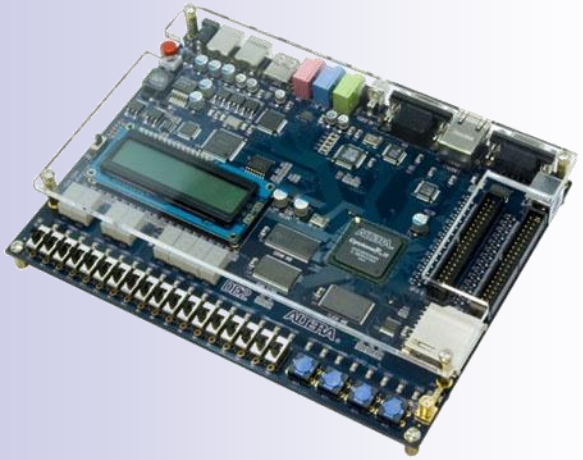
Logic Systems and Processors
cz:Logické systémy a procesory



Lecturer: Richard Šusta

susta@fel.cvut.cz, richard@susta.cz
+420 2 2435 **7359**

Version: 1.2 Oct 2, 2024
added slide 28, corrections of typos

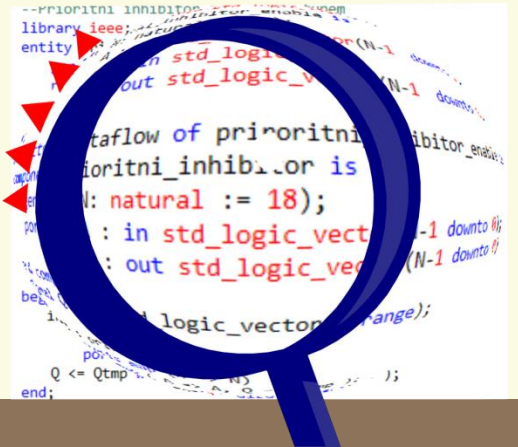
CTU-FEE in Prague, CR – subject BE5B35LSP

The next lecture, we begin with VHDL

I recommend studying Chapter 2
in the textbook

Introduction to circuit design in VHDL I.

It is only 14 pages long, starting from pg. 8,
and explains the basics of
VHDL syntax with the aid of
simple examples.

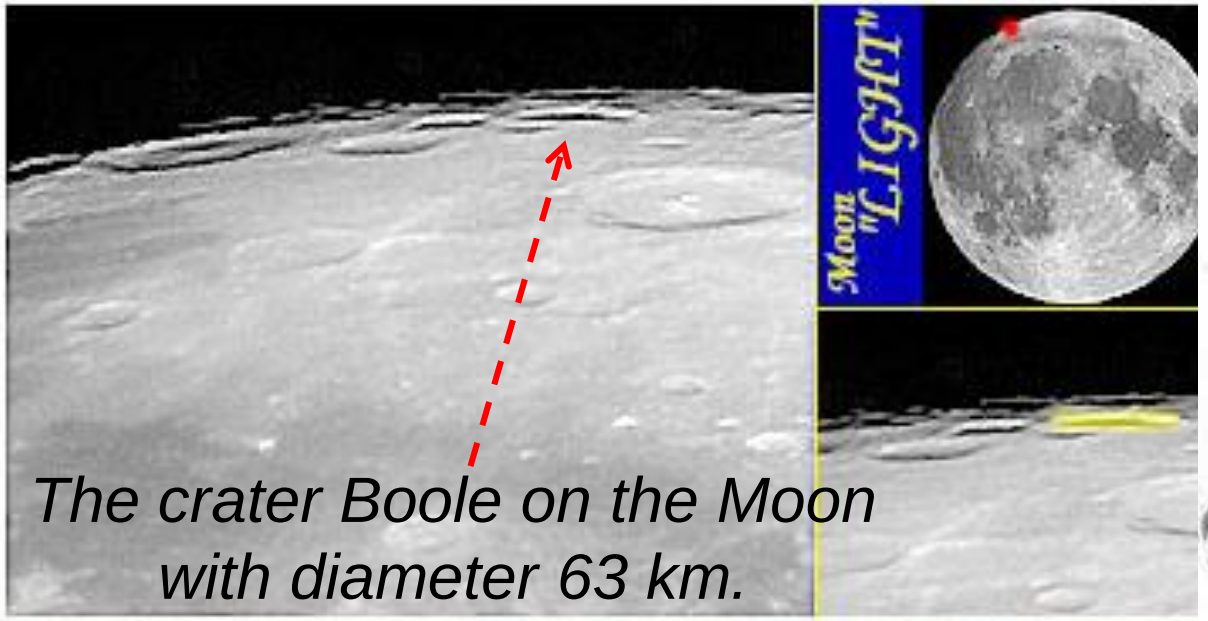


Boolean Algebra

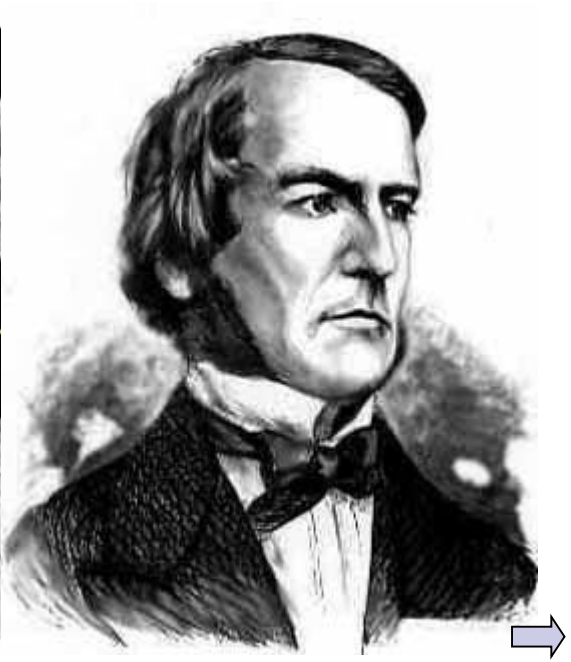
Mathematical Foundation of Logic

George Boole (1815-1864)

- He was born into a family of shoemakers in the English industrial city of Lincoln and learned mathematics by self-study at a school for poorer children.
- He approached logic in a new way reducing it to a simple algebra **incorporated logic into mathematics**;
- The axiomatic foundations of Boolean algebraic systems was performed by Huntington (1904).
- In 1913, "Boolean algebra" was first suggested by Sheffer as name for the system specified by Huntington postulates. (*Note: Sheffer also invented Sheffer's stroke aka NAND operation*)



The crater Boole on the Moon with diameter 63 km.



[Source: http://www.astrosurf.com/cidadao/

A Boolean algebra is a six tuple <B, +, ● , ' , 0, 1>, where

- B is the set of elements having **at least two** elements x, y ∈ B such that x ≠ y;
- 0 is the minimum element; and 1 is the maximum element;
- three operations: unary ' , binary + and ● satisfying (Huntington's) postulates

Postulate	(a)	(b)
Closure *Closure*	y + x = z ∈ B; ∀ x, y ∈ B	y ● x = z ∈ B; ∀ x, y ∈ B
Commutativity *Commutativity*	$a + b = b + a$	$a \bullet b = b \bullet a$
Neutrality *Identity*	$a + 0 = a$	$a \bullet 1 = a$
Distributability *Distributivity*	$a + (b \bullet c) = (a + b) \bullet (a + c)$	$a \bullet (b + c) = a \bullet b + a \bullet c$
Complementarity *Complementation*	$a + a' = 1$	$a \bullet a' = 0$

Note: Huntington's postulates do not include associativity, but we can derive it from them.

1. **Closure (***cz: uzávěr množiny***)**:
   A set S is said to be *closed* with respect to a binary operator ● if whenever y ● x = z ∈ S; for all x, y ∈ S
   *Ex: operation + or * in S= ℤ ={...-2, -1, 0, 1, 2 ...}*
2. **Associative law**: a binary operator ● on a set S is said to be associative whenever (x ● y) ● z = x ● (y ● z) for all x, y, z, ∈ S
   Ex: in ℤ assoc. +, but non-assoc. -, (7-5)-1 ≠ 7-(5-1)
3. **Commutative law:** x ● y = y ● x for all x, y ∈ S
4. **Identity element:** A set S is said to have an identity element if there exists an element e ∈ S; e ● x = x ● e = x for every x ∈ S
   *Ex) The element 0 with + , or 1 with * on the set of integers.*
5. **Inverse element** x ● y = e
   Ex) In ℤ , a + (-a) = 0
6. **Distributive law** ● operator is said to be distributive over operator ○ whenever x ● (y ○ z) = (x ● y) ○ (x ● z)

Boolean algebra is called

- **Boolean logic**
  aka (also known as) Switching algebra
  when |B| = 2
- In logic, it is called as
  Multi-Valued Logic (MVL)
  when |B| > 2

9 values of MVL-9 used in VHDL

Stiff source,
i.e., with **a low output impedance,**
in the case of voltage logic

Forcing 1	**'1'**
Forcing Unknown	'X'
Forcing 0	**'0'**

Non-stiff (weak) source
with **higher output impedance**
in the case of voltage logic

Weak 1	*'H'*
Weak Unknown	*'W'*
Weak 0	*'L'*

Wired Or/And

FPGAs usually cannot implement

High impedance (disconnected)

High Impedance	**'Z'**

Design note
- any value can be here

Don't Care	**'–'**

Simulation reports

Uninitialized	**'U'**

In MLV-9, operations are defined by tables

Examples of tables

Link	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

AND	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'0'	'U'	'U'	'U'	'0'	'U'	'U'
'X'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'Z'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'W'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'L'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'-'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'

All tables can be found in the scripts:

Introduction to circuit design in VHDL I. - concurrent commands, p. 75

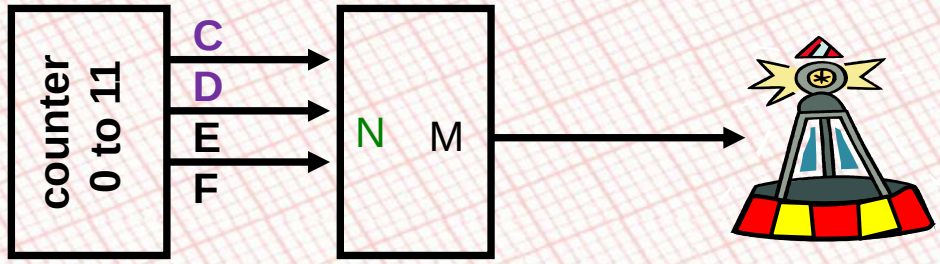
Logical functions

How to define them?

Example B2: EAEA Morse Beacon...

	8	4	2	1	
N	C	D	E	F	Y
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12-15	1	1	-	-	-

wildcards only in inputs



EA = . .- = 01000101111 00

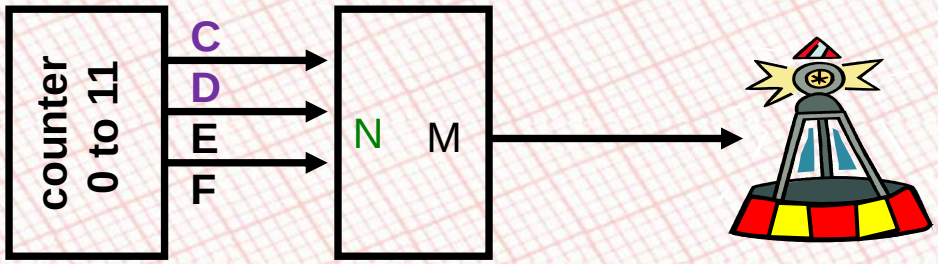
don't care only in outputs

Morse beacon EAEA... - notes

- *The counter only outputs numbers from 0 to 11, so we assigned output X to the unused inputs 12 to 15 - don't care*
- *In state 0 (C=0, D=0, E=0, F=0) we send 0 and the first dot (letter E) starts only in count 1. The reason for a similar choice is the idle state - counters have an initialization state of 0 and we want the beacon light off in 0.*
- *According to the Morse code standard, a period lasts 1 period and a comma lasts 3 periods. The spaces between commas and periods of the same character are 1 period long*
  and the spaces between characters are 3 periods.
- *We'll end the transmission with a pair of zeros. Together with the initial 0, the continuous transmission will create a gap between E and A of the required length of 3 periods.*

Morse Beacon EAEA...

	8	4	2	1	
N	C	D	E	F	Y
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12-15	1	1	-	-	-



EA = . .- = 01000101111100

- onset - must be in 1: m(1,5,7,8,9)
- offset - must be in 0: M(0,2,3,4,6,10,11)
- don't care set (0 or 1)?:

dc(12,13,14,15) What's better? =>

We don't know; we need minimization methods!

Y = m(1,5,7,8,9) + dc(12,13,14,15)

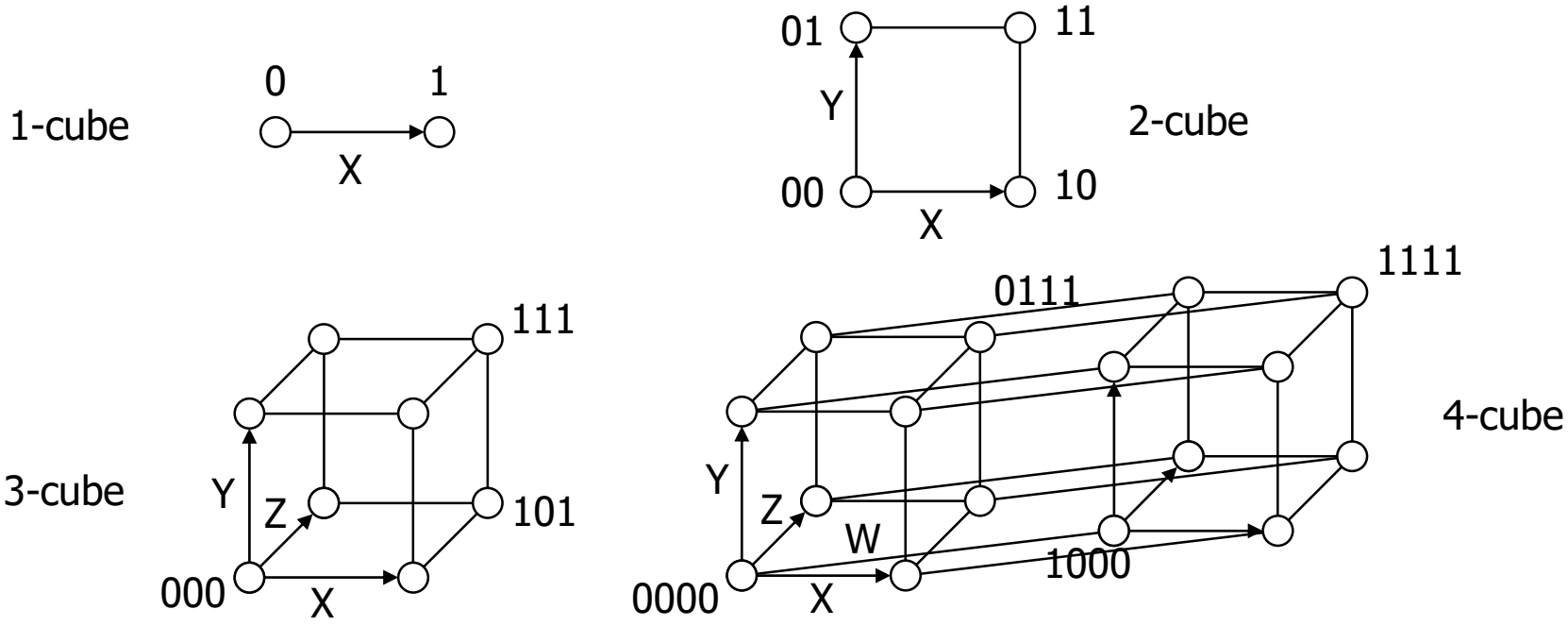
or Y = **M(0,2,3,4,6,11,11) + dc(12,13,14,15)**

Minimizing

Karnaugh's maps

Boolean cubes ~ plots of logic functions

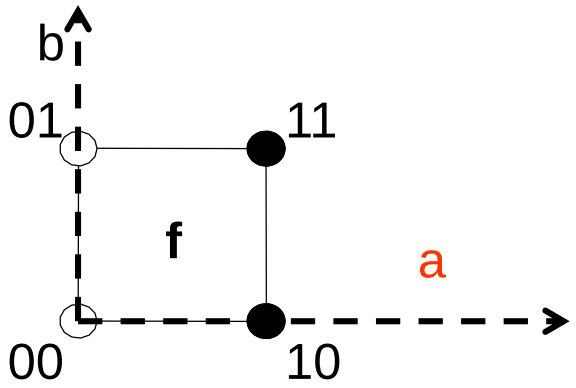
- **n input variables = n-dimensional "cube"**



Truth tables as Boolean cubes

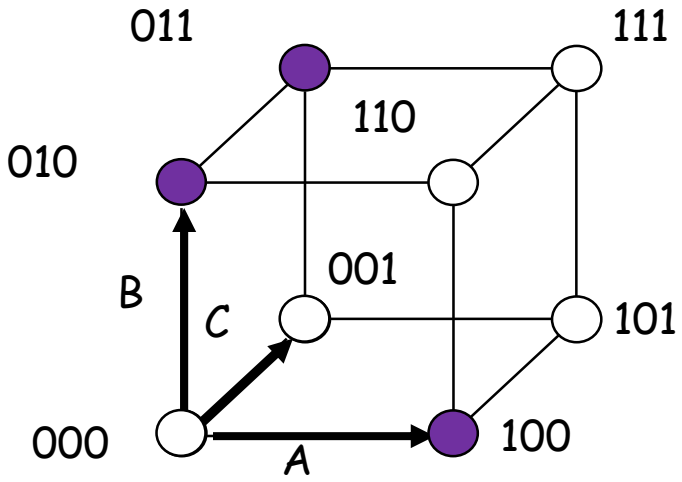
On-set - 1 nodes; **Off-set** - 0 nodes

a	b	f
0	0	0
0	1	0
1	0	1
1	**1**	1



a	b	f
0	-	0
1	-	1

	Inputs	Outputs
N	A B C	y
0	0 0 0	0
1	0 0 1	0
2	0 1 0	1
3	0 1 1	1
4	1 0 0	1
5	1 0 1	0
6	1 1 0	0
7	1 1 1	0

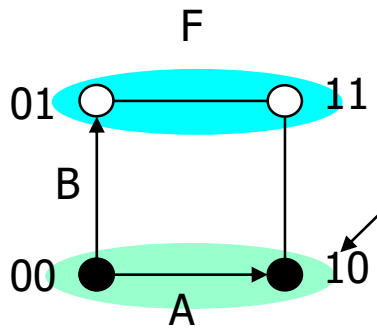


- **Join the nodes on the edge into one.**

  Here the output F does not depend on A, only on B

- Example:

A	B	F
0	0	1
0	1	0
1	0	1
1	1	0



two nodes of dimension 0, each describing only one output, are paired together in a pair (a dimension 1 segment)

A	B	F
-	0	1
-	1	0

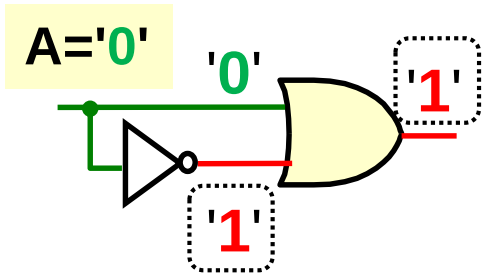
Theorems:

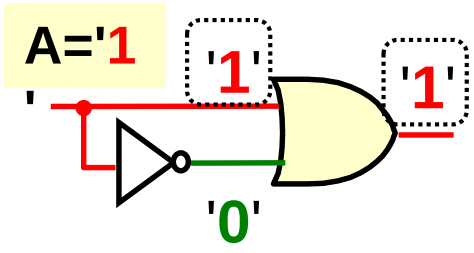
(not A and b) or (A and b) = b

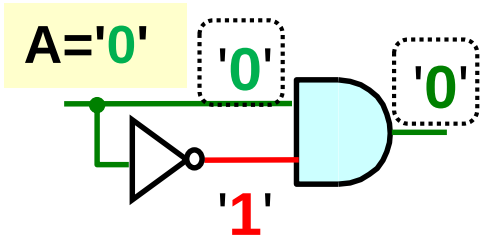
(not A or b) and (A or b) = b

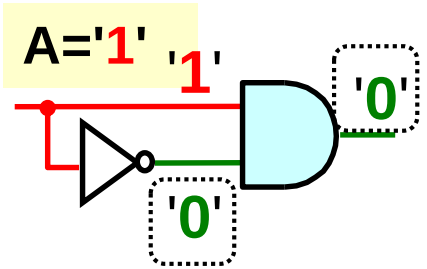
(not A and b) or (A and b) = (not A or A) and b = b

Postulate	*OR version*	*And the versions*
Complementation	$A + A' = '1'$	$A \bullet A' = '0'$



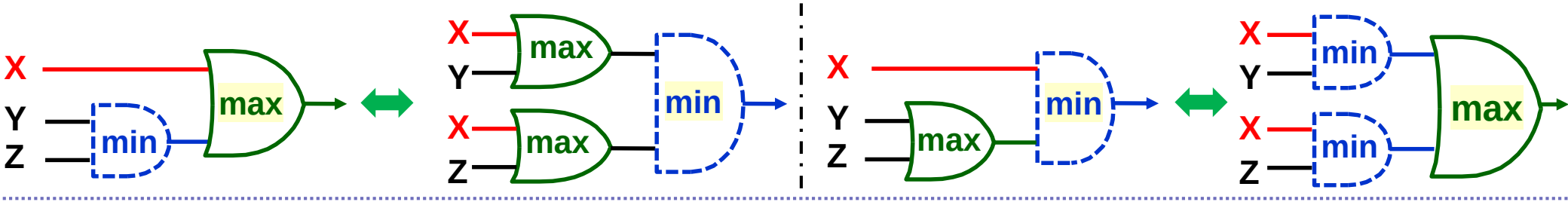




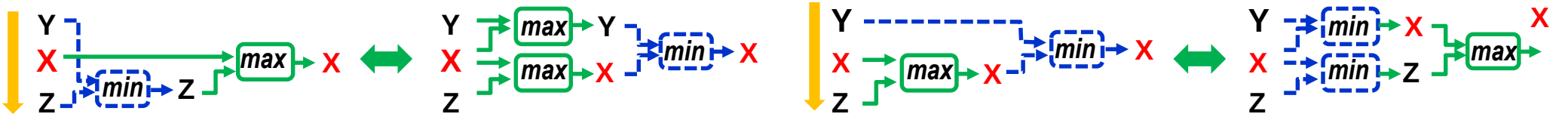


(not A or b) and (A or b) = (not A and A) or b = b

Postulate	*OR version*	*And the versions*
Distributability *Distributive Law*	$x + (y \bullet z) = (x + y) \bullet (x + z)$	$x \bullet (y + z) = (x \bullet y) + (x \bullet z)$

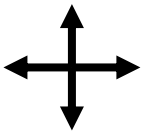


The proof for a selected order of input values



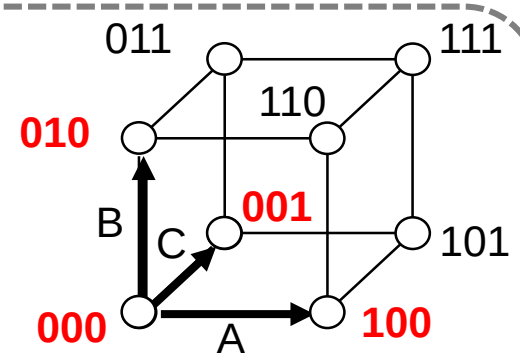
Karnaugh charts (KM)

- KM are Boolean cubes developed in the plane.
- The Hamiltonian path through all vertices of the cube without repetition forms the Gray code (Gray code ordering). On it, only one bit changes when moving horizontally or vertically in the map, e.g. 00, 01, 11, 10.
- Compliance with Gray's order is a prerequisite for using KM!

