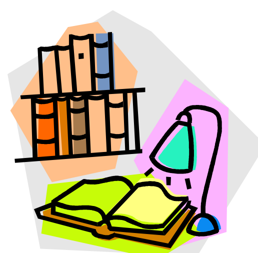
Overview: Number Systems & Codes

* Introduction
* Number Systems
* Binary and Hex Arithmetic
* Negative Numbers & Floating Point
* Codes: Binary, Decimal, Alphanumeric

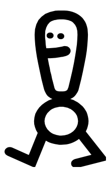


Chapters 1 & 2 – "Digital Design: Principles and Practices" book

Analog *versus* Digital

- **Analog devices** process time-varying signals that can have any value across a continuous range *and* produce results that are also in continuous form.
  - **Examples** of continuous signals: voltage, current, force.

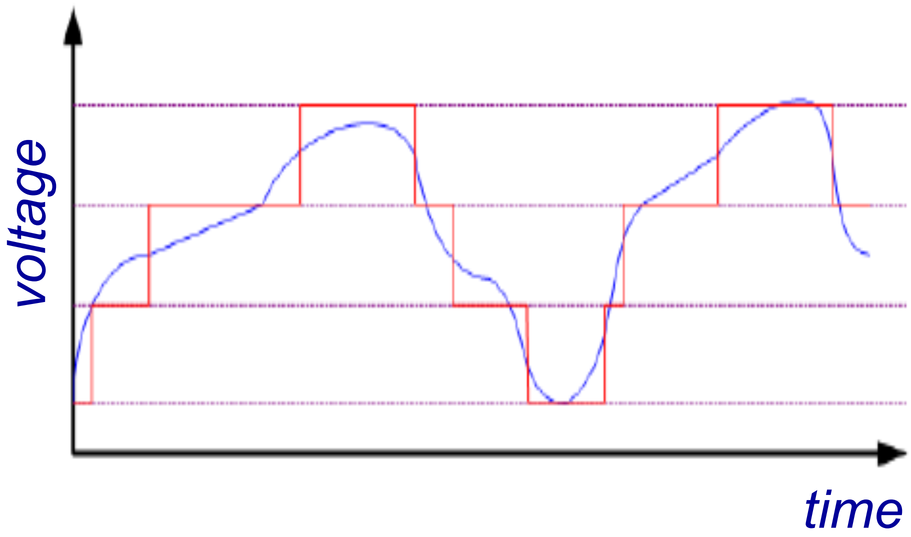
- **Digital devices** process signals that take on only two discrete values (such as *0* and *1*) *and* produce output that can be represented by *0* and *1*.
  - **Examples** of digital devices: CDs, DVDs.



Digital circuits/systems also process time-varying signals!

Going Digital

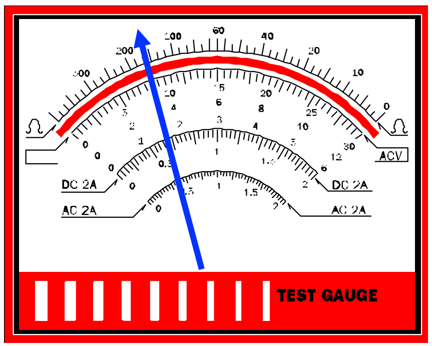
- The world around us is analog but digital systems are simple to understand and use …
  - Common practice is to convert analog signals into digital form for efficient processing of signals.
  - However, it is not possible to avoid loss of some accuracy (information) due to this conversion, because digital systems can only represent fixed (finite or discrete) sets of values.



Digital Data: Advantages

- **Analog** has ambiguity; **Digital** has only one interpretation.

Analog Voltage meter

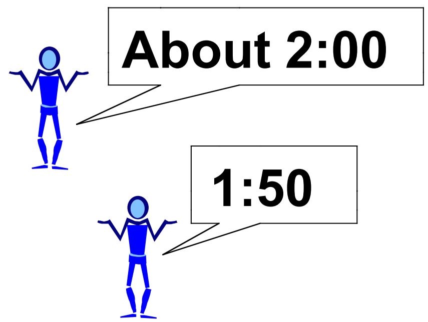


About 100

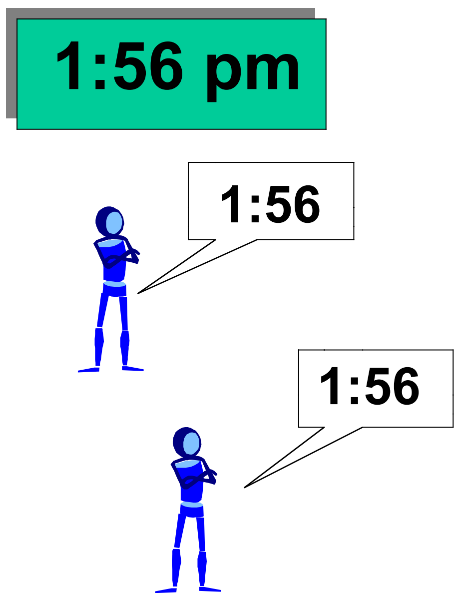
Digital Voltage meter

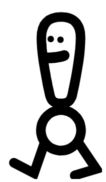
103.5

Analog Clock



Digital Clock

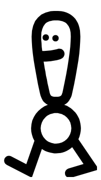




No ambiguous information on the digital clock and digital voltage meter!

Digital (instead of Analog) Circuits: Why?

- Reproducing Results: analog circuit outputs vary with temperature, power-supply voltage, …
- Flexibility and Functionality: problem in digital form can be solved using a set of logical steps.
- Programmability: use of HDL and software tools.
- Speed: digital devices can produce results very quickly.
- Economy: mass-production made possible; this means putting a lot of functionality in a small place (the IC).



Much of today's digital design is done by writing programs in HDLs.

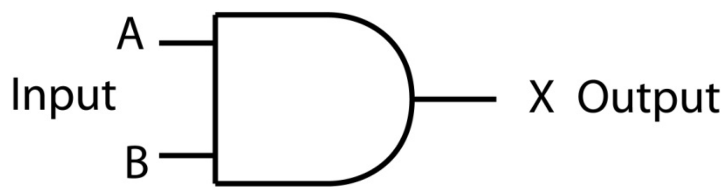
Binary Representation

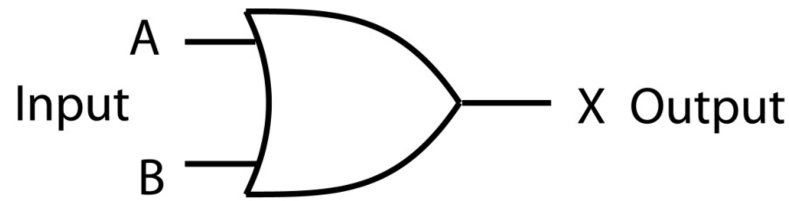
- Basis of all digital data is binary representation.
- Binary → means 'two'

