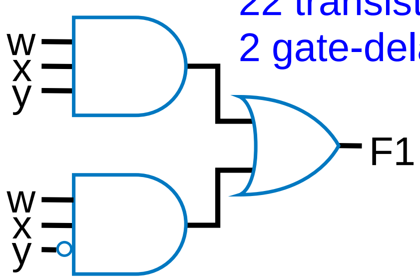
Logic Optimizations and Tradeoffs

Introduction

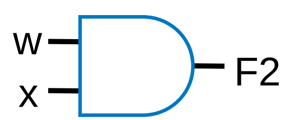
- We now know how to build digital circuits
  - How can we build ***better*** circuits?
- Let's consider two important design criteria
  - ***Delay*** – the time from inputs changing to new correct stable output
  - ***Size*** – the number of transistors
  - For quick estimation, assume
    - Every gate has delay of "1 gate-delay"
    - Every gate *input* requires 2 transistors
    - Ignore inverters

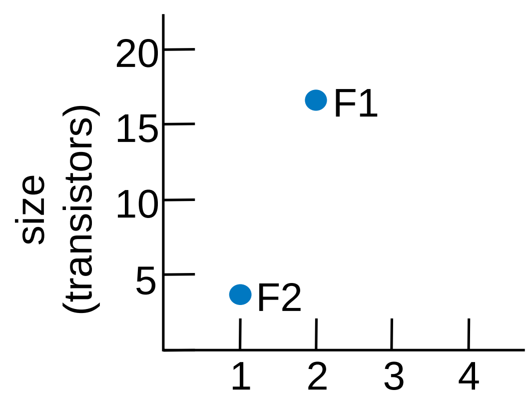
Transforming F1 to F2 represents an ***optimization***: Better in all criteria of interest

22 transistors
2 gate-delays



6 transistors
1 gate-delay





$F1 = wxy + wxy'$ $\quad$ $F2 = wx$

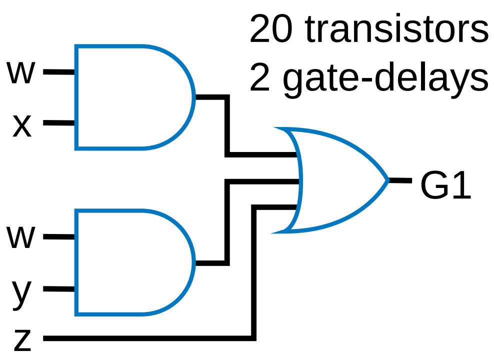
$= wx(y+y') = wx$

(a) (b) (c)

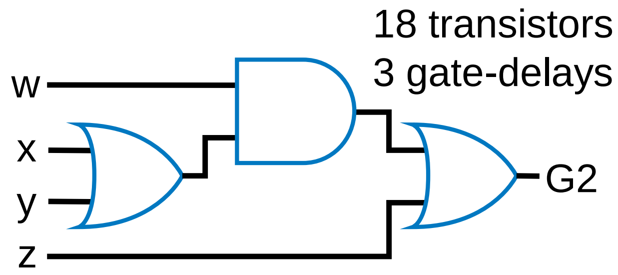
Introduction

- Tradeoff
  - Improves some, but worsens other, criteria of interest

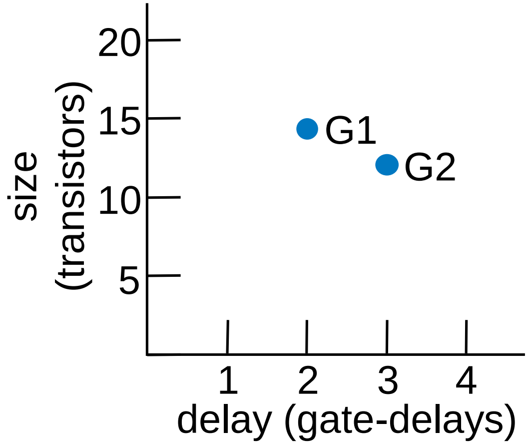
Transforming G1 to G2 represents a ***tradeoff***: Some criteria better, others worse.



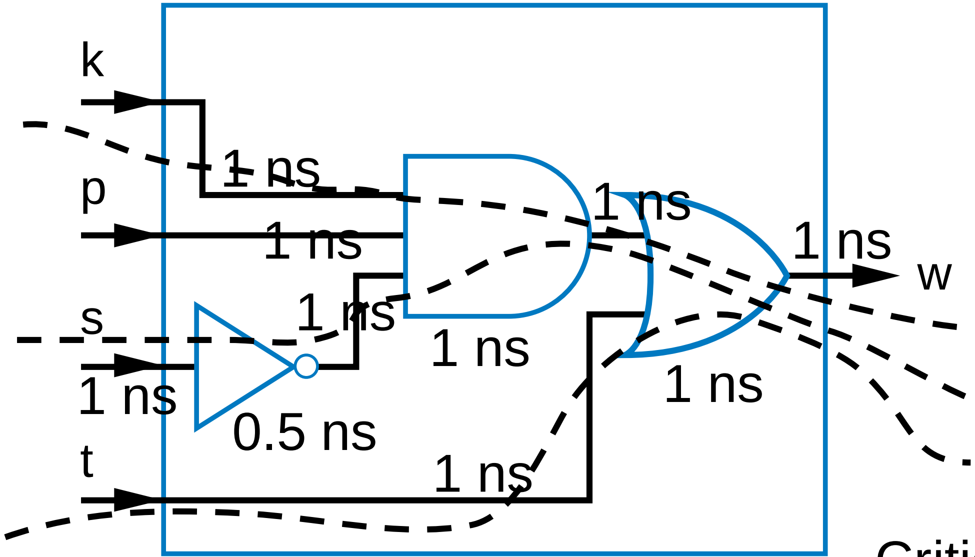
G1 = wx + wy + z



G2 = w(x+y) + z



Circuit Delay and Critical Path



1+1+1+1+1 = 5 ns

1+0.5+1+1+1+1+1 = 6.5 ns

1+1+1 = 3 ns

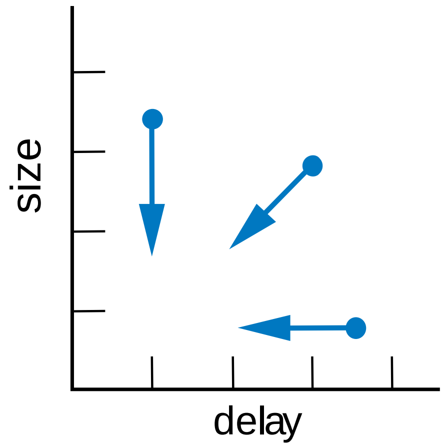
Critical path delay = 6.5 ns
Hence, circuit’s delay is 6.5 ns

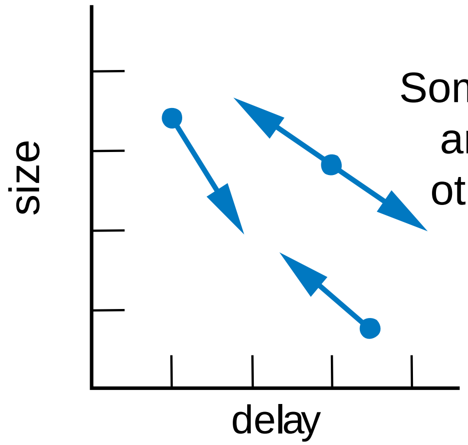
- Wires also have delay
- Assume gates and wires have delays as shown
- Path delay – time for input to affect output
- Critical path – path with longest path delay
- Circuit delay – delay of critical path

Introduction

Optimizations

All criteria of interest are improved (or at least kept the same)





Tradeoffs

Some criteria of interest are improved, while others are worsened

a

- We obviously prefer optimizations, but often must accept tradeoffs
  - You can't build a car that is the most comfortable, and has the best fuel efficiency, and is the fastest – you have to give up something to gain other things.

FUNCTION OPTIMIZATION

- Switching Function Representations can be Classified in Terms of Levels

- Number of Levels, k, is Number of Unique Boolean (binary) Operators

EXAMPLES

$$f_1 = x_1 + x_2 + x_3$$

$$f_2 = \overline{ab\overline{c}d}$$

1-Level

$$f_3 = ab + c\overline{d} + \overline{a}fe$$

$$f_4 = (a+b+\overline{c})(\overline{b}+c+d)$$

$$f_5 = a\overline{b} \oplus c\overline{d} \oplus abc$$

2-Level

$$f_6 = [(ab+\overline{c})(cd+\overline{e})] \oplus (a\overline{b}+ce)\square dc$$

multi-level

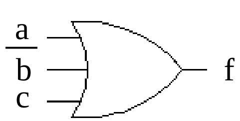
- 2^{2^n} Possible Switching Functions of *n* Variables (actually much fewer types obtained by permuting and/or complementing input variables)
- 1-Level Forms
  - Cannot Represent all Possible Functions
- 2-Level Forms
  - Can Represent all Possible Functions
  - With Additional Restrictions – ***CANONICAL***
- *k*-Level Forms ($k \geq 2$)
  - Many Different Ways to Represent a Given Functions
- If a multi-input Gate Represents a (binary or greater) Boolean Operator
  - Expression can Represent a Netlist
  - *k* Indicates "depth" of Netlist

EXAMPLES

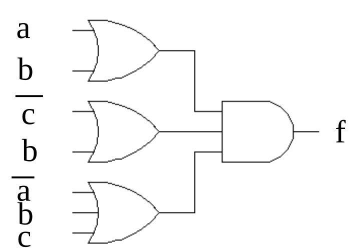
2-Level

Multi-Level

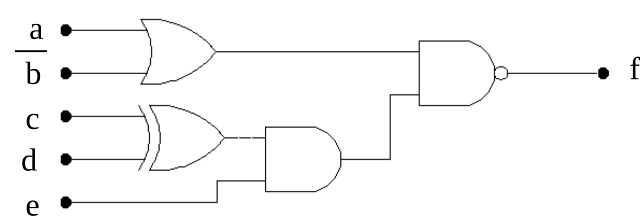
1-Level



$$f = a + \overline{b} + c$$



$$f = (a+b)(b+\overline{c})(\overline{a}+c+d)$$



$$f = \overline{(a+\overline{b})(c \oplus d)e}$$

2-Level Canonical Forms

Truth table is the unique signature of a Boolean function

Many alternative expressions (and gate realizations) may have the same truth table

Canonical form: standard form for a Boolean expression provides a unique algebraic signature

Sum of Products Form

also known as disjunctive normal form, minterm expansion

A	B	C	F	$\overline{F}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$$\begin{array}{ccccccccc} 011 & & 100 & & 101 & & 110 & & 111 \\ A'\,B\,C & + & A\,B'\,C' & + & A\,B'\,C & + & A\,B\,C' & + & A\,B\,C \end{array}$$

$$F = A'\,B\,C + A\,B'\,C' + A\,B'\,C + A\,B\,C' + A\,B\,C$$

$$F' = A'\,B'\,C' + A'\,B'\,C + A'\,B\,C'$$

Two Level Canonical Forms

Sum of Products

A	B	C	Minterms	F
0	0	0	$\bar{A}\,\bar{B}\,\bar{C} = m_0$	0
0	0	1	$\bar{A}\,\bar{B}\,C = m_1$	0
0	1	0	$\bar{A}\,B\,\bar{C} = m_2$	0
0	1	1	$\bar{A}\,B\,C = m_3$	1
1	0	0	$A\,\bar{B}\,\bar{C} = m_4$	1
1	0	1	$A\,\bar{B}\,C = m_5$	1
1	1	0	$A\,B\,\bar{C} = m_6$	1
1	1	1	$A\,B\,C = m_7$	1

product term / minterm:

ANDed product of literals in which each variable appears exactly once, in true or complemented form (but not both!)

F in canonical form:

F(A,B,C) = Σm(3,4,5,6,7)
= m3 + m4 + m5 + m6 + m7
= A' B C + A B' C' + A B' C
+ A B C' + A B C

canonical form/minimal form

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

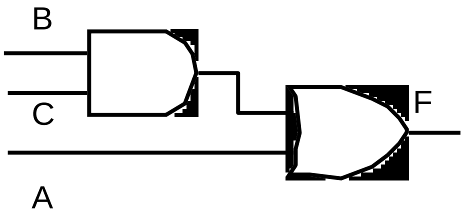
= A B' + A' B C + A B

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

= A + A' B C
= A + B C

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

Shorthand Notation for Minterms of 3 Variables



2-Level AND/OR Realization

2 Level Canonical Forms

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

A	B	C	Maxterms	F	$\overline{F}$
0	0	0	$A + B + C = M_0$	0	1
0	0	1	$A + B + \overline{C} = M_1$	0	1
0	1	0	$A + \overline{B} + C = M_2$	0	1
0	1	1	$A + \overline{B} + \overline{C} = M_3$	1	0
1	0	0	$\overline{A} + B + C = M_4$	1	0
1	0	1	$\overline{A} + B + \overline{C} = M_5$	1	0
1	1	0	$\overline{A} + \overline{B} + C = M_6$	1	0
1	1	1	$\overline{A} + \overline{B} + \overline{C} = M_7$	1	0

Maxterm:

ORed sum of literals in which each variable appears exactly once in either true or complemented form, but not both!

Maxterm form:

Find truth table rows where F is 0
0 in input column implies true literal
1 in input column implies complemented literal

Maxterm Shorthand Notation for a Function of Three Variables

F(A,B,C) = ΠM(0,1,2)

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

F'(A,B,C) = ΠM(3,4,5,6,7)

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

Gate Logic: Incompletely Specified Functions

n input functions have 2^n possible input configurations

for a given function, not all input configurations may be possible

this fact can be exploited during circuit minimization!