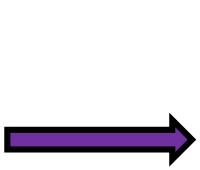


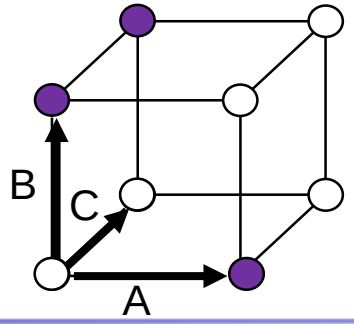
C \ A B	00	01	11	10
0	**000**	**010**	110	**100**
1	**001**	011	111	101

000
and its neighbors



C \ A B	00	01	11	10
0	0	1	0	1
1	0	1	0	0

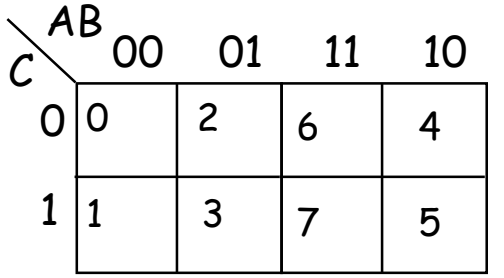


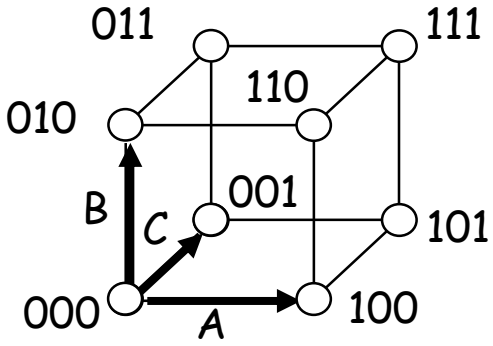


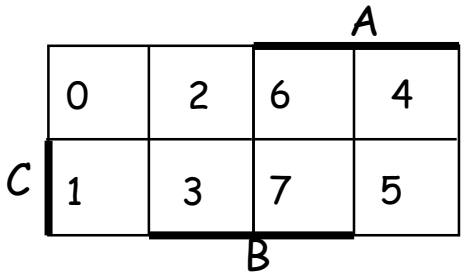
	Inputs	Outputs
N	A B C	y
0	0 0 0	0
1	0 0 1	0
2	0 1 0	1
3	0 1 1	1
4	1 0 0	1
5	1 0 1	0
6	1 1 0	0
7	1 1 1	0

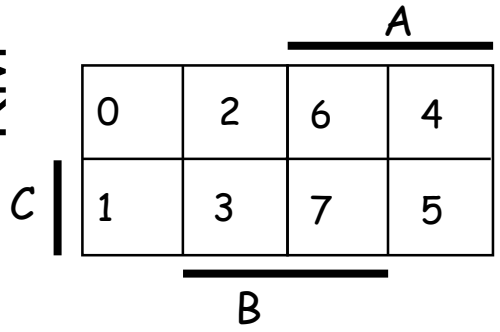
KM - Karnaugh maps in different styles

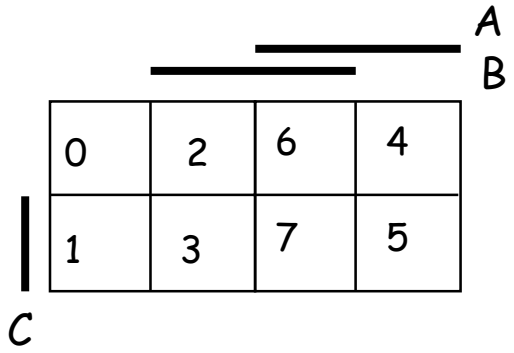
Different styles for KM











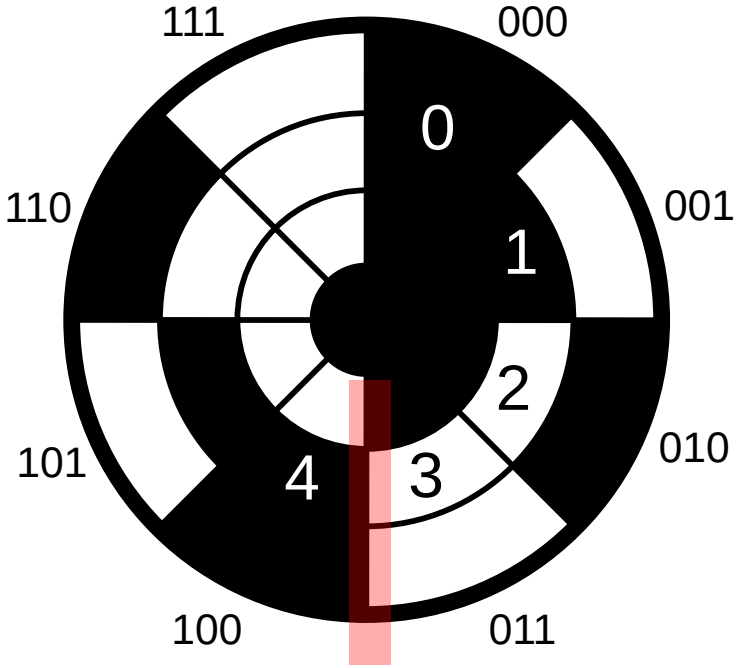
There are many possible Gray code and orders for numbers. If we begin from 0 number, we obtain:

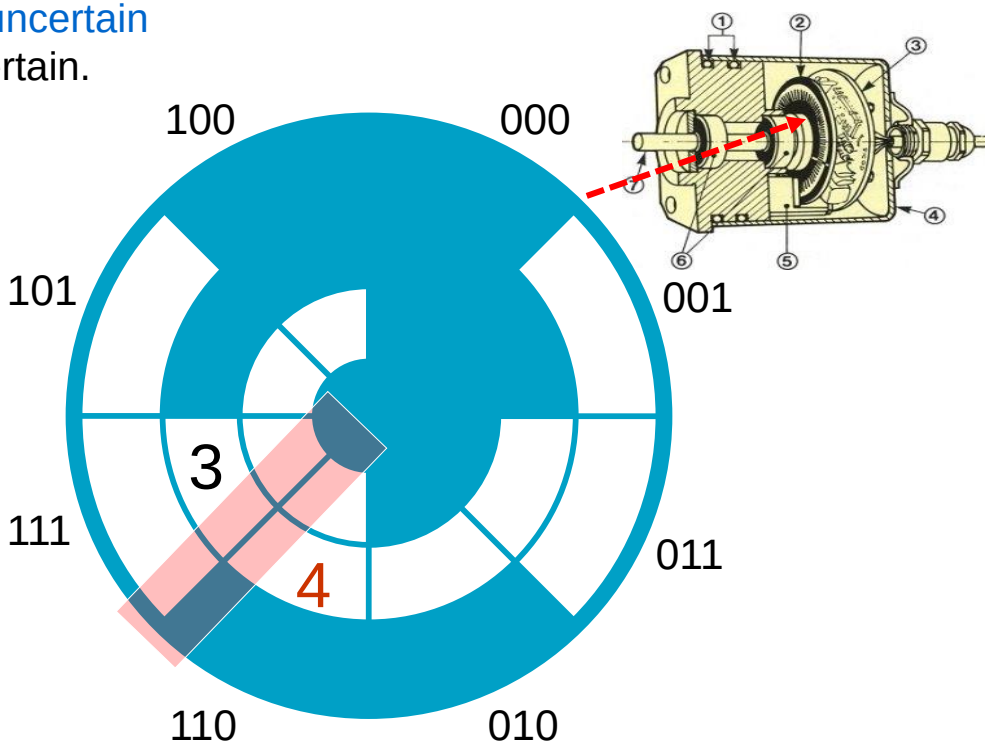
- **2** codes/orders of 2 bits binary numbers
  0 1 3 2, 0 2 3 1
- **18** codes/orders of 3 bits binary numbers
  0 1 3 2 6 7 5 4, 0 1 3 2 6 4 5 7, 0 1 3 7 5 4 6 2
  0 1 5 4 6 7 3 2, 0 1 5 4 6 2 3 7, 0 1 5 7 3 2 6 4
  0 2 3 1 5 4 6 7, 0 2 3 1 5 7 6 4, 0 2 3 7 6 4 5 1
  0 2 6 7 3 1 5 4, 0 2 6 4 5 7 3 1, 0 2 6 4 5 1 3 7
  0 4 5 7 6 2 3 1, 0 4 5 1 3 2 6 7, 0 4 5 1 3 7 6 2
  0 4 6 7 5 1 3 2, 0 4 6 2 3 1 5 7, 0 4 6 2 3 7 5 1
- **5712** codes/orders of 4 bits binary numbers.
- **5859364320** codes/orders of 5 bits binary numbers.

Note: For more details, you can search "Hamiltonian paths on hypercube" (i.e. n-dimensional cube)

Gray Code

- Gray codes are not suitable for computations, they are used for measurements or communications, such as optical encoders or digital TV.
- Fast changes between position 3 and 4:
  - Gray code: 111 <-> 110 - only 1 bit is uncertain
  - Binary: 100 <-> 011, all 3 bits are uncertain.





- Some systems use Gray code based ordering, i.e. numbers are sorted by the way that adjacent binary codes differ only in one bit, e.g. Karnaugh maps.
  (*adjacent = having a common boundary = cz:sousedící*)

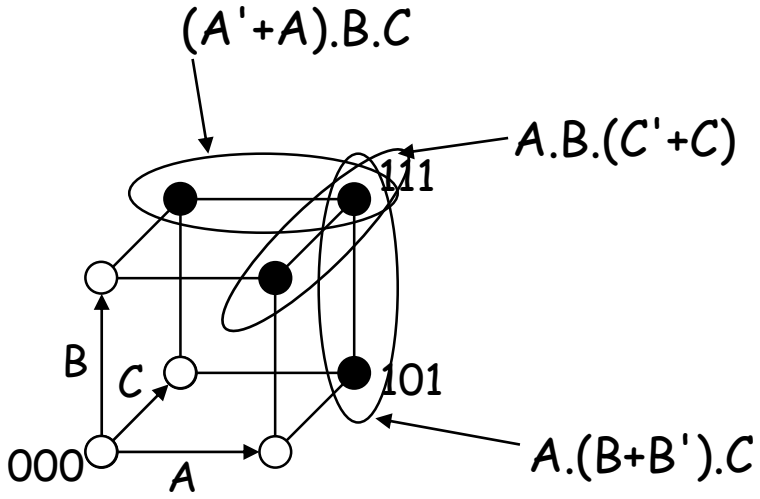
Special Codes

Dec.	Binary 8 4 2 1	Octal 421	Hex (VHDL syntax)	BCD binary / hex	One hot	Gray code	Gray code ordered numbers
0	0000	00	x”0”	0000 0000 = x"00"	0000 0000 0000 0001	0000	0
1	0001	01	x”1”	0000 0001 = x"01"	0000 0000 0000 0010	0001	1
2	0010	02	x”2”	0000 0010 = x"02"	0000 0000 0000 0100	0011	3
3	0011	03	x”3”	0000 0011 = x"03"	0000 0000 0000 1000	0010	2
4	0100	04	x”4”	0000 0100 = x"04"	0000 0000 0001 0000	0110	6
5	0101	05	x”5”	0000 0101 = x"05"	0000 0000 0010 0000	0111	7
6	0110	06	x”6”	0000 0110 = x"06"	0000 0000 0100 0000	0101	5
7	0111	07	x”7”	0000 0111 = x"07"	0000 0000 1000 0000	0100	4
8	1000	10	x”8”	0000 1000 = x"08"	0000 0001 0000 0000	1100	12
9	1001	11	x”9”	0000 1001 = x"09"	0000 0010 0000 0000	1101	13
10	1010	12	x”A”	0001 0000 = x"10"	0000 0100 0000 0000	1111	15
11	1011	13	x”B”	0001 0001 = x"11"	0000 1000 0000 0000	1110	14
12	1100	14	x”C”	0001 0010 = x"12"	0001 0000 0000 0000	1010	10
13	1101	15	x”D”	0001 0011 = x"13"	0010 0000 0000 0000	1011	11
14	1110	16	x”E”	0001 0100 = x"14"	0100 0000 0000 0000	1001	9
15	1111	17	x”F”	0001 0101 = x"15"	1000 0000 0000 0000	1000	8

Always only 1-bit changes between adjacent codes

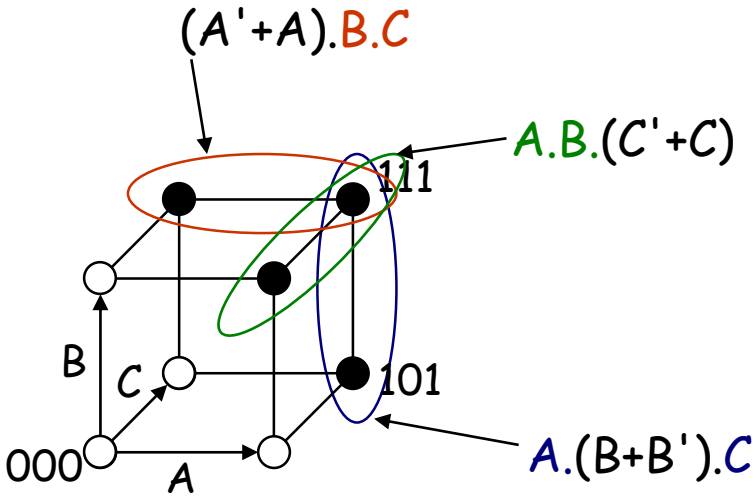
- **Majority function** (aka **median operator**) has *n* inputs and one output which value is false when *n*/2 or more inputs are false, and true otherwise.
- Question: Design 3-bit majority function:

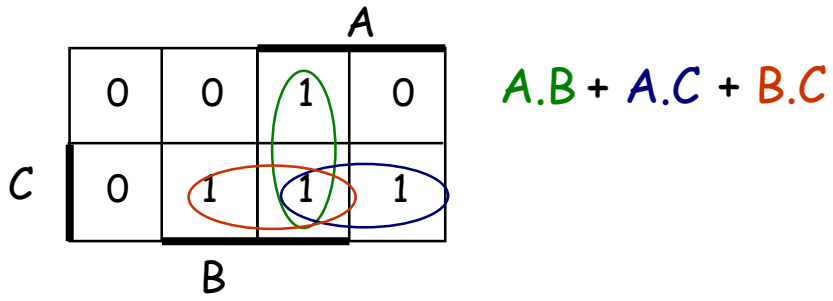
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



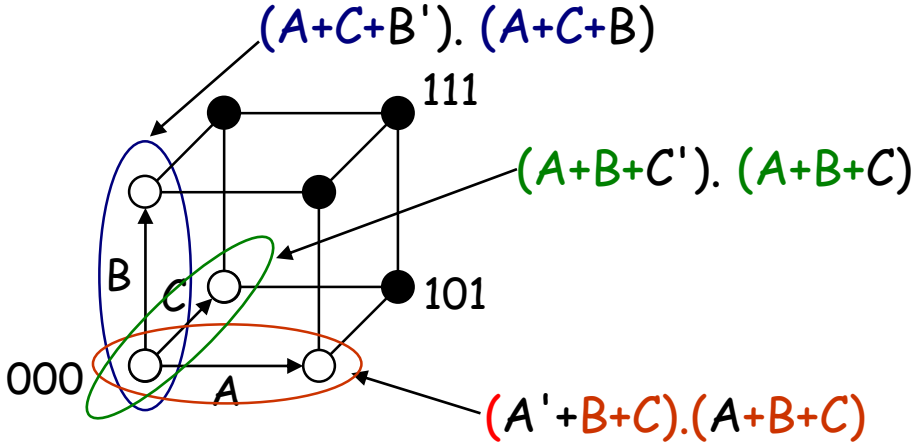
$$Y = B.C+A.B+A.C$$

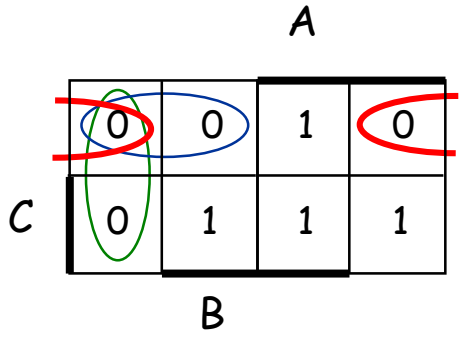
Karnaugh map versus cube





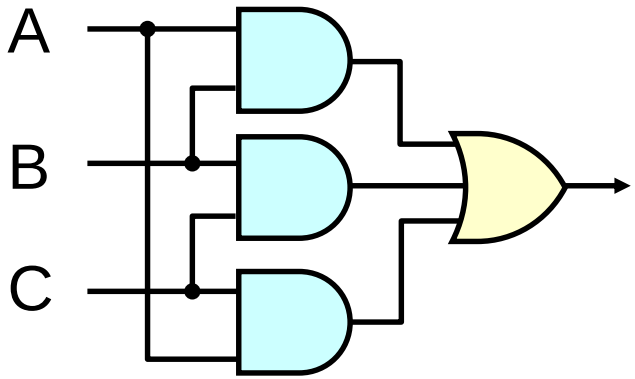
A.B + A.C + B.C





(A+B) . (B+C) . (A+C)

Majority of 3 in the diagram

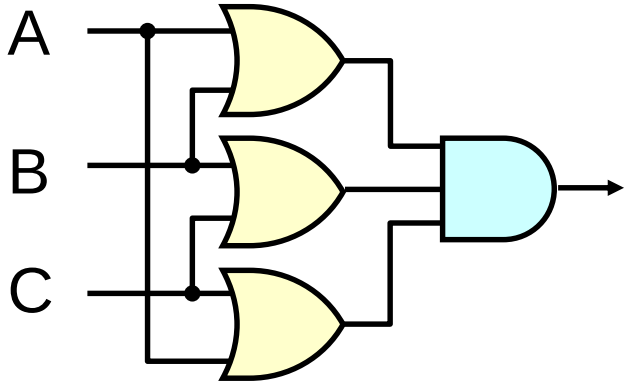


'1' when at least two = '1'

(A and B)

or (B and C)

or (C and A)



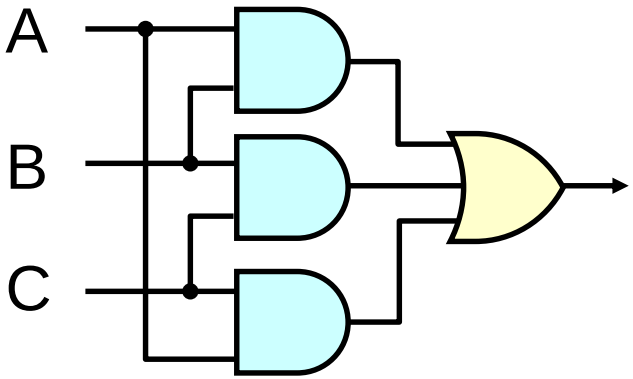
'0' when at least two = '0'

