CEDT – Digital Logic 2023
Day 2
Two Level Simplification
Logic Realization
Non-gate logics

Working with combinational logic

- Simplification
  - two-level simplification
  - exploiting don't cares
  - algorithm for simplification
- Logic realization
  - two-level logic and canonical forms realized with NANDs and NORs
  - multi-level logic, converting between ANDs and ORs
- Time behavior
- Hardware description languages

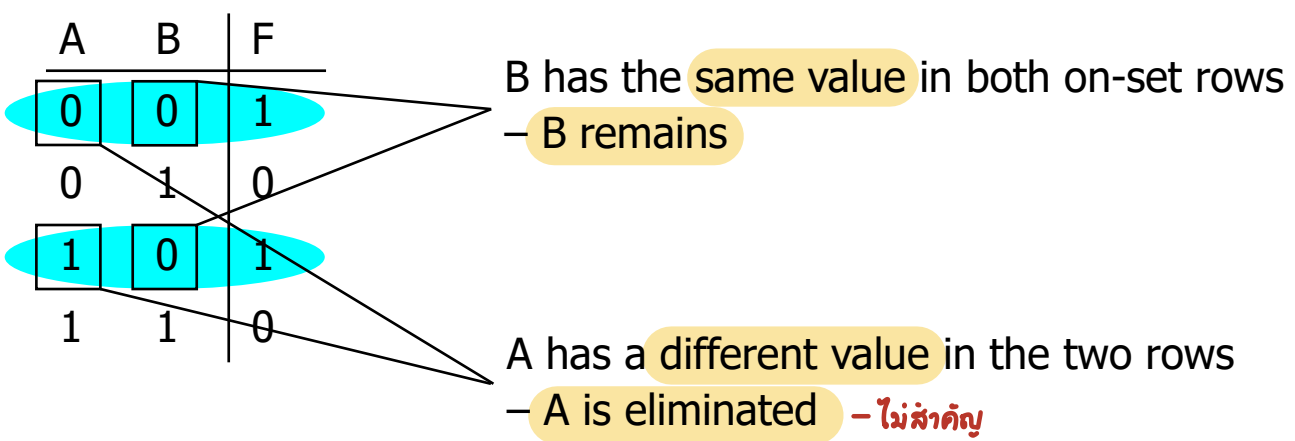
Simplification of two-level combinational logic

- 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

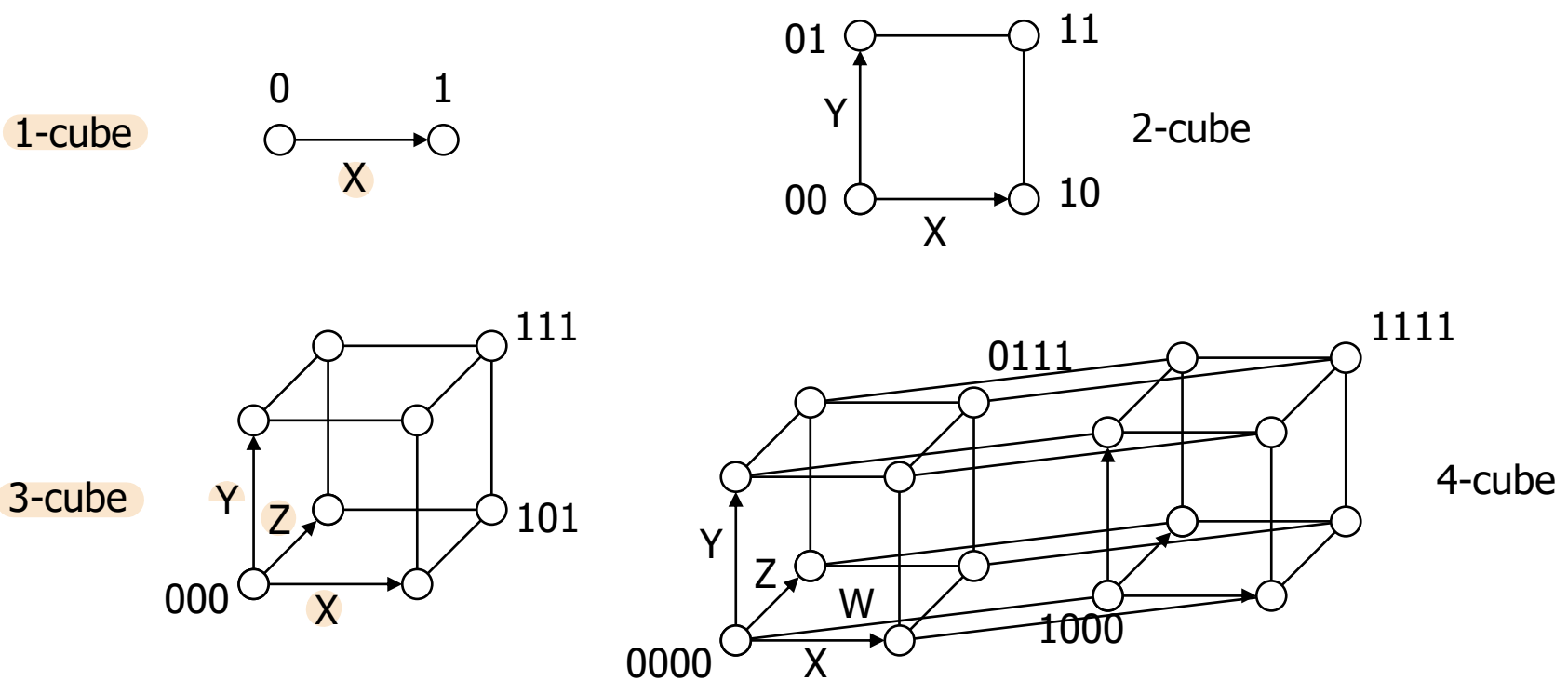
- Key tool to simplification: A (B' + B) = A *
- Essence of simplification of two-level logic
  - find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

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



Boolean cubes

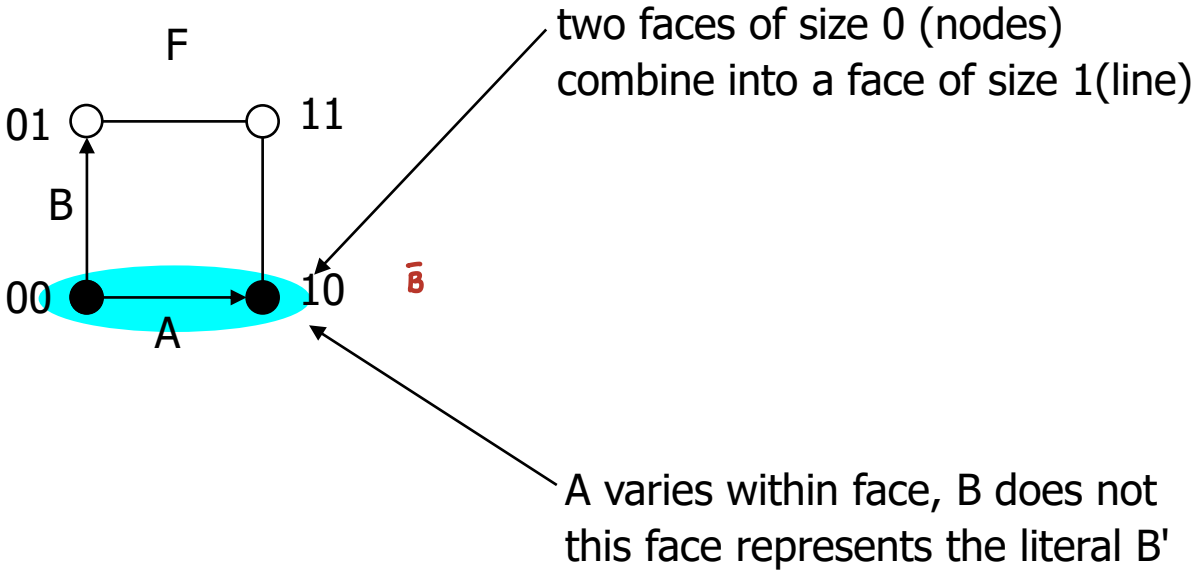
- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"



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

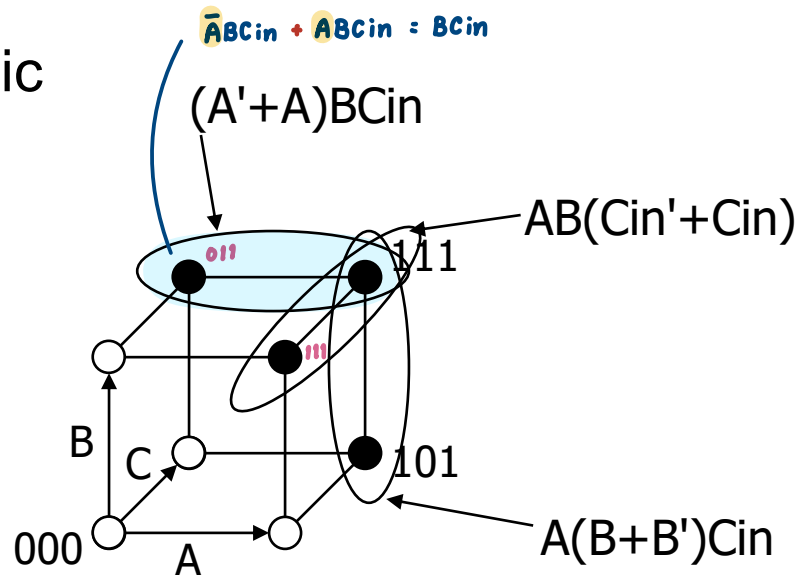


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

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

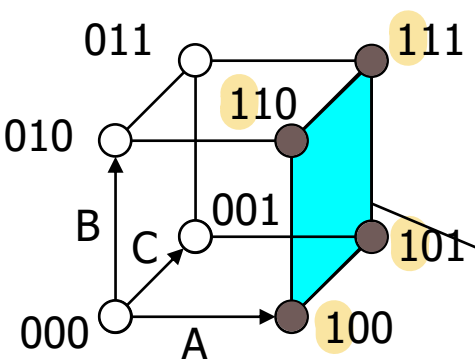


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

Cout = BCin+AB+ACin 2 literal

Higher dimensional cubes

- Sub-cubes of higher dimension than 2



F(A,B,C) = Σm(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

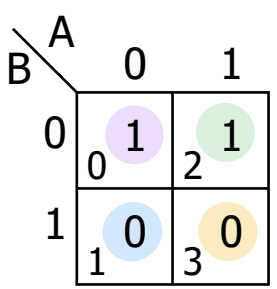
This subcube represents the literal A

m-dimensional cubes in a 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

Karnaugh maps

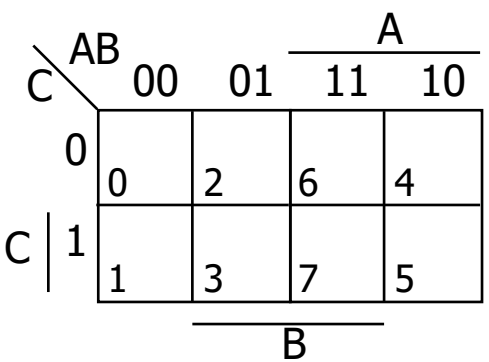
- Flat map of Boolean cube
  - wrap–around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - guide to applying the uniting theorem
  - on-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

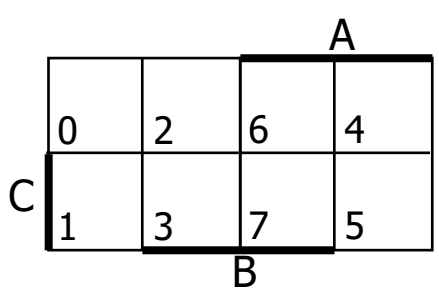


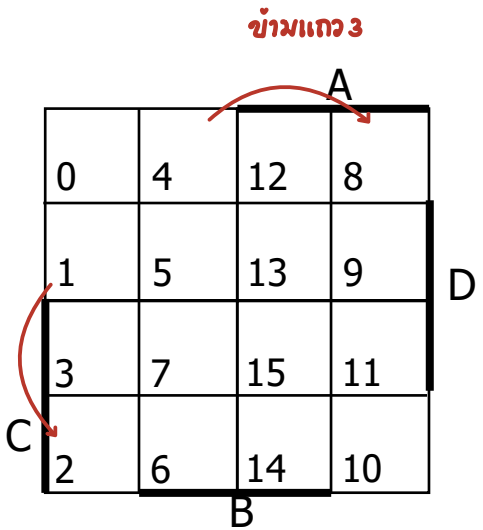
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

Karnaugh maps (cont'd)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - only a single bit changes in code for adjacent map cells





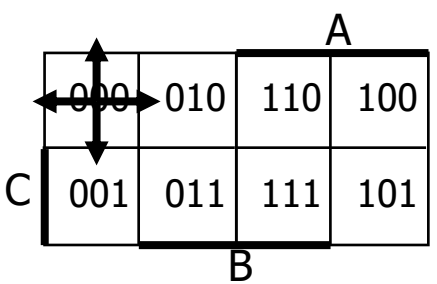


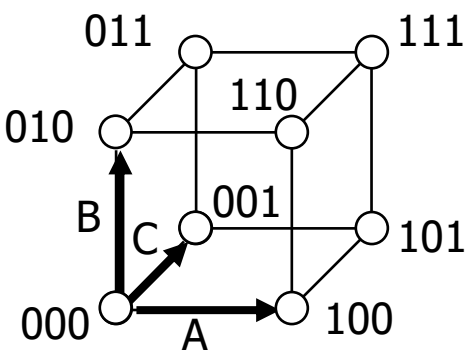
13 = 1101= ABC'D

Adjacencies in Karnaugh maps

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

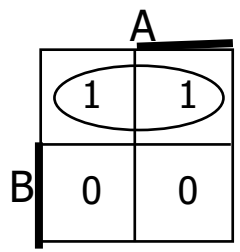
อยู่ข้างกันใน K-map = ข้างกันใน Boolean Cube



