Combinational logic

- **Logic functions, truth tables, and switches**
  - NOT, AND, OR, NAND, NOR, XOR, . . .
  - minimal set
- **Axioms and theorems of Boolean algebra**
  - proofs by re-writing
  - proofs by perfect induction
- **Gate logic**
  - networks of Boolean functions
  - time behavior
- **Canonical forms**
  - two-level
  - incompletely specified functions
- **Simplification**
  - Boolean cubes and Karnaugh maps
  - two-level simplification

Possible logic functions of two variables

- **There are 16 possible functions of 2 input variables:**
  - in general, there are 2**(2**n) functions of n inputs

X	Y	16 possible functions (F0–F15)															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		0	X and Y		X		Y	X ≠ Y	X or Y	X nor Y not (X or Y)	X = Y	not Y		not X		X nand Y not (X and Y)	1

as you can see 0, 1, X, X', Y, Y', X • Y, X + Y are only half of the possible functions

Cost of different logic functions

- **Different functions are easier or harder to implement**
  - each has a cost associated with the number of switches needed

  - 0 (F0) and 1 (F15): require 0 switches, directly connect output to low/high
  - X (F3) and Y (F5): require 0 switches, directly connect output to one of inputs
  - X' (F12) and Y' (F10): require 2 switches for "inverter" or NOT-gate
  - X nor Y (F4) and X nand Y (F14): require 4 switches

  - X or Y (F7) and X and Y (F1): require 6 switches

  - X = Y (F9) and X ≠ Y (F6): require 16 switches

  - thus, because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice

Minimal set of functions

- **Can we implement all logic functions from NOT, NOR, and NAND?**
  - Yes!
  - For example, implementing X <u>and</u> Y
    is the same as implementing <u>not</u> (X <u>nand</u> Y)

- **In fact, we can do it with only NOR or only NAND**
  - NOT is just a NAND or a NOR with both inputs tied together

X	Y	X nor Y
0	0	1
1	1	0

X	Y	X nand Y
0	0	1
1	1	0

  - and NAND and NOR are "duals",
    that is, its easy to implement one using the other

X nand Y ≡ not ((not X) nor (not Y))
X nor Y ≡ not ((not X) nand (not Y))

- **But lets not move too fast . . .**
  - lets look at the mathematical foundation of logic

- **An algebraic structure consits of**
  - a set of elements B
  - binary operations { + , • }
  - and a unary operation { ' }
  - such that the following axioms hold:

1. the set B contains at least two elements, a, b, such that $a \neq b$
2. closure: $a + b$ is in B — $a \bullet b$ is in B
3. commutativity: $a + b = b + a$ — $a \bullet b = b \bullet a$
4. associativity: $a + (b + c) = (a + b) + c$ — $a \bullet (b \bullet c) = (a \bullet b) \bullet c$
5. identity: $a + 0 = a$ — $a \bullet 1 = a$
6. distributivity: $a + (b \bullet c) = (a + b) \bullet (a + c)$ — $a \bullet (b + c) = (a \bullet b) + (a \bullet c)$
7. complementarity: $a + a' = 1$ — $a \bullet a' = 0$

- **Boolean algebra**
  - B = {0, 1}
  - + is logical OR, • is logical AND
  - **'** is logical NOT

- **All algebraic axioms hold**

Logic functions and Boolean algebra

- **Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ', +, and •**

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X'	X' • Y
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

X	Y	X'	Y'	X • Y	X' • Y'	(X • Y) + (X' • Y')
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

$$(X \bullet Y) + (X' \bullet Y') \quad \equiv \quad X = Y$$

Boolean expression that is true when the variables X and Y have the same value and false, otherwise

X, Y are Boolean algebra variables

Axioms and theorems of Boolean algebra

- **Useful axioms and theorems of Boolean algebra:**
  - identity
    1. $X + 0 = X$ 1D. $X \bullet 1 = X$
  - null
    2. $X + 1 = 1$ 2D. $X \bullet 0 = 0$
  - idempotency:
    3. $X + X = X$ 3D. $X \bullet X = X$
  - involution:
    4. $(X')' = X$
  - complementarity:
    5. $X + X' = 1$ 5D. $X \bullet X' = 0$
  - commutativity:
    6. $X + Y = Y + X$ 6D. $X \bullet Y = Y \bullet X$
  - associativity:
    7. $(X + Y) + Z = X + (Y + Z)$ 7D. $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$

Axioms and theorems of Boolean algebra (cont'd)

- **Useful axioms and theorems of Boolean algebra (continued):**
  - distributivity:
    - 8. $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$ 8D. $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$
  - uniting:
    - 9. $X \cdot Y + X \cdot Y' = X$ 9D. $(X + Y) \cdot (X + Y') = X$
  - absorption:
    - 10. $X + X \cdot Y = X$ 10D. $X \cdot (X + Y) = X$
  - a second form of absorption:
    - 11. $(X + Y') \cdot Y = X \cdot Y$ 11D. $(X \cdot Y') + Y = X + Y$
  - de Morgan's:
    - 12. $(X + Y + ...)' = X' \cdot Y' \cdot ...$ 12D. $(X \cdot Y \cdot ...)' = X' + Y' + ...$
  - generalized de Morgan's:
    - 13. $f'(X1,X2,...,Xn,0,1,+,\cdot) = f(X1',X2',...,Xn',1,0,\cdot,+)$

Axioms and theorems of Boolean algebra (cont'd)

- **Useful axioms and theorems of Boolean algebra (continued):**
  - duality:
    - 14. $(X + Y + ...)^D = X \bullet Y \bullet ...$ 15D. $(X \bullet Y \bullet ...)^D = X + Y + ...$
  - generalized duality:
    - 15. $f^D(X1,X2,...,Xn,0,1,+,\bullet) = f(X1,X2,...,Xn,1,0,\bullet,+)$
  - multiplication and factoring:
    - 16. $(X + Y) \bullet (X' + Z) = X \bullet Z + X' \bullet Y$ 16D. $X \bullet Y + X' \bullet Z = (X + Z) \bullet (X' + Y)$
  - concensus:
    - 17. $(X \bullet Y) + (Y \bullet Z) + (X' \bullet Z) = X \bullet Y + X' \bullet Z$ 17D. $(X + Y) \bullet (Y + Z) \bullet (X' + Z) = (X + Y) \bullet (X' + Z)$

- **Duality**
  - a dual of a Boolean expression is derived by replacing
    - • by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
  - <u>any theorem that can be proven is thus also proven for its dual!</u>
  - a meta-theorem (a theorem about theorems)

Proving theorems

- **Using the axioms of Boolean algebra:**
  - e.g., prove the theorem: $X \bullet Y + X \bullet Y' = X$
  - e.g., prove the theorem: $X + X \bullet Y = X$

Proving theorems (cont'd)

- **Using the axioms of Boolean algebra:**
  - e.g., prove the theorem: $X \bullet Y + X \bullet Y' = X$

distributivity (8)	$X \bullet Y + X \bullet Y'$	=	$X \bullet (Y + Y')$
complementarity (5)	$X \bullet (Y + Y')$	=	$X \bullet (1)$
identity (1D)	$X \bullet (1)$	=	X √

  - e.g., prove the theorem: $X + X \bullet Y = X$

identity (1D)	$X + X \bullet Y$	=	$X \bullet 1 + X \bullet Y$
distributivity (8)	$X \bullet 1 + X \bullet Y$	=	$X \bullet (1 + Y)$
identity (2)	$X \bullet (1 + Y)$	=	$X \bullet (1)$
identity (1D)	$X \bullet (1)$	=	X √

Proving theorems (cont'd)

- **Using perfect induction (complete truth table):**
  - e.g., de Morgan's:

$(X + Y)' = X' \bullet Y'$

NOR is equivalent to AND with inputs complemented

X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

$(X \bullet Y)' = X' + Y'$

NAND is equivalent to OR with inputs complemented

X	Y	X'	Y'	(X • Y)'	X' + Y'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Applying the theorems to simplify expressions

- **The theorems of Boolean algebra can simplify Boolean expressions**
  - e.g., full adder's carry-out function

$$Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin$$

Applying the theorems to simplify expressions (cont'd)

- **The theorems of Boolean algebra can simplify Boolean expressions**
  - e.g., full adder's carry-out function (same rules apply to any function)

