

---

Digital Logic Design

ECEN 3233

Module 3: Working with Combinational Logic

---

Louis G. Johnson

School of Electrical and Computer Engineering

Oklahoma State University

Fall 2007

# 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

# 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)
  - we'll concentrate on this for SOP and POS realizations

# The uniting theorem

- Key tool to simplification:  $A(B' + B) = A$
- Can be visualized as a Boolean Cube (a geometric relationship in space)
- 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
0	0	1
0	1	0
1	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

# 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	0	1
B	0	1	1
	1	0	0

0 1

1 3

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

		A			
		00	01	11	10
C	0	0	2	6	4
	1	1	3	7	5
		B			

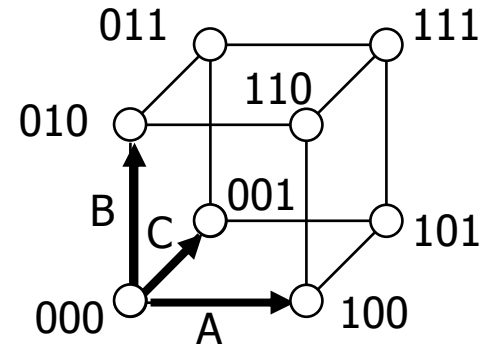
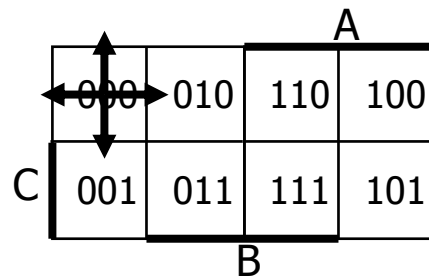
		A			
		0	2	6	4
C	0	0	2	6	4
	1	1	3	7	5
		B			

		A			
		0	4	12	8
C	0	0	4	12	8
	1	1	5	13	9
		D			
C	0	3	7	15	11
	1	2	6	14	10
		B			

$$13 = 1101 = ABC'D$$

# Adjacencies in Karnaugh maps

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



# Karnaugh map examples

■  $F =$

	A	
	1	1
B	0	0

$B'$

■  $C_{out} =$

	A			
	0	0	1	0
C <sub>in</sub>	0	1	1	1
	B			

$AB + AC_{in} + BC_{in}$

■  $f(A,B,C) = \Sigma m(0,4,5,7)$

	A			
	1	0	0	1
C	0	0	1	1
	B			

$AC + B'C' + \cancel{AB'}$

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

# More Karnaugh map examples

		A	
	0	0	1 1
C	0	0	1 1
		B	

$$G(A,B,C) = A$$

		A	
	1	0	0 1
C	0	0	1 1
		B	

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

		A	
	0	1 1	0
C	1	1	0 0
		B	

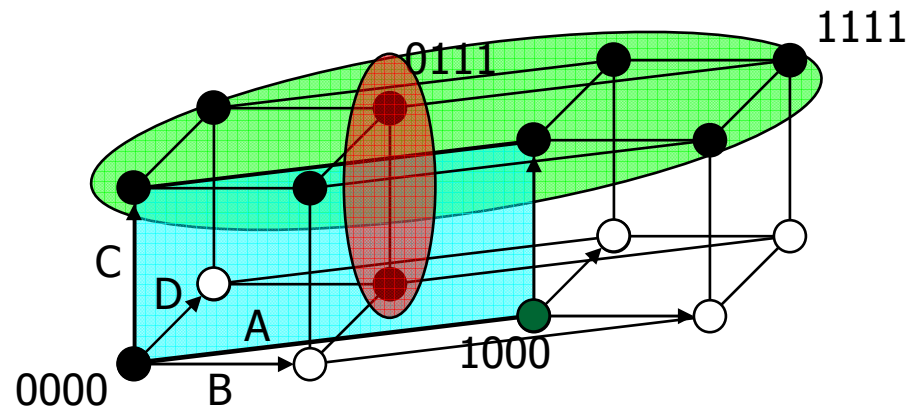
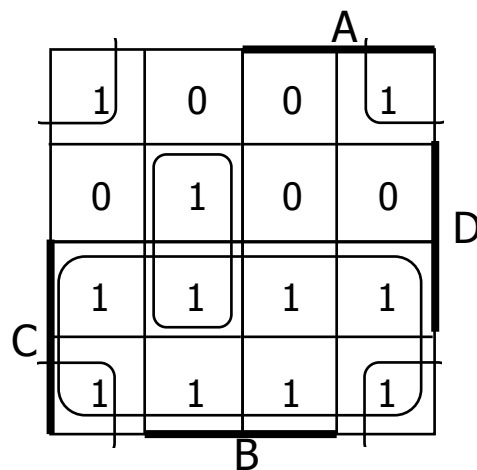
F' simply replace 1's with 0's and vice versa