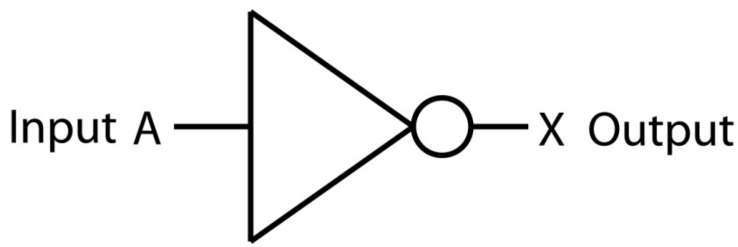
  1, 0 // True, False // Hot, Cold // On, Off
- Computers (digital systems) represent data in the binary system using:
  - Electrical voltages (e.g., in processors, memory);
  - Magnetism (e.g., in hard disks, floppy);
  - Light (e.g., in CD, DVD).
- However, we *must be able to handle more than just two values* for real world problems.
  - 1, 0, 56
  - True, False, Maybe
  - On, Off, Leaky
  - Hot, Cold, Warm, Cool

Gates

- **Gate**: most fundamental building block of a digital device or system.
  - A digital system (a chip) consists of many, many gates. They have *one or more digital inputs* and *one digital output*.
  - Gates are digital devices that perform various basic logic operations.
- **Basic gate types** are:
  - AND gate
  - OR gate
  - NOT gate







Digital Abstraction

- Digital circuits are built with analog components *and* deal with analog voltages and currents.
- **Digital abstraction** allows analog behaviour to be ignored by associating a *range* of voltages with each logic value:
  - **Examples**:
    - signals in a digital system may be restricted to two levels -5 and + 5 volts, corresponding to two discrete values of *0* and *1*.
    - *high* and *low* are often
      used to represent *1* and *0*
      when discussing electronic logic.

voltage	binary number	logic
+ 5 volts	1	true
– 5 volts	0	false

Queen Mary
University of London

Integrated Circuits

- **Integrated Circuit** (**IC**): A collection of one or more gates fabricated on a single silicon chip to achieve a specific function.
  - ICs usually consist of "legs", referred to as *pins* or *DIPs*.
  - Pins are input/output connectors; their functionality can be obtained from the pin diagram or data sheet.
  - In educational labs, DIPs are usually packaged with *14 pins*.

Dual-In-line-Pin

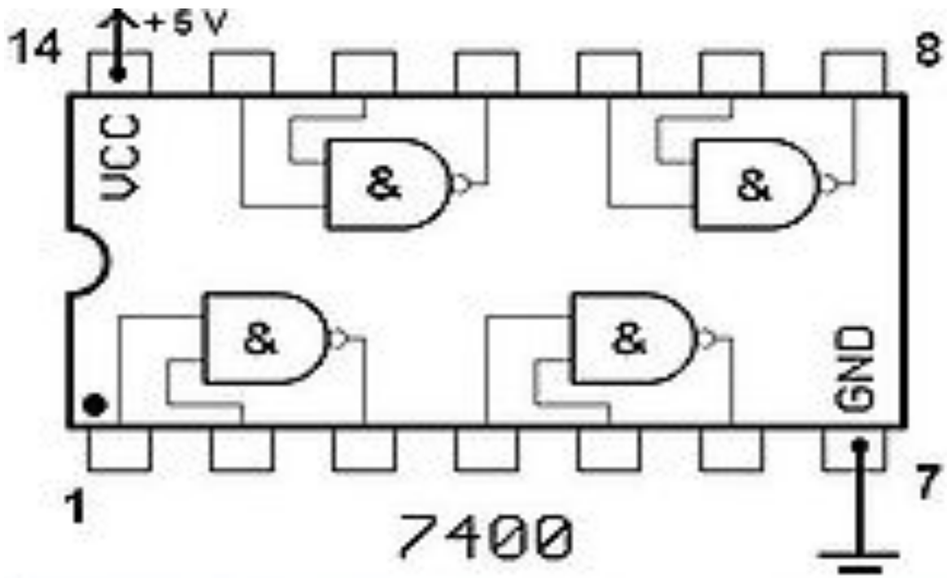
Classification of ICs based on size (i.e., number of gates)

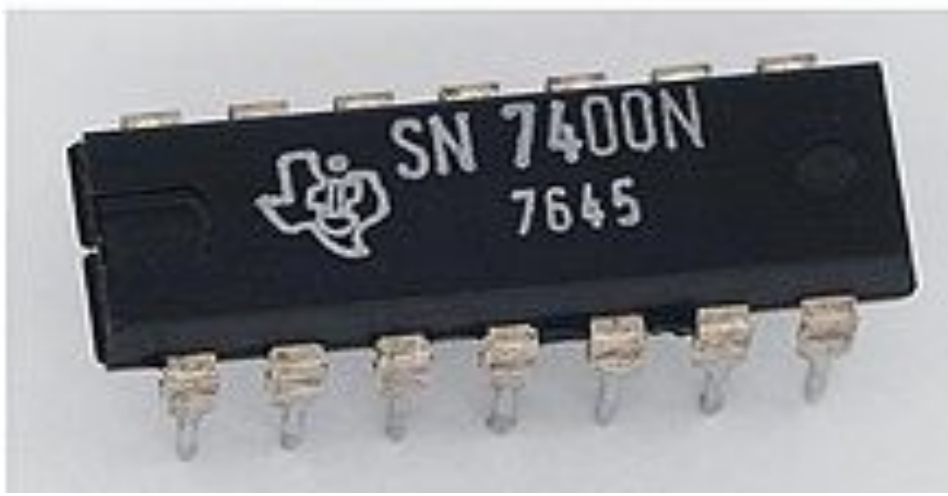
Name	Number of Gates
Small-Scale Integration (**SSI**)	< 20
Medium-Scale Integration (**MSI**)	20 – 200
Large-Scale Integration (**LSI**)	200 – 200000
Very Large-Scale Integration (**VLSI**)	≈ 1 million **transistors**

measure used for VLSIs

Queen Mary
University of London

From Wikipedia: the 7400 Series



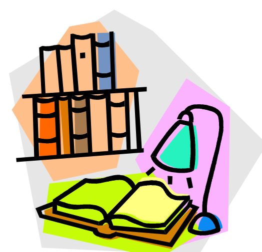


Software for Digital Design

- Software is widely used in digital design.
  It can reduce design time, design cost, and improve design quality.
- It has been mainly used for:
  - drawing schematic diagrams;
  - circuit simulation and modelling;
  - testing and debugging;
  - timing analysis.
- **Example**:
  - VHDL software package

Overview: Number Systems & Codes

- **Introduction**
- **Number Systems**
- **Binary and Hex Arithmetic**
- **Negative Numbers & Floating Point**
- **Codes: Binary, Decimal, Alphanumeric**

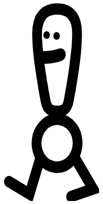


Chapters 1 & 2 – “Digital Design: Principles and Practices” book

Queen Mary
University of London

Positional Number Systems

- In positional number systems, each number is represented by a string of digits; each digit position has an associated weight.
- The value of the number is equal to the weighted sum of all digits, with the weights determined by the digit's position and the base (or radix) of the numbering system.
- **Examples**:
  - Decimal (base 10) → most common (should be familiar!)
    - Digits in base 10 range from 0 to 9.
    - $321_{10} = (3 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$
    - The subscript in 321_{10} is the base.



