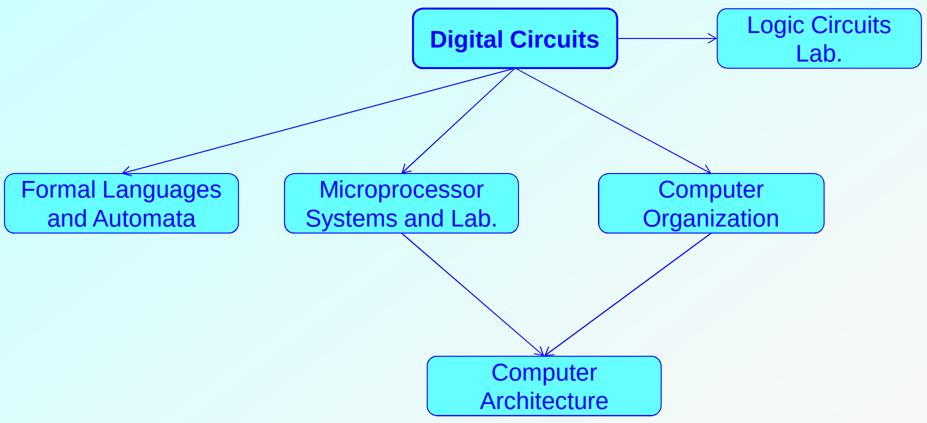
# DIGITAL CIRCUITS

**Dr. Feza BUZLUCA**
**Istanbul Technical University**
**Computer Engineering Department**
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

## Connection between the courses



## Where are the digital circuits used?

Today, almost each electronic device is implemented as a digital circuit.

Examples:

- The central processing unit (CPU): It is a synchronous sequential circuit.
  We will see this type of digital circuits in the second part of this course.
- Memories of computers: We will see flip-flops and latches which are digital circuits and building blocks of memories.
- Home electronics: TV, control unit of the wash machine, video and audio devices.
- Cars: ABS, engine control
- Cell phones

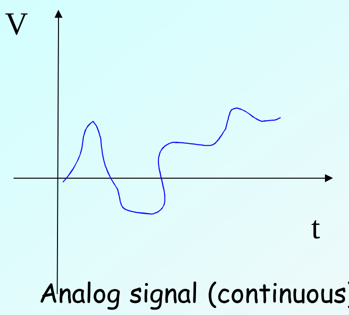
## Analog – Digital Signals:

In the real world, many physical quantities (current, voltage, temperature, light intensity, etc.) vary within a continuous range.

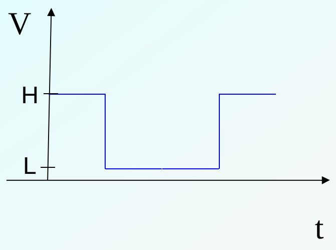
This kind of signals, which can get any possible value between the limits are called as **analog signals**.

An analog signal has a theoretically infinite resolution.

**Binary digital signals**, can take at a certain moment only one of two possible values: 0 - 1, high - low, true- false, open - closed.



Analog signal (continuous)



Binary Digital Signal (discrete)

## Advantages of Digital Systems:

Because of their advantages, today digital systems are used in many areas instead of the old analog systems.

**Examples:** Photography, video, audio, automobile engines, telephone systems and so on.

**Advantages of Digital Systems:**

• Mathematics of digital design is simpler than the mathematics of the analogue systems (Boolean algebra).

• Digital systems are easier to test and debug.

• Flexibility and programmability. Today, digital systems can be implement in the form of programmable computers (embedded system).

• In this way, devices can be re-programmed according to new requirements without changing the hardware.

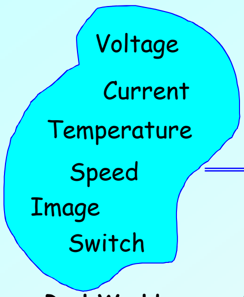
• Digital data can be stored and processed in computer systems.

• Digital systems work faster.

• Digital systems are growing smaller and getting cheaper.

• Digital systems continue to evolve.

