CH-3: Working With Combinational Logic

Contemporary Logic Design

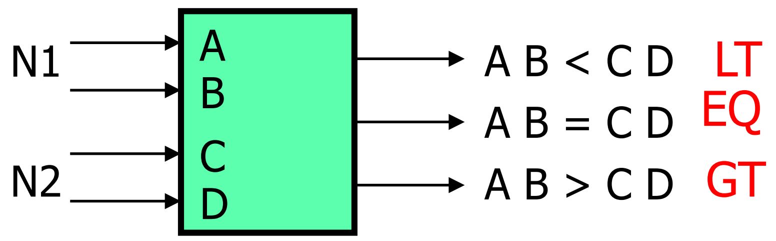
YONSEI UNIVERSITY

Fall 2016

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 (HDL)

Design Example: Two-bit Comparator

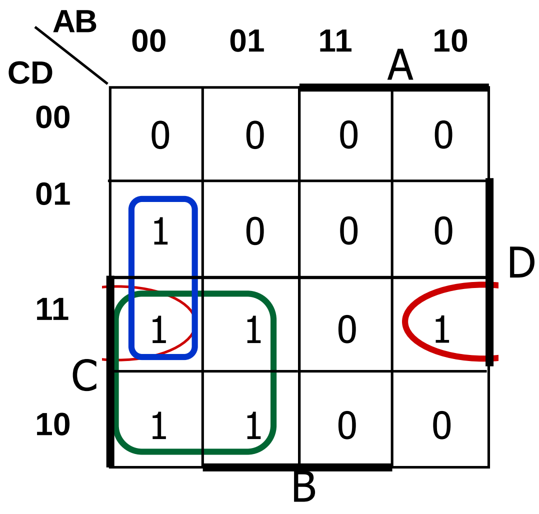


A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

Block diagram
and
truth table

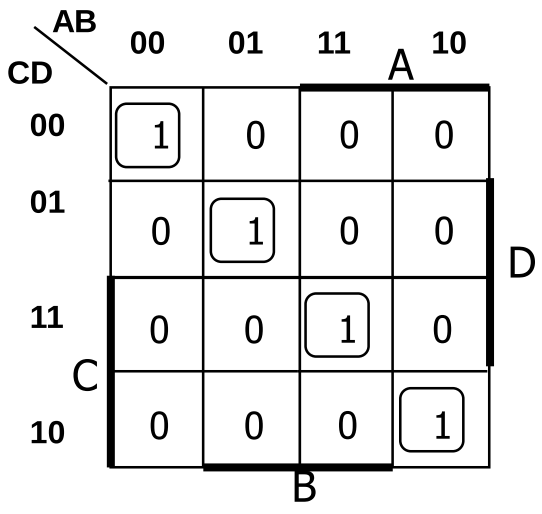
we'll need a 4-variable Karnaugh map
for each of the 3 output functions

Design Example: Two-bit Comparator



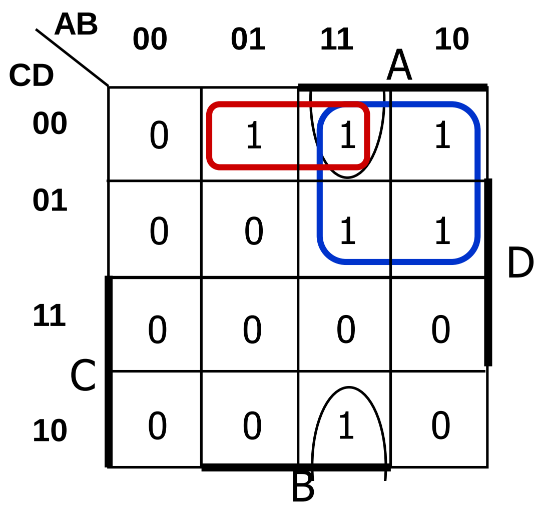
CD \ AB	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

K-map for LT



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

K-map for EQ



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

K-map for GT

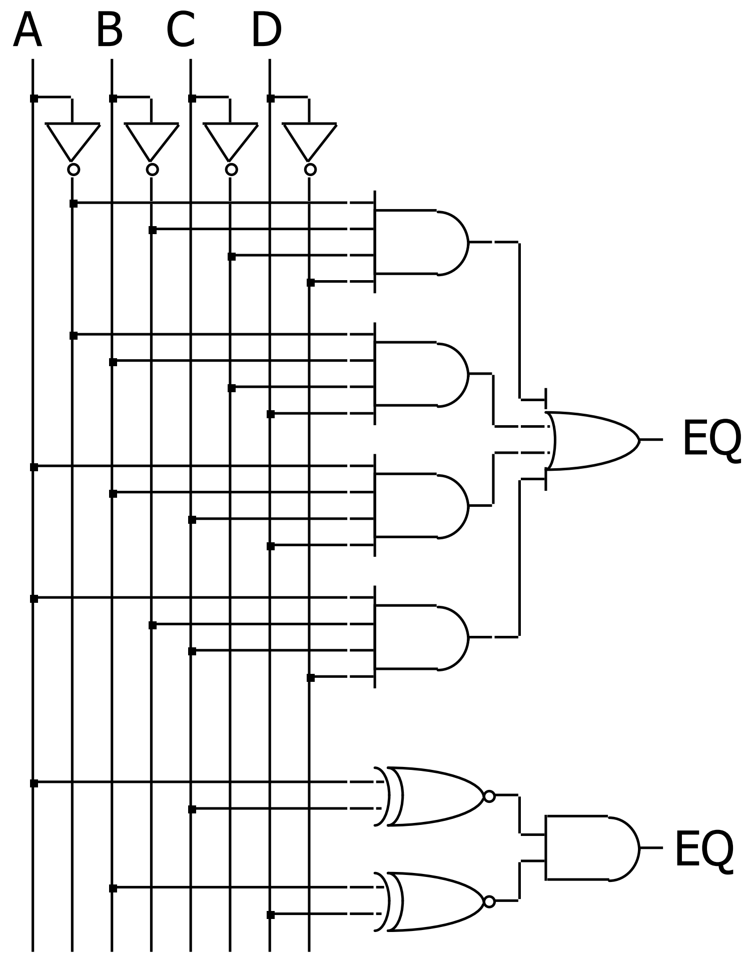
LT = A' B' D + A' C + B' C D

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

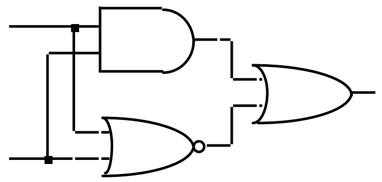
GT = B C' D' + A C' + A B D'

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

Design Example: Two-bit Comparator

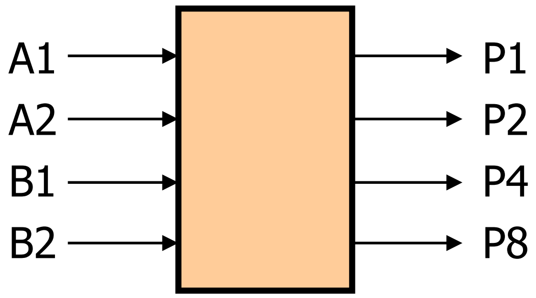


two alternative implementations of EQ with and without XOR



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

K-map for P8

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

P8 = A2A1B2B1

K-map for P4

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

P4 = A2B2B1'
+ A2A1'B2

K-map for P2

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

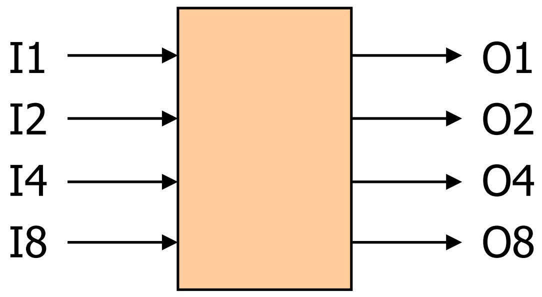
P2 = A2'A1B2
+ A1B2B1'
+ A2B2'B1
+ A2A1'B1

K-map for P1

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

P1 = A1B1

Design Example: BCD Increment by 1

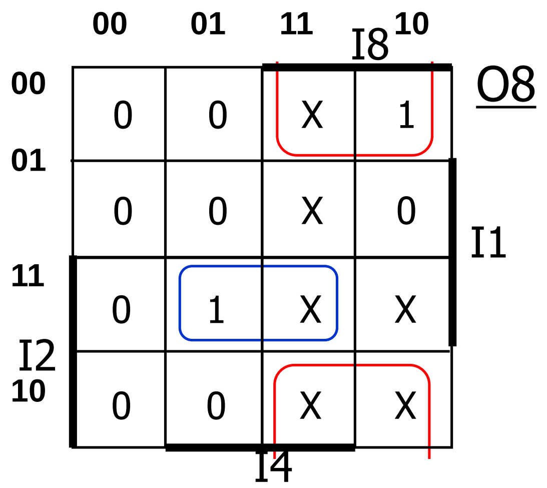


Block diagram
and
truth table

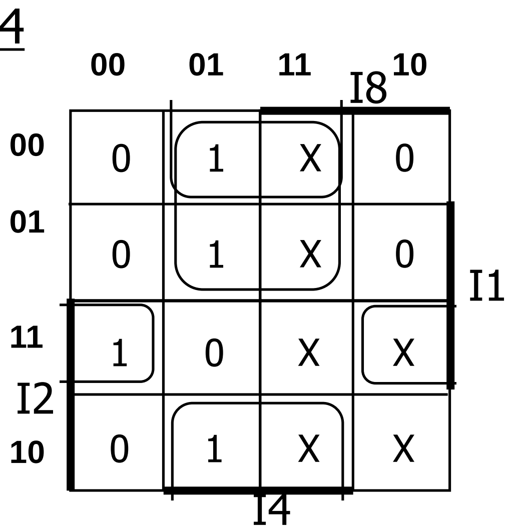
I8	I4	I2	I1	O8	O4	O2	O1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

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

Design Example: BCD Increment by 1



O8	00	01	11	10
00	0	0	X	1
01	0	0	X	0
11	0	1	X	X
10	0	0	X	X



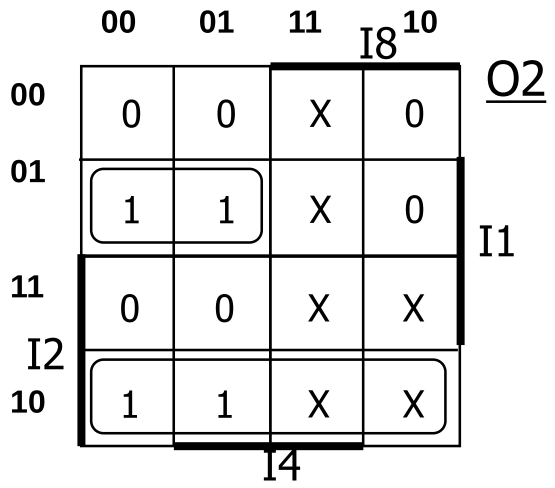
O4	00	01	11	10
00	0	1	X	0
01	0	1	X	0
11	1	0	X	X
10	0	1	X	X

O8 = I4 I2 I1 + I8 I1'

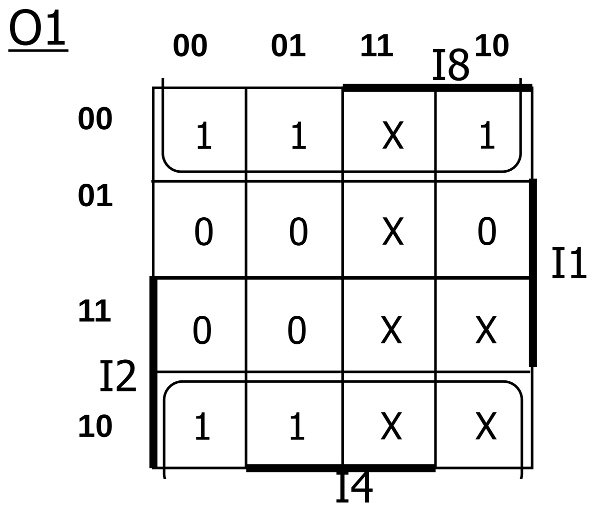
O4 = I4 I2' + I4 I1' + I4’ I2 I1

O2 = I8’ I2’ I1 + I2 I1'

O1 = I1'



O2	00	01	11	10
00	0	0	X	0
01	1	1	X	0
11	0	0	X	X
10	1	1	X	X

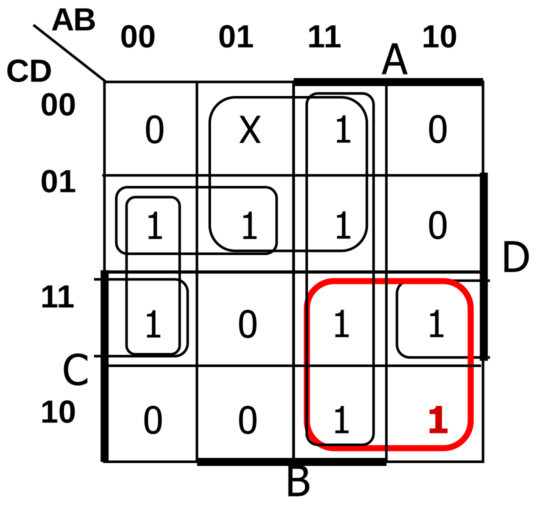


O1	00	01	11	10
00	1	1	X	1
01	0	0	X	0
11	0	0	X	X
10	1	1	X	X

Defining Terms for 2-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 <u>can't be combined with another</u> to form a larger subcube
- <u>*Essential prime implicant*</u>
  - prime implicant is *essential* <u>if it alone covers an element of ON-set</u>
  - 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)
  - <u>cover the ON-set with as few prime implicants as possible</u>
    (minimize number of product terms)