If no subscript is shown, assume a decimal number.

Queen Mary University of London

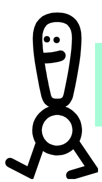
Number Systems: At a Glance

- Table shows *the first 17 numbers of each of the most common number systems*.

Decimal	Binary	8-bit Binary	Octal	Hexadecimal
0	0	00000000	0	0
1	1	00000001	1	1
2	10	00000010	2	2
3	11	00000011	3	3
4	100	00000100	4	4
5	101	00000101	5	5
6	110	00000110	6	6
7	111	00000111	7	7
8	1000	00001000	10	8
9	1001	00001001	11	9
10	1010	00001010	12	A
11	1011	00001011	13	B
12	1100	00001100	14	C
13	1101	00001101	15	D
14	1110	00001110	16	E
15	1111	00001111	17	F
16	10000	00010000	20	10

Positional Number Systems – Again! (1/2)

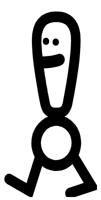
- **More examples**:
  - **Binary** (base 2) → preferred system for electronic IC
    - Digits in base 2 range from 0 to 1 (also called *binary number system*).
    - A digit in base 2 is also called a 'bit'.
    - Numbers tend to be long, e.g.: $321_{10} = 101000001_2$
  - **Octal** (base 8) → seen in older computers and electronics
    - Digits in base 8 range from 0 to 7.
    - Easy to convert to/from binary, e.g.: $(101)(000)(001)_2 = 501_8$



Base 2 is used to represent numbers in a digital system.

Positional Number Systems – Again! (2/2)

- **More examples**:
  - **Hexadecimal** (base 16) → most common display of binary information
    - Uses letters A – F to represent additional digits (i.e., values 10 to 15).
    - Digits in base 16 range from 0 to 16-1, i.e., values in the set { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }.
    - Base 16 is also called Hexadecimal or just "Hex".
  - **Other bases**: Digits in base ***R*** range from **0** to **(*R* – 1)**.



There are also **non-positional number systems** e.g., *Roman numerals*: I, II, III, IV, V,…. Only really useful for year dating!

Least & Most Significant Digits

$$53_{10} = 110101_2$$

Most Significant Digit
(has weight of 2^5=32).

For base 2, also called
Most Significant Bit (MSB).

Always LEFTMOST digit.

Least Significant Digit
(has weight of 2^0=1).

For base 2, also called
Least Significant Bit (LSB).

Always RIGHTMOST digit.

Queen Mary
University of London

Positional Notation

- **Value of the number** is determined by multiplying each digit by a weight and then summing.
- The **weight** of each digit is a POWER of the BASE and is determined by its position.
- **Examples**:

$$
\begin{aligned}
11_2 &= (1\text{x}2^1) + (1\text{x}2^0) \\
&= 2 + 1 \\
&= 3_{10}
\end{aligned}
$$

$$
\begin{aligned}
1011.11_2 &= (1\text{x}2^3)+(0\text{x}2^2)+(1\text{x}2^1)+(1\text{x}2^0)+(1\text{x}2^{-1})+(1\text{x}2^{-2}) \\
&= 8 + 0 + 2 + 1 + 0.5 + 0.25 \\
&= 11.75_{10}
\end{aligned}
$$

Queen Mary
University of London

Conversion: Any Base to Decimal

- Converting from ANY base to decimal is done by multiplying each digit by its weight and summing.
  - **Binary to Decimal**

$$1011.01_2 = (1x2^3)+(0x2^2)+(1x2^1)+(1x2^0)+(0x2^{-1})+(1x2^{-2})$$
$$= 8 + 0 + 2 + 1 + 0 + 0.25$$
$$= 11.25_{10}$$

  - **Hex to Decimal**

$$A2F_{16} = (10x16^2) + (2x16^1) + (15x16^0)$$
$$= (10x256) + (2x16) + (15x1)$$
$$= 2560 + 32 + 15$$
$$= 2607_{10}$$

Queen Mary University of London

Conversion: Decimal to Any Base

- Divide number ***N*** by base ***R*** until quotient is **0**. The *remainder* at EACH step is a digit in base ***R***, from Least Significant Digit to Most Significant Digit.
  - **Example**: Convert 53_{10} to binary.

```
53 ÷ 2 = 26   r= 1 (LSB)
26 ÷ 2 = 13   r= 0
13 ÷ 2 = 6    r= 1
 6 ÷ 2 = 3    r= 0
 3 ÷ 2 = 1    r= 1
 1 ÷ 2 = 0    r= 1 (MSB)
```

Quotient = 0 so stop.

Answer: $53_{10} = 110101_2$

Check your work:

110101_2

$= 1x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0$

$= 32 + 16 + 0 + 4 + 0 + 1$

$= 53$

Queen Mary University of London

Conversion: Non-Integer to Any Base

- Conversion requires 2 steps:
  - **A**: divide the integer part of number ***N*** by base ***R*** until quotient is **0**; *remainder* at EACH step is a digit in base ***R***, from LSD to MSD.
  - **B**: multiply the number **N**'s decimal part by base ***R***; the new number's integer part will be a new digit in base ***R*** of the number's fractional part, from MSD to LSD.
- **Example**: 5.2_{10} to binary (with up to *3 decimal places*)

Integer Part	Fractional Part
5 ÷ 2 = 2 r=**1** **(LSD)**	0.2 x 2 = **0**.4 **(MSD)**
2 ÷ 2 = 1 r=**0**	0.4 x 2 = **0**.8
1 ÷ 2 = 0 r=**1** **(MSD)**	0.8 x 2 = **1**.6
	...

$\boxed{\textbf{Answer: } 5.2_{10} \cong 101.001_2}$

Example: Non-integer to Binary Conversion

- Convert 225.225_{10} to binary.