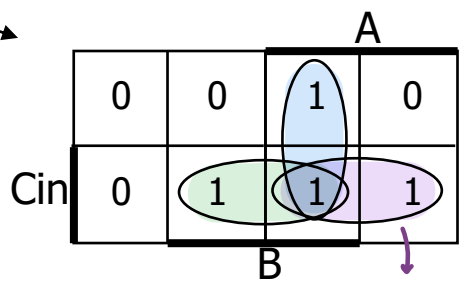


Karnaugh map examples

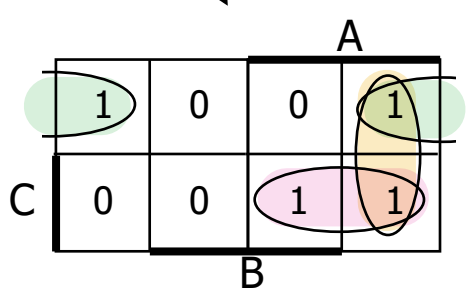
- F =
- Cout =
- f(A,B,C) = Σm(0,4,5,7)



B′



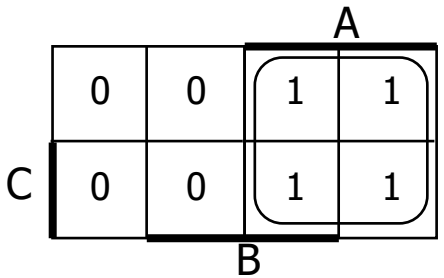
AB + ACin + BCin



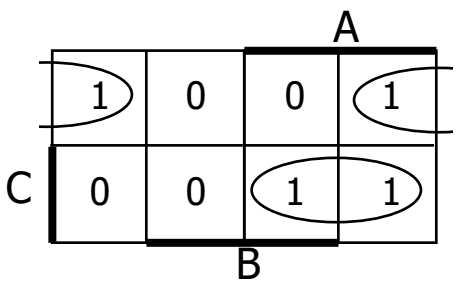
AC + B′C′ + ~~AB′~~

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

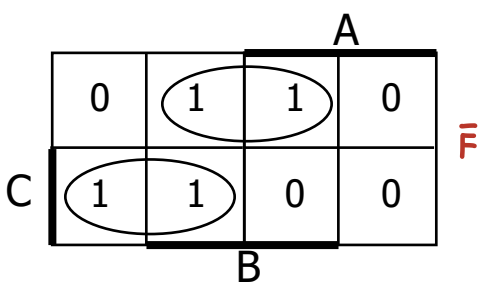
More Karnaugh map examples



G(A,B,C) = A



F(A,B,C) = Σm(0,4,5,7) = AC + B'C'



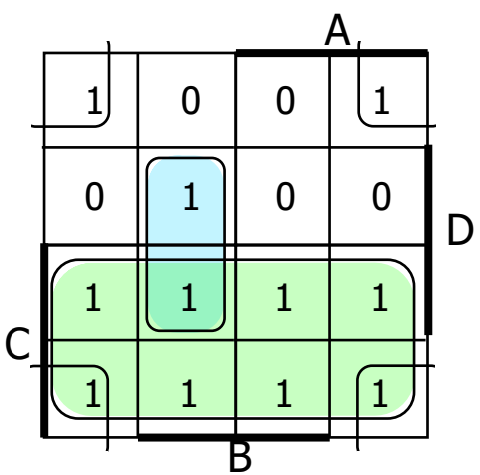
$\bar{F}$

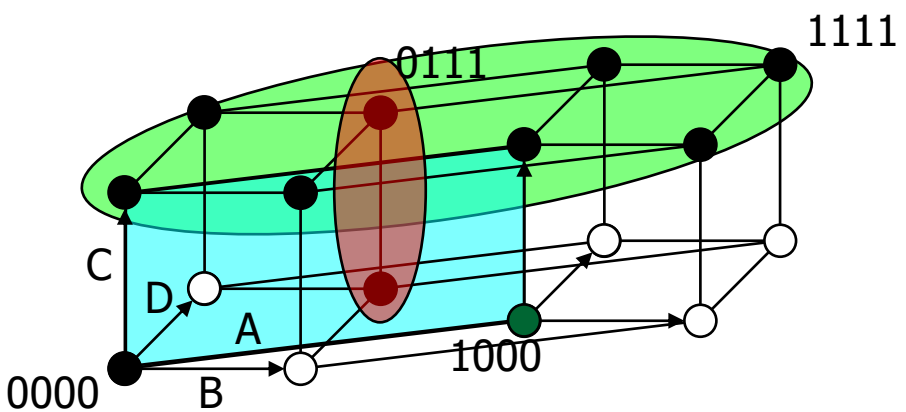
F' simply replace 1's with 0's and vice versa
F'(A,B,C) = Σ m(1,2,3,6)= BC' + A'C

Karnaugh map: 4-variable example

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

  F = C + A′BD + B′D′

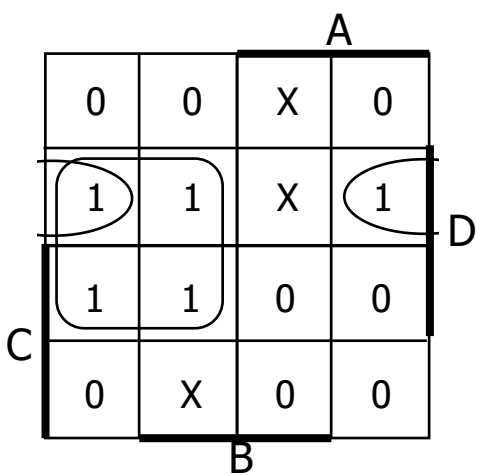




find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)

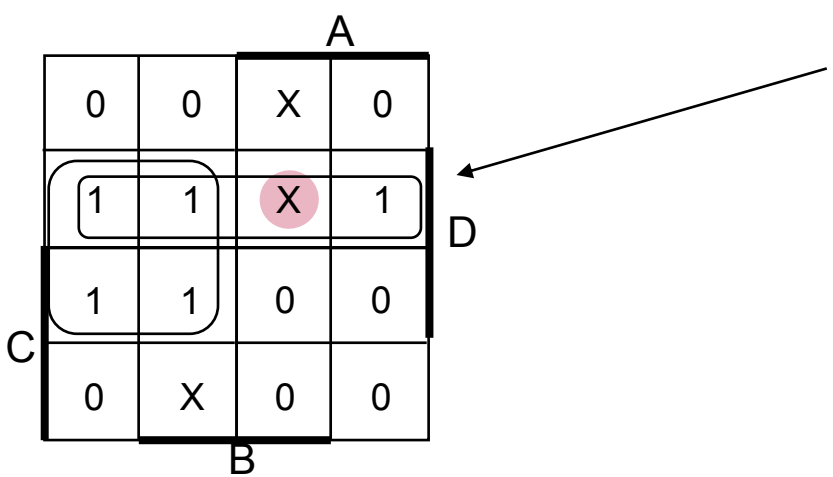
Karnaugh maps: don't cares

- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - without don't cares
    - f = A′D + B′C′D



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

- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - f = A'D + B'C'D without don't cares
  - f = A'D + C'D with don't cares

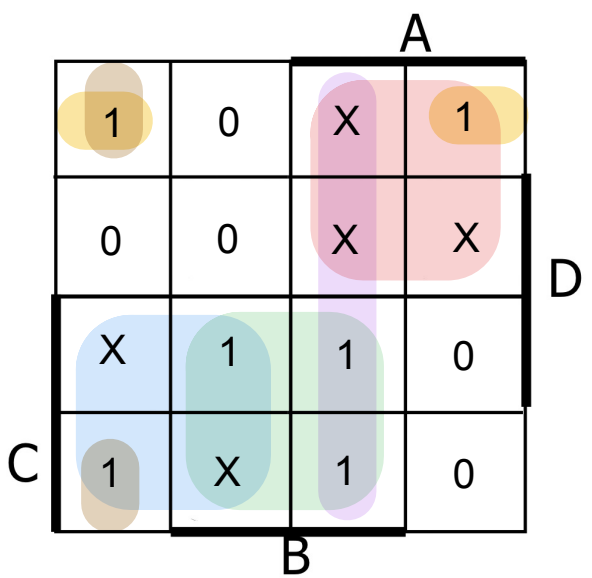


by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

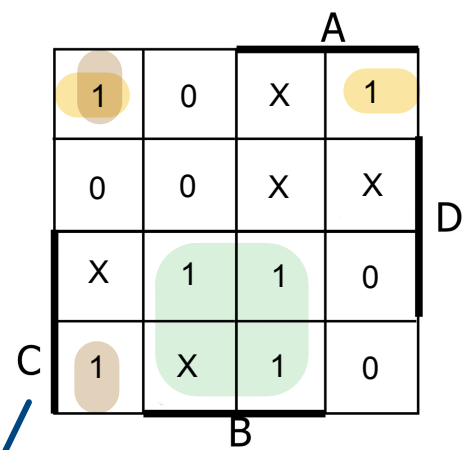
don't cares can be treated as
1s or 0s
depending on which is more
advantageous

Activity

- Minimize the function F = Σ m(0, 2, 7, 8, 14, 15) + d(3, 6, 9, 12, 13)



F = ~~AC'~~ + ~~A'C~~ + BC + ~~AB~~ + A'B'D' + B'C'D'

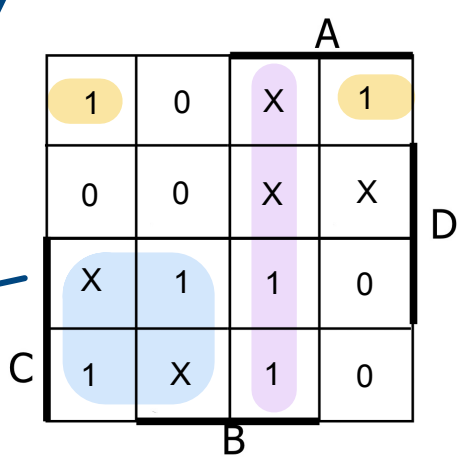


3 literals

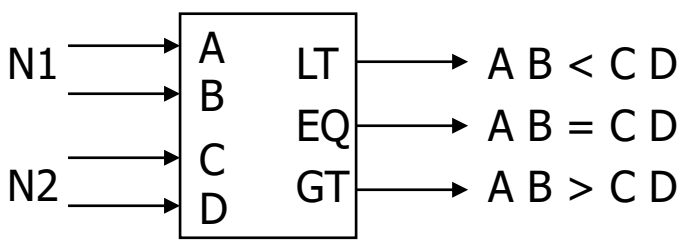
F = BC + A'B'D' + B'C'D'

ย่อกว่า F = A'C + AB + B'C'D'

3 literals



Design example: two-bit comparator

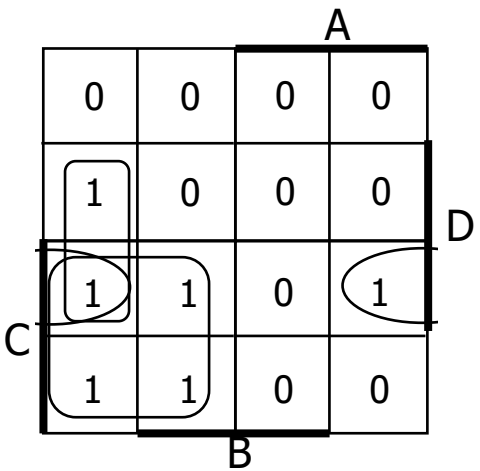


block diagram
and
truth table

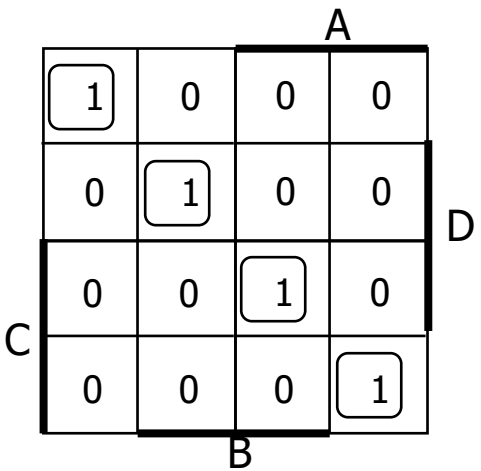
A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

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

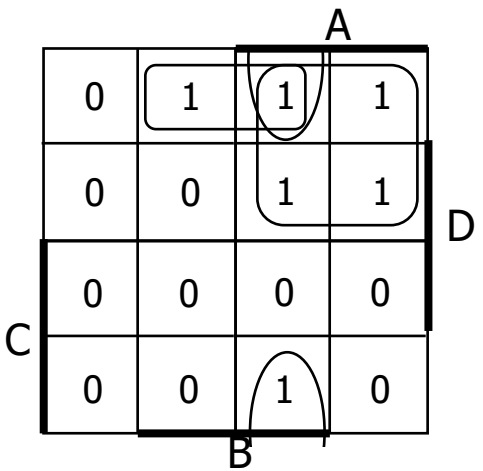
Design example: two-bit comparator (cont'd)



K-map for LT



K-map for EQ



K-map for GT

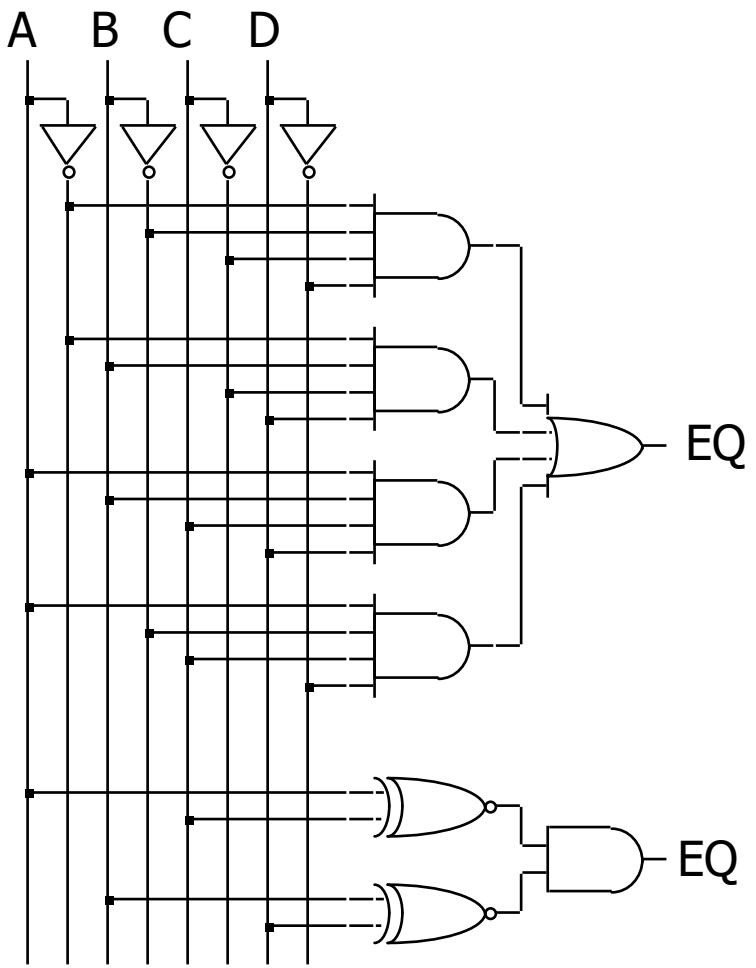
LT = A' B' D + A' C + B' C D

EQ = A' B' C' D' + A' B C' D + A B C D + A B' C D' = (A xnor C) • (B xnor D)

