



Queen Mary

University of London

Science and Engineering

EBU4202: Digital Circuit Design

Number Systems & Codes

Dr. Md Hasanuzzaman Sagor (Hasan)

Dr. Chao Shu (Chao)

Dr. Farha Lakhani (Farha)

School of Electronic Engineering and Computer Science,
Queen Mary University of London,
London, United Kingdom.

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

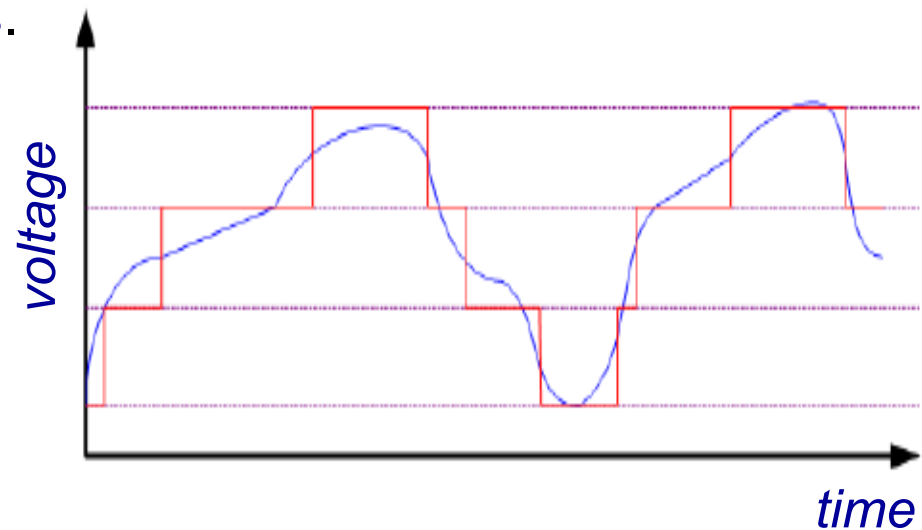
- **Analog devices** process time-varying signals that can have any value across a **continuous** range *and* produce results that are also in **continuous** form.
 - **Examples** of continuous signals: voltage, current, force.
- **Digital devices** process signals that take on only two **discrete**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

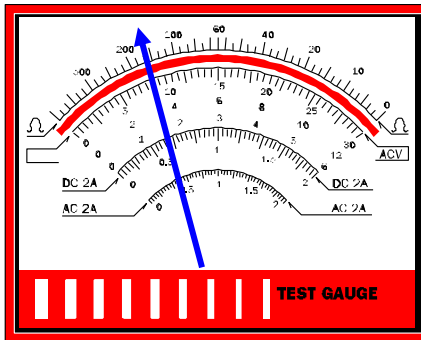
- The **world around us is analog** but digital systems are simple to understand and use ...
 - Common practice is to **convert analog signals into digital form** for **efficient processing of signals**.
 - However, it is **not possible to avoid loss of some accuracy** (information) due to this conversion, because **digital systems can only represent fixed** (finite or discrete) **sets of values**.
 - NOTE: the values obtained in a digitized system must be coded using 1s and 0s.



Digital Data: Advantages

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

Analog Voltage meter

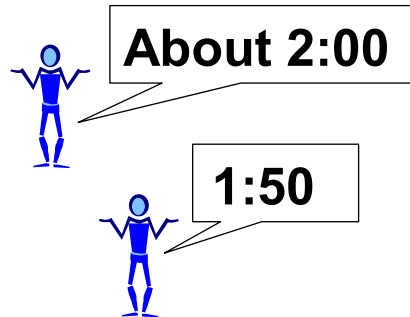
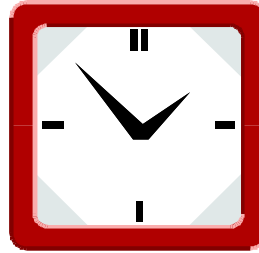