E.g., Binary Coded Decimal Digit Increment by 1

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 input patterns should never be encountered in practise associated output values are "Don't Cares"

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

Incompletely Specified Functions

Don't Cares and Canonical Forms

Canonical Representations of the BCD Increment by 1 Function with don’t cares added

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

$Z = \Sigma m(0, 2, 4, 6, 8) + d(10, 11, 12, 13, 14, 15)$
(Any subset of don’t cares can be added to simplify)

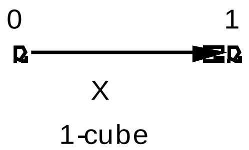
$Z = M1 \bullet M3 \bullet M5 \bullet M7 \bullet M9 \bullet D10 \bullet D11 \bullet D12 \bullet D13 \bullet D14 \bullet D15$
(All don’t cares are multiplied. But any subset can be chosen for simplification)

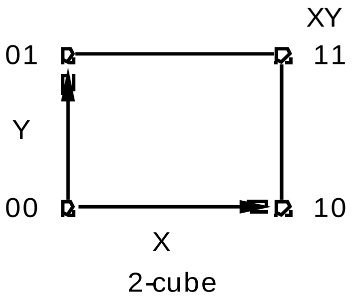
$Z = \Pi M(1, 3, 5, 7, 9) \bullet D(10, 11, 12, 13, 14, 15)$

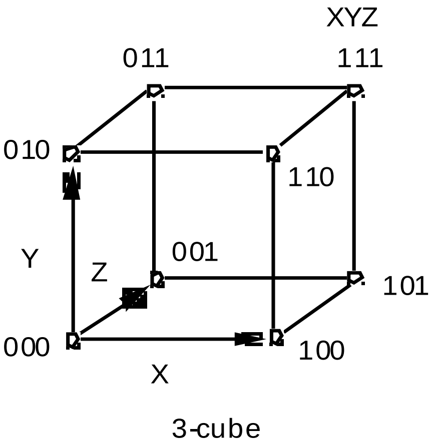
Two-Level Simplification

Boolean Cubes

Visual technique for identifying when the Uniting Theorem can be applied

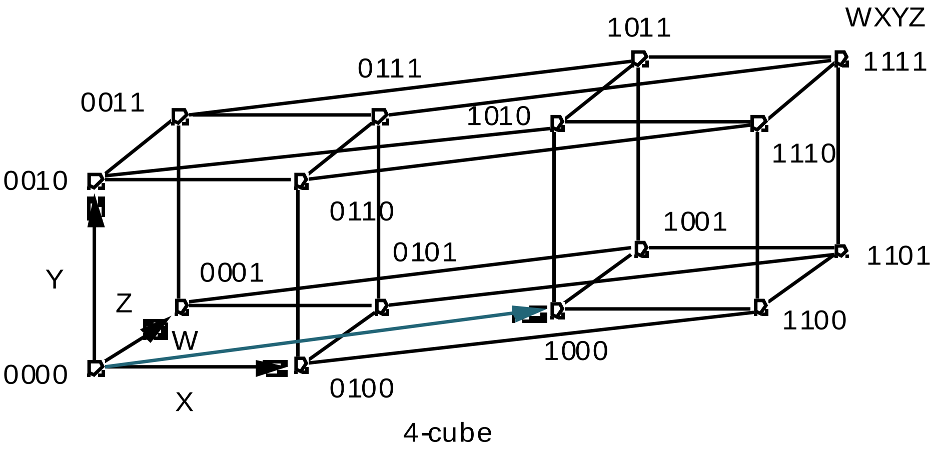






Just another way to represent the truth table

n input variables = n dimensional "cube"



Two-Level Simplification

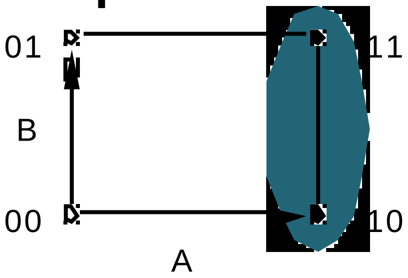
Mapping Truth Tables onto Boolean Cubes

ON-set = filled-in nodes

OFF-set = empty nodes

DC-set = X'd nodes

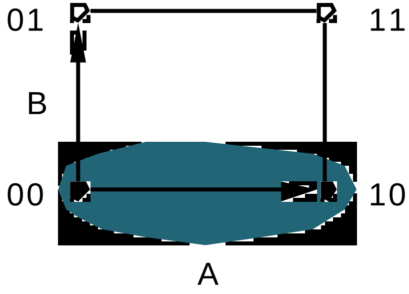
F



A asserted and unchanged

B varies within loop

G



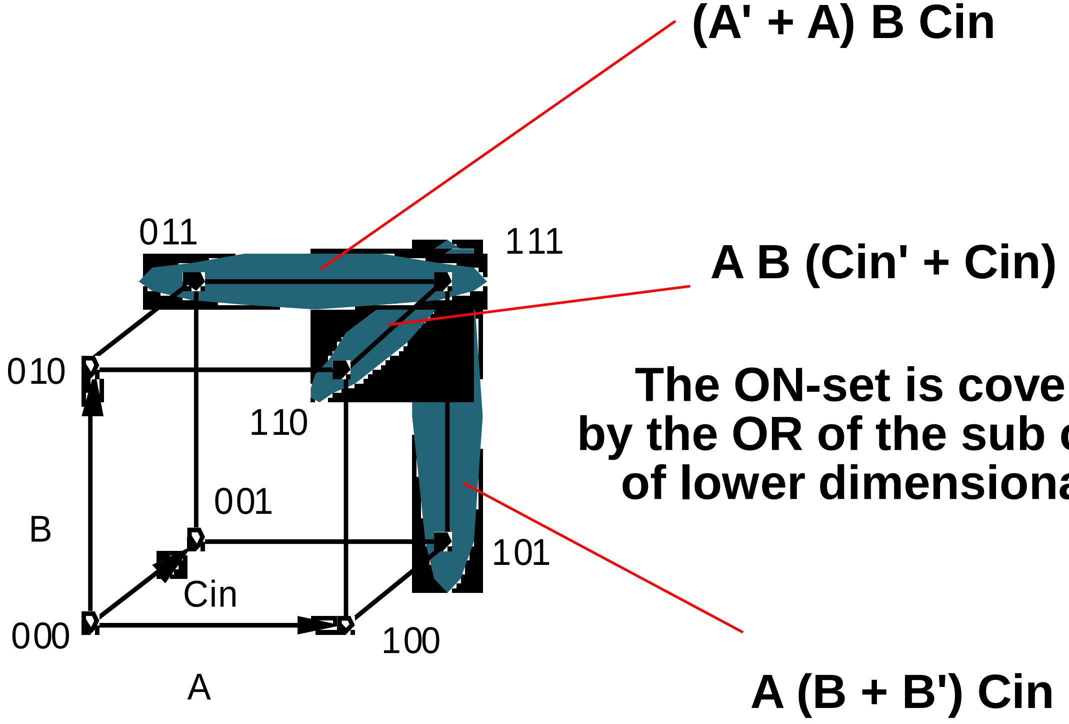
A varies within loop

B complemented and unchanged

Two-Level Simplification

Three variable example: Full Adder Carry Out

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

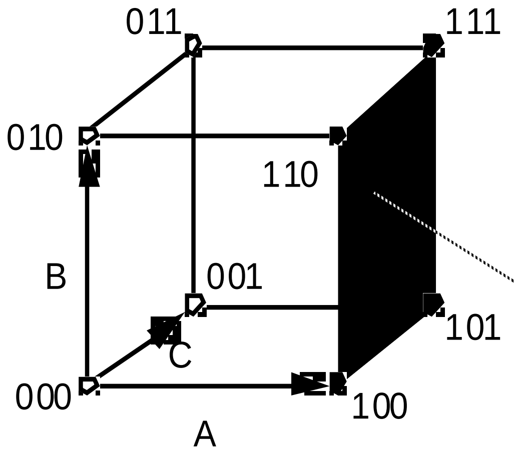


The ON-set is covered by the OR of the sub cubes of lower dimensionality

Cout = B Cin + A B + A Cin

Two-Level Simplification

Sub cubes of Higher Dimensions than 2



F(A,B,C) = Σm(4,5,6,7)

On-set forms a rectangle,
i.e., a cube of two dimensions

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

A is asserted and unchanged
B and C vary

This sub cube represents the literal A

Two-Level Simplification

In a 3-cube:

a 0-cube, i.e., a single node, yields a term in three literals

a 1-cube, i.e., a line of two nodes, yields a term in two literals

a 2-cube, i.e., a plane of four nodes, yields a term in one literal

a 3-cube, i.e., a cube of eight nodes, yields a constant term "1"

In general,

an m - sub cube within an n-cube (m < n) yields a term with n - m literals

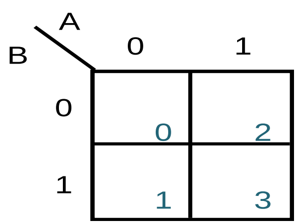
Two-Level Simplification

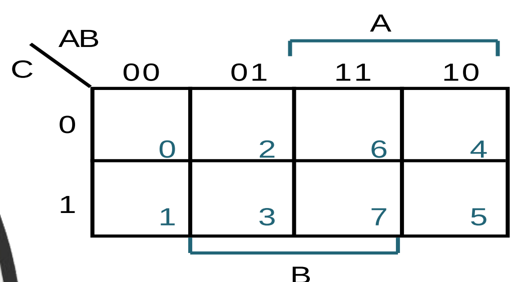
Karnaugh Map Method

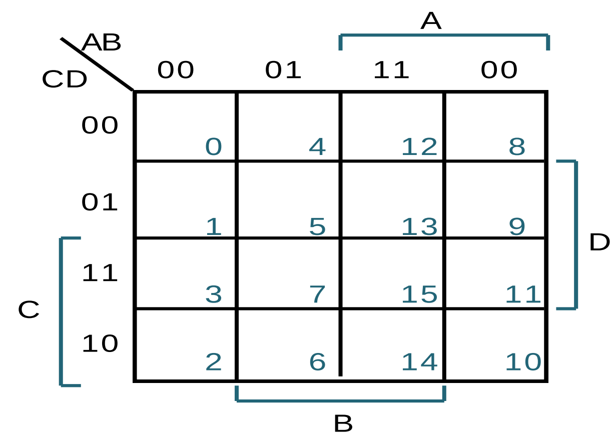
K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions

Beyond that, computer-based methods are needed

2-variable K-map







4-variable K-map

***bering Scheme:* 00, 01, 11, 10**
Code — only a single bit changes from code word to next code word

Karnaugh Maps for Two-Level Size Optimization

- Easy to miss possible opportunities to combine terms when doing algebraically
- ***Karnaugh Maps (K-maps)***
  - **Graphical** method to help us find opportunities to combine terms
  - Minterms <u>differing in one variable</u> are *adjacent* in the map
  - Can clearly see opportunities to combine terms – look for adjacent 1s
    - For F, clearly two opportunities
    - Top left circle is *shorthand* for: x'y'z'+x'y'z = x'y'(z'+z) = x'y'(1) = x'y'
    - Draw circle, write term that has all the literals except the one that changes in the circle
      - Circle xy, x=1 & y=1 in both cells of the circle, but z changes (z=1 in one cell, 0 in the other)
    - Minimized function: OR the final terms

Notice not in binary order

K-map

F x \ yz	00	01	11	10
0	x'y'z'	x'y'z	x'yz	x'yz'
1	xy'z'	xy'z	xyz	xyz'

Treat left & right as adjacent too

F = x'y'z + xyz + xyz' + x'y'z'

F x \ yz	00	01	11	10
0	1	1	0	0
1	0	0	1	1

F x \ yz	00	01	11	10
0	1	1	0	0
1	0	0	1	1

x'y'   xy

F = x'y' + xy

Easier than algebraically:

F = xyz + xyz' + x'y'z' + x'y'z
F = xy(z + z') + x'y'(z + z')
F = xy*1 + x'y'*1
F = xy + x'y'

K-maps

- Four adjacent 1s means two variables can be eliminated
  - Makes intuitive sense – those two variables appear in all combinations, so one term *must* be true
  - Draw one big circle – *shorthand* for the algebraic transformations above

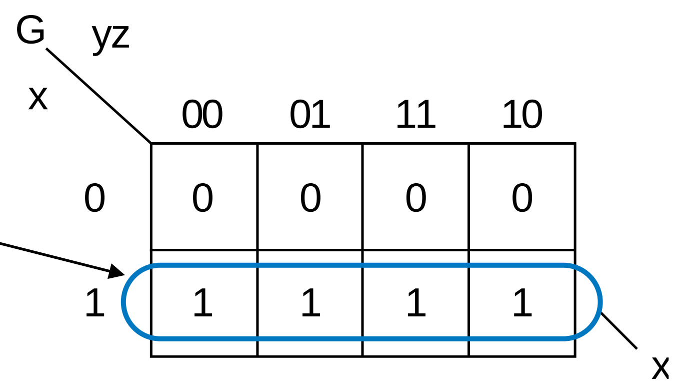
$G = xy'z' + xy'z + xyz + xyz'$

$G = x(y'z' + y'z + yz + yz')$ *(must be true)*

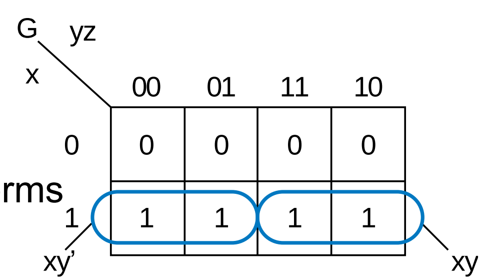
$G = x(y'(z'+z) + y(z+z'))$

$G = x(y'+y)$

$G = x$



Draw the biggest circle possible, or you'll have more terms than really needed



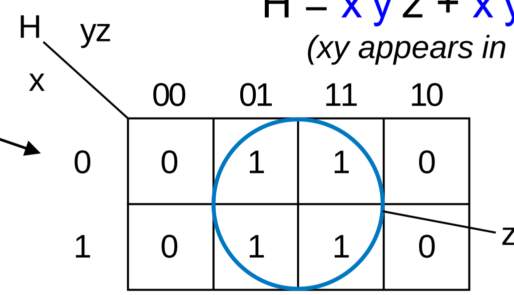
a

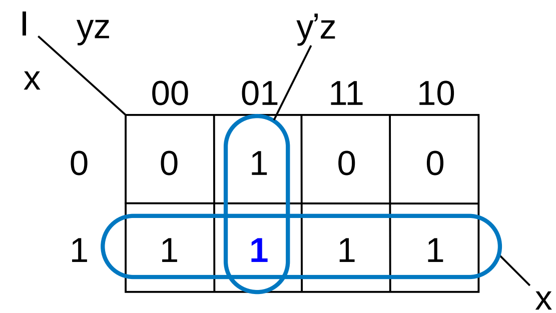
K-maps

- Four adjacent cells can be in shape of a square
- *OK* to cover a 1 twice
  - Just like duplicating a term
    - Remember, c + d = c + d + d
- No *need* to cover 1s more than once
  - Yields extra terms – not minimized

H = x'y'z + x'yz + xy'z + xyz

(xy appears in all combinations)