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

# Karnaugh map: 4-variable example

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

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



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

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

	A			
	0	0	X	0
	1	1	X	1
	1	1	0	0
C	0	X	0	0
	B			

The Karnaugh map is a 4x4 grid with variables A, B, C, and D. The columns are labeled A (0, 0, X, 0) and the rows are labeled C (0, 1, 1, 0). The cells contain values: (A=0, C=0) is 0; (A=0, C=1) is 1; (A=0, C=1) is 1; (A=0, C=0) is 0. (A=0, C=0) is X; (A=0, C=1) is X; (A=0, C=1) is 0; (A=0, C=0) is 0. (A=1, C=0) is 0; (A=1, C=1) is 1; (A=1, C=1) is 1; (A=1, C=0) is 0. (A=1, C=0) is 0; (A=1, C=1) is 0; (A=1, C=1) is 0; (A=1, C=0) is 0. Two groups are circled: a horizontal group of 1s in the second row (A=0, C=1) and a vertical group of 1s in the first column (A=0, C=0 and A=0, C=1).

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

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

□  $f = A'D + B'C'D$

without don't cares

□  $f = A'D + C'D$

with don't cares

		A		
	0	0	X	0
	1	1	X	1
	1	1	0	0
C	0	X	0	0
		B		

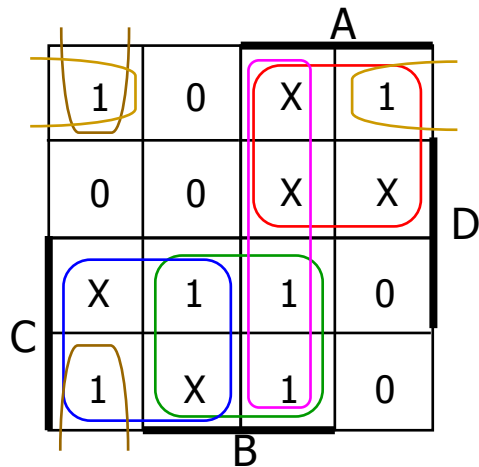
D

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

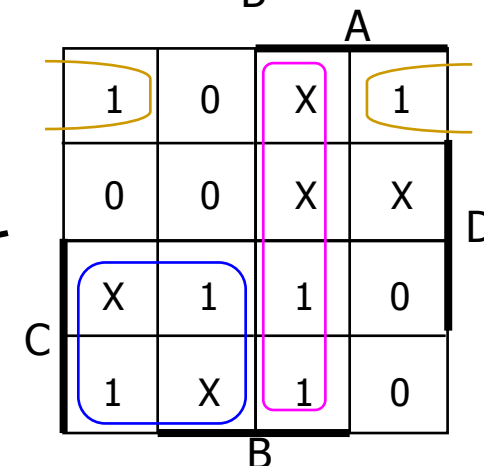
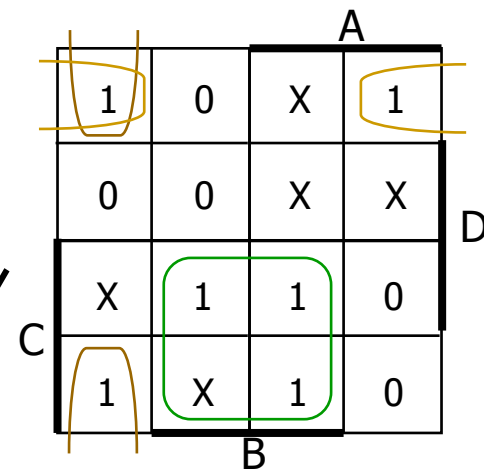
- Minimize the function  $F = \Sigma m(0, 2, 7, 8, 14, 15) + d(3, 6, 9, 12, 13)$



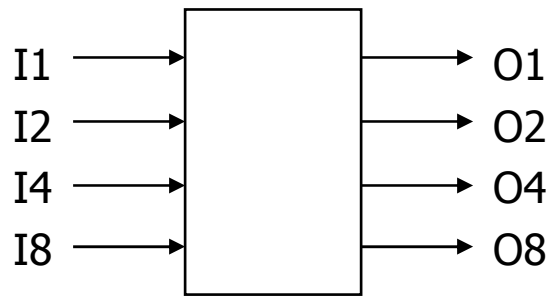
$$F = \cancel{AC'} + \cancel{A'C} + BC + \cancel{AB} + A'B'D' + B'C'D'$$

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

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



# 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	0
1	0	0	1	0	X	X	X
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)