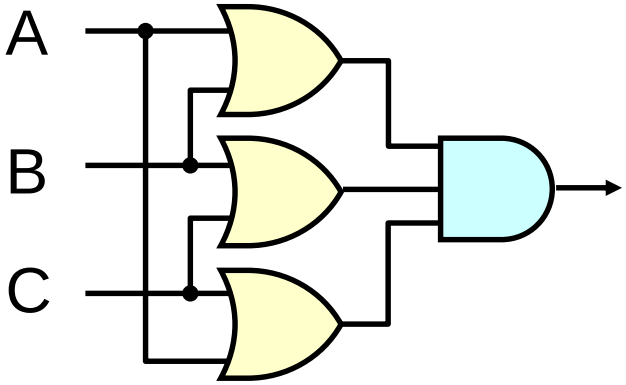
(A or B)

and (B or C)

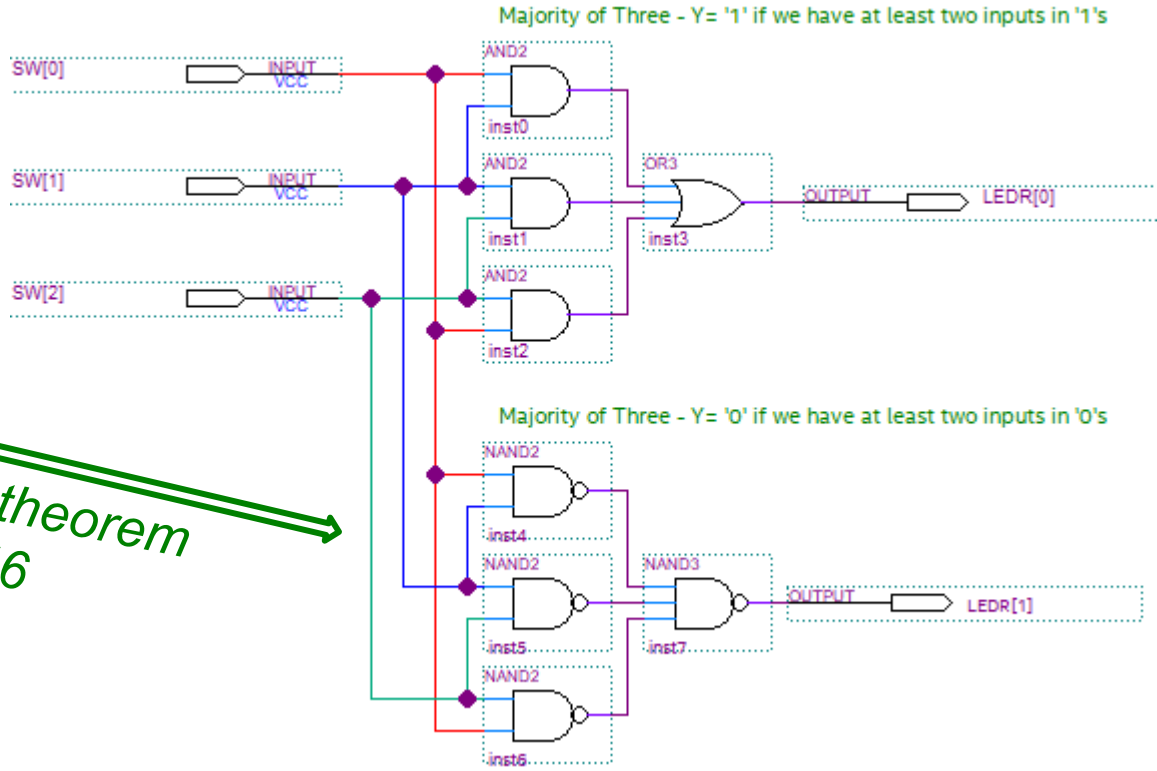
and (C or A)

Majority of 3 versions





De Morgan theorem
slide 46

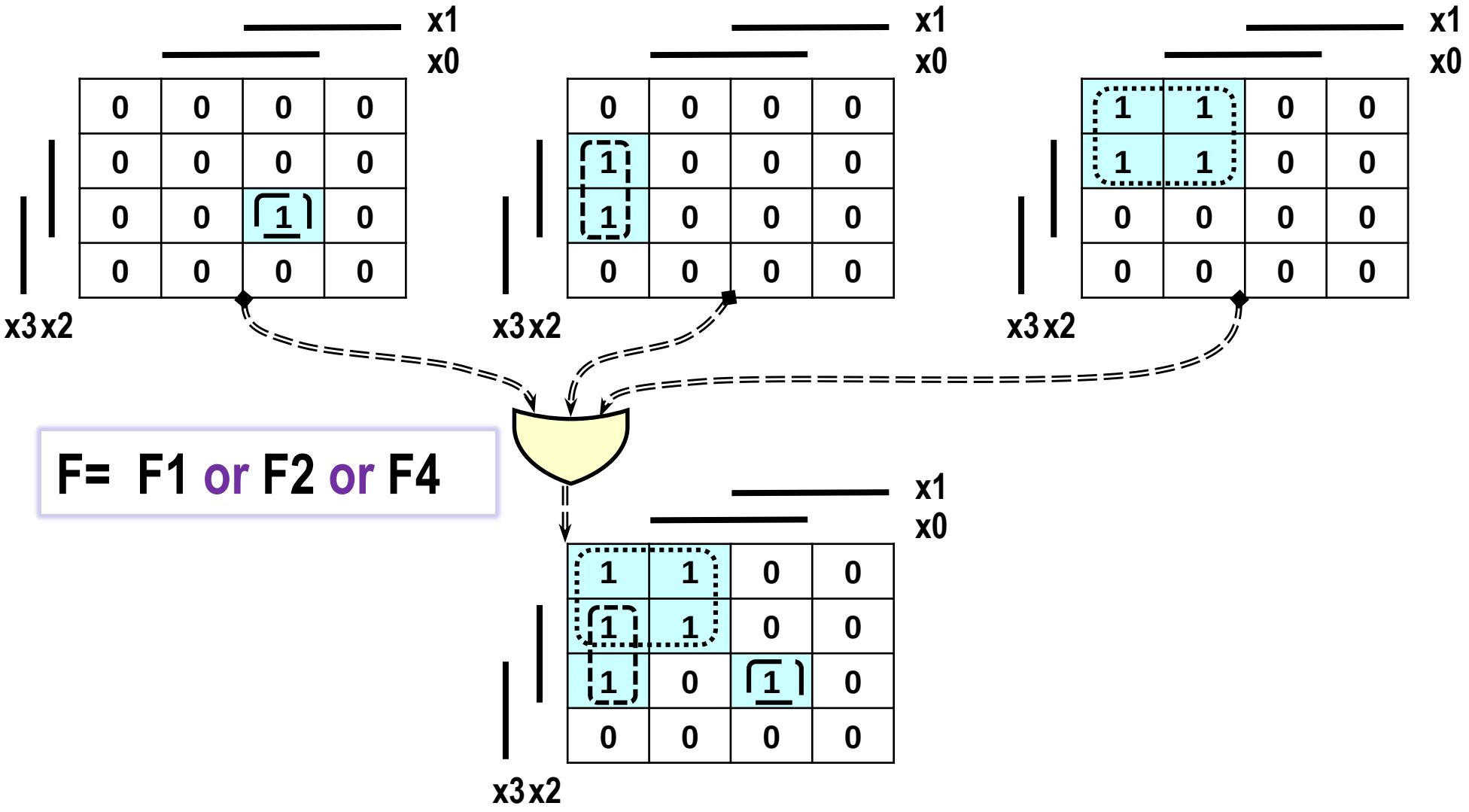


Principle SoP = Sum Of Products

F1= (x0 and x1 and x2 and x3)

F2= (not x0 and not x1 and x2)

F4 = (not x1 and not x3)



Principle PoS = Product Of Sums

F1= (x0 or x1 or x2 or x3)

x3 x2 \ x1 x0	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

F2= (not x0 or not x2 or x3)

x3 x2 \ x1 x0	00	01	11	10
00	1	1	1	1
01	1	0	0	1
11	1	1	1	1
10	1	1	1	1

F4 = (not x1 or not x2)

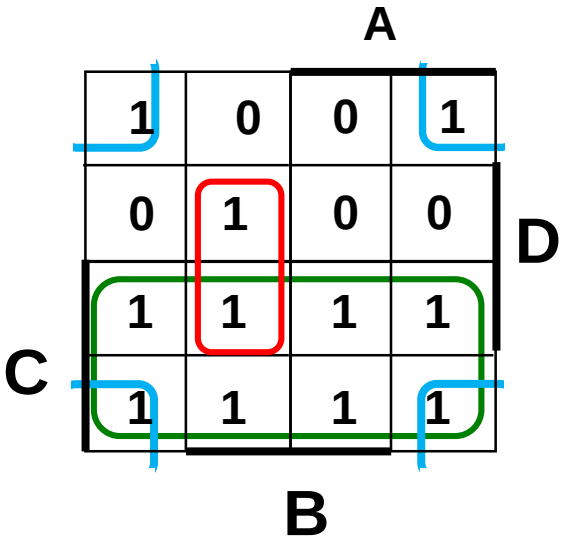
x3 x2 \ x1 x0	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	1	1	0	0
10	1	1	1	1

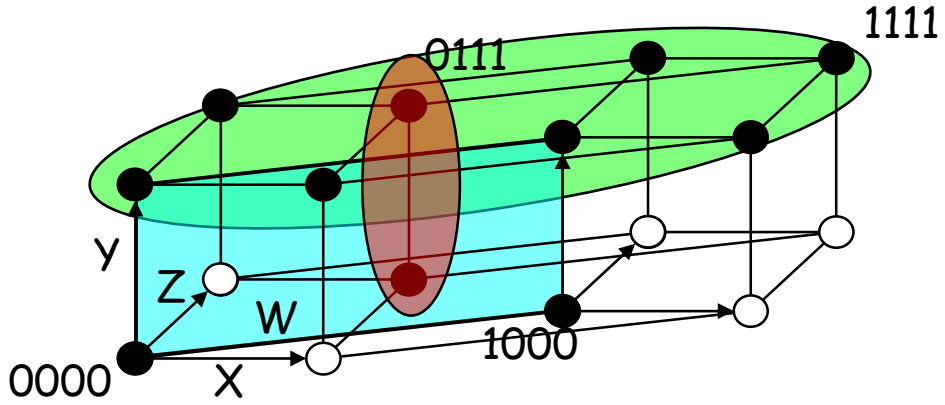
F= F1 and F2 and F4

x3 x2 \ x1 x0	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	1	1	0	0
10	1	1	1	1

- F(A,B,C,D) = m(0,2,3,5,6,7,8,10,11,14,15)

F = C + (A'BD)+ (B'D')

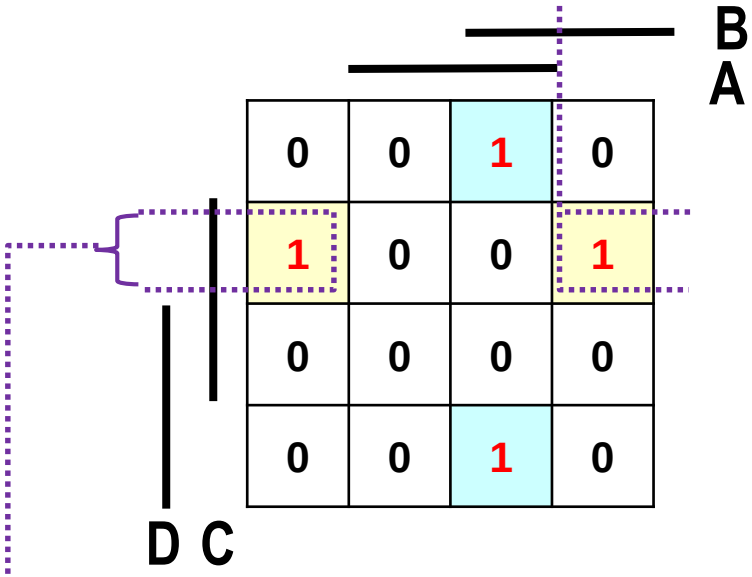


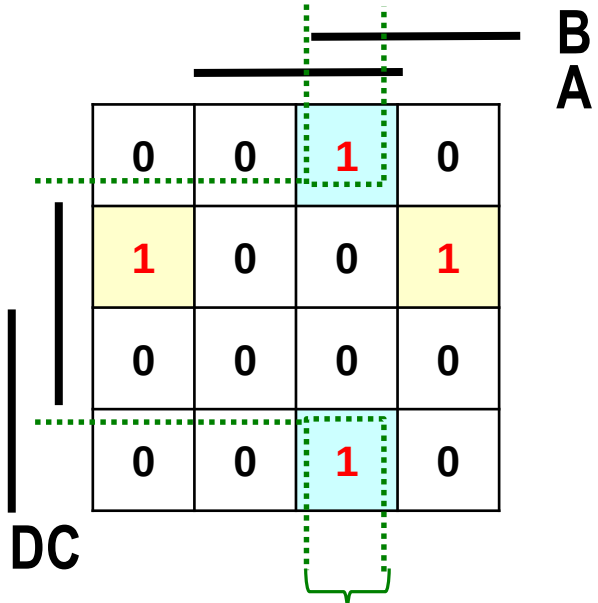


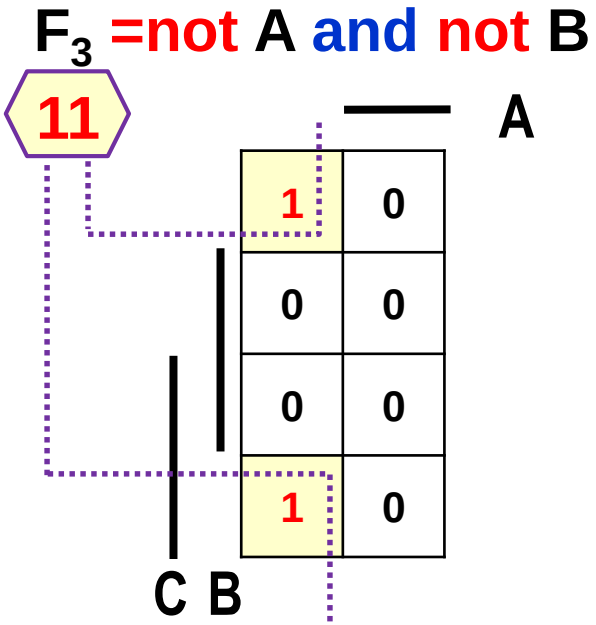
We try to find the minimum number of coverage.

Note: Each coverage includes some number of fields determined by a power of 2 - i.e. 1, 2, 4, 8, 16, etc.

