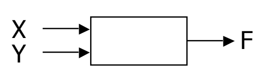
Combinational Logic (mostly review!)

- Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . .
  - Minimal set
- Axioms and theorems of Boolean algebra
  - Proofs by re-writing
  - Proofs by perfect induction
- Gate logic
  - Networks of Boolean functions
  - Time behavior
- Canonical forms
  - Two-level
  - Incompletely specified functions

Possible Logic Functions of Two Variables

- 16 possible functions of 2 input variables:
  - 2**(2**n) functions of n inputs



16 possible functions (F0–F15)

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

0
X and Y
X
Y
X xor Y
X or Y
X nor Y
not (X or Y)
X = Y
not Y
not X
X nand Y
not (X and Y)
1

Cost of Different Logic Functions

- Some are easier, others harder, to implement
  - Each has a cost associated with the number of switches needed
  - 0 (F0) and 1 (F15): require 0 switches, directly connect output to low/high
  - X (F3) and Y (F5): require 0 switches, output is one of inputs
  - X' (F12) and Y' (F10): require 2 switches for "inverter" or NOT-gate
  - X nor Y (F4) and X nand Y (F14): require 4 switches
  - X or Y (F7) and X and Y (F1): require 6 switches
  - X = Y (F9) and X $\oplus$ Y (F6): require 16 switches

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

Minimal Set of Functions

- Implement any logic functions from NOT, NOR, and NAND?
  - For example, implementing X and Y
    is the same as implementing not (X nand Y)
- Do it with only NOR or only NAND
  - NOT is just a NAND or a NOR with both inputs tied together

X	Y	X nor Y
0	0	1
1	1	0

X	Y	X nand Y
0	0	1
1	1	0

  - and NAND and NOR are "duals", i.e., easy to implement one using the other

X nand Y $\equiv$ not ((not X) nor (not Y))
X nor Y $\equiv$ not ((not X) nand (not Y))

- Based on the mathematical foundations of logic: Boolean Algebra

Algebraic Structure

- Consists of
  - Set of elements B
  - Binary operations { + , • }
  - Unary operation { ' }
  - Following axioms hold:

1. set B contains at least two elements, a, b, such that a ≠ b
2. closure: a + b is in B; a • b is in B
3. commutativity: a + b = b + a; a • b = b • a
4. associativity: a + (b + c) = (a + b) + c; a • (b • c) = (a • b) • c
5. Identity: a + 0 = a; a • 1 = a
6. distributivity: a + (b • c) = (a + b) • (a + c); a • (b + c) = (a • b) + (a • c)
7. complementarity: a + a' = 1; a • a' = 0

Boolean Algebra

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

- All algebraic axioms hold

Logic Functions and Boolean Algebra

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

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

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

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

(X • Y) + (X' • Y') ≡ X = Y

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

X, Y are Boolean algebra variables

Axioms and Theorems of Boolean Algebra

- Identity
  1. X + 0 = X 1D. X • 1 = X
- Null
  2. X + 1 = 1 2D. X • 0 = 0
- Idempotency:
  3. X + X = X 3D. X • X = X
- Involution:
  4. (X')' = X
- Complementarity:
  5. X + X' = 1 5D. X • X' = 0
- Commutativity:
  6. X + Y = Y + X 6D. X • Y = Y • X
- Associativity:
  7. (X + Y) + Z = X + (Y + Z) 7D. (X • Y) • Z = X • (Y • Z)

Axioms and Theorems of Boolean Algebra (cont'd)

- Distributivity:
  8. $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$ 8D. $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$
- Uniting:
  9. $X \cdot Y + X \cdot Y' = X$ 9D. $(X + Y) \cdot (X + Y') = X$
- Absorption:
  10. $X + X \cdot Y = X$ 10D. $X \cdot (X + Y) = X$
  11. $(X + Y') \cdot Y = X \cdot Y$ 11D. $(X \cdot Y') + Y = X + Y$
- Factoring:
  12. $(X + Y) \cdot (X' + Z) = X \cdot Z + X' \cdot Y$ 12D. $X \cdot Y + X' \cdot Z = (X + Z) \cdot (X' + Y)$
- Consensus:
  13. $(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z$ 13D. $(X + Y) \cdot (Y + Z) \cdot (X' + Z) = (X + Y) \cdot (X' + Z)$