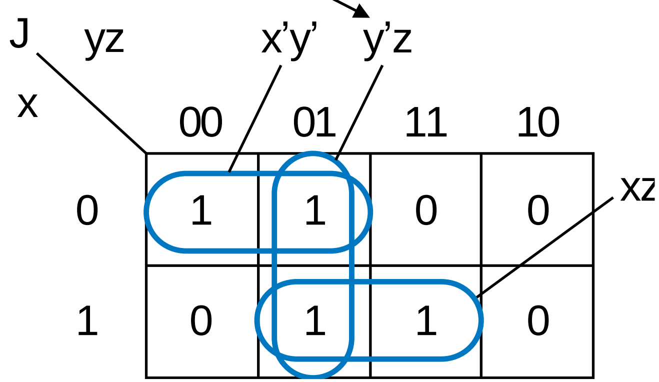
The two circles are shorthand for:

I = x'y'z + xy'z' + xy'z + xyz + xyz'

I = x'y'z + xy'z + xy'z' + xy'z + xyz + xyz'

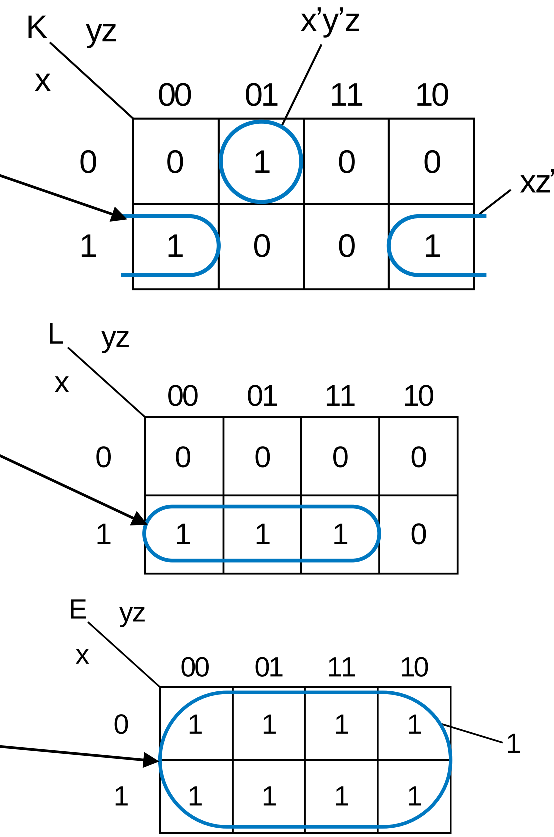
I = (x'y'z + xy'z) + (xy'z' + xy'z + xyz + xyz')

I = (y'z) + (x)



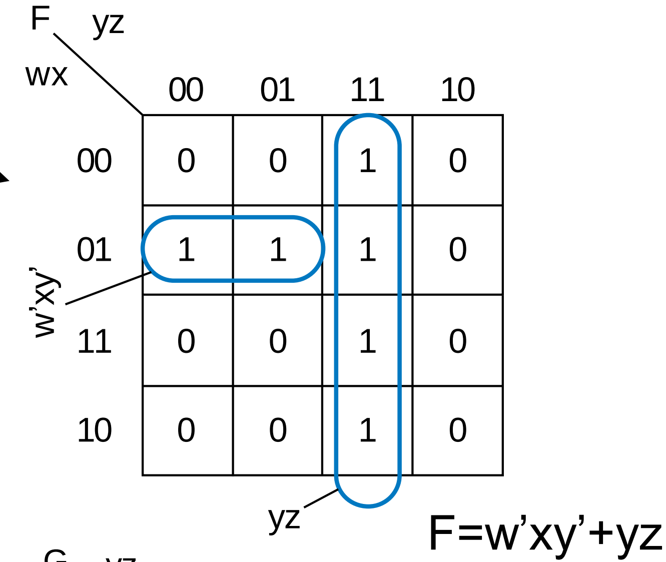
K-maps

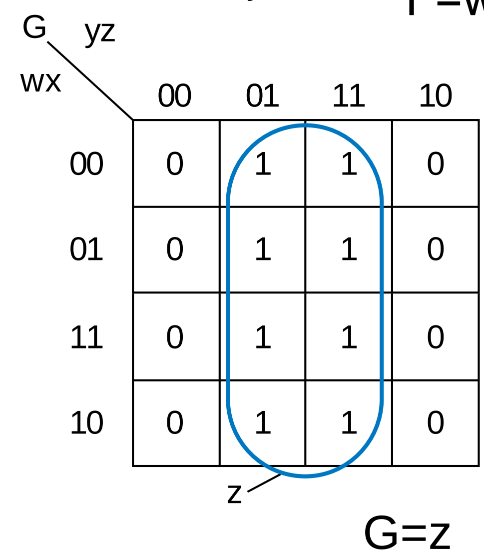
- Circles can cross left/right sides
  - Remember, edges are adjacent
    - Minterms differ in one variable only
- Circles must have 1, 2, 4, or 8 cells – 3, 5, or 7 not allowed
  - 3/5/7 doesn't correspond to algebraic transformations that combine terms to eliminate a variable
- Circling all the cells is OK
  - Function just equals 1

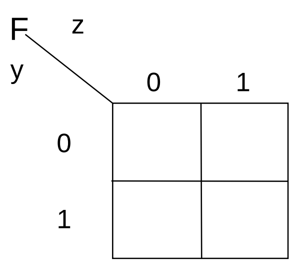


K-maps for Four Variables

- Four-variable K-map follows same principle
  - Adjacent cells differ in one variable
  - Left/right adjacent
  - Top/bottom also adjacent
- 5 and 6 variable maps exist
  - But hard to use
- Two-variable maps exist
  - But not very useful – easy to do algebraically by hand







Two-Level Size Optimization Using K-maps

General K-map method

1. *Convert* the function's equation into sum-of-minterms form
2. *Place* 1s in the appropriate K-map cells for each minterm
3. *Cover* all 1s by drawing the fewest largest circles, with every 1 included at least once; write the corresponding term for each circle
4. *OR* all the resulting terms to create the minimized function.

Two-Level Size Optimization Using K-maps

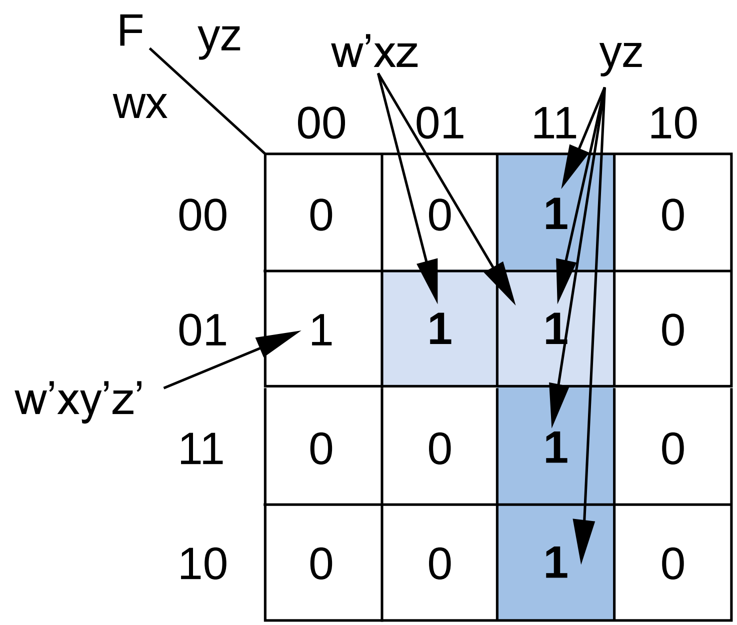
General K-map method

1. *Convert* the function's equation into sum-of-minterms form
2. *Place* 1s in the appropriate K-map cells for each minterm

Common to revise (1) and (2):

- Create *sum-of-products*
- Draw 1s for each product

Ex: F = w'xz + yz + w'xy'z'



Two-Level Size Optimization Using K-maps

General K-map method

1. *Convert* the function's equation into sum-of-minterms form
2. *Place* 1s in the appropriate K-map cells for each minterm
3. *Cover* all 1s by drawing the fewest largest circles, with every 1 included at least once; write the corresponding term for each circle
4. *OR* all the resulting terms to create the minimized function.

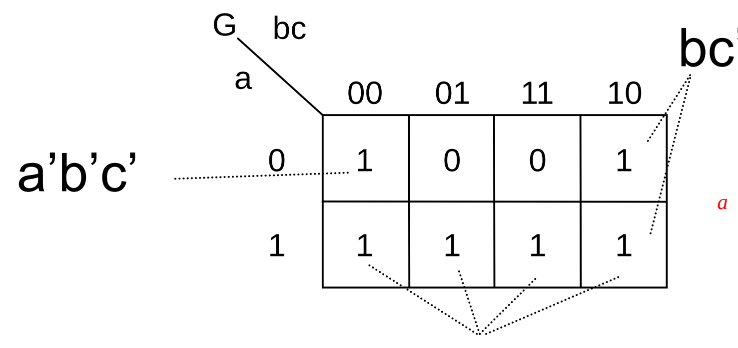
Example: Minimize:

G = a + a'b'c' + b*(c' + bc')

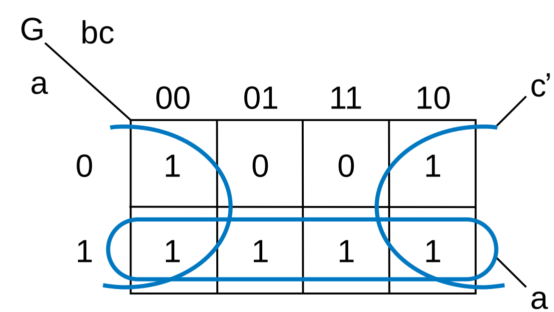
1. Convert to sum-of-products

G = a + a'b'c' + bc' + bc'

2. Place 1s in appropriate cells



3. Cover 1s

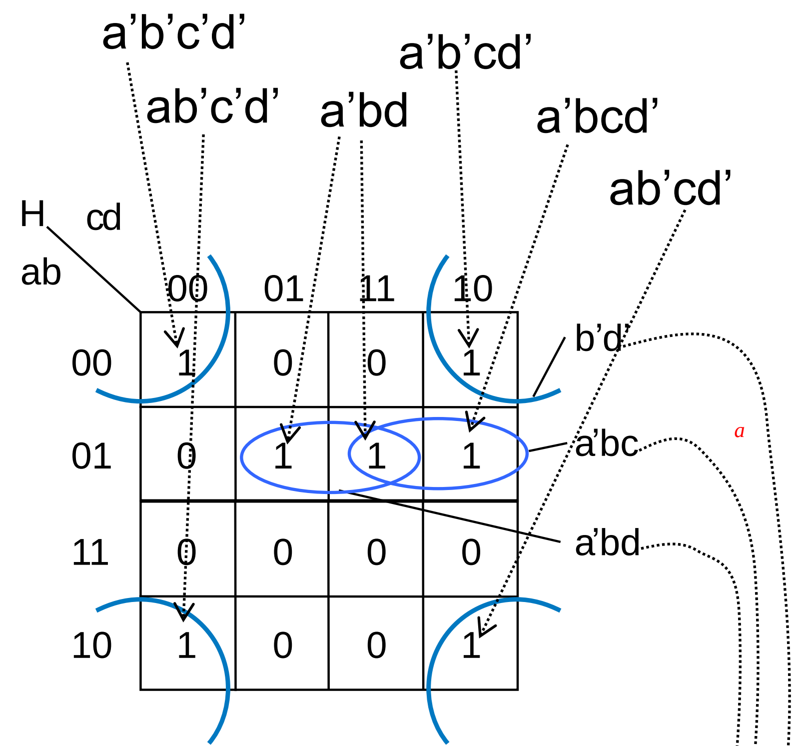


4. OR terms: **G = a + c'**

Two-Level Size Optimization Using K-maps
– Four Variable Example

- Minimize:
  - H = a’b’(cd’ + c’d’) + ab’c’d’ + ab’cd’ + a’bd + a’bcd’

1. Convert to sum-of-products:
   - H = a’b’cd’ + a’b’c’d’ + ab’c’d’ + ab’cd’ + a’bd + a’bcd’
2. Place 1s in K-map cells
3. Cover 1s
4. OR resulting terms

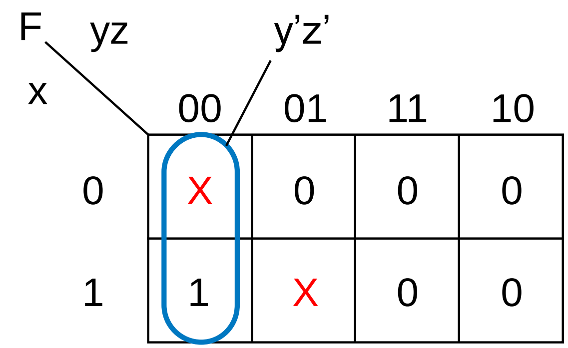


Funny-looking circle, but remember that left/right adjacent, and top/bottom adjacent

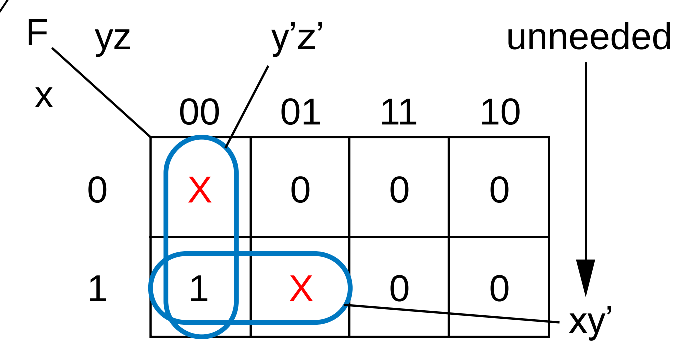
H = b’d’ + a’bc + a’bd

Don’t Care Input Combinations

- What if we know that particular input combinations can never occur?
  - e.g., Minimize F = xy’z’, given that x’y’z’ (xyz=000) can *never* be true, and that xy’z (xyz=101) can *never* be true
  - So it doesn’t matter what F outputs when x’y’z’ or xy’z is true, because those cases *will never occur*
  - Thus, make F be 1 or 0 for those cases *in a way that best minimizes the equation*
- On K-map
  - Draw **X**s for don’t care combinations
    - Include X in circle ONLY if minimizes equation
    - Don’t include other Xs