Examples to Illustrate Terms



6 prime implicants:

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

essential

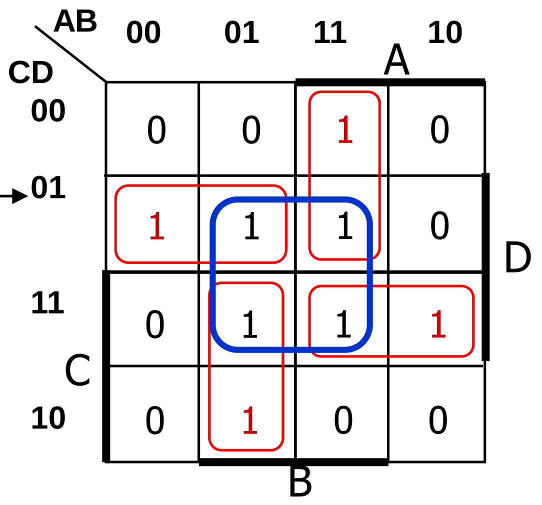
minimum cover: AC + BC' + A'B'D

5 prime implicants:

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

essential

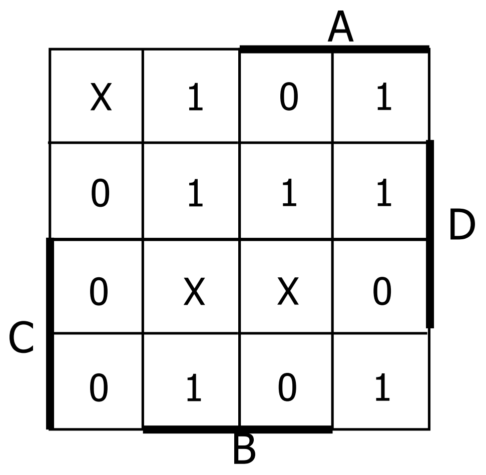
minimum cover: 4 essential implicants

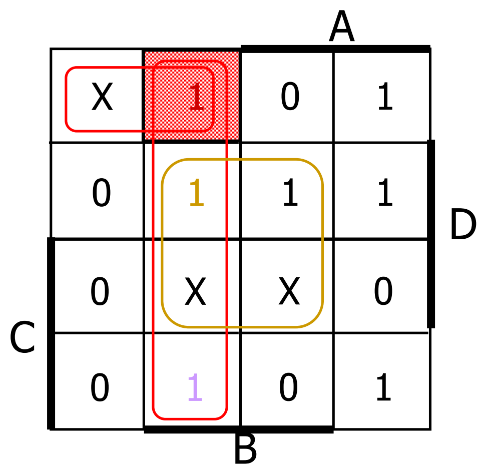


Algorithm for Two-level Simplification

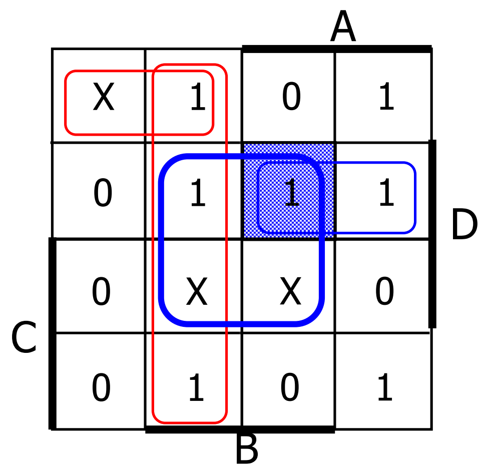
- Algorithm: minimum sum-of-products expression from a Karnaugh map
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number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

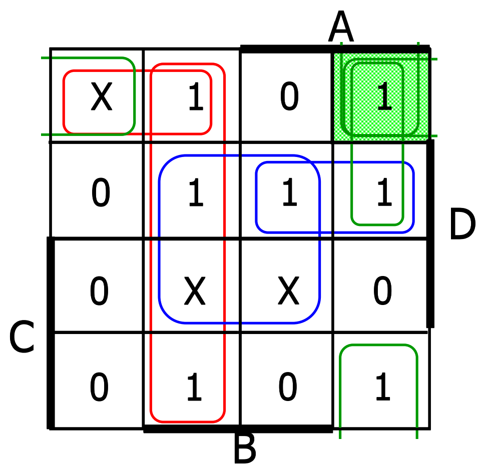




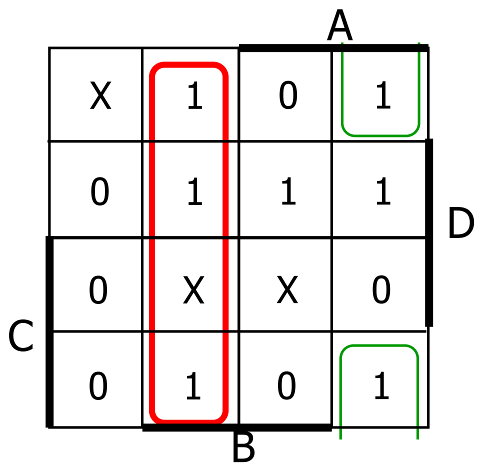
2 primes around A'BC'D'



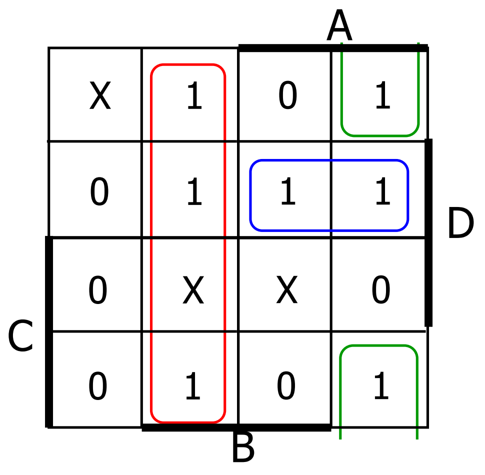
2 primes around ABC'D



3 primes around AB'C'D'



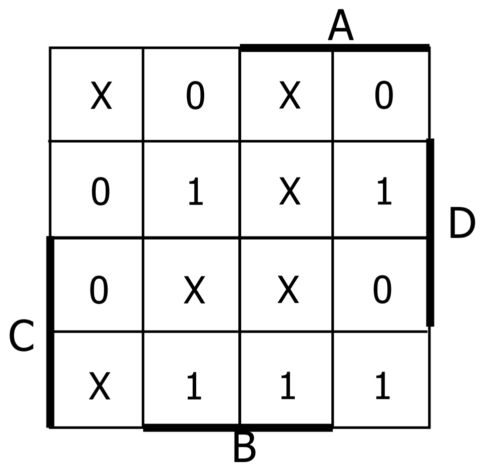
2 essential primes



minimum cover (3 primes)

Activity

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

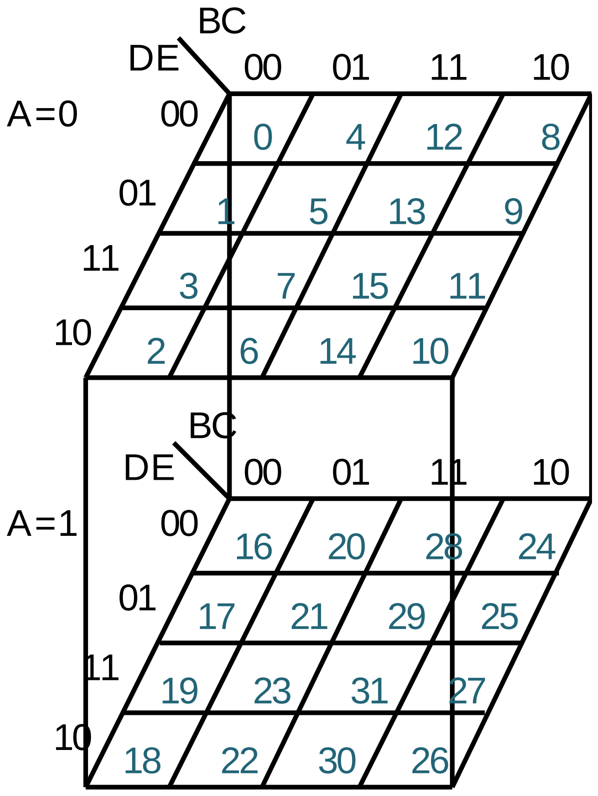


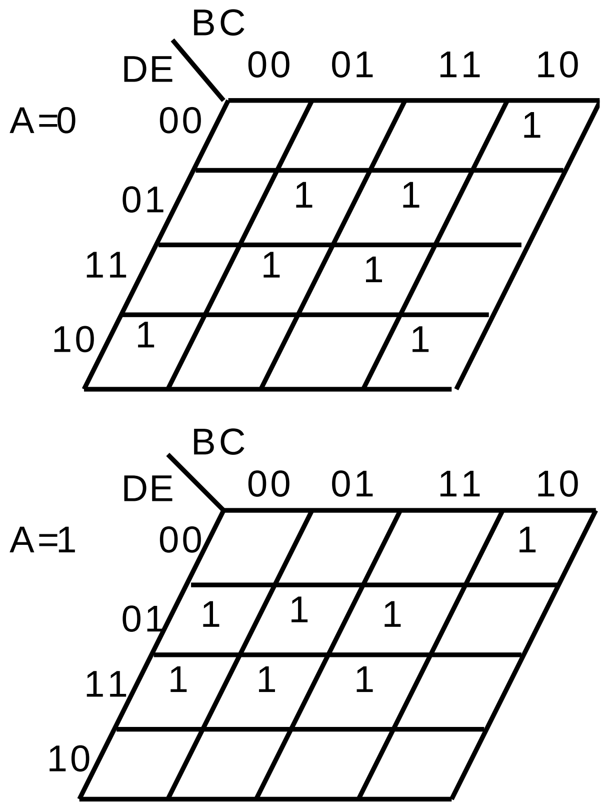
X	0	X	0
0	1	X	1
0	X	X	0
X	1	1	1

- Which are essential prime implicants?
- What is the minimum cover?

5 Variable K-Map

5-Variable K-maps



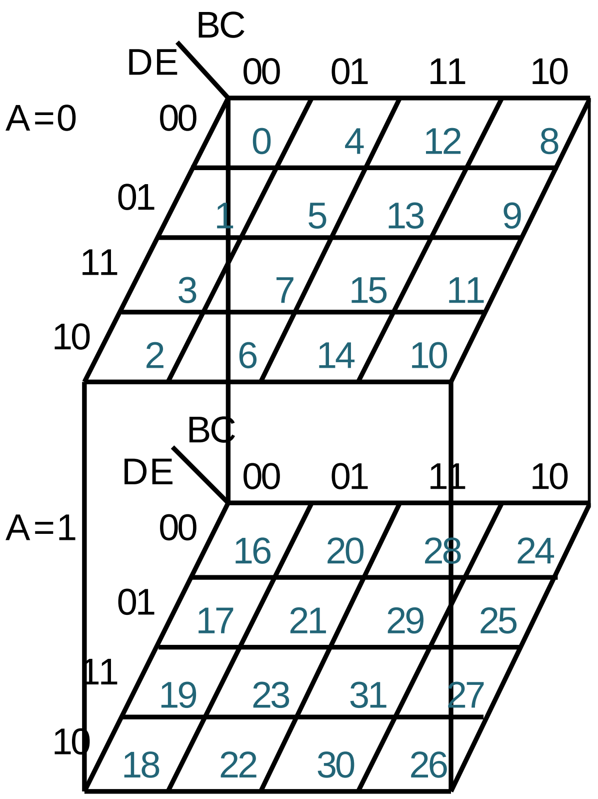


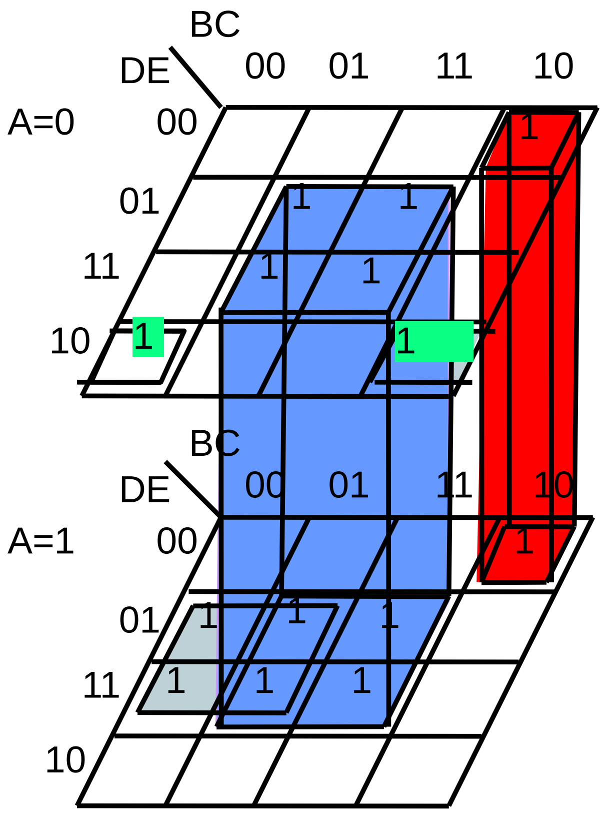
F(A,B,C,D,E) = Σm(2,5,7,8,10, 13,15,17,19,21,23,24,29 31)