Axioms and Theorems of Boolean Algebra (cont'd)

- deMorgan's:
  14. $(X + Y + ...)' = X' \cdot Y' \cdot ...$ 14D. $(X \cdot Y \cdot ...)' = X' + Y' + ...$
- Generalized de Morgan's:
  15. $f'(X1,X2,...,Xn,0,1,+,\cdot) = f(X1',X2',...,Xn',1,0,\cdot,+)$
- Establishes relationship between $\cdot$ and +

Axioms and Theorems of Boolean Algebra (cont'd)

- Duality
  - Dual of a Boolean expression is derived by replacing • by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
  - Any theorem that can be proven is thus also proven for its dual!
  - Meta-theorem (a theorem about theorems)
- Duality:
  - 16. $X + Y + \ldots \Leftrightarrow X \cdot Y \cdot \ldots$
- Generalized duality:
  - 17. $f(X1,X2,\ldots,Xn,0,1,+,\cdot) \Leftrightarrow f(X1,X2,\ldots,Xn,1,0,\cdot,+)$
- Different than deMorgan's Law
  - This is a statement about theorems
  - This is not a way to manipulate (re-write) expressions

Proving Theorems (Rewriting Method)

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

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

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

identity (1D)	$X + X \cdot Y$	=	$X \cdot 1 + X \cdot Y$
distributivity (8)	$X \cdot 1 + X \cdot Y$	=	$X \cdot (1 + Y)$
identity (2)	$X \cdot (1 + Y)$	=	$X \cdot (1)$
identity (1D)	$X \cdot (1)$	=	X ✓

Proving Theorems (Perfect Induction)

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

$(X + Y)' = X' \cdot Y'$
NOR is equivalent to AND
with inputs complemented

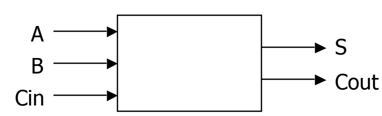
X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

$(X \cdot Y)' = X' + Y'$
NAND is equivalent to OR
with inputs complemented

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

Simple Example

- 1-bit binary adder
  - Inputs: A, B, Carry-in
  - Outputs: Sum, Carry-out



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S = A'\ B'\ Cin + A'\ B\ Cin' + A\ B'\ Cin' + A\ B\ Cin$

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

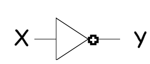
Apply the Theorems to Simplify Expressions

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

Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin
= A' B Cin + A B' Cin + A B Cin' + A B Cin + A B Cin
= A' B Cin + A B Cin + A B' Cin + A B Cin' + A B Cin
= (A' + A) B Cin + A B' Cin + A B Cin' + A B Cin
= (1) B Cin + A B' Cin + A B Cin' + A B Cin
= B Cin + A B' Cin + A B Cin' + A B Cin + A B Cin
= B Cin + A B' Cin + A B Cin + A B Cin' + A B Cin
= B Cin + A (B' + B) Cin + A B Cin' + A B Cin
= B Cin + A (1) Cin + A B Cin' + A B Cin
= B Cin + A Cin + A B (Cin' + Cin)
= B Cin + A Cin + A B (1)
= B Cin + A Cin + A B

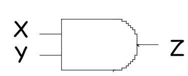
From Boolean Expressions to Logic Gates

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



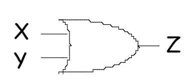
X	Y
0	1
1	0

- AND X • Y XY X ∧ Y



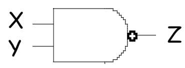
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

- OR X + Y X ∨ Y

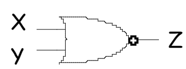


X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

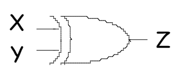
From Boolean Expressions to Logic Gates (cont'd)