About 100

Digital Voltage meter

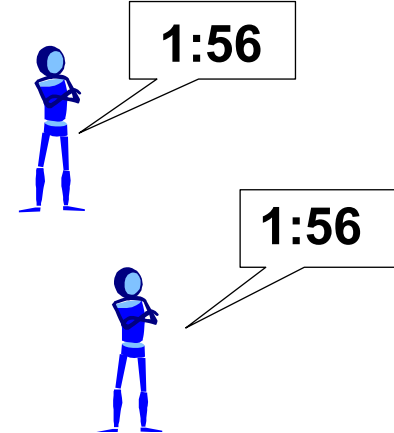
103.5

Analog Clock



Digital Clock

1:56 pm



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 (Hardware Description Language) 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 Integrated Circuit (IC).



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

Binary Representation

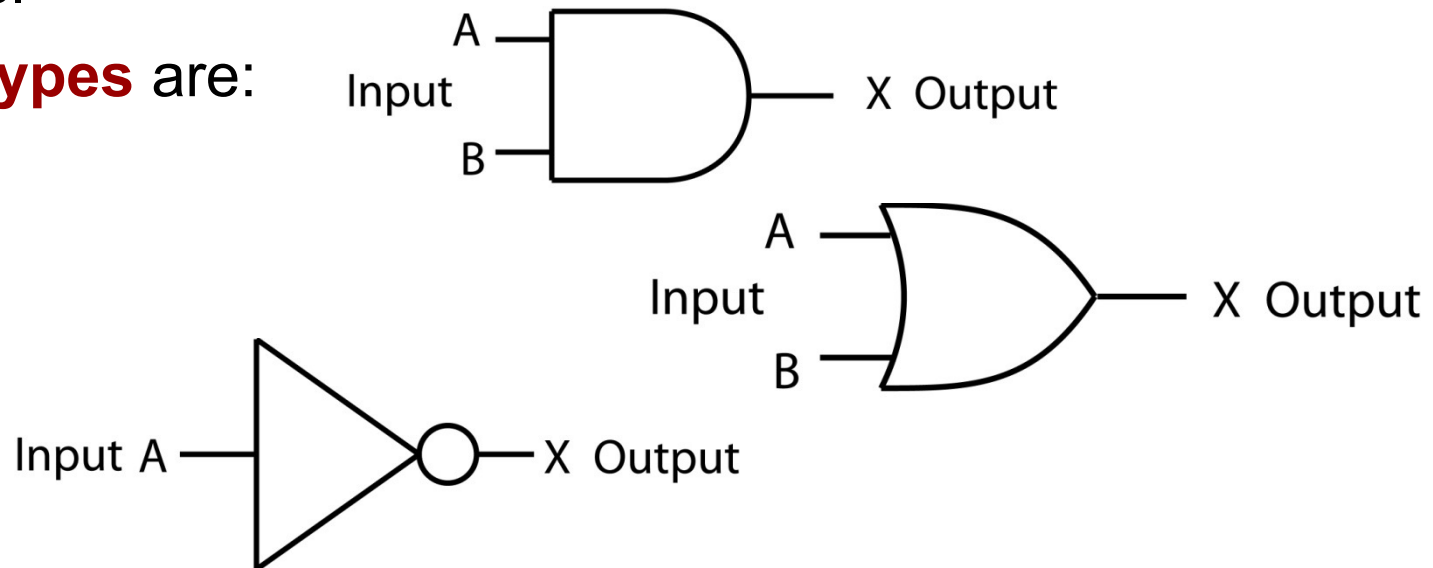
- Basis of all digital data is binary representation.
- Binary → means 'two'
1, 0 // True, False // Hot, Cold // On, Off
- Computers (digital systems) represent data in the binary system using:
 - Electrical voltages (e.g., in processors, memory);
 - Magnetism (e.g., in hard disks);
 - Light (e.g., in CD, DVD).
- However, we *must be able to handle more than just two values* for real world problems.
 - 1, 0, 56
 - On, Off, Leaky
 - True, False, Maybe
 - 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 e.g. 5 and 0 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.
- NOTE: alternative systems may use +5 Volts and -5 volts to represent logic 1 and logic 0.