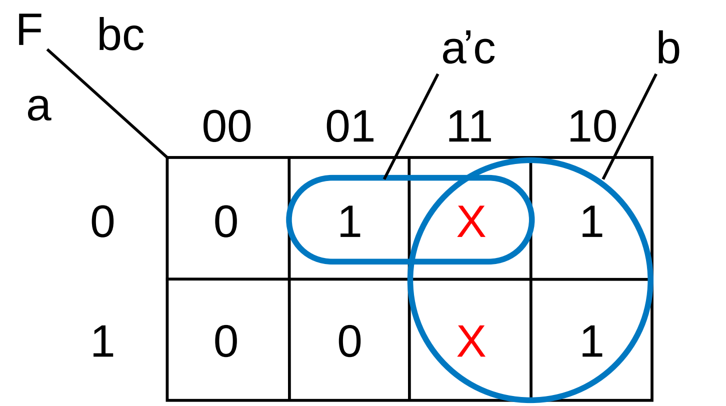
Good use of don’t cares



Unnecessary use of don’t cares; results in extra term

Optimization Example using Don’t Cares

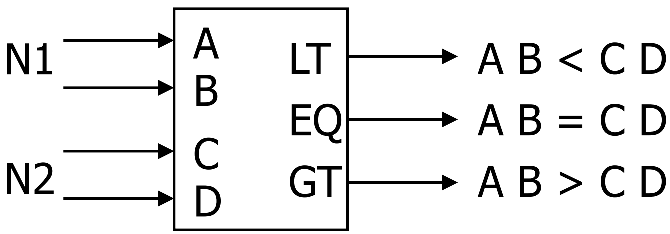
- Minimize:
  - F = a’bc’ + abc’ + a’b’c
  - Given don’t cares: a’bc, abc



F = a’c + b

- Note: Introduce don’t cares with caution
  - Must be *sure* that we really don’t care what the function outputs for that input combination
  - If we do care, even the slightest, then it’s probably safer to set the output to 0

Design example: two-bit comparator

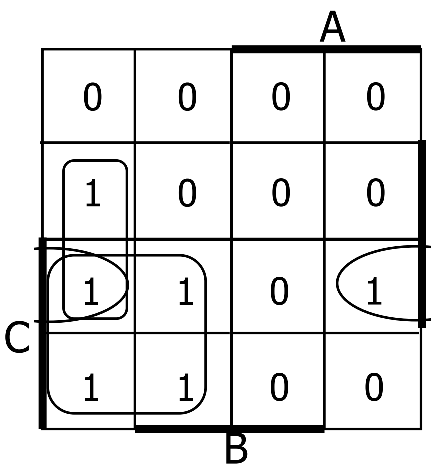


A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

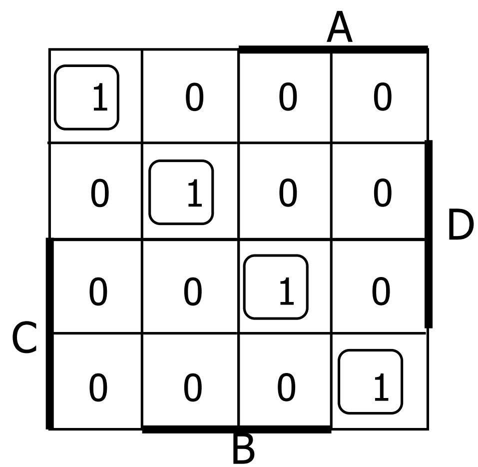
block diagram
and
truth table

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

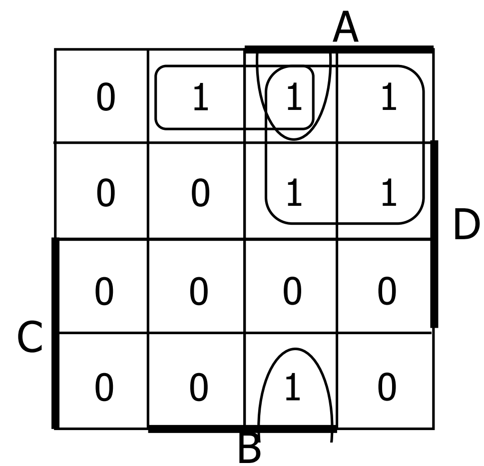
Design example: two-bit comparator (cont'd)



K-map for LT



K-map for EQ



K-map for GT

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

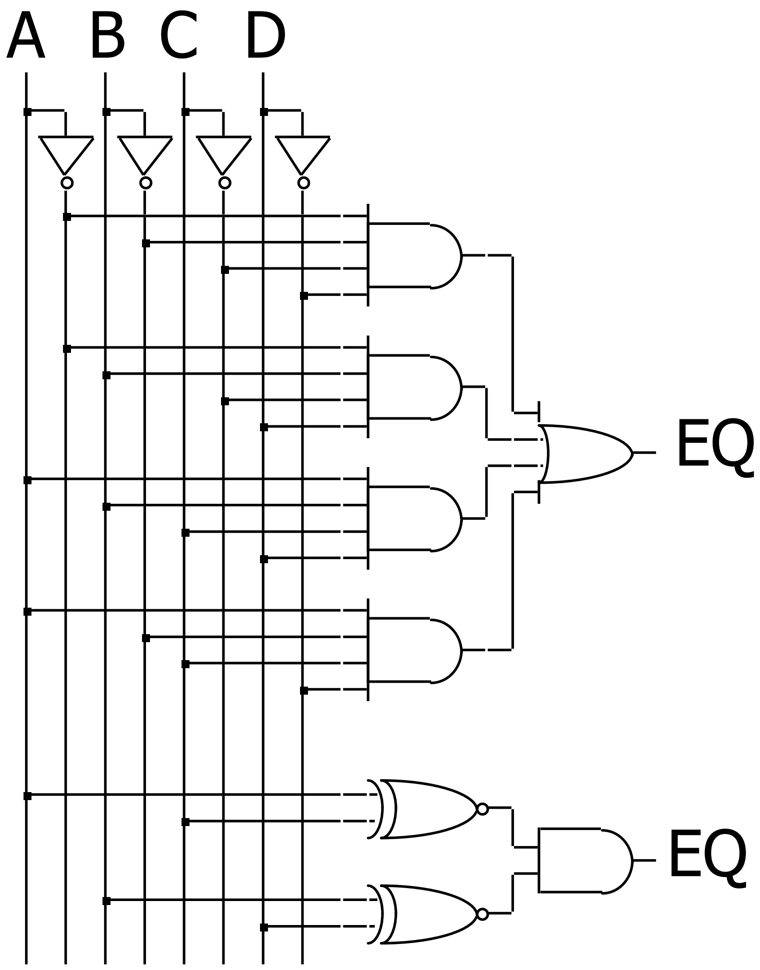
= (A xnor C) • (B xnor D)

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

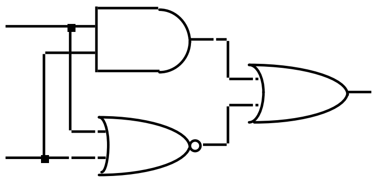
GT = B C' D' + A C' + A B D'

LT and GT are similar (flip A/C and B/D)

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

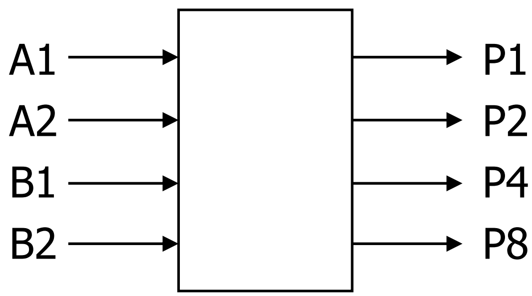


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier

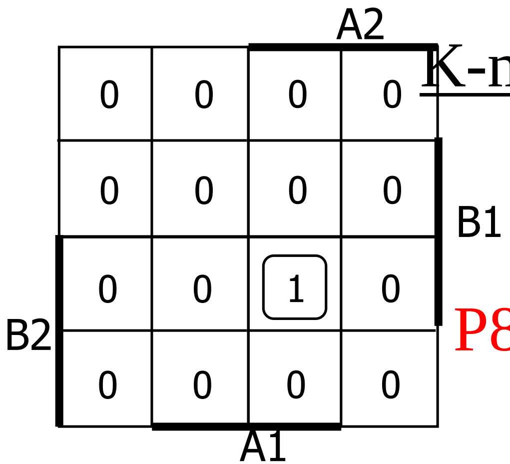


block diagram
and
truth table

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

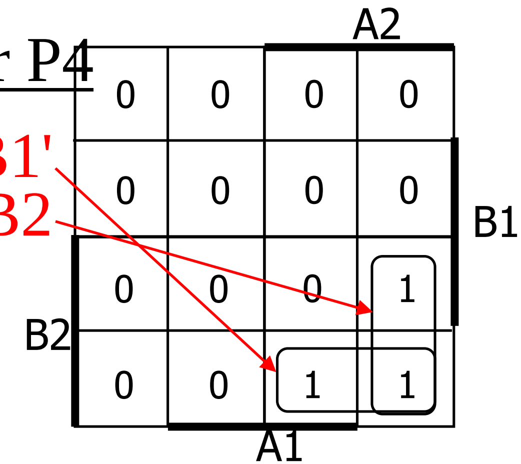
Design example: 2x2-bit multiplier (cont'd)



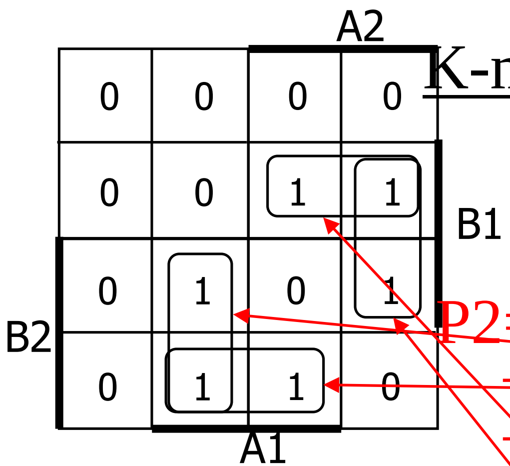
K-map for P8

P8 = A2A1B2B1

K-map for P4



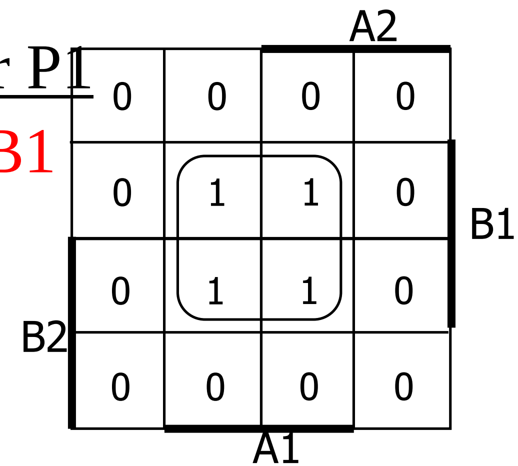
P4= A2B2B1'
+ A2A1'B2



K-map for P2

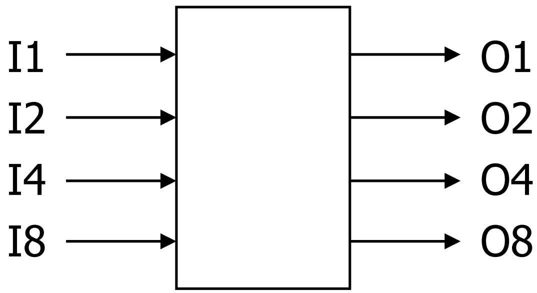
P2= A2'A1B2
+ A1B2B1'
+ A2B2'B1
+ A2A1'B1

K-map for P1



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 above columns 3–4; I4 below columns 2–3; I1 beside rows 2–3; I2 beside rows 3–4)

	0	0	X	1
	0	0	X	0
I2	0	1	X	X
I2	0	0	X	X

O4 (I8 above columns 3–4; I4 below columns 2–3; I1 beside rows 2–3; I2 beside rows 3–4)

	0	1	X	0
	0	1	X	0
I2	1	0	X	X
I2	0	1	X	X

O8 = I4 I2 I1 + I8 I1'

O4 = I4 I2' + I4 I1' + I4' I2 I1

O2 = I8' I2' I1 + I2 I1'

O1 = I1'

O2 (I8 above columns 3–4; I4 below columns 2–3; I1 beside rows 2–3; I2 beside rows 3–4)

	0	0	X	0
	1	1	X	0
I2	0	0	X	X
I2	1	1	X	X

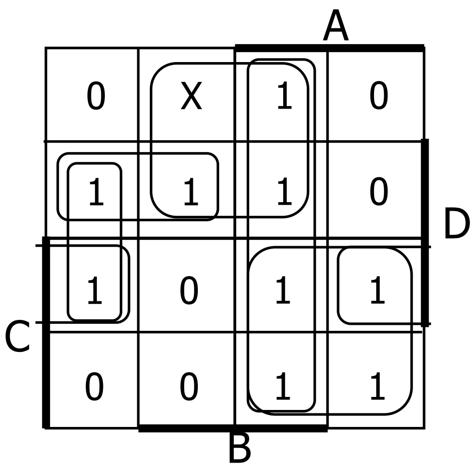
O1 (I8 above columns 3–4; I4 below columns 2–3; I1 beside rows 2–3; I2 beside rows 3–4)