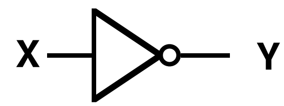
$$
\begin{aligned}
\text{Cout} &= A'\,B\,Cin + A\,B'\,Cin + A\,B\,Cin' + A\,B\,Cin \\
&= A'\,B\,Cin + A\,B'\,Cin + A\,B\,Cin' + \boxed{A\,B\,Cin + A\,B\,Cin} \\
&= A'\,B\,Cin + \underline{A\,B\,Cin} + A\,B'\,Cin + A\,B\,Cin' + \underline{A\,B\,Cin} \\
&= (A' + A)\,B\,Cin + A\,B'\,Cin + A\,B\,Cin' + A\,B\,Cin \\
&= (1)\,B\,Cin + A\,B'\,Cin + A\,B\,Cin' + A\,B\,Cin \\
&= B\,Cin + A\,B'\,Cin + A\,B\,Cin' + \boxed{A\,B\,Cin + A\,B\,Cin} \\
&= B\,Cin + A\,B'\,Cin + \underline{A\,B\,Cin} + A\,B\,Cin' + \underline{A\,B\,Cin} \\
&= B\,Cin + A\,(B' + B)\,Cin + A\,B\,Cin' + A\,B\,Cin \\
&= B\,Cin + A\,(1)\,Cin + A\,B\,Cin' + A\,B\,Cin \\
&= B\,Cin + A\,Cin + A\,B\,(Cin' + Cin) \\
&= B\,Cin + A\,Cin + A\,B\,(1) \\
&= B\,Cin + A\,Cin + A\,B
\end{aligned}
$$

associativity

identity

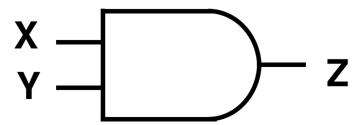
From Boolean expressions to logic gates

- **NOT** X' $\overline{X}$ ~X



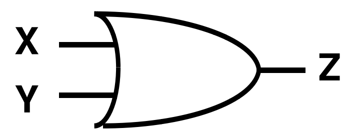
X	Y
0	1
1	0

- **AND** X • Y XY X ∧ Y



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

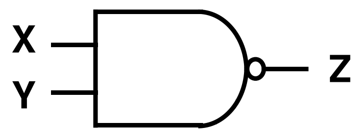
- **OR** X + Y X ∨ Y



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

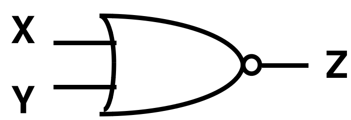
From Boolean expressions to logic gates (cont'd)

- **NAND**



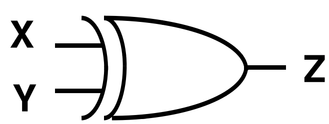
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- **NOR**



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

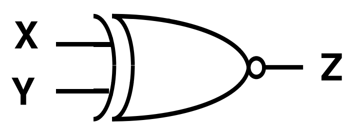
- **XOR**
  $X \oplus Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

X <u>xor</u> Y = X Y' + X' Y
X or Y but not both
("inequality", "difference")

- **XNOR**
  X = Y



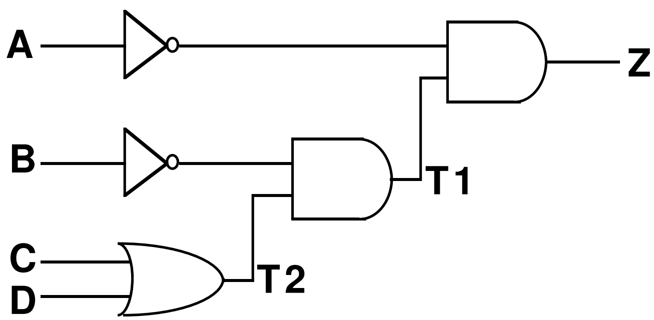
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

X <u>xnor</u> Y = X Y + X' Y'
X and Y are the same
("equality", "coincidence")

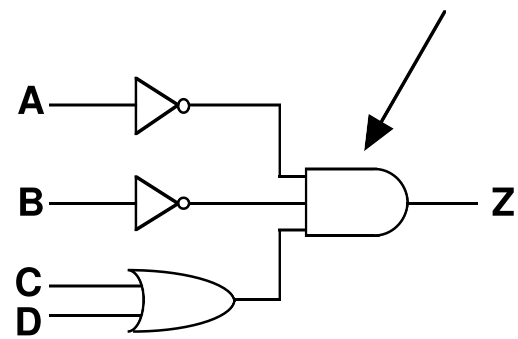
From Boolean expressions to logic gates (cont'd)

- **More than one way to map expressions to gates**

  – e.g., $Z = A' \bullet B' \bullet (C + D) = (A' \bullet (\underbrace{B' \bullet (\overbrace{C + D}^{T2})}_{T1}))$