## Steps of Digital System Implementation:



**Real World (Physical world, Problem Domain)**

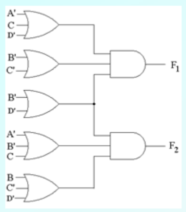
→

0, 1

and, or not

**Mathematical World Model**
(Boolean Algebra)

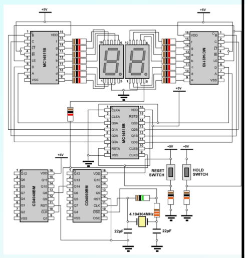
→

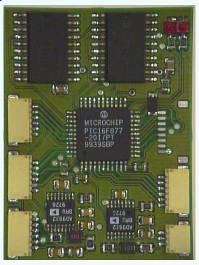


HDL Program
(*Hardware Description Language*)

```
WHEN(A & B) THEN X1-C;
ELSE X2 – E # F;
X2 – (!A # C);
......
```

→



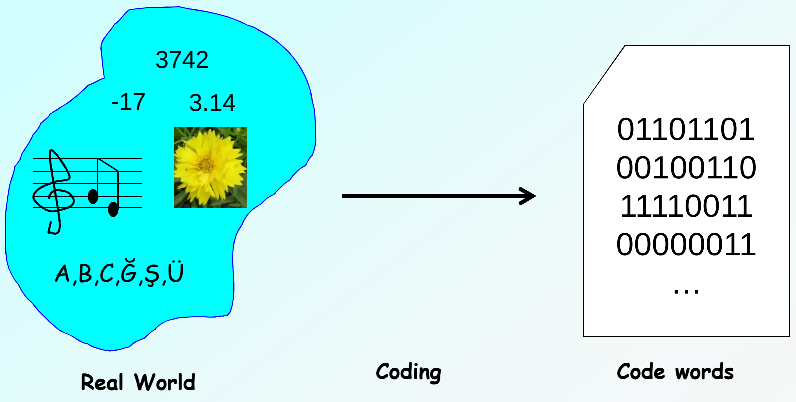


**Abstraction, Modeling** **Design** **Implementation**

## Binary Digital Coding:

As digital systems operate on binary digital signals, they can process only two different values (binary data).

Therefore, physical quantities (voltage, temperature, etc.), and any kind of data (letter, number, color, sound) must be converted to binary numbers (digitally coded) before they can be processed by digital circuits.



## Digital Coding (cont'd):

Using n bits (*binary digit*) $2^n$ different "things" can be represented.

For example, an 8 bit (*binary digit*) long binary number can represent $2^8$ (256) different "things".

These can be 256 different colors, 256 symbols, integers between 0 and 255, integers from 1 to 256, or integers between -128 and +127.

There are different coding systems (methods) for different type of data.

The meaning of a binary value (for example, 10001101) is determined by the system (hardware or software) that process this number.

This value may represent a number, a color, or another type of data.

Especially the coding of numbers is important.

Therefore in this course we will give some basic information about the coding methods of numbers.

## BCD (Binary Coded Decimal) :

Each decimal number between 0-9 is represented by a four bit pattern.

**Natural BCD:**

| Number: | BCD Code: | Number: | BCD Code: |
|---|---|---|---|
| 0: | 0000 | 5: | 0101 |
| 1: | 0001 | 6: | 0110 |
| 2: | 0010 | 7: | 0111 |
| 3: | 0011 | 8: | 1000 |
| 4: | 0100 | 9: | 1001 |

**Example:**
Number: 805
BCD:1000 0000 0101

It is a **redundant code**. Because by using 4 bits 16 different code values can be created, but only 10 of them are used.

It is difficult to perform arithmetical operations on BCD numbers.

Therefore todays computer systems do not use the BCD coding to represent the numbers.

## Positional (weighted) Coding:

Each digit of the number has an associated weight.

**Natural Binary Coding:** Binary (2) radix (base) is used to represent numbers with positional (weighted) coding.

Example: 11010 = $1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 26$

The leftmost bit is called the *Most Significant Bit – MSB (high-order*).

The rightmost bit is called the *Least Significant Bit – LSB (low-order*).

In today's computers natural binary coding is used to represent numbers.

**Hamming distance:** The Hamming distance between two n-bit long code word is the number of bit positions in which they are different.