	1	1	X	1
	0	0	X	0
I2	0	0	X	X
I2	1	1	X	X

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

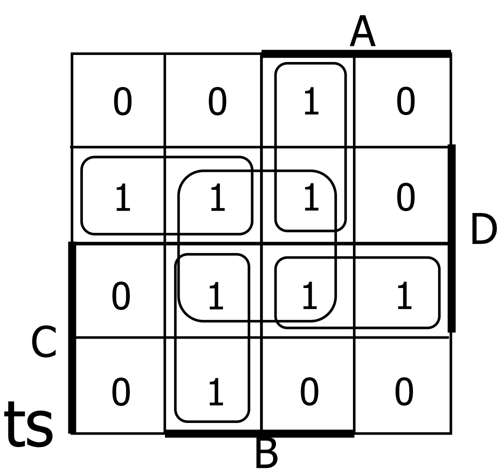
essential

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

5 prime implicants:
BD, ABC', ACD, A'BC, A'C'D

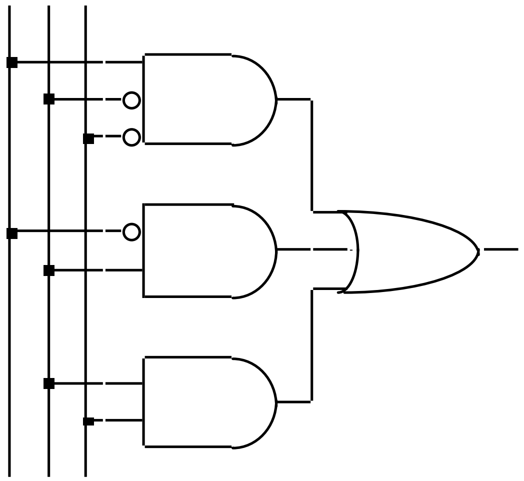
essential

minimum cover: 4 essential implicants

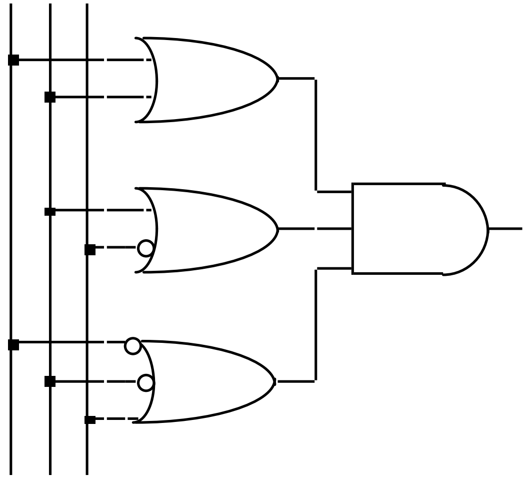


Implementations of two-level logic

- Sum-of-products
  - AND gates to form product terms (minterms)
  - OR gate to form sum

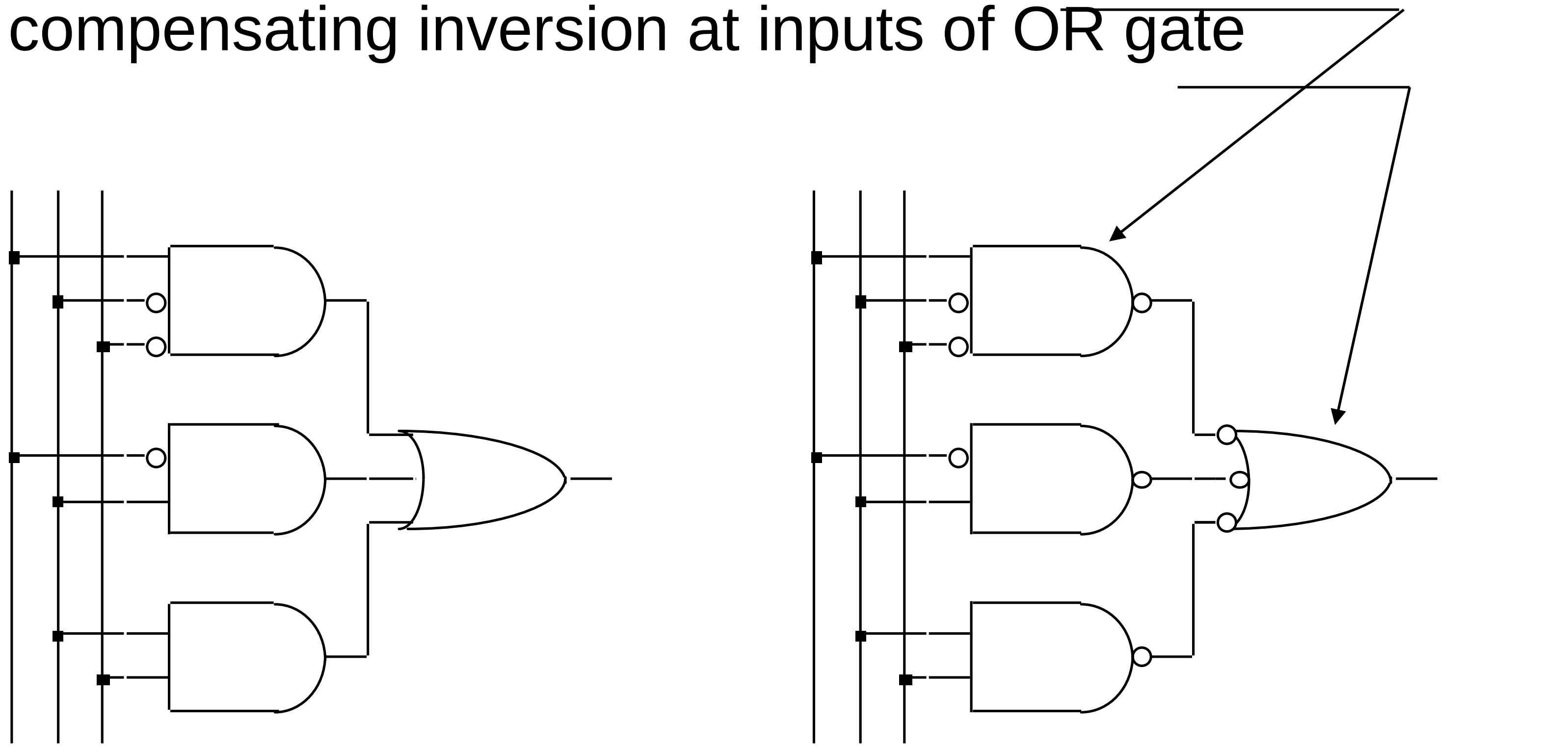


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



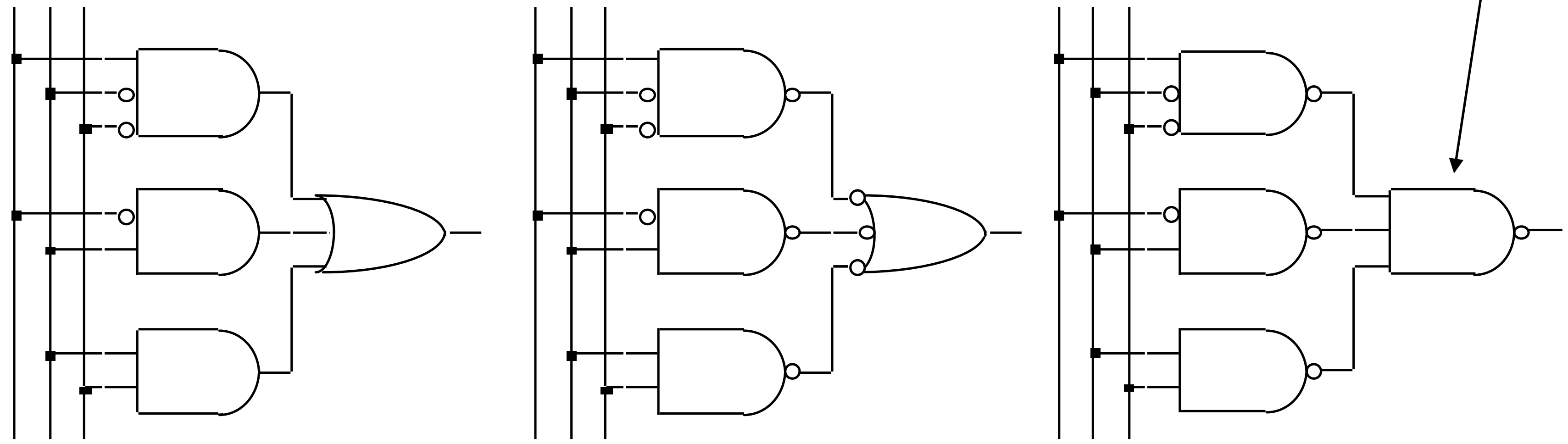
Two-level logic using NAND gates

- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate



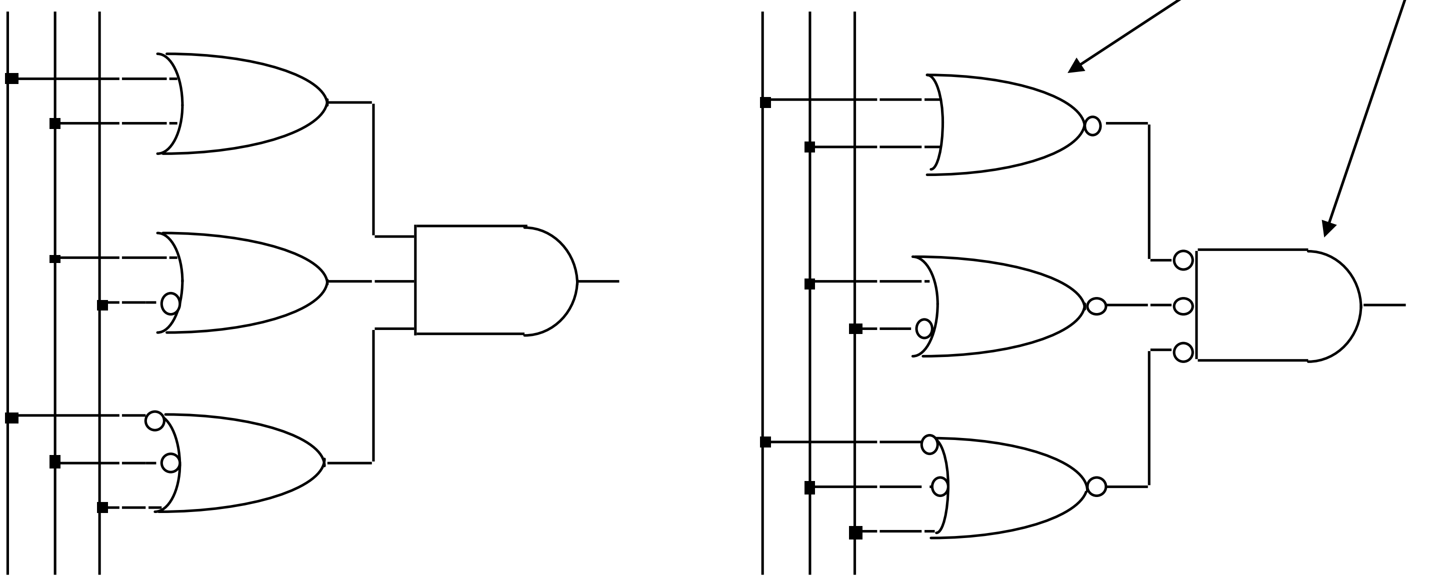
Two-level logic using NAND gates (cont'd)

- OR gate with inverted inputs is a NAND gate
  - de Morgan's: $A' + B' = (A \cdot B)'$
- Two-level NAND-NAND network
  - inverted inputs are not counted
  - in a typical circuit, inversion is done once and signal distributed



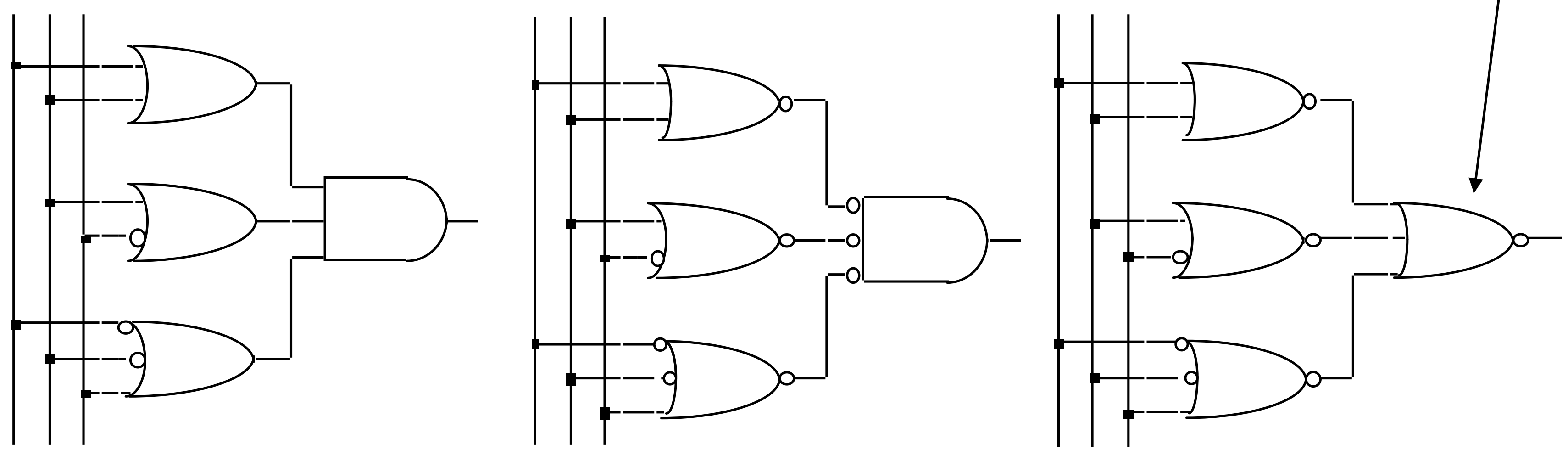
Two-level logic using NOR gates

- Replace maxterm OR gates with NOR gates
- Place compensating inversion at inputs of AND gate



Two-level logic using NOR gates (cont'd)

- AND gate with inverted inputs is a NOR gate
  - de Morgan's: A' • B' = (A + B)'
- Two-level NOR-NOR network
  - inverted inputs are not counted
  - in a typical circuit, inversion is done once and signal distributed