- NAND 

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

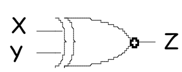
- NOR 

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

- XOR
  $X \oplus Y$ 

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

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

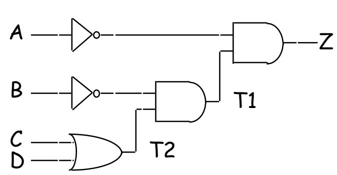
- XNOR
  X = Y 

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

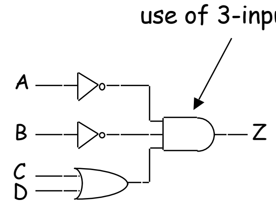
X xnor Y = X Y + X' Y'
X and Y are the same
("equality", "coincidence")

From Boolean Expressions to Logic Gates (cont'd)

- More than one way to map expressions to gates
  - e.g., $Z = A' \cdot B' \cdot (C + D) = (A' \cdot \underbrace{(B' \cdot \overbrace{(C + D)}^{T2})}_{T1})$

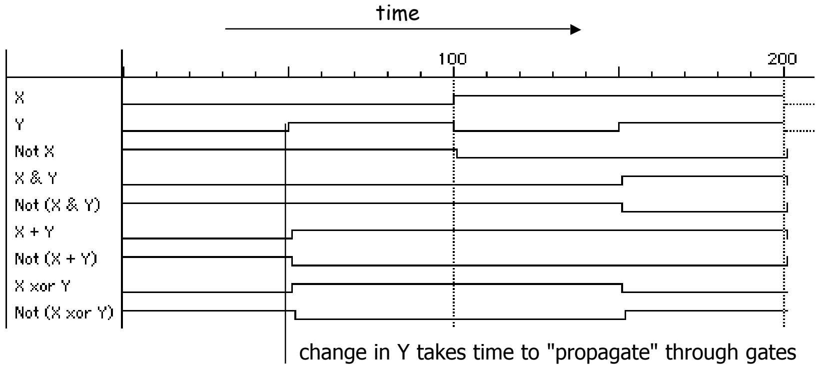


use of 3-input gate



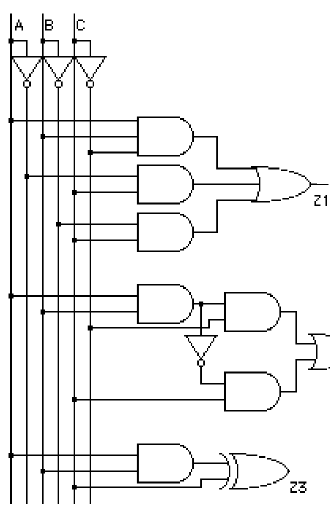
Waveform View of Logic Functions

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



Choosing Different Realizations of a Function

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



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

multi-level realization (gates with fewer inputs)

XOR gate (easier to draw but costlier to build)

Which Realization is Best?

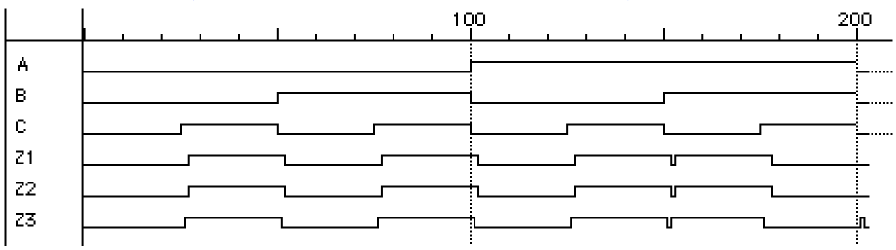
- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transistors per literal
    - why not count inverters?
  - Fewer literals means less transistors
    - smaller circuits
  - Fewer inputs implies faster gates
    - gates are smaller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

Which is the Best Realization? (cont'd)

- Reduce number of levels of gates
  - Fewer level of gates implies reduced signal propagation delays
  - Minimum delay configuration typically requires more gates
    - wider, less deep circuits
- How do we explore tradeoffs between increased circuit delay and size?
  - Automated tools to generate different solutions
  - Logic minimization: reduce number of gates and complexity
  - Logic optimization: reduction while trading off against delay

Are All Realizations Equivalent?