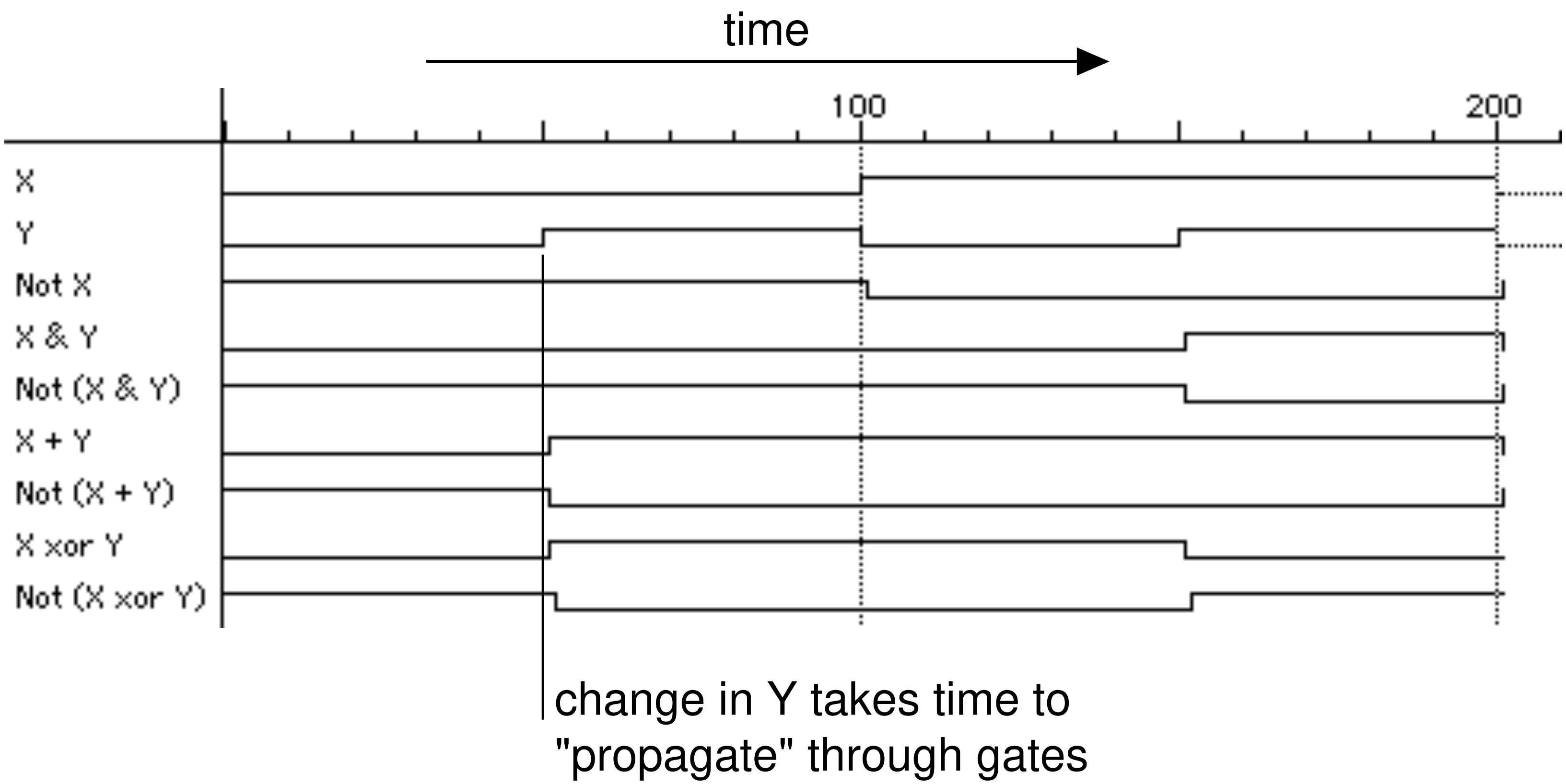


use of 3-input gate



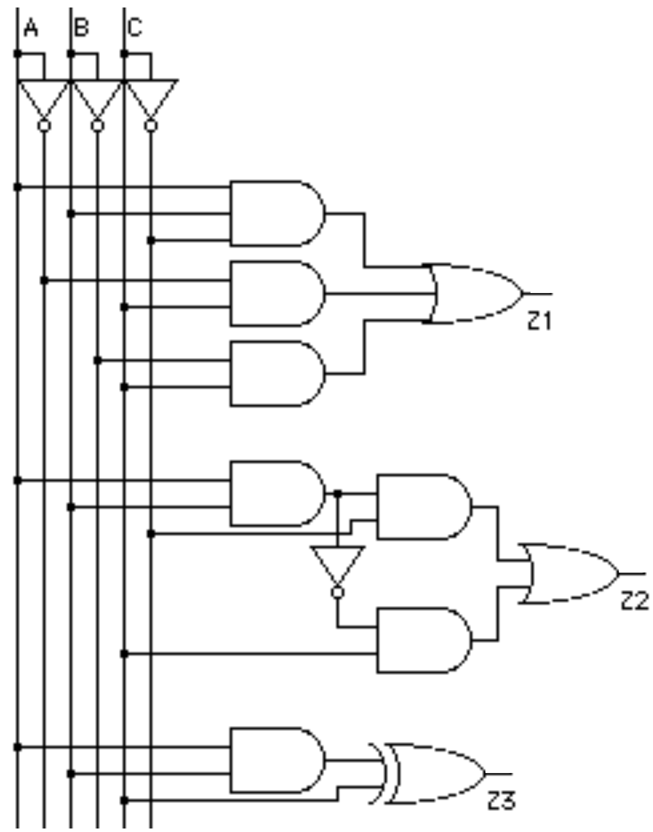
The waveform view of logic functions

- **Just a sideways truth table**
  - but note how edges don't line up exactly
  - it takes time for a gate to switch its output!



How to choose between different realizations of a function?

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



two-level realization
(we don't count NOT gates)

multi-level realization
(gates with fewer inputs)

XOR gate (easier to draw
but costly to build)

Which is the best realization?

- **Reduce number of inputs**
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transitors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are smaller and thus also faster
  - fan-ins (# of gate inputs) are limited in some technologies
- **Reduce number of gates**
  - fewer gates (and the packages they come in) means smaller circuits
    - directly influences manufacturing costs

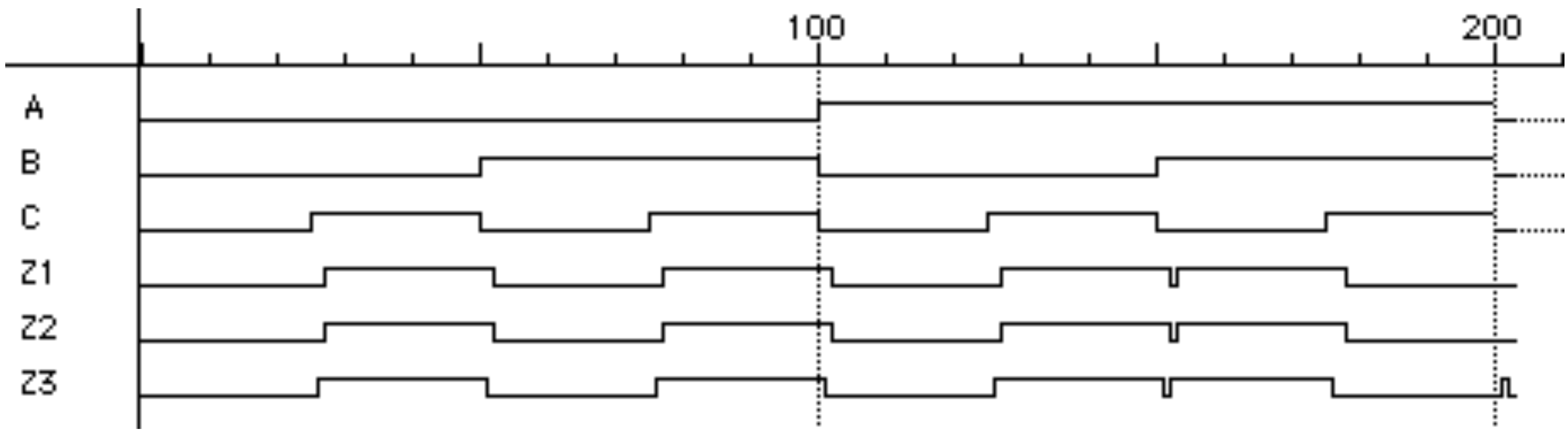
Which is the best realization? (cont'd)

- **Reduce number of levels of gates**
  - fewer level of gates implies reduced signal propagation delays
  - minimum delay configuration typically requires more gates
    - wider, less deep circuits

- **How do we explore tradeoffs between increased circuit delay and size?**
  - automated tools to generate different solutions
  - logic minimization: reduce number of gates and complexity
  - logic optimization: reduction while trading off against delay

Does every realization behave the same way?

- **Under the same input stimuli, the three alternative implementations have almost the same waveform behavior**
  - delays are different
  - glitches (hazards) may arise
  - these variations are due to differences in number of gate levels and structure
- **The three implementations are functionally equivalent**



Implementing Boolean functions

- **Technology independent**
  - canonical forms
  - two-level forms
  - multi-level forms

- **Technology choices**
  - packages of a few gates
  - regular logic
  - two-level programmable logic
  - multi-level programmable logic

- **Truth table is the unique signature of a Boolean function**

- **Many alternative gate realizations may have the same truth table**

- **Canonical forms**
  - standard forms for a Boolean expression
  - provides a unique algebraic signature

Sum-of-products canonical form

- Also known as disjunctive normal form
- Also known as minterm expansion

A	B	C	F	F'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$$F = \overset{001}{A'B'C} + \overset{011}{A'BC} + \overset{101}{AB'C} + \overset{110}{ABC'} + \overset{111}{ABC}$$

$$F' = A'B'C' + A'BC' + AB'C'$$

Sum-of-products canonical form (cont'd)

- **Product term (or minterm)**
  - ANDed product of literals – input combination for which output is true
  - each variable appears exactly once, in true or inverted form (but not both)

A	B	C	minterms	
0	0	0	A'B'C'	m0
0	0	1	A'B'C	m1
0	1	0	A'BC'	m2
0	1	1	A'BC	m3
1	0	0	AB'C'	m4
1	0	1	AB'C	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

short-hand notation for minterms of 3 variables

F in canonical form:

$$
\begin{aligned}
F(A, B, C) &= \Sigma m(1,3,5,6,7) \\
&= m1 + m3 + m5 + m6 + m7 \\
&= A'B'C + A'BC + AB'C + ABC' + ABC
\end{aligned}
$$

canonical form $\neq$ minimal form

$$
\begin{aligned}
F(A, B, C) &= A'B'C + A'BC + AB'C + ABC + ABC' \\
&= (A'B' + A'B + AB' + AB)C + ABC' \\
&= ((A' + A)(B' + B))C + ABC' \\
&= C + ABC' \\
&= ABC' + C \\
&= AB + C
\end{aligned}
$$

Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion

A	B	C	F	F'
0	0	0	0	~~1~~
0	0	1	1	0
0	1	0	0	~~1~~
0	1	1	1	0
1	0	0	0	~~1~~
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$$F = \underbrace{(A + B + C)}_{000}\ \underbrace{(A + B' + C)}_{010}\ \underbrace{(A' + B + C)}_{100}$$

$$F' = (A + B + C')\ (A + B' + C')\ (A' + B + C')\ (A' + B' + C)\ (A' + B' + C')$$

Product-of-sums canonical form (cont'd)

- **Sum term (or maxterm)**
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, in true or inverted form (but not both)

A	B	C	maxterms	
0	0	0	A+B+C	M0
0	0	1	A+B+C'	M1
0	1	0	A+B'+C	M2
0	1	1	A+B'+C'	M3
1	0	0	A'+B+C	M4
1	0	1	A'+B+C'	M5
1	1	0	A'+B'+C	M6
1	1	1	A'+B'+C'	M7

short-hand notation for maxterms of 3 variables

F in canonical form:

$$
\begin{aligned}
F(A, B, C) &= \Pi M(0,2,4) \\
&= M0 \bullet M2 \bullet M4 \\
&= (A + B + C)\ (A + B' + C)\ (A' + B + C)
\end{aligned}
$$