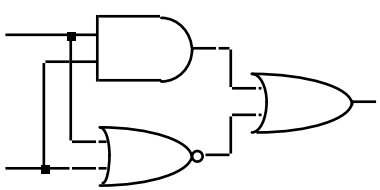
GT = B C' D' + A C' + A B D'

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

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

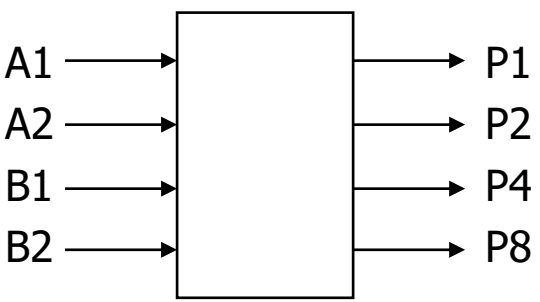


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier

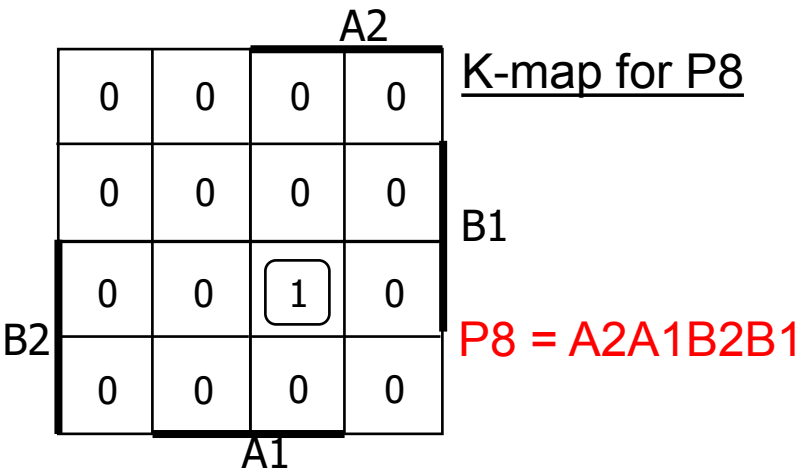


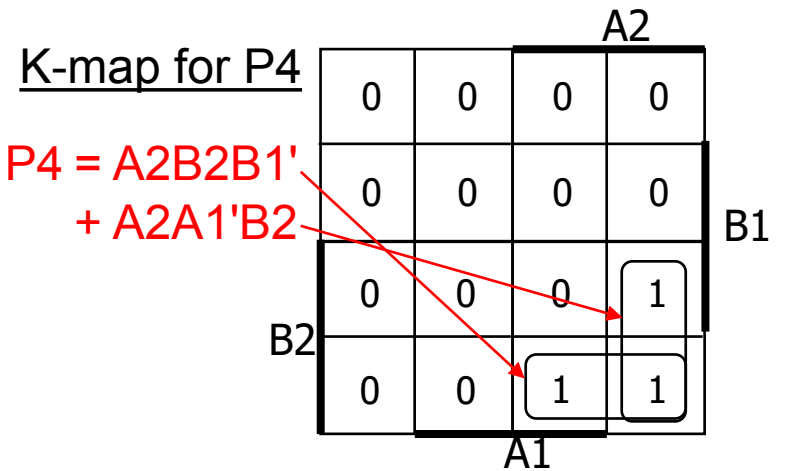
block diagram
and
truth table

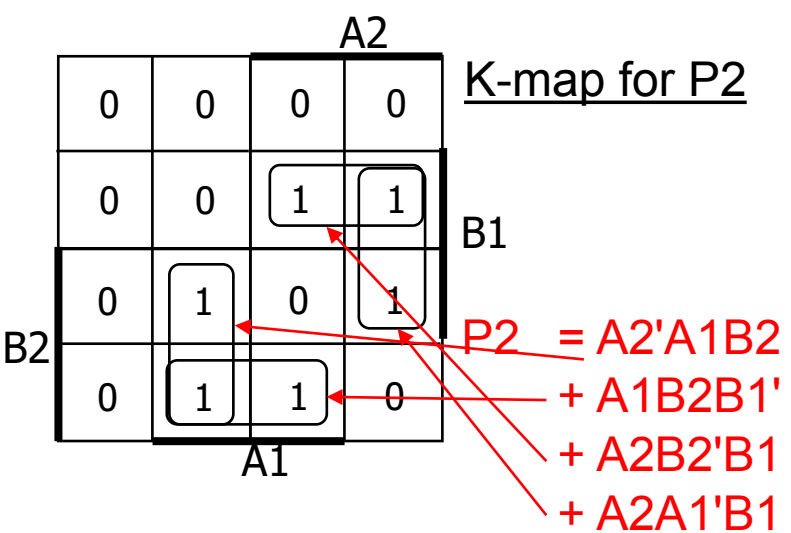
A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

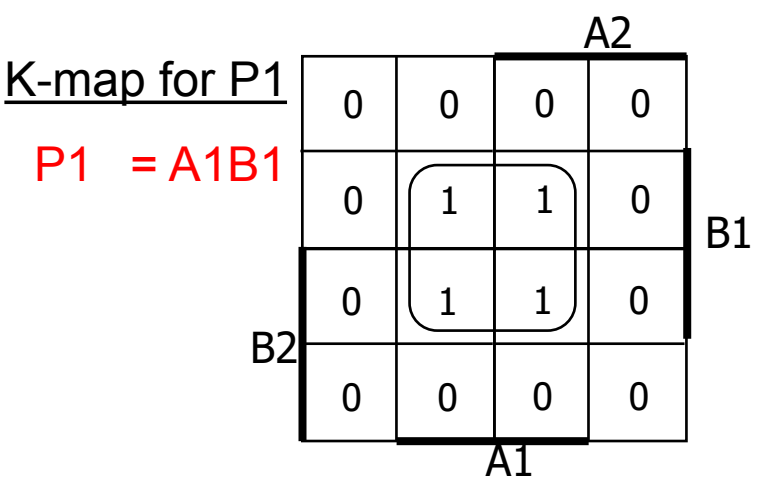
4-variable K-map
for each of the 4
output functions

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



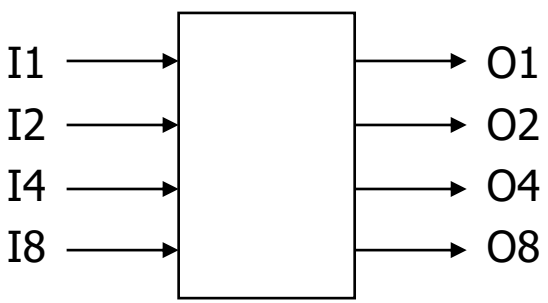






Design example: BCD increment by 1

(Binary-coded decimal)



block diagram
and
truth table

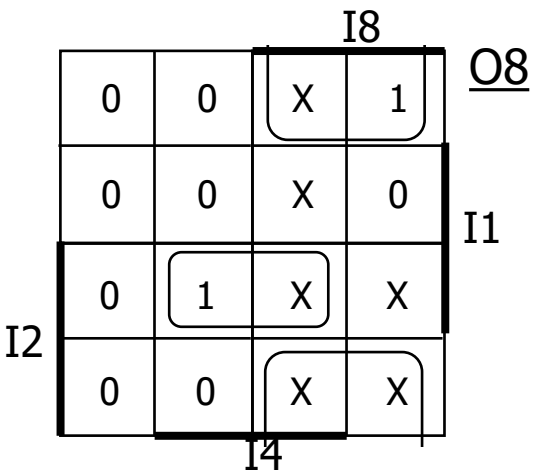
+1

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

0-9 ที่เหลือ don't care

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

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

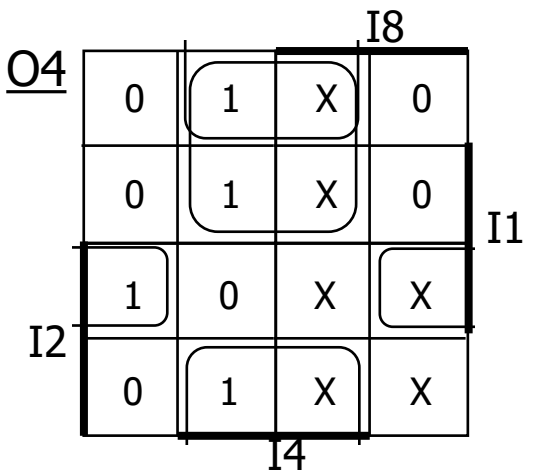


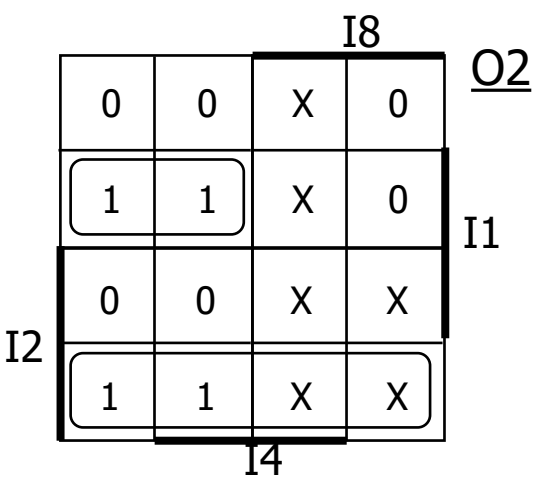
O8 = I4 I2 I1 + I8 I1'

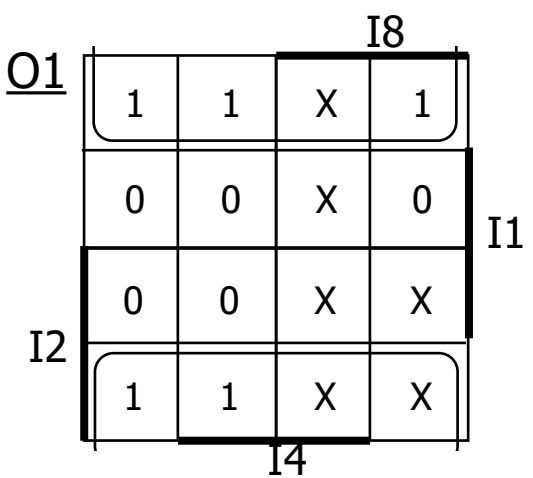
O4 = I4 I2' + I4 I1' + I4' I2 I1

O2 = I8' I2' I1 + I2 I1'

O1 = I1'



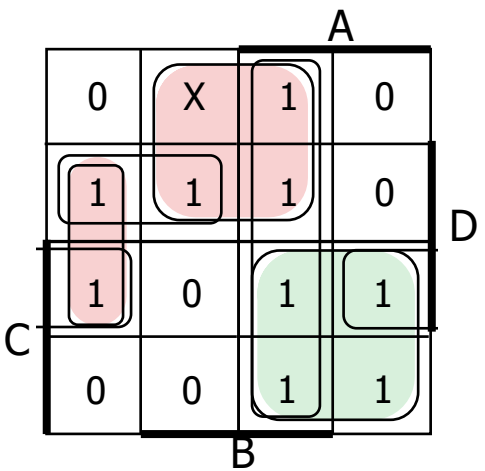




Definition of terms for two-level simplification

- 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

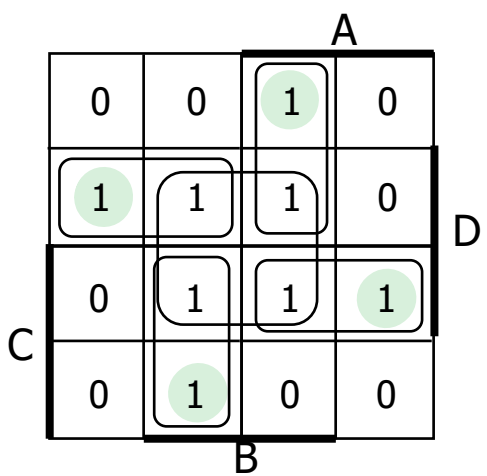
minimum cover: AC + BC' + A'B'D

5 prime implicants:

