Lecture
0_2.3

# From Boolean Functions to Digital Circuits


**Paolo PRINETTO**
**Politecnico di Torino (Italy)**
**University of Illinois at Chicago, IL (USA)**

Paolo.Prinetto@polito.it
prinetto@uic.edu

www.testgroup.polito.it
www.comitato-girotondo.org

# License Information

**This work is licensed under the Creative Commons BY-NC License**


To view a copy of the license, visit:
http://creativecommons.org/licenses/by-nc/3.0/legalcode


Rel. 14.06.2018

# Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.


Rel. 14.06.2018

# Goal

- This lecture presents a global overview of how a Boolean Function can be mapped into (implemented by) a digital circuit

# Prerequisites

- Lecture 0_2.2

# Homework

- None

# Outline

- Geometric interpretation of Boolean Algebras
- Values taken by functions
- How representing functions?
- Problems to face
- From functions to circuits
- Remarks
- Lessons learned


Rel. 14.06.2018

# Geometric interpretation

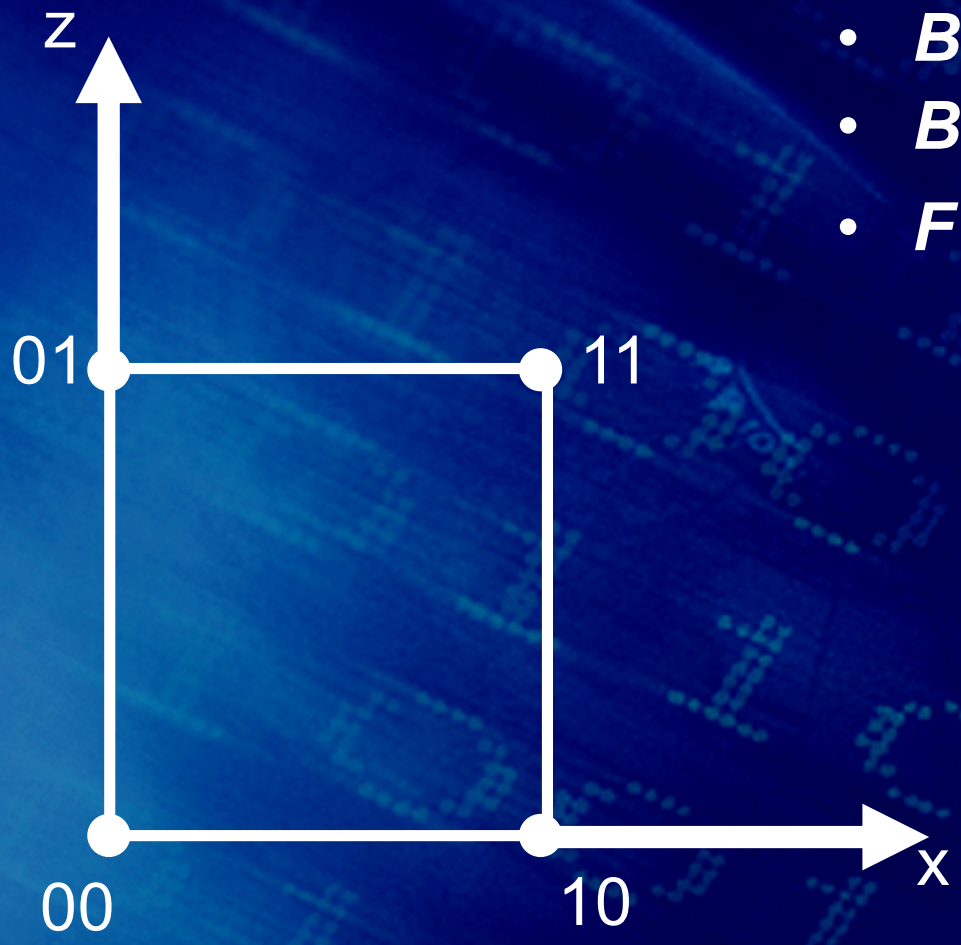
The carrier $B^n$ of a Boolean Algebra can be seen as an *n-dimensional space*, where each generic element $v \in B^n$ (usually called a *vertex*), is represented by a vector of n coordinates, each $\in B$


Rel. 14.06.2018

# *1-input functions*

- ***B*** **= { 0, 1}**
- ***B***
- ***F*** **= F (x)**

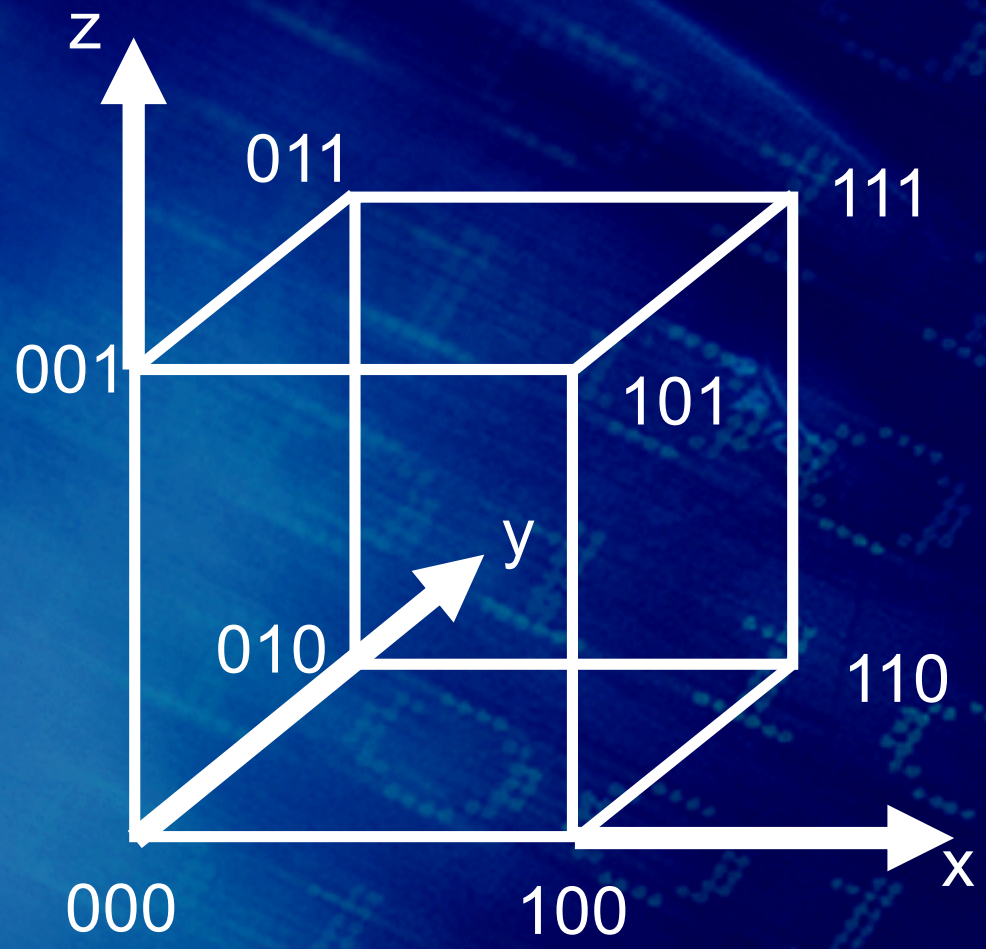
# 2-input functions




- $B = \{ 0, 1 \}$
- $B^2$
- $F = F(x, z)$


Rel. 14.06.2018

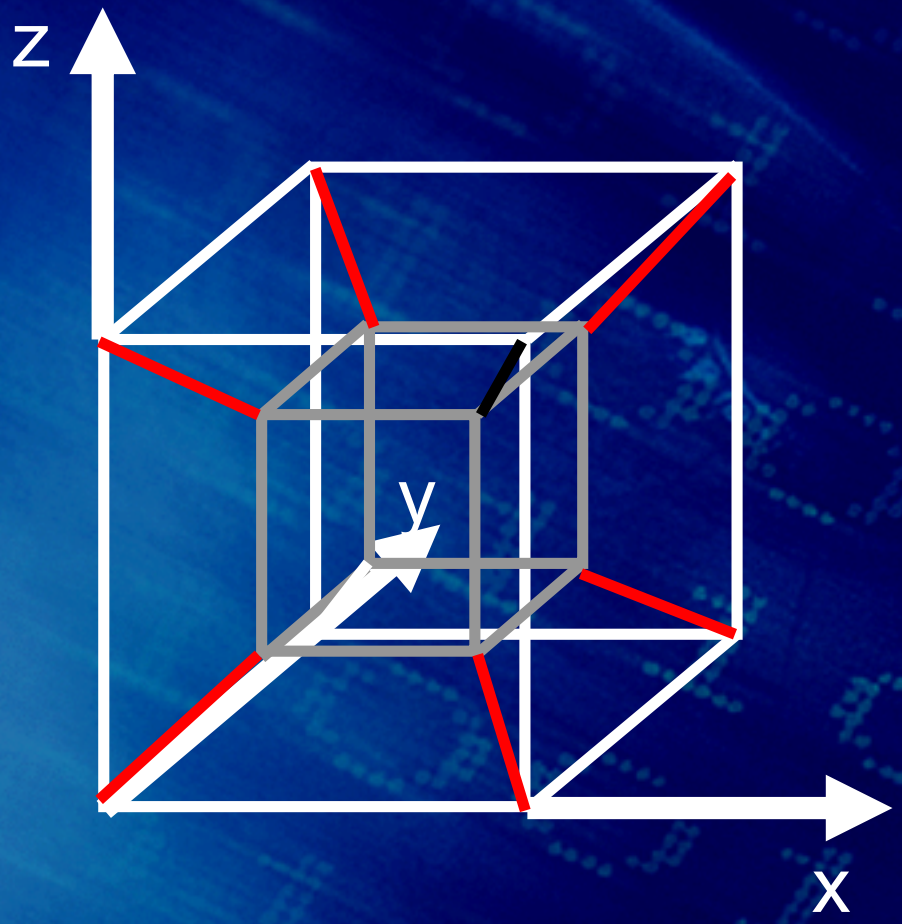
# 3-input functions




- $B = \{ 0, 1 \}$
- $B^3$
- $F = F(x, y, z)$


Rel. 14.06.2018

# *4-input functions*




- $B = \{ 0, 1 \}$
- $B^4$
- $F = F(x, y, z, w)$

W



Rel. 14.06.2018

# Outline

- Boolean Algebras Definitions
- Examples of Boolean Algebras
- Geometric interpretation of Boolean Algebras
- **Values taken by functions**
- How representing functions?
- Boolean Algebras properties

# *Values taken by functions*

– **In general, in each vertex V, a function may:**

. ***get the value 0*** $\Rightarrow$ **V** $\in$ ***off-set***

. ***get the value 1*** $\Rightarrow$ **V** $\in$ ***on-set***

. ***be not specified*** $\Rightarrow$ **V** $\in$ ***don't-care set***



Rel. 14.06.2018

# *Values taken by functions*

– In general, in each vertex V, a function may:

. *get the value 0* ⇒ V ∈ *off-set*

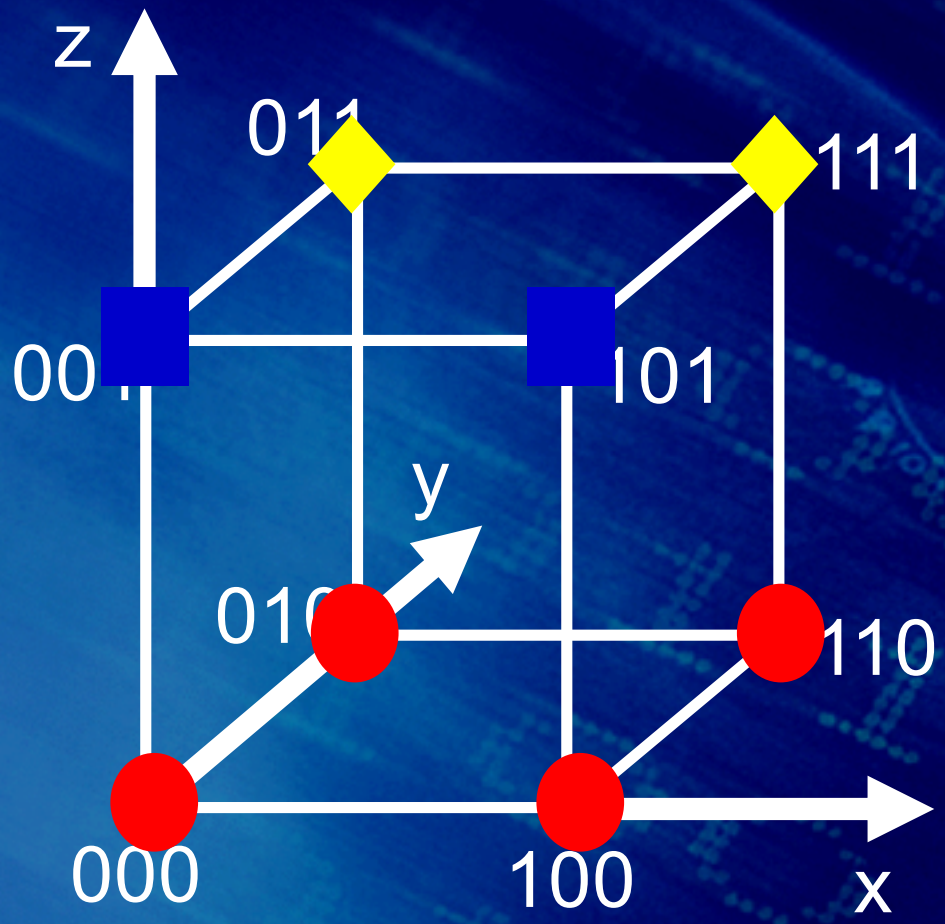
. *get the value 1* ⇒ V ∈ *on-set*

. *be not specified* ⇒ V ∈ *don't-care set*

**Typically used to represent combinations of input variables not relevant in a given design**



Rel. 14.06.2018




↔ 0

↔ 1

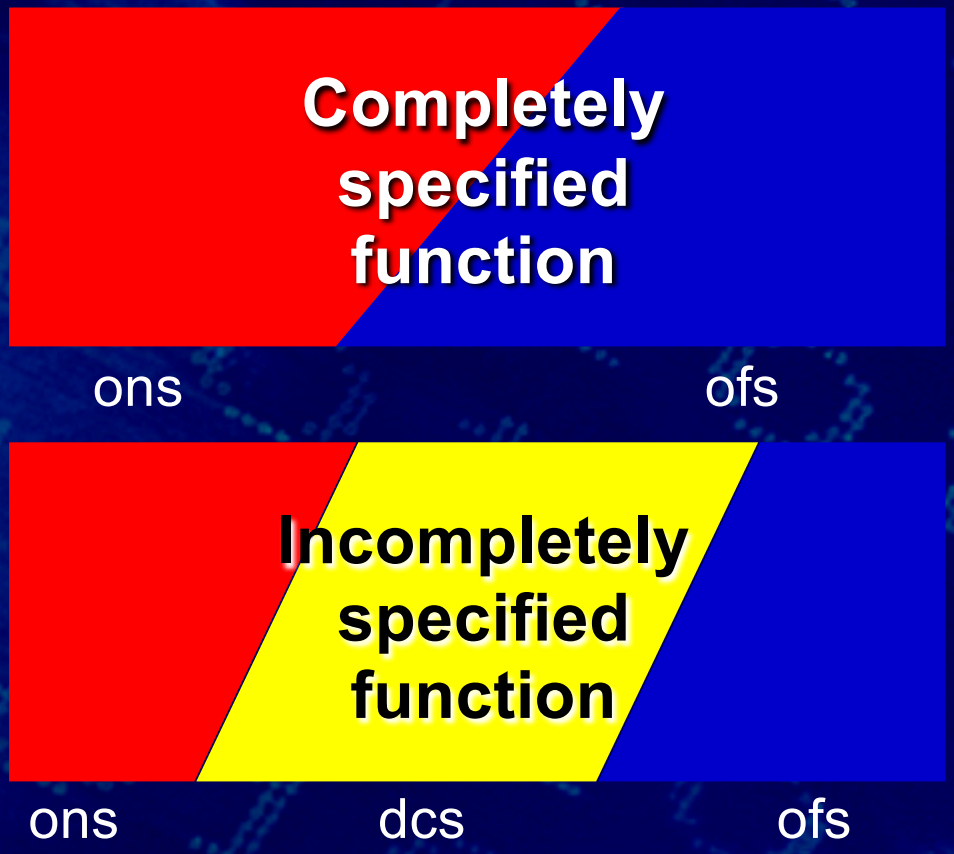
↔ –



Rel. 14.06.2018

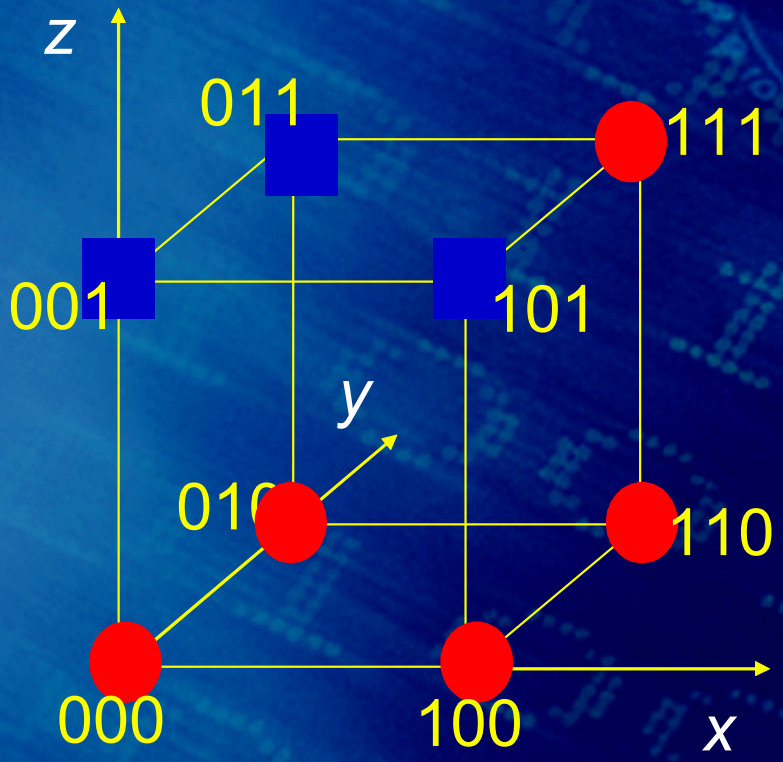
# *Completely vs. incompletely specified functions*