- Under the same inputs, the alternative implementations have *almost* the same waveform behavior
  - Delays are different
  - Glitches (hazards) may arise
  - Variations due to differences in number of gate levels and structure
- Three implementations are functionally equivalent



Implementing Boolean Functions

- Technology independent
  - Canonical forms
  - Two-level forms
  - Multi-level forms
- Technology choices
  - Packages of a few gates
  - Regular logic
  - Two-level programmable logic
  - Multi-level programmable logic

Canonical Forms

- Truth table is the unique signature of a Boolean function
- Many alternative gate realizations may have the same truth table
- Canonical forms
  - Standard forms for a Boolean expression
  - Provides a unique algebraic signature

Sum-of-Products Canonical Forms

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

F = *001 011 101 110 111*

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

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

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

Sum-of-Products Canonical Form (cont'd)

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

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

short-hand notation for minterms of 3 variables

F in canonical form:

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

canonical form ≠ minimal form

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

Product-of-Sums Canonical Form

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

F = *000* *010* *100*

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

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

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

Product-of-Sums Canonical Form (cont'd)

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

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

short-hand notation for maxterms of 3 variables

F in canonical form:

F(A, B, C) = ΠM(0,2,4)
= M0 • M2 • M4
= (A + B + C) (A + B' + C) (A' + B + C)

canonical form ≠ minimal form

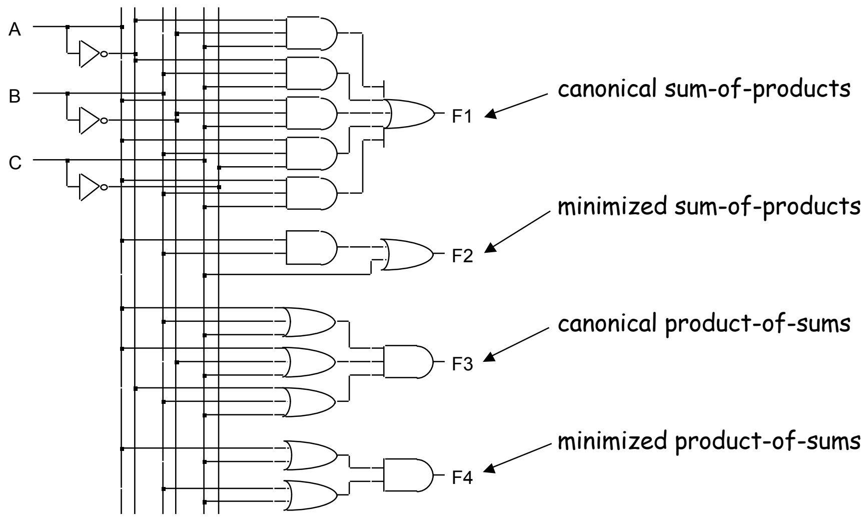
F(A, B, C) = (A + B + C) (A + B' + C) (A' + B + C)
= (A + B + C) (A + B' + C)
(A + B + C) (A' + B + C)
= (A + C) (B + C)

S-o-P, P-o-S, and deMorgan's Theorem

- Sum-of-products
  - F' = A'B'C' + A'BC' + AB'C'
- Apply de Morgan's
  - (F')' = (A'B'C' + A'BC' + AB'C')'
  - F = (A + B + C) (A + B' + C) (A' + B + C)

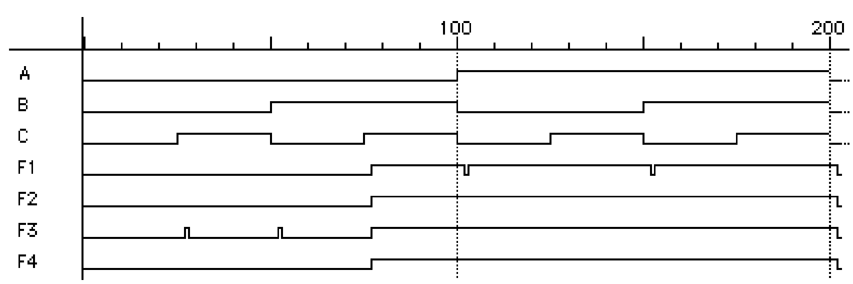
- Product-of-sums
  - F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')
- Apply de Morgan's
  - (F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'
  - F = A'B'C + A'BC + AB'C + ABC' + ABC