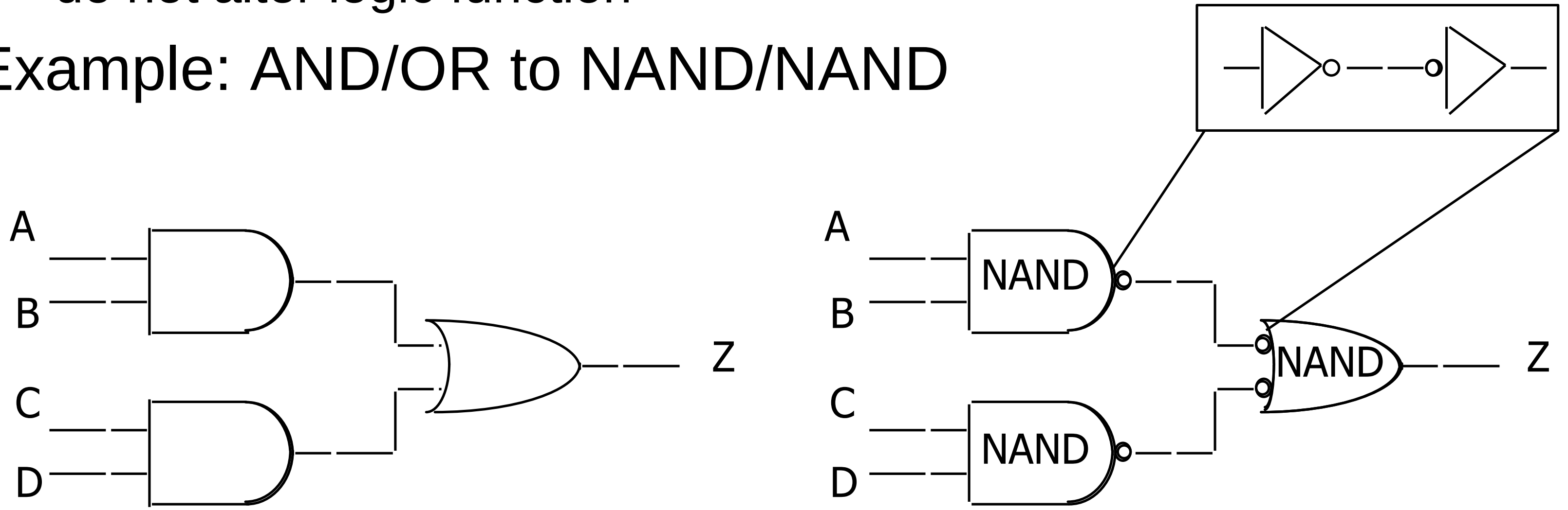


Two-level logic using NAND and NOR gates

- NAND-NAND and NOR-NOR networks
  - de Morgan's law: (A + B)' = A' • B'   (A • B)' = A' + B'
  - written differently: A + B = (A' • B')'   (A • B) = (A' + B')'
- In other words —
  - OR is the same as NAND with complemented inputs
  - AND is the same as NOR with complemented inputs
  - NAND is the same as OR with complemented inputs
  - NOR is the same as AND with complemented inputs

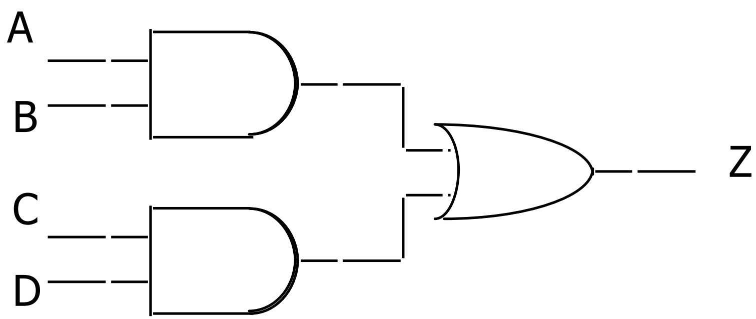
Conversion between forms

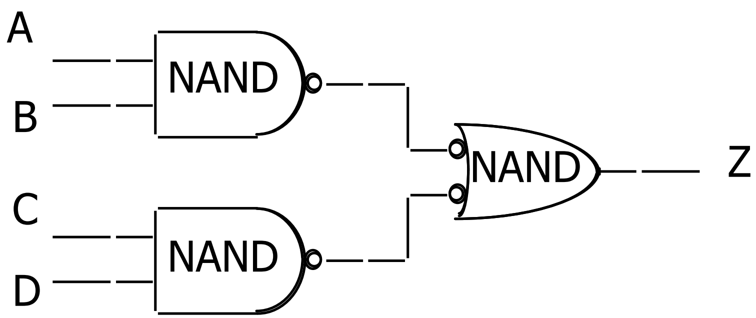
- Convert from networks of ANDs and ORs to networks of NANDs and NORs
  - introduce appropriate inversions ("bubbles")
- Each introduced "bubble" must be matched by a corresponding "bubble"
  - conservation of inversions
  - do not alter logic function
- Example: AND/OR to NAND/NAND



Conversion between forms (cont'd)

- Example: verify equivalence of two forms

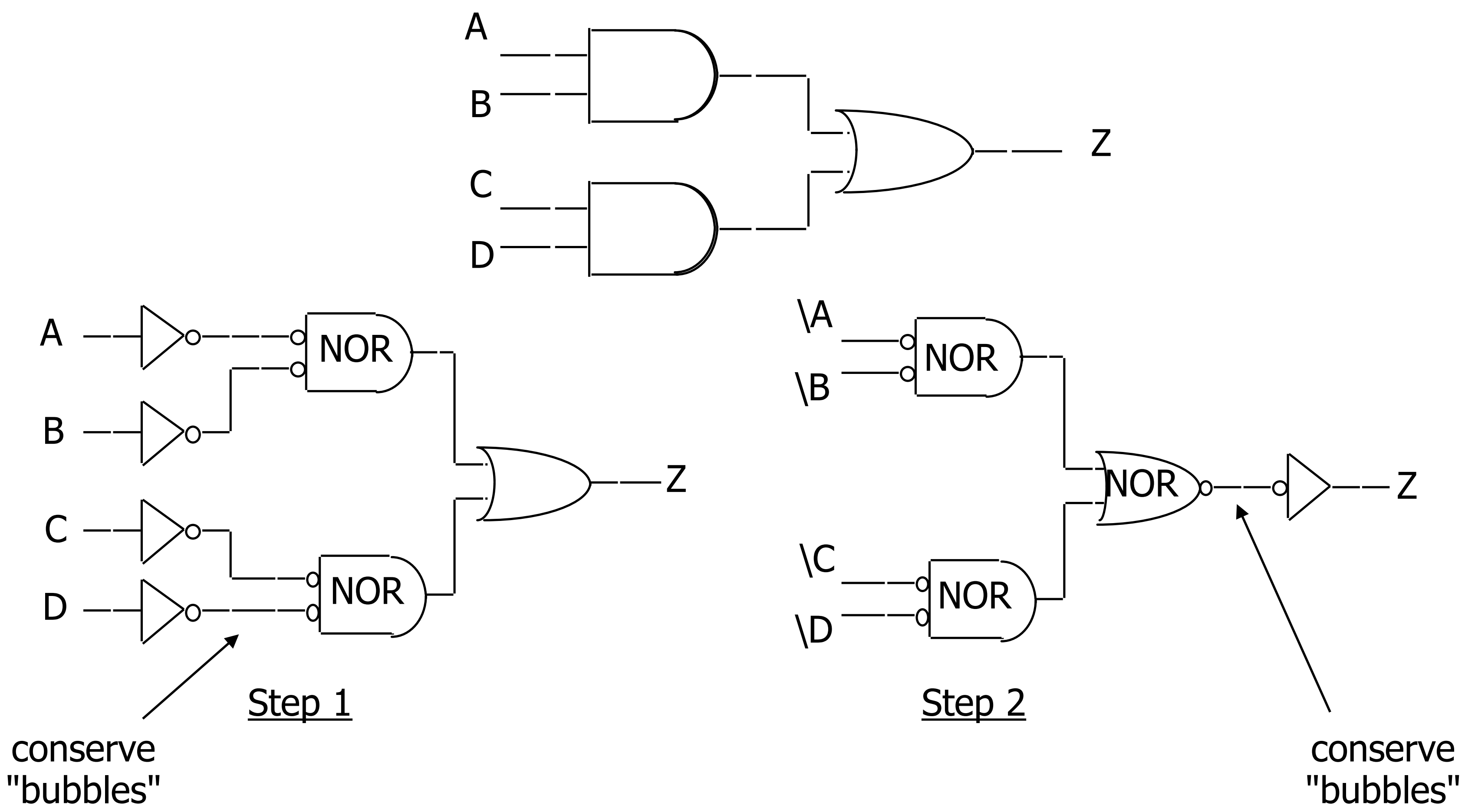




$$
\begin{aligned}
Z &= [\ (A \bullet B)' \bullet (C \bullet D)'\ ]' \\
&= [\ (A' + B') \bullet (C' + D')\ ]' \\
&= [\ (A' + B')' + (C' + D')'\ ] \\
&= (A \bullet B) + (C \bullet D) \quad \checkmark
\end{aligned}
$$

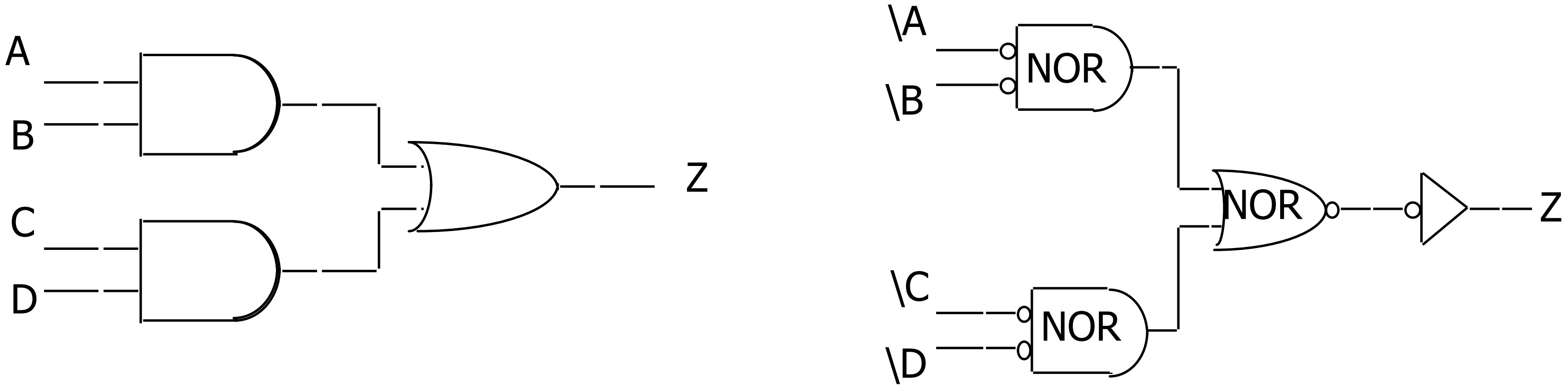
Conversion between forms (cont'd)

- Example: map AND/OR network to NOR/NOR network



Conversion between forms (cont'd)

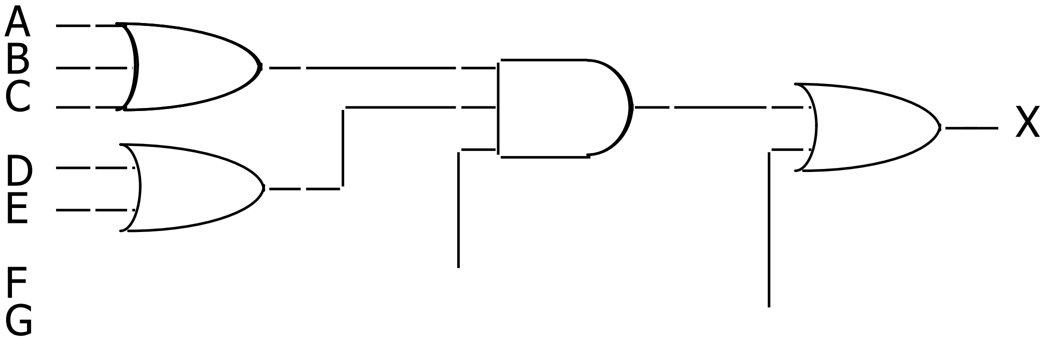
- Example: verify equivalence of two forms



$$
\begin{aligned}
Z &= \{\ [\ (A' + B')' + (C' + D')'\ ]'\ \}' \\
&= \{\ (A' + B') \bullet (C' + D')\ \}' \\
&= (A' + B')' + (C' + D')' \\
&= (A \bullet B) + (C \bullet D) \quad \checkmark
\end{aligned}
$$

Multi-level logic

- x = A D F + A E F + B D F + B E F + C D F + C E F + G
  - reduced sum-of-products form – already simplified
  - 6 x 3-input AND gates + 1 x 7-input OR gate (that may not even exist!)
  - 25 wires (19 literals plus 6 internal wires)
- x = (A + B + C) (D + E) F + G
  - factored form – not written as two-level S-o-P
  - 1 x 3-input OR gate, 2 x 2-input OR gates, 1 x 3-input AND gate
  - 10 wires (7 literals plus 3 internal wires)



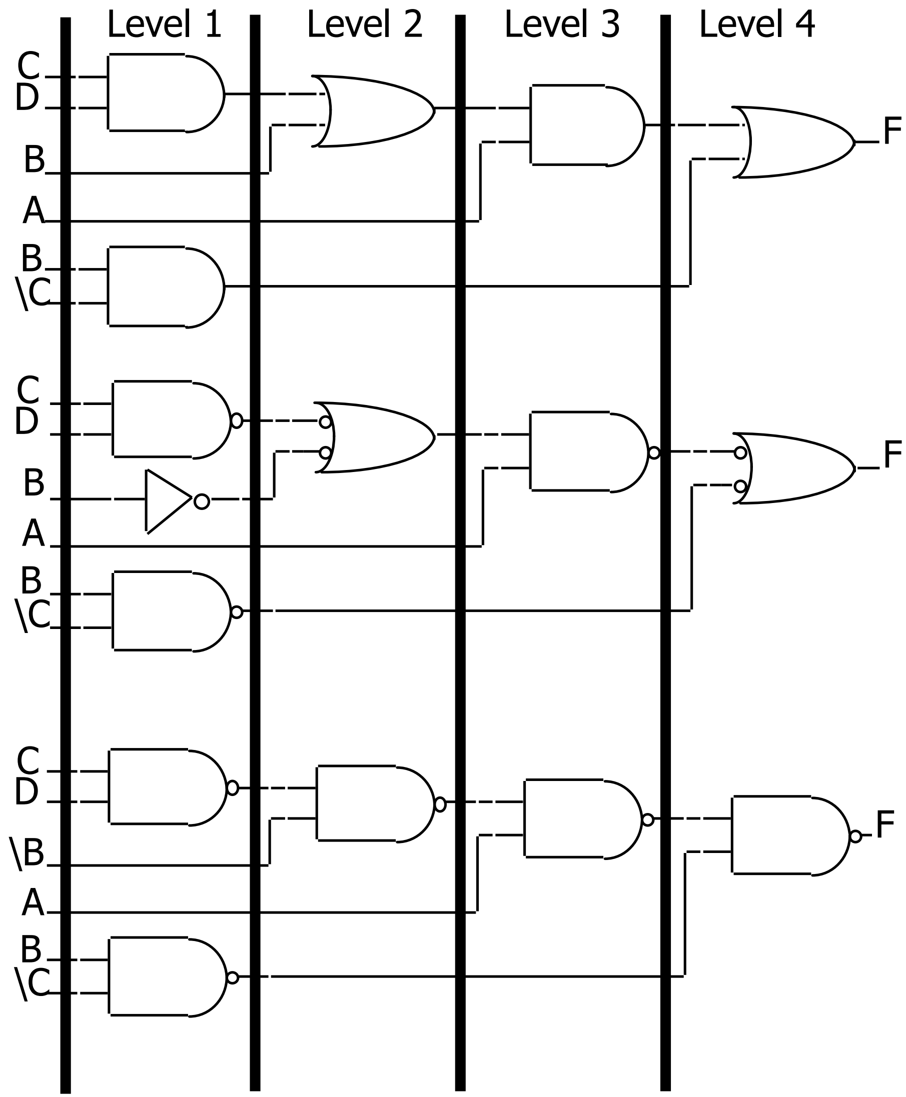
Conversion of multi-level logic to NAND gates