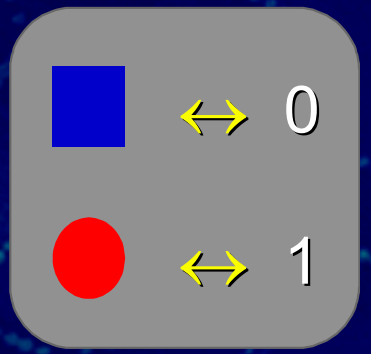
- **A function f is *completely specified* iff dcs ( f ) = ∅**
- **f is *incompletely specified* otherwise**



# Note

- For sake of simplicity, without loosing generality, in the sequel of the lecture we shall use, as a case study, the following completely specified 3-input function ***F = F (x, y, z)*** :





Rel. 14.06.2018

# Outline

- Boolean Algebras Definitions
- Examples of Boolean Algebras
- Geometric interpretation of Boolean Algebras
- Values taken by functions
- How representing functions?
- Boolean Algebras properties

# *How representing functions?*

- **We need a way "*to represent*" or "*to express*" the value that the function F gets in each vertex, i.e., in each point of its domain.**


Rel. 14.06.2018

# Boolean Functions representation

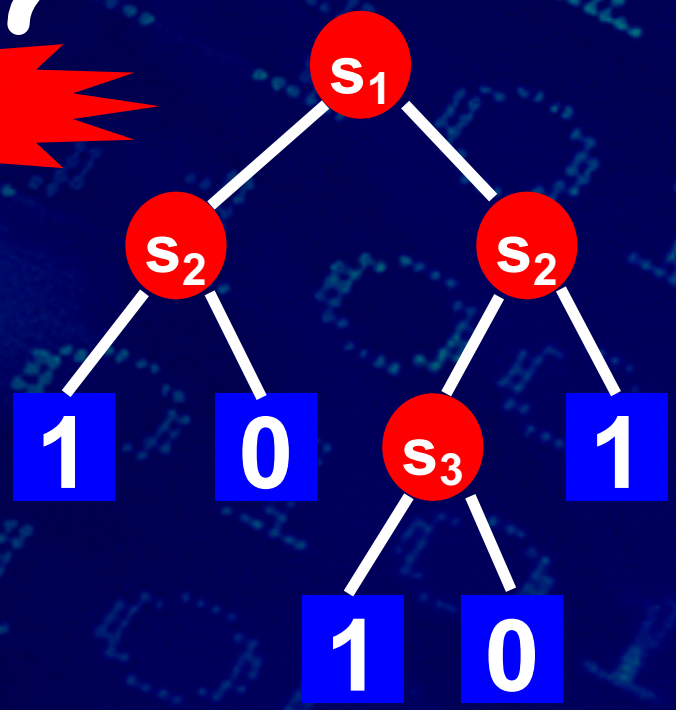
- Boolean Functions are mostly represented by stating their on-set and don't-care set.
- These can, in turn, be represented according to several criteria, each particularly suited for a target application.


Rel. 14.06.2018

# An example


$s_1' s_2' + s_1 s_3' + s_1 s_2$

function *f*




```
.i 3
.o 1
00- 1
1-0 1
11- 1
.e
```

# Boolean Functions representation

Exhaustive Representations

Compact Representations

ITE Expressions

Graphs

# Boolean Functions representation

Exhaustive Representations

Compact Representations

ITE Expressions

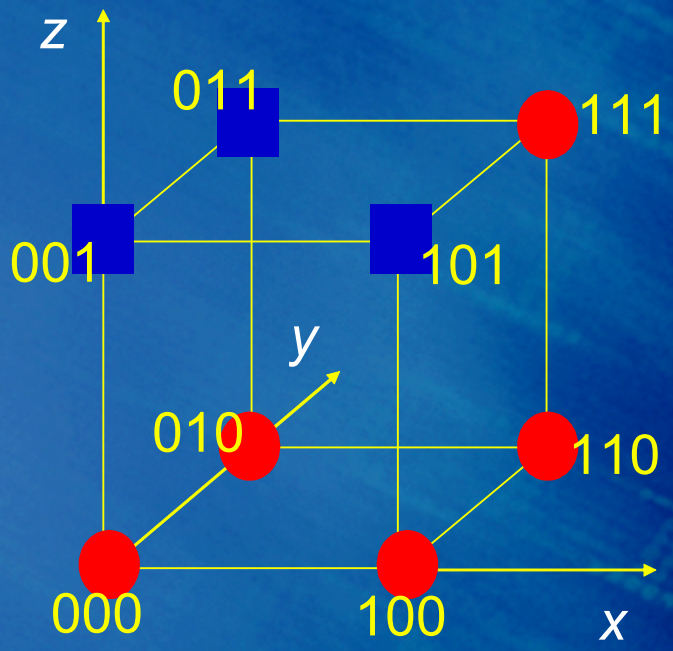
Graphs

# An exhaustive approach

- We could explicitly represent the values assumed by the function for each of the possible combinations of the input variables ⇒ $2^n$ entries !!!
- Two alternatives:
  - Tabular representation: *Truth Table*
  - Matrix representation: *Karnaugh maps*

# Truth table

- **It specifies the value the function gets for each input combination:**





| x y z | F |
|---|---|
| 0 0 0 | 1 |
| 0 0 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 0 |
| 1 0 0 | 1 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 1 |


Rel. 14.06.2018

# Usage

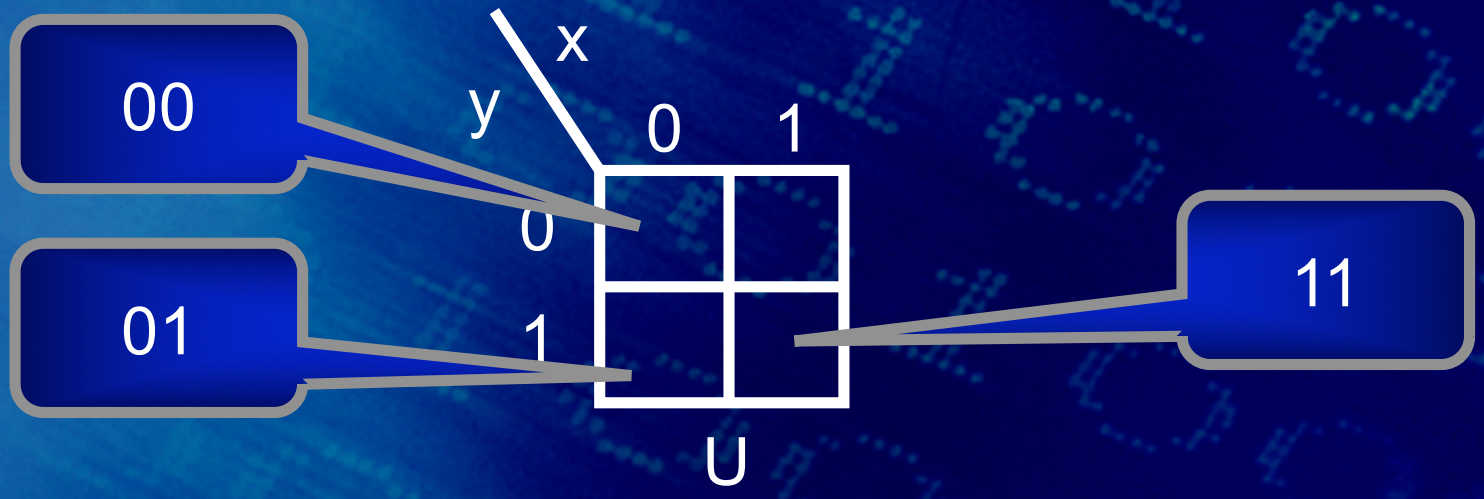
- When dealing with few inputs, they can be used to prove theorems
- Sometimes helpful in manual design, as preliminary step to generate the equivalent Karnaugh maps

# *Karnaugh maps*

***Karnaugh maps*** **(*K-map* for short) were first introduced by Maurice Karnaugh in 1953.**

**An n input variable boolean function is represented by a matrix of $2^n$ cells.**

**Each cell specifies the value of the function when its input variables get the values of the corresponding row and column.**





Rel. 14.06.2018

# Karnaugh maps

| x y z | F |
|---|---|
| 0 0 0 | 1 |
| 0 0 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 0 |
| 1 0 0 | 1 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 1 |



Rel. 14.06.2018

# Boolean Functions representation

Exhaustive Representations

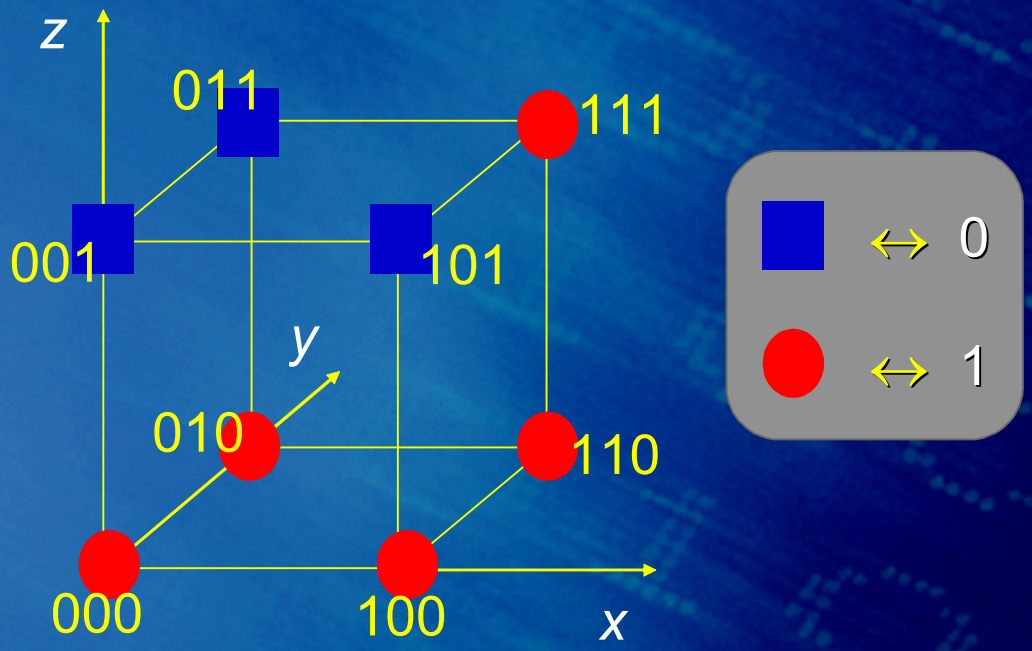
Compact Representations

ITE Expressions

Graphs

# An intuitive approach

- **We could, for instance, deciding of expressing only the vertices in which the function gets the value 1:**