Integral Part	*Fractional Part*
225 ÷ 2 = 112 r = 1 (LSD)	0.225 x 2 = 0.45 (MSD)
112 ÷ 2 = 56 r = 0	0.45 x 2 = 0.9
56 ÷ 2 = 28 r = 0	0.9 x 2 = 1.8
28 ÷ 2 = 14 r = 0	0.8 x 2 = 1.6
14 ÷ 2 = 7 r = 0	0.6 x 2 = 1.2
7 ÷ 2 = 3 r = 1	0.2 x 2 = 0.4
3 ÷ 2 = 1 r = 1	0.4 x 2 = 0.8
1 ÷ 2 = 0 r = 1 (MSD)	0.8 x 2 = 1.6
0	...

$$225.225_{10} = 11100001.00111001_2$$

Queen Mary University of London

Example: Decimal to Hex Conversion

- Convert 53_{10} to hexadecimal.

```
53  ÷  16  = 3      r = 5   (LSD)
 3  ÷  16  = 0      r = 3   (MSD)
 0
```

Answer: $\mathbf{53_{10} = 35_{16}}$

Check your work:

Example: Non-Integer to Hex Conversion

- Convert 265.78_{10} to hexadecimal.

Integer part		*Fractional part*
265 ÷ 16 = 16	r = 9 (LSD)	0.78 x 16 = 12.48 (MSD)
16 ÷ 16 = 1	r = 0	0.48 x 16 = 7.68
1 ÷ 16 = 0	r = 1 (MSD)	0.68 x 16 = 10.88
0		0.88 x 16 = 14.08 etc.

Answer:

Queen Mary
University of London

Conversion: Hex (base 16) to Binary

- Each Hex digit represents 4 bits.
- **To convert a Hex number to Binary**, simply convert each Hex digit to its 4-bit value.
- *Hex digits to Binary*:

```
0 = 0000        8 = 1000
1 = 0001        9 = 1001
2 = 0010        A = 1010
3 = 0011        B = 1011
4 = 0100        C = 1100
5 = 0101        D = 1101
6 = 0110        E = 1110
7 = 0111        F = 1111
```

Queen Mary
University of London

Hex to Binary *vs.* Binary to Hex

- ***Hex to Binary***: Convert each Hex digit to binary.
  - *Examples*:

$$A2F_{16} = 1010\ 0010\ 1111_2$$

$$345_{16} = 0011\ 0100\ 0101_2$$

- ***Binary to Hex*** is just the opposite:
  - Create groups of 4 bits, starting with least significant bits.
  - If the last group may not have 4 bits, then pad with zeros. (note: only correct for unsigned numbers, we are going to talk about this soon)
  - *Example*:

$$1010001_2 = 0101\ 0001_2 = 51_{16}$$

Padded with a zero.

Trick: Conversion to Decimal

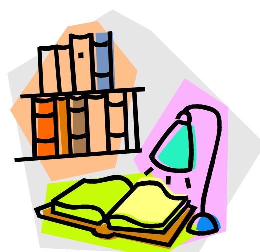
- If faced with a large binary number that has to be converted to decimal:
  - first convert the Binary number to Hex;
  - then convert the Hex to Decimal.
- ***Example***:

$$
\begin{aligned}
110111110011_2 &= 1101 \quad 1111 \quad 0011 \\
&= \;\; D \qquad F \qquad 3 \\
&= (13 \times 16^2) + (15 \times 16^1) + (3 \times 16^0) \\
&= (13 \times 256) + (15 \times 16) + (3 \times 1) \\
&= 3328 + 240 + 3 \\
&= 3571_{10}
\end{aligned}
$$

Queen Mary University of London

Overview: Number Systems & Codes

* Introduction
* Number Systems
* Binary and Hex Arithmetic
* Negative Numbers & Floating Point
* Codes: Binary, Decimal, Alphanumeric



Chapters 1 & 2 – "Digital Design: Principles and Practices" book

Queen Mary
University of London

Addition and Subtraction

- To learn how to add and subtract in other bases we will first revisit it in decimal!

Addition

Carries

```
 00        01
  10        11
 +09       +09
 019       020
```

Subtraction

Borrows

```
 00   10 + 0    03   10 + 4
  10  = 10       44  = 14
 -04            -09
 006            035
```

Adding and Subtracting: Rules

- *Rules for other bases* are just the same as for decimal numbers:
  - **Carry** to next place when you exceed base!
  - **Borrow** the base amount when you need to subtract!
- So, in *binary*:

Addition

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = 0 + (carry 1)
1 + 1 + 1 = 1 + (carry 1)
```

Subtraction

```
1 - 0 = 1
1 - 1 = 0
(borrow 2) 0 - 1 = 1
```

Queen Mary University of London

Examples: Binary Addition & Subtraction

1 1 1 1 0 0 0 *carries* 1 1 1 1 1 1 1 1 1 1

1 0 1 1 1 0 1
+ 1 1 1 0 1 0
1 0 0 1 0 1 1 1

1 0 1 0 1 0 1 1 1 1
+ 1 1 1 1 1 1 1 1 1 1
1 1 0 1 0 1 0 1 1 1 0

0 2 0 2 *borrows* 0 2

~~1~~ ~~0~~ 1 1 ~~1~~ ~~0~~ 1
- 1 1 1 0 1 0
1 0 0 0 1 1

1 0 1 0 ~~1~~ ~~0~~ 1 1 1 1
- 1 0 0 0 0 1 1 1 1 1
0 0 1 0 0 1 0 0 0 0

Adding and Subtracting in Hex

- **Rules** are the same as for decimal numbers:
  - Carry to next place when result of operation (i.e., addition or subtraction) equals base, only the base is now *16*.
- **Examples**:

```
carries →      1  1
           A  7  3  4  8  6  5
           +  1  2  5  9  D  6
          ---------------------
           A  8  5  A  2  3  B
                          (11)
```

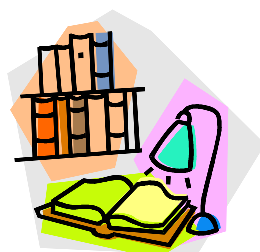
```
                   17
borrows →       9  1  19
           A  8  5  A  2  3  B
           -  1  2  5  9  D  6
          ---------------------
           A  7  3  4  8  6  5
```

***Remember**: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15*

Overview: Number Systems & Codes

* **Introduction**
* **Number Systems**
* **Binary and Hex Arithmetic**
* **Negative Numbers & Floating Point**
* **Codes: Binary, Decimal, Alphanumeric**

Section 10.11 – "Digital Design from Zero to One" book



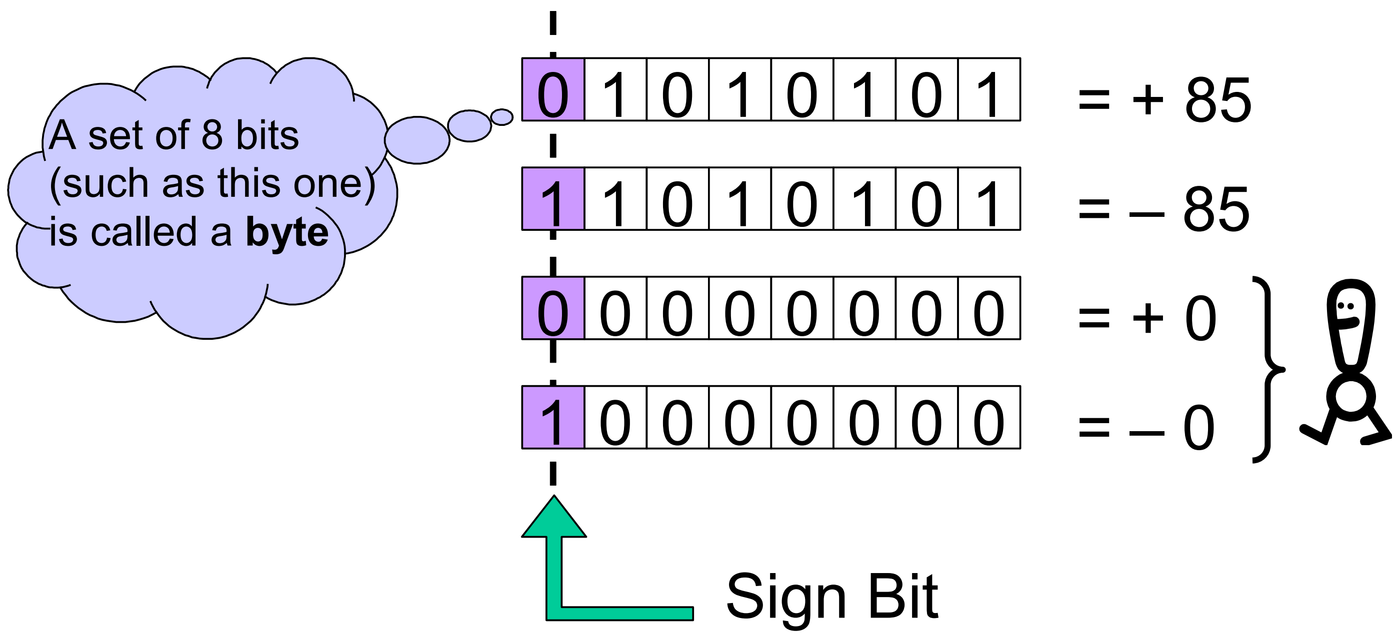
Chapters 1 & 2 – "Digital Design: Principles and Practices" book

Unsigned, Signed and Negative Numbers

- Digital computers store numbers in a special electronic device (memory) called register, with properties:
  - A fixed size (number of elements); each element holds 0 or 1.
  - The register size is typically a power of 2 e.g., 2, 4, 8, 16, …
  - An *n*-bit register can represent one of 2^n distinct values.
  - Numbers stored in a register can be signed or unsigned.
- Negative numbers are essential, and any computer not capable of dealing with them would not be particularly useful.
  - How can such numbers be represented?
    - *Sign-Magnitude Representation*.
    - *Two's Complement*.

Signed Magnitude Representation

- Magnitude and symbol representing +/ –
  - *Decimal*: Uses "+" or "–" as necessary.
    - **Examples**: +92; –15.
  - *Binary* (8 bits): MSB represents the sign (*0* = positive; *1* = negative).



Two's Complement

- Used by most machines and languages to ***represent integers***.
- Fixes the ***–0*** in the signed magnitude, and simplifies machine hardware for arithmetic.
- Divides bit patterns into a ***positive half*** and a ***negative half***. ***Zero*** is considered positive.
- ***n*** bits create a range over **[$-\ 2^{n-1}$ … $2^{n-1} - 1$]**.

Code	Simple	Signed	2's Comp
0000	0	+0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-0	-8
1001	9	-1	-7
1010	10	-2	-6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

Conversion: Binary – 2's Complement

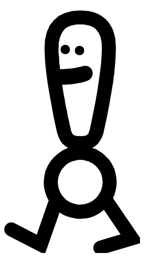
- Conversion to 2's complement:
  - *Positive numbers*: same as simple binary.
  - *Negative numbers*:
    1. Obtain the *n-bit* simple binary equivalent.
    2. Invert the bits of that representation.
    3. Add *1* to the result.

- Example: Convert -320_{10} to 16-bit 2's complement.

$$
\begin{array}{rl}
320_{10} & = 0000\ 0001\ 0100\ 0000_2 \\
 & = 1111\ 1110\ 1011\ 1111 \quad \Leftarrow \text{Invert} \\
 & + 0000\ 0000\ 0000\ 0001 \quad \Leftarrow \text{Add } 1 \\
-320_{10} & = 1111\ 1110\ 1100\ 0000
\end{array}
$$

Sign and Zero Extension

- Consider the value 64_{10}:
  - the *8-bit* representation is `40h` `0100 0000`
  - the *16-bit* equivalent is `0040h` `0000 0000 0100 0000`
- Consider the value -64_{10}:
  - the *8-bit* 2's complement is `C0h` `1100 0000`
  - the *16-bit* equivalent is `FFC0h` `1111 1111 1100 0000`



Rule: To sign extend a value from some number of bits to a greater number of bits, copy the sign bit into all the additional bits in the new format.

Signed Binary Arithmetic

- **Operations** (addition & subtraction only for now):
  - 8-bit Signed Magnitude Arithmetic: similar to standard addition and subtraction, but only working with *0* and *1*.
    - **Examples**:
      - `0101 0110 + 0110 0011` = ?
      - `0101 0110 - 0110 0011` = ?

Overflow!!

  - 2's Complement Binary Arithmetic:
    - Addition and subtraction are the same operation (since a subtraction symbol becomes a negative second number).
      - **Example**: $23_{10} - 45_{10} = 23_{10} + (-45_{10})$

Queen Mary
University of London

Overflow (*when we run out of space …*) – 1/2

- Digital systems have a *finite amount of space to store numbers*.
  - Example: Numbers stored in an 8-bit processor can have a maximum of 8 bits.

- Overflow: When the result of an arithmetic or logical operation is a value that requires more space than is available.
  - Examples:

```
  0111
 +0111
  1110
```

overflow causes incorrect sign

```
  1000
 +1000
 10000
```

result has overflowed (the MSB is lost)

Queen Mary University of London

Overflow (*when we run out of space …*) – 2/2

- Avoiding overflow (the designer's responsibility):
  - You must *constrain the range of allowed inputs* to an operation and *ensure that there is enough space* available for the maximum and minimum outputs.
  - When the *output requires more bits than available*, we can truncate the LSB before using that value in further operations.
    - This results in lower accuracy/precision, but removing the MSB would cause huge errors!

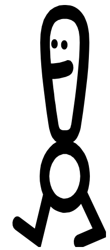
Example: $23_{10} + 45_{10}$

First, put numbers into binary and then into *two's complement*:

23_{10} →

```
23 ÷ 2 = 11   r = 1
11 ÷ 2 =  5   r = 1
 5 ÷ 2 =  2   r = 1
 2 ÷ 2 =  1   r = 0
 1 ÷ 2 =  0   r = 1
```

→ 0001 0111



Both numbers are positive, so 2's complement is same as binary number.

45_{10} →

```
45 ÷ 2 = 22   r = 1
22 ÷ 2 = 11   r = 0
11 ÷ 2 =  5   r = 1
 5 ÷ 2 =  2   r = 1
 2 ÷ 2 =  1   r = 0
 1 ÷ 2 =  0   r = 1
```

→ 0010 1101

```
  0001 0111
+ 0010 1101
-----------
  0100 0100
```

Queen Mary University of London

Example: $23_{10} - 45_{10}$

This can be rewritten as: $23_{10} + (-45_{10})$, so we need to convert the binary representation of 45_{10} to two's complement.