		I8		
		X	1	<u>O8</u>
		X	0	I1
I2		1	X	
		X	X	
		I4		

$$O8 = I4 I2 I1 + I8 I1'$$

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

$$O2 = I8' I2' I1 + I2 I1'$$

$$O1 = I1'$$

		I8		
<u>O4</u>		1	X	0
		1	X	0
I2		1	0	X
		1	X	X
		I4		

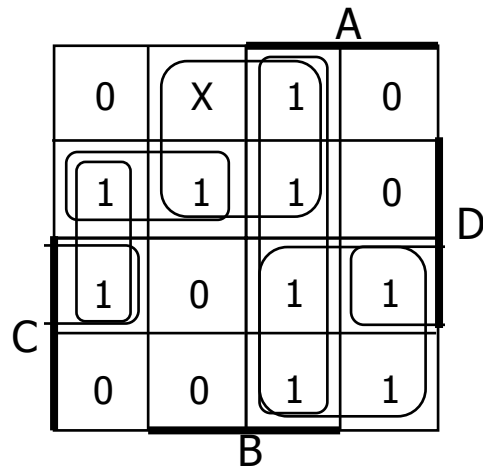
		I8		
		X	0	<u>O2</u>
		X	0	I1
I2		0	0	X
		1	1	X
		I4		

		I8		
<u>O1</u>		1	1	X
		1	1	1
I2		0	0	X
		0	0	X
		1	1	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

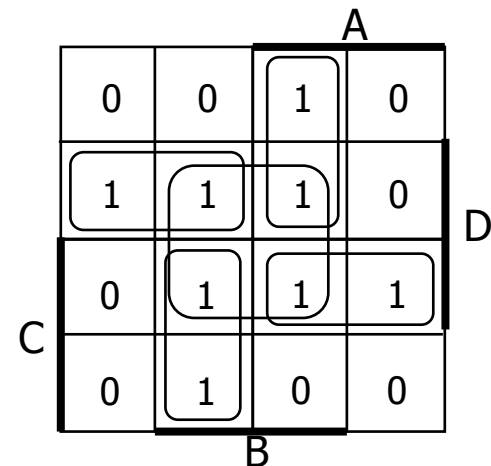
minimum cover:  $AC + BC' + A'B'D$

5 prime implicants:

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

essential

minimum cover: 4 essential implicants

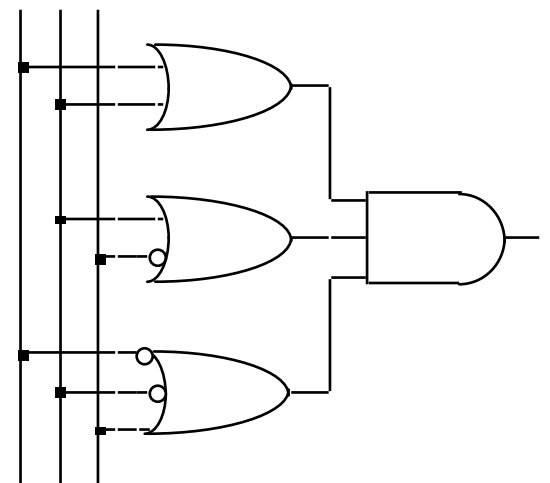
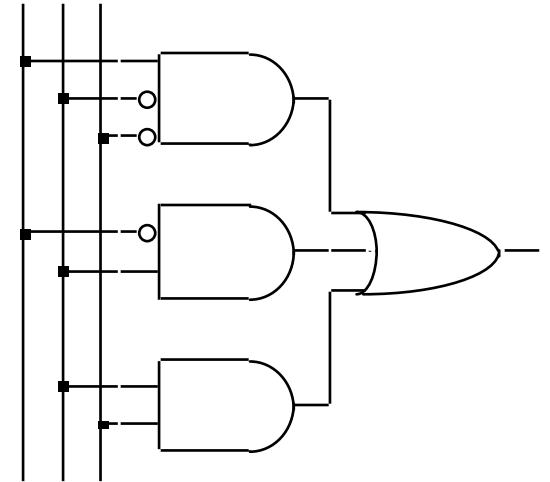


# Algorithm for two-level simplification

- 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
    - 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