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

essential

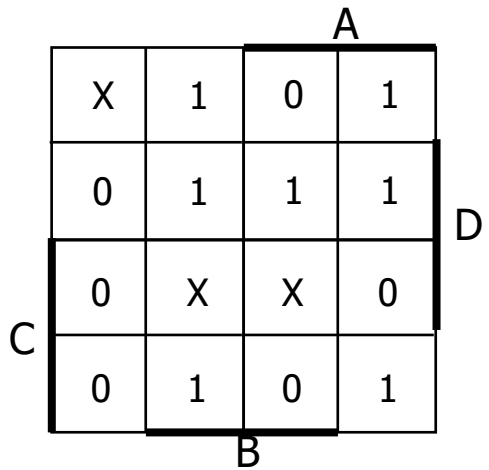
minimum cover: 4 essential implicants

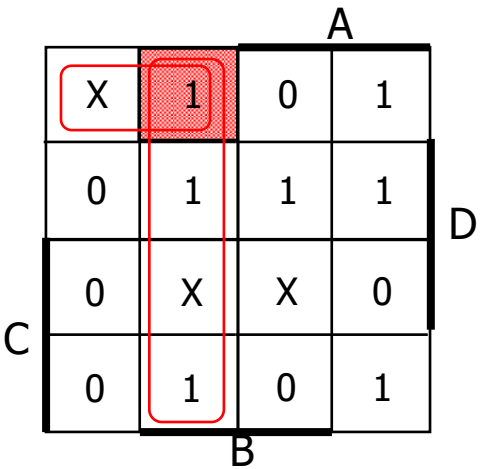


Algorithm for two-level simplification

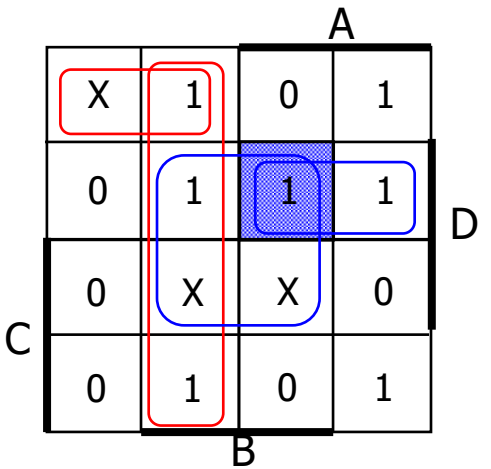
- Algorithm: minimum sum-of-products expression from a Karnaugh map
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number 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)

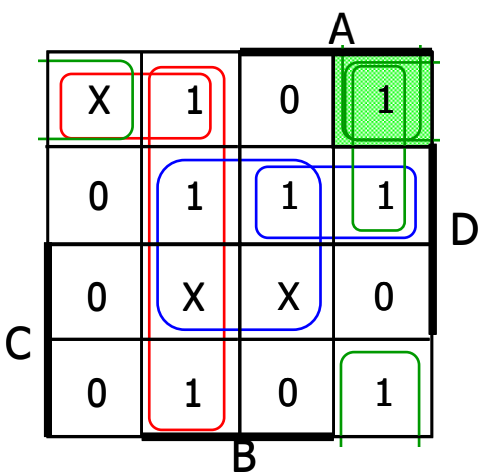




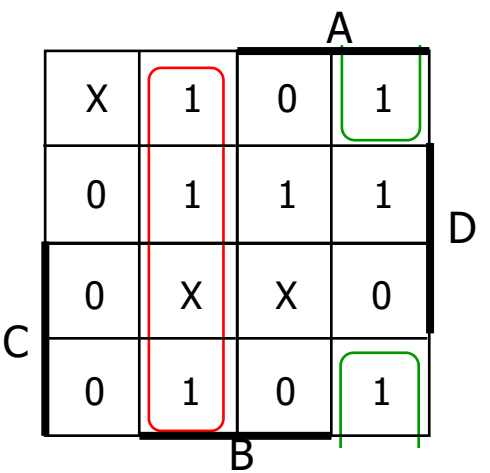
2 primes around A'BC'D'



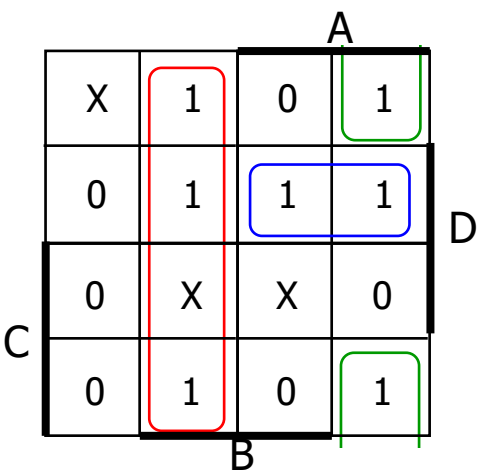
2 primes around ABC'D



3 primes around AB'C'D'



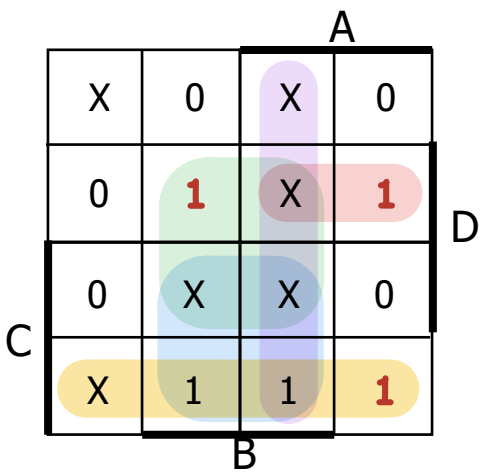
2 essential primes



minimum cover (3 primes)

Activity

- List all prime implicants for the following K-map:



✓ ✓ ✓
CD' BC BD AB AC'D

- Which are essential prime implicants? CD' BD AC'D
- What is the minimum cover? CD' BD AC'D

Quine-McCluskey Method * ไม่ออกสอบ ตาลาย ไอกาสท่าผิดสูง

Tabular method to systematically find all prime implicants

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

Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

Implication Table

จำนวนเลข 1	Column I	
0	0000 0	
1	0100 4	
	1000 8	
2	0101 5	
	0110 6	
	1001 9	
	1010 10	
3	0111 7	
	1101 13	
4	1111 15	

Quine-McCluskey Method

Tabular method to systematically find all prime implicants

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

Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

Step 2: Apply Uniting Theorem— Compare elements of group w/ N 1's against those with N+1 1's. Differ by one bit implies adjacent. Eliminate variable and place in next column.

E.g., 0000 vs. 0100 yields 0-00
0000 vs. 1000 yields -000

When used in a combination, mark with a check. If cannot be combined, mark with a star. These are the prime implicants.

Repeat until no further combinations can be made.

Implication Table

Column I	Column II
0000	0-00
	-000
0100	
1000	010-
	01-0
0101	100-
0110	10-0
1001	
1010	01-1
	-101
0111	011-
1101	1-01
1111	-111
	11-1

Quine-McCluskey Method

Tabular method to systematically find all prime implicants

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

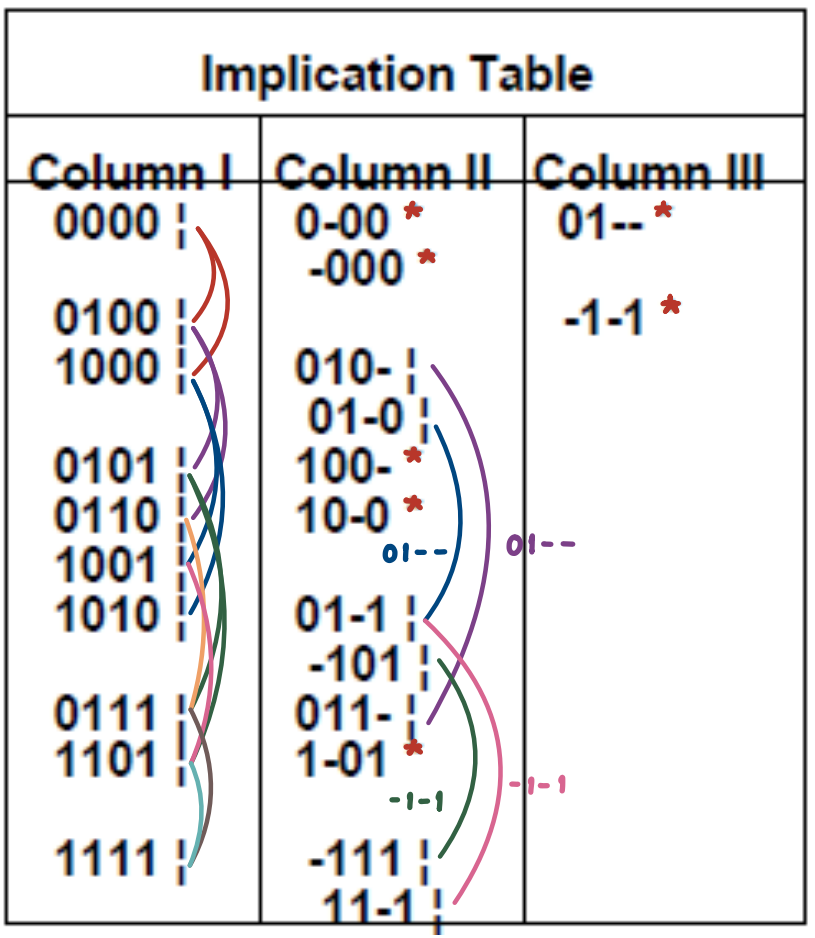
Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

Step 2: Apply Uniting Theorem— Compare elements of group w/ N 1's against those with N+1 1's. Differ by one bit implies adjacent. Eliminate variable and place in next column.

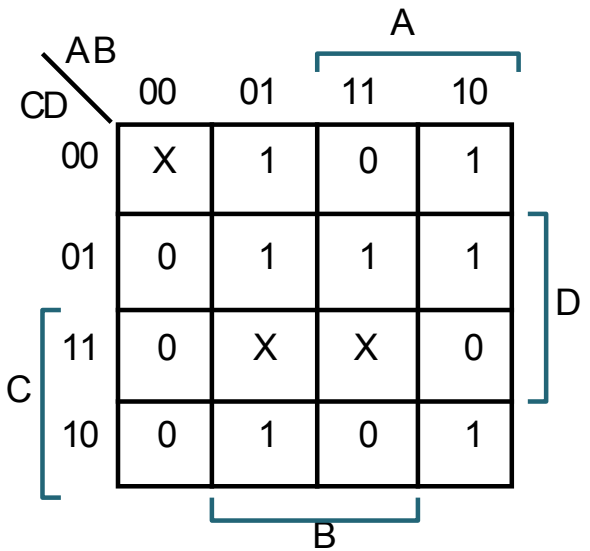
E.g., 0000 vs. 0100 yields 0-00
0000 vs. 1000 yields -000

When used in a combination, mark with a check. If cannot be combined, mark with a star. These are the prime implicants.



Repeat until no further combinations can be made.

Quine-McCluskey Method



Prime Implicants:

0-00 = A' C' D'

100- = A B' C'

1-01 = A C' D

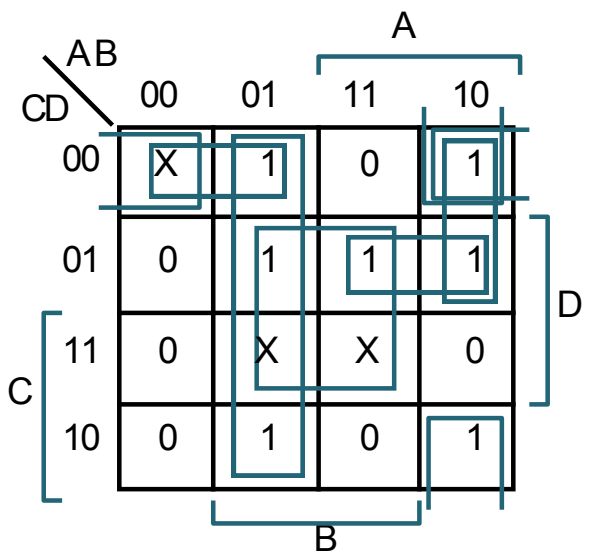
-1-1 = B D

-000 = B' C' D'

10-0 = A B' D'

01-- = A' B

Quine-McCluskey Method Continued



Prime Implicants:

0-00 = A' C' D'

100- = A B' C'

1-01 = A C' D

-1-1 = B D

-000 = B' C' D'

10-0 = A B' D'

01-- = A' B

Stage 2: find smallest set of prime implicants that cover the ON-set

recall that essential prime implicants must be in all covers

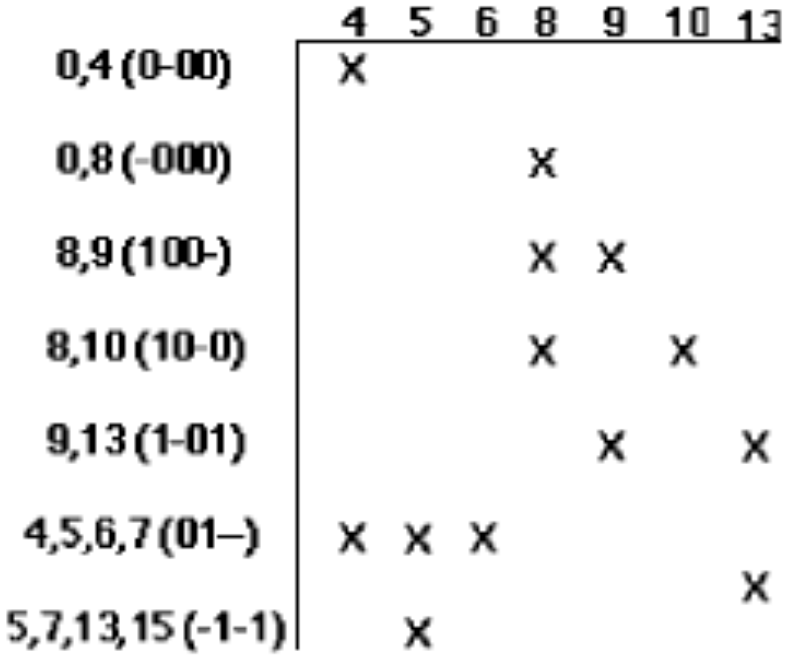
another tabular method– the prime implicant chart *