Four Alternative Two-level Implementations of F = AB + C



Waveforms for the Four Alternatives

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



Mapping Between Canonical Forms

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

Incompletely Specified Functions

- Example: binary coded decimal increment by 1
  - BCD digits encode decimal digits 0 – 9 in bit patterns 0000 – 1001

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

off-set of W

on-set of W

don't care (DC) set of W

these inputs patterns should never be encountered in practice – "don't care" about associated output values, can be exploited in minimization

Notation for Incompletely Specified Functions

- Don't cares and canonical forms
  - So far, only represented on-set
  - Also represent don't-care-set
  - Need two of the three sets (on-set, off-set, dc-set)

- Canonical representations of the BCD increment by 1 function:

  - $Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15$
  - $Z = \Sigma$ [$m(0,2,4,6,8) + d(10,11,12,13,14,15)$]

  - $Z = M1 \cdot M3 \cdot M5 \cdot M7 \cdot M9 \cdot D10 \cdot D11 \cdot D12 \cdot D13 \cdot D14 \cdot D15$
  - $Z = \Pi$ [$M(1,3,5,7,9) \cdot D(10,11,12,13,14,15)$]

Simplification of Two-level Combinational Logic

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

Administrative Announcement

- All discussion sections to meet in 125 Cory
- Moving F 10-11 AM discussion to F 11-noon
- Students on wait list:
  - W 9-12 Lab is still available
  - W 5-8 lab is at capacity
  - We can take a VERY small number of students into the Tu labs
  - Email your preference to Head TA Po-Kai
- Instructional Web now mirrors Randy's web site
  - http://inst.eecs.Berkeley.edu/~cs150
  - HW #1 and Lab #1 now on-line

The Uniting Theorem

- Key tool for simplification: A (B' + B) = A
- Essence of simplification:
  - 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'

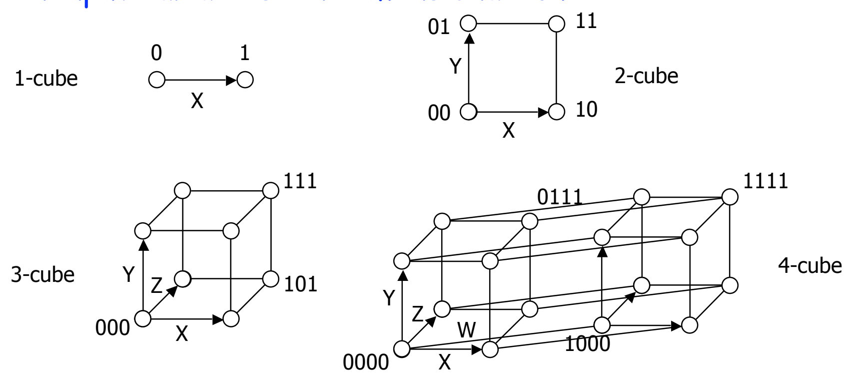
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

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

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

Boolean Cubes

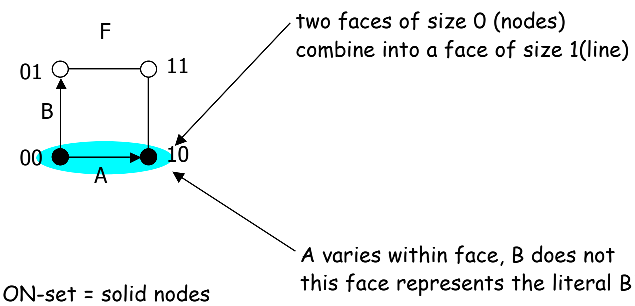
- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"



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

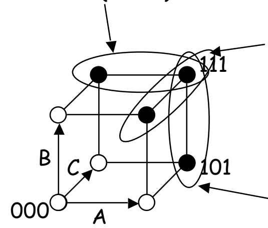


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

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

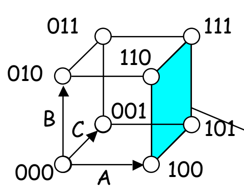
Higher Dimensional Cubes

- Sub-cubes of higher dimension than 2

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

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

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



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

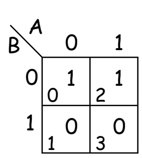
This subcube represents the literal A

m-Dimensional Cubes in an n-Dimensional Boolean Space

- In a 3-cube (three variables):
  - 0-cube, i.e., a single node, yields a term in 3 literals
  - 1-cube, i.e., a line of two nodes, yields a term in 2 literals
  - 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
  - 3-cube, i.e., a cube of eight nodes, yields a constant term "1"
- In general,
  - m-subcube within an n-cube (m < n) yields a term with n – m literals