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

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

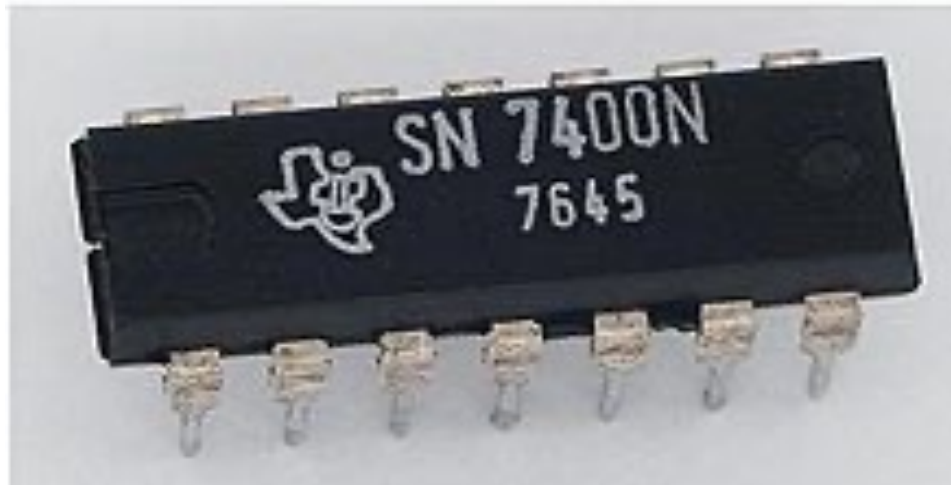
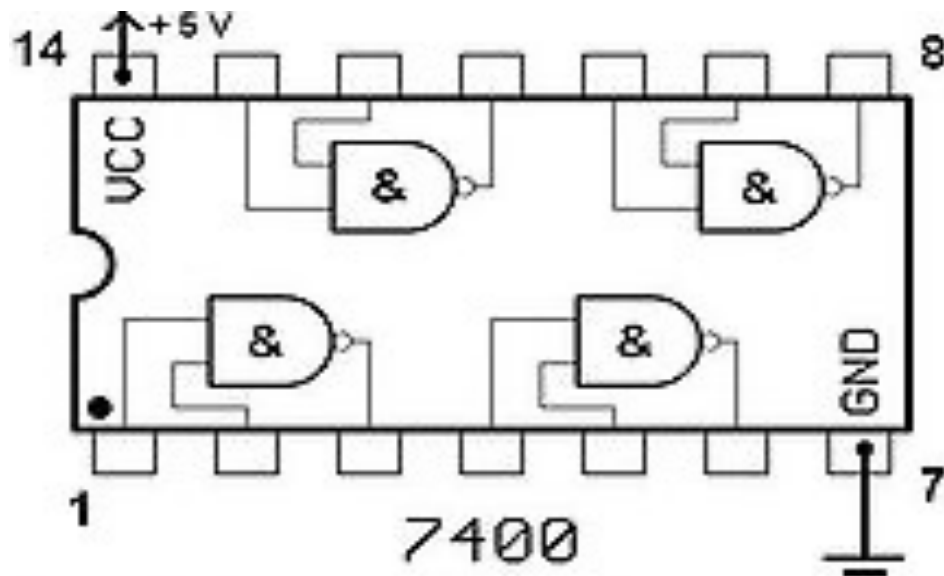
Classification of ICs based on size (i.e., number of gates)

Name	Number of Gates
SmallScale Integration (SSI)	< 20
MediumScale Integration (MSI)	20 – 200
LargeScale Integration (LSI)	200 – 200000
Very LargeScale Integration (VLSI)	≈ 1 million transistors

measure
used for VLSIs

NOTE: The ICs need a power supply connection.

From Wikipedia 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 (Very High Speed HDL)
- Other simulation software can be used to connect gates and other devices together into a circuit.