Example: Hamming distance between 011 and 101 is 2.

*Richard Wesley Hamming (1915-1998) Mathematician, USA*

**Adjacent Codes:** Between each pair of successive code words Hamming distance is 1 (only one bit changes).

In addition, if the distance between the first and last code word is 1 then this code is called circular.

Example: A circular BCD code (different from the natural BCD)

| Number: | Code: | Number: | Code: |
|---|---|---|---|
| 0: | 0000 | 5: | 1110 |
| 1: | 0001 | 6: | 1010 |
| 2: | 0011 | 7: | 1000 |
| 3: | 0010 | 8: | 1100 |
| 4: | 0110 | 9: | 0100 |

**Gray Code:** A a **binary** (base 2), **non-redundant** and **circular** (also adjacent) coding system that represents $2^n$ elements is called as a **Gray code**.

Example: A 2 bit Gray code:

| Num.: | Code: |
|---|---|
| 0: | 00 |
| 1: | 01 |
| 2: | 11 |
| 3: | 10 |

The patent of the Gray Code has been taken by physicist *Frank Gray* in 1953 as he worked in Bell Labs.

## Representation of Numbers in Digital Systems (and Computers)

In this course we will deal with *integer* s.

Representation of the *floating point* numbers will be showed in the Computer Architecture course (See. http://www.buzluca.info/mimari).

Before coding the numbers we have to decide to work with **unsigned** or **signed** numbers.

Because unsigned and signed numbers are encoded and represented differently.

### Representation of Unsigned Numbers:

Unsigned integers are represented in computers by "natural binary weighted (positional) coding".

Example: $215_{10} = (1101\ 0111)_2 = 1{\cdot}2^7 + 1{\cdot}2^6 + 0{\cdot}2^5 + 1{\cdot}2^4 + 0{\cdot}2^3 + 1{\cdot}2^2 + 1{\cdot}2^1 + 1{\cdot}2^0$

215/2 = 107 remainder 1 (low-order bit "*Least Significant Bit – LSB*") last (right-most) bit
107/2 = 53 remainder 1
53/2 = 26 remainder 1
26/2 = 13 remainder 0
13/2 = 6 remainder 1
6/2 = 3 remainder 0
3/2 = 1 remainder 1
1/2 = 0 remainder 1 (high-order bit "*Most Significant Bit – MSB*") first (left-most) bit

The **greatest unsigned** number that can be represented with 8 bits: $1111\ 1111_2 = 255_{10}$

The **smallest unsigned** number that can be represented with 8 bits: $0000\ 0000_2 = 0_{10}$

## Representation of Signed Numbers:

The (high-order) most significant bit denotes the sign of the number .

Positive numbers start with a “0”,

Negative numbers start with a “1”.

**Positive** numbers are represented (like unsigned numbers) in computers by "natural binary weighted (positional) coding”.

Remember, the positive binary number must start with 0.

The range of **positive signed** numbers that can be represented with 8 bits:

Between 0000 0000 and 0111 1111 (Decimal: between 0 and +127)

**Negative** numbers are represented by **2's complement** system.

In this system negative numbers are represented by the two's complement of the positive number (absolute value).

Getting the 2’s complement:

- First invert (1's complement) the number. Change 0 to 1, 1 to 0.
- Then add 1 to the inverted number.

2's complement system makes it easy to add or subtract two numbers without sign and magnitude checks.

## Examples for Negative Numbers:

| | | | |
|---|---|---|---|
| 8-bit $+5_{10}$ | : 0000 0101 | 4-bit $+7_{10}$ | : 0111 |
| 1's complement | : 1111 1010 | 1's complement | : 1000 |
| Add 1 | : + 1 | Add 1 | : + 1 |
| Result $-5_{10}$ | : 1111 1011 | Result $-7_{10}$ | : 1001 |

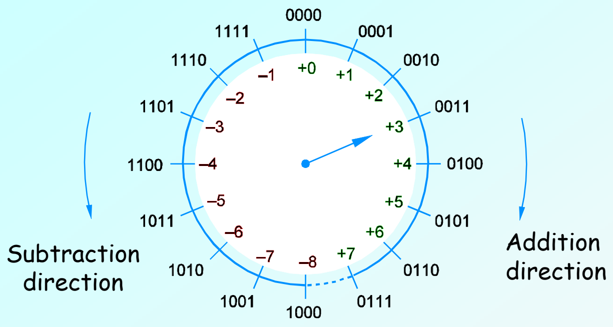
2's complement operation **changes the sign** of a number.