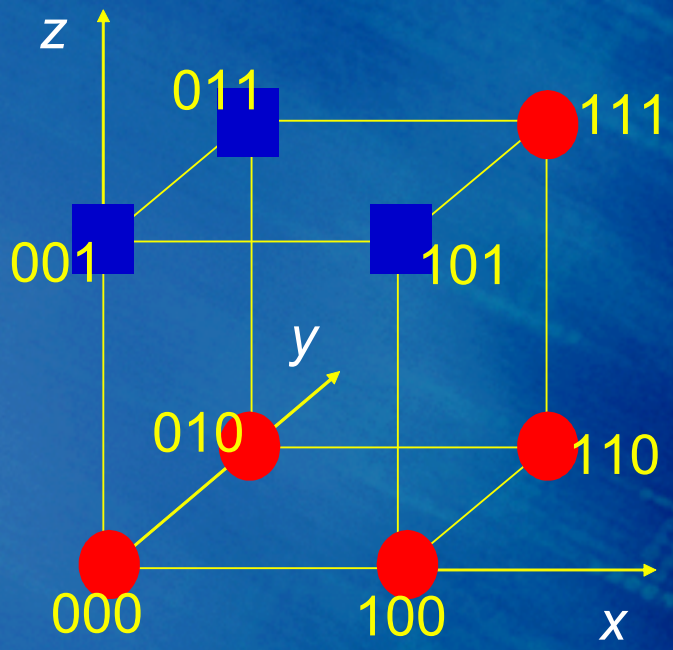




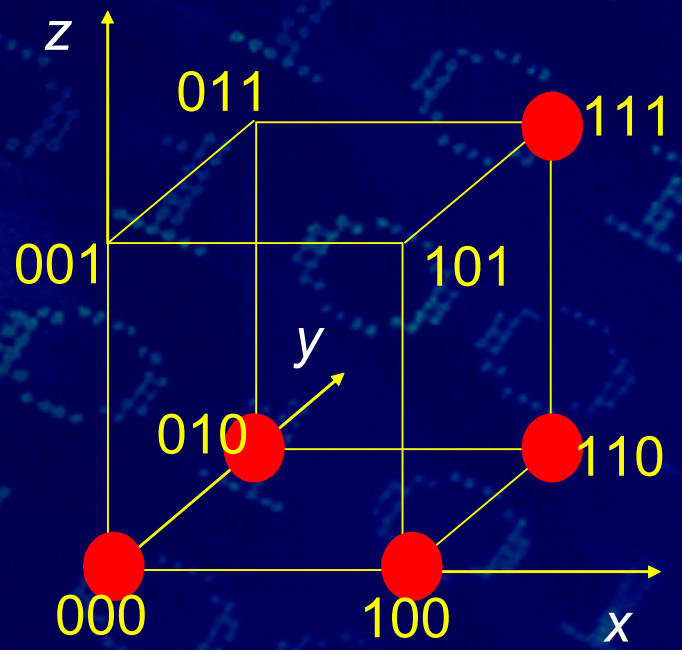
Rel. 14.06.2018

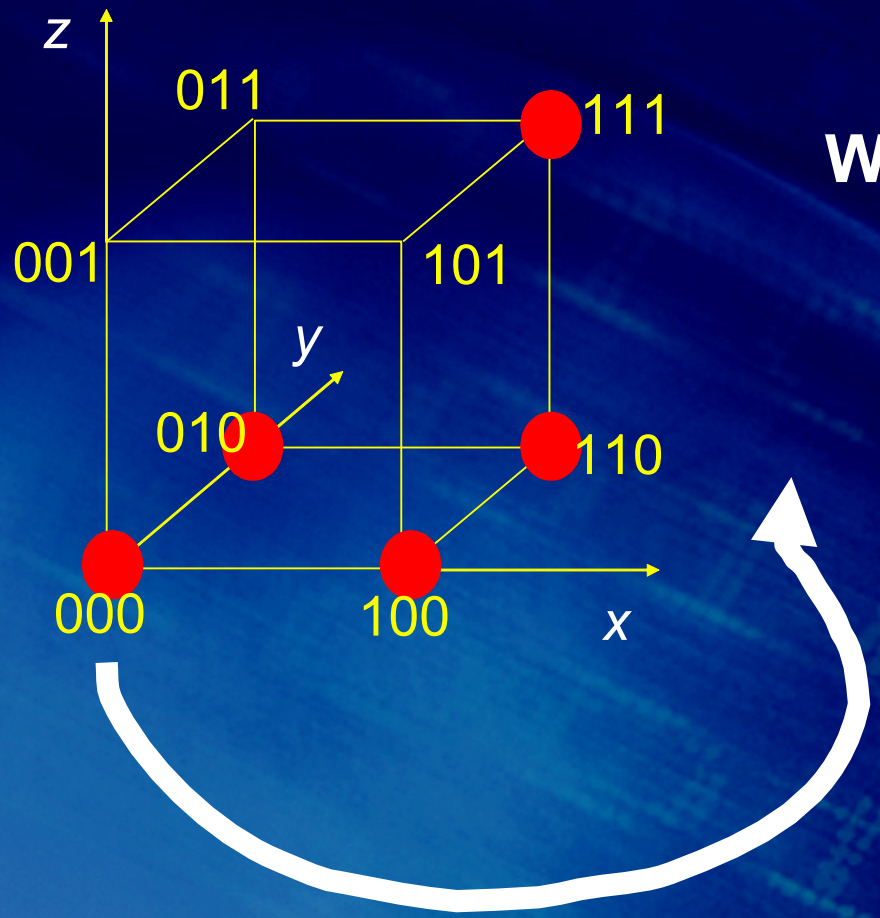
# *An intuitive approach*

**We could, for instance, deciding of expressing only the vertices in which the function gets the value 1:**





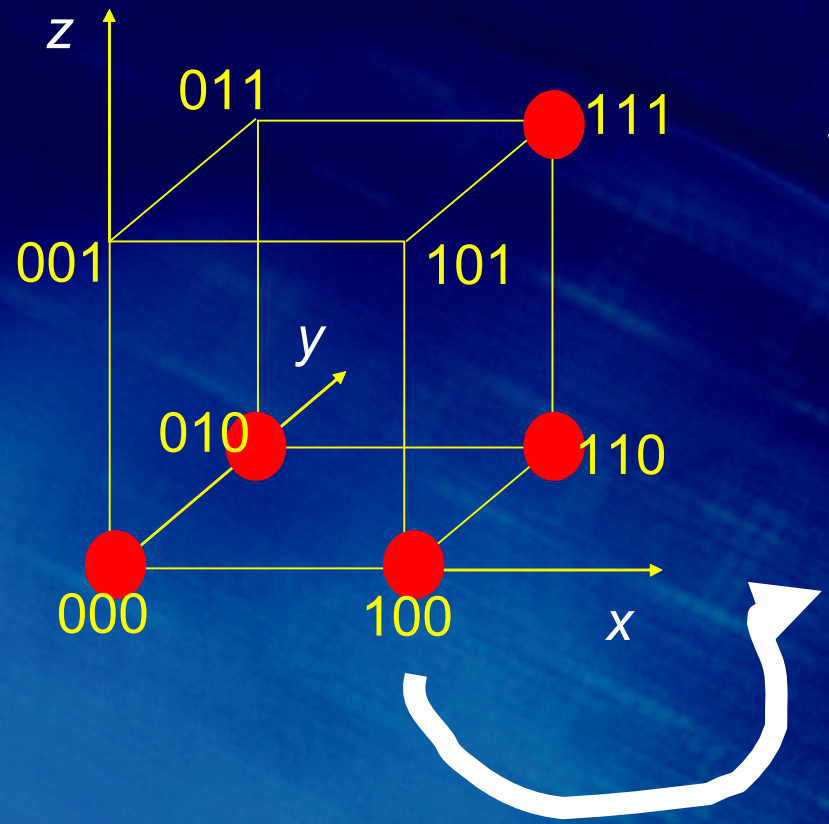





**We could write:**

**F = 1 when**

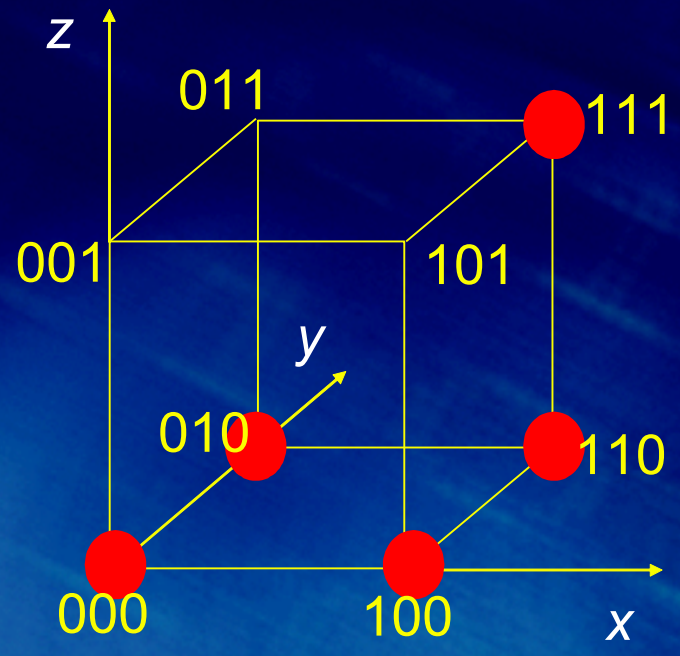
**( (x=0) and (y=0) and (z=0) )**




**We could write:**

**F = 1 when**

**( (x=0) and (y=0) and (z=0) )**
**or**
**( (x=1) and (y=0) and (z=0) )**

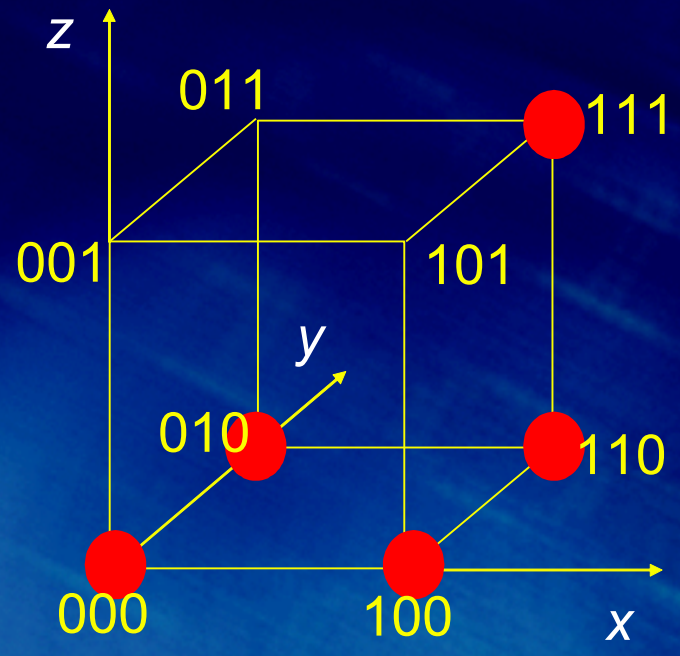


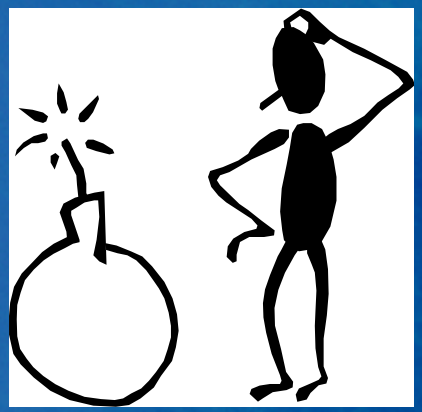

**We could write:**

**F = 1 when**

**( (x=0) and (y=0) and (z=0) )**
**or**
**( (x=1) and (y=0) and (z=0) )**
**or**
**( (x=1) and (y=1) and (z=0) )**
**or**
**( (x=0) and (y=1) and (z=0) )**
**or**
**( (x=1) and (y=1) and (z=1) ).**


Rel. 14.06.2018



**We could write:**

**F = 1 when**

**( (x=0) and (y=0) and (z=0) )**
**or**
**( (x=1) and (y=0) and (z=0) )**
**or**
**( (x=1) and (y=1) and (z=0) )**
**or**
**( (x=0) and (y=1) and (z=0) )**
**or**
**( (x=1) and (y=1) and (z=1) ).**


Rel. 14.06.2018

# *Let's try to simplify*

(x=1) → x
(x=0) → x'
and → · →
or → +


Rel. 14.06.2018

# *Let's try to simplify*

(x=1) → x
(x=0) → x'
and → · →
or → +

F = 1 when

( (x=0) and (y=0) and (z=0) ) or

( (x=1) and (y=0) and (z=0) ) or

( (x=1) and (y=1) and (z=0) ) or

( (x=0) and (y=1) and (z=0) ) or

( (x=1) and (y=1) and (z=1) )

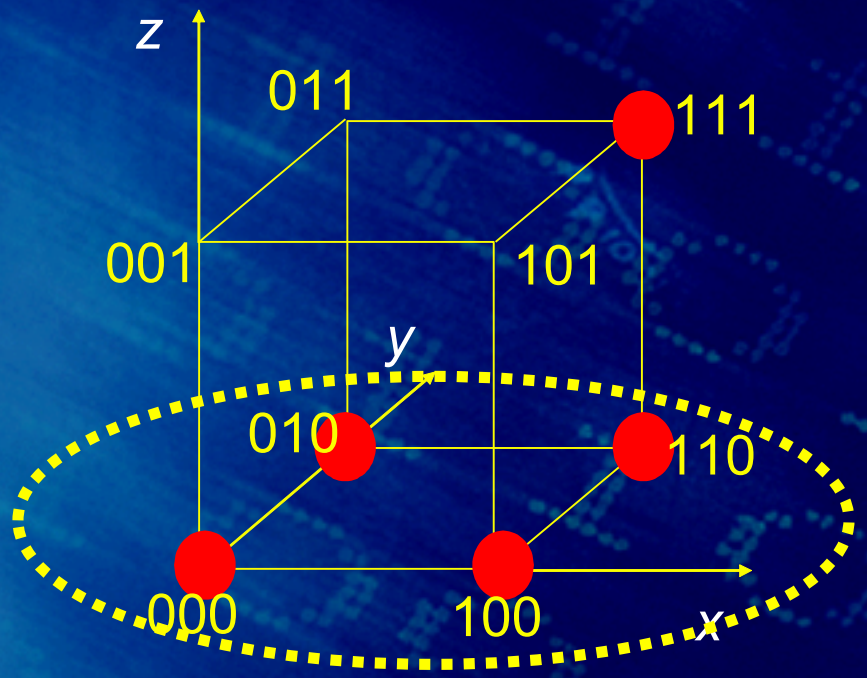
# *Let's try to simplify*

(x=1) → x
(x=0) → x'
and → · →
or → +

| | | |
|---|---|---|
| F = 1 when | | F = |
| ( (x=0) and (y=0) and (z=0) ) or | | x'y'z' + |
| ( (x=1) and (y=0) and (z=0) ) or | → | x y'z' + |
| ( (x=1) and (y=1) and (z=0) ) or | | x y z' + |
| ( (x=0) and (y=1) and (z=0) ) or | | x'y z' + |
| ( (x=1) and (y=1) and (z=1) ) | | x y z |