```
45₁₀  = 0010 1101
        1101 0010   <- Invert
       +0000 0001   <- Add 1
        1101 0011 = -45₁₀
```

Now just add!

```
     23    0001 0111
+   -45   +1101 0011
    -22    1110 1010
          -0000 0001   <- Subtract 1
           1110 1001
           0001 0110   <- Invert to convert to simple binary
```

Check

This is -22_{10} really, as we're just checking!

$= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 16 + 0 + 4 + 2 + 0 = 22_{10}$

Queen Mary University of London

Floating Point (f.p.) Numbers: Why and What

- What: Way to represent real numbers using digits or bits, based on *scientific notation* (used to represent exact values):
  - Value = ± 1.Mantissa x Base$^{(\text{Exponent})}$
  - Base chosen: usually a multiple of 2, especially for computer arithmetic.
- Why: Often necessary to be able of representing both large and small real numbers.
- F. p. notation: not as accurate as integer notation, but saves bits in representing both large and small real numbers.
- F. p. numbers are usually *normalised*:
  - The *Significand* (1.Mantissa) is shifted to the left until all leading zeros disappear, as this efficiently uses the bits available for the *Significand*.
  - The value is kept the same by appropriate adjustment of the *Exponent*.

Floating Point Formats (1/3)

- The three types (corresponding to different computer architectures) we will examine briefly:
  - **IBM System 360/370**: supports a hexadecimal floating-point format;
  - **DEC PDP 11/VAX**;
  - **IEEE-754**: similar to IBM's system, but with longer *Exponent* and shorter *Mantissa*.

IBM System 360/370 Format

	7 bits	24 bits
s	e	f

$$\text{Value} = (-1)^s \, 0.f \times 16^{(e-64)}$$

Floating Point Formats (2/3)

DEC PDP 11/VAX Format

s	e (8 bits)	f (23 bits)

Value = $(-1)^s \; 0.1f \times 2^{(e-128)}$

IEEE-754 Format

sign

exponent

significant fractional part (*or* mantissa)

bias

s	e (8 bits)	f (23 bits)

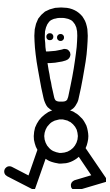
Normal Value = $(-1)^s \; 1.f \times 2^{(e-\boxed{127})}$

Denormal Value = $(-1)^s \; 0.f \times 2^{(1-127)}$

Queen Mary
University of London

Floating Point Formats (3/3)

- **IEEE-754** format is similar to **DEC** except:
  - it uses an offset of ***127*** instead of ***128***;
  - the hidden bit is to the left of the ***radix point***;
  - it can represent ***-0*** as well as ***+0***;
  - it can represent values very close to zero through the use of the ***denormals***.



Floating point numbers have a finite resolution; they can only represent discrete values.

Example (1/2): IEEE-754 FP

Represent 0.0390625_{10} in floating point format:

$$
\begin{aligned}
0.0390625_{10} &= 0.0000101_2 \\
&= 1.01_2 \times 2^{-5} \\
&= 1.01_2 \times 2^{(122\ -\ 127)} \\
&= 1.01_2 \times 2^{(01111010_2\ -\ 127)}
\end{aligned}
$$

Value = $(-1)^0 \times 1.01_2 \times 2^{(01111010_2\ -\ 127)}$

	8 bits	23 bits
0	01111010	01000000000000000000000

Binary Representation

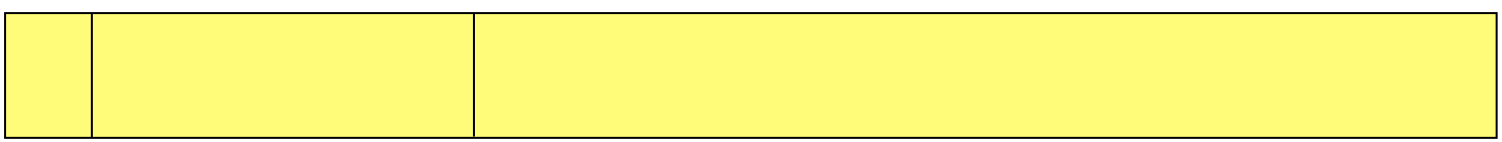
Example (2/2): IEEE-754 FP

Represent -0.75_{10} in floating point format:

$$
\begin{aligned}
-\ 0.75_{10} &= -\ 0.11_2 \\
&= -\ 1.1_2 \times 2^{-1} \\
&= -\ 1.1_2 \times 2^{(126\ -\ 127)} \\
&= -\ 1.1_2 \times 2^{(01111110_2\ -\ 127)}
\end{aligned}
$$

Value =

8 bits **23 bits**



Binary Representation

Example: 8-bit f.p. with excess-3 exponent code

- General format of a f.p. number:

$$\text{value} = \pm \underbrace{1.\overbrace{\text{mmm...mm}}^{\textit{mantissa}}}_{\textit{significand}} \times 2^{\overbrace{\scriptstyle(\overbrace{\text{eee...ee}}^{\textit{excess-coded exponent}} - \text{bias})}_{\textit{true value of exponent}}}$$

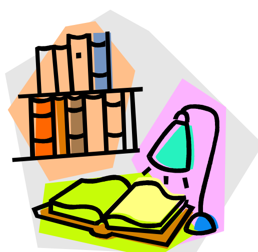
- Example:
  - Represent the number 7.25_{10} in f.p. format with *excess-3* exponent code.

$$7.25_{10} = (-1)^0 \times 111.01_2$$
$$= (-1)^0 \times 1.1101 \times 2^2$$
$$= (-1)^0 \times 1.1101 \times 2^{5-3}$$
$$= (-1)^0 \times 1.1101 \times 2^{101_2-3}$$

0	101	1101

Overview: Number Systems & Codes

* **Introduction**