- $F = A\,(B + C\,D) + B\,C'$

original AND-OR network

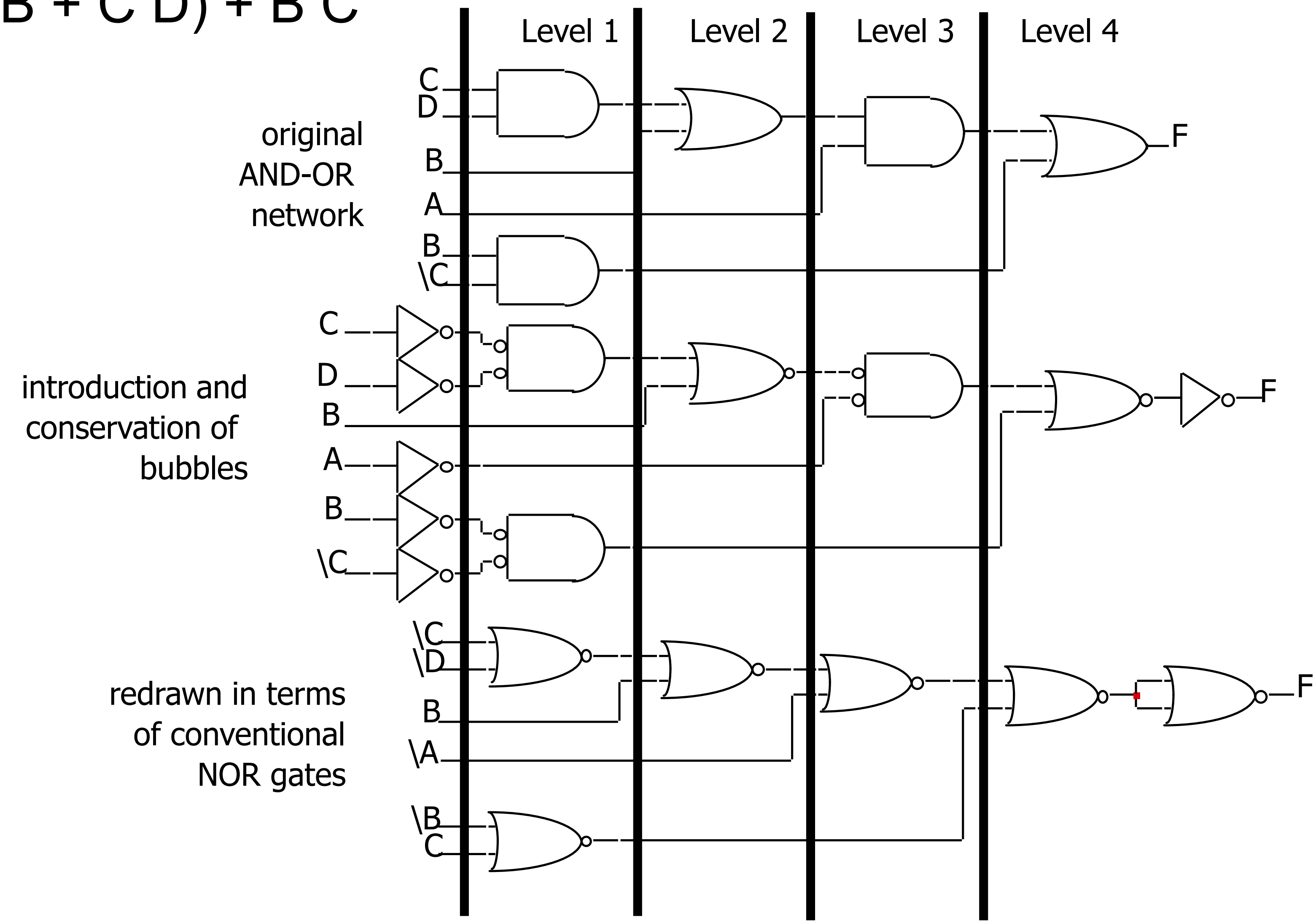
introduction and conservation of bubbles

redrawn in terms of conventional NAND gates



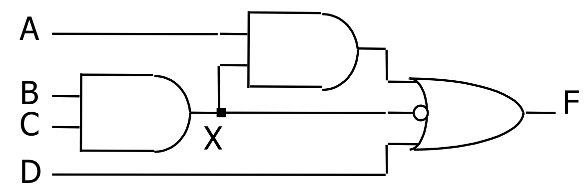
Conversion of multi-level logic to NORs

- $F = A (B + C D) + B C'$

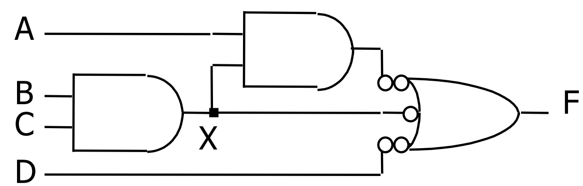


Conversion between forms

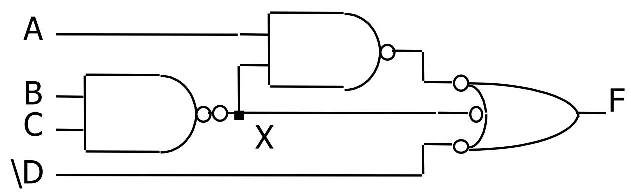
- Example



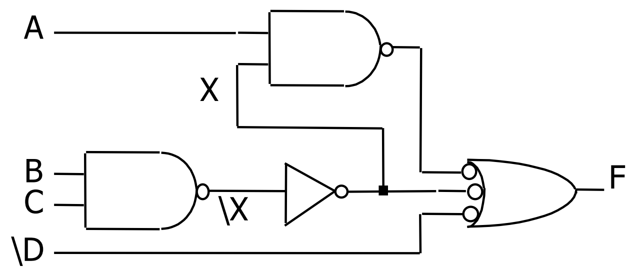
original circuit



add double bubbles to
invert all inputs of OR gate



add double bubbles to
invert output of AND gate



insert inverters to eliminate
double bubbles on a wire

Five-variable K-maps – f(a,b,c,d,e)

de

bc

d

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

c

b

e

a= 0

de

bc

d

m_{16}	m_{17}	m_{19}	m_8
m_{20}	m_{21}	m_{23}	m_{22}
m_{28}	m_{29}	m_{31}	m_{30}
m_{24}	m_{25}	m_{27}	m_{26}

c

b

e

a= 1

bc

de

b

m_0	m_4	m_{12}	m_8
m_1	m_5	m_{13}	m_9
m_3	m_7	m_{15}	m_{11}
m_2	m_6	m_{14}	m_{10}

e

d

c

a= 0

bc

de

b

m_{16}	m_{20}	m_{28}	m_{24}
m_{17}	m_{21}	m_{29}	m_{25}
m_{19}	m_{23}	m_{31}	m_{27}
m_{18}	m_{22}	m_{30}	m_{26}

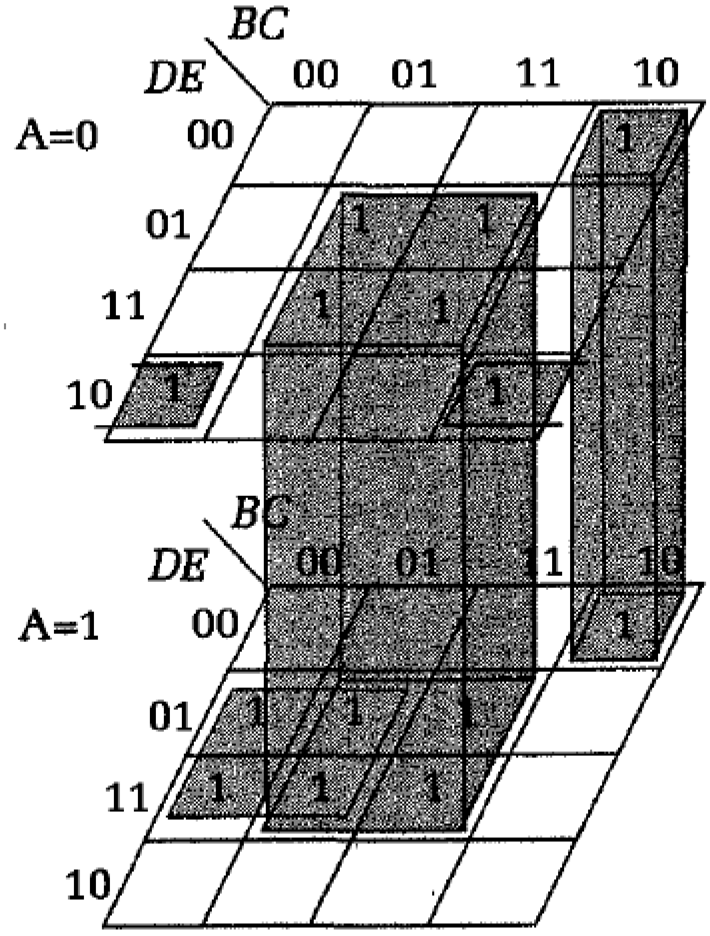
e

c

a= 1

Five-variable K-maps – f(A,B,C,D,E)

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



F=CE+AB'E +BC'D'E'+A'C'DE'

Simplify f(a,b,c,d,e)=Σm(0,1,4,5,6,11,12,14,16,20,22,28,30,31)

de, bc (d, c, b, e)

1	1	0	0
1	1	0	1
1	0	0	1
0	0	1	0

a= 0

de, bc (d, c, b, e)

1	0	0	0
1	0	0	1
1	0	1	1
0	0	0	0

a= 1

f = ce’

Σm(4,6,12,14,20,22,28,30)

+ a’b’d’ Σm(0,1,4,5)

+ b’d’e’ Σm(0,4,16,20)

+ abcd Σm(30,31)

+ a’bc’de m11

Simplify f(a,b,c,d,e)=Σm(0,1,4,5,6,11,12,14,16,20,22,28,30,31)

bc

de

1	1	1	0
1	1	0	0
0	0	0	1
0	1	1	0

A= 0

bc

de

1	1	1	0
0	0	0	0
0	0	1	0
0	1	1	0

A= 1

f = ce’

Σm(4,6,12,14,20,22,28,30)

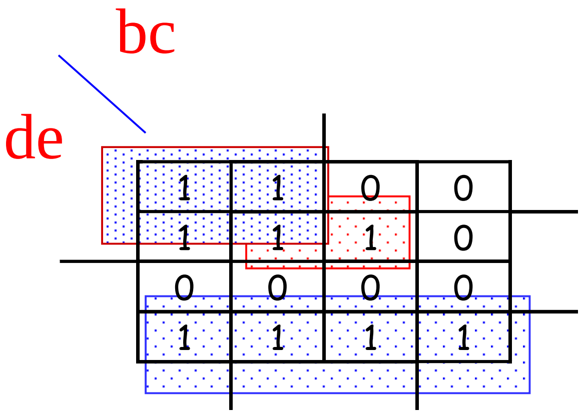
+ a’b’d’ Σm(0,1,4,5)

+ b’d’e’ Σm(0,4,16,20)

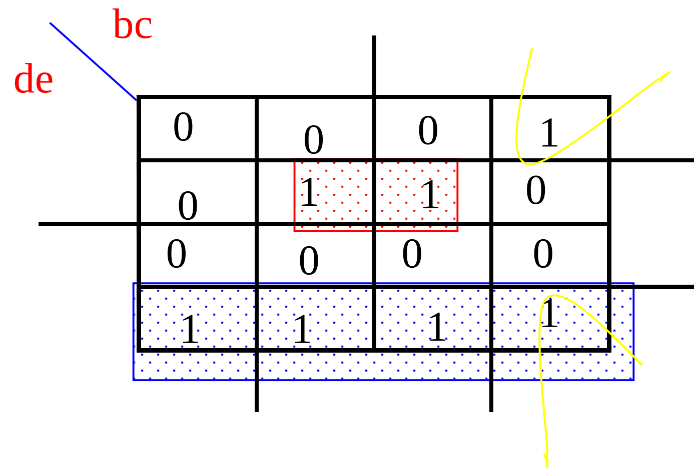
+ abcd Σm(30,31)

+ a’bc’de m11

Simplify f(a,b,c,d,e)=Σm(0,1,2,4,,5,6,10,13,,14,18,21,22,24,26,29,30)



a= 0



a= 1

f = de’

Σm(2,6,10,14,18,22,26,30)

+ a’b’d’ Σm(0,1,4,5)

+ cd’e Σm(5,13,21,29)