# *A further step*

**Instead of representing separately the 4 vertices**

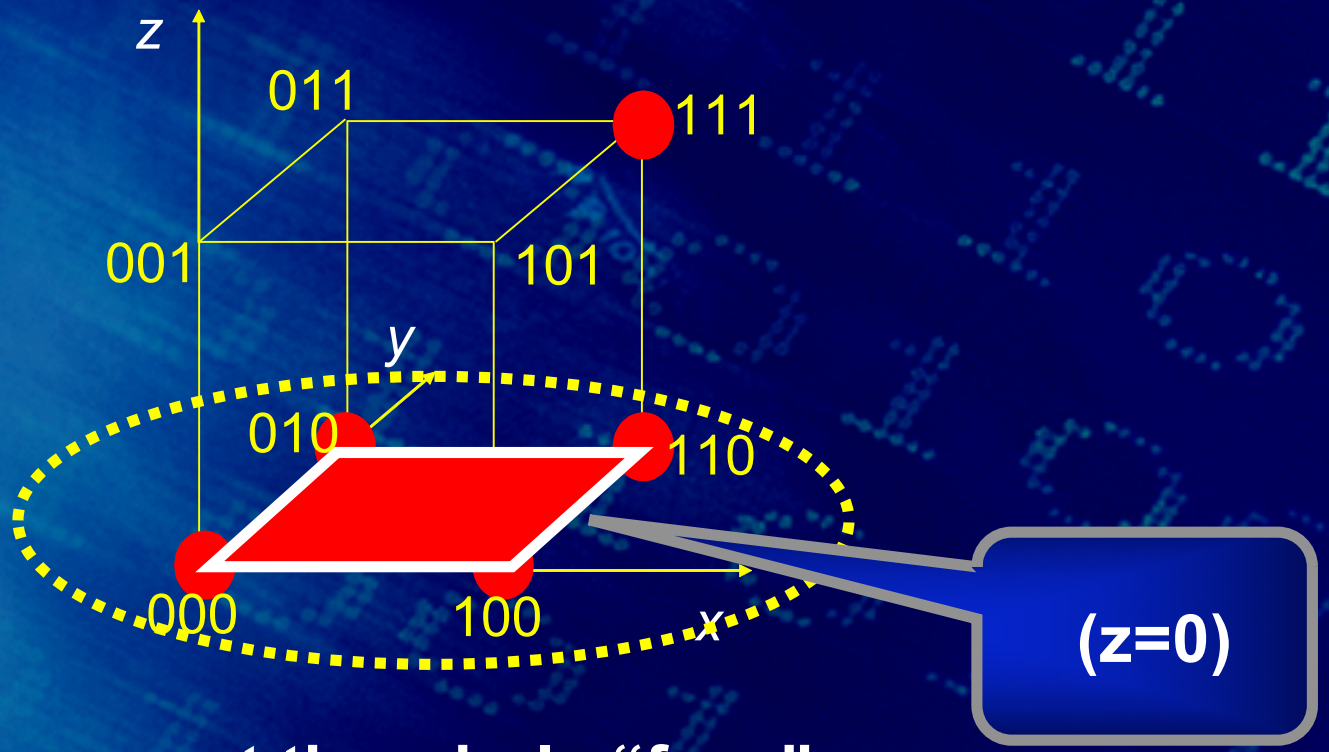



**we could represent the whole “face”**


Rel. 14.06.2018

# *A further step*

**Instead of representing separately the 4 vertices**

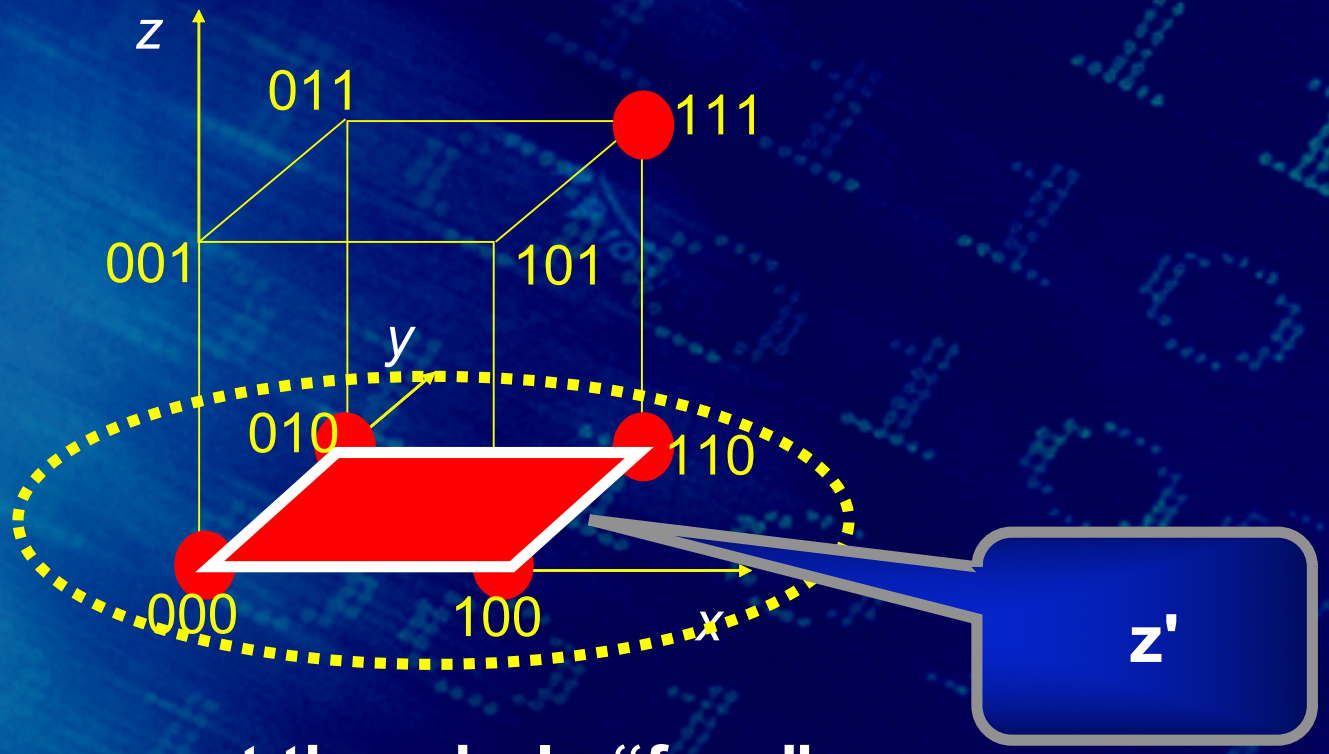



**we could represent the whole “face”**


Rel. 14.06.2018

# *A further step*

**Instead of representing separately the 4 vertices**

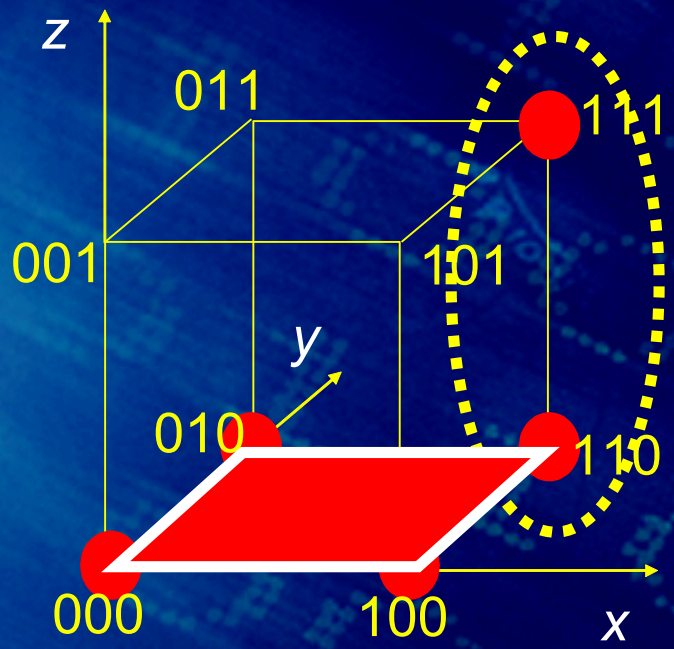



**we could represent the whole “face”**


Rel. 14.06.2018

# *A further step (3)*

**... and instead of representing separately the 2 vertices**

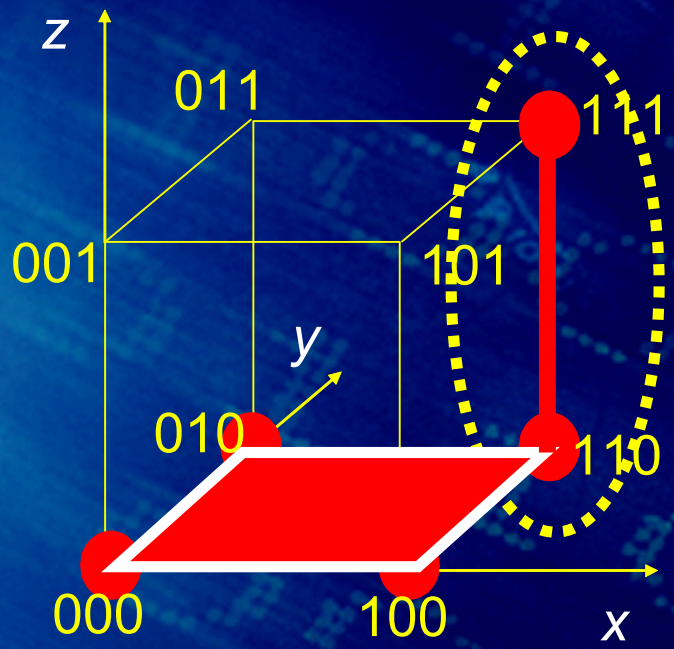



**we could represent the related "edge"**


Rel. 14.06.2018

# *A further step (3)*

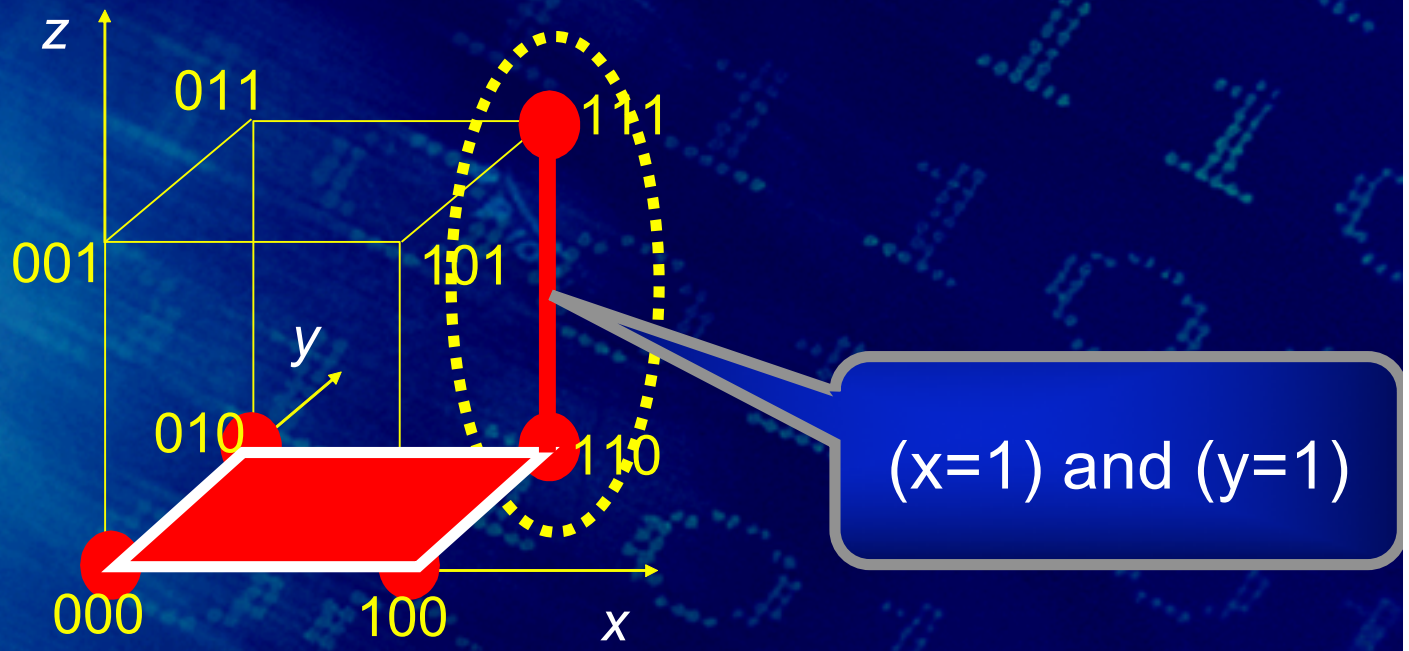
**... and instead of representing separately the 2 vertices**




**we could represent the related "edge"**

# A further step *(4)*

**... and instead of representing separately the 2 vertices**

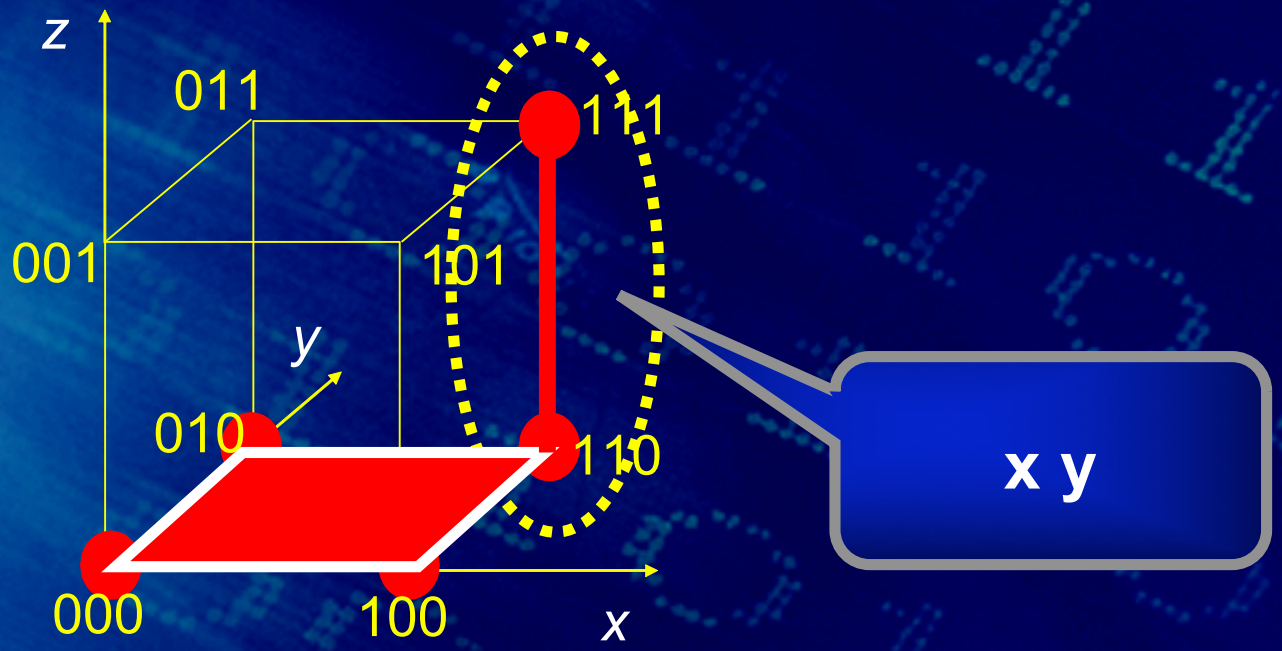



**we could represent the related "edge"**


Rel. 14.06.2018

# *A further step (5)*

**... and instead of representing separately the 2 vertices**




**we could represent the related "edge"**


Rel. 14.06.2018

# *Thus obtaining*

F =

x'y'z' +

x y'z' +

x y z' +

x'y z' +

x y z

→ F = z' + x y



Rel. 14.06.2018

# *Let's formalized the concept*

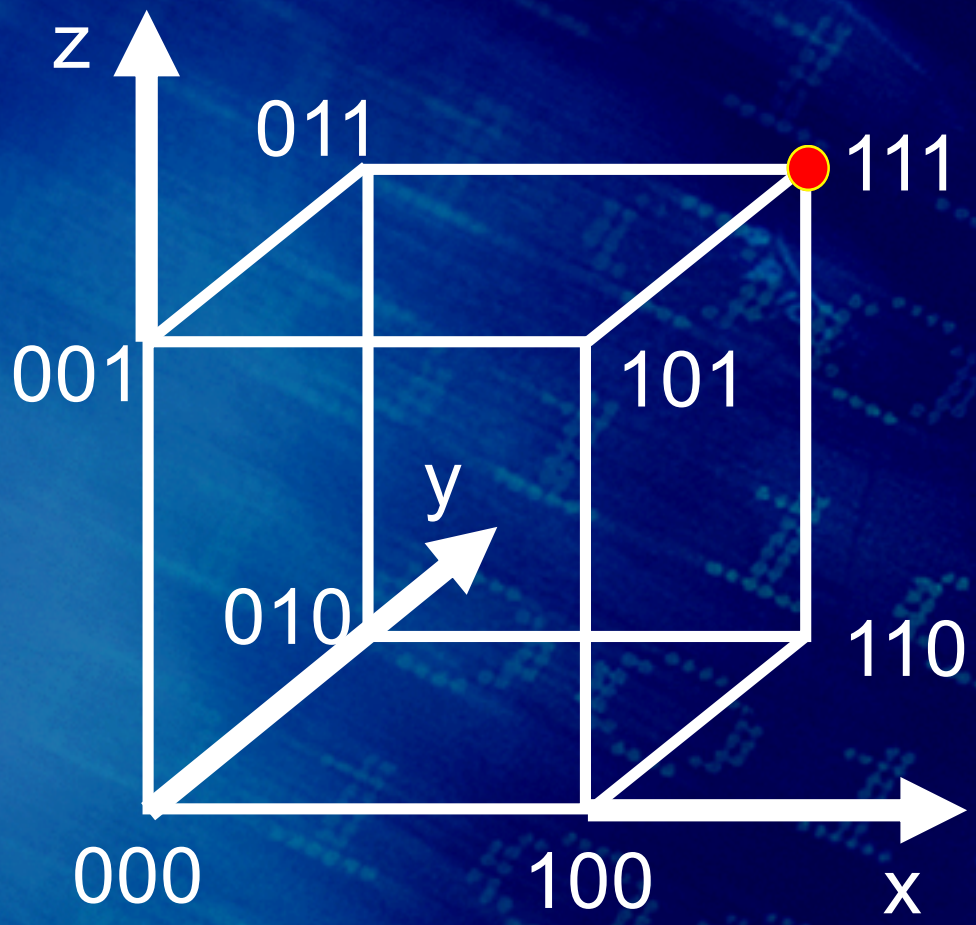
## *K-cube*

A k-dimensional sub-space
(or k-cube)

$S_k \subseteq B^n$

is a set of $2^k$ vertices, in which n-k input variables get a same constant value