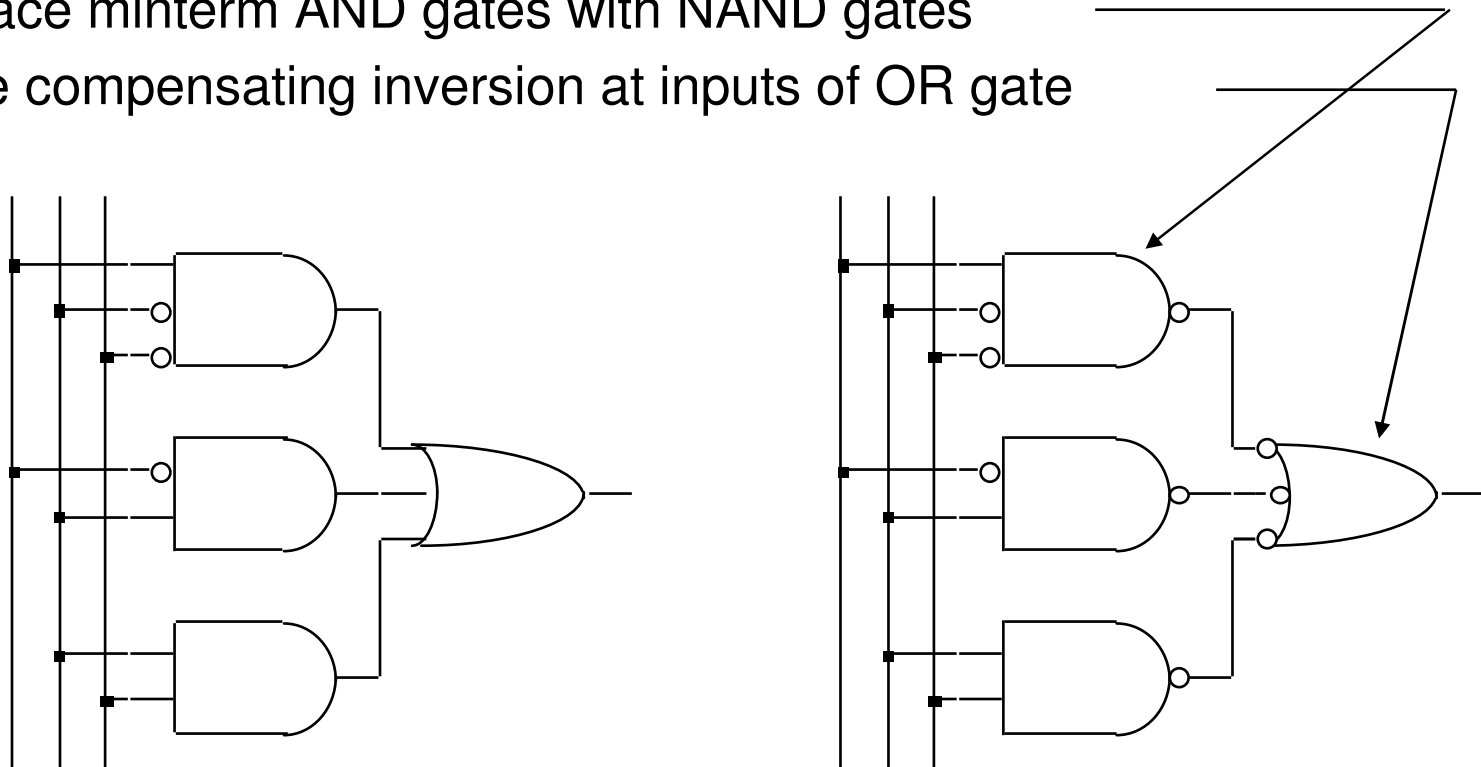
# Implementations of two-level logic

- 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



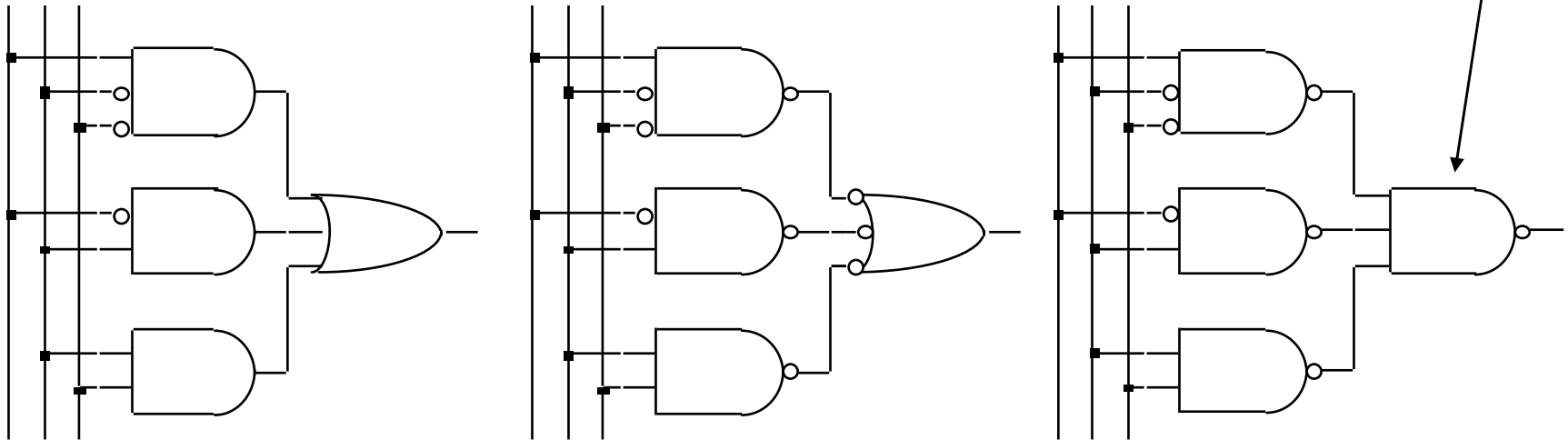