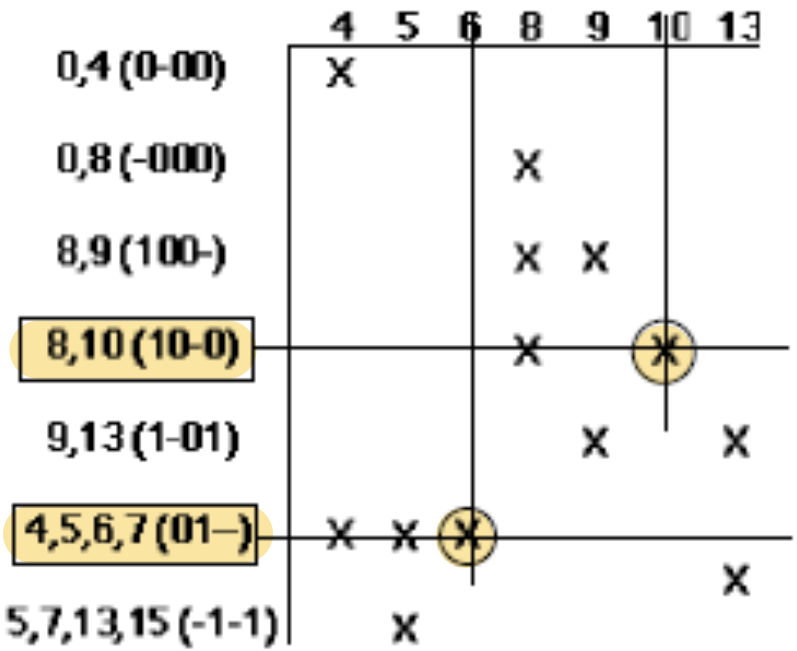
Prime Implicant Chart

	4	5	6	8	9	10	13
0,4 (0-00)	X						
0,8 (-000)				X			
8,9 (100-)				X	X		
8,10 (10-0)				X		X	
9,13 (1-01)					X		X
4,5,6,7 (01--)	X	X	X				
5,7,13,15 (-1-1)		X					X

rows = prime implicants
columns = ON-set elements
place an "X" if ON-set element is covered by the prime implicant

Prime Implicant Chart

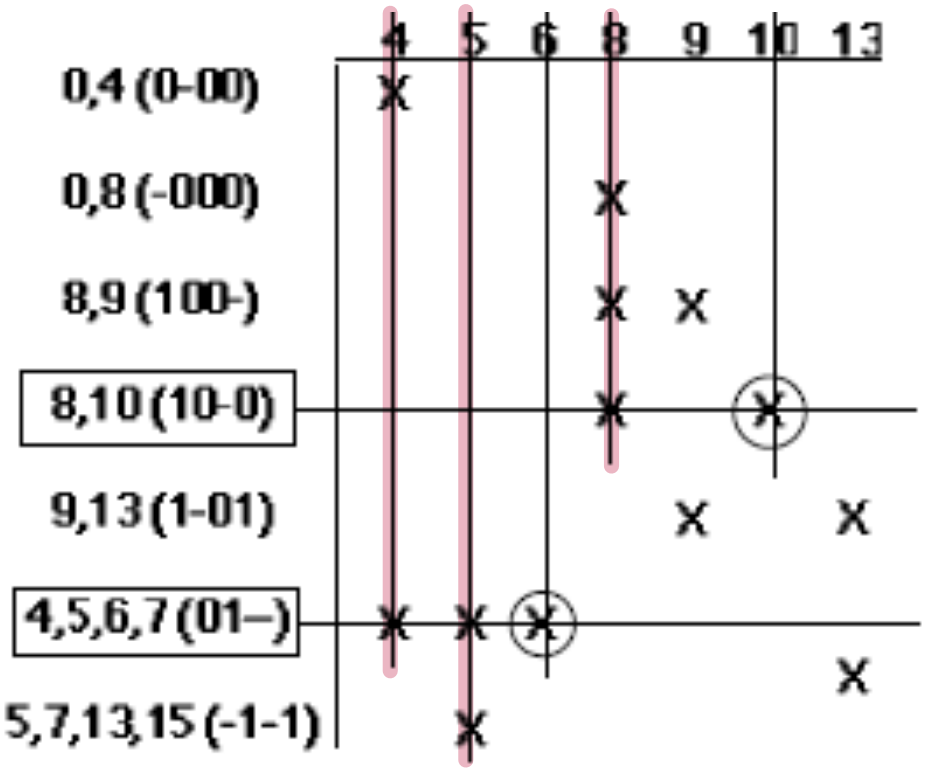




rows = prime implicants
columns = ON-set elements
place an "X" if ON-set element is covered by the prime implicant

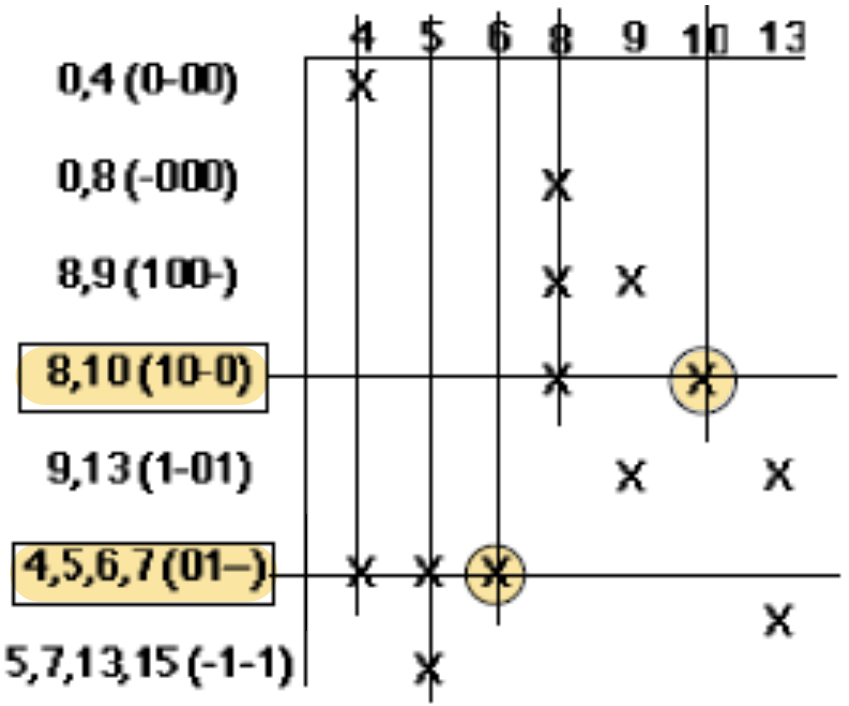
If column has a single X, than the implicant associated with the row is essential. It must appear in minimum cover

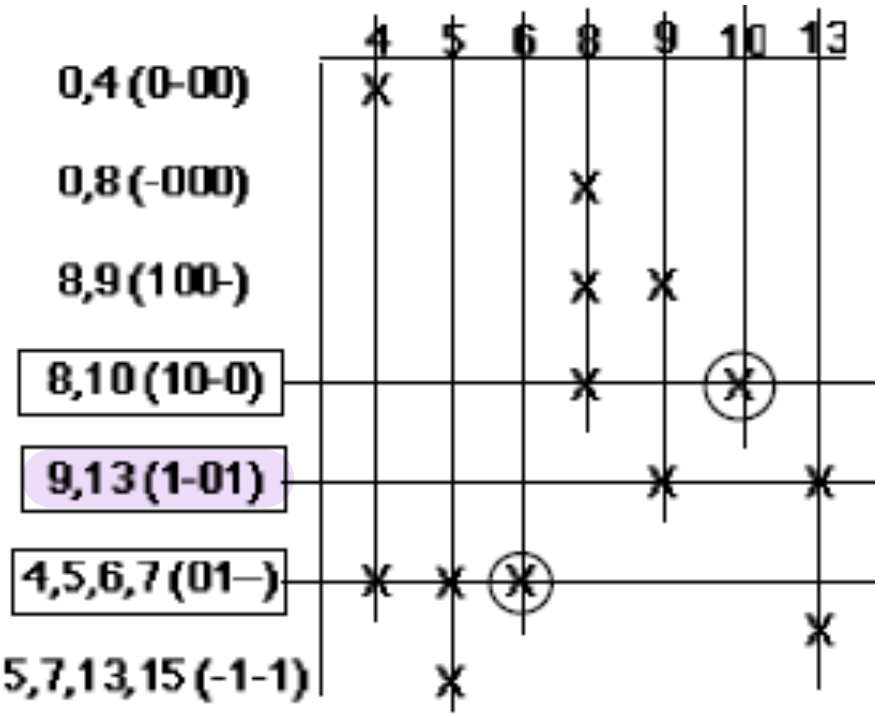
Prime Implicant Chart



Eliminate all columns covered by essential primes

Prime Implicant Chart





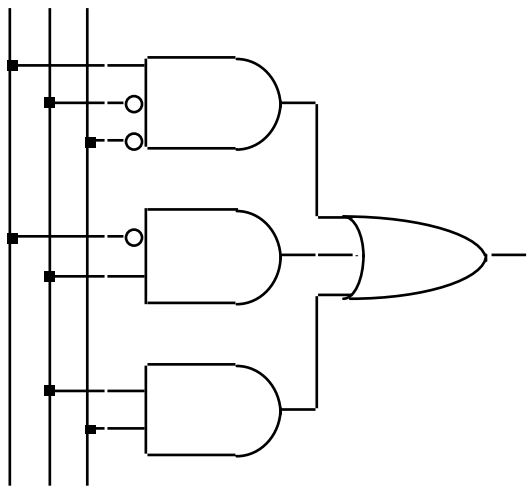
Eliminate all columns covered by essential primes

Find minimum set of rows that cover the remaining columns

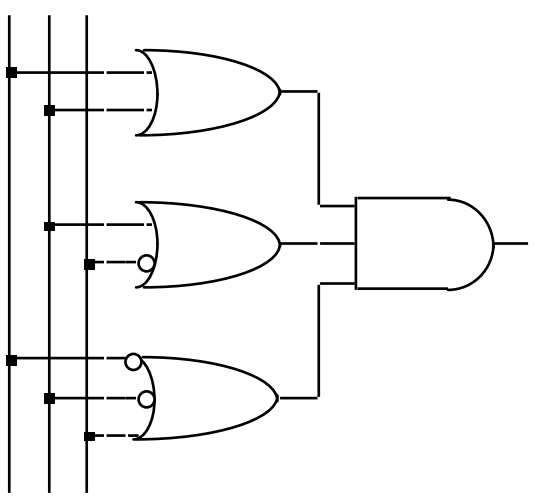
$$f = A\,B'\,D' + A\,C'\,D + A'\,B$$

Implementations of two-level logic

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

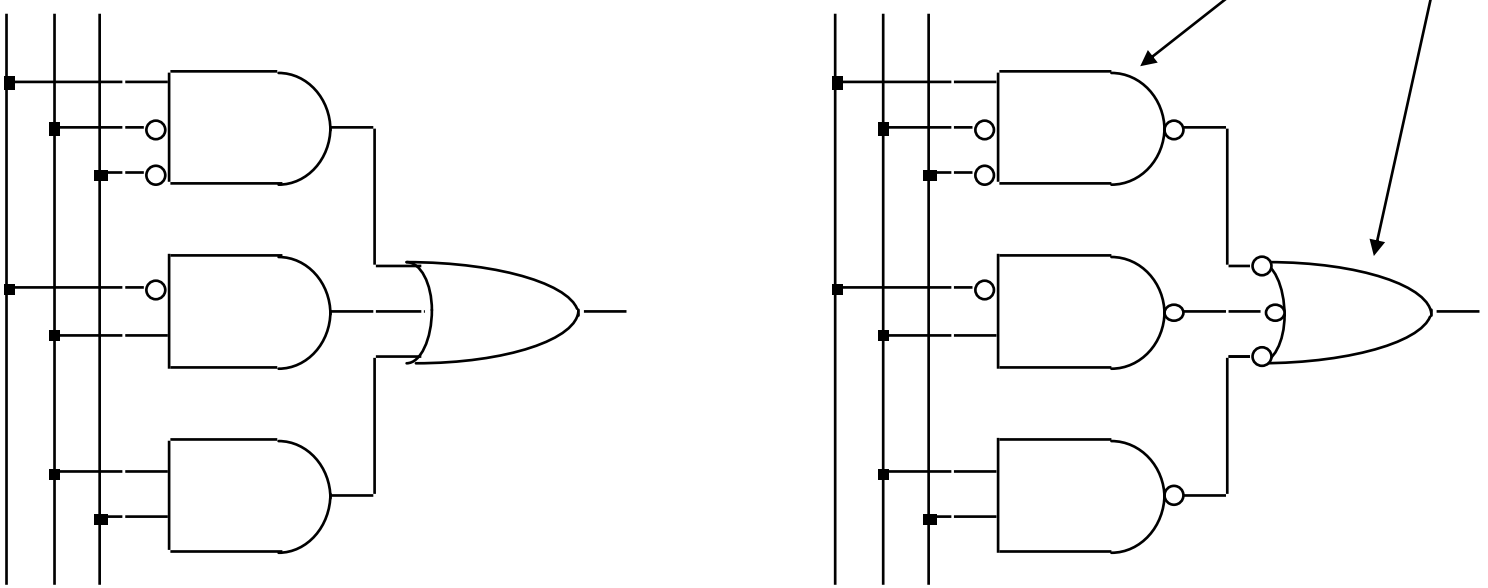


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



Two-level logic using NAND gates

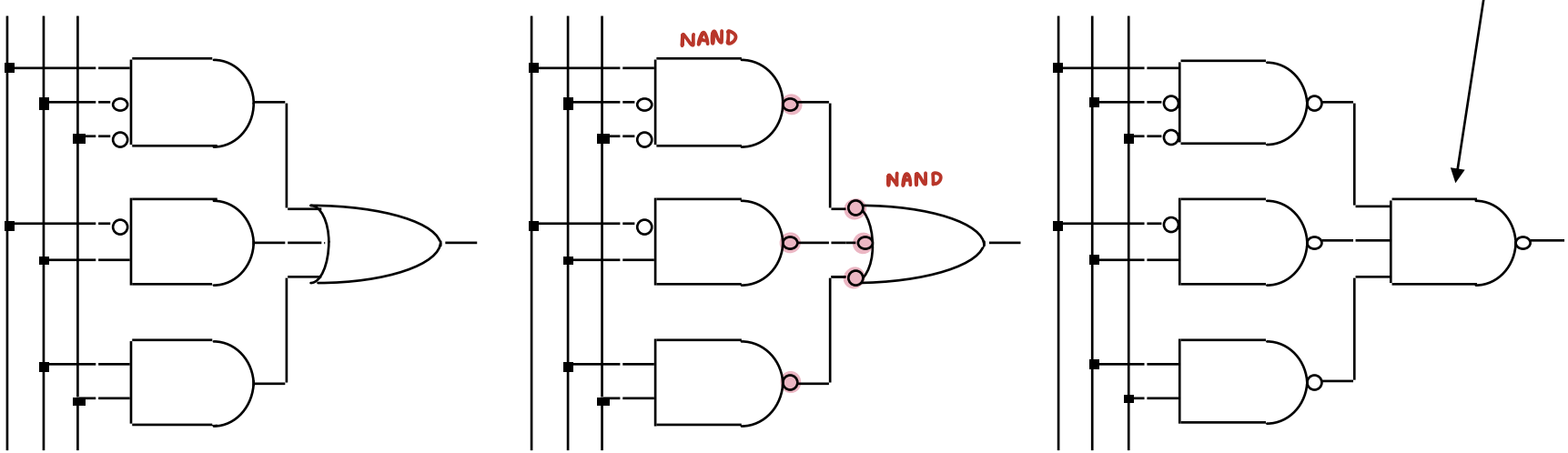
- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate



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

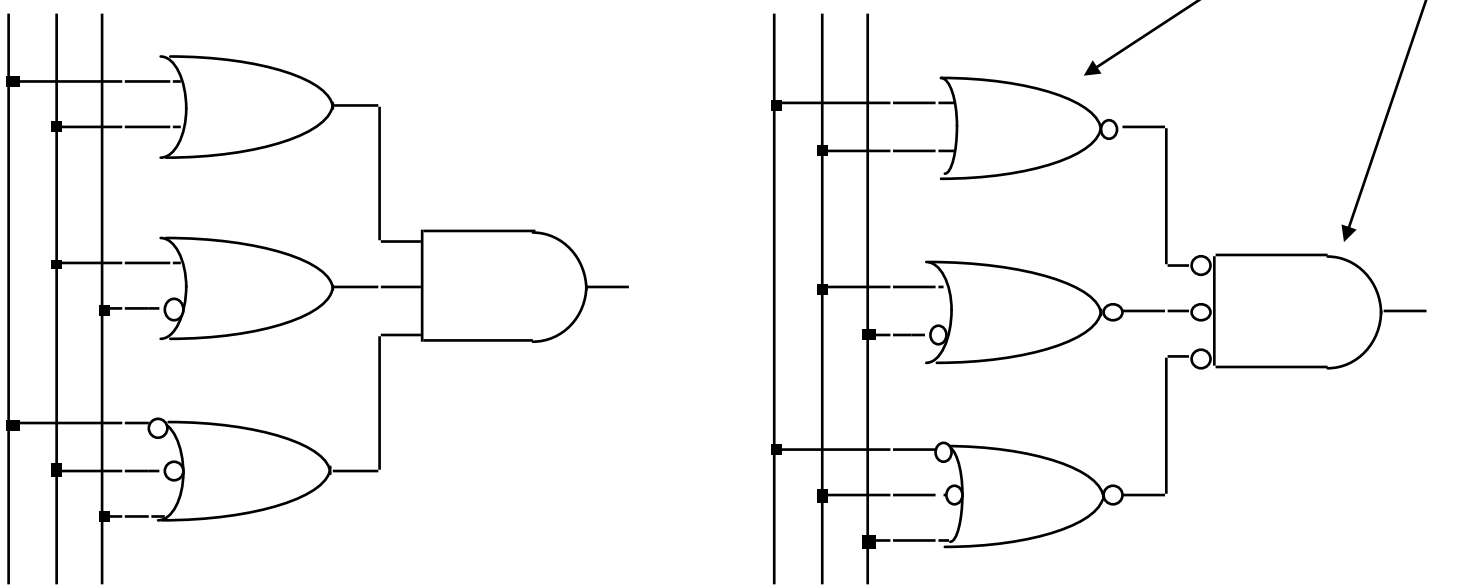
(SoP)

- OR gate with inverted inputs is a NAND gate
  - de Morgan's: A' + B' = (A • 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

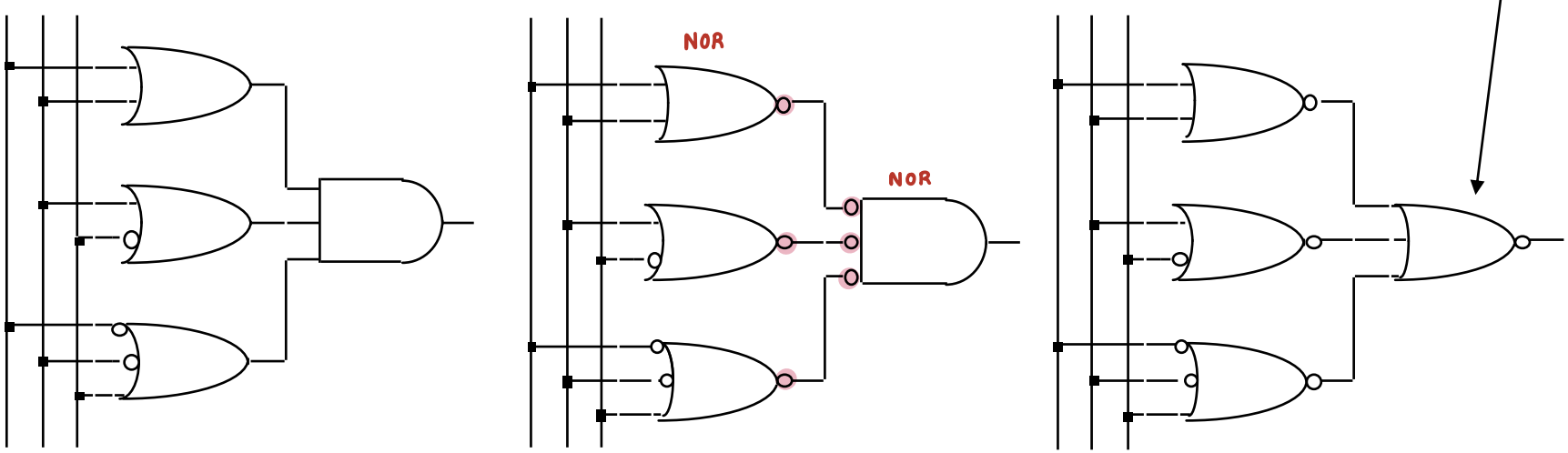
- Replace maxterm OR gates with NOR gates
- Place compensating inversion at inputs of AND gate



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

(PoS)

- 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

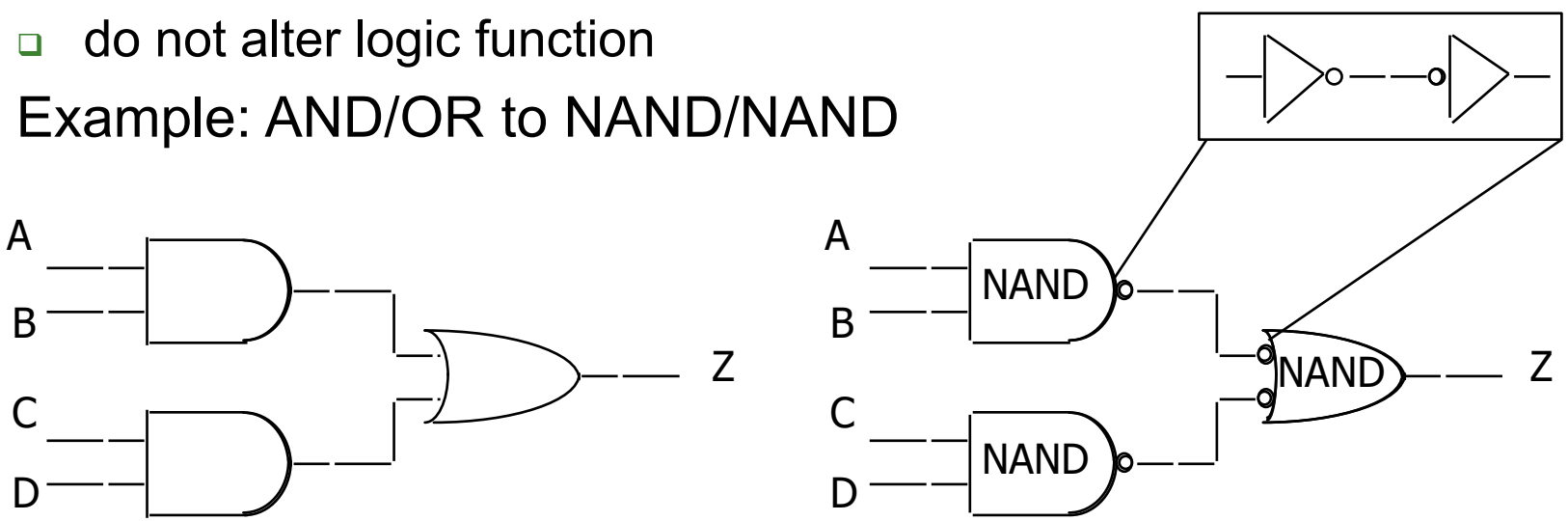


Two-level logic using NAND and NOR gates

- NAND-NAND and NOR-NOR networks
  - de Morgan's law: (A + B)' = A' • B' ① (A • B)' = A' + B' ②
  - written differently: A + B = (A' • B')' ③ (A • B) = (A' + B')' ④
- 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

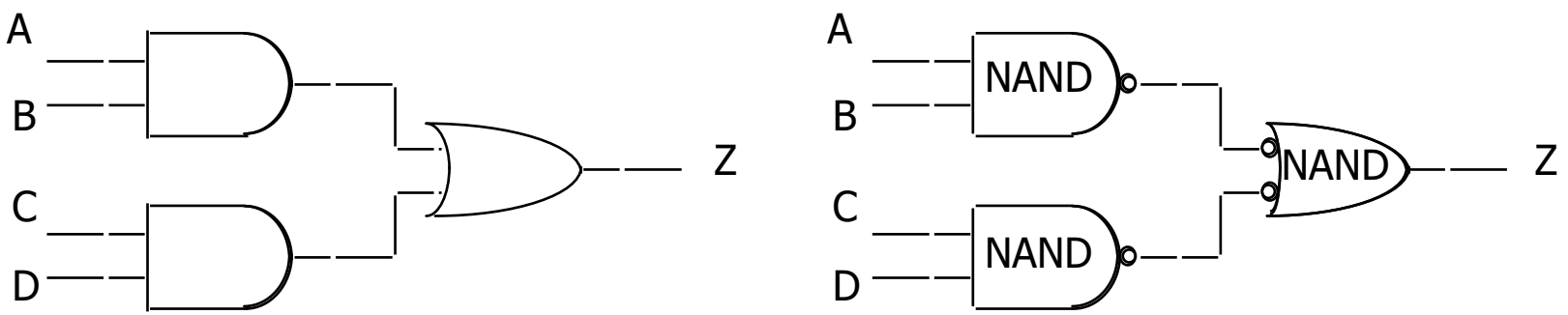
Conversion between forms

- 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)

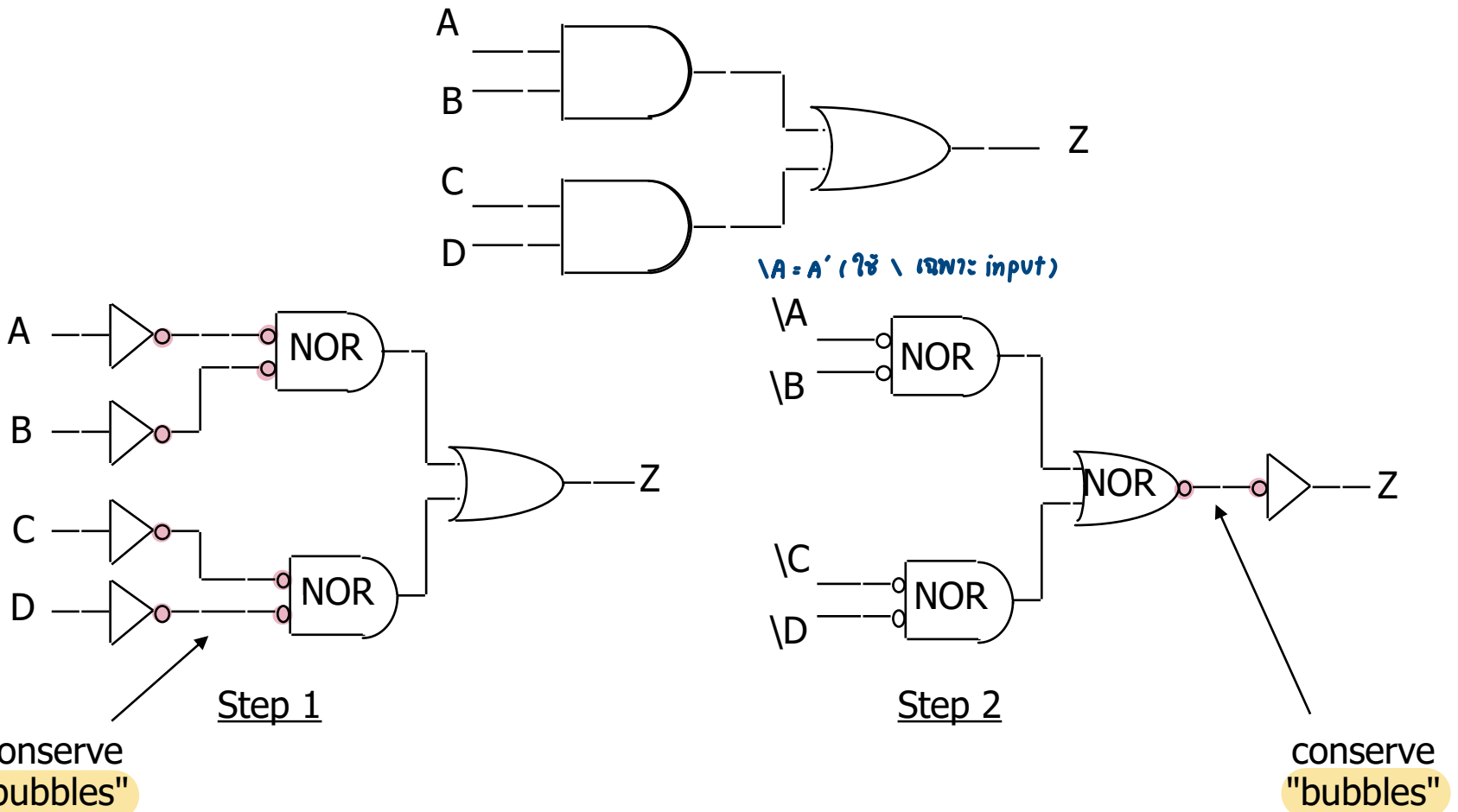
- Example: verify equivalence of two forms