Overview: Number Systems & Codes

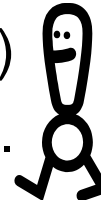
- * Introduction
- * **Number Systems**
- * Binary and Hex Arithmetic
- * Negative Numbers & Floating Point
- * Codes: Binary, Decimal, Alphanumeric



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

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.
 - $321_{10} = (3 \cdot 10^2) + (2 \cdot 10^1) + (1 \cdot 10^0)$
 - The **subscript** in 321_{10} is the **base**.



If no subscript is shown, assume a decimal number.

Number Systems: At a Glance

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

Decimal	Binary	8bit Binary No.	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**' (binary digit).
 - 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,.... We will not consider these. Only really useful for year dating!

Least & Most Significant Digits

$$53_{10} = \textcolor{red}{1}1010\textcolor{green}{1}_2$$


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.

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:**

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

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

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

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

– Hex to Decimal

$$\begin{aligned} A2F_{16} &= (10 \times 16^2) + (2 \times 16^1) + (15 \times 16^0) \\ &= (10 \times 256) + (2 \times 16) + (15 \times 1) \\ &= 2560 + 32 + 15 \\ &= 2607_{10} \end{aligned}$$

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.

$53 \div 2 = 26$	$r =$	1	(LSB)
$26 \div 2 = 13$	$r =$	0	
$13 \div 2 = 6$	$r =$	1	
$6 \div 2 = 3$	$r =$	0	
$3 \div 2 = 1$	$r =$	1	
$1 \div 2 = 0$	$r =$	1	(MSB)

Quotient = 0 so stop.

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

Check your work:

110101_2

$$\begin{aligned} &= 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 32 + 16 + 0 + 4 + 0 + 1 \\ &= 53 \end{aligned}$$

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

$$\begin{array}{lcl} 5 \div 2 = 2 & r=1 & \text{(LSD)} \\ 2 \div 2 = 1 & r=0 & \\ 1 \div 2 = 0 & r=1 & \text{(MSD)} \end{array}$$

Fractional Part

$$\begin{array}{lcl} 0.2 \times 2 = 0.4 & \text{(MSD)} & \\ 0.4 \times 2 = 0.8 & & \\ 0.8 \times 2 = 1.6 & & \\ \text{etc.} & & \end{array}$$

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

Note: it is often not possible to get an EXACT value

Example: Non-integer to Binary Conversion

- Convert 225.225 to binary

Integer Part

$225 \div 2 = 112$	$r = 1$ (LSD)
$112 \div 2 = 56$	$r = 0$
$56 \div 2 = 28$	$r = 0$
$28 \div 2 = 14$	$r = 0$
$14 \div 2 = 7$	$r = 0$
$7 \div 2 = 3$	$r = 1$
$3 \div 2 = 1$	$r = 1$
$1 \div 2 = 0$	$r = 1$ (MSD)
0	

Fractional Part

$0.225 \times 2 = 0.45$	(MSD)
$0.45 \times 2 = 0.9$	
$0.9 \times 2 = 1.8$	
$0.8 \times 2 = 1.6$	
$0.6 \times 2 = 1.2$	
$0.2 \times 2 = 0.4$	
$0.4 \times 2 = 0.8$	
$0.8 \times 2 = 1.6$	
etc.	

Therefore $225.225_{10} \cong 11100001.00111001_2$

Example: Decimal to Hex Conversion

- Convert 53_{10} to hexadecimal.

$$53 \div 16 = 3 \quad r = 5 \quad (\text{LSD})$$

$$3 \div 16 = 0 \quad r = 3 \quad (\text{MSD})$$

0

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

Check your work:

Example: Non-Integer to Hex Conversion

- Convert 265.78_{10} to hexadecimal.