=

5 Variable K-Map

5-Variable K-maps

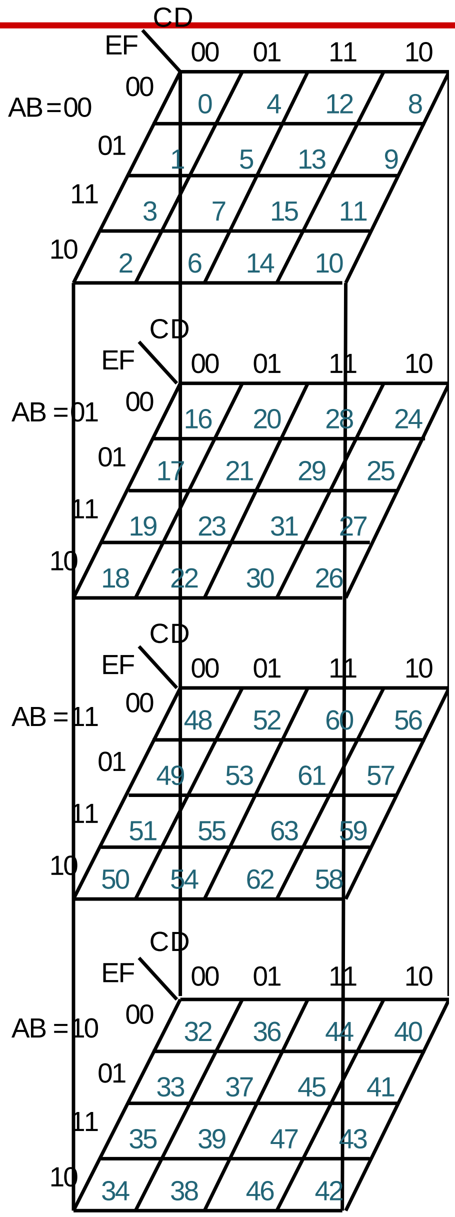




F(A,B,C,D,E) = Σm(2,5,7,8,10, 13,15,17,19,21,23,24,29 31)

= C E + A B' E + B C' D' E' + A' C' D E'

6 Variable K-Map

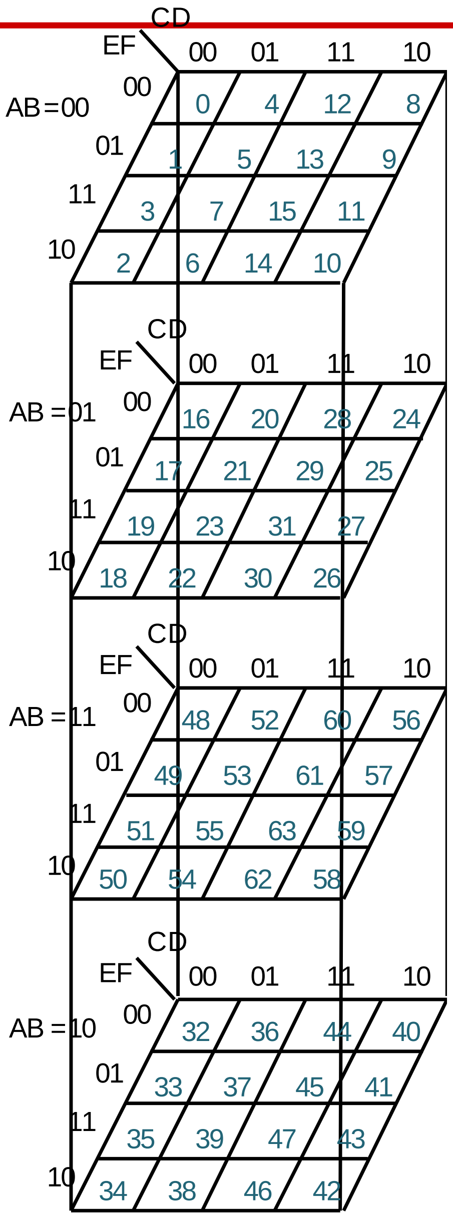


6- Variable K-Maps

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

=

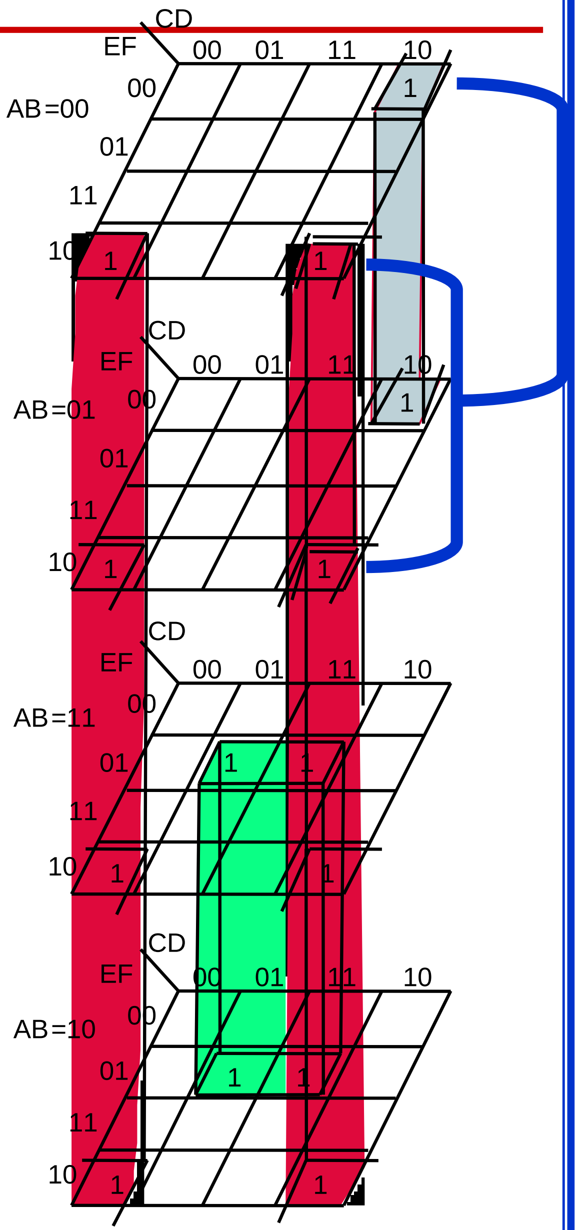
6 Variable K-Map



6- Variable K-Maps

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

= D' E F' + A D E' F + A' C D' F'



Gate Logic: CAD Tools for Simplification

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.

Implication Table

Column I	
0000	
0100	
1000	
0101	
0110	
1001	
1010	
0111	
1101	
1111	

Gate Logic: CAD Tools for Simplification

Quine-McCluskey Method

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

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	

Gate Logic: CAD Tools for Simplification

Quine-McCluskey Method

F(A,B,C,D) = Σm(4,5,6,8,9,10,13) + Σd(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.

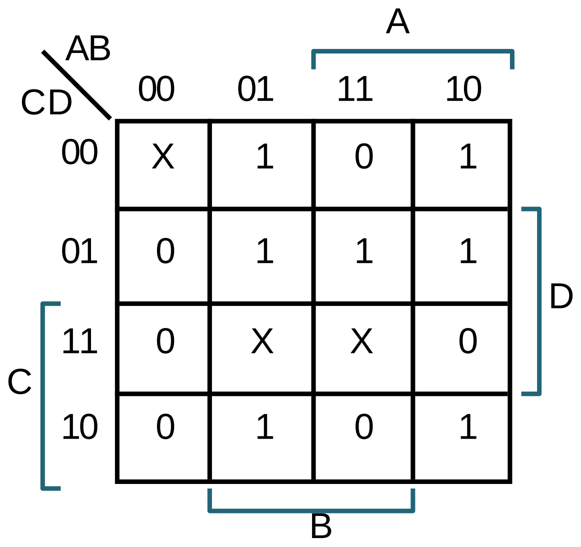
Implication Table

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

Repeat until no further combinations can be made.

Gate Logic: CAD Tools for Simplification

Quine-McCluskey Method Continued



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

Prime Implicants:

0-00 = A' C' D'

100- = A B' C'

1-01 = A C' D

-1-1 = B D

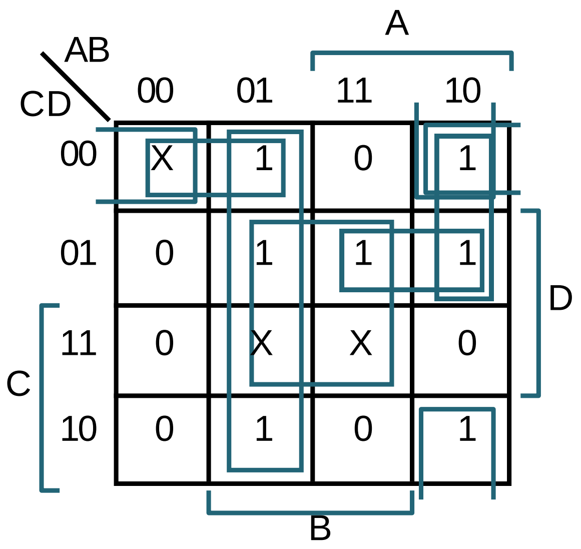
-000 = B' C' D'

10-0 = A B' D'

01-- = A' B

Gate Logic: CAD Tools for Simplification

Quine-McCluskey Method Continued



Prime Implicants:

0-00 = A' C' D'	**-000 = B' C' D'**
100- = A B' C'	**10-0 = A B' D'**
1-01 = A C' D	**01-- = A' B**
-1-1 = B D	

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

Gate Logic: CAD Tools for Simplification

Prime Implicant Chart

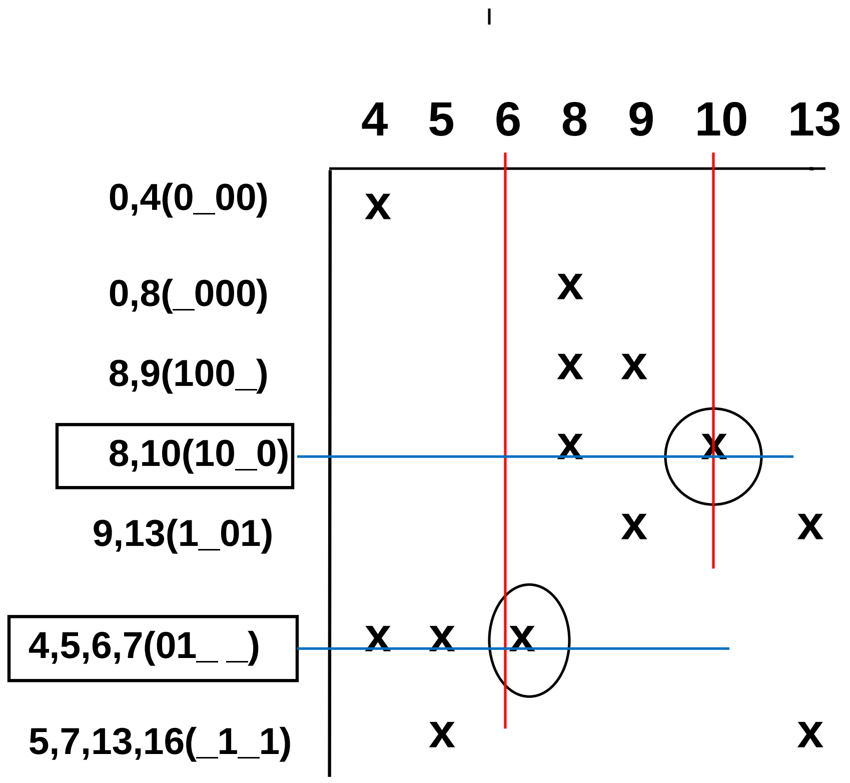
	4	5	6	8	9	10	13
0,4(0_00)	X						
0,8(_000)				X			
8,9(100_)				X	X		
8,10(10_0)				X		X	
9,13(1_01)					X		X
4,5,6,7(01_ _)	X	X	X				
5,7,13,16(_1_1)		X					X

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

Gate Logic: CAD Tools for Simplification

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				
5,7,13,16(_1_1)		X					X



	4	5	6	8	9	10	13
0,4(0_00)	X						
0,8(_000)				X			
8,9(100_)				X	X		
$\boxed{\text{8,10(10\_0)}}$				X		X	
9,13(1_01)					X		X
$\boxed{\text{4,5,6,7(01\_ \_)}}$	X	X	X				
5,7,13,16(_1_1)		X					X