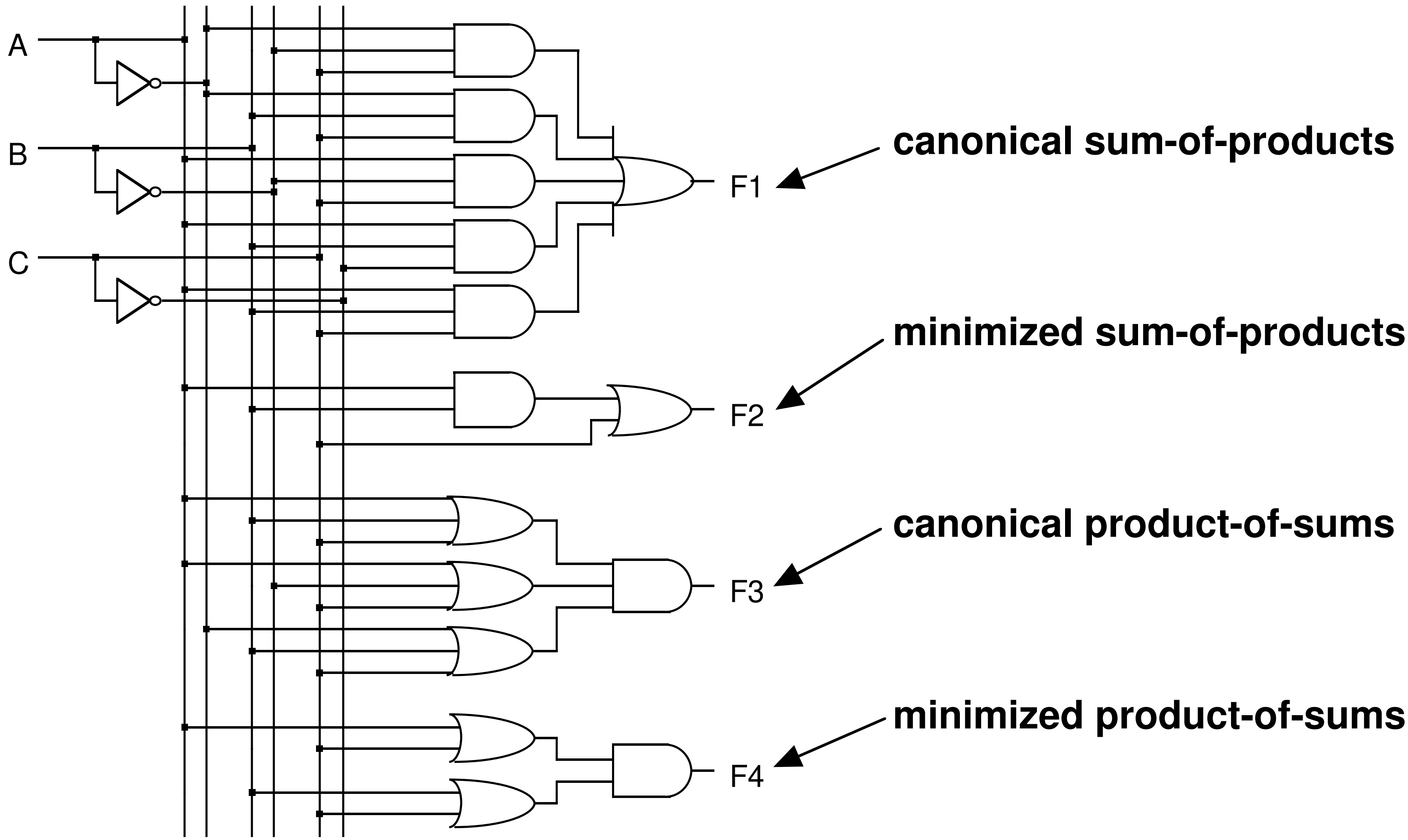
canonical form $\neq$ minimal form

$$
\begin{aligned}
F(A, B, C) &= (A + B + C)\ (A + B' + C)\ (A' + B + C) \\
&= (A + B + C)\ (A + B' + C) \\
&\quad\ (A + B + C)\ (A' + B + C) \\
&= (A + C)\ (B + C)
\end{aligned}
$$

S-O-P, P-O-S, and de Morgan's theorem

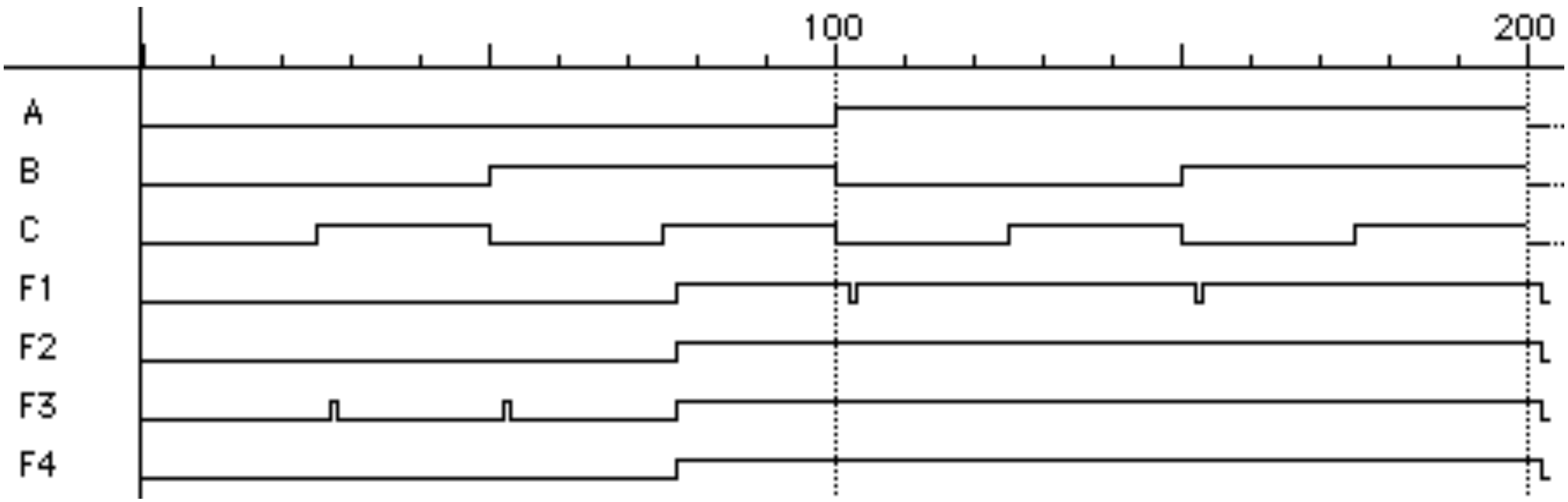
- **Sum-of-products**
  - $F' = A'B'C' + A'BC' + AB'C'$
- **Apply de Morgan's**
  - $(F')' = (A'B'C' + A'BC' + AB'C')'$
  - $F = (A + B + C)\ (A + B' + C)\ (A' + B + C)$
- **Product-of-sums**
  - $F' = (A + B + C')\ (A + B' + C')\ (A' + B + C')\ (A' + B' + C)\ (A' + B' + C')$
- **Apply de Morgan's**
  - $(F')' = (\ (A + B + C')\ (A + B' + C')\ (A' + B + C')\ (A' + B' + C)\ (A' + B' + C')\ )'$
  - $F = A'B'C + A'BC + AB'C + ABC' + ABC$

Four alternative two-level implementations of F = AB + C



Waveforms for four alternatives

- **Waveforms are essentially identical**
  – except for timing hazards (glitches)
  – delays almost identical (modeled as a delay per level, not type of gate or number of inputs to gate)



Mapping between canonical forms

- **Minterm to maxterm conversion**
  - use maxterms whose indices do not appear in minterm expansion
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7) = \Pi M(0,2,4)$
- **Maxterm to minterm conversion**
  - use minterms whose indices do not appear in maxterm expansion
  - e.g., $F(A,B,C) = \Pi M(0,2,4) = \Sigma m(1,3,5,6,7)$
- **Minterm expansion of F to minterm expansion of F'**
  - use minterms whose indices do not appear
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7)$ $F'(A,B,C) = \Sigma m(0,2,4)$
- **Maxterm expansion of F to maxterm expansion of F'**
  - use maxterms whose indices do not appear
  - e.g., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Pi M(1,3,5,6,7)$

- **Functions of n inputs have 2^n possible input configurations**
  - for a given function, not all input combinations may be possible
  - this can be exploited during circuit minimization

- **Example: binary coded decimal increment by 1**

BCD digits encode the decimal digits 0 – 9 in the bit patterns 0000 – 1001

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

off-set of W

on-set of W

don't care (DC) set of W

these inputs patterns should never be encountered in practice – **"don't care"** about associated output values

Notation for incompletely specified functions

- **Don't cares and canonical forms**
  - so far, only represented on-set
  - also represent don't-care-set
  - need two of the three sets (on-set, off-set, dc-set)