+ abc’d’ Σm(26, 24)

six-variable K-maps – f(a,b,c,d,e,f)

cd

ef

d

m_0	m_4	m_{12}	m_9
m_1	m_5	m_{13}	m_{10}
m_3	m_7	m_{15}	m_{12}
m_2	m_8	m_{13}	m_{11}

c

b

e

a= 0 b=0

cd

ef

d

m_{16}	m_{20}	m_{28}	m_{24}
m_{17}	m_{21}	m_{29}	m_{25}
m_{19}	m_{23}	m_{31}	m_{27}
m_{18}	m_{22}	m_{30}	m_{26}

c

b

e

a= 0 b=1

cd

ef

m_{48}	m_{52}	m_{60}	m_{56}
m_{49}	m_{53}	m_{61}	m_{57}
m_{51}	m_{55}	M_{63}	M_{59}
m_{50}	m_{54}	m_{62}	m_{58}

e

c

a= 1 b=1

cd

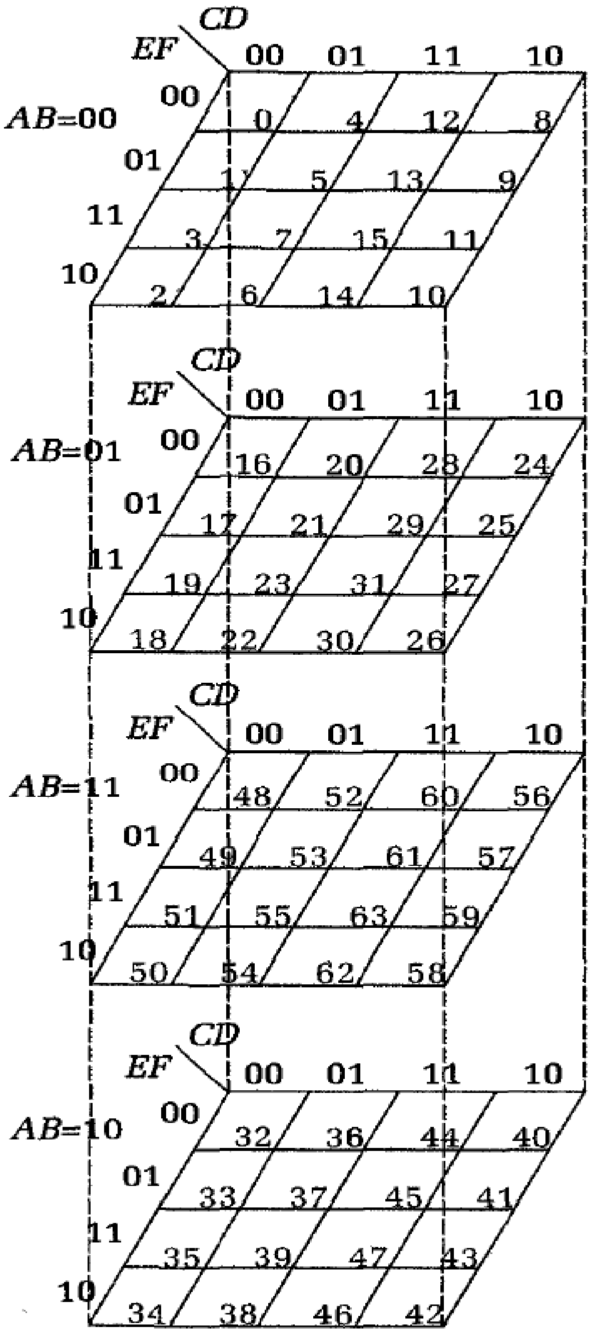
ef

m_{32}	m_{36}	m_{44}	m_{40}
m_{33}	m_{37}	m_{45}	m_{41}
m_{35}	m_{39}	M_{47}	m_{43}
M_{34}	m_{38}	m_{46}	m_{42}

c

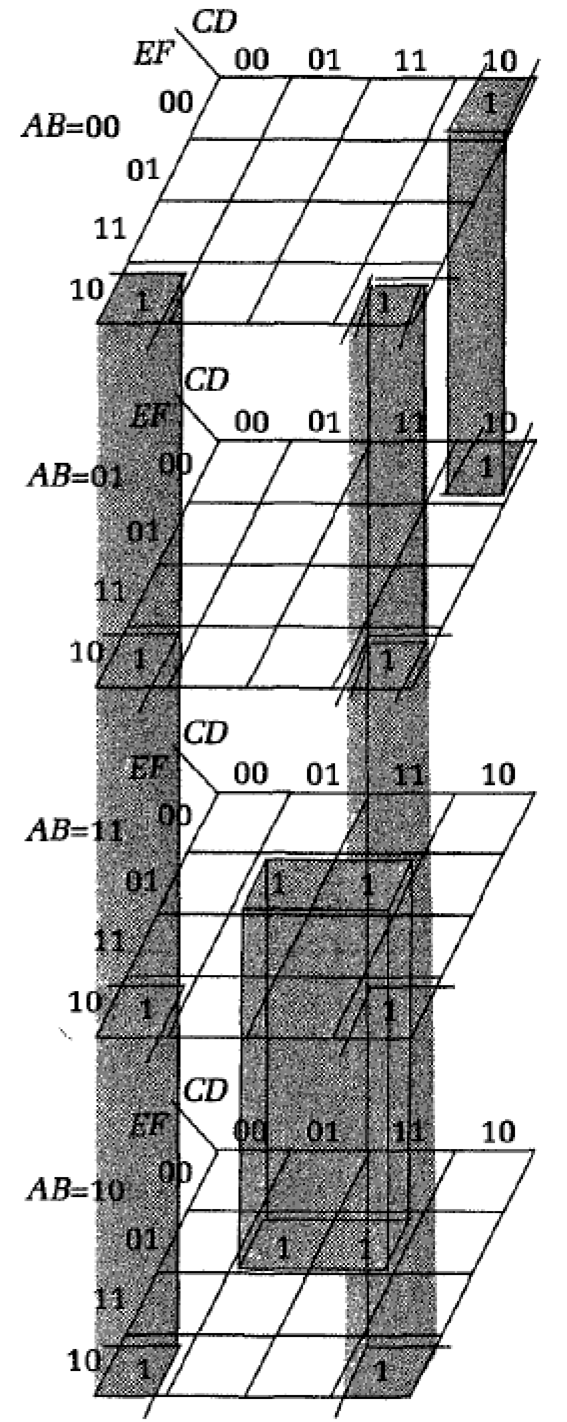
a= 1 b=0

Six-variable K-maps f(A,B,C,D,E)



Six-variable K-maps – f(A,B,C,D,E)

$$F(A,B,C,D,E,F) = \sum m(2,8,10,18,24,26,34,37,42,45,50,53,58,61)$$

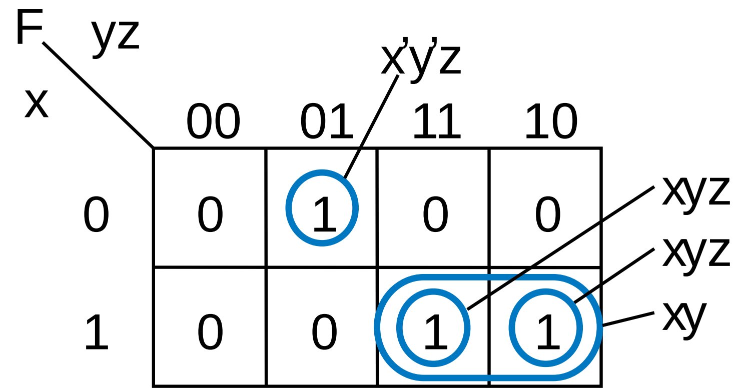


F= D'EF'+ADE'F+ A'CD'F'

Automated Two-Level Logic Size Optimization

Basic Concepts Underlying Automated Two-Level Logic Size Optimization

- Definitions
  - ***On-set***: All minterms that define when F=1
  - ***Off-set***: All minterms that define when F=0
  - ***Implicant***: Any product term (minterm or other) that when 1 causes F=1
    - On K-map, any legal (but not necessarily largest) circle
    - Cover: Implicant xy ***covers*** minterms xyz and xyz'
  - ***Expanding*** a term: removing a variable (like larger K-map circle)
    - xyz → xy is an expansion of xyz



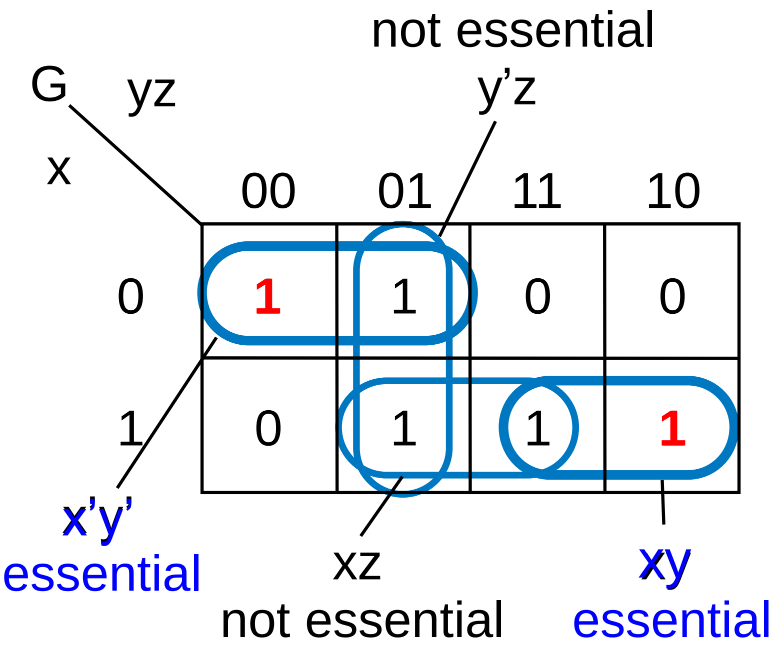
4 implicants of F

*Note: We use K-maps here just for intuitive illustration of concepts; automated tools do **not** use K-maps.*

- ***Prime implicant***: Maximally expanded implicant – any expansion would cover 1s not in on-set
  - x'y'z, and xy, above
  - But not xyz or xyz' – they can be expanded

Basic Concepts Underlying Automated Two-Level Logic Size Optimization

- Definitions (cont)
  - ***Essential prime implicant***: The only prime implicant that covers a particular minterm in a function's on-set
    - Importance: We ***must*** include ***all*** essential PIs in a function's cover
    - In contrast, some, but not all, non-essential PIs will be included



Automated Two-Level Logic Size Optimization Method

TABLE 6.1 Automatable tabular method for two-level logic size optimization.

	Step	Description
1	*Determine prime implicants*	Starting with minterm implicants, methodically compare all pairs (actually, all pairs whose numbers of uncomplemented literals differ by one) to find opportunities to combine terms to eliminate a variable, yielding new implicants with one less literal. Repeat for new implicants. Stop when no implicants can be combined. All implicants not covered by a new implicant are prime implicants.
2	*Add essential prime implicants to the function's cover*	Find every minterm covered by only one prime implicant, and denote that prime implicant as essential. Add essential prime implicants to the cover, and mark all minterms covered by those implicants as already covered.
3	*Cover remaining minterms with nonessential prime implicants*	Cover the remaining minterms using the minimal number of remaining prime implicants.

- Steps 1 and 2 are exact
- Step 3: Hard. Checking all possibilities: exact, but computationally expensive. Checking some but not all: heuristic.

Tabulation Method (Quine-McCluskey)

- STEP 1:
  - Convert Minterm List (specifying F) to Prime Implicant List
- STEP 2:
  - Choose All Essential Prime Implicants
- STEP 3:
  - Construct Cover Table

Tabulation Method (Quine-McCluskey)

1. Partition Prime Implicants (or minterms) According to Number of 1's

2. Check Adjacent Classes for Cube Merging Building a New List

3. If Entry in New List Covers Entry in Current List – Disregard Current List Entry

4. If Current List = New List
   HALT
   Else
   Current List $\leftarrow$ New List
   New List $\leftarrow$ NULL
   Go To Step 1

STEP 1 - EXAMPLE

$$F = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum (0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
8	1	0	0	0
3	0	0	1	1
5	0	1	0	1
10	1	0	1	0
11	1	0	1	1
13	1	1	0	1
15	1	1	1	1

STEP 1 - EXAMPLE

$$F = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum(0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube			
0,1	0	0	0	-
0,2	0	0	-	0
0,8	-	0	0	0
1,3	0	0	-	1
1,5	0	-	0	1
2,3	0	0	1	-
2,10	-	0	1	0
8,10	1	0	-	0
3,11	-	0	1	1
5,13	-	1	0	1
10,11	1	0	1	-
11,15	1	-	1	1
13,15	1	1	-	1

STEP 1 - EXAMPLE

$$F = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum(0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube				
0,1	0	0	0	-	✓
0,2	0	0	-	0	✓
0,8	-	0	0	0	✓
1,3	0	0	-	1	✓
1,5	0	-	0	1	
2,3	0	0	1	-	✓
2,10	-	0	1	0	✓
8,10	1	0	-	0	✓
3,11	-	0	1	1	✓
5,13	-	1	0	1	
10,11	1	0	1	-	✓
11,15	1	-	1	1	
13,15	1	1	-	1	

Minterm	Cube			
0,1,2,3	0	0	-	-
0,8,2,10	-	0	-	0
2,3,10,11	-	0	1	-

STEP 1 - EXAMPLE

$F = \{m_0, m_1, m_2, m_3, m_5, m_8, m_{10}, m_{11}, m_{13}, m_{15}\} = \sum(0, 1, 2, 3, 5, 8, 10, 11, 13, 15)$

Minterm	Cube				
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Minterm	Cube				
0,1	0	0	0	-	✓
0,2	0	0	-	0	✓
0,8	-	0	0	0	✓
1,3	0	0	-	1	✓
1,5	0	-	0	1	PI
2,3	0	0	1	-	✓
2,10	-	0	1	0	✓
8,10	1	0	-	0	✓
3,11	-	0	1	1	✓
5,13	-	1	0	1	PI
10,11	1	0	1	-	✓
11,15	1	-	1	1	PI
13,15	1	1	-	1	PI

Minterm	Cube			
0,1,2,3	0	0	-	-
0,8,2,10	-	0	-	0
2,3,10,11	-	0	1	-

PI's = {00--, -01-, -0-0, 0-01, -101, 1-11, 11-1}

PI's ={A'B', B'C, B'D', A'C'D,BC'D, ACD, ABD}

STEP 2 – Construct Cover Table

- PIs Along Vertical Axis
- Minterms Along Horizontal Axis

	0	1	2	3	5	8	10	11	13	15
A' B'	x	x	x	x						
B'C			x	x			x	x		
B'D'	x		x			x	x			
A'C'D		x			x					
BC'D					x				x	
ACD								x		x
ABD									x	x

F={A'B',+ B'D' +CB'+ BC'D+ABD}

Problem with Methods that Enumerate all Minterms or Compute all Prime Implicants

- Too many minterms for functions with many variables
  - Function with 32 variables:
    - 2^{32} = 4 billion possible minterms.
    - Too much compute time/memory
- Too many computations to generate all prime implicants
  - Comparing every minterm with every other minterm, for 32 variables, is (4 billion)^2 = 1 quadrillion computations
  - Functions with many variables could requires days, months, years, or more of computation – unreasonable

Solution to Computation Problem

- Solution
  - Don't generate all minterms or prime implicants
  - Instead, just take input equation, and try to "iteratively" improve it
  - Ex: F = abcdefgh + abcdefgh'+ jklmnop
    - Note: 15 variables, may have thousands of minterms
    - But can minimize just by combining first two terms:
      - F = abcdefg(h+h') + jklmnop = abcdefg + jklmnop

Summary for multi-level logic

- Advantages
  - circuits may be smaller
  - gates have smaller fan-in
  - circuits may be faster
- Disadvantages
  - more difficult to design
  - tools for optimization are not as good as for two-level
  - analysis is more complex

8bit mult-sample design