## Two-level logic using NAND gates

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



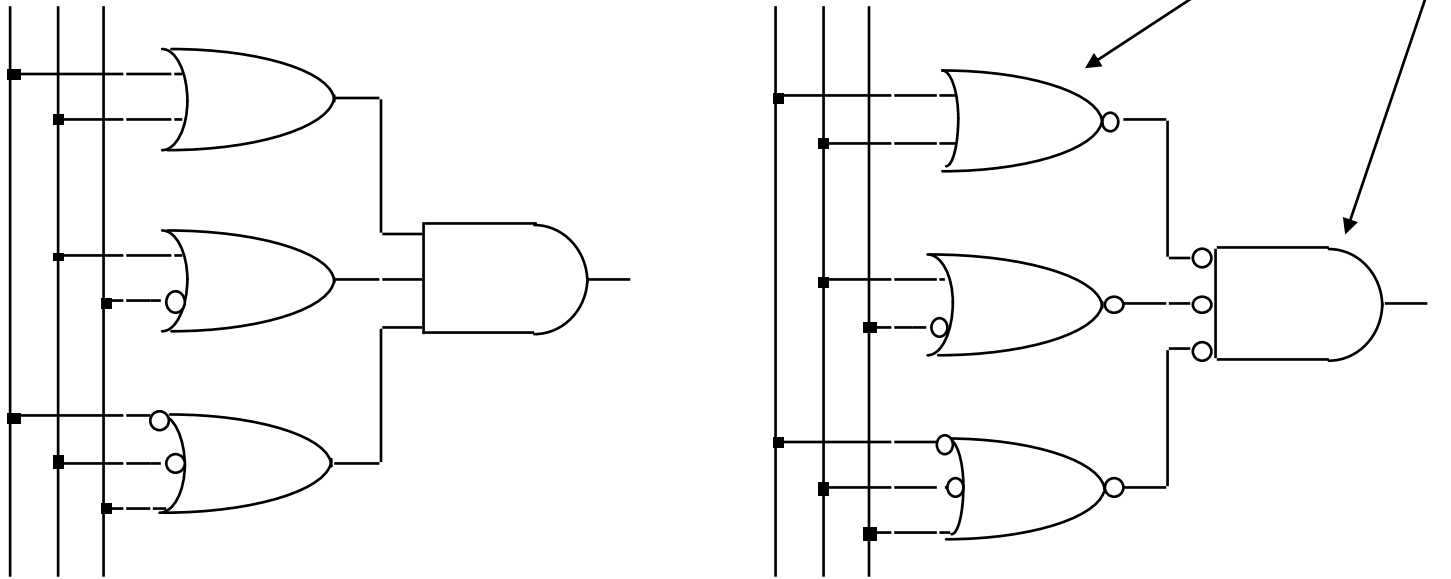
## Two-level logic using NAND gates (cont'd)