Continuity across the edges



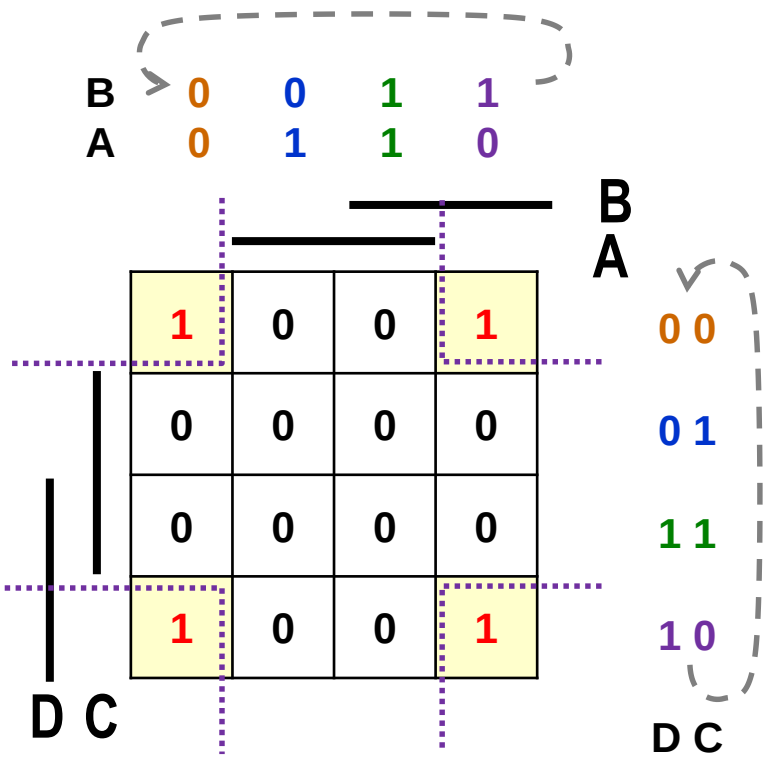


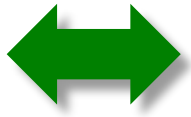


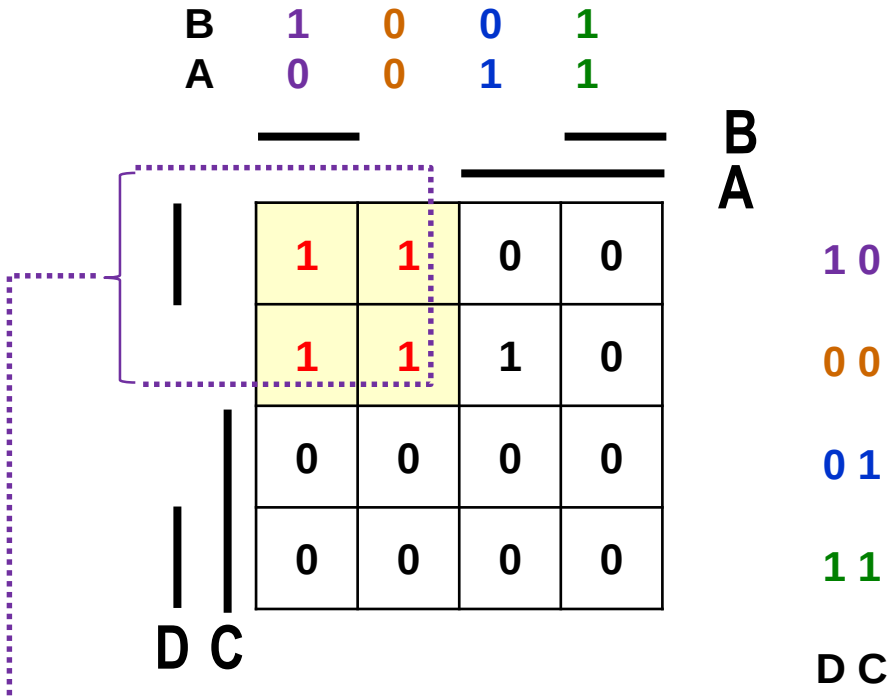
11 F_1 = not A and C and not D

11 F_2 =A and B and not C

Connection through the corners







1111 F_4 = not A and not C

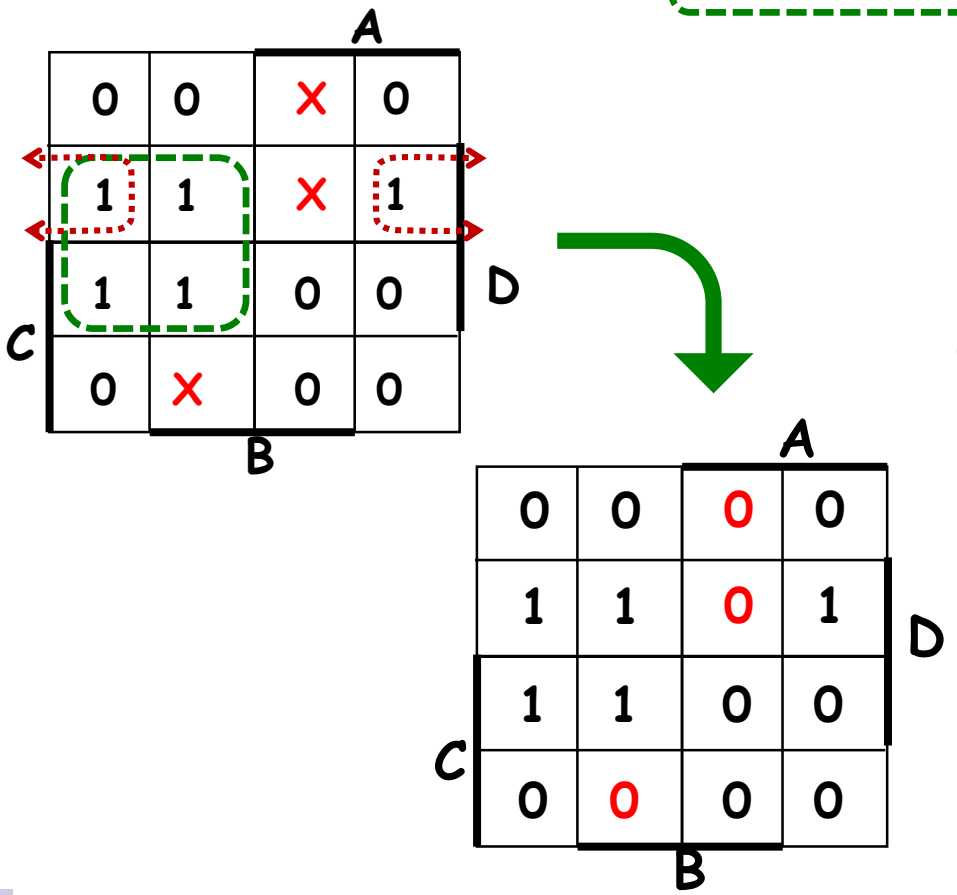
1111 F_5 = not A and not C

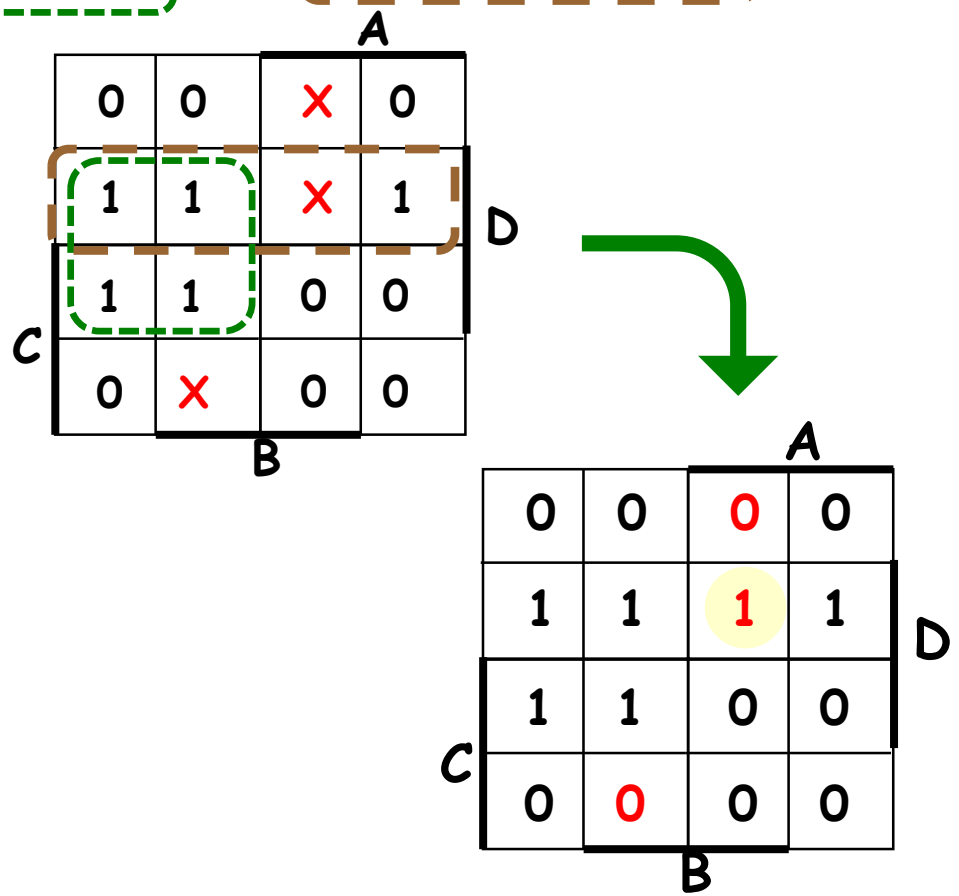
KM and "don't care" states: 0, 1, x

f(A,B,C,D) = m(1,3,5,7,9) + dc(6,12,13)

without "don't care": (not A and D) or (not B and not C and D)

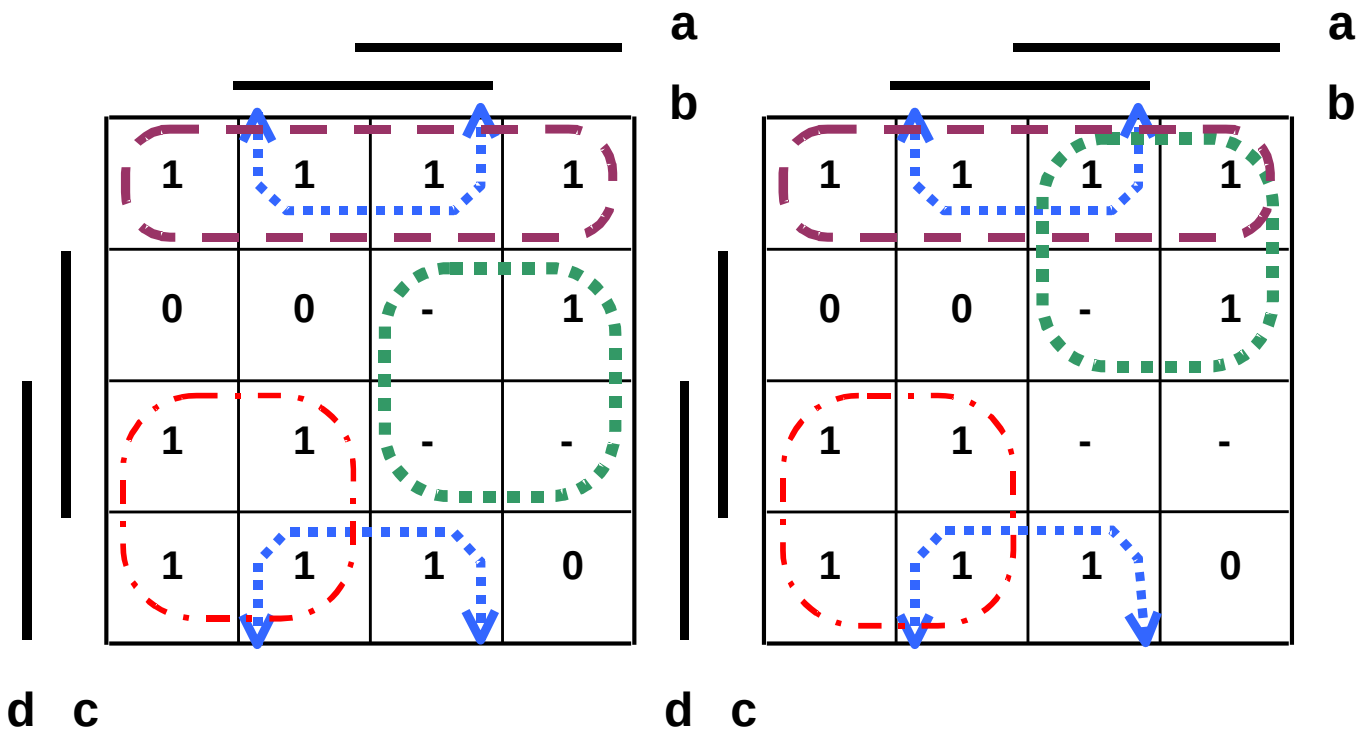
with "don't care": (not A and D) or (not C and D)





Karnaugh maps: don't cares

dcba	Y
0000	1
0001	1
0010	1
0011	1
0100	0
0101	1
0110	0
0111	X
1000	1
1001	0
1010	1
1011	1
1100	1
1101	X
1110	1
1111	X



$$F_{left}(x) = \bar{c}.\bar{d} + b.\bar{c} + a.c + \bar{a}.d \qquad F_{right}(x) = \bar{c}.\bar{d} + b.\bar{c} + a.\bar{d} + \bar{a}.d$$

$$F_{zeros}(x) = (a + \bar{c} + d).(\bar{a} + b + \bar{d})$$

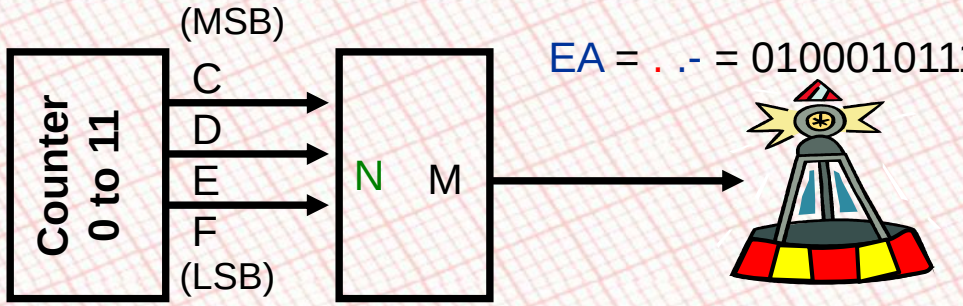
$$F_{left}(x) \neq F_{right}(x) \neq F_{zeros}(x);$$

$$F_{left}(x) = F_{right}(x) = F_{zeros}(x); \ [\forall x; F_n(x) = 0 \vee 1]$$

If we cover
don't care states
- SoP thermae, will be 1.
- PoS therms, they'll be 0.

Morse Beacon EAEA...

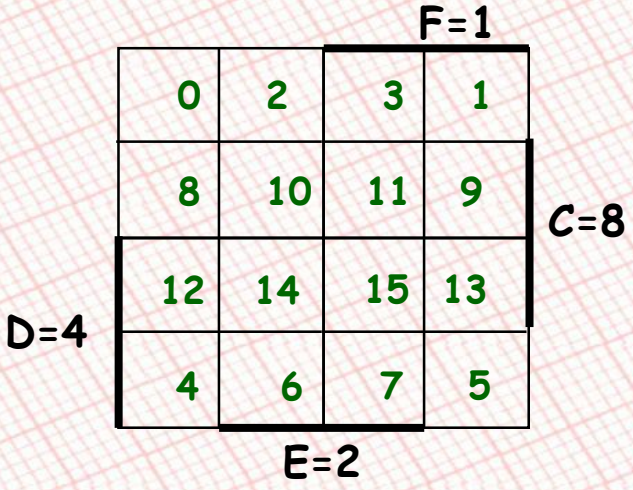
	8	4	2	1	
N	C	D	E	F	M
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12-15	1	1	-	-	-

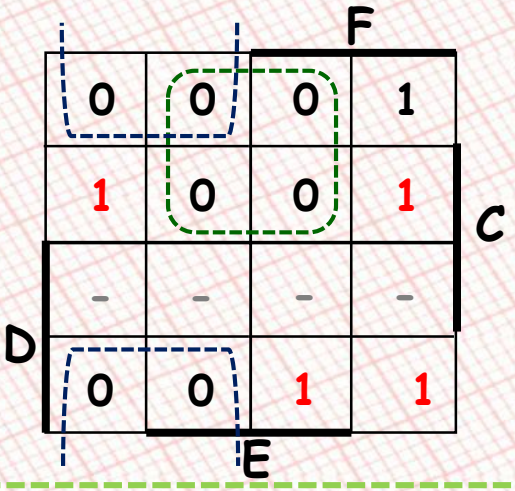


EA = . .- = 010001011100

Step 1 *- rewrite the table into KM according to the order o*

Step 2 - cover KM





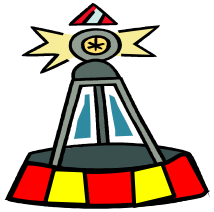
P-o-S: **M = (F+C) . (E'+D)**

M<= (F or C) and (not E or D);

Step 3: You can create your own S-O-P beacon coverage.

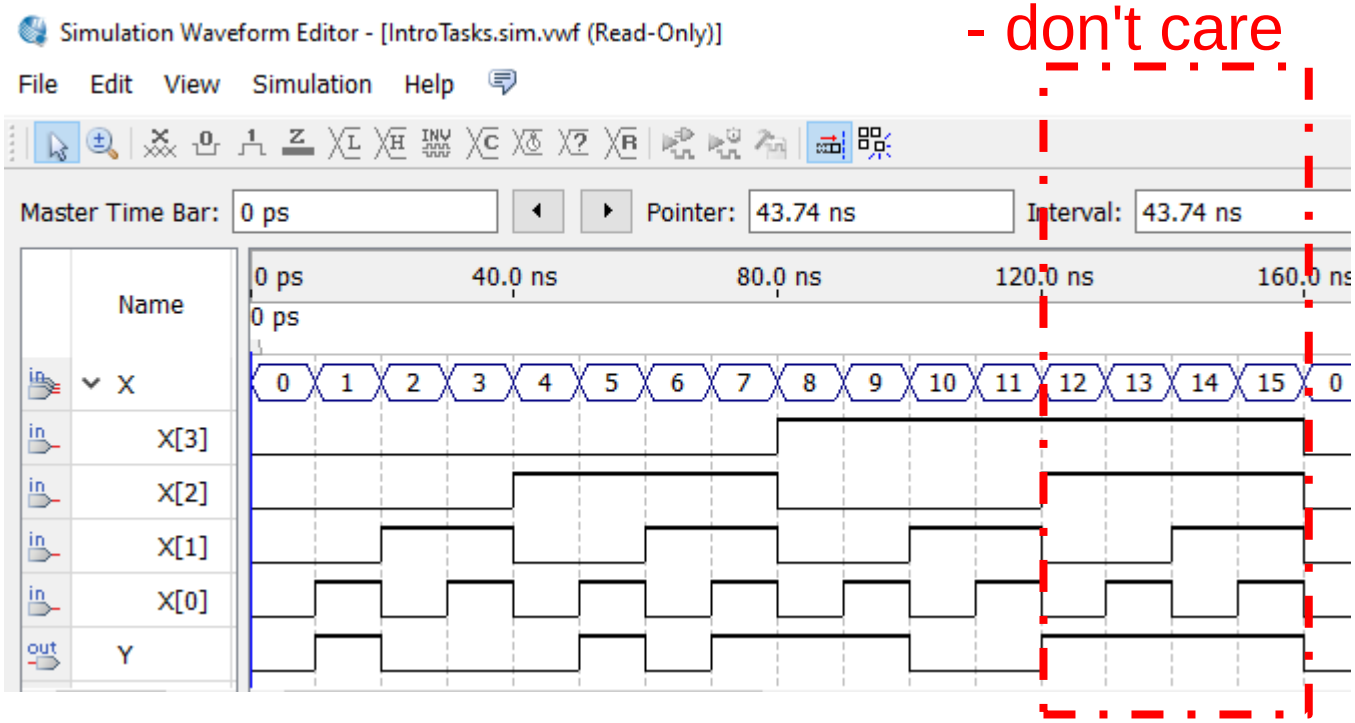
Testing an expression in VHDL

```
library ieee; use ieee.std_logic_1164.all;
entity MorseEA is port ( X : in std_logic_vector(3 downto 0);
                         Y : out std_logic );
end entity;
architecture behavioral of MorseEA is
begin
        Y <= (X(0) or X(3)) and (not X(1) or X(2));
end architecture;
```



P-o-S: **(F or C) and (not E or D);**

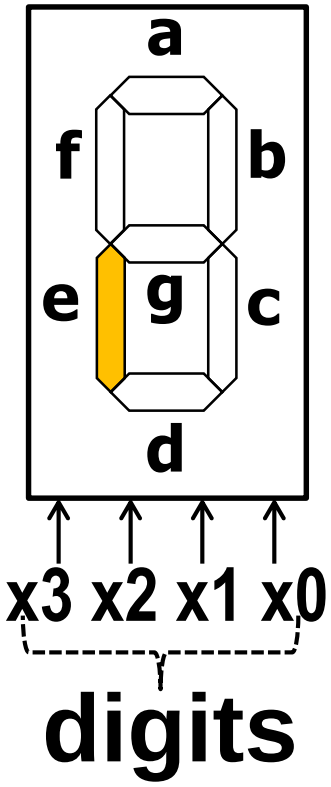
Simulation

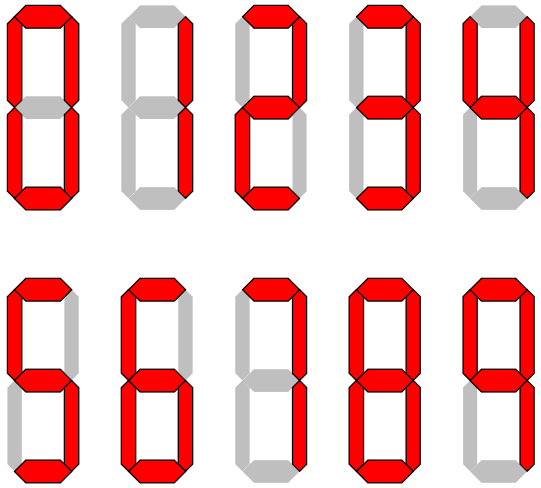


Karnaugh's maps

Examples

Example: minimize e-LEDs



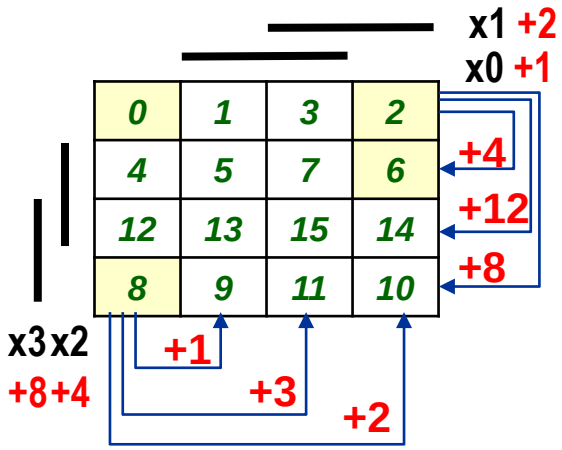


N	x:3210	e
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10-11	101-	x
12-15	11--	x

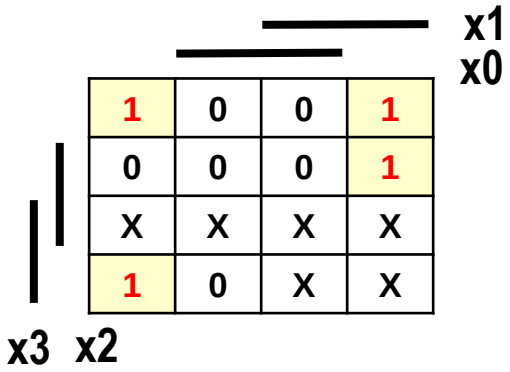
Example: minimize e-LEDs

N	x:3210	e
0	0000	1
1	0001	0
2	0010	1
3	0011	0
4	0100	0
5	0101	0
6	0110	1
7	0111	0
8	1000	1
9	1001	0
10-11	101-	x
12-15	11--	x

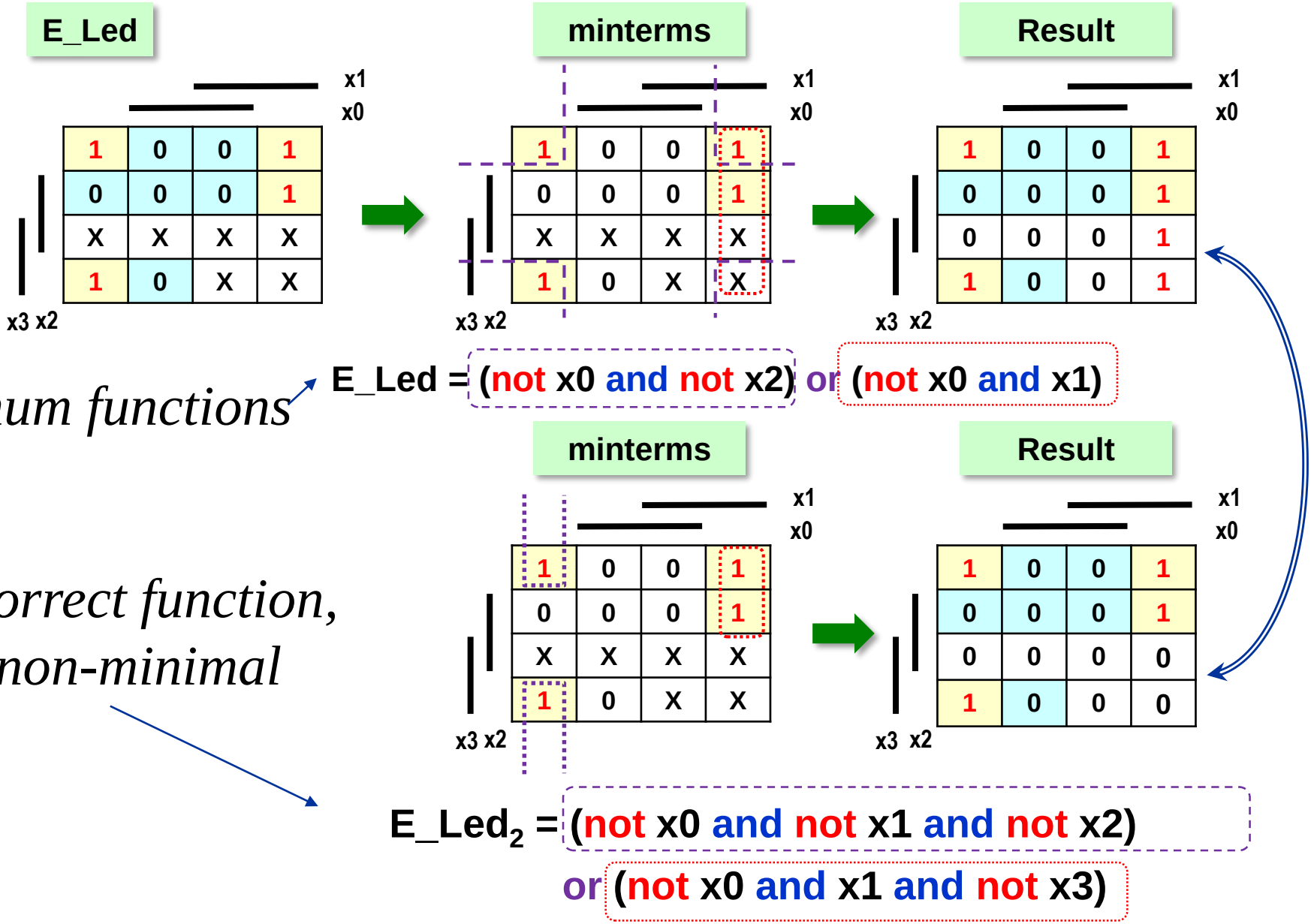
Auxiliary KM



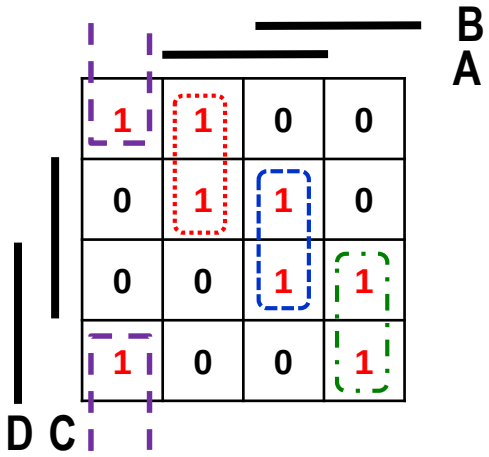
Final KM
e-LED



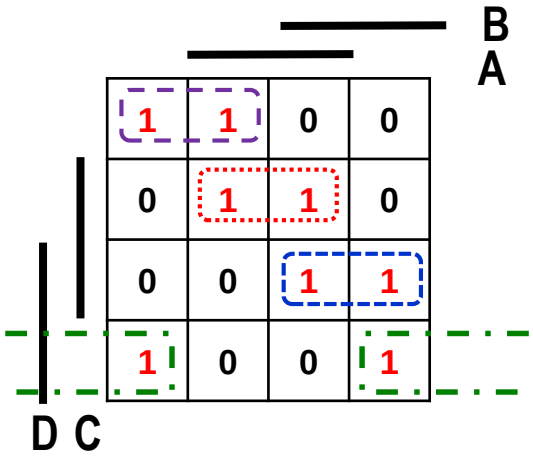
Usage don't care



Sometimes more possible minimum coverage



$F4_{13}$ = (not A and not B and not C)
or (A and not B and not D)
or (A and B and C)
or (not A and B and D)



