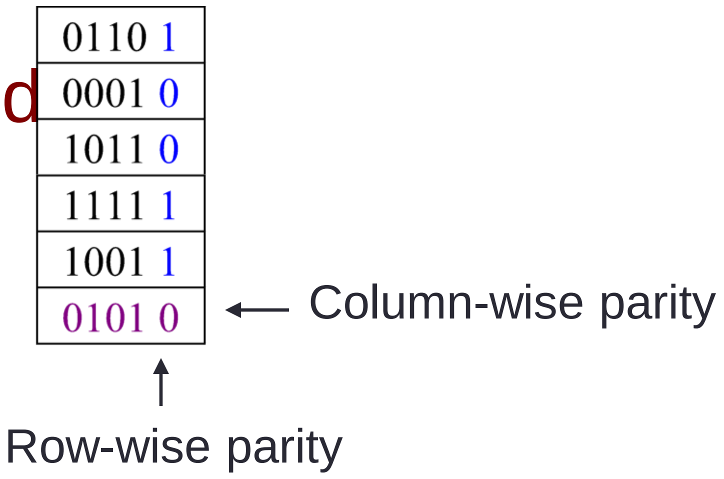
- Parity bit
  - Even parity: additional bit added to make total number of 1's even.
  - Odd parity: additional bit added to make total number of 1's odd.
- Example of odd parity on ASCII values.


Parity bits

Character	ASCII Code
0	0110000 1
1	0110001 0
. . .	. . .
9	0111001 1
:	0111010 1
A	1000001 1
B	1000010 1
. . .	. . .
Z	1011010 1
[	1011011 0
\	1011100 1

ERROR DETECTION (3/4)

- Parity bit can detect odd number of errors but not even number of errors.
  - Example: Assume odd parity,
    - 10011 □ 10001 (detected)
    - 10011 □ 10101 (not detected)
- Parity bits can also be applied to a block of data.



ERROR DETECTION (4/4)

- Sometimes, it is not enough to do error detection. We may want to correct the errors.
- Error correction is expensive. In practice, we may use only single-bit error correction.
- Popular technique: Hamming code

ERROR CORRECTION (1/7)

- Given this 3-bit code C_1

  { 000, 110, 011, 101 }
- With 4 code words, we actually need only 2 bits.
  - We call this *k*, the number of original message bits.
- To add error detection/correction ability, we use more bits than necessary.
  - In this case, the length of each codeword is 3
- We define code efficiency (or rate) by

  k / length of codeword
- Hence, efficiency of C_1 is 2/3.

ERROR CORRECTION (2/7)

- Given this 3-bit code C_1

  { 000, 110, 011, 101 }
- Can C_1 detect a single bit error?
- Can C_1 correct a single bit error?

Sometimes, we use “1 error” for “single bit error”, “2 errors” for “2 bits error”, etc.

ERROR CORRECTION (3/7)

- The distance *d* between any two code words in a code is the sum of the number of differences between the codewords.
  - Example: *d*(000, 110) = 2; *d*(0110,1011) = 3.
- The Hamming distance *δ* of a code is the minimum distance between any two code words in the code.
  - Example: The Hamming distance of C_1 is 2.
- A code with Hamming distance of 2 can detect 1 error.

ERROR CORRECTION (4/7)

- Given this 6-bit code C_2

  { 000000, 111000, 001110, 110011 }
- What is its efficiency?
- What is its Hamming distance?
- Can it correct 1 error?

- Can it correct 2 errors?

Self-study

ERROR CORRECTION (5/7)

- Hamming code: a popular error-correction code
- Procedure
  - Parity bits are at positions that are powers of 2 (i.e. 1, 2, 4, 8, 16, …)
  - All other positions are data bits
  - Each parity bit checks some of the data bits
    - Position 1: Check 1 bit, skip 1 bit (1, 3, 5, 7, 9, 11, …)
    - Position 2: Check 2 bits, skip 2 bits (2, 3, 6, 7, 10, 11, …)
    - Position 4: Check 4 bits, skip 4 bits (4-7, 12-15, 20-23, …)
    - Position 8: Check 8 bits, skip 8 bits (8-15, 24-31, 40-47, …)
  - Set the parity bit accordingly so that total number of 1s in the positions it checks is even.

Self-study

ERROR CORRECTION (6/7)

- Example: Data 10011010
- Insert positions for parity bits:

  _ _ 1 _ 0 0 1 _ 1 0 1 0

  - Position 1: ? _ 1 _ 0 0 1 _ 1 0 1 0 □ so ? must be 0
  - Position 2: 0 ? 1 _ 0 0 1 _ 1 0 1 0 □ so ? must be 1
  - Position 4: 0 1 1 ? 0 0 1 _ 1 0 1 0 □ so ? must be 1
  - Position 8: 0 1 1 1 0 0 1 ? 1 0 1 0 □ so ? must be 0

Answer: 0 1 1 1 0 0 1 0 1 0 1 0

Self-study

ERROR CORRECTION (7/7)

- Suppose 1 error occurred and the received data is:

  0 1 1 1 0 0 1 0 1 $\boxed{1}$ 1 0

- How to determine which bit is in error?
- Check which parity bits are in error.
  - Answer: parity bits 2 and 8.
- Add the positions of these erroneous parity bits
  - Answer: 2 + 8 = 10. Hence data bit 10 is in error.

Corrected data: 0 1 1 1 0 0 1 0 1 0 1 0

READING ASSIGNMENT

- Conversion between standard Gray code and binary
  - DLD page 1-24.

Q&A