- 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



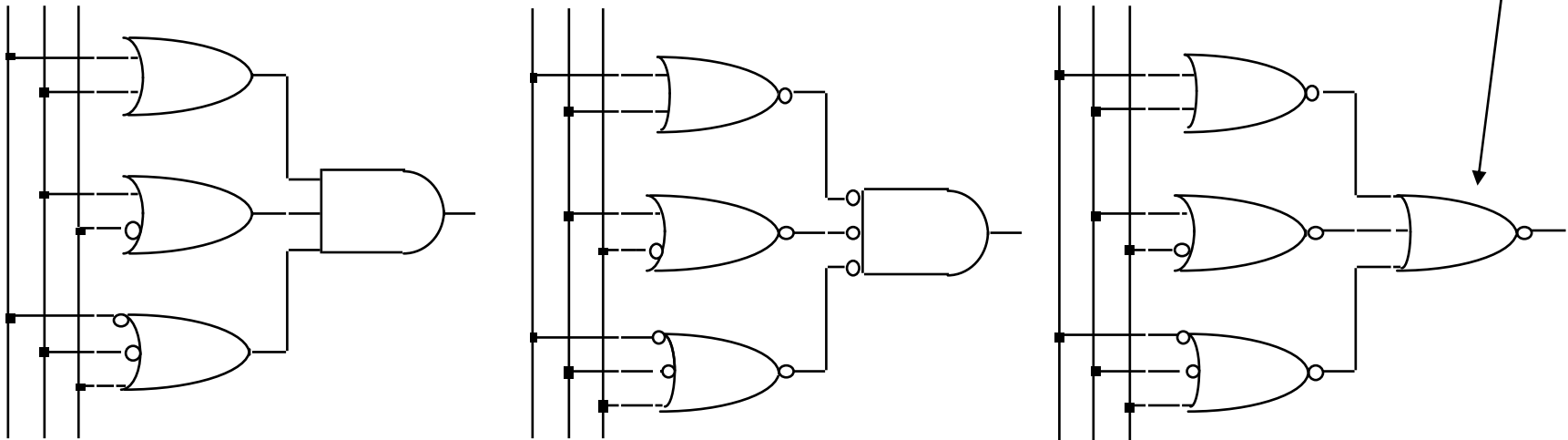
# Two-level logic using NOR gates

- 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:  $A' \cdot B' = (A + B)'$
- 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:  $(A + B)' = A' \cdot B'$        $(A \cdot B)' = A' + B'$
- written differently:  $A + B = (A' \cdot B')'$        $(A \cdot B) = (A' + B)'$

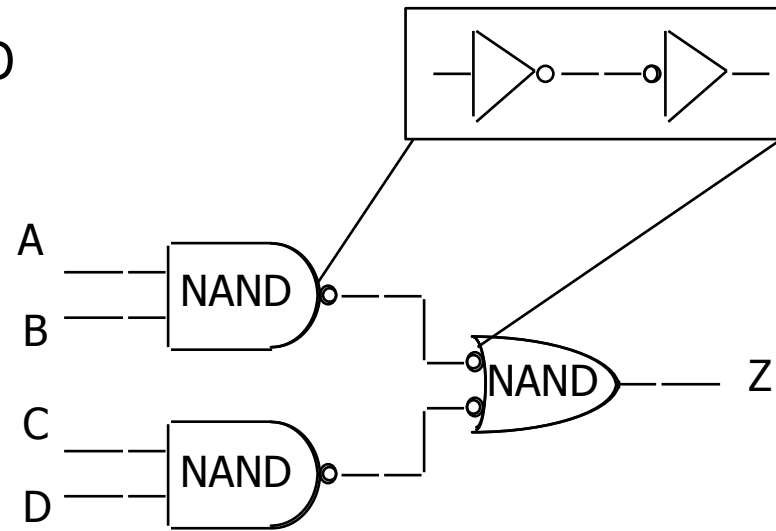
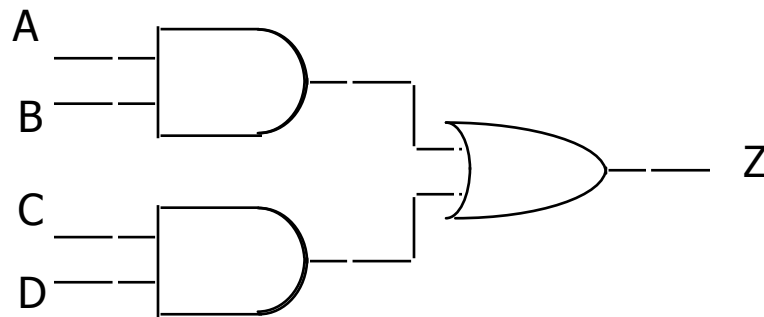
- In other words —