Karnaugh Maps

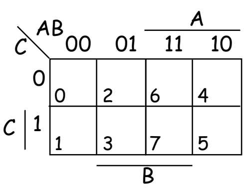
- Flat map of Boolean cube
  - Wrap–around at edges
  - Hard to draw and visualize for more than 4 dimensions
  - Virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - Guide to applying the uniting theorem
  - On-set elements with only one variable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

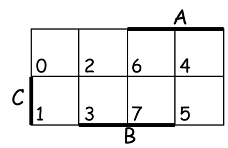


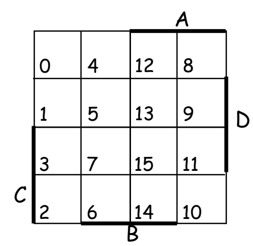
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

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



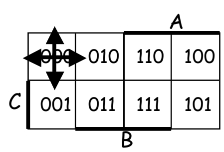


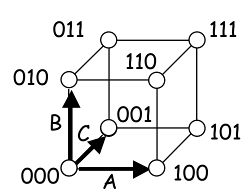


13 = 1101= ABC'D

Adjacencies in Karnaugh Maps

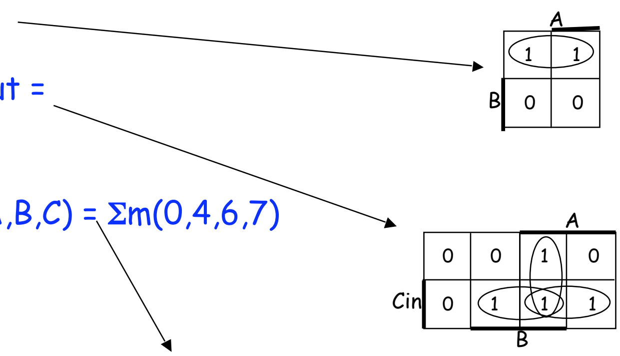
- Wrap from first to last column
- Wrap top row to bottom row





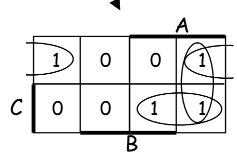
Karnaugh Map Examples

- F =
- Cout =
- f(A,B,C) = Σm(0,4,6,7)



B'

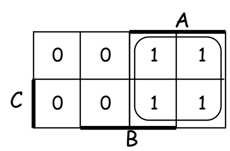
AB + ACin+ BCin



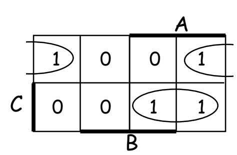
AC+ B'C' + ~~AB'~~

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

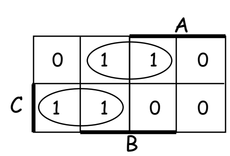
More Karnaugh Map Examples



G(A,B,C) = A



F(A,B,C) = Σm(0,4,5,7) = AC + B'C'



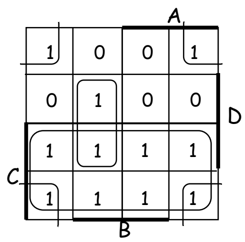
F' simply replace 1's with 0's and vice versa

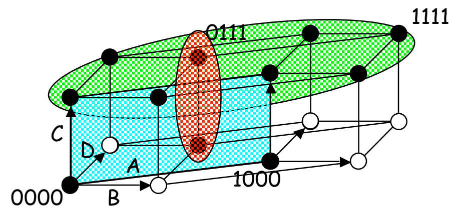
F'(A,B,C) = Σ m(1,2,3,6) = BC' + A'C

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'

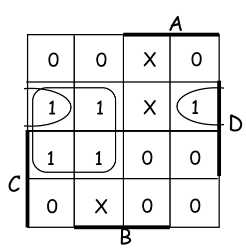




find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)