$F4_{13}$ = (not B and not C and not D)
or (A and C and not D)
or (B and C and D)
or (not A and not C and D)

Mini-question: Are the functions listed the same?

x1<=(A and not C) or (C and not A);

x2<=(A and not B) xor (A and C);

x3<=(A or C) and (not A or not C);

x4<=(A xor C) or (A and not C);

Mini-question: Are the functions listed the same?

x1<=(A and not C) or (C and not A);

x2<=(A and not B) xor (A and C);

x3<=(A or C) and (not A or not C);

x4<=(A xor C) or (A and not C);

x2

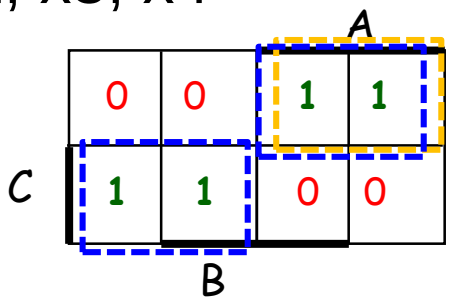
C \ A	0	0	0	1
0	0	0	1	0
1	0	0	1	0

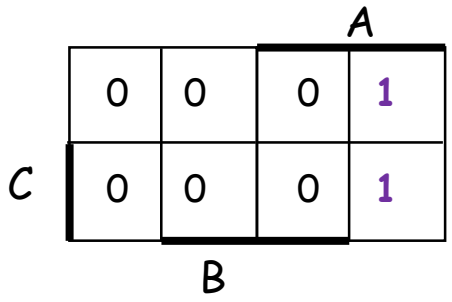
B

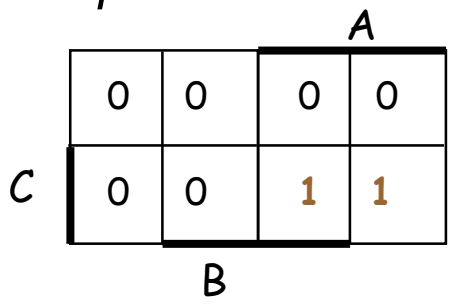
We solve KM

xor of maps

x1, x3, x4

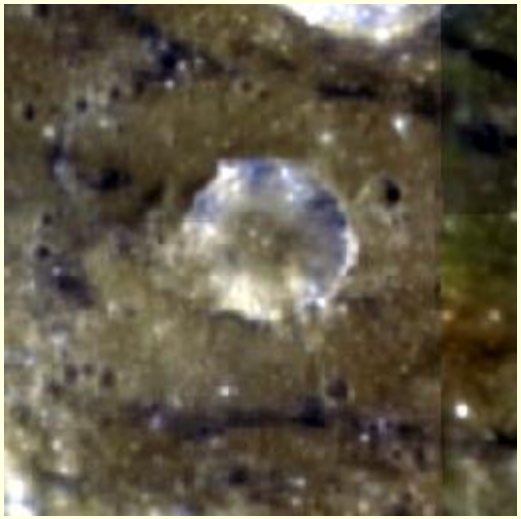






De Morgan

also with the crater

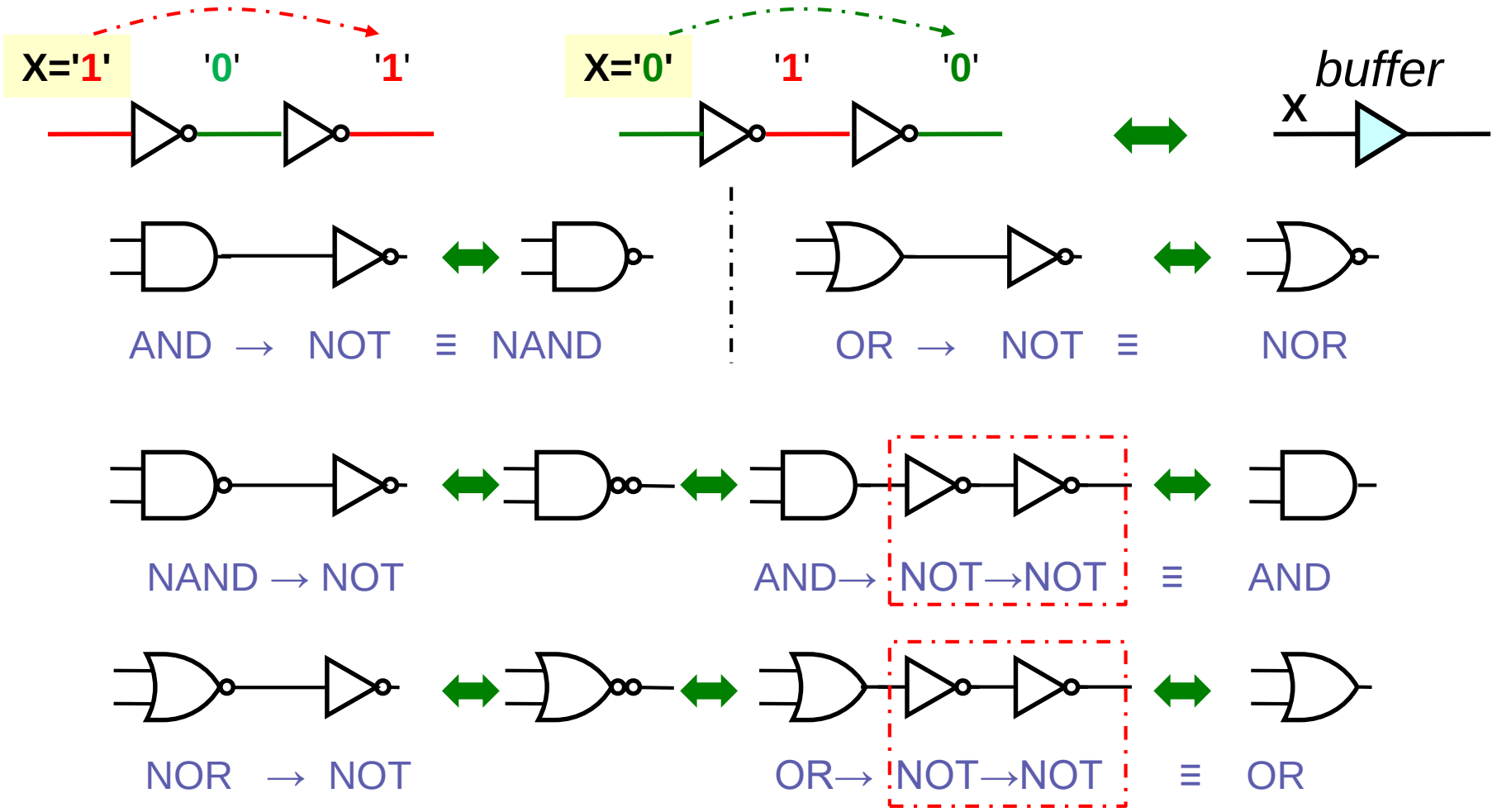


Negations of Logic

Source of image De Morgan (crater). (2022, February 24) In *Wikipedia.* https://en.wikipedia.org/wiki/De_Morgan_(crater)

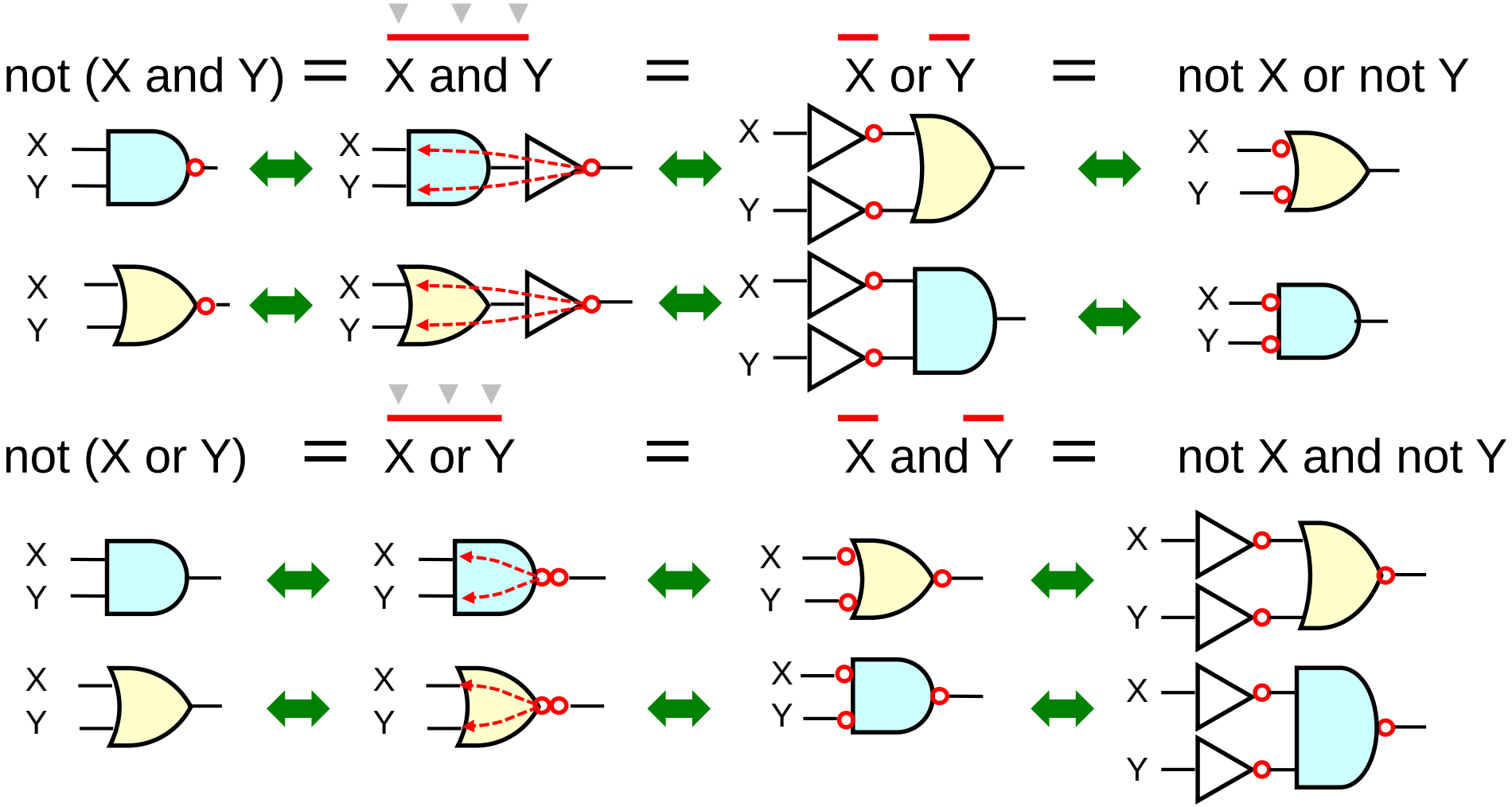
Double Negation

Theorem	
Double Negation	not (not x) = x

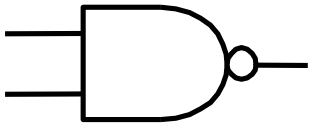
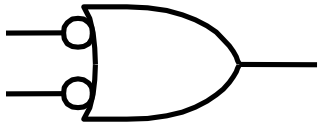
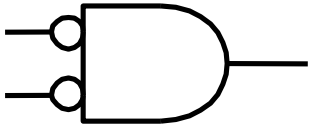
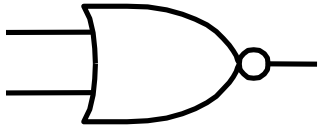
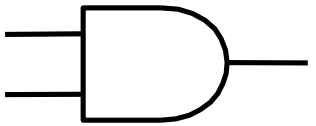
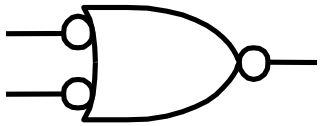
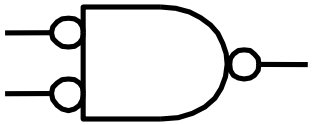
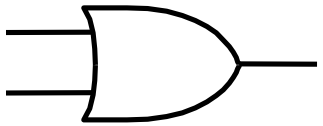


DeMorgan

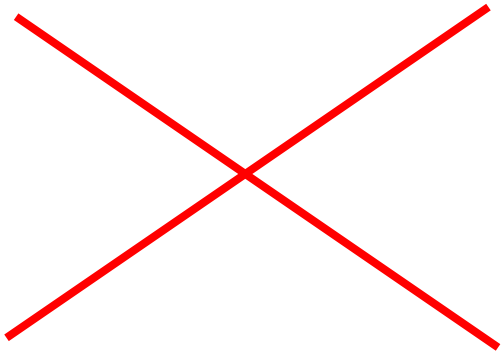
Theorem	*OR version*	*And version*
DeMorgan	not (x + y) = not x ● not y	not (x ● y) = not x + not y

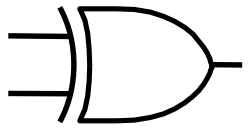


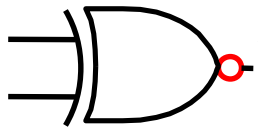
DeMorgan' Equivalents of Gates

Operation	AND		OR
NAND		=	
NOR		=	
AND		=	
OR		=	

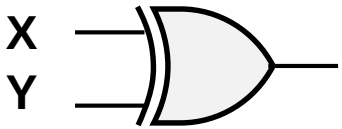
DeMorgan Rule does not apply for XOR



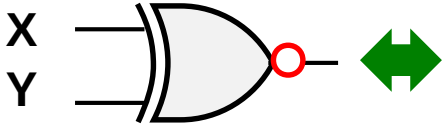
XOR  $Y = a \oplus b$

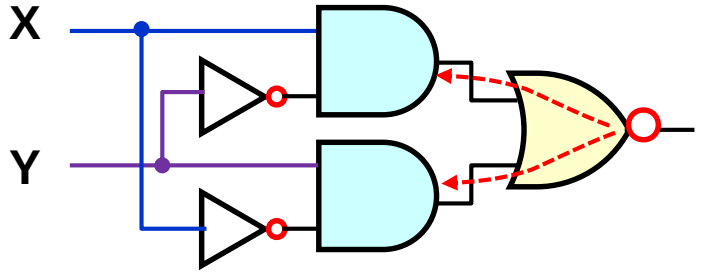
XNOR  Y=(a ≡ b)

X xor Y

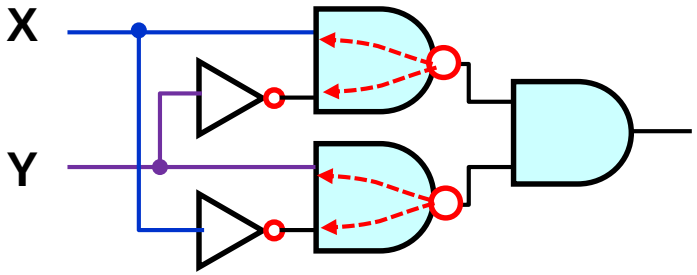


X xor Y = (X and not Y) or (Y and not X)

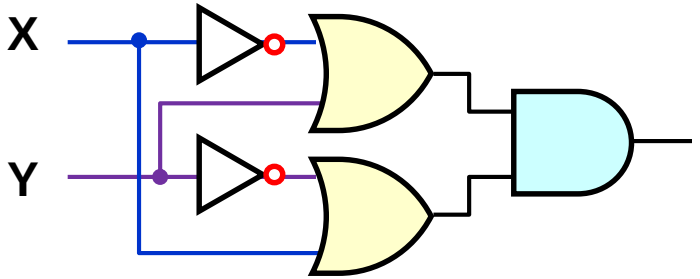




X xnor Y



X xnor Y

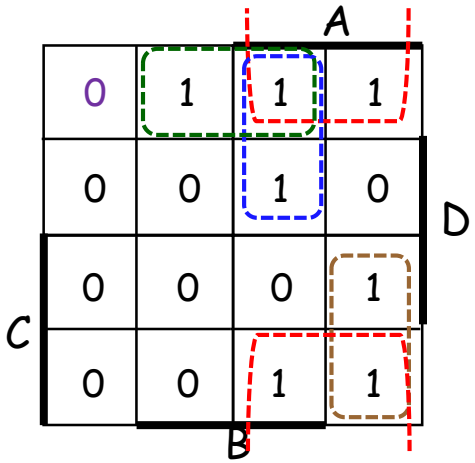


DeMorgan's theorem follows from KM

De Morgan's theorem is SoP and PoS coverage of the negated elements in the same positions

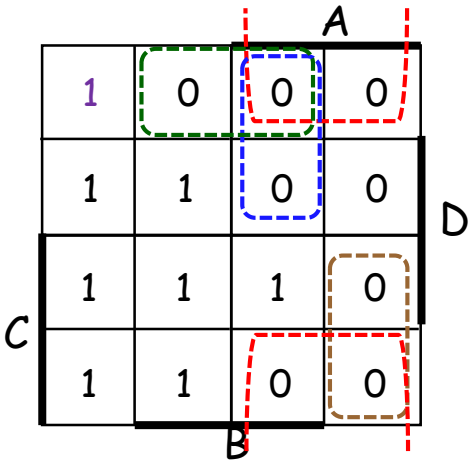
SoP: Detector of primes

$F(w,x,y,z) = m(1,2,3,5,7,11,13)$



PoS: Detector of non-primes

$F'(w,x,y,z) = M(0,4,6,8,9,10,12,14,15)$



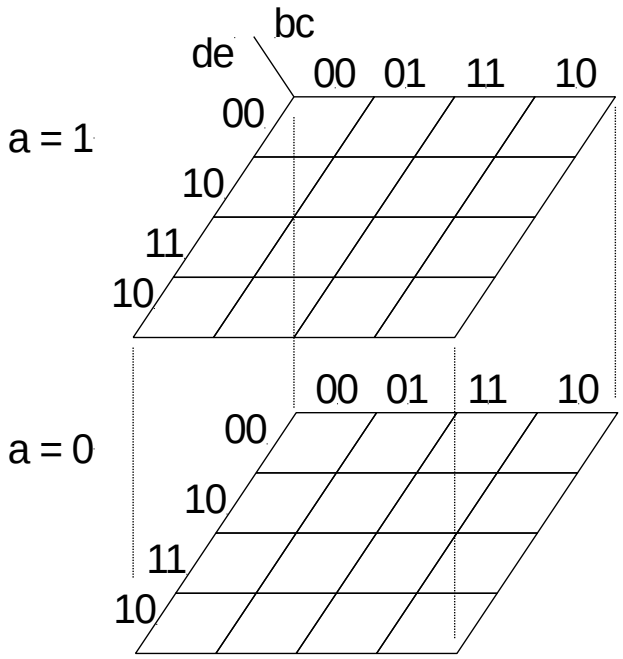
(A'+D).(A'+B+C').(A'+B'+C).(B'+C+D)