# 0-cube = vertex




$B = \{ 0, 1 \}$

$B^3$

n = 3

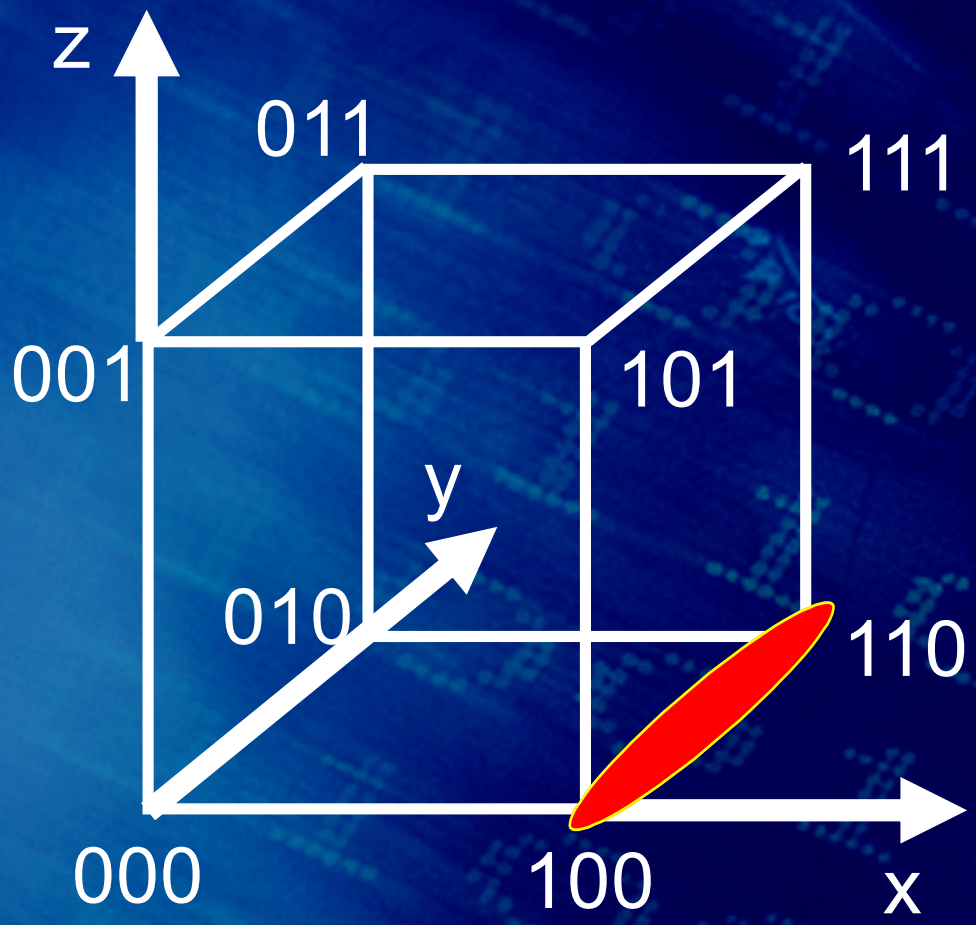
k = 0

n-k = 3



Rel. 14.06.2018

# *1-cube = edge*




$B = \{ 0, 1 \}$

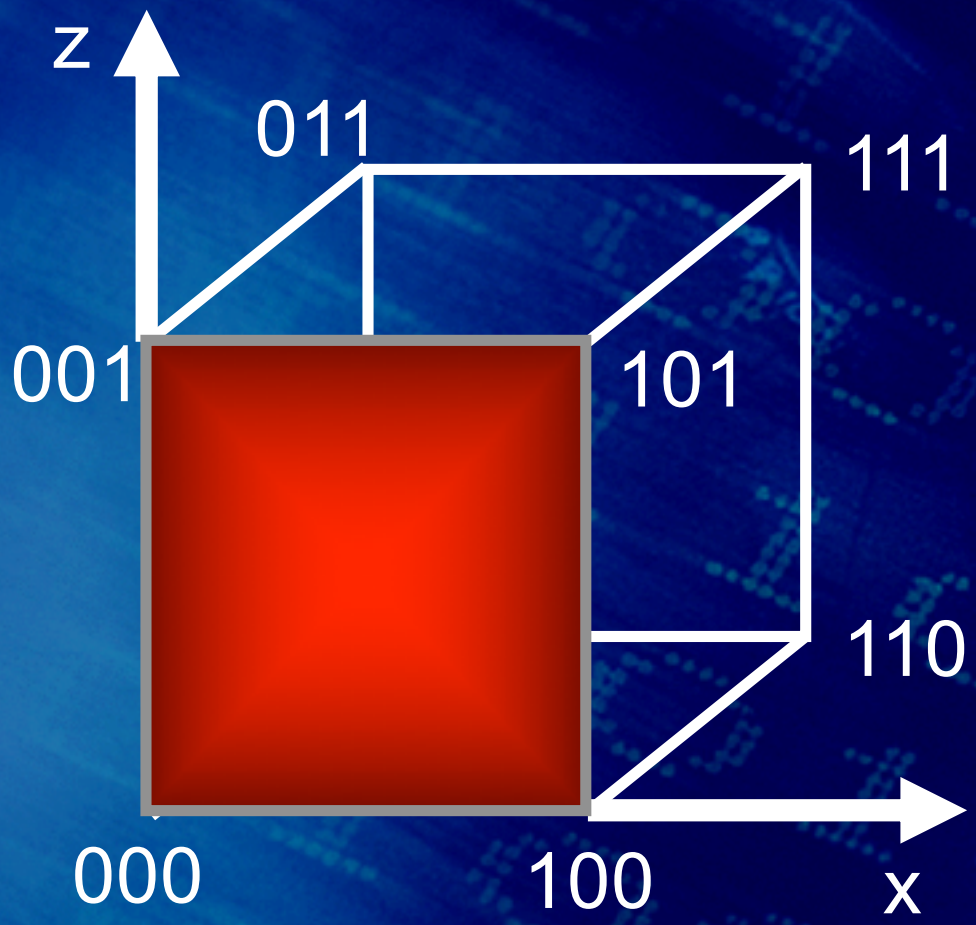
$B^3$

n = 3

k = 1

n-k = 2

# *2-cube = face*




$B = \{ 0, 1 \}$

$B^3$

n = 3

k = 2

n-k = 1



Rel. 14.06.2018

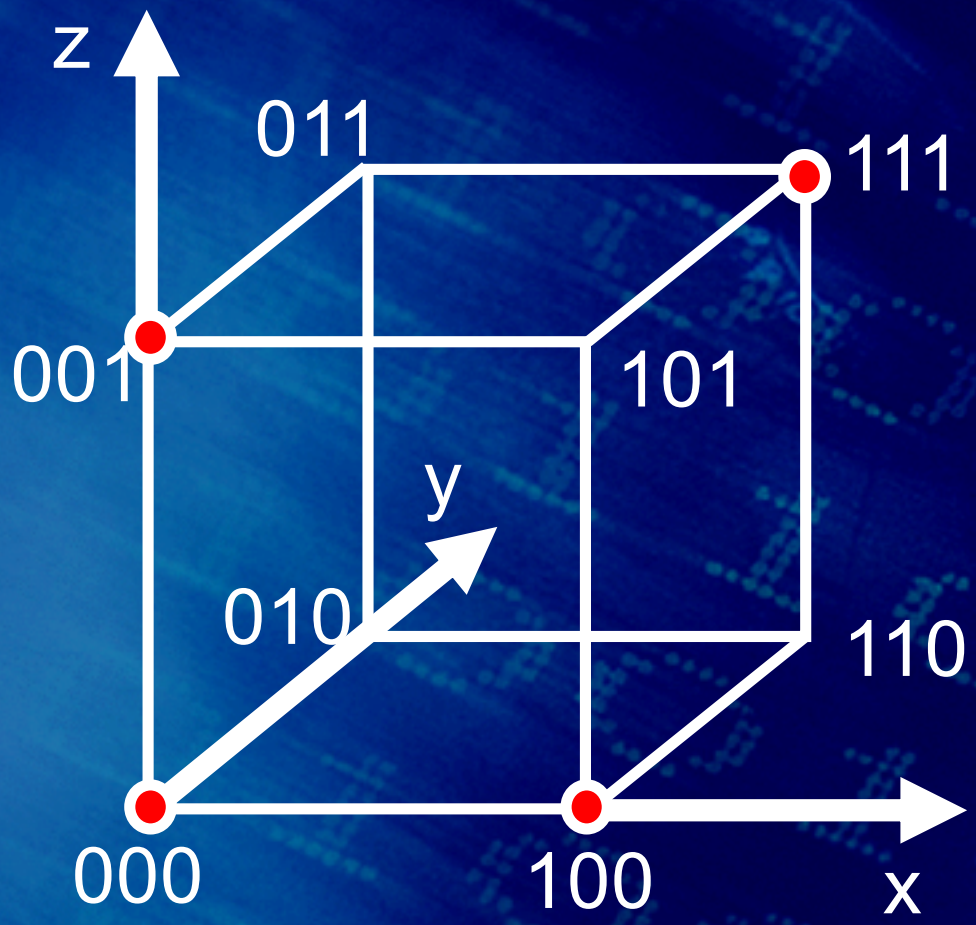
# *Remark #1*

- **Not all the sub-sets of $B^n$ with cardinality $2^k$ are k-cubes.**

# An example of set of $2^2$ vertices which is NOT a 2-cube

$B = \{ 0, 1 \}$

$B^3$



# *Remark #2*

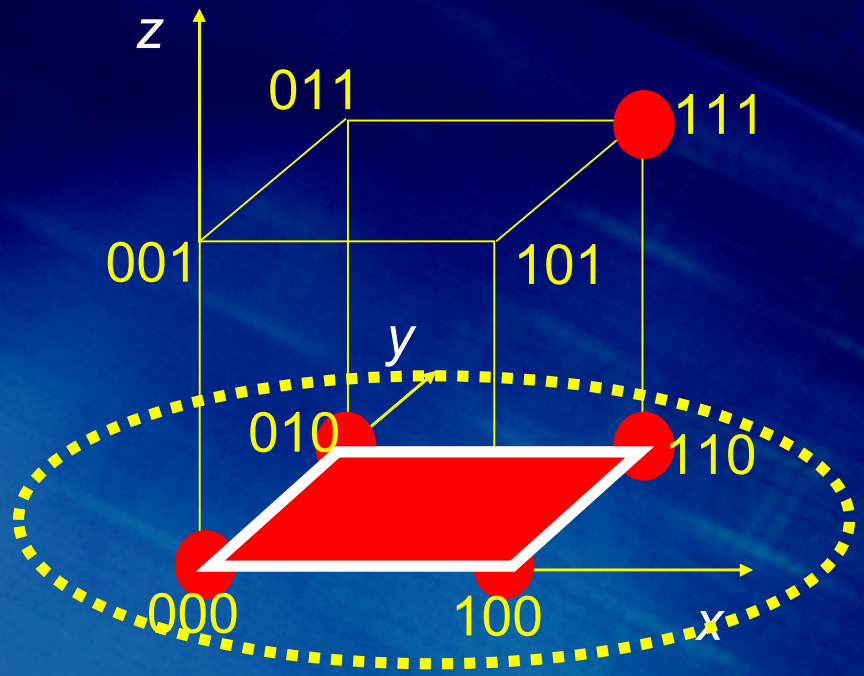
- **K-cubes are easy to find when resorting to geometrical (spatial) representations, since we look for physically adjacent vertices…**
- **But not so easy when the function is represented in 2 dimensions, only**

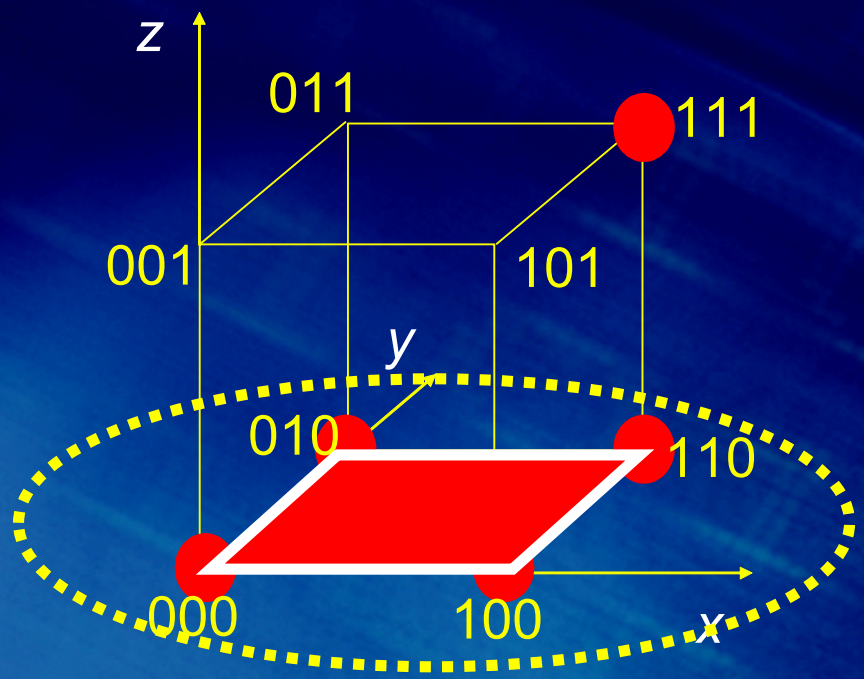
# ***Karnaugh Maps***

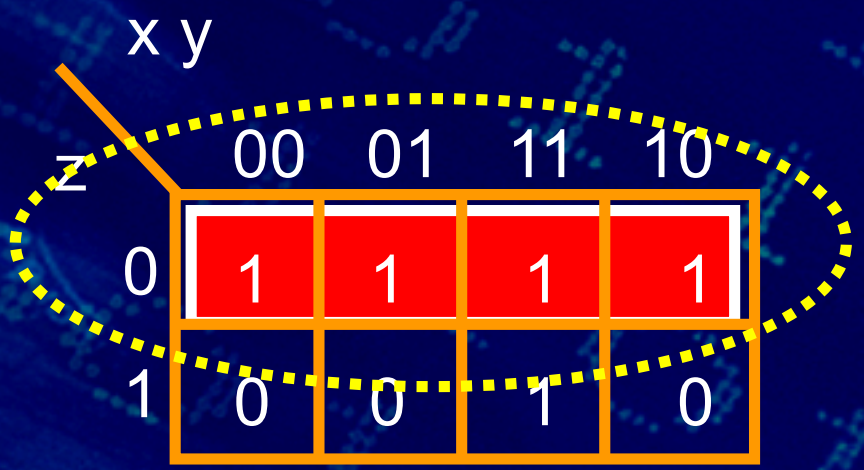
They try to make the task easier…

# *Karnaugh maps*

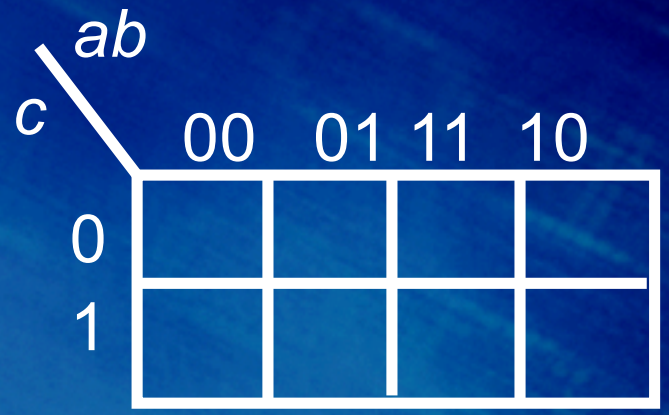
- **Values are assigned to rows and columns in such a way that cells that are physically adjacent in the geometric representation be physically adjacent in the planar representation, as well.**
- **Rows and columns are thus usually labeled to follow the reflected Gray code sequence:**