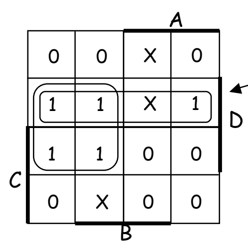
Karnaugh Maps: Don't Cares

- 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)

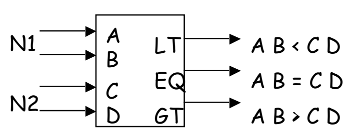
- $f(A,B,C,D) = \Sigma\, m(1,3,5,7,9) + d(6,12,13)$
  - f = A'D + B'C'D — without don't cares
  - f = A'D + C'D — with don't cares



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

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

Design Example: Two-bit Comparator

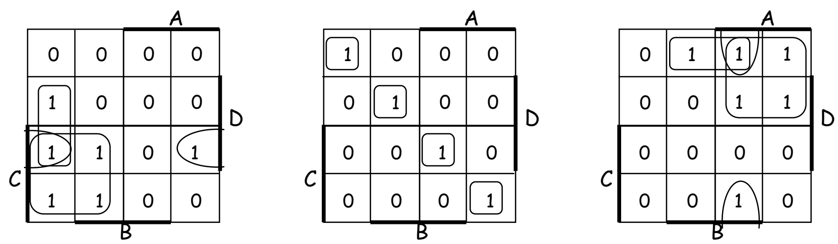


block diagram
and
truth table

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

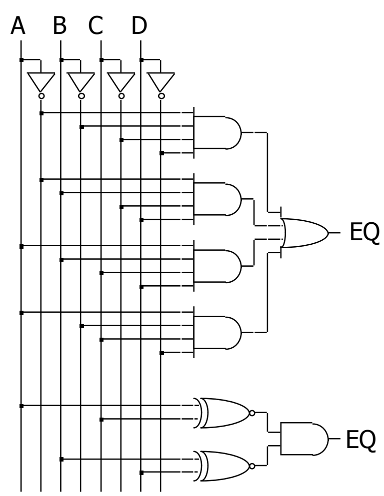
LT = A' B' D + A' C + B' C D

EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD' = (A xnor C) • (B xnor D)

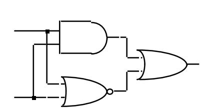
GT = B C' D' + A C' + A B D'

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

Design Example: Two-bit Comparator (cont'd)

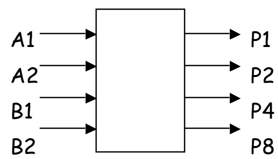


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design Example: 2x2-bit Multiplier

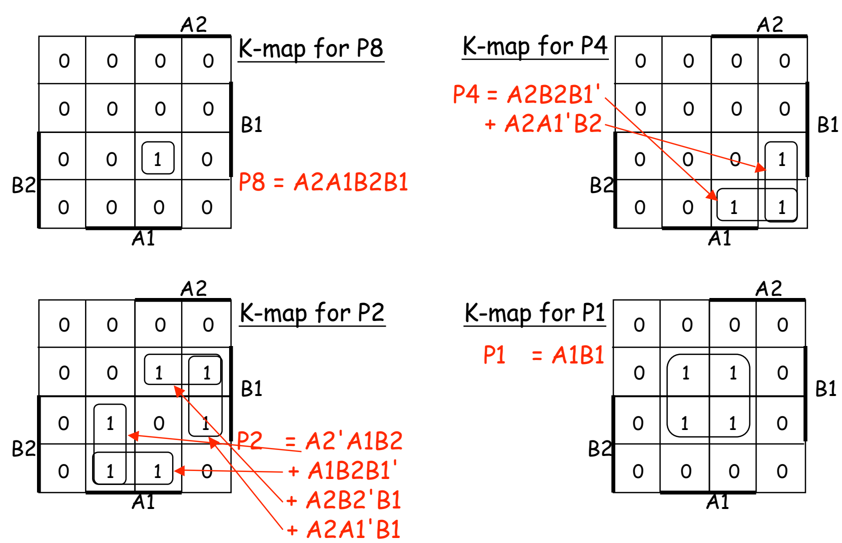


block diagram
and
truth table

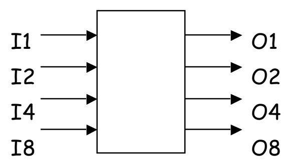
A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