Integer part

$$265 \div 16 = 16 \quad r = 9 \text{ (LSD)}$$

$$16 \div 16 = 1 \quad r = 0$$

$$1 \div 16 = 0 \quad r = 1 \text{ (MSD)}$$

0

Fractional part

$$0.78 \times 16 = 12.48 \text{ (MSD)}$$

$$0.48 \times 16 = 7.68$$

$$0.68 \times 16 = 10.88$$

$$0.88 \times 16 = 14.08 \text{ etc.}$$

Answer:

Remember to use Hex digits for values greater than decimal 9

Example: Non-Integer to Hex Conversion

- Convert 265.78_{10} to hexadecimal.

Integer part

$$265 \div 16 = 16 \quad r = 9 \text{ (LSD)}$$

$$16 \div 16 = 1 \quad r = 0$$

$$1 \div 16 = 0 \quad r = 1 \text{ (MSD)}$$

0

Fractional part

$$0.78 \times 16 = 12.48 \text{ (MSD)}$$

$$0.48 \times 16 = 7.68$$

$$0.68 \times 16 = 10.88$$

$$0.88 \times 16 = 14.08 \text{ etc.}$$

Answer: 109.C7AE

Remember to use Hex digits for values greater than decimal 9

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

1 = 0001

2 = 0010

3 = 0011

4 = 0100

5 = 0101

6 = 0110

7 = 0111

8 = 1000

9 = 1001

A = 1010

B = 1011

C = 1100

D = 1101

E = 1110

F = 1111

Hex to Binary *versus* Binary to Hex

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

- *Examples:*

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

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

- **Binary to Hex** is just the opposite:

- Create groups of 4 bits, starting with least significant bits.
- If the last group does not have 4 bits, then pad with zeros for unsigned numbers.

- *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:**

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

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

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*:

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$1 + 1 = 0 + (\text{carry } 1)$$

$$1 + 1 + 1 = 1 + (\text{carry } 1)$$

Addition

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$(\text{borrow } 2) 0 - 1 = 1$$

Subtraction

Examples: Binary Addition & Subtraction

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

1 0 0 1 0 1 1 1

carries

1 1 1 1 1 1 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	<i>borrows</i>	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 →

$$\begin{array}{rcccccc} & & & 1 & 1 & & \\ \text{A} & 7 & 3 & 4 & 8 & 6 & 5 \\ + & 1 & 2 & 5 & 9 & \text{D} & 6 \\ \hline \text{A} & 8 & 5 & \text{A} & 2 & 3 & \text{B} \end{array}$$

(11)

borrows →

$$\begin{array}{rcccccc} & & & 17 & & & \\ & & & 9 & 1 & 19 & \\ \text{A} & 8 & 5 & \text{A} & 2 & 3 & \text{B} \\ - & 1 & 2 & 5 & 9 & \text{D} & 6 \\ \hline \text{A} & 7 & 3 & 4 & 8 & 6 & 5 \end{array}$$

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.*

NOTE: To correctly interpret a number, you need to be told which representation is being used!


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).

A set of 8 bits
(such as this one)
is called a **byte**

0	1	0	1	0	1	0	1	= + 85
1	1	0	1	0	1	0	1	= – 85
0	0	0	0	0	0	0	0	= + 0
1	0	0	0	0	0	0	0	= – 0

↑ Sign Bit

} 

Two's Complement

4 Bit Binary

- 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 **$[-2^{n-1} \dots 2^{n-1} - 1]$** .

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.

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

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.
 - **Example:**
 - $0101\ 0110 + 0110\ 0011 = ?$ (10111001)
Signed magnitude so appears to be negative! **Overflow!!**
 - 2's Complement Binary Arithmetic:
 - Addition and subtraction are the same operation (since a subtraction symbol becomes a negative second number).
 - **Example:** $23_{10} - 45_{10} = 23_{10} + (-45_{10})$

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**:

$$\begin{array}{r} 0111 \\ +0111 \\ \hline 1110 \end{array}$$

overflow causes
incorrect sign

$$\begin{array}{r} 1000 \\ +1000 \\ \hline 10000 \end{array}$$

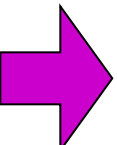
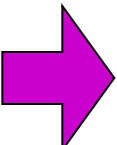
result has overflowed
(the MSB is lost)