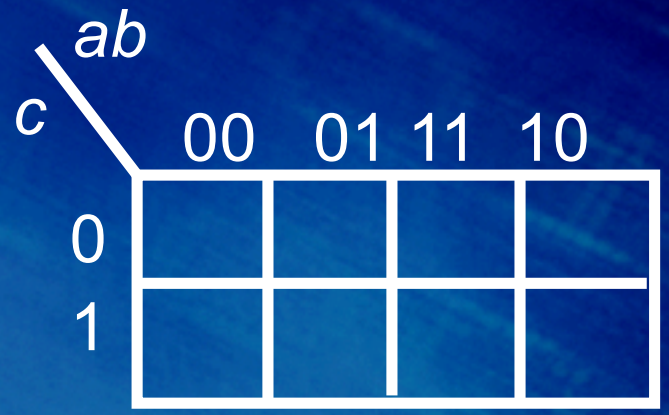


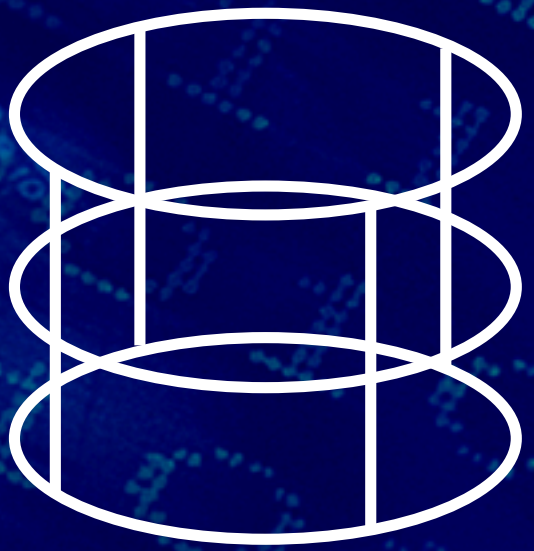
# 3 input maps



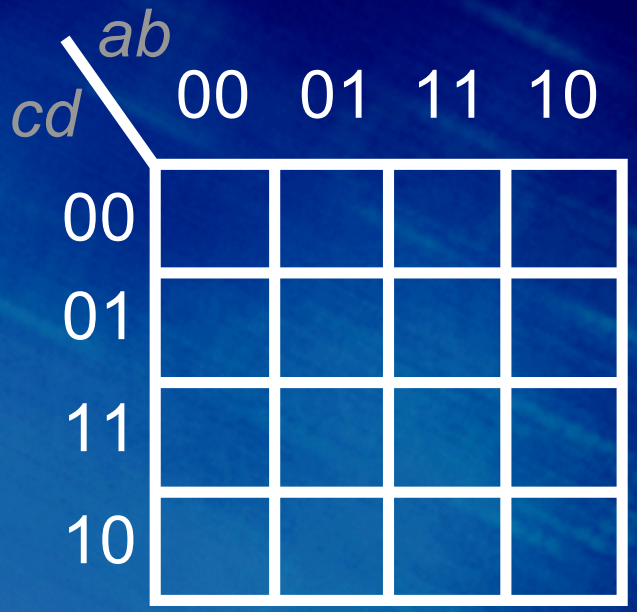
# 3 input maps



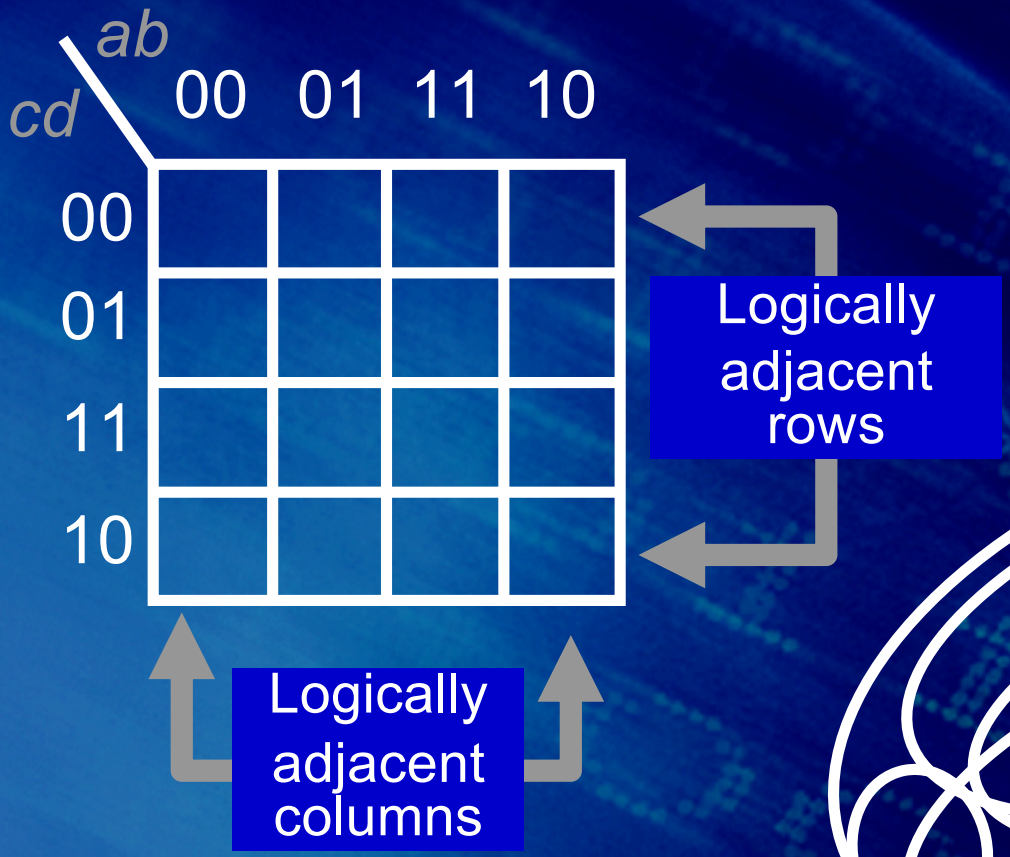

**Logically adjacent columns**



# 4 input maps



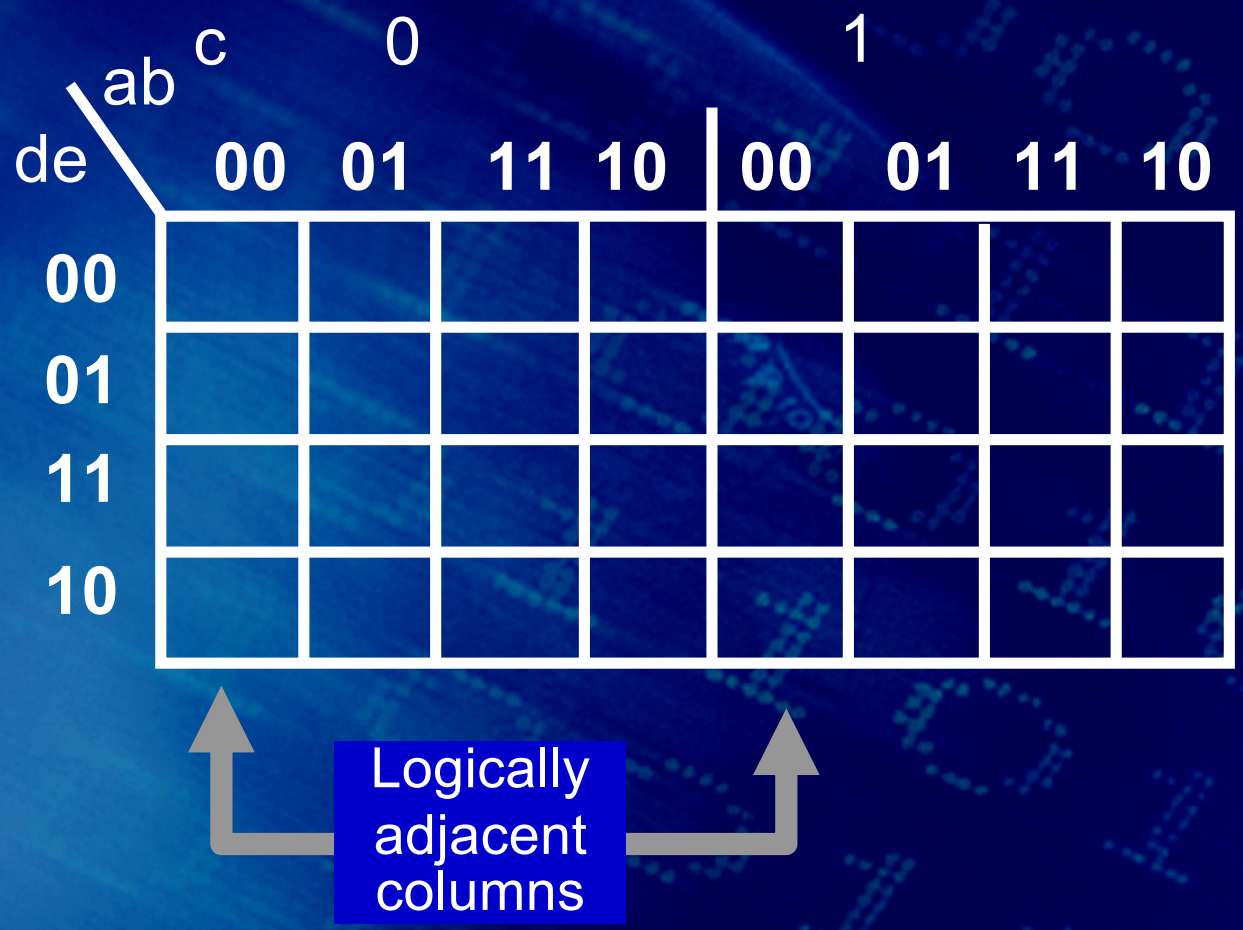
# 4 input maps





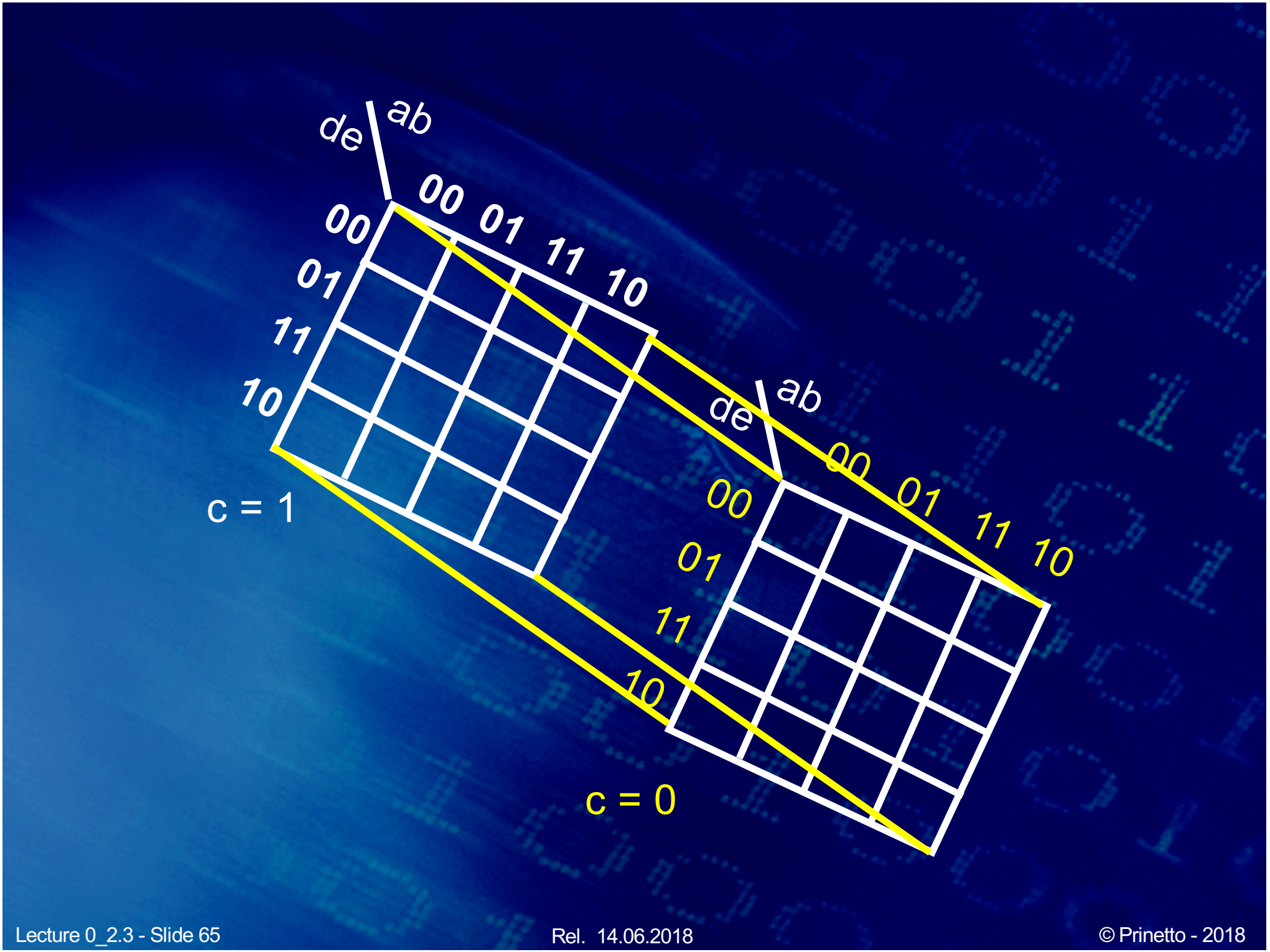
Rel. 14.06.2018

# 5 input maps





Rel. 14.06.2018



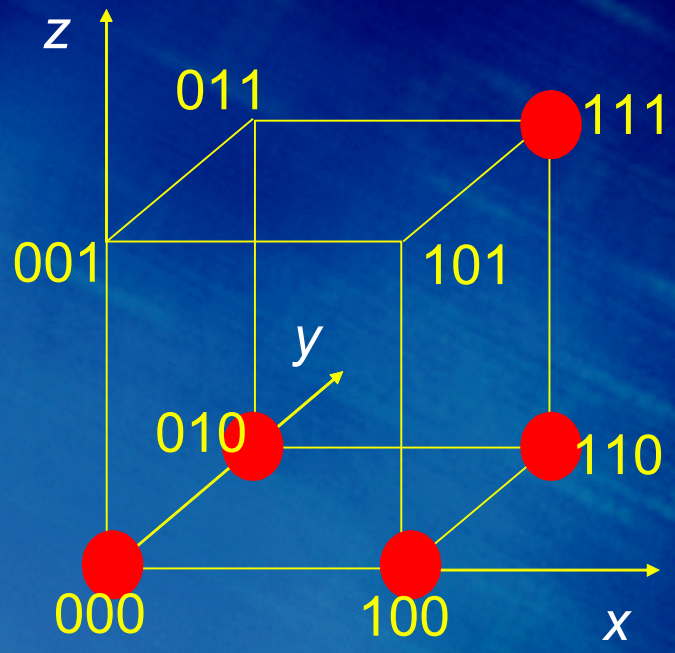


Rel. 14.06.2018

# *Remark #3*

- **The adopted solution is perfect for humans but not so efficient for machines…**