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

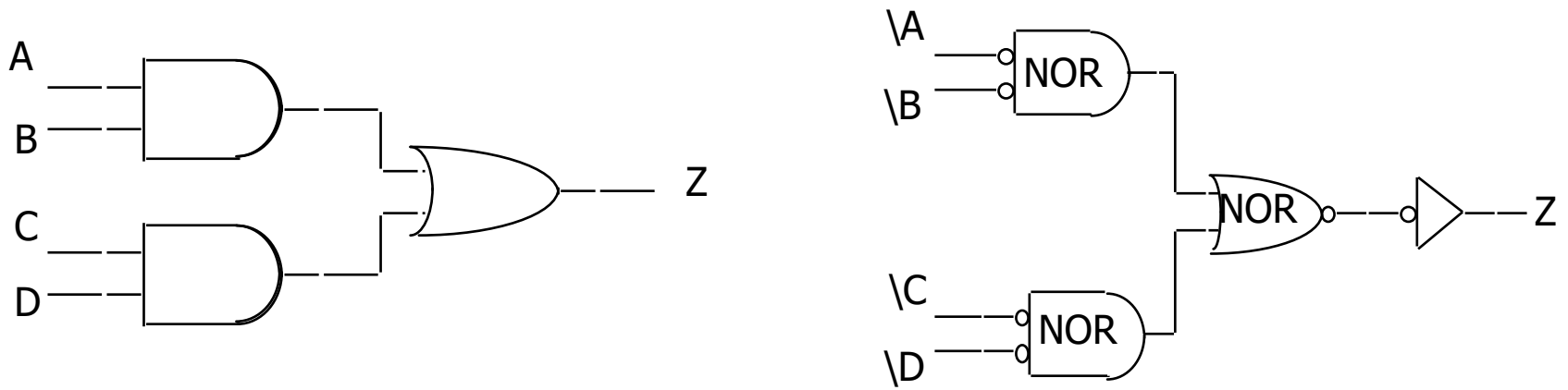
Conversion between forms (cont'd)

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



Conversion between forms (cont'd)

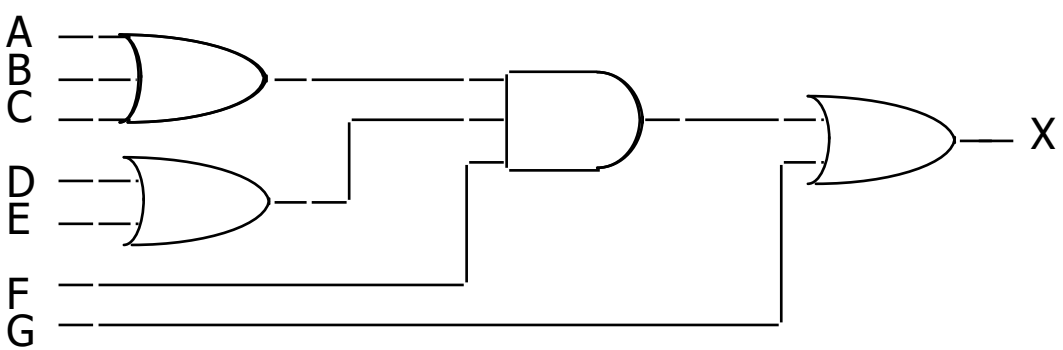
- Example: verify equivalence of two forms



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

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)



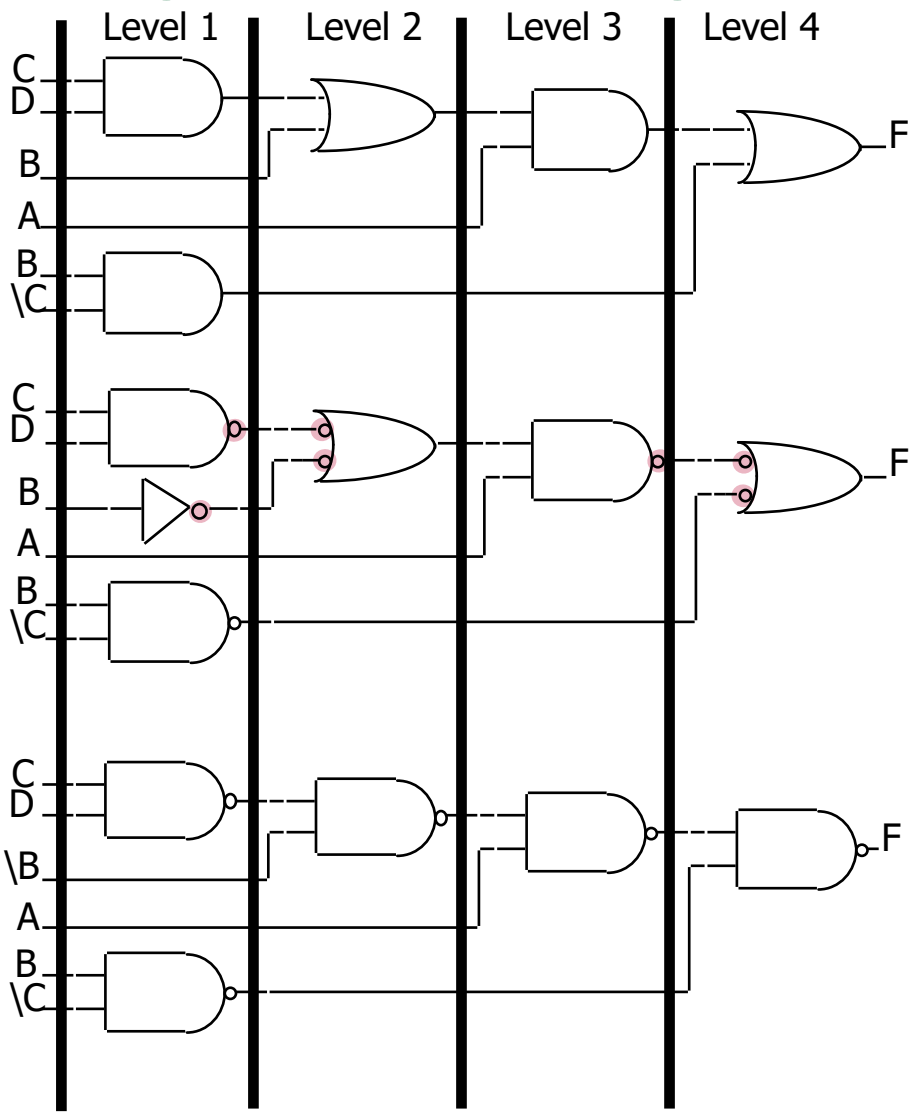
Conversion of multi-level logic to NAND gates

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

original AND-OR network

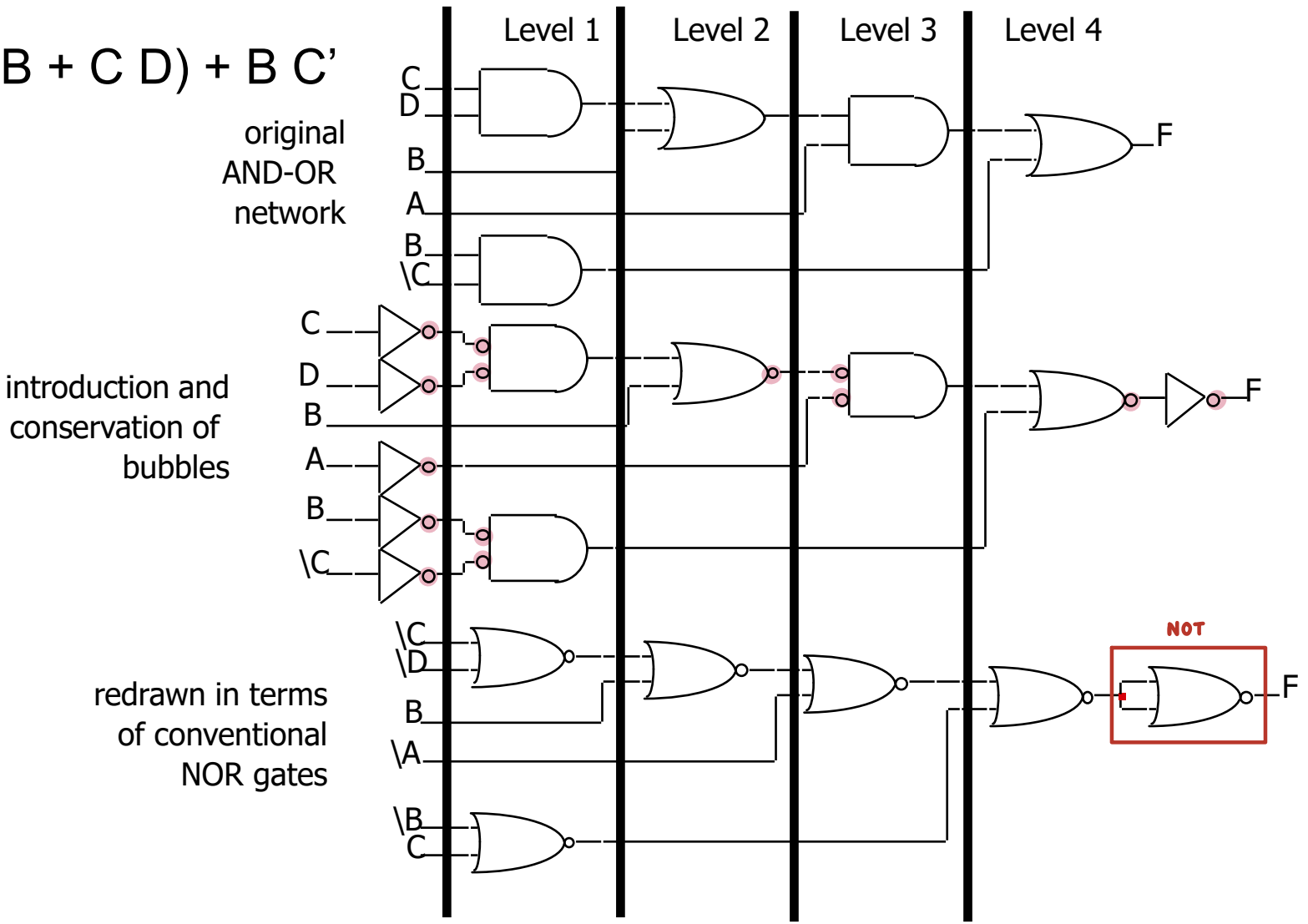
introduction and conservation of bubbles

redrawn in terms of conventional NAND gates



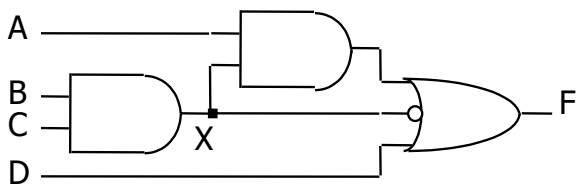
Conversion of multi-level logic to NORs

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

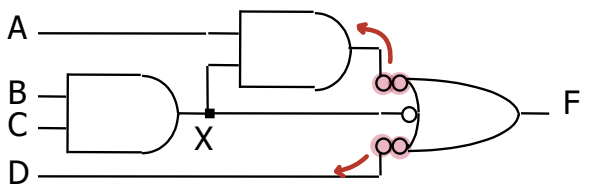


Conversion between forms

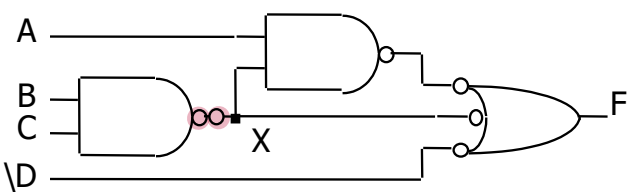
- Example



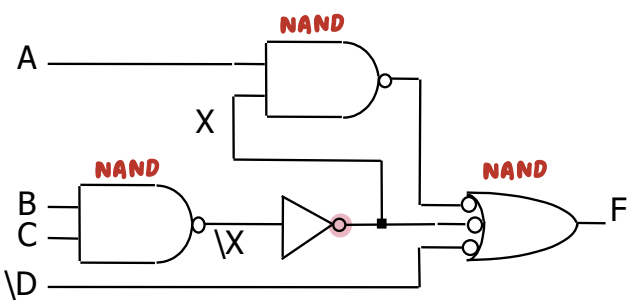
original circuit



add double bubbles to
invert all inputs of OR gate



add double bubbles to
invert output of AND gate

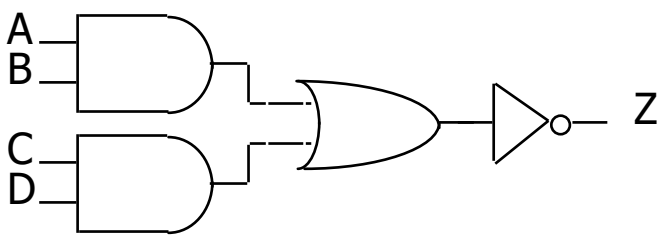


insert inverters to eliminate
double bubbles on a wire

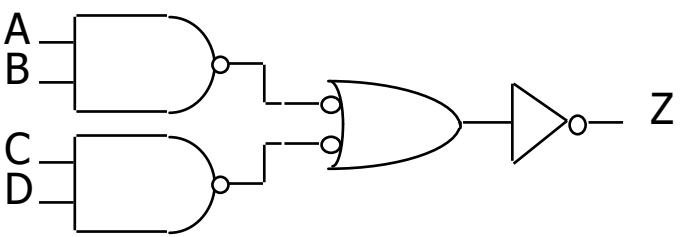
AND-OR-invert gates

- AOI function: three stages of logic — AND, OR, Invert
  - multiple gates "packaged" as a single circuit block