Applying 2's complement operation to a negative number makes it positive

Example: Converting a negative number to positive :

| | |
|---|---|
| 8-bit $-5_{10}$ | : 1111 1011 |
| 1's complement | : 0000 0100 |
| Add 1 | : + 1 |
| Result $+5_{10}$ | : 0000 0101 |

Signed numbers can be presented on a circular graphic.
4-bit numbers:



The 4 bits long negative number with the greatest absolute value: 1000 = -8

The 4 bits long negative number with the smallest absolute value: 1111 = -1

The 8 bits long negative number with the greatest absolute value:
1000 0000 = -128

The 8 bits long negative number with the smallest absolute value:
1111 1111 = -1

## Extension (Sign Extension) of Binary Numbers

In digital systems certain number of bits (memory locations) are allocated for binary numbers.

In some cases it is necessary to write a number to a memory space with more bits than necessary. For example 8-bit number to 16-bit memory.

Sometimes it is necessary to perform an operation between two numbers with different lengths.

In such cases the short number is extended (word length is increased).

For example; extension from 4 bits to 8 bits or from 8 bits to 16 bits.

Extension operation is different for unsigned and signed numbers.

**Unsigned Numbers:** The high-order part of the binary number is filled with "0"s.

Example: 4 bit $3_{10}$: 0011 8 bit $3_{10}$: 0000 0011
Example: 4 bit $9_{10}$: 1001 8 bit $9_{10}$: 0000 1001

**Signed Numbers:** The high-order part of the binary number is filled with the value of the sign bit. This operation is called **sign extension**.

Example: 4 bit $+3_{10}$: 0011 8 bit $+3_{10}$: 0000 0011
Example: 4 bit $-7_{10}$: 1001 8 bit $-7_{10}$: 1111 1001
Example: 4 bit $-1_{10}$: 1111 8 bit $-1_{10}$ : 1111 1111

## Hexadecimal (base 16) Numbers

Binary numbers are necessary, because digital circuits can directly process binary values.

But it is difficult for humans to work with the large number of bits for even a relatively small binary number. (Example: 1110010001011010)

Therefore, for documentation (to write and read easily) *hexadecimal* (base 16) numbers are used.

Converting from binary to hexadecimal is easy:

- Each hex. digit maps to 4 bits. See the table.
- Separate the bits of the binary into groups of 4 bits
- Map each group to a single hexadecimal digit.

**Example:**

$01011101_2$ = 0101 1101 (Binary)
= 5 D (Hexadecimal)

Converting to base 10:

$5D_{16} = (5 \times 16) + 13 = 93$

Result: $01011101_2 = 5D_{16} = 93_{10}$

| Decimal | Binary | Hex. |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |

To express hexadecimal numbers usually the symbols **$** and **h** are used.
Example: $5D or 5Dh.

## Addition and Subtraction Operations in Digital Systems

In computers integer arithmetical operations are performed by the *Arithmetic Logic Unit* – ALU.

Integer addition and subtraction operations are performed on unsigned and signed numbers in the same way (because of 2's complement representation).

But the interpretation of the result is different for unsigned and signed numbers.

Before the operation between the numbers with different lengths sign extension is necessary.

Extension operation is different for unsigned and signed numbers (see 1.13).

**Addition:**

<u>Unsigned Integers:</u>

The result of the addition of two n-bit unsigned numbers can be a (n+1)-bit number. (n+1)th bit is called the **carry**.

| a b | Carry | Result |
|---|---|---|
| 0 0 | 0 | 0 |
| 0 1 | 0 | 1 |
| 1 0 | 0 | 1 |
| 1 1 | 1 | 0 |

**Example:** Addition of two 8-bit unsigned numbers

```
   01110101: 117
+  01100011:  99
   -------------
   11011000: 216
```

No Carry (Carry=0)

```
   11111111: 255
+  00000001:   1
   -------------
  100000000: 256
```

(Carry=1)

## Addition:

Signed Integers:

- The operation is performed as with unsigned numbers, but the result is interpreted differently.
- Adding two's-complement numbers requires no special processing even if the operands have opposite signs.
- If (n+1)th bit arises by adding two n-bit signed numbers, this bit is ignored.