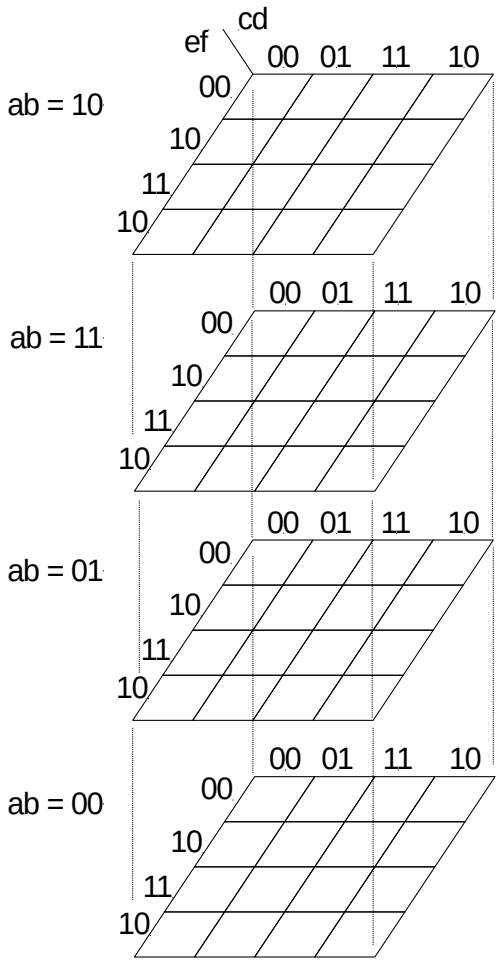
A.D'+A.B'.C+A.B.C'+B.C'.D'

Karnaugh Maps

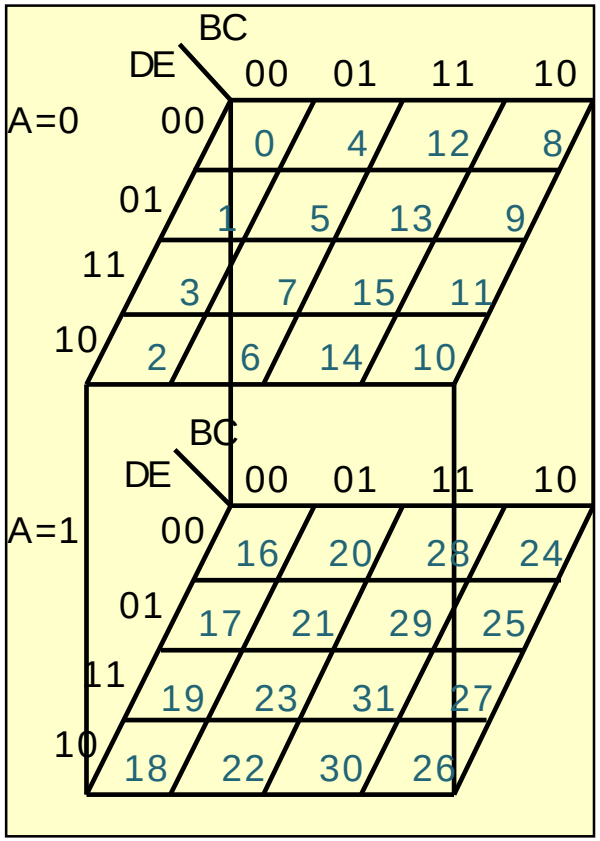
Greather than 4x4

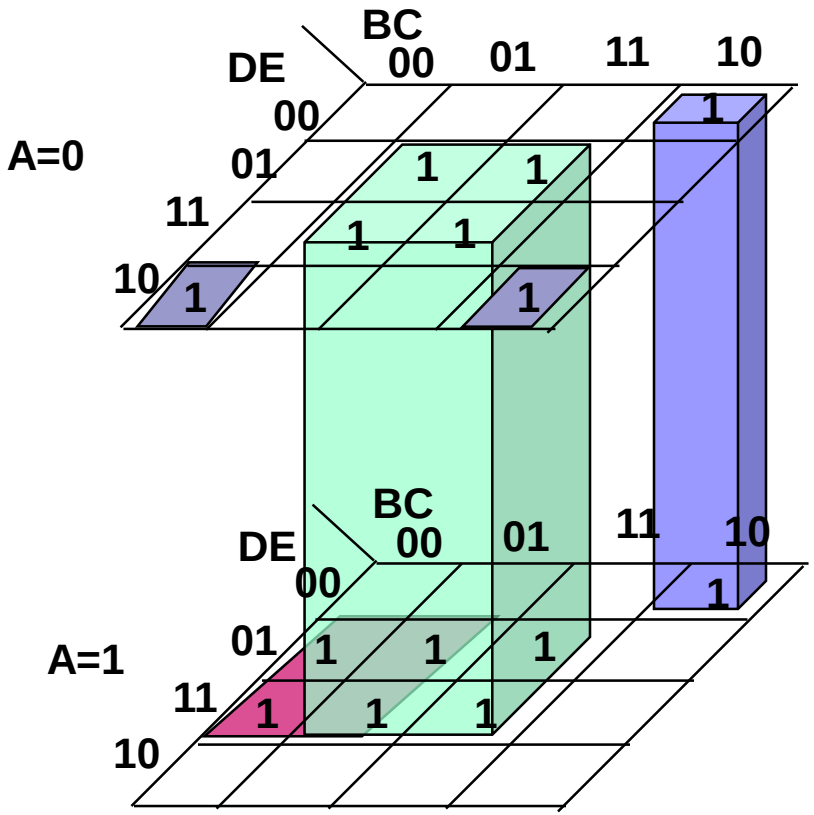
Higher Dimensional K-maps





5-Variable K-maps



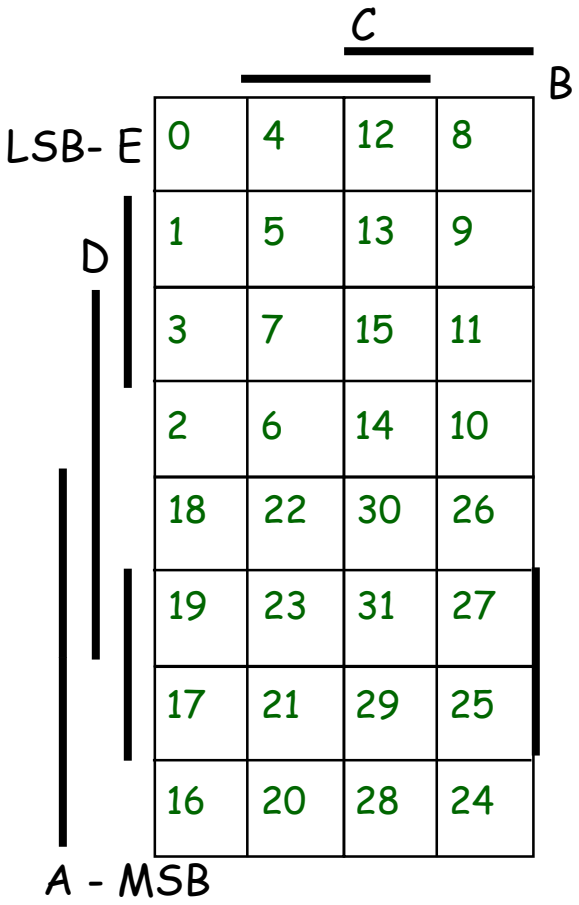


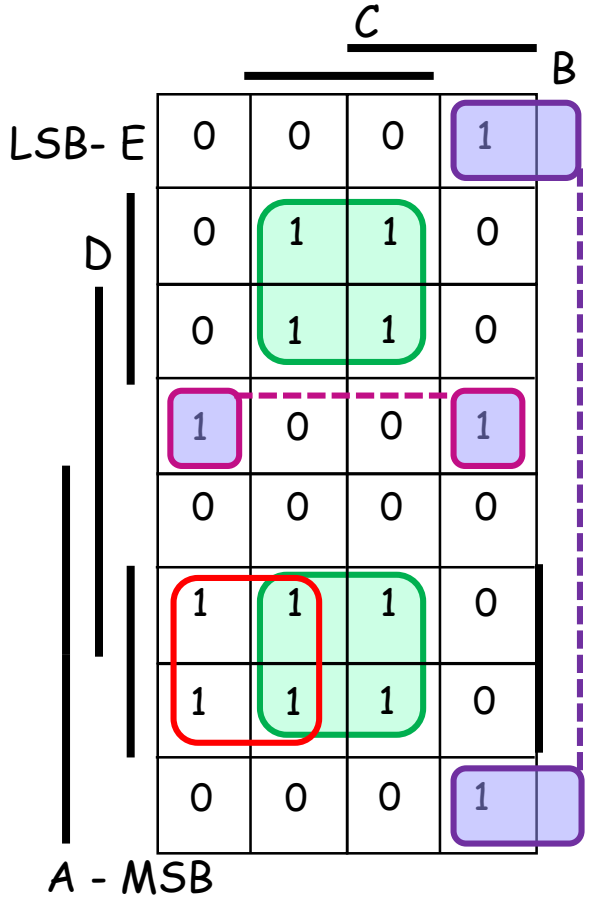
$f(A,B,C,D,E) = \Sigma m(2,5,7,8,10, 13,15,17,19,21,23,24,29\ 31)$

$= C\,E + A\,B'\,E + B\,C'\,D'\,E' + A'\,C'\,D\,E'$

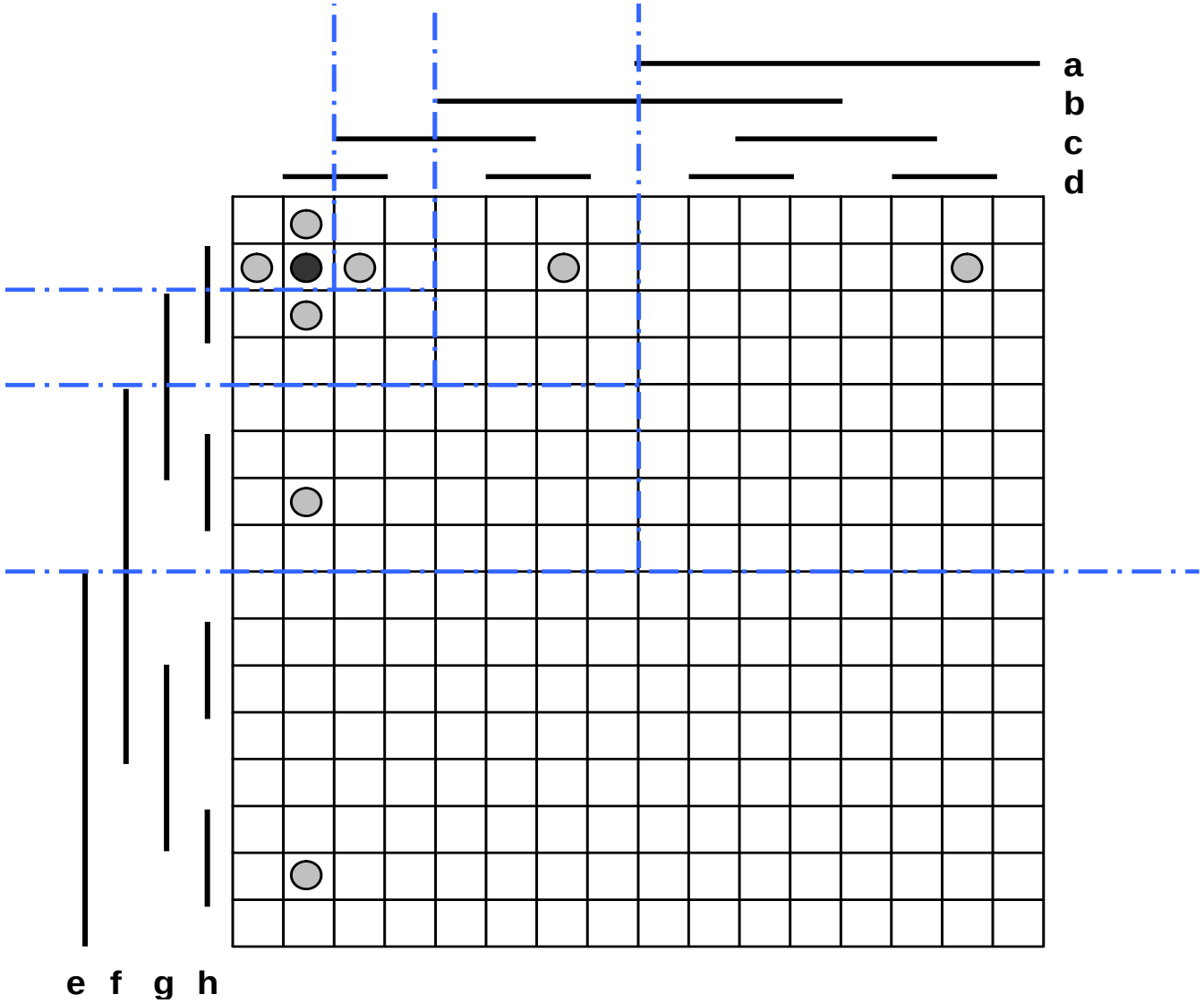
5-input KM 2/2

f(A,B,C,D,E) = Σm(2,5,7,8,10, 13,15,17,19,21,23,24,29 31)





Neighbors for 8 inputs KM

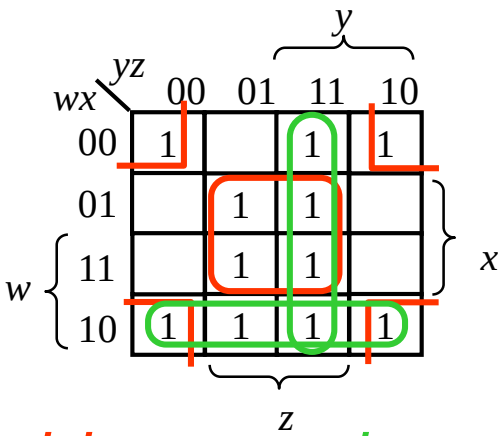


Karnaugh's Maps

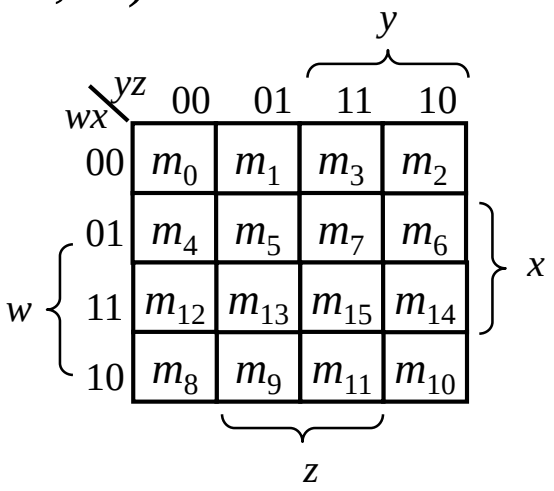
Implicants

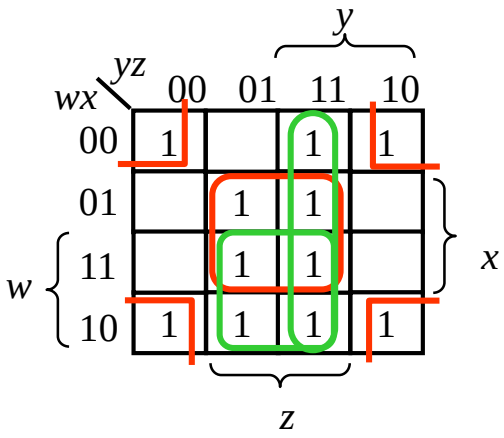
Multiple solutions (coverages) can exist

$$F(w,x,y,z) = \Sigma(0,2,3,5,7,8,9,10,11,13,15)$$

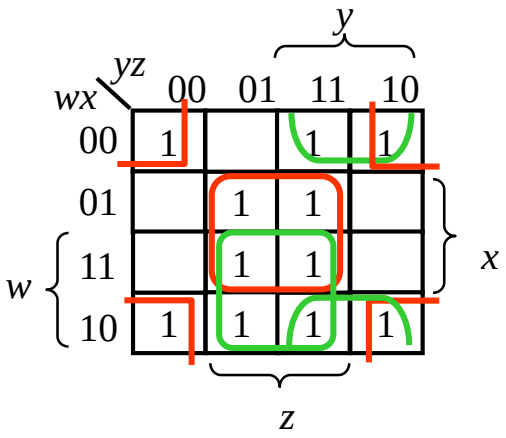


$x'z' + xz + wx' + yz$

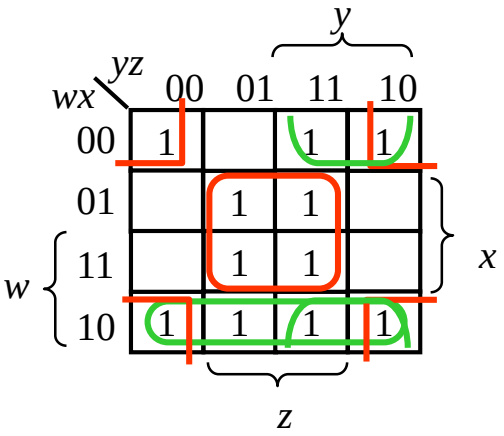




$x'z' + xz + wz + yz$



$x'z' + xz + wz + x'y$

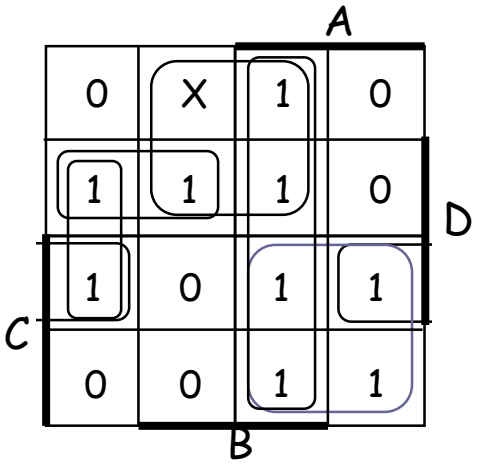


$x'z' + xz + wx' + yz$

Two-level K-map simplification

- Implicant
  - it covers "1"s
- Prime implicant
  - larger as possible
- Essential prime implicant
  - it alone covers a "1"
- Method:
  - Grow implicants into prime implicants
  - Cover all "1" (minimize number of product terms)

Example of Implicants



6 prime implicants:

A'B'D, BC', AC, A'C'D, AB, B'CD

essential

minimum cover: AC + BC' + A'B'D

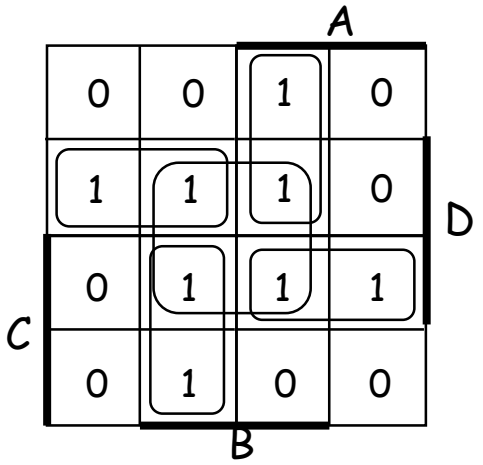
---

5 prime implicants:

BD, ABC', ACD, A'BC, A'C'D

essential

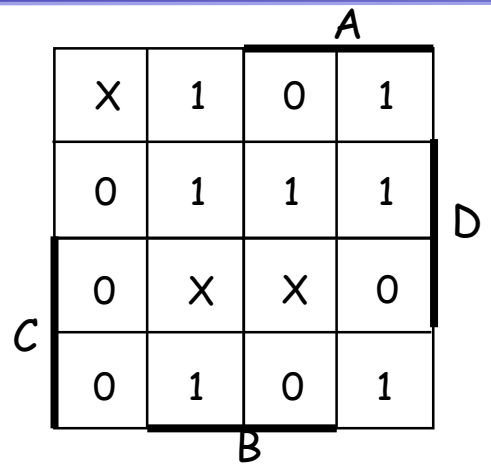
minimum cover: 4 essential implicants

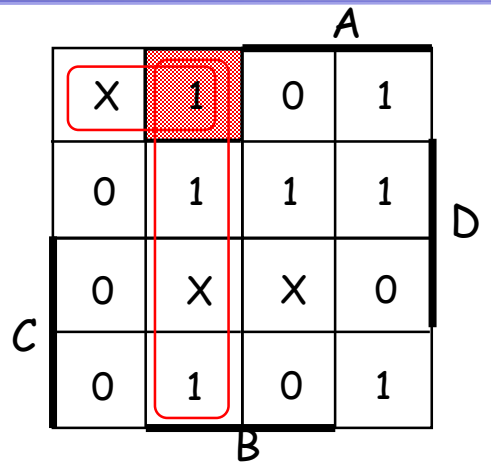


Algorithm for two-level simplification

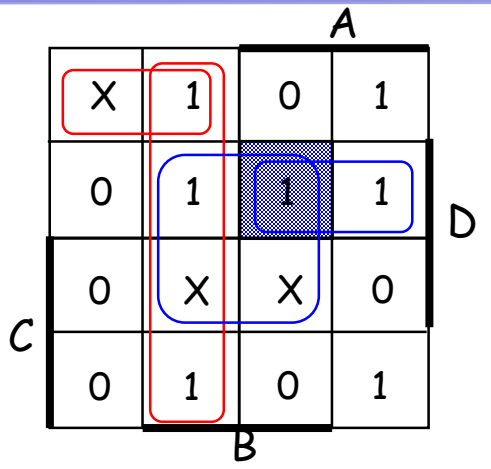
- K-map algorithm
  - Step 1: choose a "1"
  - Step 2: maximize covering implicants to prime implicants *(number of elements must be a power of 2)*
  - Repeat Steps 1 and 2 until finding all prime implicants
  - Step 3:
    - select essential implicants
  - Step 4:
    - cover remaining "1" by the smallest number of prime implicants

f(A,B,C,D) = m(4,5,6,8,9,10,13) + d(0,7,15)

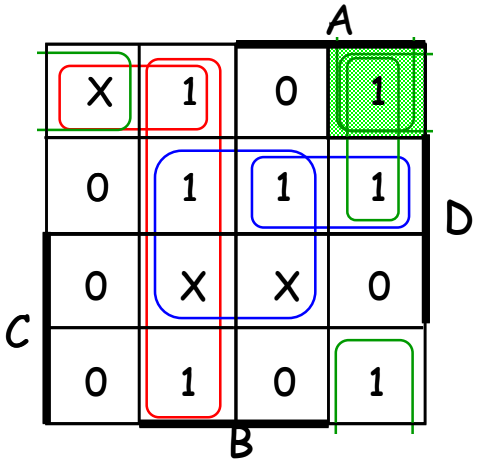




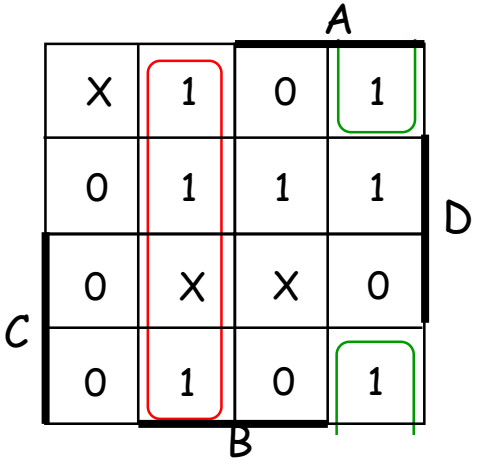
2 primes around A'BC'D'



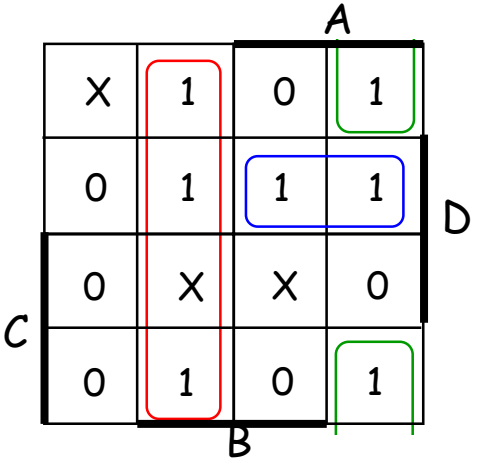
2 primes around ABC'D



3 primes around AB'C'D'



2 essential primes



minimum cover (3 primes)

[Seungryoul M.: Combination Logic, KAIST 2002]

Logic Functions

and their evaluation by multiplexers

What does SAT mean?