- **Canonical representations of the BCD increment by 1 function:**

$$Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15$$

$$Z = \Sigma\ [\ m(0,2,4,6,8) + d(10,11,12,13,14,15)\ ]$$

$$Z = M1 \bullet M3 \bullet M5 \bullet M7 \bullet M9 \bullet D10 \bullet D11 \bullet D12 \bullet D13 \bullet D14 \bullet D15$$

$$Z = \Pi\ [\ M(1,3,5,7,9) \bullet D(10,11,12,13,14,15)\ ]$$

Simplification of two-level combinational logic

- **Finding a minimal sum of products or product of sums realization**
  - exploit don't care information in the process
- **Algebraic simplification**
  - not an algorithmic/systematic procedure
  - how do you know when the minimum realization has been found?
- **Computer-aided design tools**
  - precise solutions require very long computation times, especially for functions with many inputs (> 10)
  - heuristic methods employed – "educated guesses" to reduce amount of computation and yield good if not best solutions
- **Hand methods still relevant**
  - to understand automatic tools and their strengths and weaknesses
  - ability to check results (on small examples)

The uniting theorem

- **Key tool to simplification: A (B' + B) = A**
- **Essence of simplification of two-level logic**
  - find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

F = A'B'+AB' = (A'+A)B' = B'

A	B	F
$\boxed{0}$	$\boxed{0}$	1
0	1	0
$\boxed{1}$	$\boxed{0}$	1
1	1	0

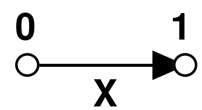
B has the same value in both on-set rows
– B remains

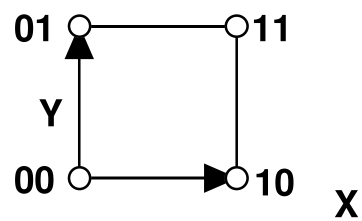
A has a different value in the two rows
– A is eliminated

Boolean cubes

- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"

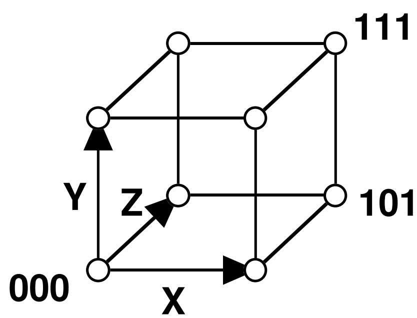
1-cube

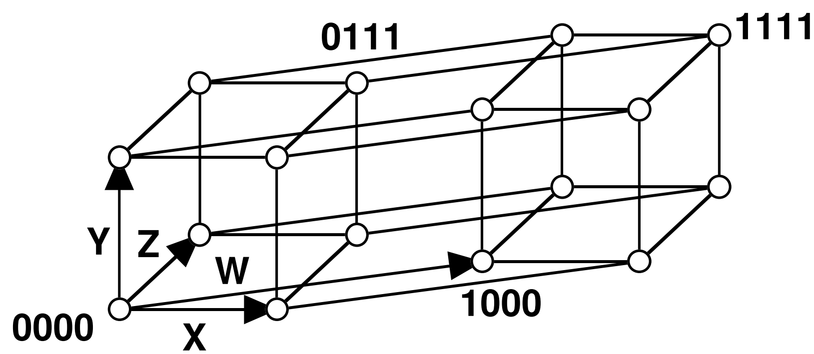




2-cube

3-cube





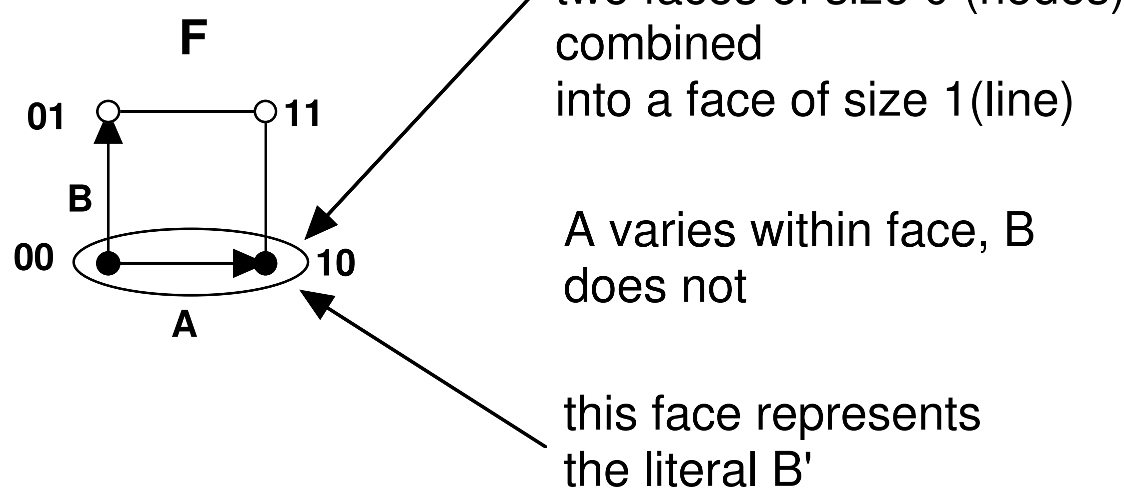
4-cube

Mapping truth tables onto Boolean cubes

- **Uniting theorem combines two "faces" of a cube into a larger "face"**

- **Example:**

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

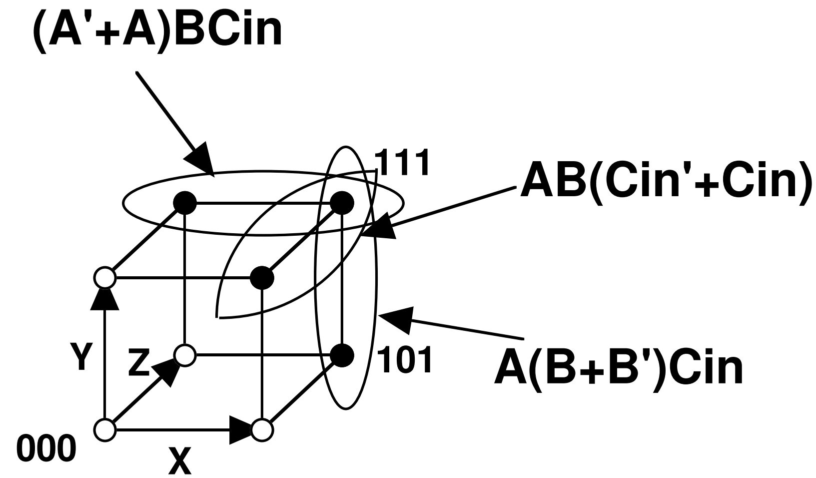


ON-set = solid nodes
OFF-set = empty nodes
DC-set = X'd nodes

Three-variable example

- **Binary full-adder carry-out logic**

A	B	Cin	Cout
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

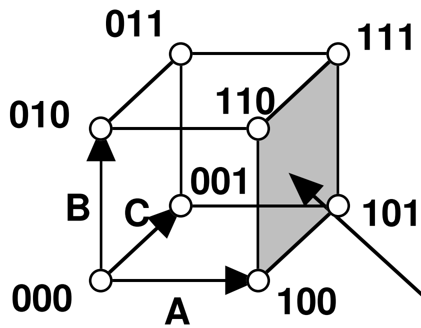


The on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality

Cout = BCin+AB+ACin

Higher dimensional cubes

- **Sub-cubes of higher dimension than 2**



$F(A,B,C) = \sum m(4,5,6,7)$

on-set forms a square
i.e., a cube of dimension 2

represents an expression in one variable
i.e., 3 dimensions – 2 dimensions

A is asserted (true) and unchanged
B and C vary

This subcube represents the literal A

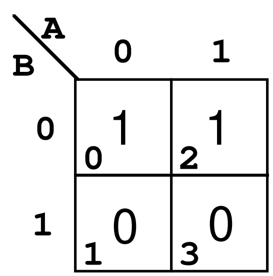
m dimensional cubes in an n dimensional Boolean space

- **In a 3-cube (three variables):**
  – a 0-cube, i.e., a single node, yields a term in 3 literals
  – a 1-cube, i.e., a line of two nodes, yields a term in 2 literals
  – a 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
  – a 3-cube, i.e., a cube of eight nodes, yields a constant term "1"

- **In general,**
  – an m-subcube within an n-cube ($m < n$) yields a term with $n – m$ literals

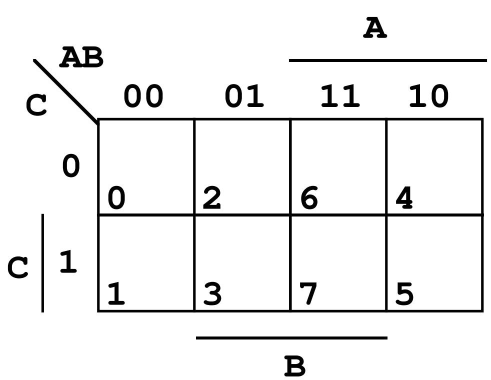
- **Flat map of Boolean cube**
  - wrap–around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions

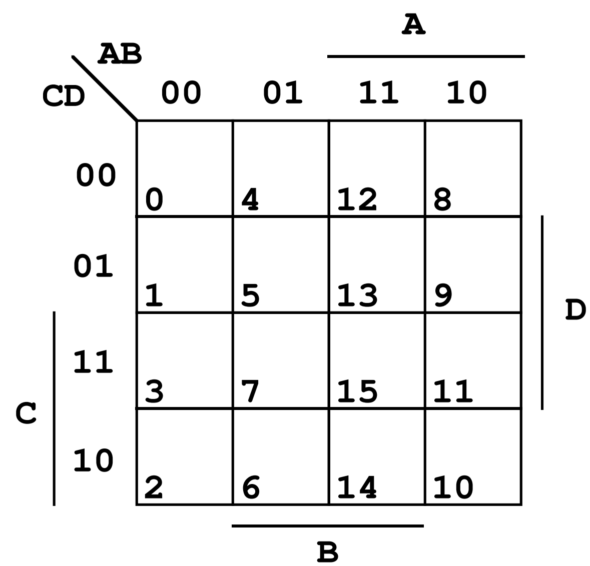
- **Alternative to truth-tables to help visualize adjacencies**
  - guide to applying the uniting theorem
  - on-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table



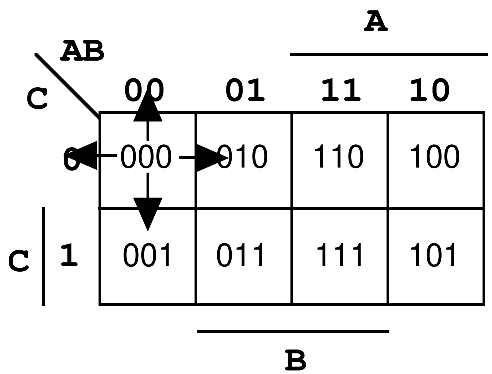
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

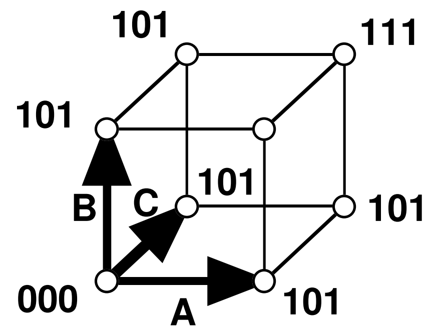
- **Computer-based methods are necessary for practical applications**
- **Numbering scheme based on Gray–code**
  - e.g., 00, 01, 11, 10
  - only a single bit changes in code for adjacent map cells





- Wrap from first to last column
- Wrap top row to bottom row





Karnaugh map examples

- F =

B \ A	0	1
0	1	1
1	0	0

- Cout =

Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

(A over columns 11, 10; B under columns 01, 11)

- f(A,B,C) = $\Sigma m(0,4,6,7)$

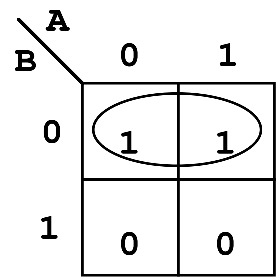
C \ AB	00	01	11	10
0	1	0	0	1
1	0	0	1	1

(A over columns 11, 10; B under columns 01, 11)

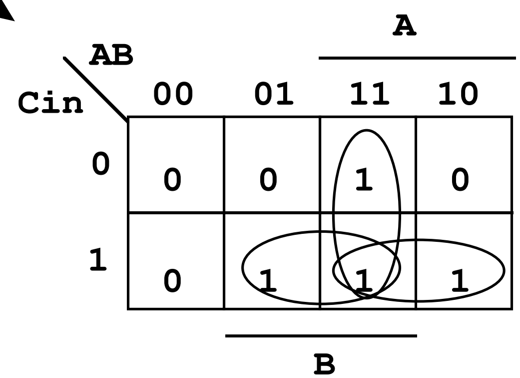
obtain the complement of the function by covering 0s with subcubes

Karnaugh map examples (cont'd)

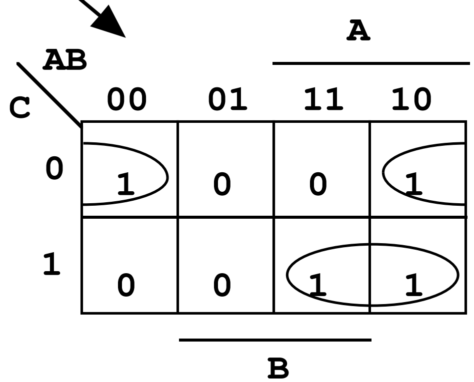
- F = B'



- Cout = AB + BCin + ACin



- f(A,B,C) = Σm(0,4,6,7)
  = B'C'+AC



More Karnaugh map examples

C \ AB	00	01	11	10
0	0	0	1	1
1	0	0	1	1

A: columns 11, 10; B: columns 01, 11

F(A,B,C) =

C \ AB	00	01	11	10
0	1	0	0	1
1	0	0	1	1

A: columns 11, 10; B: columns 01, 11

$F(A,B,C) = \sum m(0,4,5,7)$

F =

C \ AB	00	01	11	10
0	0	1	1	0
1	1	1	0	0

A: columns 11, 10; B: columns 01, 11

F' simply replace 1's with 0's and vice versa

$F'(A,B,C) = \sum m(1,2,3,6)$

F' =

Karnaugh map: 4-variable example

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

F =

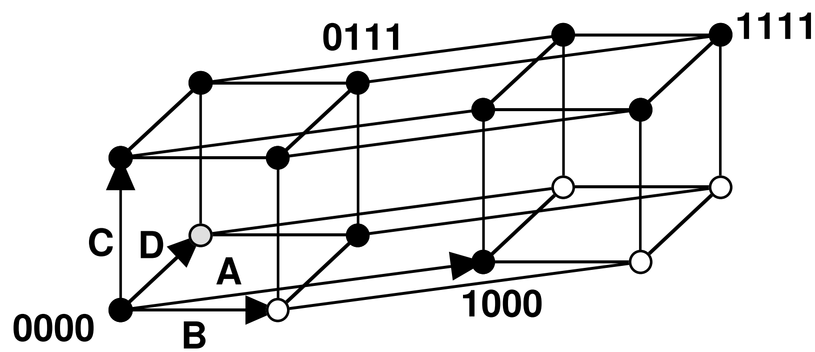
A

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

D

C

B

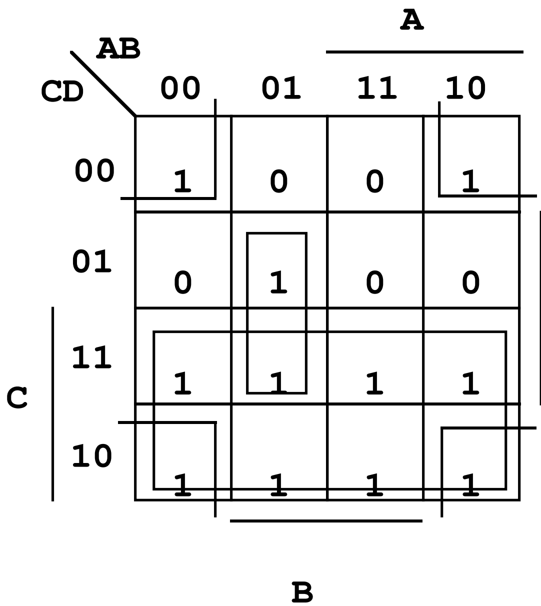


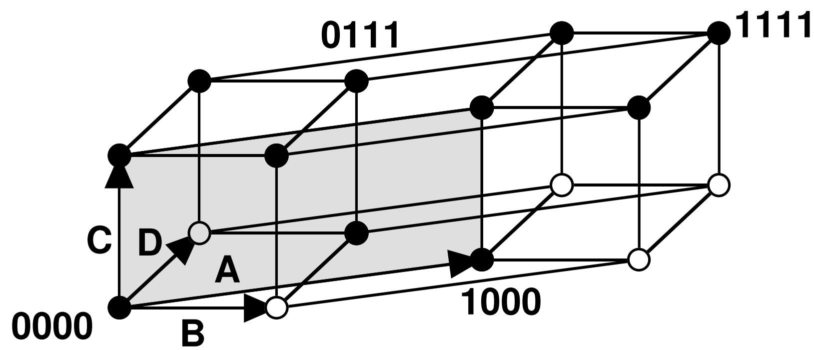
find the smallest number of the largest possible subcubes to cover the ON-set (fewer terms with fewer inputs per term)

Karnaugh map: 4-variable example (cont'd)