**Example:** Addition of 8 -bit signed numbers

```
  11111111:   -1            11111111:   -1
+ 00000001:   +1          + 11111111:   -1
 100000000:    0           111111110:   -2

 Ignored     Sign (+)      Ignored     Sign (-)
```

**Attention:**
While working with n-bit numbers the sign bit is always the most significant bit (counting from right to left) the nth bit, not the (n+1)th one (it is carry bit).

## Overflow (signed integers):

The result of addition of n-bit signed numbers can be too large for the binary system to represent (greater than n bits).

For example, with 8 bits we can represent signed numbers between -128and +127. If the result is out of this range an **overflow** occurs.

Existence of the overflow after addition can be detected by checking the signs of the operands and the result.

In an addition operation overflow can occur in two cases:

pos + pos $\rightarrow$ neg and neg + neg $\rightarrow$ pos

**Examples:**

```
  01111111:+127                    10000000:-128
+ 00000010:  +2                  + 11111111:  -1
  10000001: Can not be represented  101111111:Can not be represented
```

Both operands are positive.
Result is negative.
There is an **overflow**.
Note: (n+1)th bit is zero.
It is ignored.

Both operands are negative.
Result is positive.
There is an **overflow**.
Note: (n+1)th bit is one.
It is ignored.

## Subtraction:

- Computers usually use the method of complements to implement subtraction.
- 2's complement of the second operand is added with the first number.

So, only one addition circuit is sufficient to perform both addition and subtraction.

Unsigned Integers:

If the result of the subtraction of two n-bit unsigned numbers, performed by 2's complement method, is a (n+1)-bit number, then there is not a barrow and the result is valid.

If the (n+1)th bit of the result is zero, the first operand is smaller than the second and there is **borrow**.

**Example:** Subtraction of 8-bit unsigned number subtraction

```
   00000101: 5                        00000101: 5
 - 00000001: 1   2's complement →   + 11111111:-1
                                     100000100: 4
                                   Carry=1 : No Borrow
```

```
   00000001: 1                        00000001: 1
 - 00000101: 5   2's complement →   + 11111011:-5
                                     011111100:  Can not be presented
                                   No Carry: Borrow
```

## Subtraction:

Signed Integers:

Subtraction on signed integers is also performed by using 2's complement method.

Carry bit is ignored.

Like in addition also while subtracting signed numbers an overflow can occur.

In subtraction overflow can occur in two cases:

pos - neg $\rightarrow$ neg and neg - pos $\rightarrow$ pos

**Examples:** Subtraction of 8-bitlik signed numbers

```
  00000101: 5                        00000101:   5
- 00001100:12   2's complement →   + 11110100:-12
                                     11111001: 2's comp.:00000111:-7
                                   Sign 1, result: negative
```

```
  11111101:  -3                       11111101:   -3
- 01111111: 127   2's complement →  + 10000001:-127
                                     101111110: can not be presented
                                    Sign:0, result: positive.
```

Neg – pos = pos. **Overflow**.

## Summary of Carry, Borrow, Overflow

**Carry:** It can occur in addition of unsigned numbers.

It shows that the result can not be represented with n bits, and (n+1)th bit is necessary.

**Borrow:** It can occur in subtraction of unsigned numbers.

It shows that first number is smaller than the second one.

The result can not be represented with unsigned numbers.

If the result of the subtraction performed by 2's complement method, is a (n+1)-bit number, then there is not a barrow and the result is valid.

**Overflow:** It can occur in addition and subtraction operations only on signed numbers.

It shows that the result can not be represented with n bits.

Overflow can be detected by checking the signs of operands and the result.

There is an overflow in the following cases:

| | |
|---|---|
| pos + pos $\rightarrow$ neg | pos – neg $\rightarrow$ neg |
| neg + neg $\rightarrow$ pos | neg – pos $\rightarrow$ pos |