Definition: the SAT problem (*Boolean satisfiability problem*):
For a given Boolean function E, written using the basic Boolean operators AND, OR, negation, and parentheses, find the values of the variables such that
the function output has a pre-specified value ('0' or '1').

- E = (a + b) . (a + b') . (a' + c) . (c' + b)

  - From the known a = b = c = TRUE - the calculation of E is easy.

- E = (a + b) . (a + b') . (a' + c) . (c' + b')

  - Finding a, b, c for E=0 or E=1? - the calculation can be challenging. It is an SAT problem.

There are 2^n possible values of n variables and the testing all of them has complexity $O(2^n)$.

Why is SAT important?

- It's NP-complete
  (NP –nondeterministic polynomial time)
  - Complete NP means that all NP task can be transformed into NP-complete in polynomial time
    = if we can solve SAT quickly, we can solve anything in NP quickly (Cook's theorem, 1971)
- Many and varied applications in itself:
  - theorem proving
  - hardware design
  - machine vision
  - ...In fact, any problem where constraints exist that have to be satisfied!

We begin Shanonn's expansion by the example

Shannon Expansion Identity

Y(A,B,C,D,E)

= (A and D) or (A and not B and E) or (not A and E) or (not C and not E)

Y_0 =Y(A,B,C,D,'0') - **Shannon's negative *cofactor of E***

= (A and D) or (A and not B and '0') or (not A and '0') or (not C and not '0')

= (A and D) or (not C and '1')

= (A and D) or not C

Y_1 =Y(A,B,C,D,'1') - **Shannon's positive *cofactor of E***

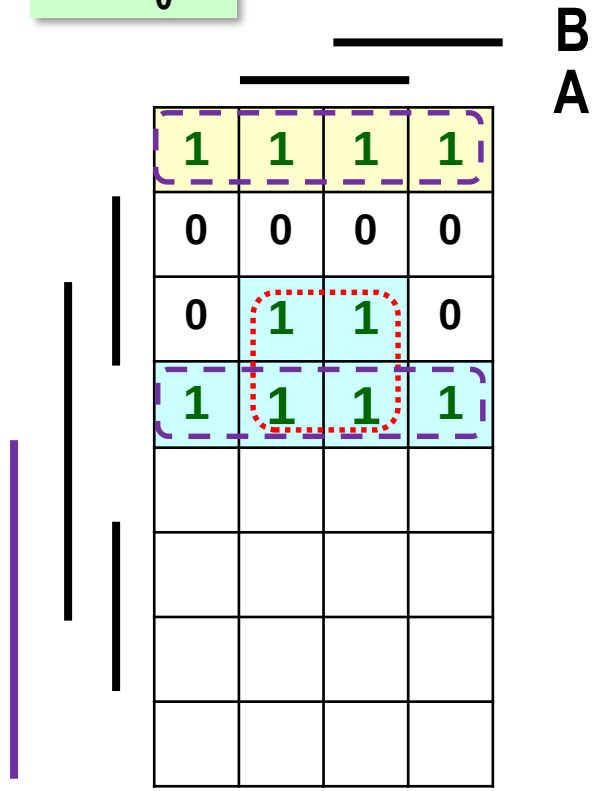
= (A and D) or (A and not B and '1') or (not A and '1') or (not C and not '1')

= (A and D) or (A and not B) or not A or (not C and '0')

= (A and D) or (A and not B) or not A

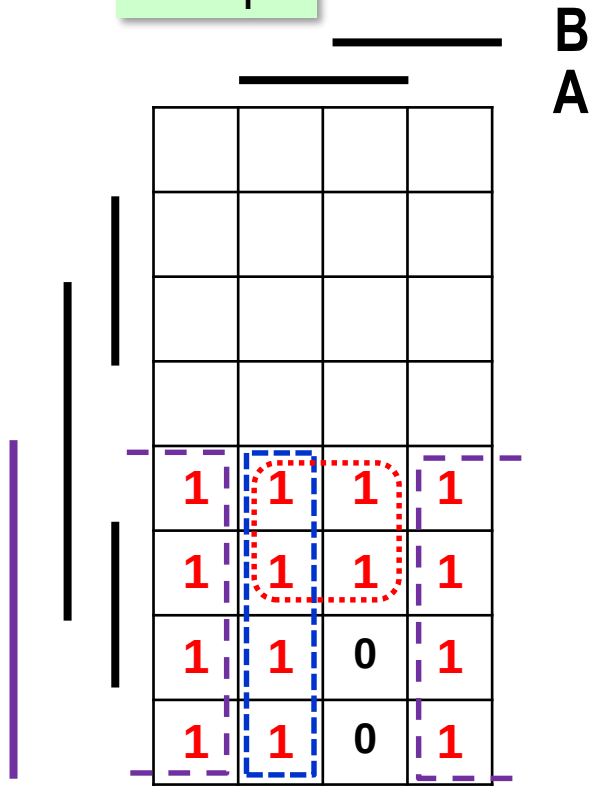
Example 2 of 2

$Y5_0$



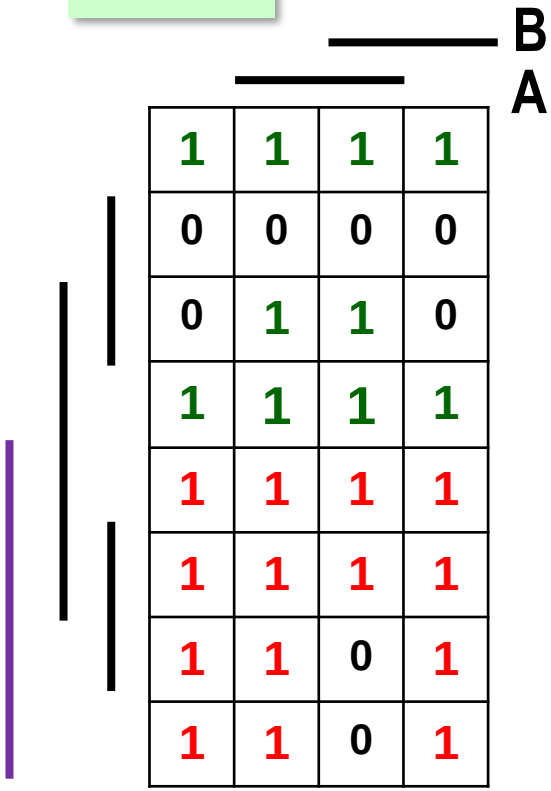
$Y5_0$ = (A and D)
or not C

$Y5_1$



$Y5_1$ = (A and D) or (A and not B)
or not A

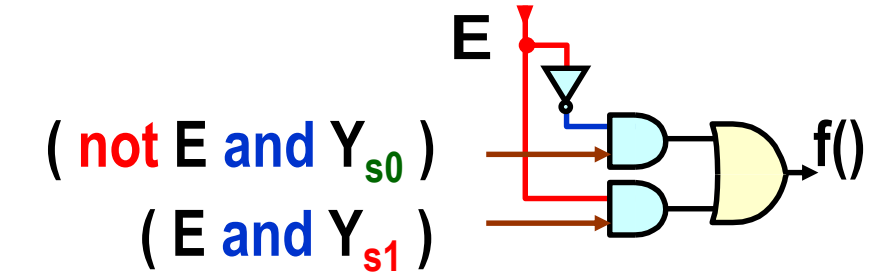
Y5

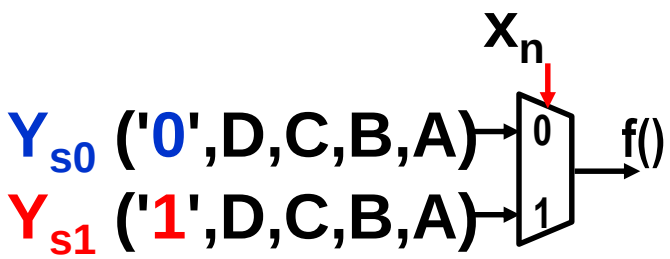


Shanonn's expansion

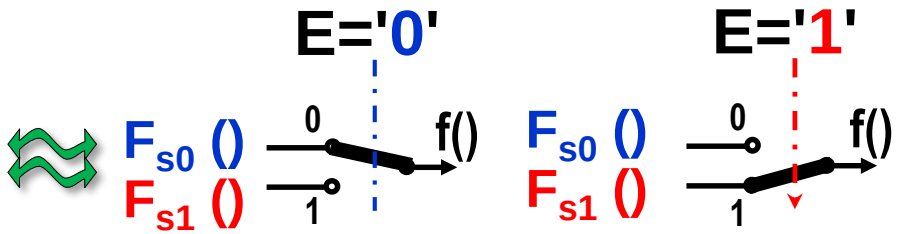
$Y = (\text{not } E \text{ and } Y_{s0})$

$\text{or } (E \text{ and } Y_{s1})$





Multiplexer 2:1



A hint of its function switches

Definition: Shannon Expansion Identity

Shannon expansion: $f : B^n \rightarrow B$, $B=\{0,1\}$

$$f = x_i \cdot f_{x_i} + x'_i f_{x'_i}$$

We say that the function f is **expanded** with respect to x_i , where

- x_i is called a splitting variable.
- f_{xi} and $f_{xi'}$ are called "**Shannon *cofactors*"**
  - positive f_{xi} **and negative** $f_{xi'}$

Let $f : B^n \rightarrow B$ be a Boolean function and $x= (x_1 , x_2 , ..., x_n)$ its variables, then its "cofactors" f_{x_i} and $f_{x'_i}$ are given by substituting after $x_i =´0´$ and $x_i =´1´$, i.e.

$$f_{x_i} (x_1 , x_2 , ..., x_n) = f (x_1 , ..., x_{i-1} , 1, x_{i+1} ,..., x)_n$$

$$f_{x'_i} (x_1 , x_2 , ..., x_n) = f (x_1 , ..., x_{i-1} , 0, x_{i+1} ,..., x)_n$$

$$f (x_1 , x_2 , ..., x_n) = x_i f (x_1 , ..., x_{i-1} , 1, x_{i+1} ,..., x_n) + x'_i f (x_1 , ..., x_{i-1} , 0, x_{i+1} ,..., x)_n$$

*cofactor = The word "**cofactor**" is also used for an algebraic complement in the calculation of determinants or as a designation for low molecular weight organic substances in enzymes.*

Minimization of F5

F5(E, D, C, B, A)

E D C \ B A	00	01	11	10
000	X	0	0	1
001	X	1	1	1
011	0	1	1	0
010	0	0	0	0
110	0	0	0	0
111	0	1	1	0
101	X	1	1	1
100	X	X	X	X

$F5_{S0}$('0', D, C, B, A)

D C \ B A	00	01	11	10
00	X	0	0	1
01	X	1	1	1
11	0	1	1	0
10	0	0	0	0

$F5_{S0}$=

(not A and not D)

or (A and C)

$F5_{S1}$('1', D, C, B, A)

D C \ B A	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	X	1	1	1
10	X	X	X	X

$F5_{S1}$=

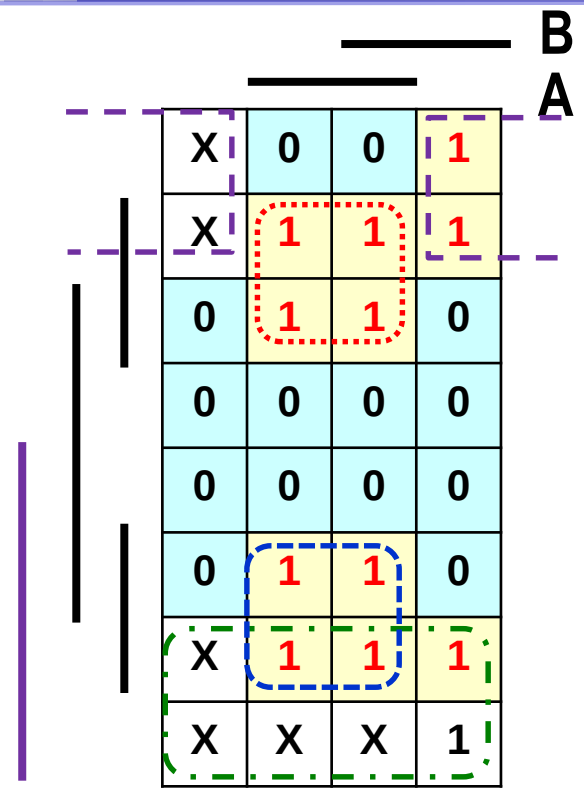
not D

or (A and C)

$F5_S$ = (not E and $F5_{S0}$) or (E and $F5_{S1}$)

$F5_S$ = (not E and ((not A and not D) or (A and C))) or (E and (not D or (A and C)))

Direct minimization



F_S

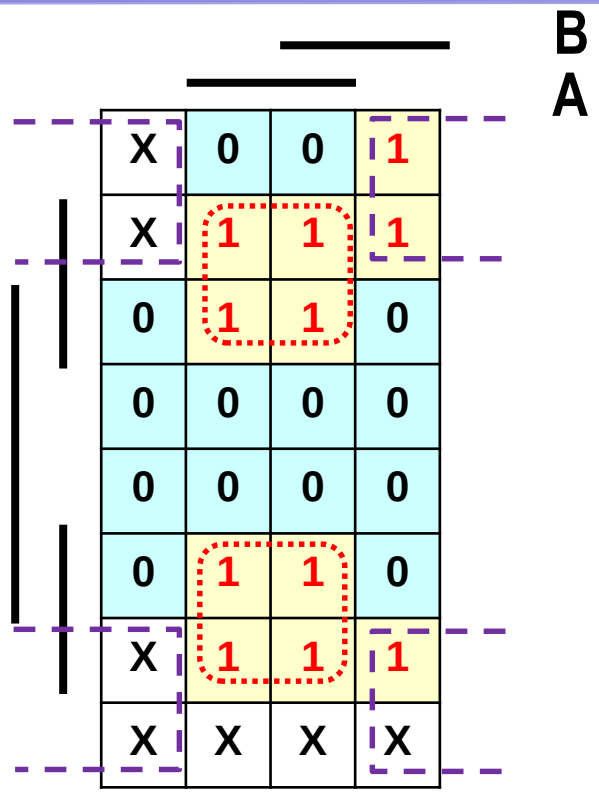
F_s= (not E and F_{S0}) or (E and F_{S1})

F_{S0}= (not A and not D) or (A and C)

F_{S1}= not D or (A and C)

Minimizing as a whole

F_m



F_m= (not A and not D)
or (A and C)

F_{S0} = not D F_{S1}= C

Example 21 of 2 - the splitting variable is E

Create KM of Y5 function:

Y5(A,B,C,D, E) = (A and D) or (A and not B and E)

or (not A and E) or (not C and not E)

$Y5_0$=Y5(A,B,C,D,'0')

= (A and D) or (A and not B and '0') or (not A and '0') or (not C and not '0')

= (A and D) or (not C and '1')

= (A and D) or not C

$Y5_1$=Y5(A,B,C,D,'1')

= (A and D) or (A and not B and '1') or (not A and '1') or (not C and not '1')

= (A and D) or (A and not B) or not A or (not C and '0')

= (A and D) or (A and not B) or not A

Example 2 of 2 - the splitting variable is D

Create KM of Y5 function:

Y5(A,B,C, D ,E) = (A and D) or (A and not B and E)
or (not A and E) or (not C and not E)

$Y5_0$=Y5(A,B,C,'0',E)
= (A and '0') or (A and not B and E) or (not A and E) or (not C and not E)
= (A and not B and E) or (not A and E) or (not C and not E)

$Y5_1$=Y5(A,B,C,'1', E)
= (A and '1') or (A and not B and E)
or (not A and E) or (not C and not E)
= A or (A and not B and E) or (not A and E) or (not C and not E)

The cofactors have complexity of the original function"

Drawbacks of Shannon Expansion

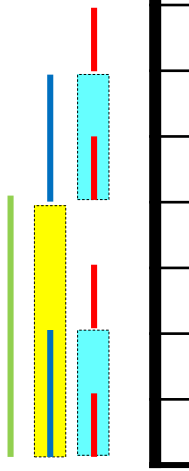
- By using the expansion, we get the result in a simpler way, but….
- …we are minimizing only sub-units, not whole function, so the result often have not minimal complexity - we sacrificed it to get a faster solution.
- …it doesn't solve everything. The complexity of cofactors result highly depends on the choice of the variable to start with, which is itself an NP-hard task.
- It does not give usable results for some functions like adders and multipliers.