# Machine oriented representation




**We could write:**

**F =**

**000 +**

**010 +**

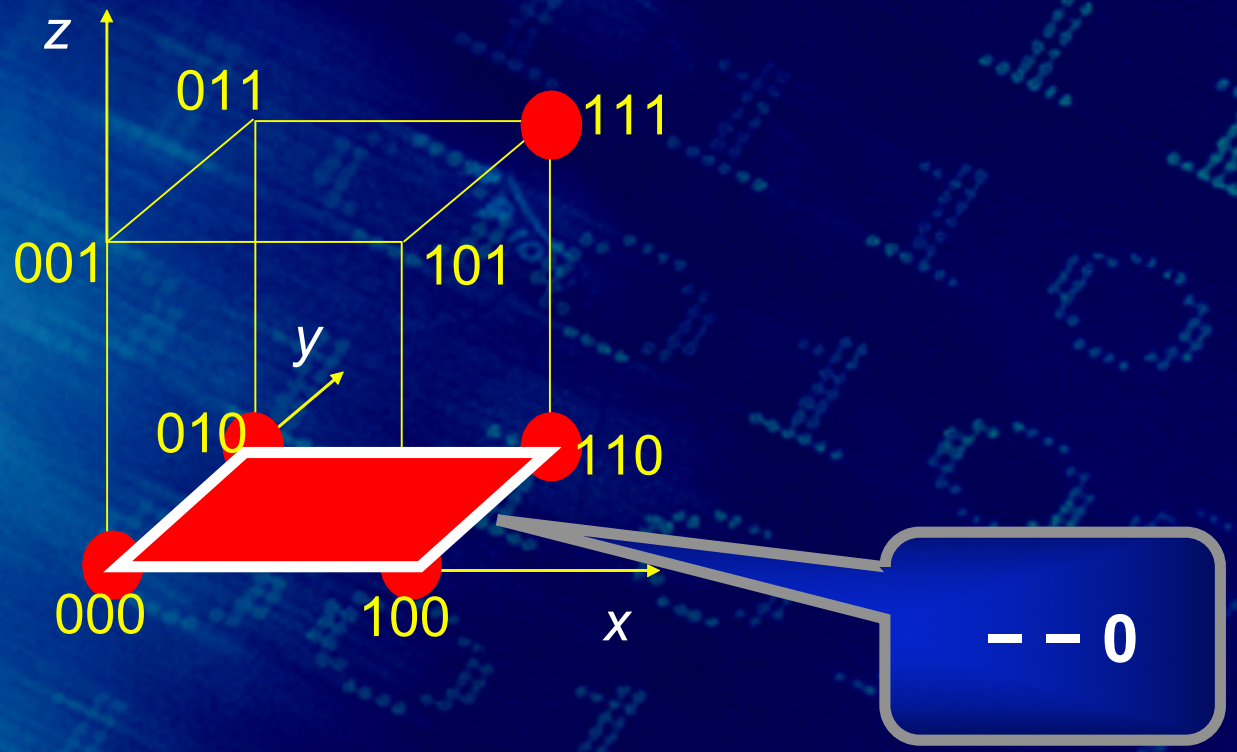
**110 +**

**100 +**

**111**

# *A further step*

**The whole “face” could be represented as**

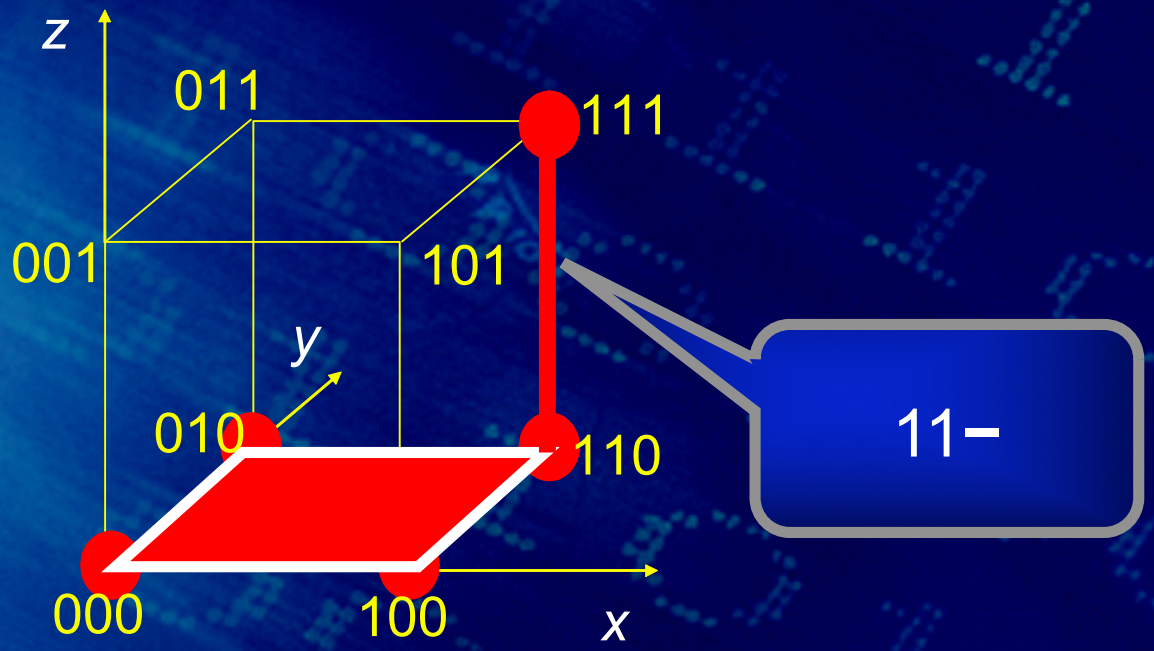




Rel. 14.06.2018

# *A further step (2)*

**... and the edge could be represented as**





Rel. 14.06.2018

# *Thus obtaining*

F =
000 +
010 +
110 +
100 +
111

→

F =
– – 0 +
1 1 –


Rel. 14.06.2018

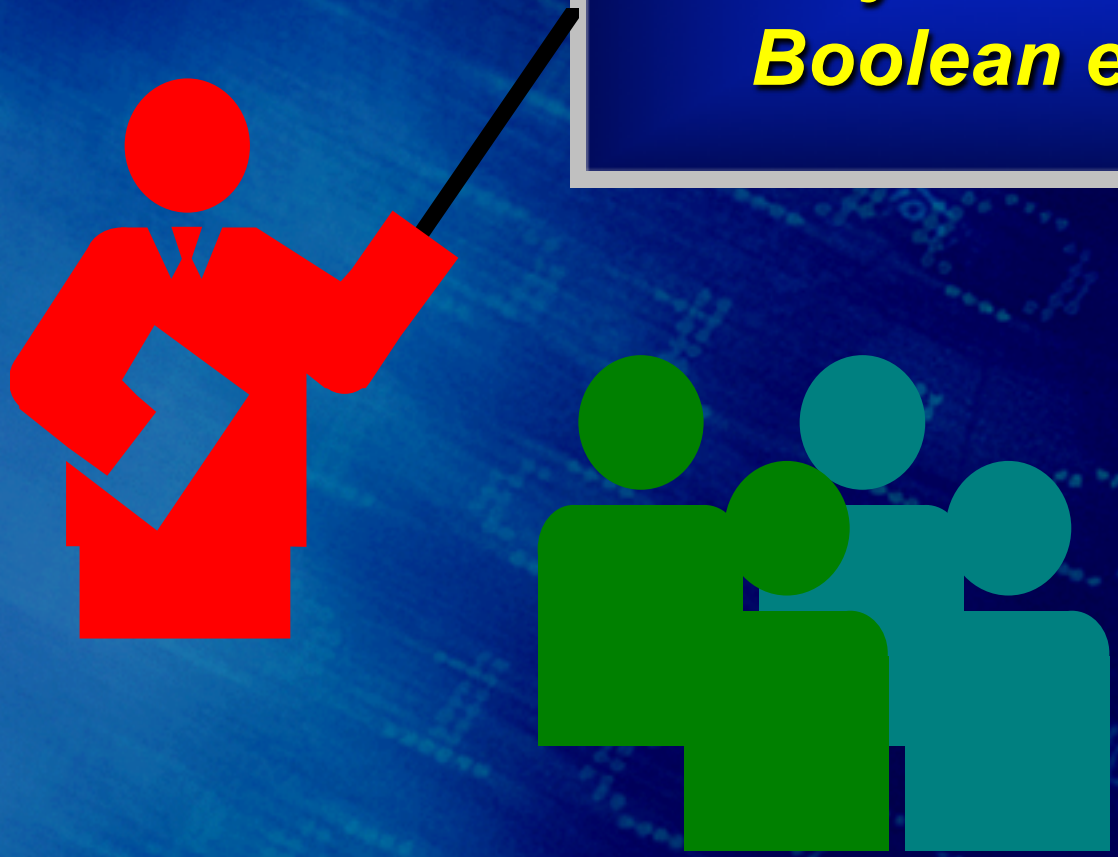
# Outline

- From functions to circuits
- Remarks
- Problems to face
- Lessons learned


Rel. 14.06.2018

## CAVEAT

***A Boolean function can be represented by an infinite # of Boolean expressions***



# *Example 1*

**The following *Boolean expressions* all represent the same *Boolean function*:**

***a***

***a + a***

***a + a + a***

***a + a + a + a***

***...***

# *Example 2*

**The following *Boolean expressions* all represent the same *Boolean function*:**

***a + x***

***a x' + x***

***a + a' x***

***a + a + x***

***(a + x) a + a' x***



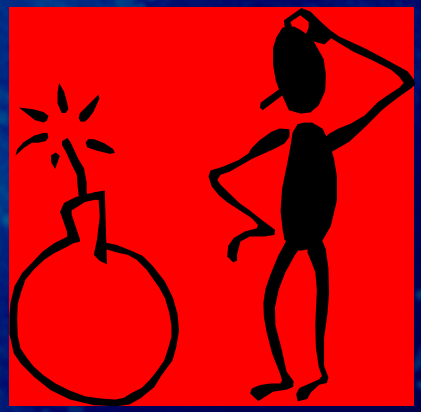
Rel. 14.06.2018

# *Problems to face*

- Check whether two representations are equivalent ( *equivalence checking* )
- Find an equivalent representation having "minimum cost" ( *minimization* )
- Guarantee that, for each representation method, each function have a unique representation ( *canonicity* )

# *Caveat*

**The three above problems have different complexity and different solutions for each representation method.**



# *Equivalence checking with Boolean Expressions*

**The equivalence between two boolean expressions can be verified:**

- ***manually***, **resorting to a set of re-write rules, mainly based on postulates, properties and theorems of boolean algebras**
- ***automatically***, **resorting to ad-hoc tools, such as *tautology checkers* or, more in general, *theorem provers*.**

# *Example 1*

**Prove that the following two expressions**

$$F = z' + x\,y$$

$$F = x'y'z' + x\,y'z' + x\,y\,z' + x'y\,z' + x\,y\,z$$

**are equivalent**

# Solutions

$$F = x'y'z' + x\,y'z' + x\,y\,z' + x'y\,z' + x\,y\,z =$$
$$(x' + x)\,y'z' + (x' + x)\,y\,z' + x\,y\,z =$$
$$y'z' + y\,z' + x\,y\,z =$$
$$(y' + y)\,z' + x\,y\,z =$$
$$z' + x\,y\,z =$$
$$z' + x\,y$$

q.e.d.


Rel. 14.06.2018

# *Example 2*

**Prove that:**

$$(c' + abd + b'd + a'b)(c + ab + bd) =$$
$$= b(a + c)(a' + c') + d(b + c)$$

# Solution *(1)*

(c’ + abd + b’d + a’b) (c + ab + bd) =

c’ (ab + bd) + c (abd + b’d + a’b) =

abc’ + bc’d + abcd + b’cd + a’bc =

abc’ + abcd + bc’d + a’bc + b’cd =

ab (c’ + cd) + bc’d + a’bc + b’cd =

ab (c’ + d) + bc’d + a’bc + b’cd =

ab (c’ + d) + bc’d + a’bc + a’bd + b’cd =

abc’ + abd + bc’d + a’bc + a’bd + b’cd =

(abd + a’bd) + abc’ + bc’d + a’bc + b’cd =

# Solution *(2)*

bd (a + a') + abc' + bc'd + a'bc + b'cd =

bd (1) + abc' + bc'd + a'bc + b'cd =

bd + abc' + bc'd + a'bc + b'cd =

bd + bc'd + abc' + a'bc + b'cd =

bd + abc' + a'bc + b'cd =

d (b +b'c) + abc' + a'bc =

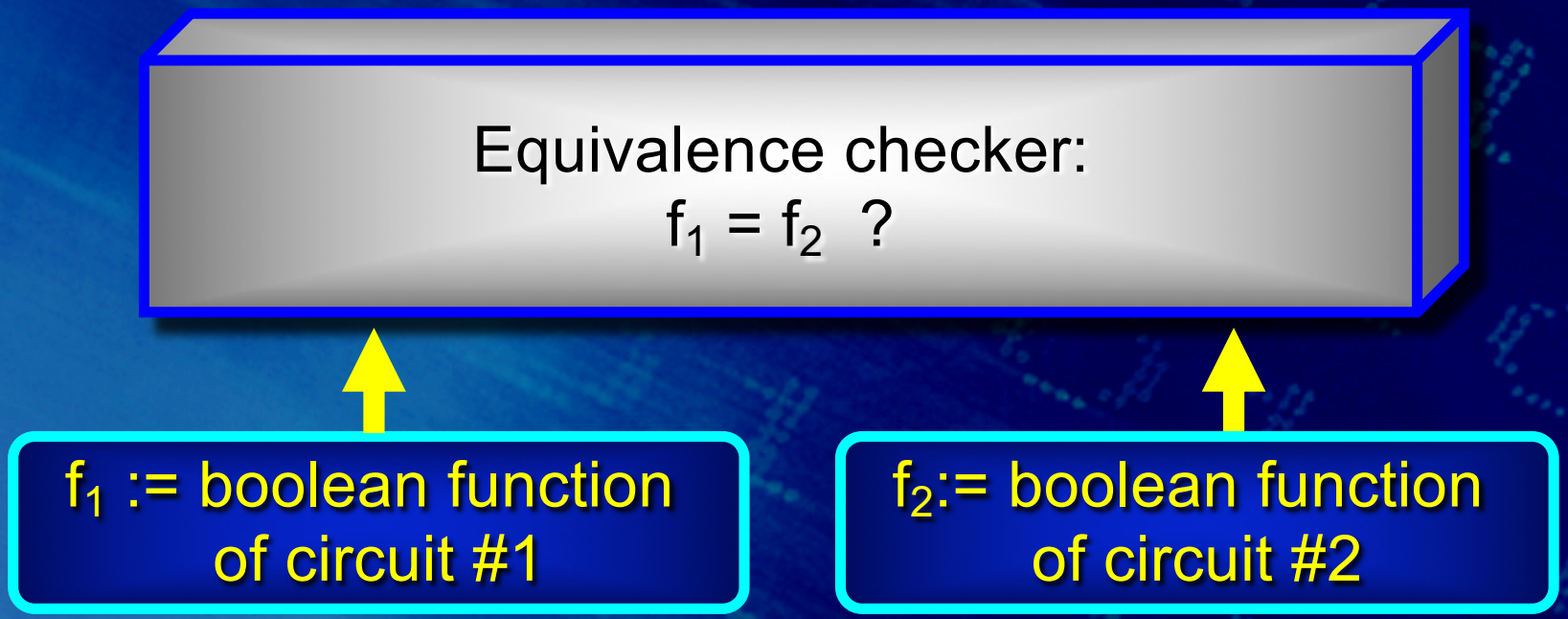
d (b + c) + abc' + a'bc =

d (b + c) + b (ac' + a'c) =

d (b + c) + b (a + c) (a' + c')

q.e.d.

# *Equivalence checker*



# *Minimizing Boolean Expressions*

# *Example*

**One could derive the expression:**

$$F = z' + x\,y$$

**from the expression:**

$$F = x'y'z' + x\,y'z' + x\,y\,z' + x'y\,z' + x\,y\,z$$

**By applying the rewriting rules and Boolean Algebra theorems:**


Rel. 14.06.2018

# Solution

$$F = x'y'z' + x\,y'z' + x\,y\,z' + x'y\,z' + x\,y\,z =$$
$$(x' + x)\,y'z' + (x' + x)\,y\,z' + x\,y\,z =$$
$$y'z' + y\,z' + x\,y\,z =$$
$$(y' + y)\,z' + x\,y\,z =$$
$$z' + x\,y\,z =$$
$$z' + x\,y$$


Rel. 14.06.2018

# *Canonicity with Boolean Expressions*

**An *SOP* expression is *canonical* iff is composed of minterms, only.**



Rel. 14.06.2018

# Canonicity with Boolean Expressions

**An *SOP* expression is *canonical* iff is composed of minterms, only.**

$$f = x' y z' + x y' z + x y z$$

*canonical*

# Canonicity with Boolean Expressions

An *SOP* expression is ***canonical*** iff is composed of minterms, only.

$f = x' y z' + x y' z + x y z$

*canonical*

$f = x' y z' + x z$

*not canonical*



Rel. 14.06.2018

# Outline

- From functions to circuits
- Remarks
- Problems to face
- Lessons learned


Rel. 14.06.2018

# *From functions to circuits (Library binding)*

**The library binding of boolean expressions is trivial:**

| *logic operation* | *logic gate* |
|---|---|
| sum | `or` |
| product | `and` |
| complement | `not` |