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

Gate Logic: CAD Tools for Simplification

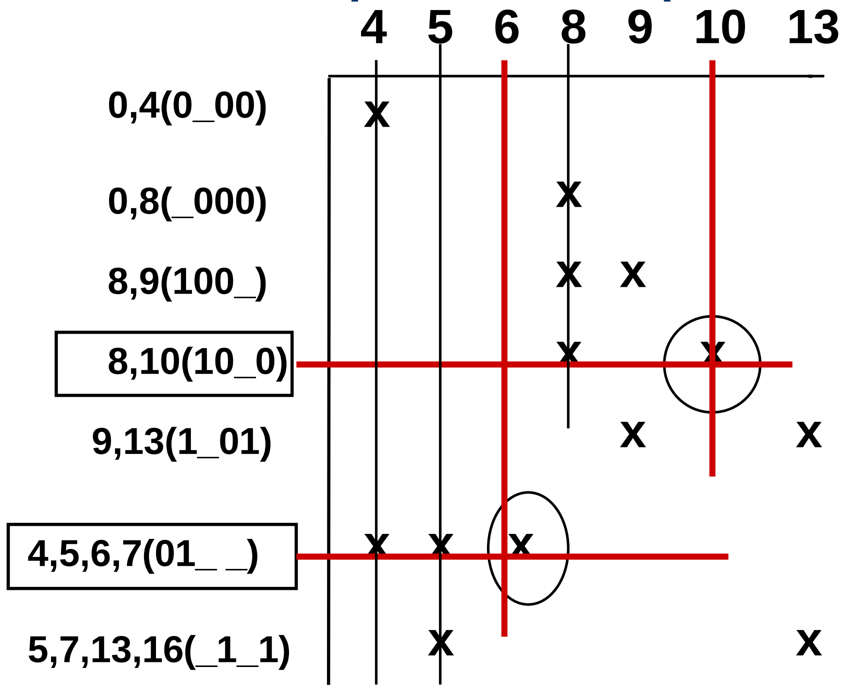
Prime Implicant Chart (Continued)

	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				
5,7,13,16(_1_1)		X					X

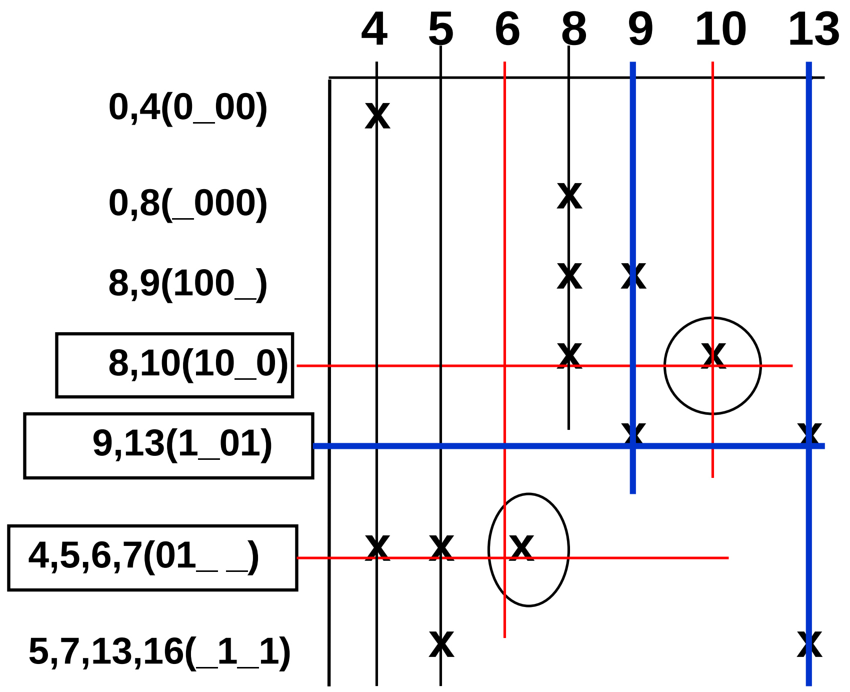
Eliminate all columns covered by essential primes

Gate Logic: CAD Tools for Simplification

Prime Implicant Chart (Continued)



	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				
5,7,13,16(_1_1)		X					X



	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				
5,7,13,16(_1_1)		X					X

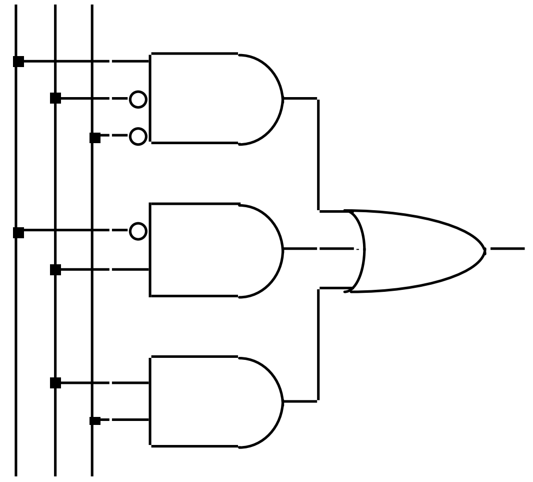
Eliminate all columns covered by essential primes

Find minimum set of rows that cover the remaining columns

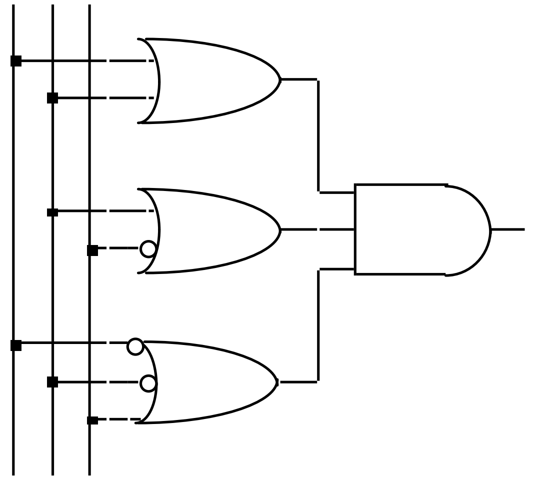
F= A B' D' + A C' D + A' B

Implementations of Two-level Logic

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

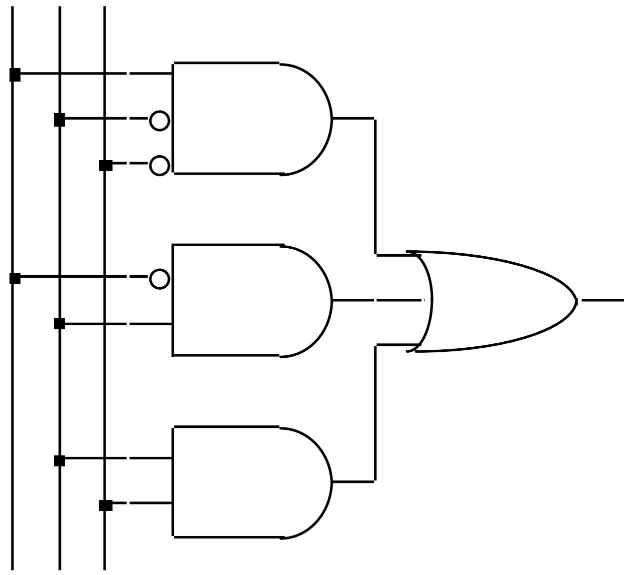


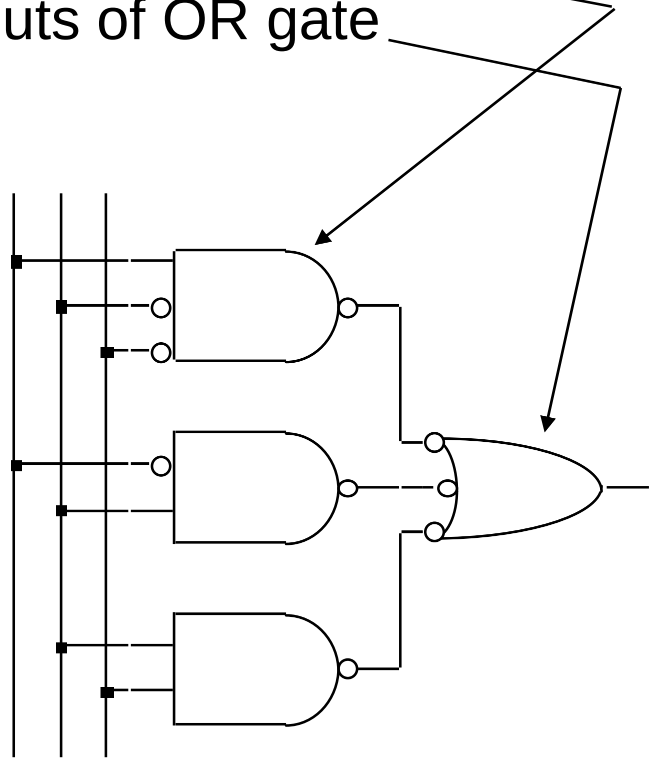
- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



Two-level Logic using NAND Gates

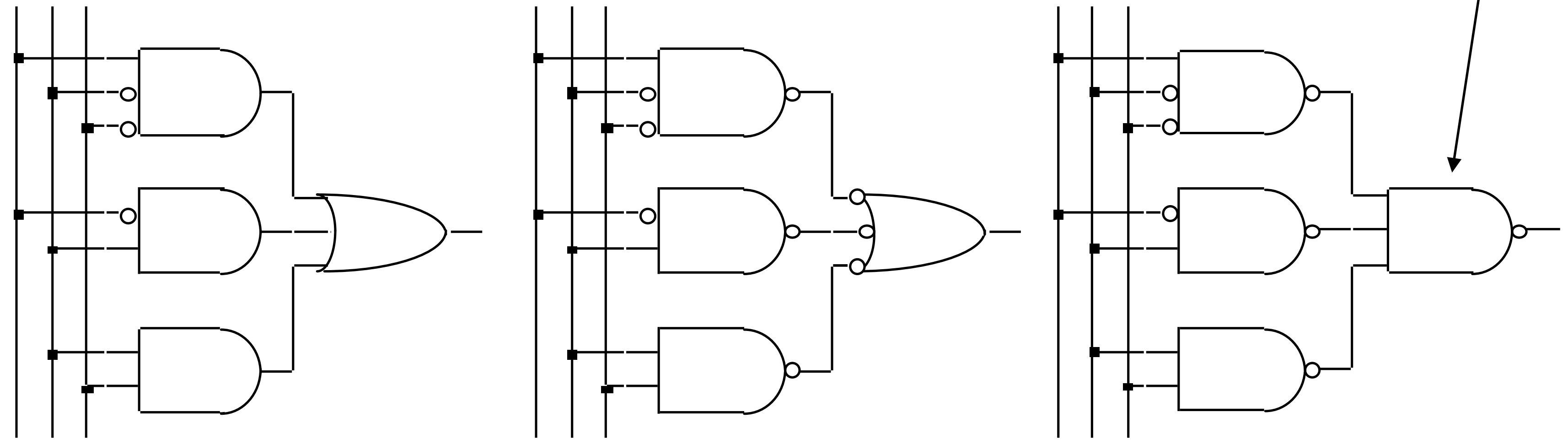
- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate