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

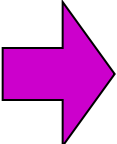
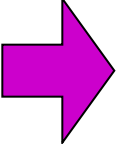
First, put numbers into binary and then into *two's complement*:

23_{10} 
$$\begin{array}{rcl} 23 \div 2 & = & 11 \quad r = 1 \\ 11 \div 2 & = & 5 \quad r = 1 \\ 5 \div 2 & = & 2 \quad r = 1 \\ 2 \div 2 & = & 1 \quad r = 0 \\ 1 \div 2 & = & 0 \quad r = 1 \end{array}$$
  $0001\ 0111$



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

$$\begin{array}{r} 0001\ 0111 \\ + 0010\ 1101 \\ \hline 0100\ 0100 \end{array}$$

45_{10} 
$$\begin{array}{rcl} 45 \div 2 & = & 22 \quad r = 1 \\ 22 \div 2 & = & 11 \quad r = 0 \\ 11 \div 2 & = & 5 \quad r = 1 \\ 5 \div 2 & = & 2 \quad r = 1 \\ 2 \div 2 & = & 1 \quad r = 0 \\ 1 \div 2 & = & 0 \quad r = 1 \end{array}$$
  $0010\ 1101$

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.

Now just add!

$$\begin{array}{r} 23 \quad 0001 \ 0111 \\ + \quad -45 \quad +1101 \ 0011 \\ \hline -22 \quad 1110 \ 1010 \end{array}$$

$45_{10} = 0010 \ 1101$
 $\quad \quad 1101 \ 0010$ ← Invert
 $\quad \quad +0000 \ 0001$ ← Add 1

 $\quad \quad 1101 \ 0011 = -45_{10}$

$$\begin{array}{r} 1110 \ 1010 \\ -0000 \ 0001 \quad \leftarrow \text{Subtract 1} \\ \hline 1110 \ 1001 \end{array}$$

$$0001 \ 0110 \quad \leftarrow \text{Invert to convert to simple binary}$$

$$\begin{aligned} &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 16 + 0 + 4 + 2 + 0 = 22_{10} \end{aligned}$$

Check

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

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):
 - $\text{Value} = \pm 1.\text{Mantissa} \times \text{Base}^{(\text{Exponent})}$
 - **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

- 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*.

IEEE-754 Format



$$\text{Normal Value} = (-1)^s 1.f * 2^{e-127}$$

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

Floating Point Formats-2

- **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**.
 - It represents a 32-bit word



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:

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

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

8-bits

23-bits

0	01111010	010000000000000000000000
----------	-----------------	---------------------------------

Binary Representation

Example (2/2): IEEE-754 FP

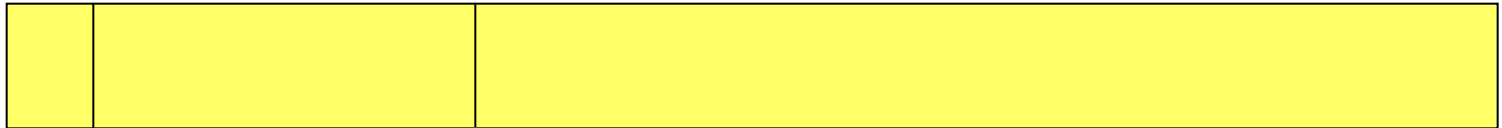
Represent -0.75_{10} in floating point format:

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

Value =

8-bits

23-bits



Binary Representation

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

- General format of a f.p. number:

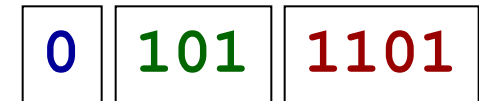
$$\text{value} = \pm 1 \underbrace{\text{mmm} \dots \text{mm}}_{\text{significand}} \times 2^{\underbrace{(\text{eee} \dots \text{ee} - \text{bias})}_{\text{true value of exponent}}}$$

mantissa *excess-coded exponent*

- Example:

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

$$\begin{aligned} 7.25_{10} &= (-1)^0 \times 111.01_2 \\ &= (-1)^0 \times 1.1101 \times 2^2 \\ &= (-1)^0 \times 1.1101 \times 2^{5-3} \\ &= (-1)^0 \times 1.\mathbf{1101} \times 2^{\mathbf{101}_2 - \mathbf{3}} \end{aligned}$$



Floating Point Bias

- The bias is $2^{k-1} - 1$, where k is the number of bits in the exponent field.
- For an 8-bit format, $k = 3$, so the bias is $2^{3-1} - 1 = 3$.
- For IEEE 32-bit, $k = 8$, so the bias is $2^{8-1} - 1 = 127$.

IEEE is (1 sign bit + 8 exponent bits + 23 mantissa bits) = 32 bits

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 **2⁴** 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 $600 \times 800 \times 8 \Rightarrow$ 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:
 $600 \times 800 \times 8 = 3840000$ bits (or just under **4 Mbits**).

$$\begin{aligned} 1 \text{ Mbits} &= 1024 \times 1024 = 2^{10} \times 2^{10} = 2^{20} \\ 1 \text{ Kbits} &= 1024 = 2^{10} \end{aligned}$$



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

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 : $\overset{1}{\underbrace{0001}} \overset{2}{\underbrace{0010}} \overset{3}{\underbrace{0011}}$
 - +123 : 1010 0001 0010 0011
 - -123 : 1011 0001 0010 0011
 - *in simple Binary*
 - 123 : 111 1011

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

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

BCD Advantages/Disadvantages

- **BCD Advantages:**

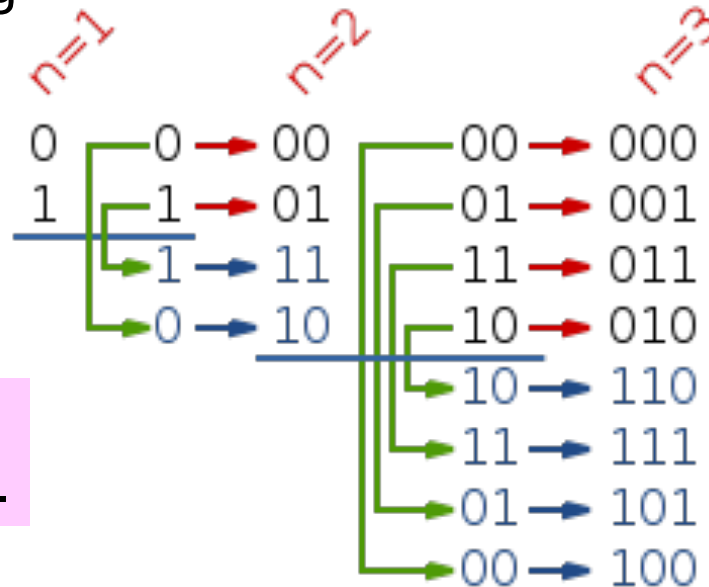
- Used in business machines and languages e.g., COBOL for precise decimal math.
- Can have arrays of BCD numbers for arbitrary precision arithmetic.

- **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

- 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

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



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


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

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 8th 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 **\$**).