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

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





find the smallest number of the largest possible subcubes to cover the ON-set

Karnaugh maps and don't cares

- $f(A,B,C,D) = \Sigma\, m(1,3,5,7,9) + d(6,12,13)$

  f = without don't cares

  f = with don't cares

	CD \ AB	00	01	11 (A)	10 (A)	
	00	0	0	X	0	
	01	1	1	X	1	D
C	11	1	1	0	0	D
C	10	0	X	0	0	
			B	B		

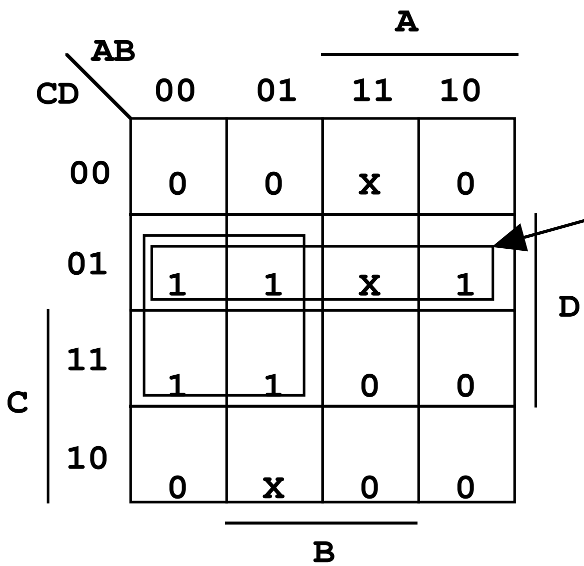
<u>don't cares</u> can be treated as 1s or 0s depending on which is more advantageous

Karnaugh maps and don't cares (cont'd)

- $f(A,B,C,D) = \Sigma\, m(1,3,5,7,9) + d(6,12,13)$

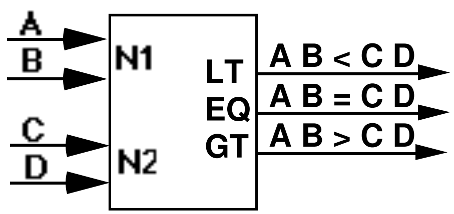
  $f = A'D + AB'C'D$ without don't cares

  $f = A'D + C'D$ with don't cares



by using don't care as a "1"
a 2-cube can be formed
rather than a 0-cube to cover
this node

Design example: two-bit comparator



A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

block diagram
and
truth table

we'll need a 4-variable Karnaugh map
for each of the 3 output functions

Design example: two-bit comparator (cont'd)

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

K-map for LT

CD \ AB	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

K-map for EQ

CD \ AB	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

K-map for GT

LT =

EQ =

GT =

Design example: two-bit comparator (cont'd)

CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

K-map for LT

CD \ AB	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

K-map for EQ

CD \ AB	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

K-map for GT

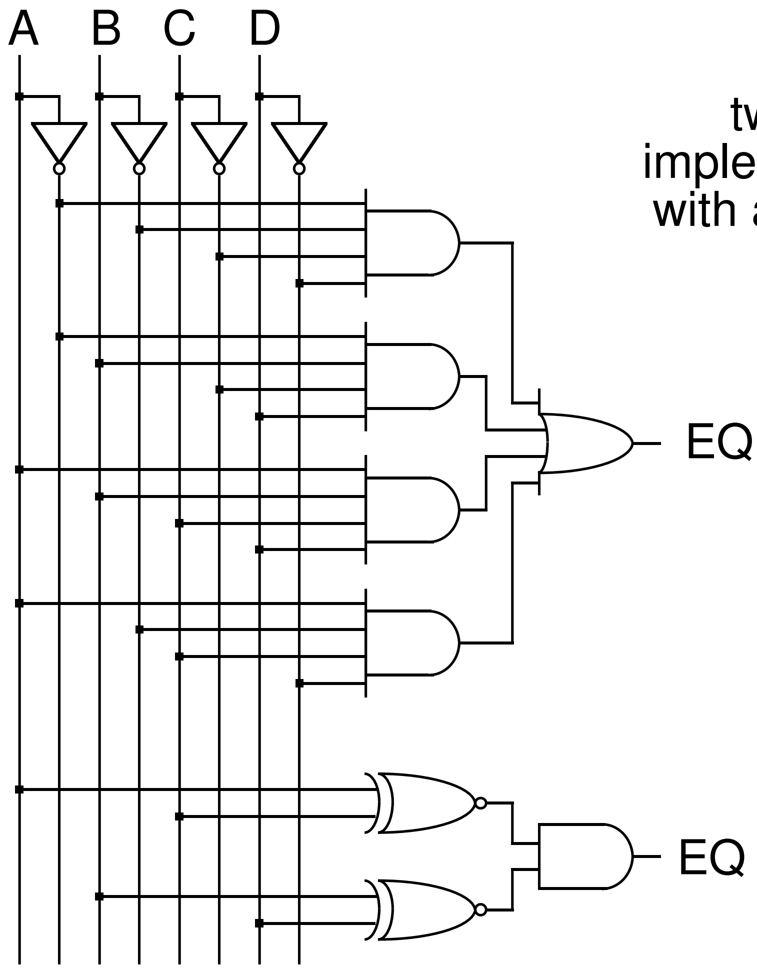
LT = A' B' D + A' C + B' C D

EQ = A' B' C' D' + A' B C' D + A B C D + A B' C D'

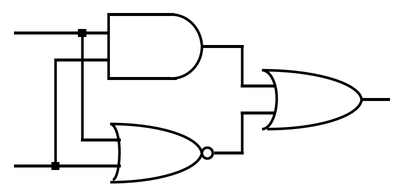
= (A xnor C) • (B xnor D)

GT = B C' D' + A C' + A B D'

Design example: two-bit comparator (cont'd)

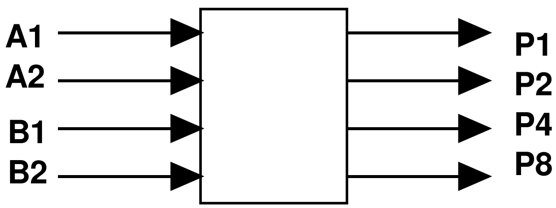


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: two-bit by two-bit multiplier



block diagram
and
truth table

A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

4-variable K-map
for each of the 4
output functions

Design example: two-bit by two-bit multiplier (cont'd)

K-map for P8

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	0
10	0	0	0	0

K-map for P4

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	1
10	0	0	1	1

K-map for P2

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	0	1
10	0	1	1	0

K-map for P1

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	0
10	0	0	0	0

Design example: two-bit by two-bit multiplier (cont'd)

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	0
10	0	0	0	0

P8 = A2A1B2B1

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	1
10	0	0	1	1

P4 = A2B2B1'
+ A2A1'B2

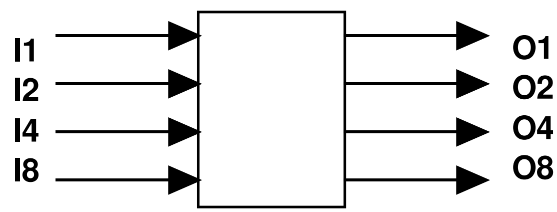
B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	0	1
10	0	1	1	0

P2 = A2'A1B2
+ A1B2B1'
+ A2B2'B1
+ A2A1'B1

B2B1 \ A2A1	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	0
10	0	0	0	0

P1 = A1B1

Design example: BCD increment by 1



block diagram
and
truth table

I8	I4	I2	I1	O8	O4	O2	O1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

4-variable K-map for each of
the 4 output functions

Design example: BCD increment by 1 (cont'd)

O8

			I8	I8	
	0	0	X	1	
	0	0	X	0	I1
I2	0	1	X	X	I1
I2	0	0	X	X	
		I4	I4		

O4

			I8	I8	
	0	1	X	0	
	0	1	X	0	I1
I2	1	0	X	X	I1
I2	0	1	X	X	
		I4	I4		

O2

			I8	I8	
	0	0	X	0	
	1	1	X	0	I1
I2	0	0	X	X	I1
I2	1	1	X	X	
		I4	I4		

O1

			I8	I8	
	1	1	X	1	
	0	0	X	0	I1
I2	0	0	X	X	I1
I2	1	1	X	X	
		I4	I4		

O8			I8		
	0	0	X	1	
	0	0	X	0	I1
I2	0	1	X	X	
	0	0	X	X	
		I4			

O4			I8		
	0	1	X	0	
	0	1	X	0	I1
I2	1	0	X	X	
	0	1	X	X	
		I4			

O8 = I4I2I1 + I8I1'

O4 = I4I2' + I4I1' + I4'I2I1

O2 = I8'I2'I1 + I2I1'

O1 = I1'