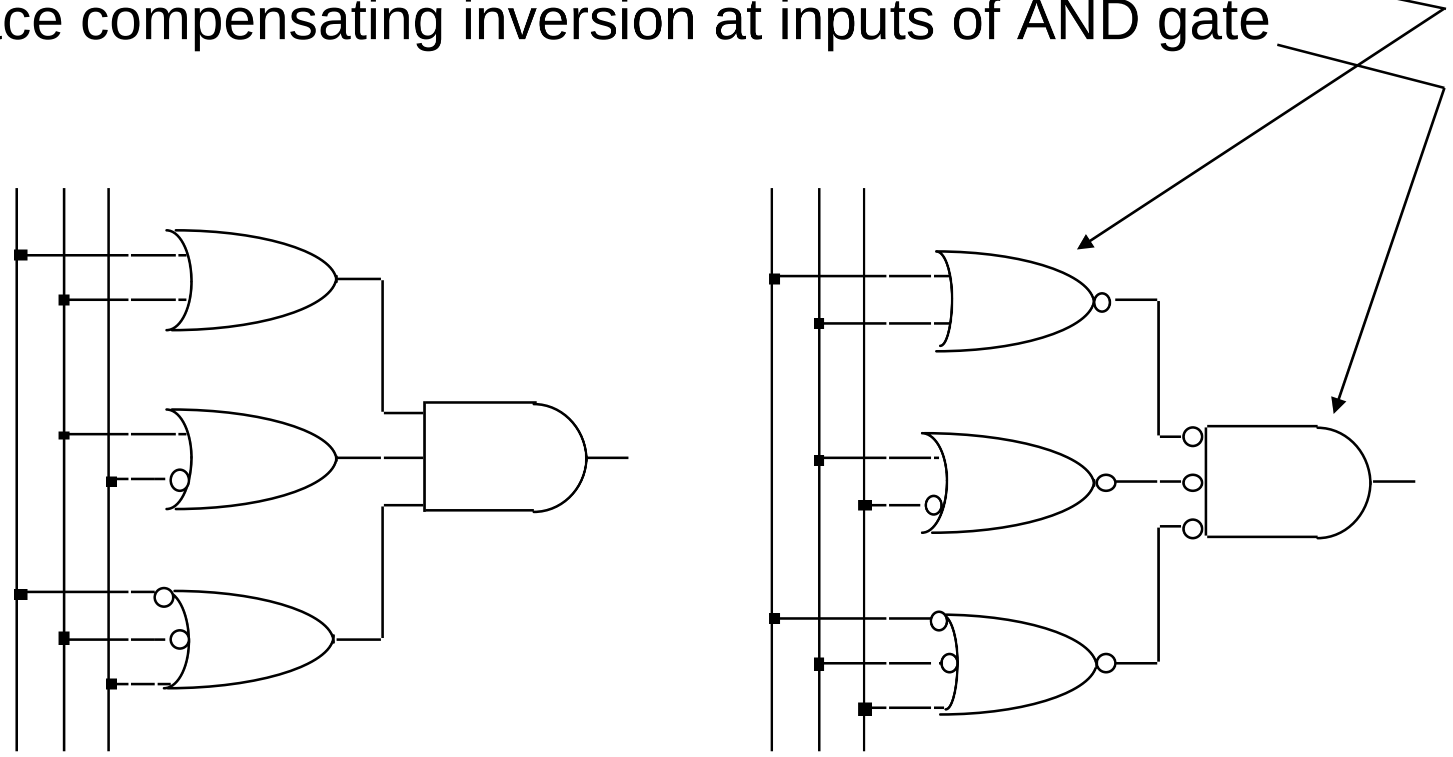
Two-level Logic using NAND Gates

- 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



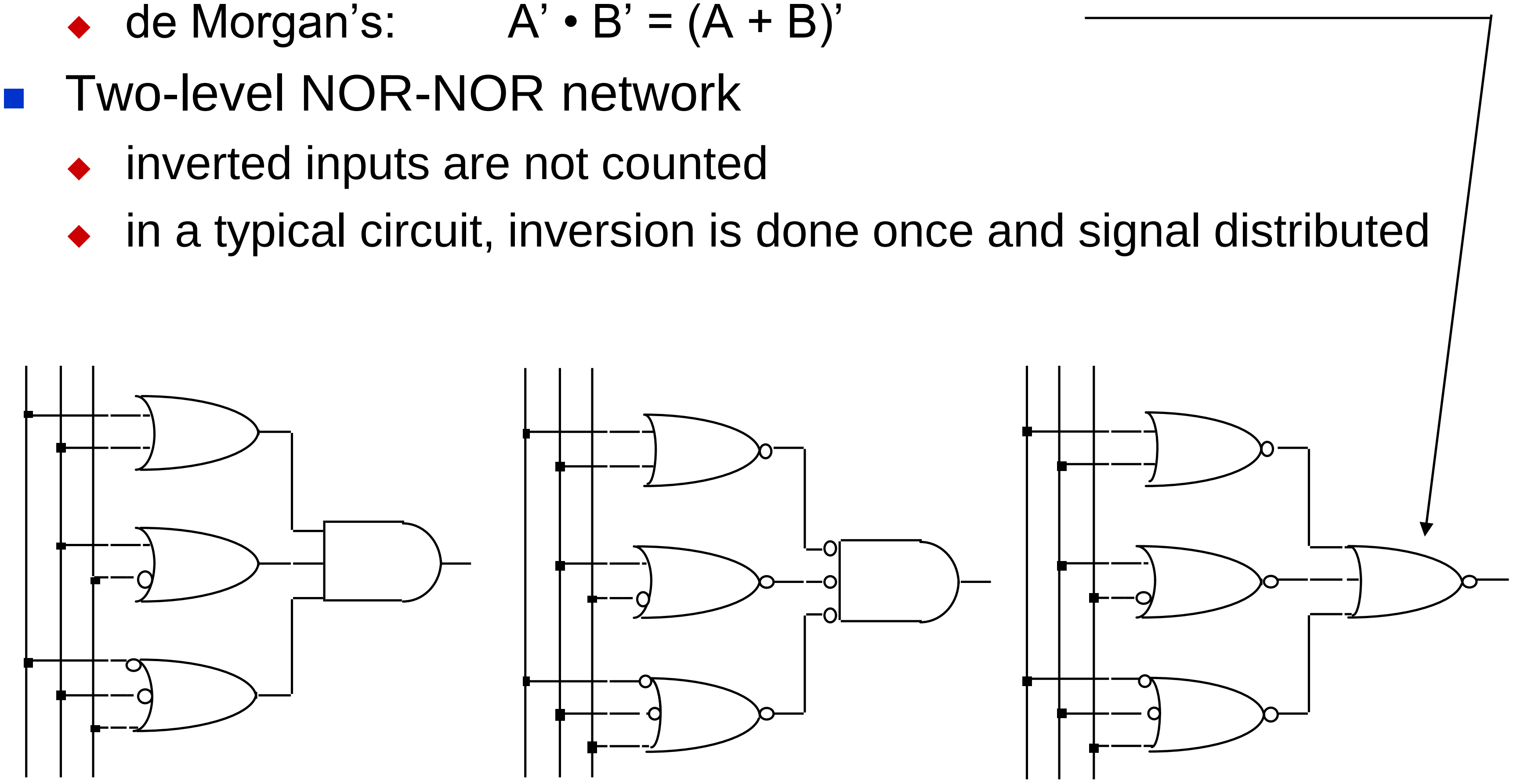
Two-level Logic using NOR Gates

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



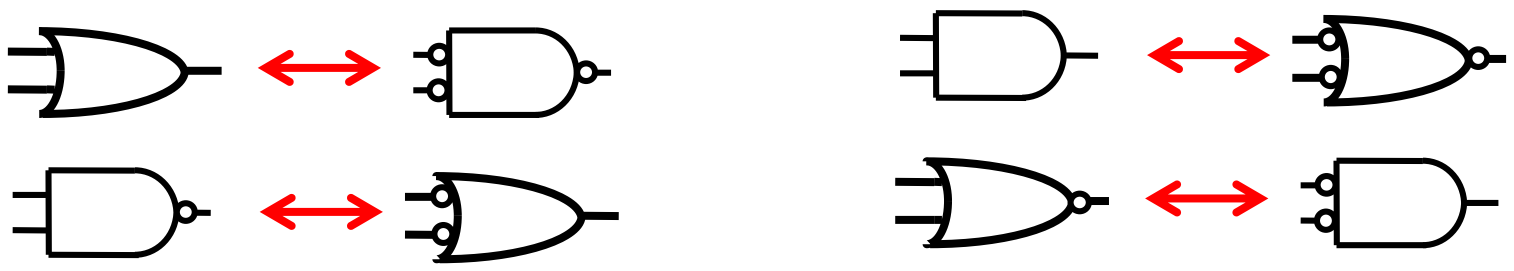
Two-level Logic using NOR Gates

- 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



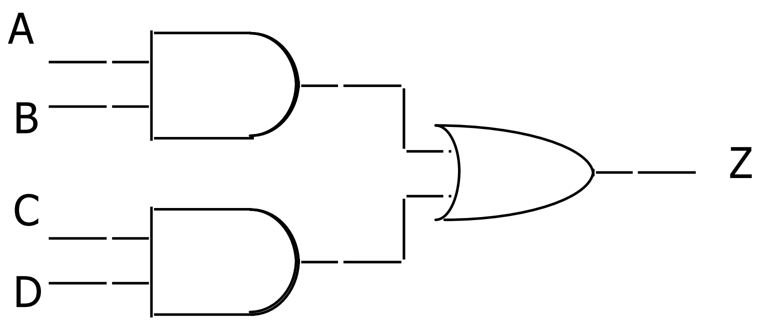
Two-level Logic using NAND/NOR Gates

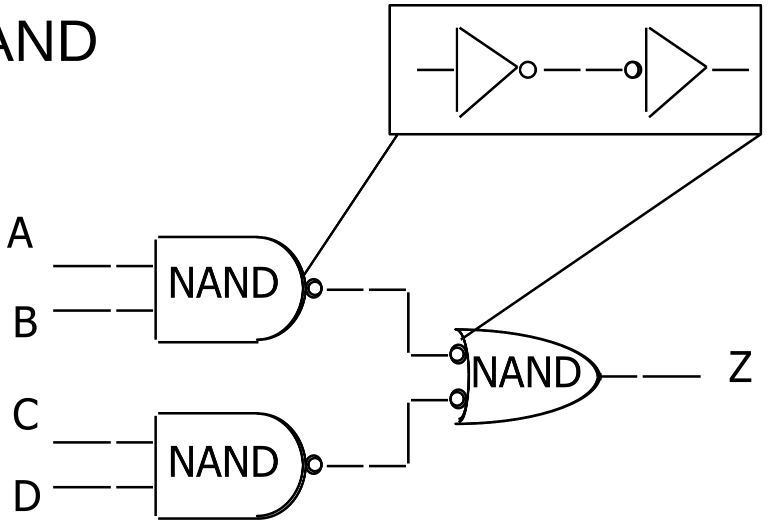
- NAND-NAND and NOR-NOR networks
  - de Morgan's law: (A + B)' = A' • B' (A • B)' = A' + B'
  - written differently: A + B = (A' • B')' (A • B) = (A' + B')'
- In other words —
  - OR is the same as NAND with complemented inputs
  - AND is the same as NOR with complemented inputs
  - NAND is the same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



Conversion Between Forms

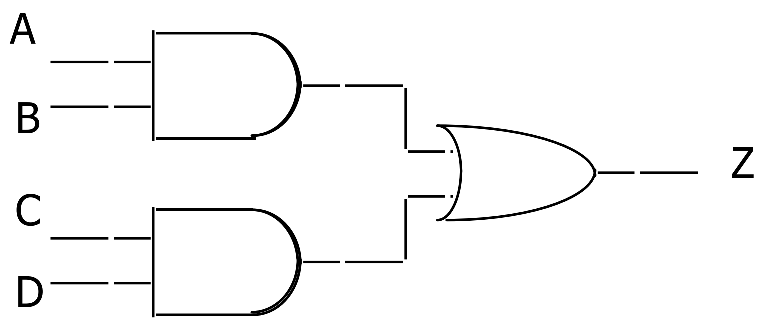
- Convert from networks of ANDs and ORs to networks of NANDs and NORs
  - introduce appropriate inversions ("bubbles")
- Each introduced "bubble" must be *matched* by a corresponding "bubble"
  - conservation of inversions
  - do not alter logic function
- Example: AND/OR to NAND/NAND

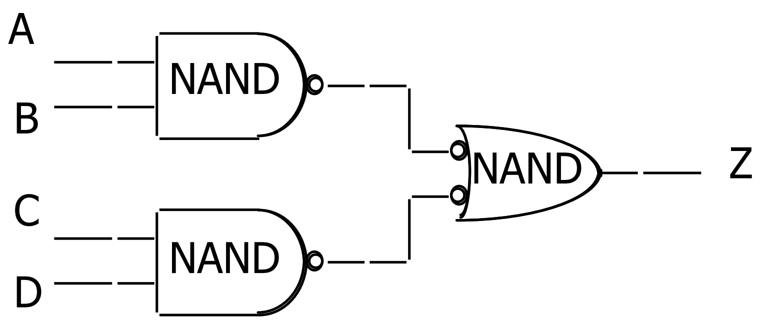




Conversion Between Forms

- Example: verify equivalence of two forms

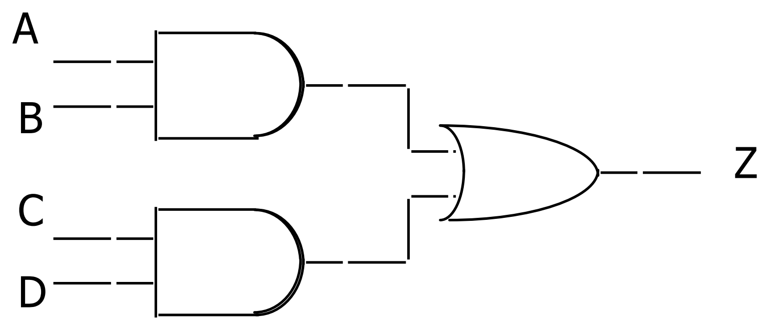


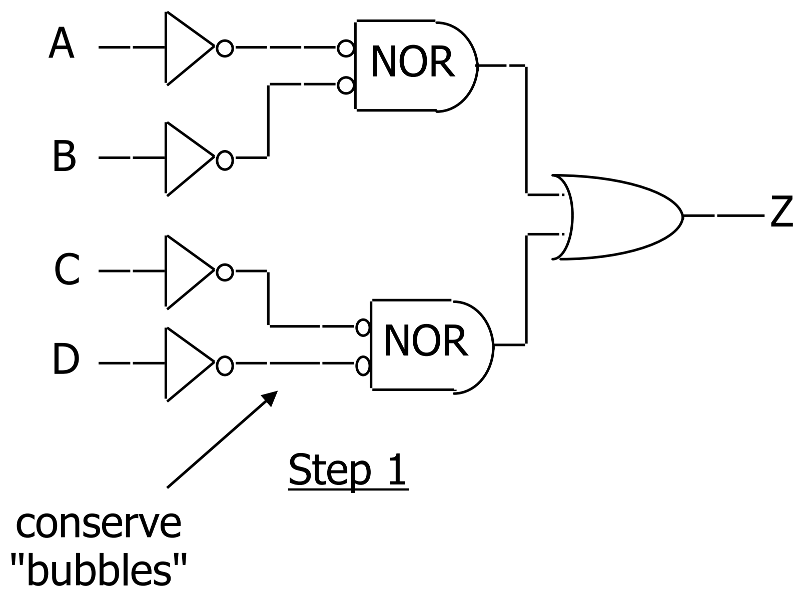


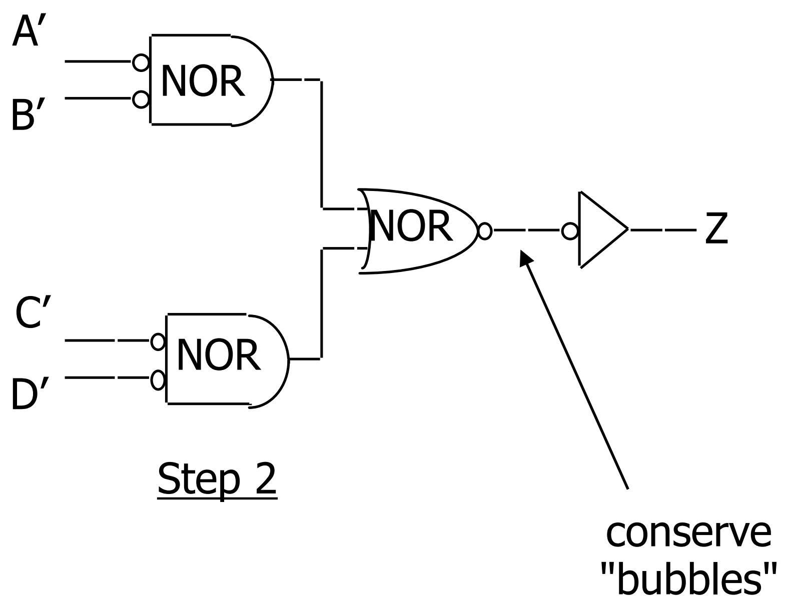
$$
\begin{aligned}
Z &= [\ (A \bullet B)' \bullet (C \bullet D)'\ ]' \\
&= [\ (A' + B') \bullet (C' + D')\ ]' \\
&= [\ (A' + B')' + (C' + D')'\ ] \\
&= (A \bullet B) + (C \bullet D) \ \checkmark
\end{aligned}
$$