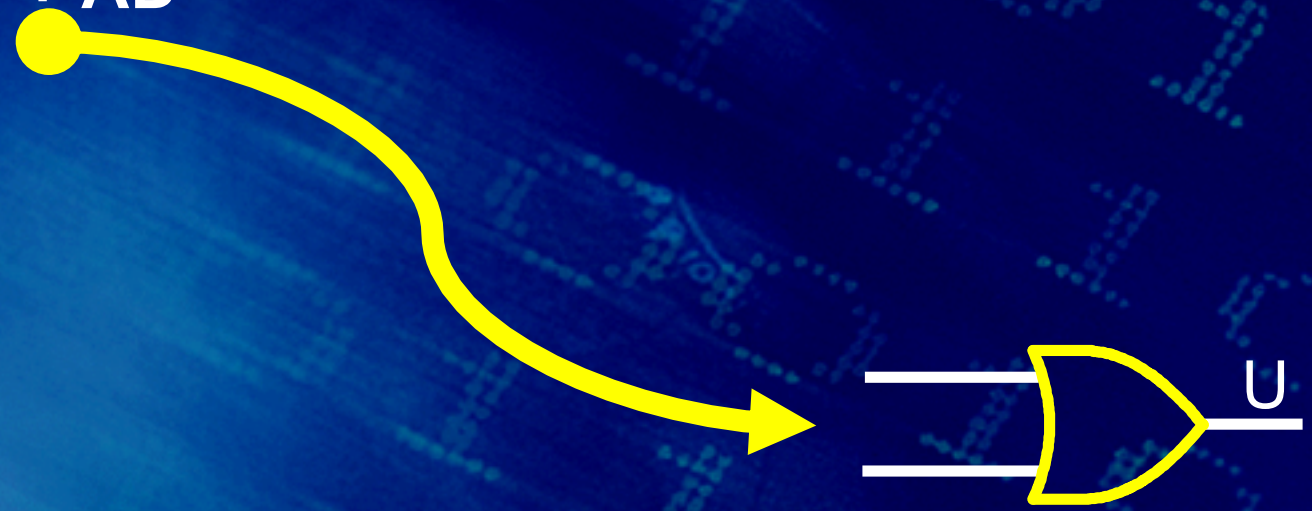

Rel. 14.06.2018

# *Example*

**U = A'B' + AD**

# Example

U = A'B' + AD

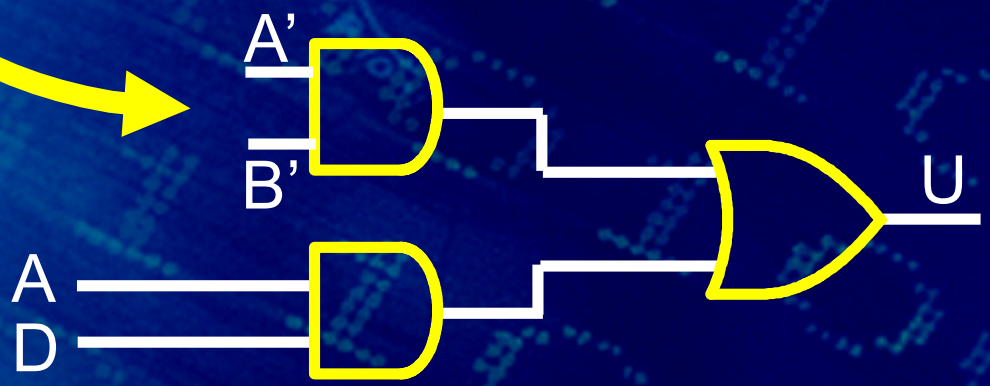


# Example

U = A'B' + AD

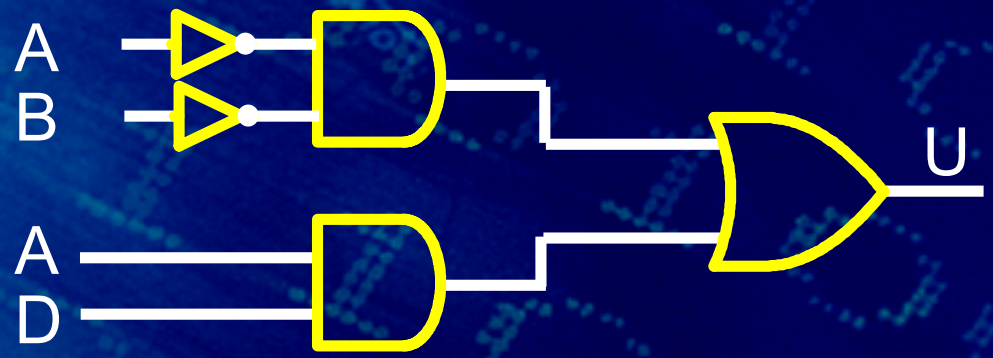
# Example

**U = A'B' + AD**



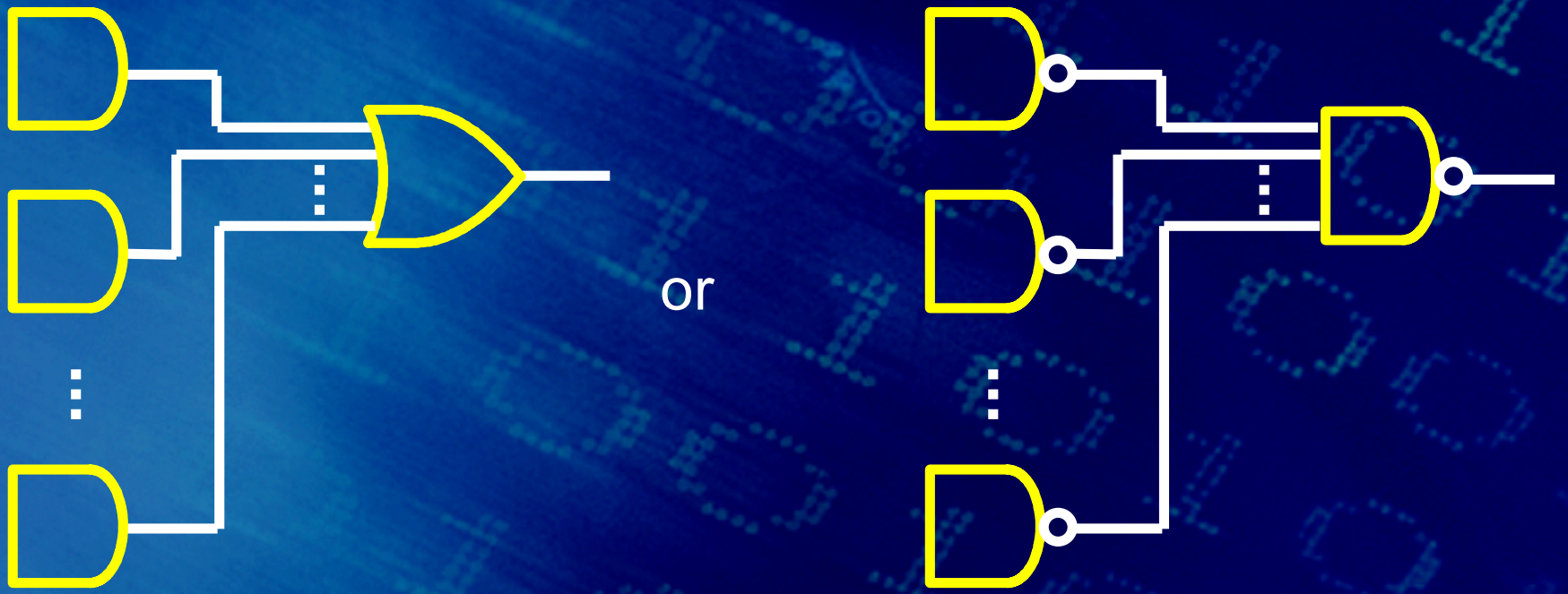
# *Example*

**U = A'B' + AD**



# *Alternative implementation*

**The above presented approach always leads to circuits implemented as 2-level networks:**



# *Alternative implementation (2)*

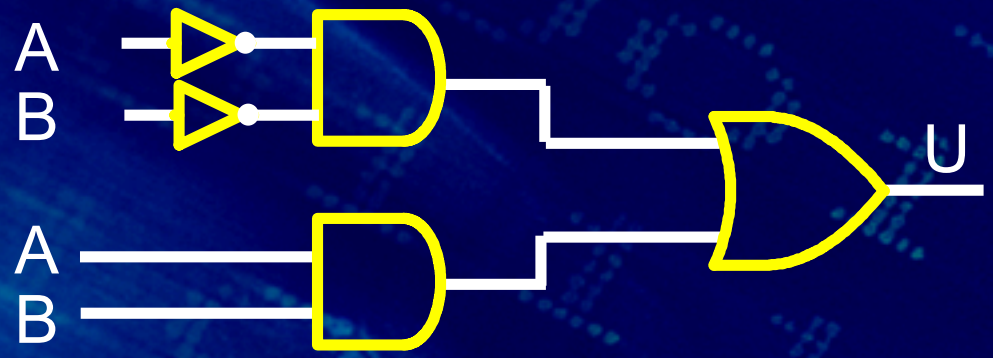
**In order to use different gates or to get different circuit "shapes", algebraic transformations on the obtained logic expressions have to be performed.**


Rel. 14.06.2018

# *Example 1*

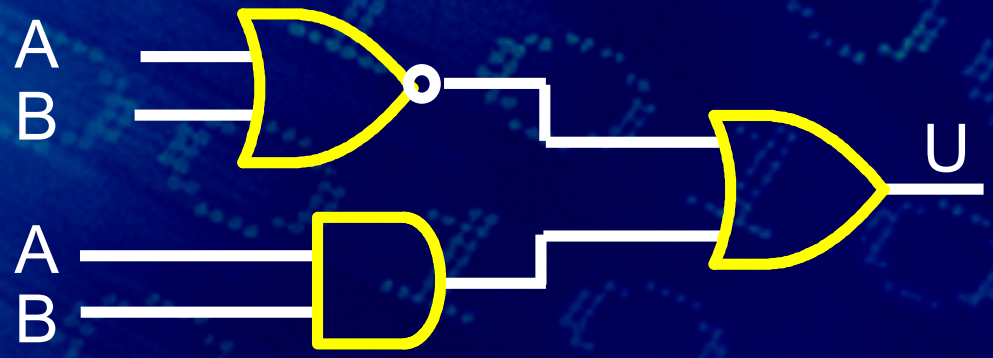
**Starting from:**

**U = A'B' + A B**




**applying the De Morgan Theorem, one gets:**

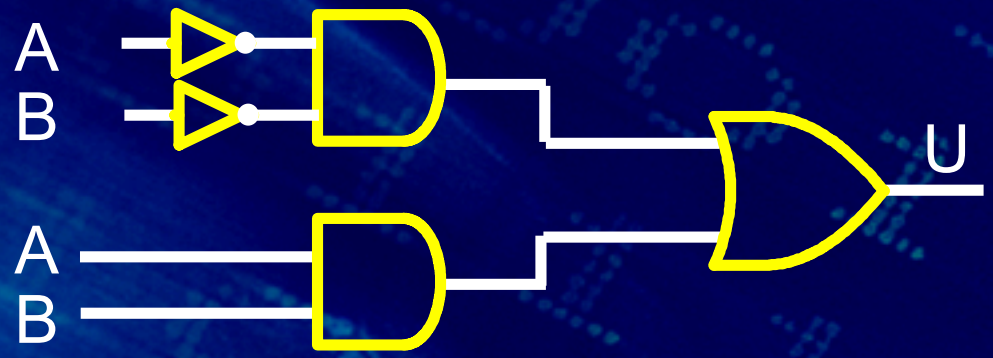
**U = (A + B)'+ A B**



# *Example 2*

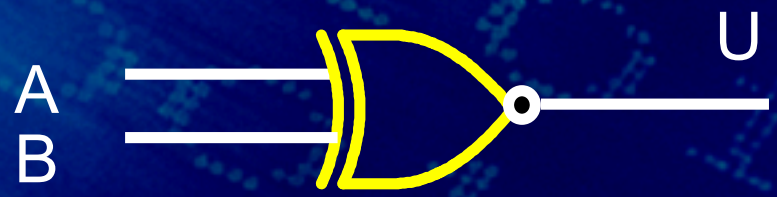
**Starting from:**

**U = A'B' + A B**




**applying the ex-nor definition, one gets:**

**U = (A+B)' + A B**



# *Practical implication*

- **If we use logic gates to implement the function**

$$F = z' + x\,y$$

**we obtain:**

# *Practical implication*

- **If we use logic gates to implement the function**

$$F = z' + x\,y$$

**we obtain:**



Rel. 14.06.2018

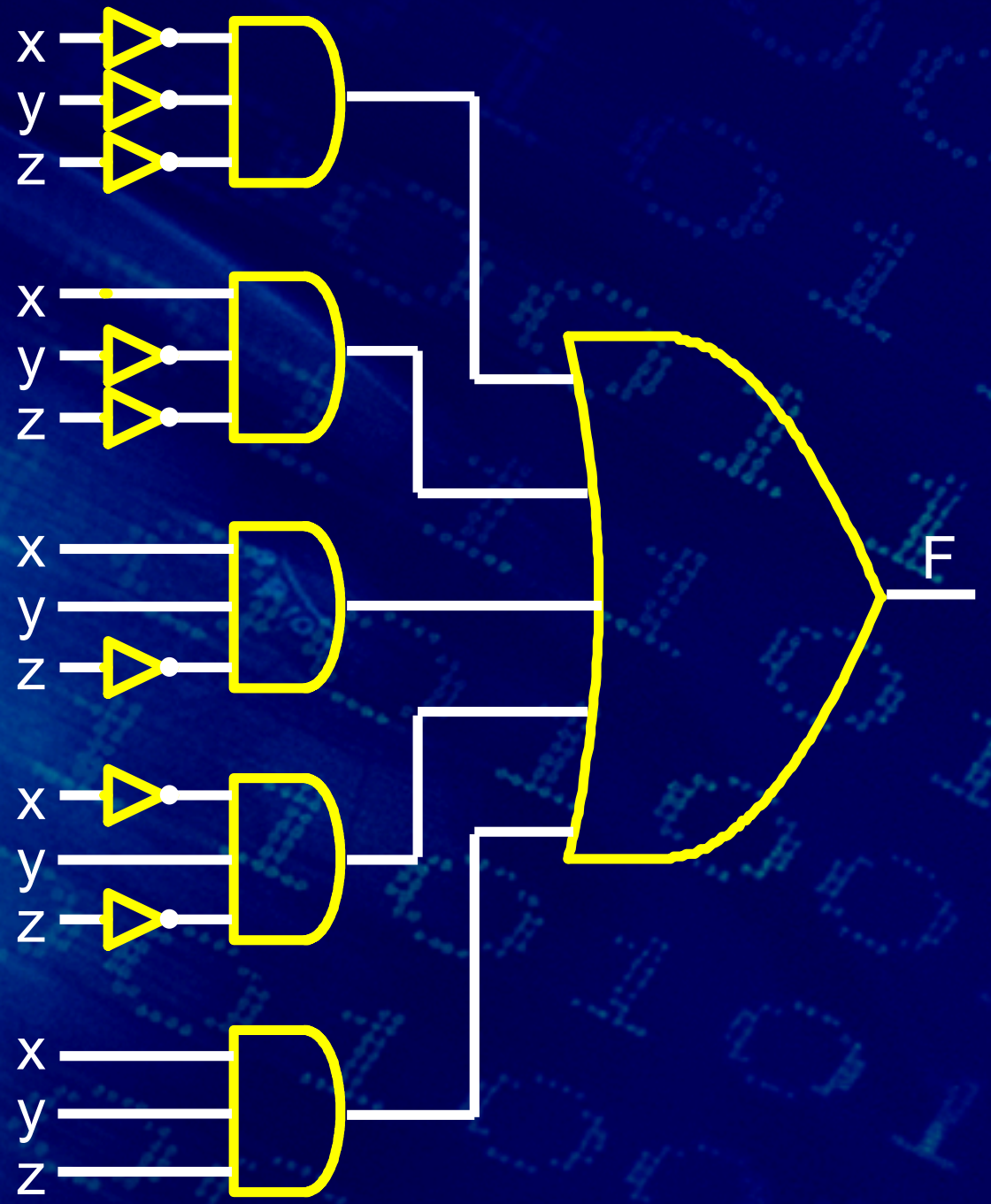
# *Practical implication* (2)

- **If we use logic gates to implement the equivalent function**

  **F = x'y'z' + x y'z' + x y z' + x'y z' + x y z**

  **we obtain:**

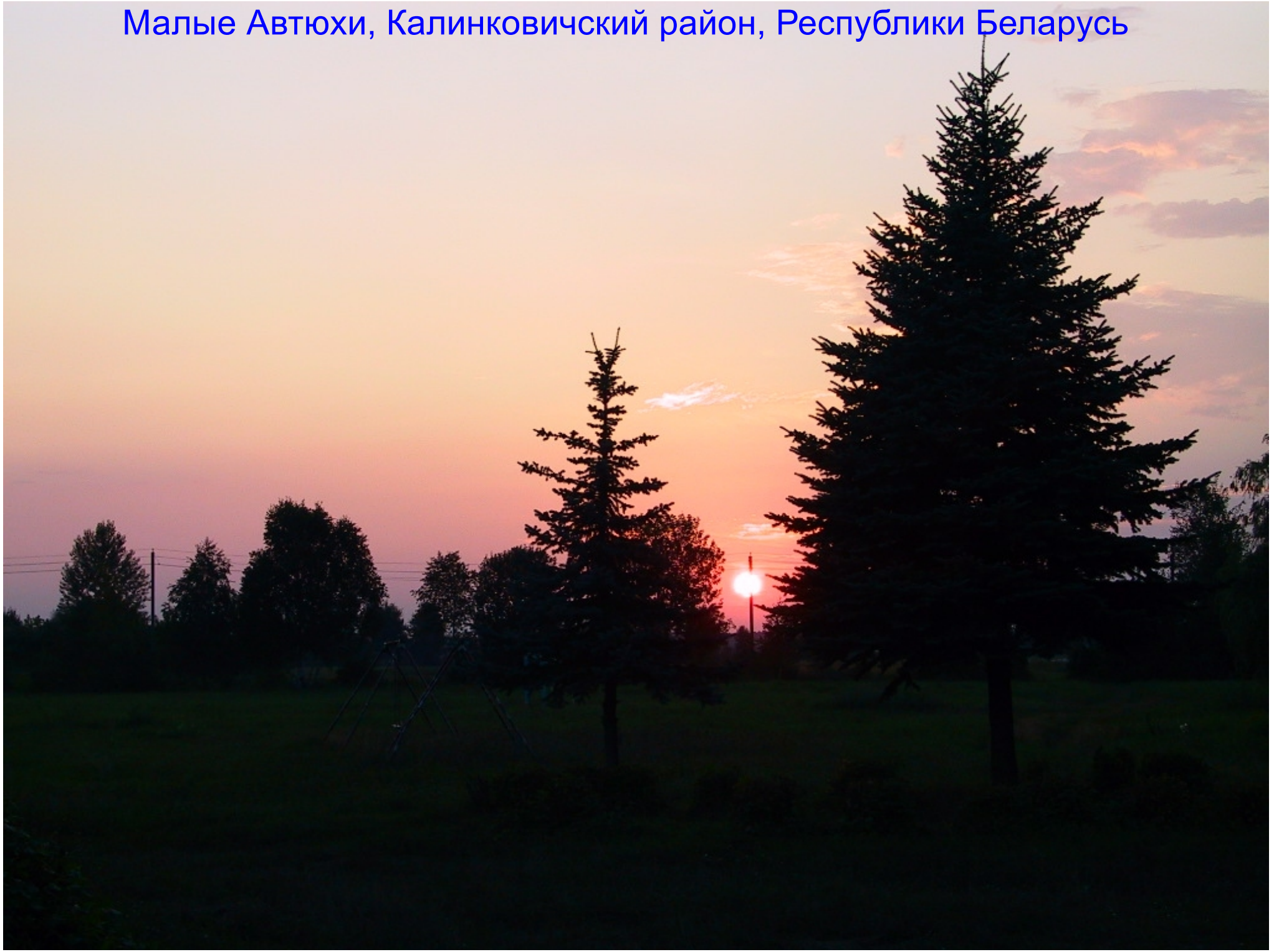
- **F =**
  **x'y'z' +**
  **x y'z' +**
  **x y z' +**
  **x'y z' +**
  **x y z**



# *Lessons learned*

1. Concept of "*cost* "
   - The "cost" of an implementation is proportional to the "cost" of the expression to be implemented
   - Minimizing expressions implies minimizing the implementation cost.


Rel. 14.06.2018


Малые Автюхи, Калинковичский район, Республики Беларусь