- 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



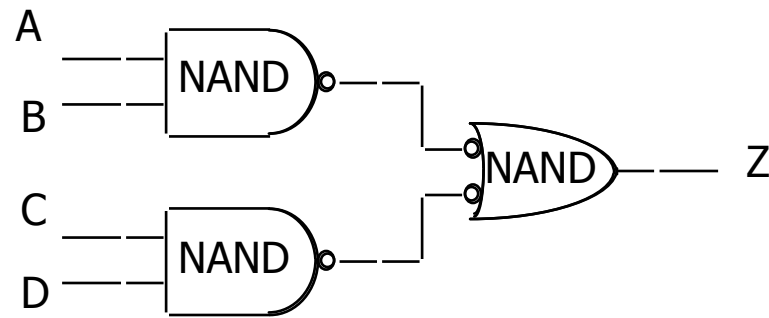
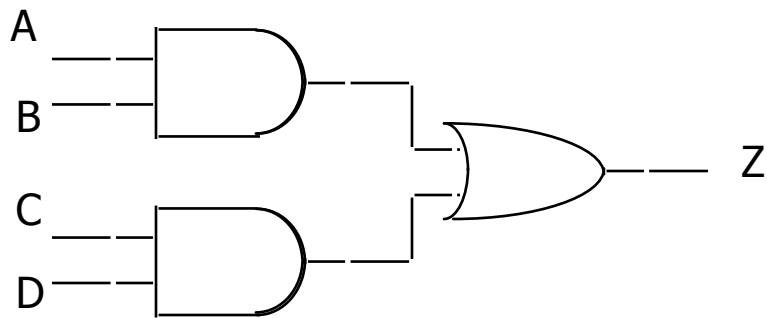
# Conversion between forms

- 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



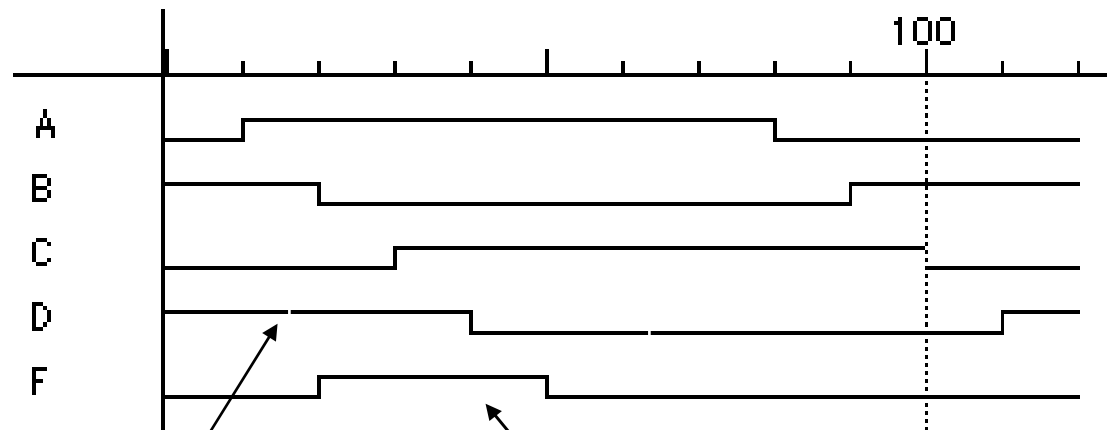
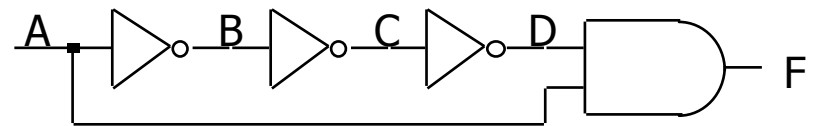
$$\begin{aligned} Z &= [ (A \cdot B)' \cdot (C \cdot D)' ]' \\ &= [ (A' + B') \cdot (C' + D') ]' \\ &= [ (A' + B')' + (C' + D')' ] \\ &= (A \cdot B) + (C \cdot D) \Rightarrow \end{aligned}$$

# Time behavior of combinational networks

- Circuit behavior is NOT ideal
- 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 - later)
- Example: pulse shaping circuit
  - $A' \cdot A = 0$
  - delays matter