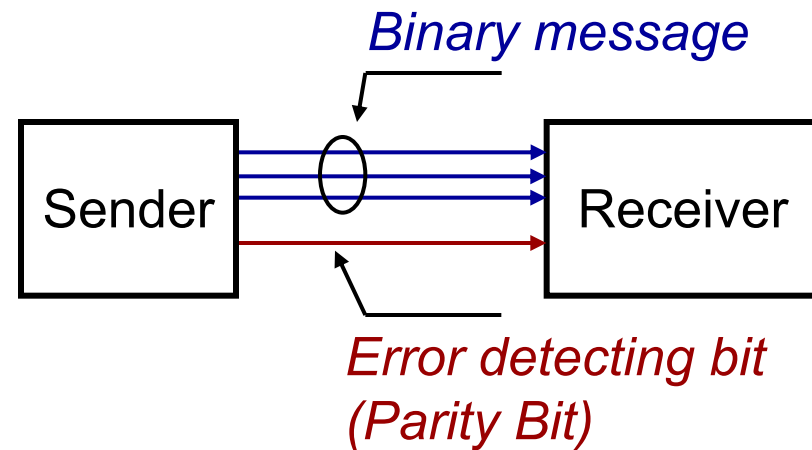
Unicode

- **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*

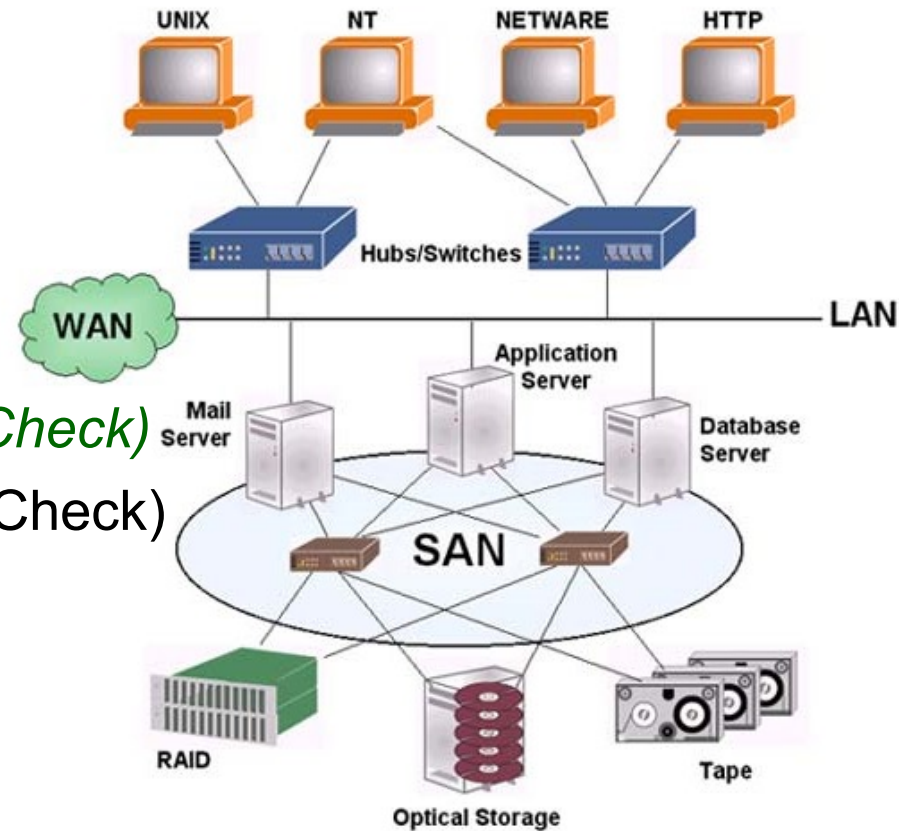
- Data transmission, ageing media, static interference, dust on media, etc, demand the ability to detect errors.
- Single bit errors can be detected by using **parity checking**.
 - Constructing a **single-error-detecting code with 2^n code words**.
 - Requires **$n+1$** bits: the **first n bits** are 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.



Parity Checking (cont.)

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

Storage Area Networks



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!**



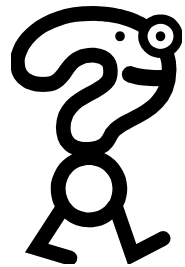
The **parity bit** can be added either at the beginning **or** at the end of the value.

Example: Odd and Even Parity

What is actually sent when even parity has been agreed.

	'S'	'E'
ASCII	101 0011	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 **1010011₂**), but value **01010010₂** is received.
 - Need to add a *1-bit parity* to make an odd or even number of bits per byte.



Assuming value **01010010₂** 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?