Conversion Between Forms

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

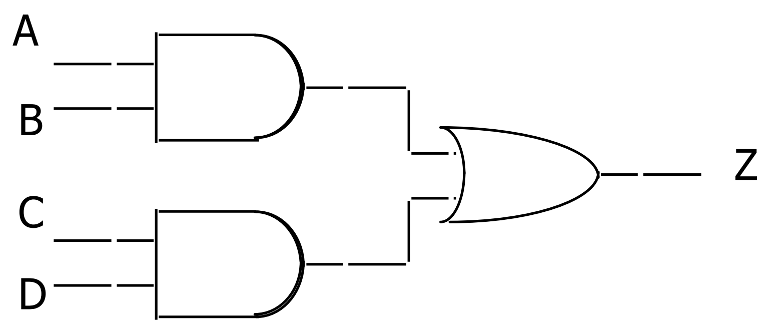


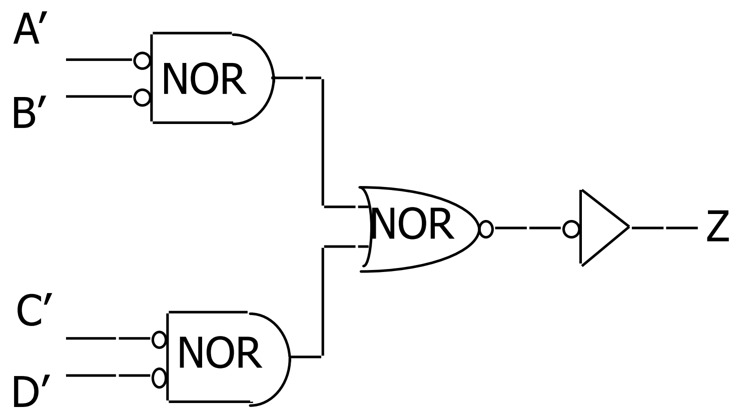




Conversion Between Forms

- Example: verify equivalence of two forms

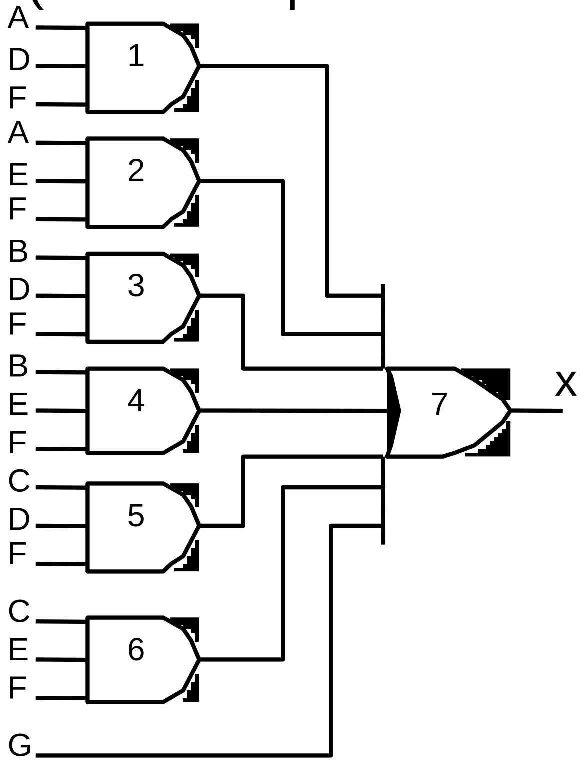




$$
\begin{aligned}
Z &= \{\ [\ (A' + B')' + (C' + D')'\ ]'\ \}' \\
&= \{\ (A' + B') \bullet (C' + D')\ \}' \\
&= (A' + B')' + (C' + D')' \\
&= (A \bullet B) + (C \bullet D) \quad \checkmark
\end{aligned}
$$

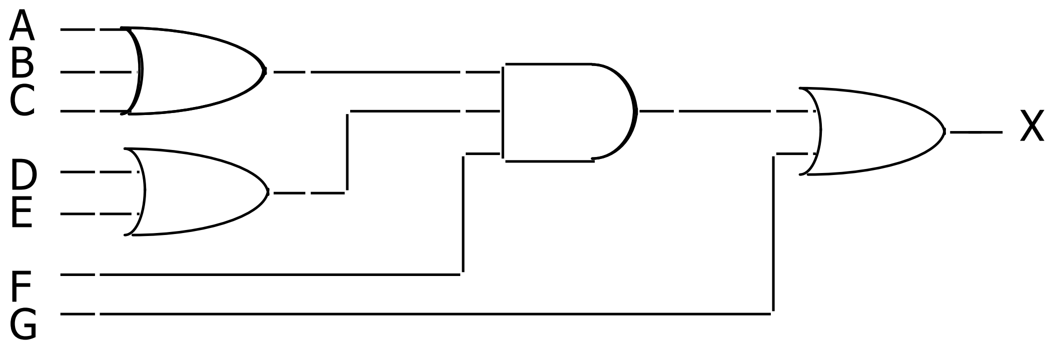
Multi-level Logic

- x = A D F + A E F + B D F + B E F + C D F + C E F + G = [(A+B+C)D + (A+B+C)E]F + G = (A+B+C)(D+E)F + G
  - 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)



Multi-level Logic

- x = A D F + A E F + B D F + B E F + C D F + C E F + G = [(A+B+C)D + (A+B+C)E]F + G = (A+B+C)(D+E)F + G
- 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)



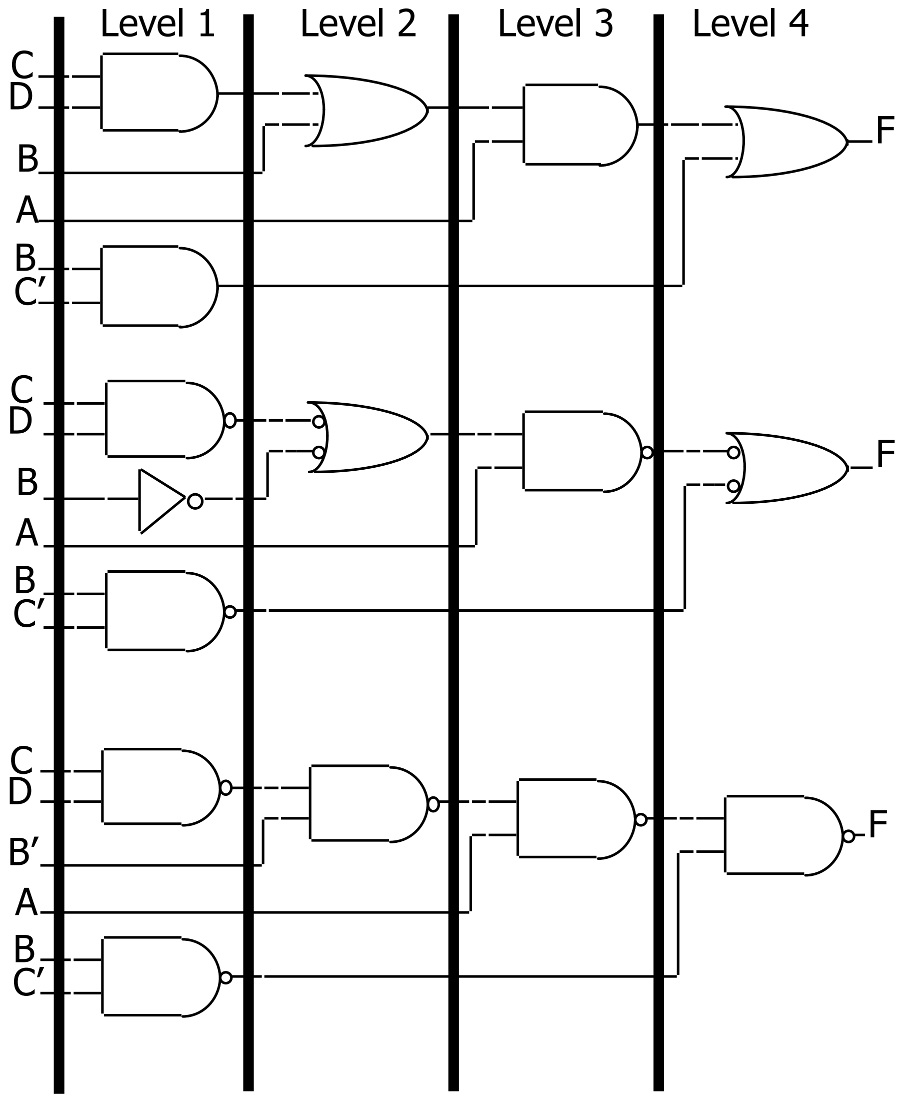
Multi-level Logic to NAND Gates

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

original AND-OR network

introduction and conservation of bubbles

redrawn in terms of conventional NAND gates



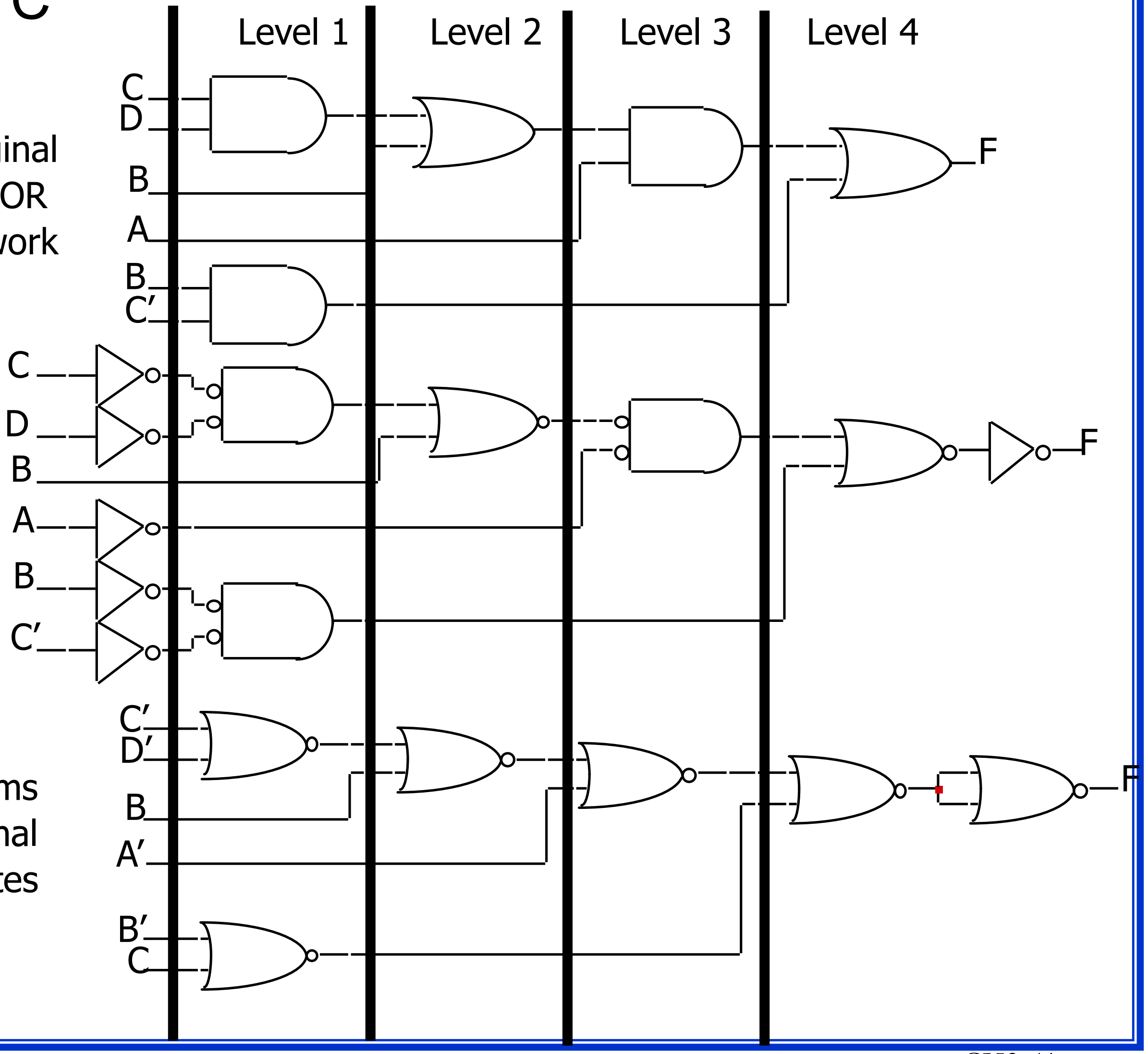
Conversion of Multi-level Logic to NORs

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

original AND-OR network

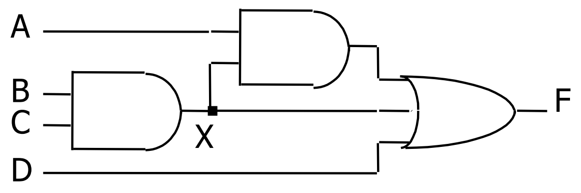
introduction and conservation of bubbles