* **Number Systems**
* **Binary and Hex Arithmetic**
* **Negative Numbers & Floating Point**
* **Codes: Binary, Decimal, Alphanumeric**



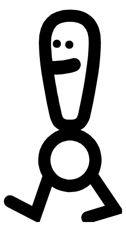
Chapters 1 & 2 – "Digital Design: Principles and Practices" book

Binary Codes (1/2)

- **1 Binary Digit** (one bit) can take on values ***0***, ***1***. We can represent the ***two*** values as:
  - (*0* = hot; *1* = cold) – (*1* = True; *0* = False) – (*1* = on; *0* = off)
- **2 Binary Digits** (two bits) can take on values of ***00***, ***01***, ***10***, ***11***. We can represent the ***four*** values as:
  - (*00* = hot, *01* = warm, *10* = cool, *11* = cold)
- **3 Binary Digits** (three bits) can take on values of ***000***, ***001***, ***010***, ***011***, ***100***, ***101***, ***110***, ***111***. We can represent the ***eight*** values as:
  - *000* = Black, *001* = Red, *010* = Pink, *011* = Yellow, *100* = Brown, *101* = Blue, *110* = Green, *111* = White.

Binary Codes (2/2)

- ***N* bits** (or *N* binary digits) can represent 2^N different values.
  - E.g., **4** bits can represent $\mathbf{2^4}$ or **16** different values.
- *N* bits can take on unsigned decimal values from *0* to 2^N-1.
- Codes are usually given in tabular form.



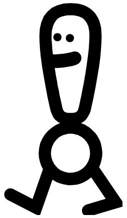
Code: Set of *n-bit* strings where different bit strings represent different numbers (or other quantities).

000	black
001	red
010	pink
011	yellow
100	brown
101	blue
110	green
111	white

Binary Data (Again!)

- The computer screen on your PC can be configured for **different resolutions**.
  - One resolution is 600 x 800 x 8 ⇒ 600 dots vertically by 800 dots horizontally, with each dot using 8 bits to take on 256 different colors.
- 8 bits are needed to represent 256 colors ($2^8 = 256$). The **total number of bits needed to represent the screen** is then: 600 x 800 x 8 = 3840000 bits (or just under **4 Mbits**).

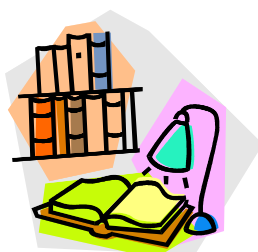
1 Mbits = 1024 x 1024 = $2^{10} \times 2^{10} = 2^{20}$
1 Kbits = 1024 = 2^{10}



Your **video card** must have at least this much memory on it!

Overview: Number Systems & Codes

* Introduction
* Number Systems
* Binary and Hex Arithmetic
* Negative Numbers & Floating Point
* Codes: Binary, Decimal, Alphanumeric



Chapters 1 & 2 – “Digital Design: Principles and Practices” book

Queen Mary
University of London

Binary Coded Decimal (BCD)

- BCD uses a *4-bit* pattern to express each digit of a *base 10* number.
- How each digit in BCD is encoded:
  - *in BCD*
    - 123: $\underbrace{\texttt{0001}}_{1}\ \underbrace{\texttt{0010}}_{2}\ \underbrace{\texttt{0011}}_{3}$
    - +123: 1010 0001 0010 0011
    - -123: 1011 0001 0010 0011
  - *in simple Binary*
    - 123 : 111 1011

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
+	1010
-	1011

Some implementations only; usually, the last 6 values are not used.

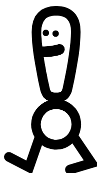
BCD Advantages/Disadvantages

- **BCD Advantages**:
  - Used in business machines and languages e.g., COBOL for precise decimal math.
  - Can have arrays of BCD numbers for arbitrary precision arithmetic.
- **BCD Disadvantages**:
  - Takes more memory because,
    - 32-bit simple binary can represent over 4 billion discrete values;
    - 32-bit BCD can hold a sign and 7 digits for a maximum of 10 million values.
  - More difficult to do maths because we must force the *base 2* computer to do *base 10* arithmetic.

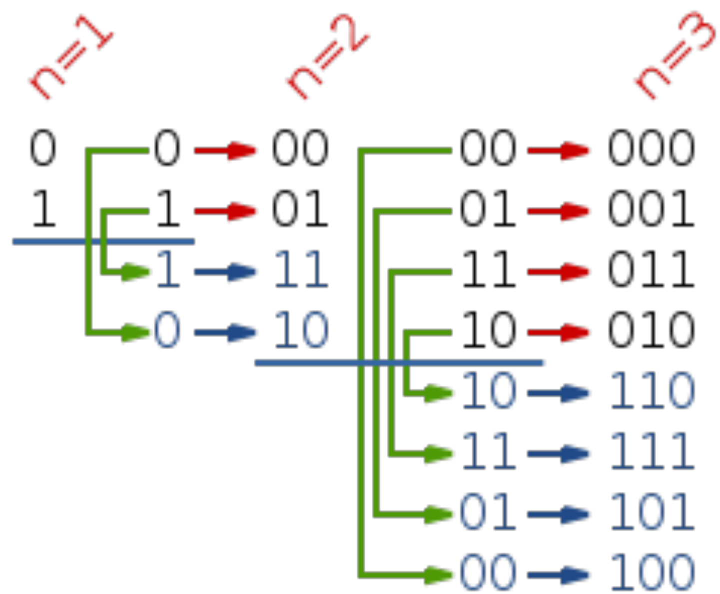
Gray Code for Decimal Digits

Gray Code		
0	⇔	0000
1	⇔	0001
2	⇔	0011
3	⇔	0010
4	⇔	0110
5	⇔	0111
6	⇔	0101
7	⇔	0100
8	⇔	1100
9	⇔	1101

- A Gray code (here, a *4-bit* code) changes by only 1 bit for adjacent values.
  - It's also called a '**thumbwheel**' code because a thumbwheel for choosing a decimal digit can only change to an adjacent value (e.g., 4 to 5 to 6, etc) with each click of the thumbwheel.



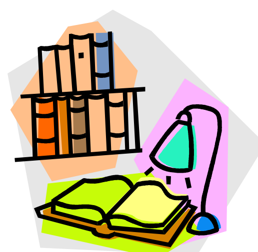
Gray Code: used e.g., in automatic braking systems.



Queen Mary
University of London