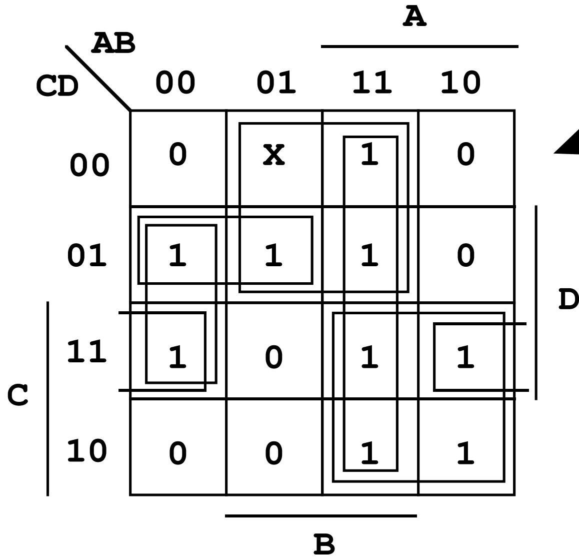
O2			I8		
	0	0	X	0	
	1	1	X	0	I1
I2	0	0	X	X	
	1	1	X	X	
		I4			

O1			I8		
	1	1	X	1	
	0	0	X	0	I1
I2	0	0	X	X	
	1	1	X	X	
		I4			

Definition of terms for two-level simplification

- **Implicant**
  - single element of ON-set or DC-set or any group of these elements that can be combined to form a subcube
- **Prime implicant**
  - implicant that can't be combined with another to form a larger subcube
- **Essential prime implicant**
  - prime implicant is essential if it alone covers an element of ON-set
  - will participate in ALL possible covers of the ON-set
  - DC-set used to form prime implicants but not to make implicant essential
- **Objective:**
  - grow implicant into prime implicants (minimize literals per term)
  - cover the ON-set with as few prime implicants as possible (minimize number of product terms)

Examples to illustrate terms

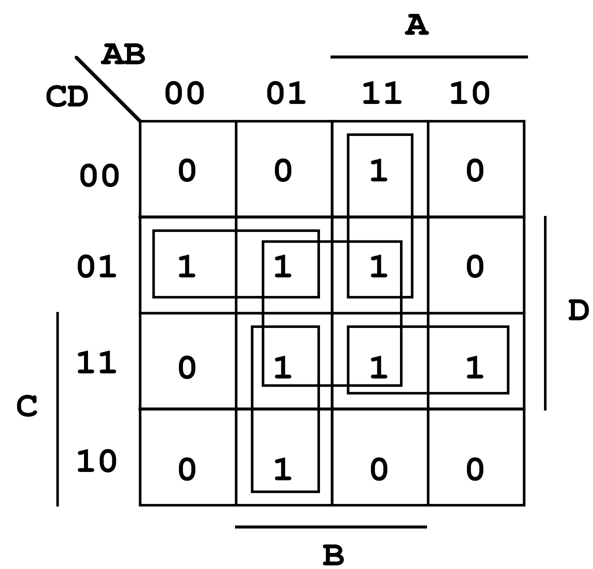


6 prime implicants:

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

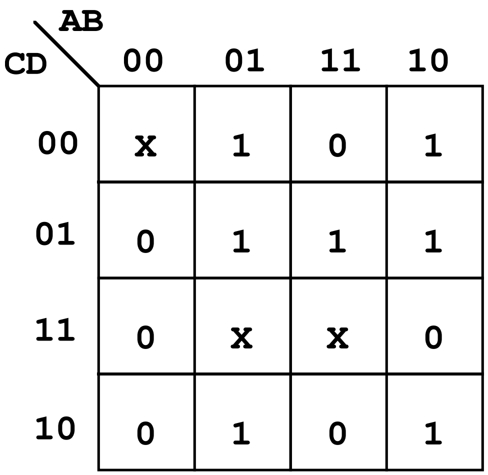
essential

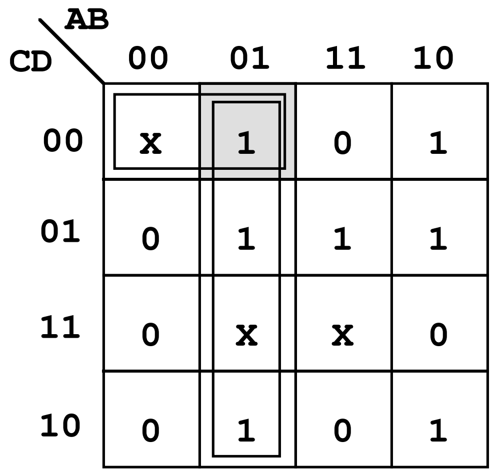
minimum cover: 4 essential implicants

Algorithm for two-level simplification

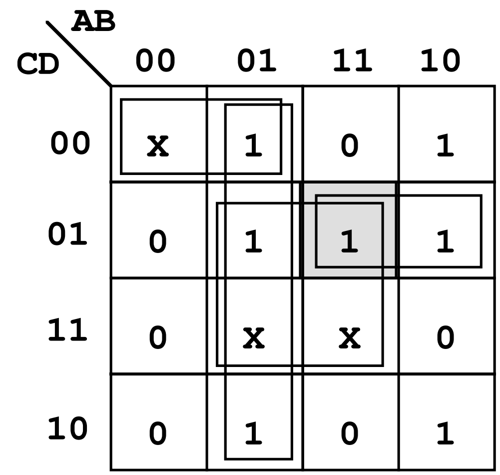
- **Algorithm: minimum sum-of-products expression from a Karnaugh map**
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - remember to consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number of elements always a power of 2)
  - Repeat Steps 1 and 2 to find all prime implicants
  - Step 3: revisit the 1s in the K-map
    - if covered by single prime implicant, it is essential, and participates in final cover
    - 1s covered by essential prime implicant do not need to be revisited
  - Step 4: if there remain 1s not covered by essential prime implicants
    - select the smallest number of prime implicants that cover the remaining 1s

Algorithm for two-level simplification (example)

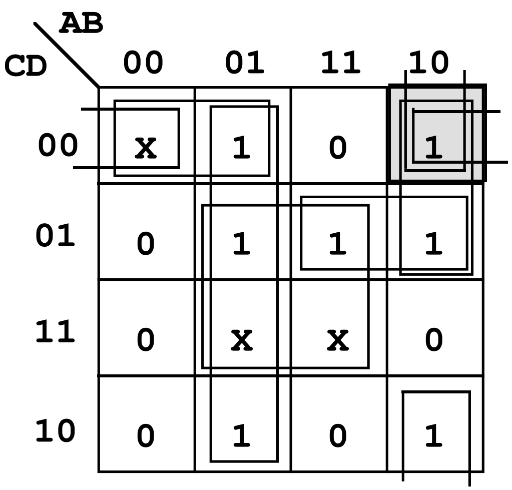




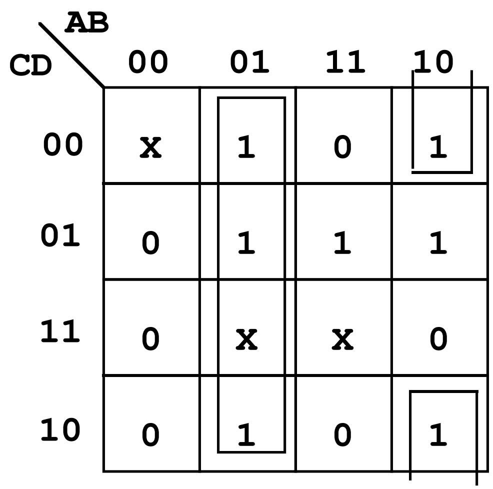
primes around A'BC'D'



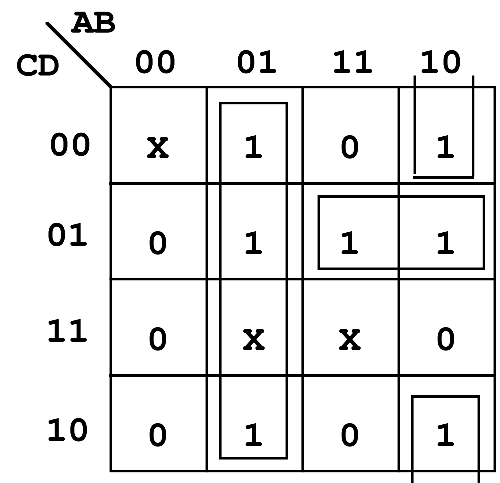
primes around ABC'D



primes around AB'C'D'



essential primes



minimum cover of 1s

Combinational logic summary

- **Logic functions, truth tables, and switches**
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- **Axioms and theorems of Boolean algebra**
  - proofs by re-writing and perfect induction
- **Gate logic**
  - networks of Boolean functions and their time behavior
- **Canonical forms**
  - two-level and incompletely specified functions
- **Simplification**
  - two-level simplification
- **Later**
  - automation of simplification
  - multi-level logic
  - design case studies
  - time behavior