redrawn in terms of conventional NOR gates

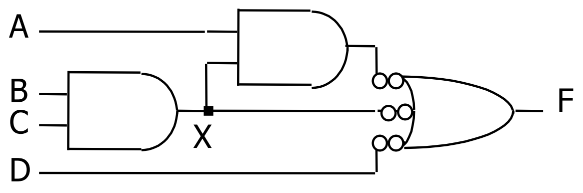


Conversion Between Forms

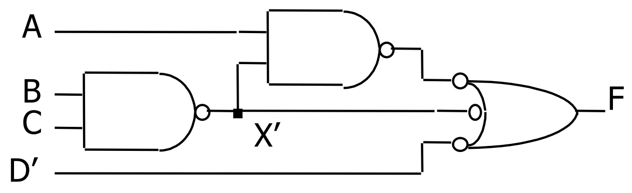
- Example



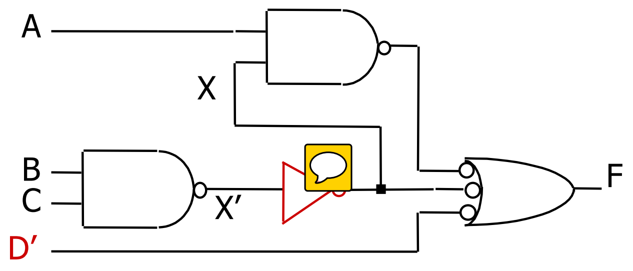
original circuit



add double bubbles to
invert all inputs of OR gate



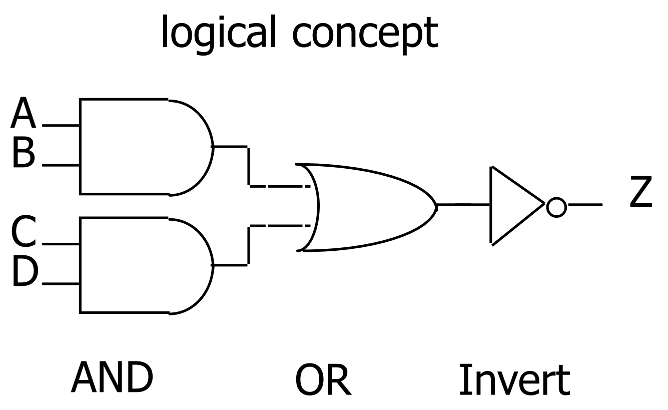
add double bubbles to
invert output of AND gate

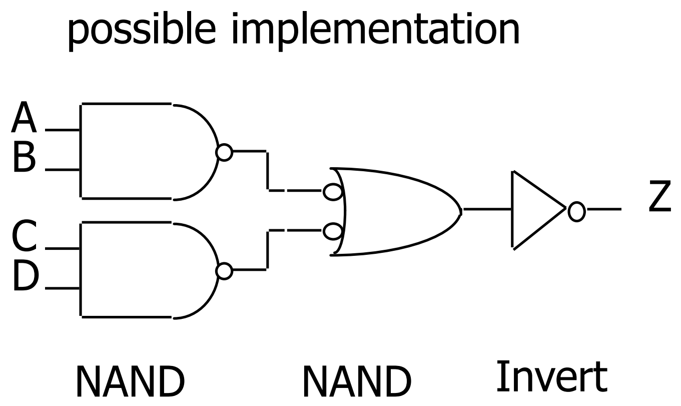


insert inverters to eliminate
double bubbles on a wire

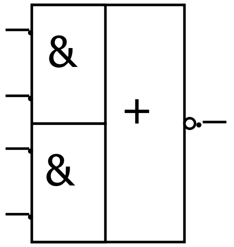
AND-OR-invert Gates

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

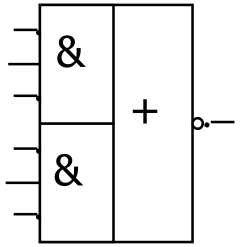




2x2 AOI gate symbol

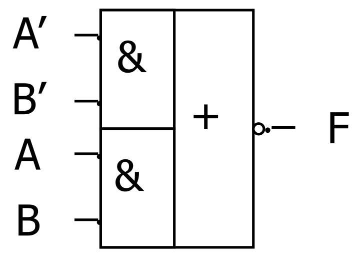


3x2 AOI gate symbol



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
  - F = A xor B = A’ B + A B’, F’ = XNOR = A’B’ + AB
  - 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

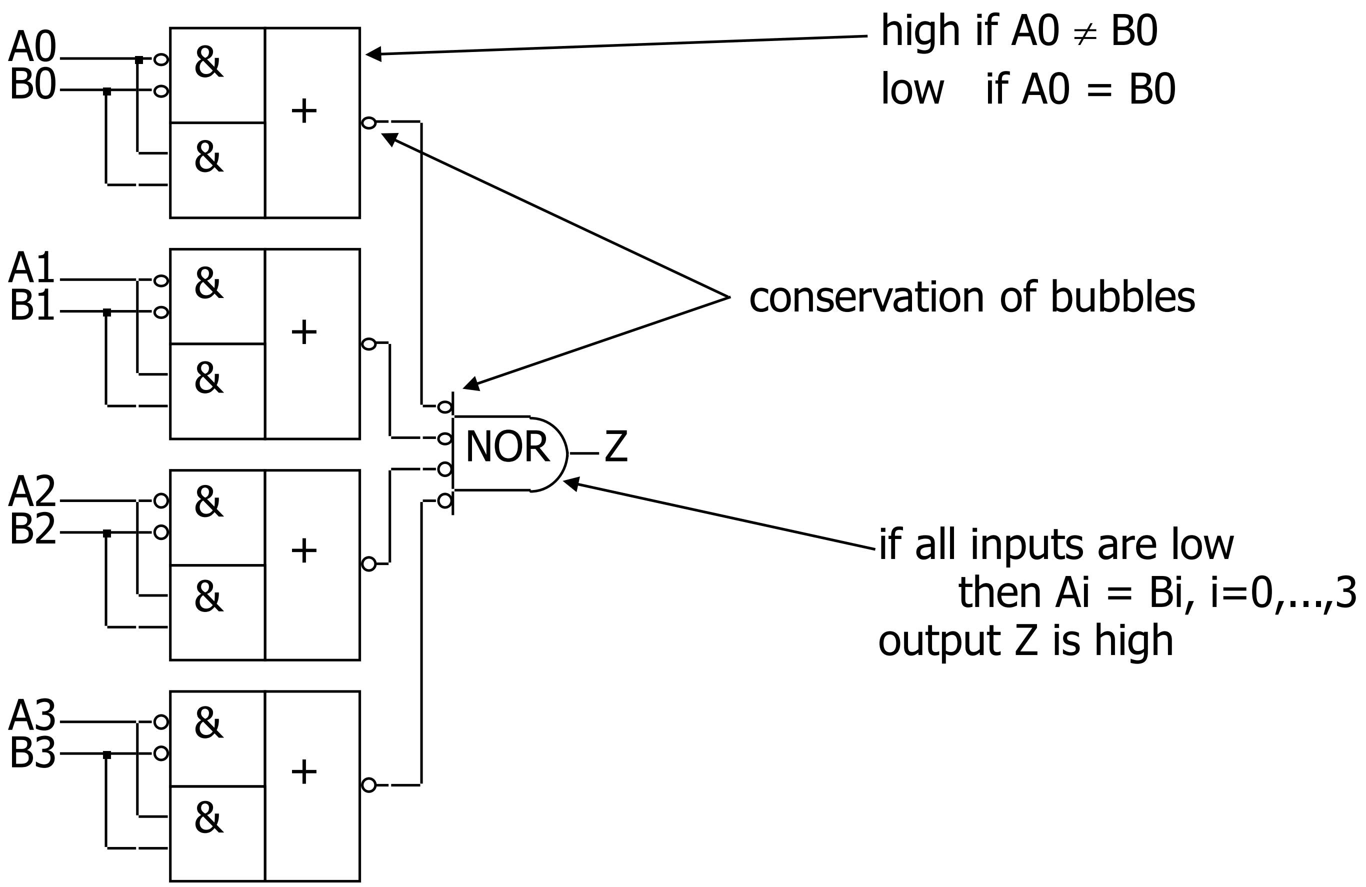
  - $F' = (A' + B') (A' + C) (B' + C)$
  - $F = [(A' + B') (A' + C) (B' + C)]'$
  - Implemented by 2-input 3-stack OAI gate

- 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

- Example: AOI implementation of 4-bit equality function



Summary for Multi-level Logic

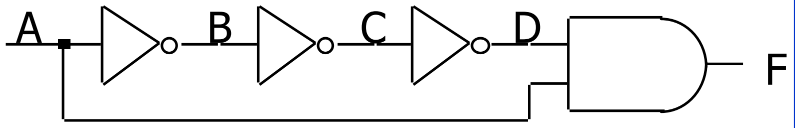
- Advantages
  - circuits may be smaller
  - gates have smaller fan-in
  - circuits may be faster
- Disadvantages
  - more difficult to design
  - tools for optimization are not as good as for two-level
  - analysis is more complex

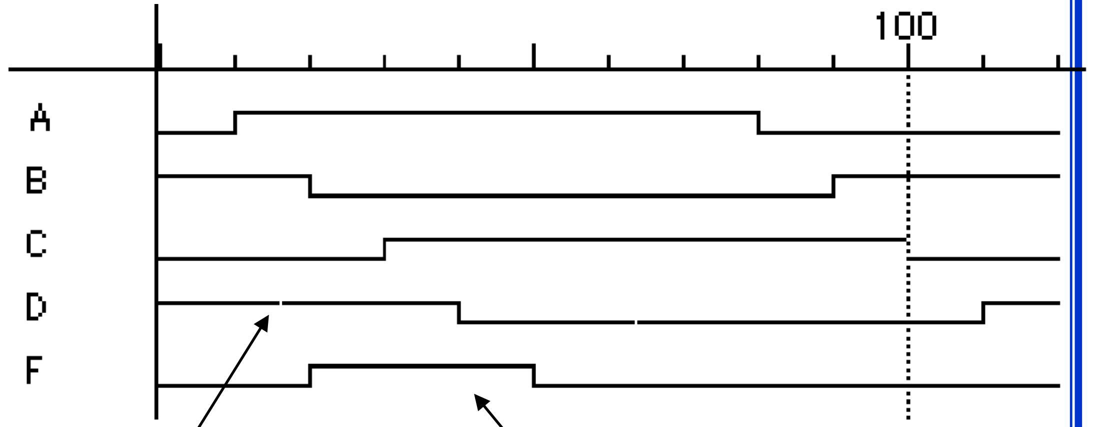
Time Behavior of Comb-Networks

- 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



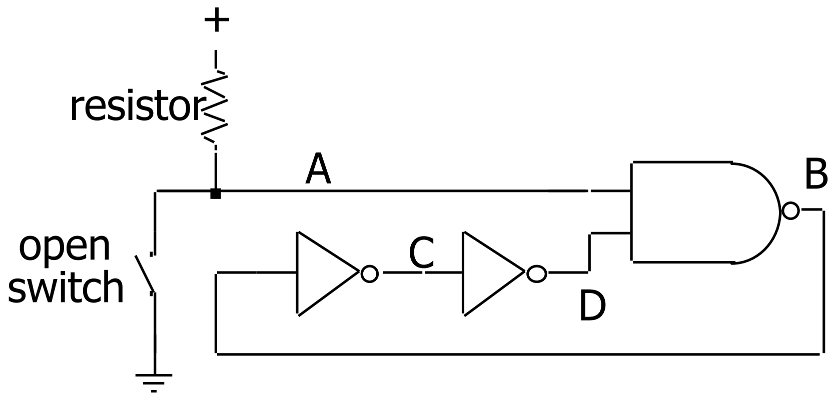


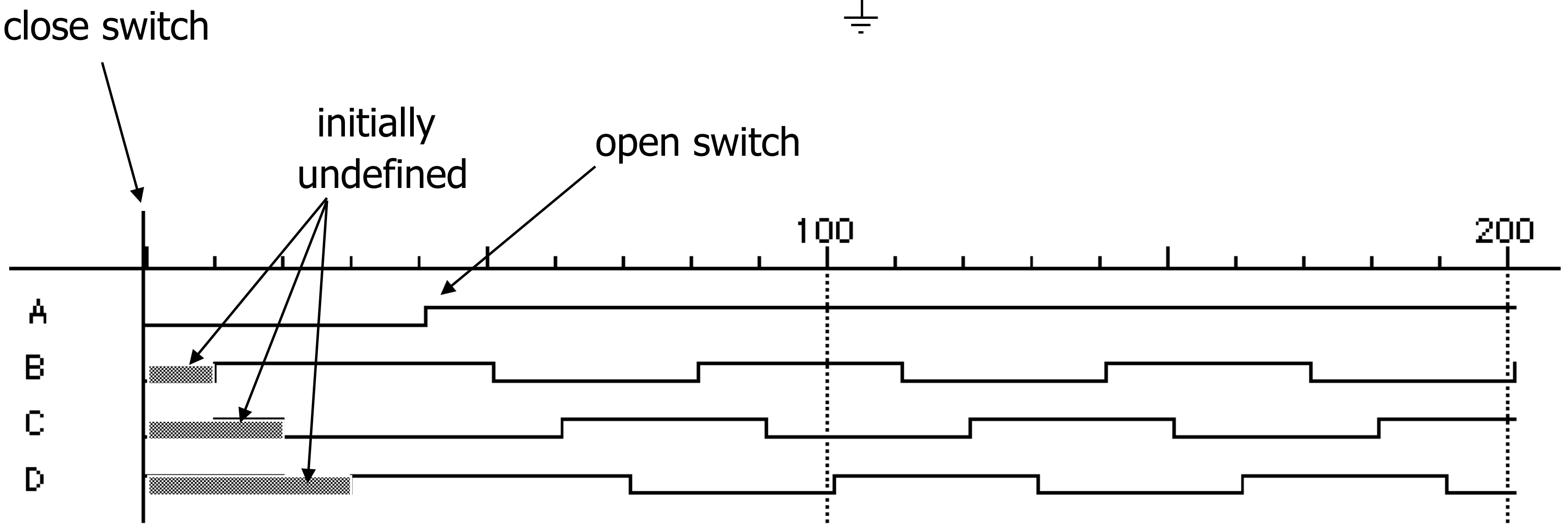
D remains high for three gate delays after A changes from low to high