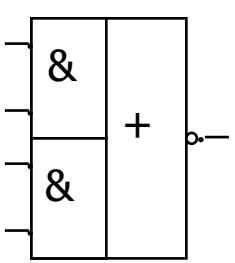
logical concept



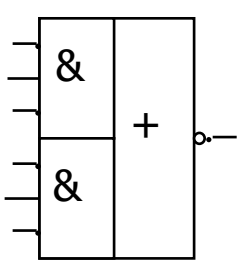
possible implementation



2x2 AOI gate symbol

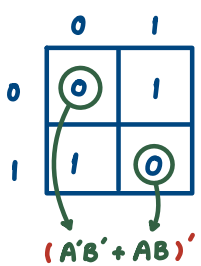


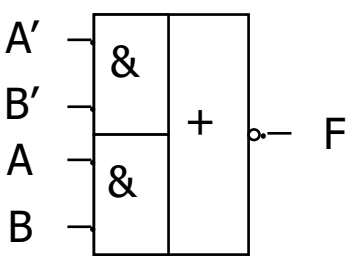
3x2 AOI gate symbol



Conversion to AOI forms

- General procedure to place in AOI form
  - compute the complement of the function in sum-of-products form
  - by grouping the 0s in the Karnaugh map
- Example: XOR implementation
  - A xor B = A' B + A B'
  - AOI form:
    - F = (A' B' + A B)'





Examples of using AOI gates

- Example:
  - F = A B + A C' + B C'
  - F = (A' B' + A' C + B' C)'
  - Implemented by 2-input 3-stack AOI gate

  - F = (A + B) (A + C') (B + C')
  - F = [(A' + B') (A' + C) (B' + C)]'
  - Implemented by 2-input 3-stack OAI gate

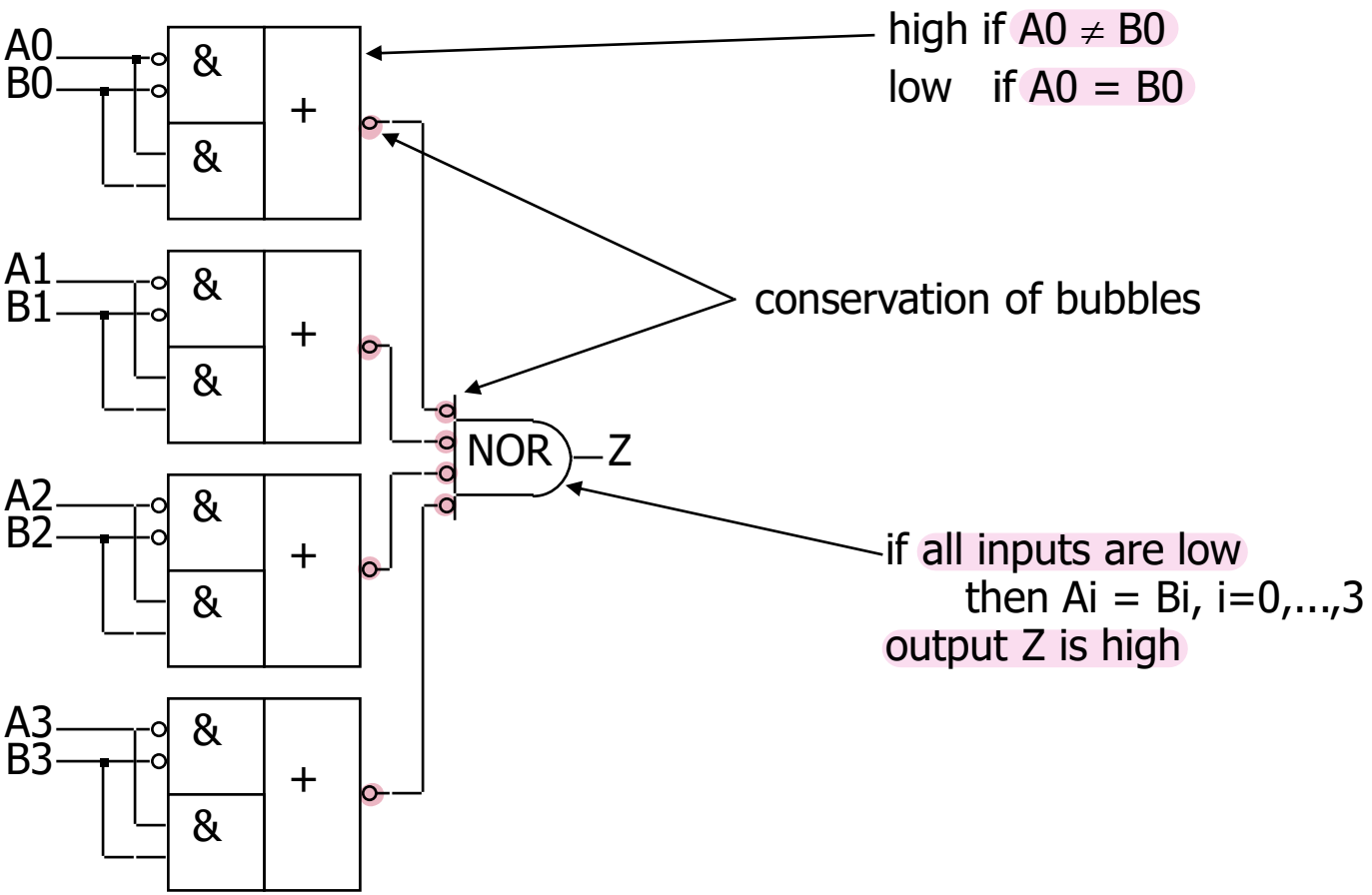
- Example: 4-bit equality function
  - Z = (A0 B0 + A0' B0')(A1 B1 + A1' B1')(A2 B2 + A2' B2')(A3 B3 + A3' B3')

each implemented in a single 2x2 AOI gate

Examples of using AOI gates (cont'd)

$A_3A_2A_1A_0 = B_3B_2B_1B_0$ เท่ากัน เมื่อทุก bit เท่ากัน

- Example: AOI implementation of 4-bit equality function



Summary for multi-level logic

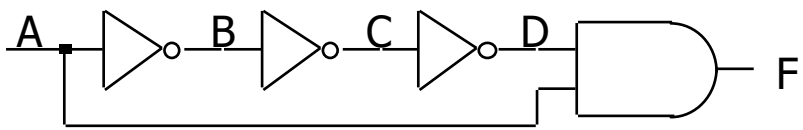
- Advantages
  - circuits may be smaller
  - gates have smaller fan-in
  - circuits may be faster
- Disadvantages
  - more difficult to design
  - tools for optimization are not as good as for two-level
  - analysis is more complex

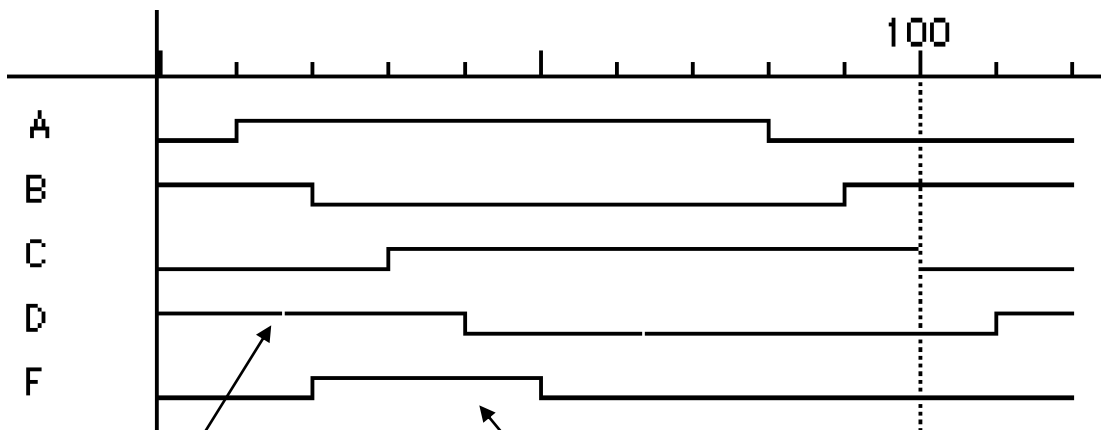
Time behavior of combinational networks

- Waveforms
  - visualization of values carried on signal wires over time
  - useful in explaining sequences of events (changes in value)
- Simulation tools are used to create these waveforms
  - input to the simulator includes gates and their connections
  - input stimulus, that is, input signal waveforms
- Some terms
  - gate delay — time for change at input to cause change at output
    - min delay – typical/nominal delay – max delay
    - careful designers design for the worst case
  - rise time — time for output to transition from low to high voltage
  - fall time — time for output to transition from high to low voltage
  - pulse width — time that an output stays high or stays low between changes

Momentary changes in outputs

- Can be useful — pulse shaping circuits
- Can be a problem — incorrect circuit operation (glitches/hazards)
- Example: pulse shaping circuit
  - A' • A = 0
  - delays matter



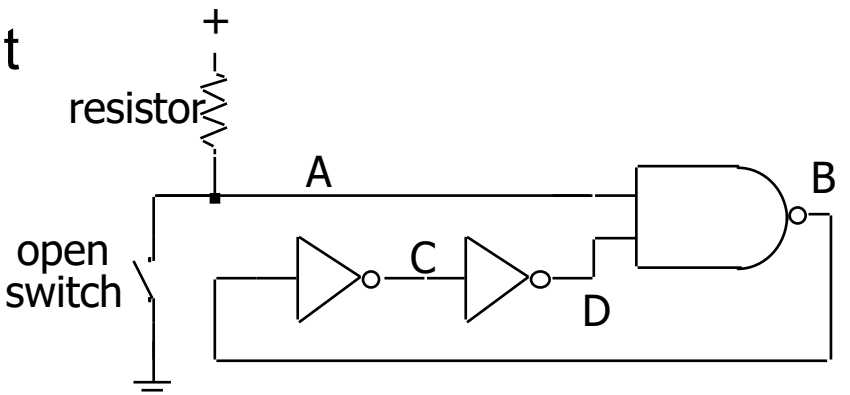


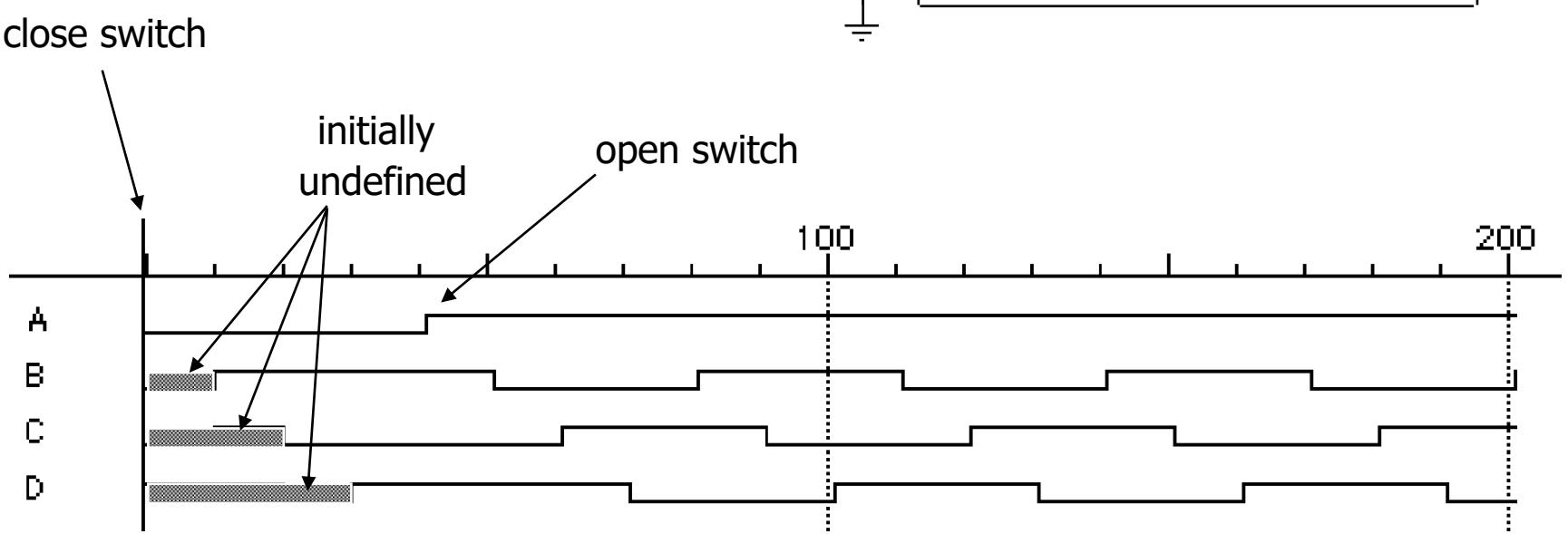
D remains high for three gate delays after A changes from low to high

F is not always 0 pulse 3 gate-delays wide

Oscillatory behavior

- Another pulse shaping circuit





Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics

HDLs

- Abel (circa 1983) - developed by Data-I/O
  - targeted to programmable logic devices
  - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
  - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
  - similar to Pascal and C
  - delays is only interaction with simulator
  - fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - similar to Ada (emphasis on re-use and maintainability)
  - simulation semantics visible
  - very general but verbose
  - IEEE standard

Verilog

- Supports structural and behavioral descriptions
- Structural
  - explicit structure of the circuit
  - e.g., each logic gate instantiated and connected to others
- Behavioral
  - program describes input/output behavior of circuit
  - many structural implementations could have same behavior
  - e.g., different implementation of one Boolean function
- We'll mostly be using behavioral Verilog in Aldec ActiveHDL
  - rely on schematic when we want structural descriptions

Hardware description languages vs. programming languages

- Program structure
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules (only leaves can be HDL in Aldec ActiveHDL)
- Assignment
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)
- Data structures
  - size explicitly spelled out - no dynamic structures
  - no pointers
- Parallelism
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)

Hardware description languages and combinational logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate's output is a function of its inputs at all times (doesn't need to wait to be "called")
- Propagation delay- concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks

Working with combinational logic summary

- Design problems
  - filling in truth tables
  - incompletely specified functions
  - simplifying two-level logic
- Realizing two-level logic
  - NAND and NOR networks
  - networks of Boolean functions and their time behavior
- Time behavior
- Hardware description languages
- Later
  - combinational logic technologies
  - more design case studies