Overview: Number Systems & Codes

* Introduction
* Number Systems
* Binary and Hex Arithmetic
* Negative Numbers & Floating Point
* Codes: Binary, Decimal, **Alphanumeric**



Chapters 1 & 2 – "Digital Design: Principles and Practices" book

Queen Mary
University of London

ASCII Code

- If there's a **need to represent characters as digital data** …
  - The **ASCII** code (i.e., **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) is a *7-bit* code for character data.
  - Typically, 8 bits are actually stored with the 8-th bit being zero or used for error detection (parity checking), where **8 bits = 1 Byte**.
    - ***Examples***:
      - 'A' = 01000001_2 = 41_{16}
      - '&' = 00100110_2 = 26_{16}

What about **other symbols or other languages**? How can they be represented?

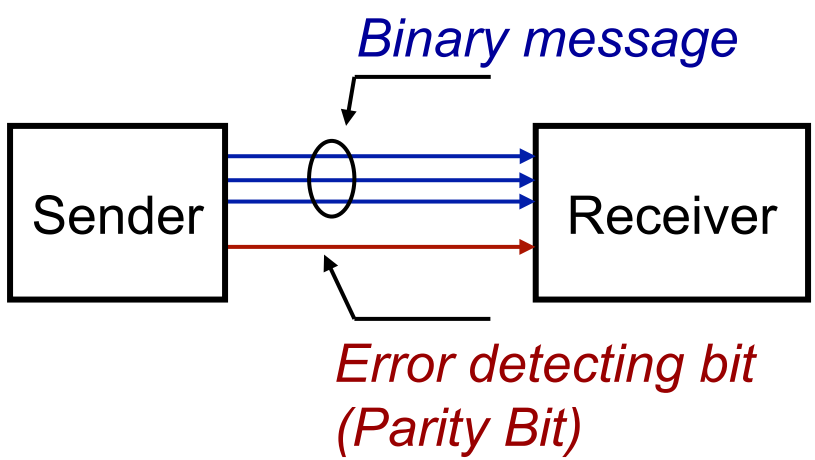
  - 7 bits can only represent **2^7 different values** (i.e., 128). This is enough to represent the **Latin alphabet** (**A-Z**, **a-z**, **0-9**, punctuation marks and some symbols like **$**).

Unicode

- **Unicode**: *16-bit* code for representing alphanumeric data. It can represent 2^{16} or 65536 different symbols: 16 bits = 2 Bytes per character.
  - **Examples**:
    - $0041_{16} - 005A_{16}$ **A – Z**
    - $0061_{16} - 007A_{16}$ **a – z**
- **Some other alphabet/symbol ranges**:
  - $3400_{16} - 3D2D_{16}$ *Korean Hangul Symbols*
  - $3040_{16} - 318F_{16}$ *Hiranga, Katakana, Bopomofo, Hangul*
  - $4E00_{16} - 9FFF_{16}$ *Han* (Chinese, Japanese, Korean)
- **Unicode** is now used by Web browsers, Java and most software.
- **Online code guide**: http://www.unicode.org/charts/

Parity Checking

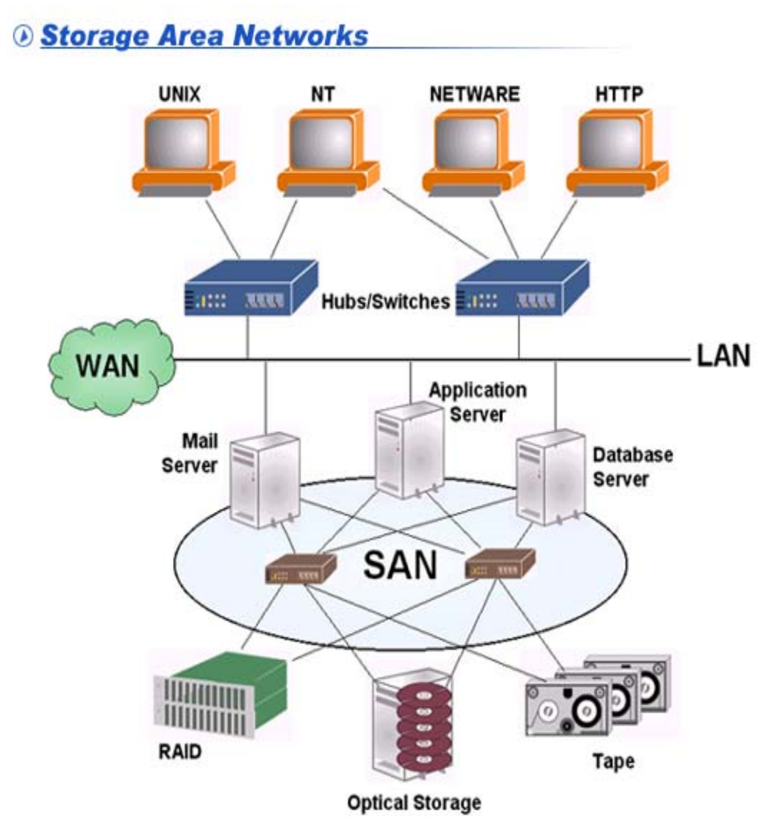
- *Simple error detection (not correction) mechanism*
  - Data transmission, ageing media, static interference, dust on media, etc, demand the ability to detect errors.
  - Single bit errors can be detected by using parity checking.
    - Constructing a single-error-detecting code with 2^n code words.
    - Requires n+1 bits: the first n bits are called information bits and the remaining bit is called the parity bit.
  - Advantages of parity checking:
    1. best possible code using only a single bit of space;
    2. requires only a number of *XOR gates* to generate.



Queen Mary University of London

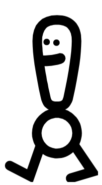
Parity Checking (*cont.*)

- *Parity Checking*: the most fundamental method of error checking.
  - Other codes e.g., *Hamming* and *VRC/LRC* are also used frequently.
  - Parity Checking in SANs (Storage Area Networks):
    - Several types of RAID (Redundant Array of Independent Disks) arrays *calculate parity* to provide redundancy against failure.



Odd and Even Parity

- **Parity bit**:
  - Even-parity code: set the parity bit to *0* if there's an even number of *1s*; set the parity bit to *1* otherwise.
  - Odd-parity code: set the parity bit to *0* if there's an odd number of *1s*; set the parity bit to *1* otherwise.
- **Detection of a 1-bit error**:
  - The parity bit is stripped by hardware after checking. The sender and receiver both agree to apply odd or even parity.
  - 2 (or an even number of) flipped bits (including the parity bit) in the same byte are not detected!



The parity bit can be added either at the beginning *or* at the end of the value.

Example: Odd and Even Parity

What is actually sent when even parity has been agreed.

	'S'	'E'
ASCII	`101 0111`	`100 0101`
Even Parity	`0101 0011`	`1100 0101`
Odd Parity	`1101 0011`	`0100 0101`

- **Example** (detection of a 1-bit error):
  - ASCII '`S`' is sent (i.e., send **`1010011`**$_2$), but value **`01010010`**$_2$ is received.
    - Need to add a *1-bit parity* to make an odd or even number of bits per byte.

Assuming value `01010010`$_2$ is received, can Parity Checking detect an error when character '`S`' is sent, if *even parity* has been agreed?

What if instead we send character '`E`' and *odd parity* has been agreed? Can Parity Checking detect an error?

Queen Mary University of London