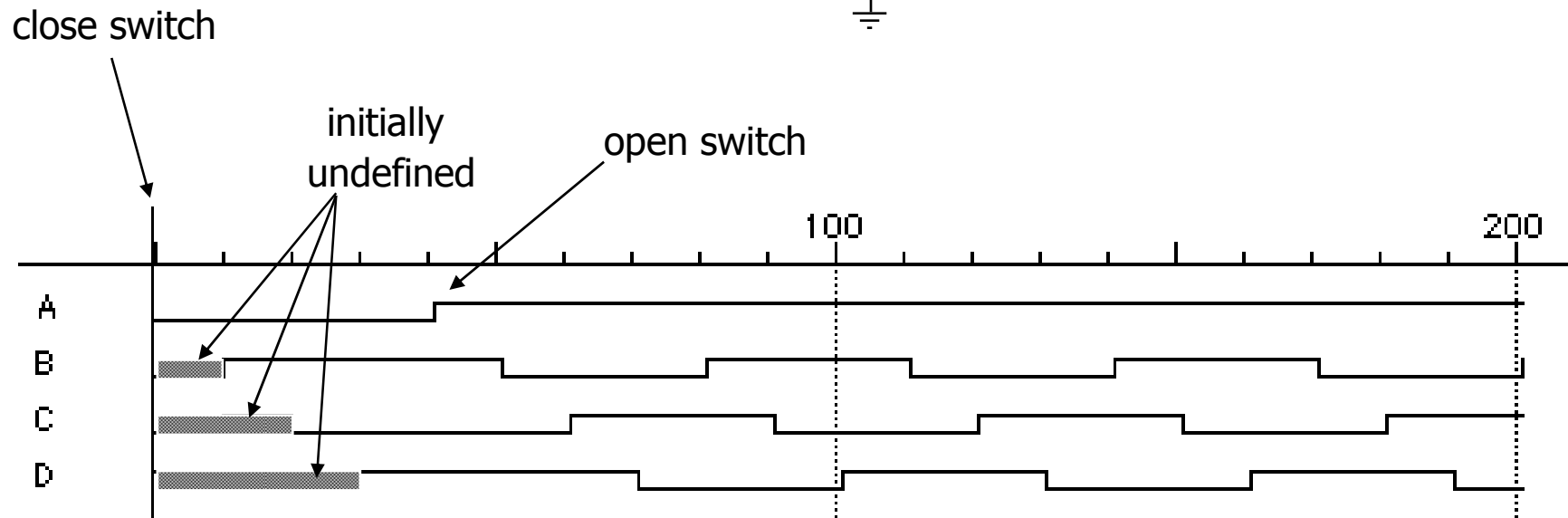
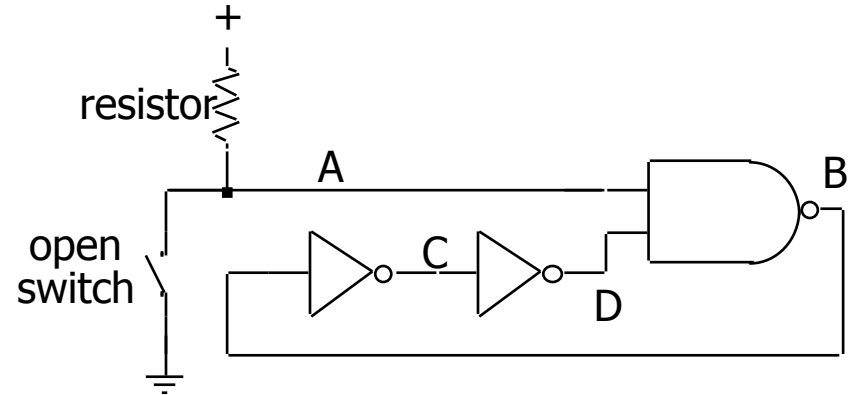


D remains high for three gate delays after A changes from low to high

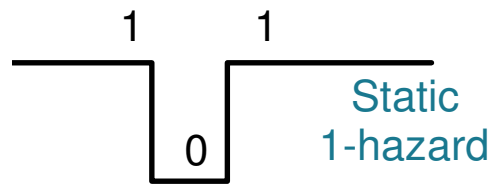
F is not always 0 pulse 3 gate-delays wide

# Oscillatory behavior

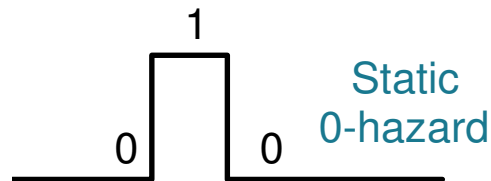
- Another pulse shaping circuit



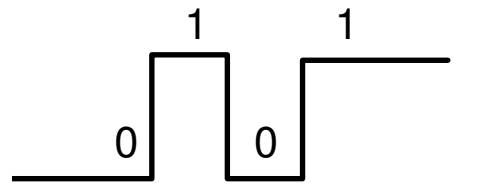
# Time Response in Combinational Networks



Input change causes output to go from 1 to 0 to 1



Input change causes output to go from 0 to 1 to 0



Input change causes a double change  
from 0 to 1 to 0 to 1 OR  
from 1 to 0 to 1 to 0

Kinds of Hazards

# Glitches and Hazards

- Hazard – a circuit is said to have a “hazard” if it is capable of producing a “glitch”
- Glitch – an unwanted switching at the outputs
  - occur because electrical paths through the circuit experience different delays
  - can be dangerous if a subsequent logic circuit “makes a decision” based on a glitch
- Solutions
  - wait until all signals are stable (let glitches occur and die away)
    - use a synchronizing signal – a “clock”
  - design hazard-free circuits
    - useful, but synchronization with a clock is generally better in the long run
    - we’ll see this later with sequential circuits

# Working with combinational logic summary

- Simplification
  - a start at understanding two-level simplification
- 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
- Later
  - combinational logic technologies
  - more design case studies