Logic Functions

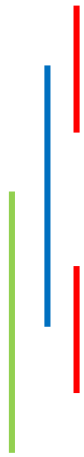
by decompositions

Example: Binary-reflected Gray Code

N
0
1
2
3
4
5
6
7

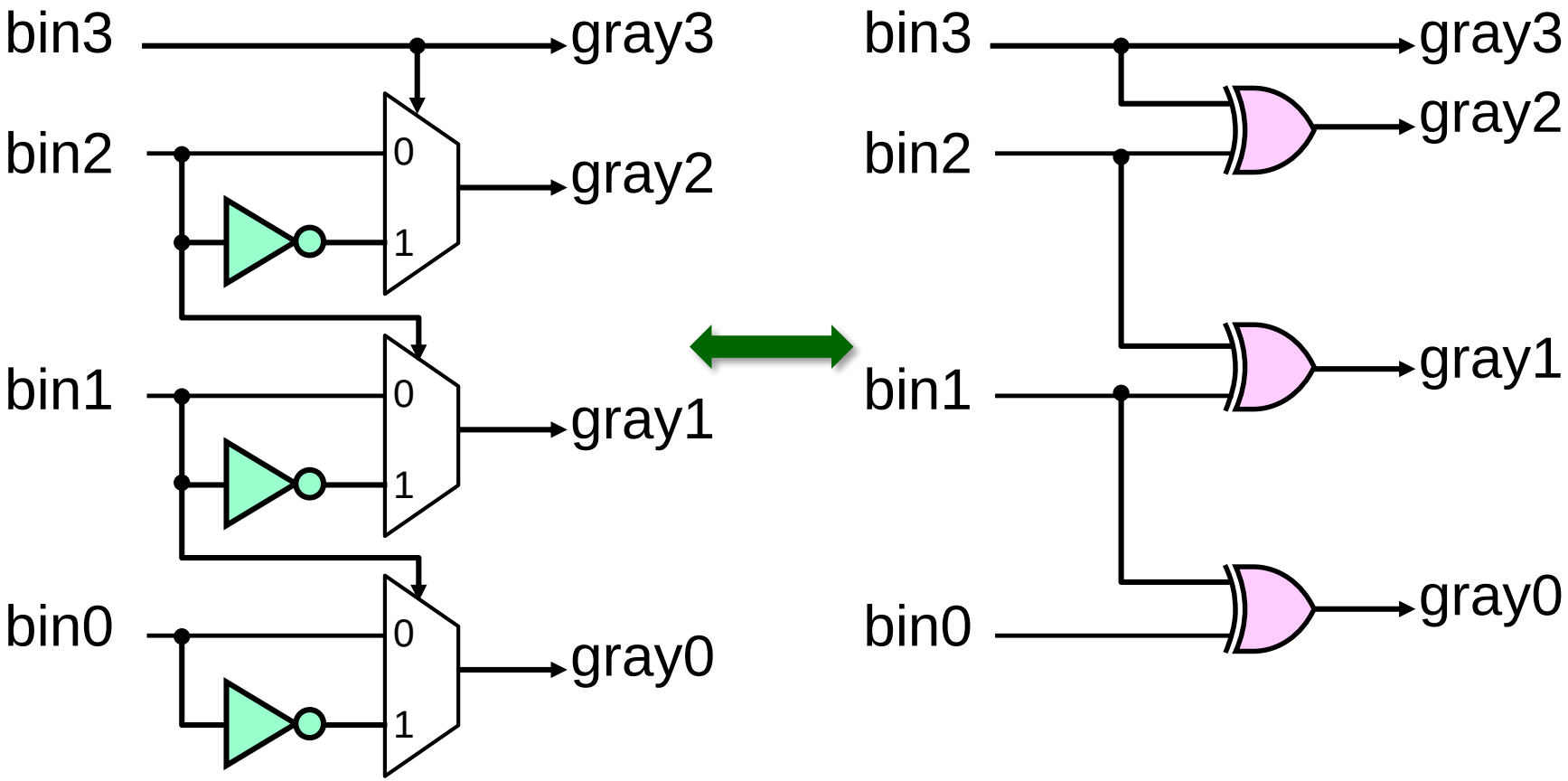


bin2	*bin1*	*bin0*
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1



gray2	*gray1*	*gray0*
0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

Connecting with XOR

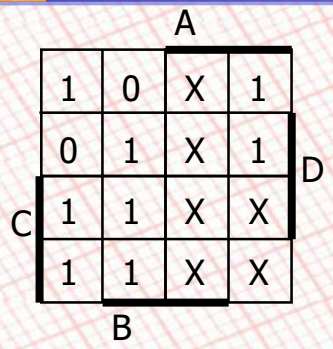


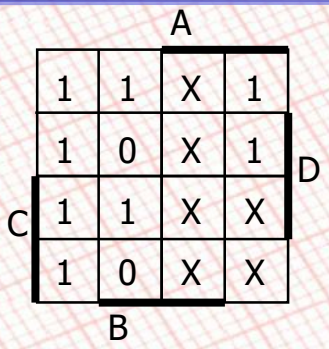

```
uint BinToGray(uint bin) { return bin ^ (bin >> 1); . } // >> shift right; ^ XOR
```

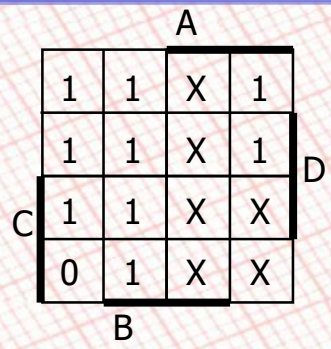
Group Minimizations

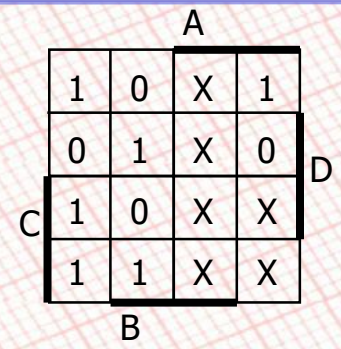
However, sometimes it is convenient to combine more complex functions by group minimization. The whole result comes out simpler than the sum of complexities when minimizing each function separately!

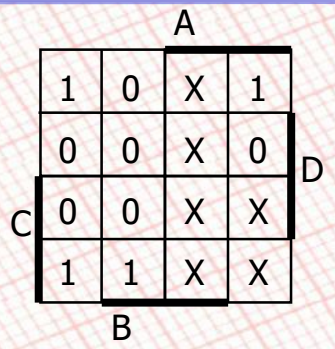
Example BCD / 7-segment

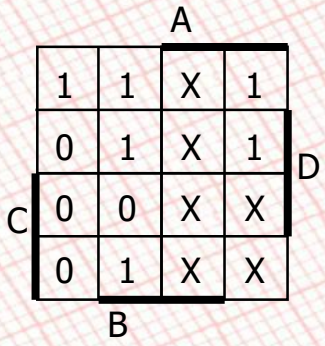


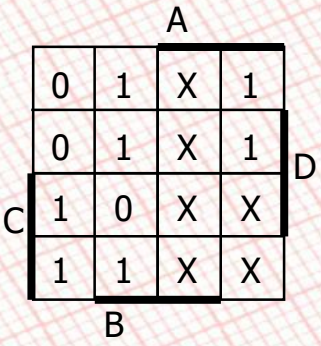






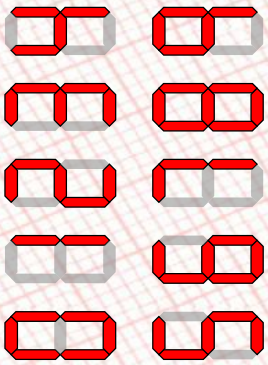


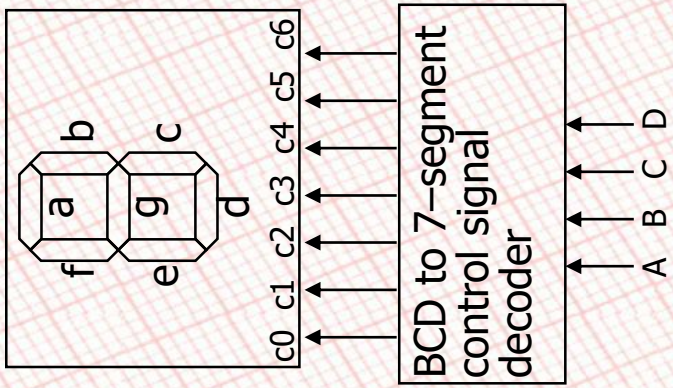




La = A + B D + C + B' D'
Lb = C' D' + C D + B'
Lc = B + C' + D
Ld = B' D' + C D' + B C' D + B' C
Le = B' D' + C D'
Lf = A + C' D' + B D' + B C'
Lg = A + C D' + B C' + B' C

A	B	C	D	La	Lb	Lc	Ld	Le	Lf	Lg
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	–	x	x	x	x	x	x	x
1	1	–	–	x	x	x	x	x	x	x





Group Minimization

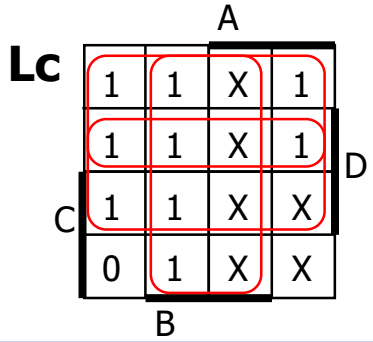
La = A + B D + C + B' D'
Lb = B' + C' D' + C D
Lc = B + C' + D
Ld = B' D' + C D' + B C' D + B' C
Le = B' D' + C D'
Lf = A + C' D' + B D' + B C'
Lg = A + C D' + B C' + B' C

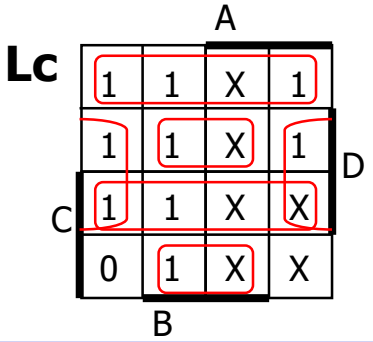
Total: 11 minterms
6 private + 5 shared

La = A +B C' D + C D + B' D' + B C D'
Lb = B' D + C' D' + C D + B' D'
Lc = B' D + B C' D + C' D' + C D + B C D'
Ld = B' D' + B C D'+B C' D + B' D
Le = B' D' + B C D'
Lf = A + C' D' + B C D' + B C' D
Lg = A + B C D' + B' C + B C'

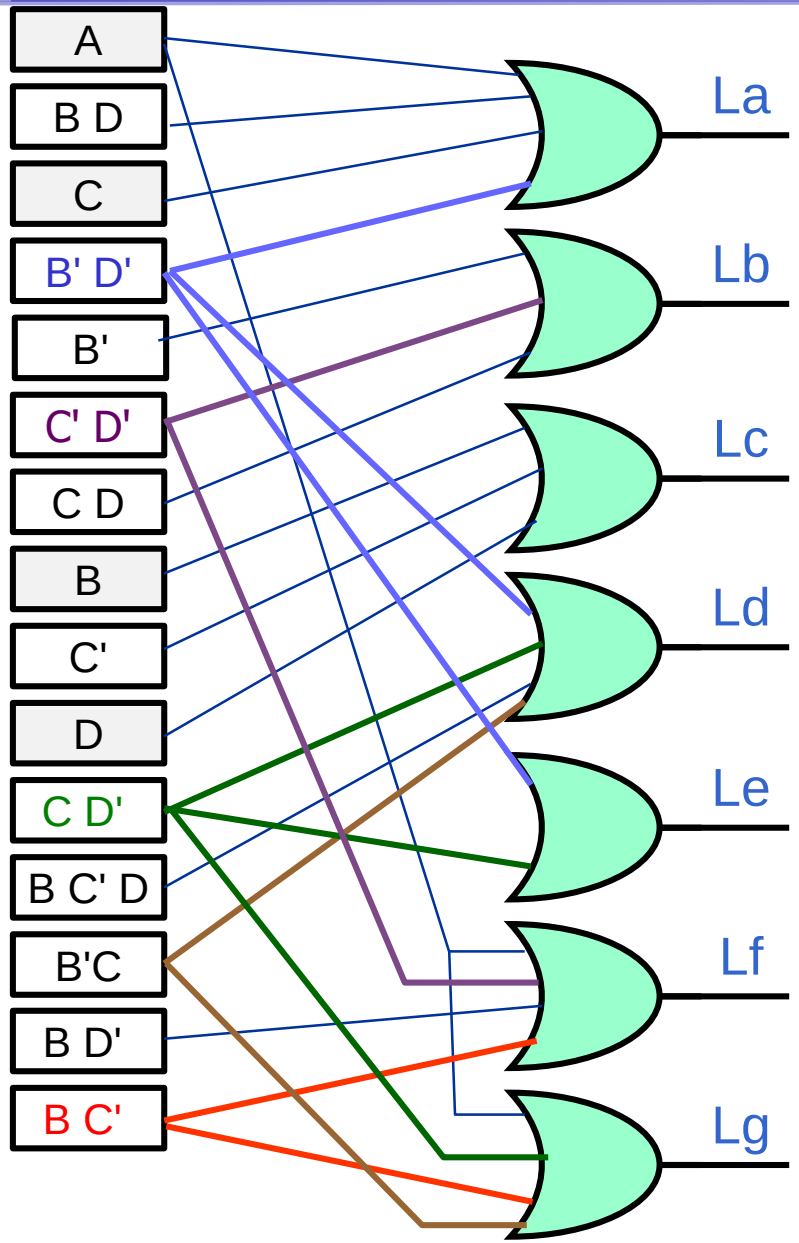
Total: 8 minterms
2 minterms private (in Lg)
+ 6 shared

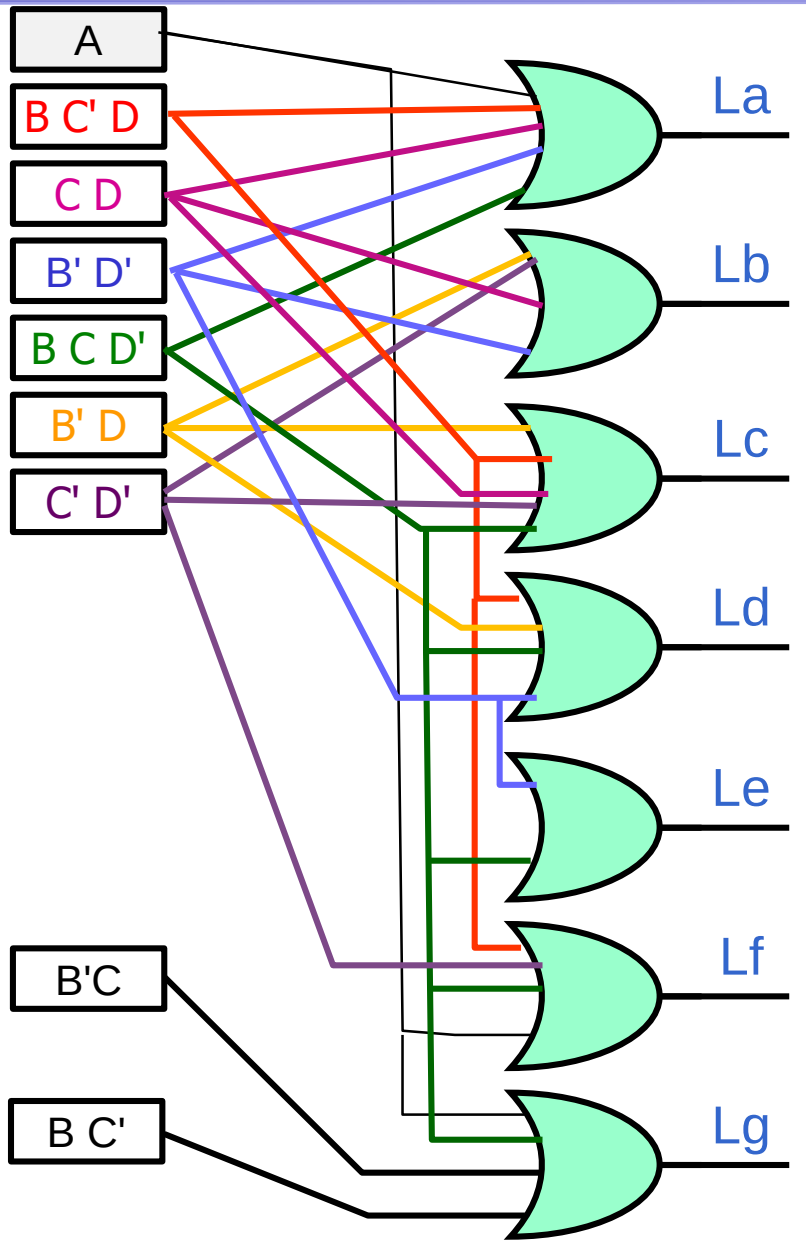
Example: The coverage of Lc is more fragmented, but its minterms are used elsewhere, so the circuit will be smaller.





Group Minimization

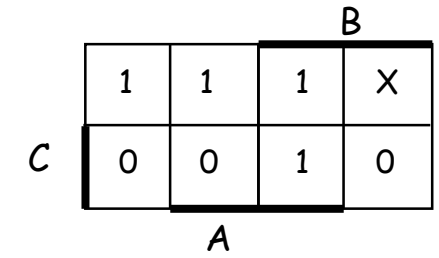


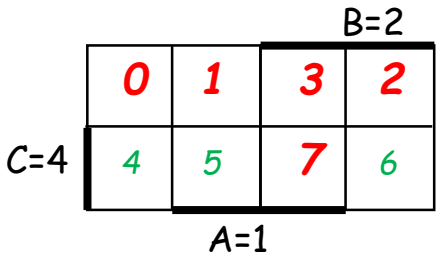


Quine-McCluskey

simple computer minimization algorithm suitable for small functions

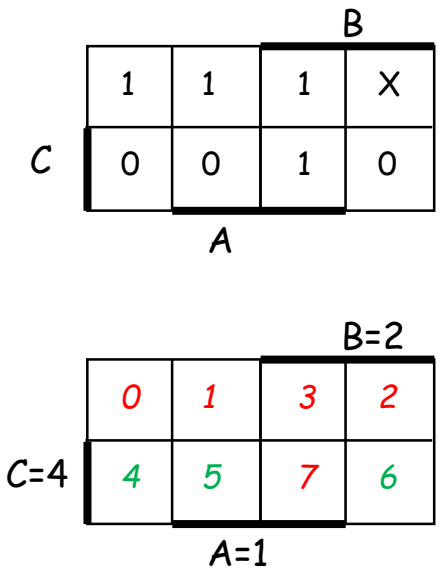
Write binary on-set and don't care set
and sort the numbers by the count of bits 1





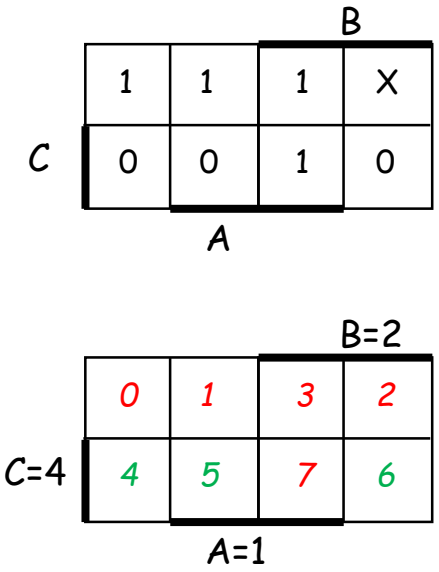
Step	CBA
0	000
1	001
2	010
3	011
7	111

We only associate each group with the following one.



	Step 1	Used	Step 2
	CBA		*CBA*
0	000	*	00-
			0-0
			0-1
1	001	*	
2	010	*	01-
3	011	*	-11
7	111	*	

Repeat until we can

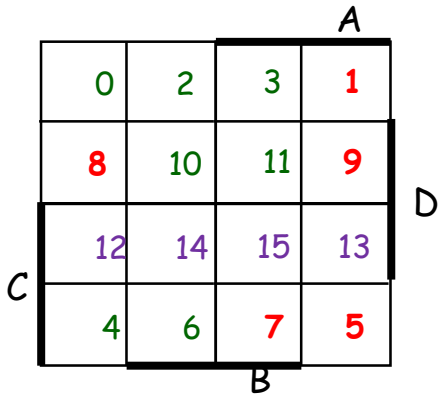


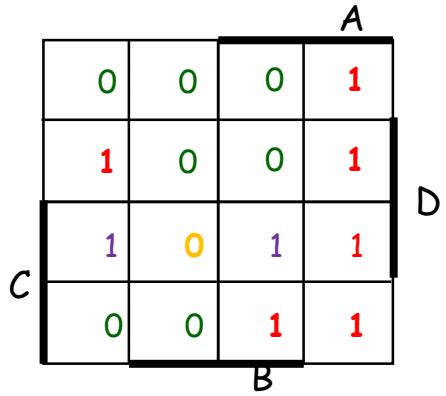
	Step 1	Used	Step 2	Used	Step 3
	CBA		*CBA*		*CBA*
0	000	*	00-	*	0--
			0-0	*	0--
			0-1	*	
1	001	*			
2	010	*	01-	*	
3	011	*	-11		
7	111	*			

We cover the on-set with the found primary implicants

(CBA) 0-- a -11 -> not C or (B and A)

EAEA beacon using McCluskey...






```
DCBA
1:0001
5:0101
7:0111
8:1000
9:1001
12:1100
13:1101
14:1110
15:1111
```

```
*1.5:0-01
*1.9:-001
*5.7:01-1
*5.13:-101
*7.15:-111
*8.9:100-
*8.12:1-00
*9.13:1-01
*12.13:110-
*12.14:11-0
*13.15:11-1
*14.15:111-
```

```
(1.5).(9.13):--01
(1.9).(5.13):--01
(5.7).(13.15):-1-1
(5.13).(7.15):-1-1
(8.9).(12.13):1-0-
(8.12).(9.13):1-0-
(12.13).(14.15):11--
(12.14).(13.15):11--
```

⇩

B'A+C.A+D.B'=
B'.(A+D)+C.A

⇐

```
(1.5).(9.13):--01
(5.7).(13.15):-1-1
(8.9).(12.13):1-0-
```

In "don't care" state 14 is 0, the result is different from PoS (A+D).(B'+C) because McCluskey emulates SoP.

- $O = 3^n \ ln(n)$

N	Complexity	x Increase
2	6	-
4	112	18,00
6	1 306	11,63
8	13 643	10,45
10	135 965	9,97
12	1 320 581	9,71
14	12 622 529	9,56
16	119 350 853	9,46
18	1 119 789 240	9,38
20	10 445 472 561	9,33
22	97 000 187 483	9,29
24	897 576 270 217	9,25
26	8 281 644 255 394	9,23
28	76 230 150 996 763	9,20
30	700 276 379 387 777	9,19
32	6 422 078 597 116 550	9,17

If one step takes 1 ns, then for N=22 => 2 seconds | for N=32 => 74.3 days.