F is not always 0 pulse 3 gate-delays wide

Oscillatory Behavior

- Another pulse shaping circuit

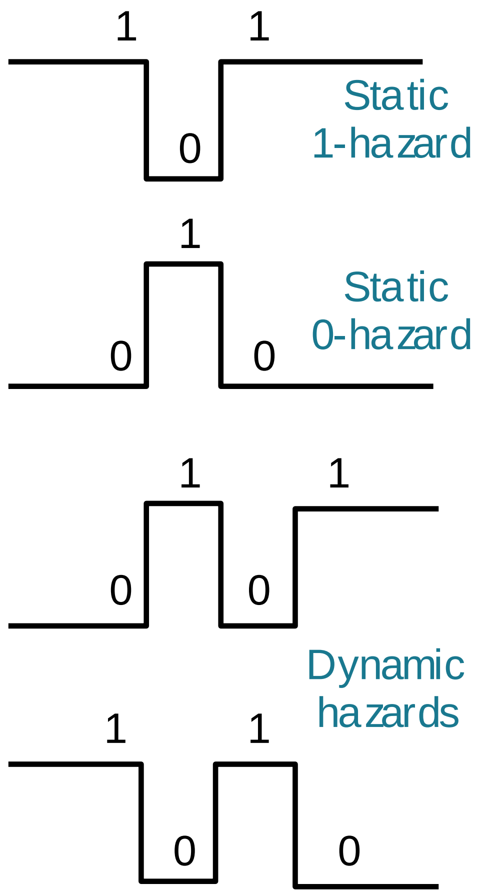




Hazards/Glitches and How to Avoid Them

- Unwanting switching at the outputs
- Occur because
  - <u>delay paths</u> through the circuit experience
  - <u>different propagation delays</u>
- Danger if logic "makes a decision" while output is unstable
  asynchronous input :these respond immediately to changes rather than waiting for a synchronizing signal called a *clock*)
- Solutions
  - wait until signals are <u>stable</u> (by using a clock)
  - <u>never use circuits with asynchronous inputs</u>
  - design <u>hazard-free circuits</u>

Hazards/Glitches and How to Avoid Them



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

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

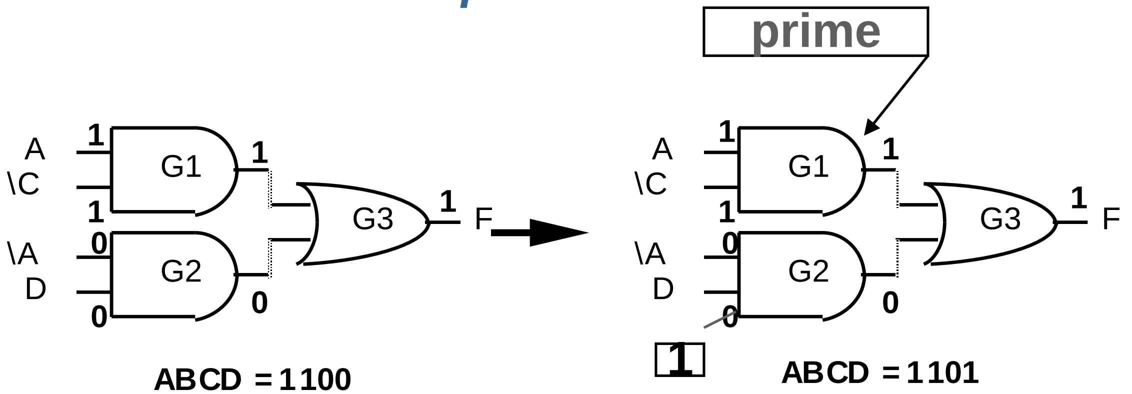
Input change causes a double change
from 0 to 1 to 0 to 1 OR
from 1 to 0 to 1 to 0

Kinds of Hazards

***Assumption*: the unexpected changes in the outputs are in response to <u>single-bit changes</u> in the inputs**

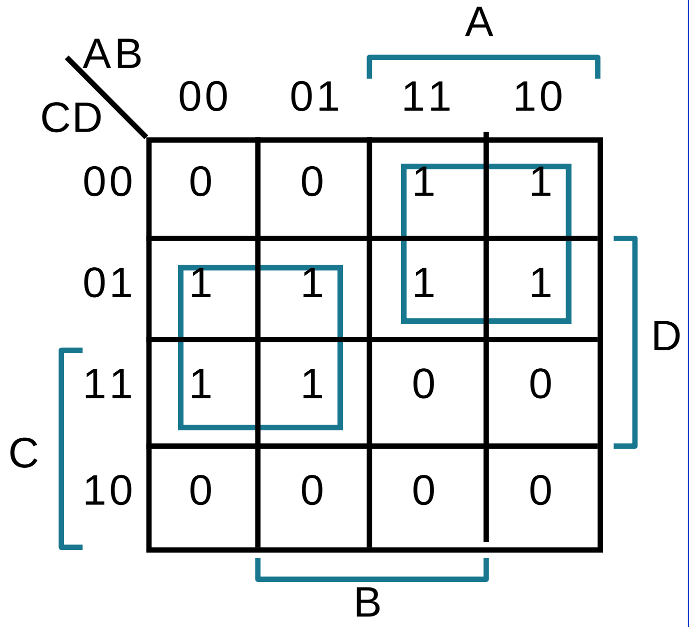
Hazards/Glitches and How to Avoid Them

Glitch Example



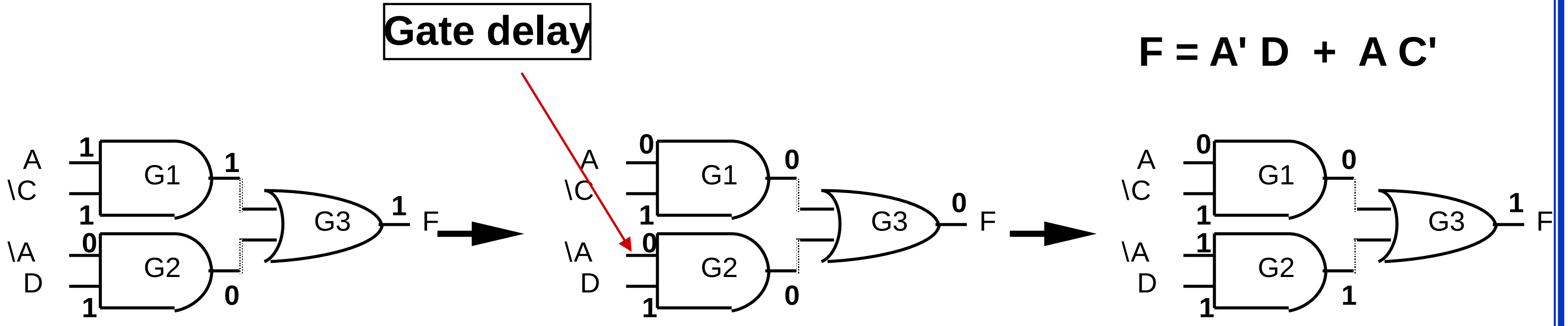
ABCD = 1100 ABCD = 1101

input change within product term



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

F = A' D + A C'



ABCD = 1101 ABCD = 0101 ($\overline{A}$ is still 0) ABCD = 0101 ($\overline{A}$ is 1)

Static 1-hazard

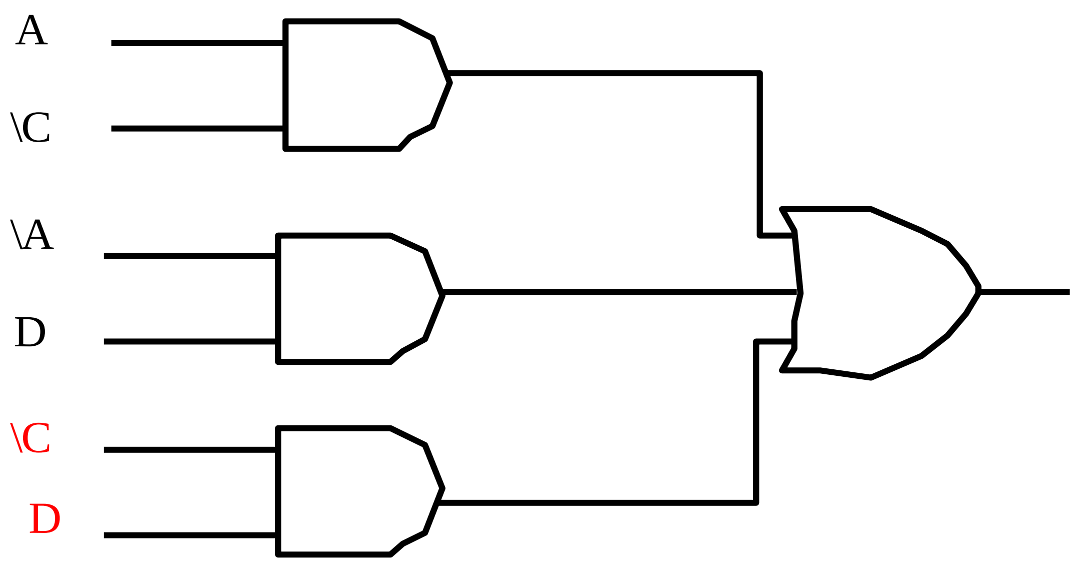
Hazards/Glitches and How to Avoid Them

- Static 1-hazard : occurs in sum of products form.
- Solution

General Strategy: *add redundant terms*

F = A' D + A C' becomes A' D + A C' + C' D

ABCD=1101-----------→ 0101, C'D=1 F=1



Hazards/Glitches and How to Avoid Them

- How about 0-hazard? Occurs in POS form

Re-express F in PoS form:

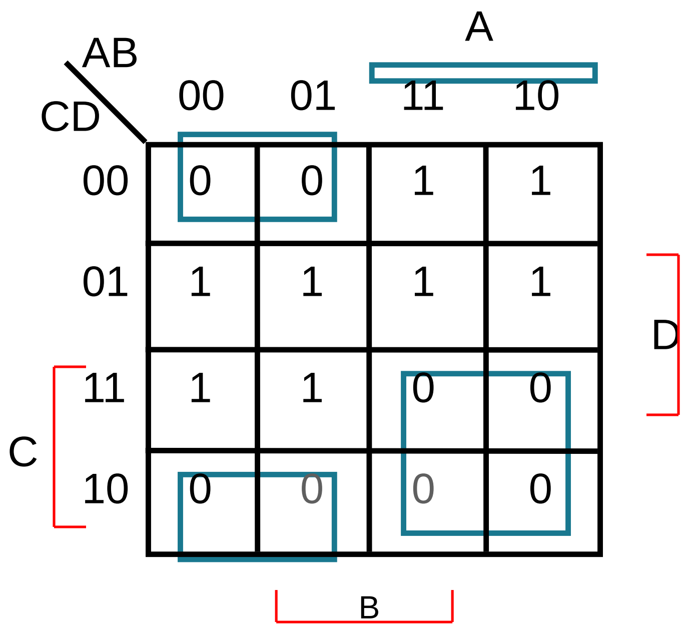
F’= AC+A’D’
F = (A' + C')(A + D)

Glitch present!: 0110 to 1110

Add term: (C' + D)

This expression is equivalent
to the hazard-free SoP form of F

All single-bit input changes should be
covered by one implicant



Hazards/Glitches and How to Avoid Them

- Start with expression that is free of static 1-hazards

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

Work with complement:

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

$$= (A' + C) (A + D') (C + D')$$

$$= A C + A C D' + C D' + A' C D' + A' D'$$

$$= A C + C D' + A' D'$$

covers all the adjacent 0's in the K-map

free of static-1 and static-0 hazards!

Hardware Description Languages (HDL)

- 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

Working with Comb-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