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

Design Example: 2x2-bit Multiplier (cont'd)



Design Example: BCD Increment by 1

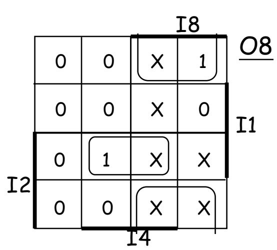


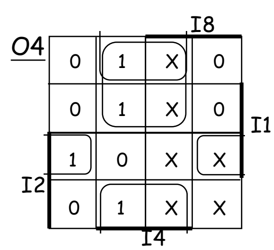
block diagram
and
truth table

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

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

Design Example: BCD Increment by 1 (cont'd)



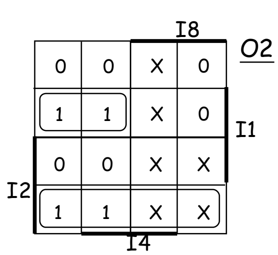


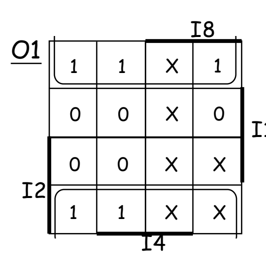
O8 = I4 I2 I1 + I8 I1'

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

O2 = I8' I2' I1 + I2 I1'

O1 = I1'

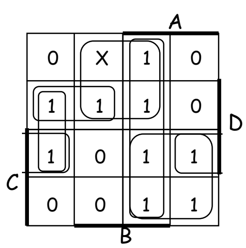




Definition of Terms for Two-level Simplification

- 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

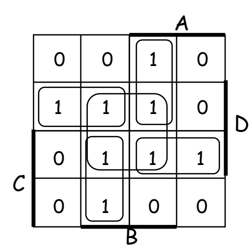
essential

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

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

essential

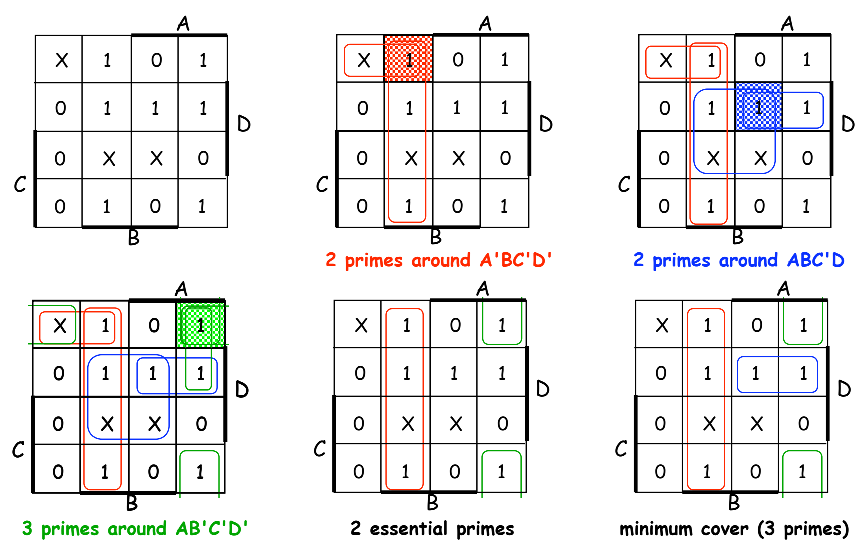
minimum cover: 4 essential implicants



Algorithm for Two-level Simplification

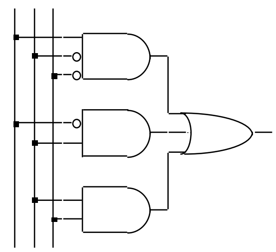
- Algorithm: minimum sum-of-products expression from a K-map
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number of elements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

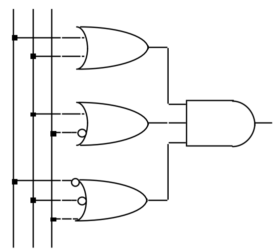


Implementations of Two-level Logic

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

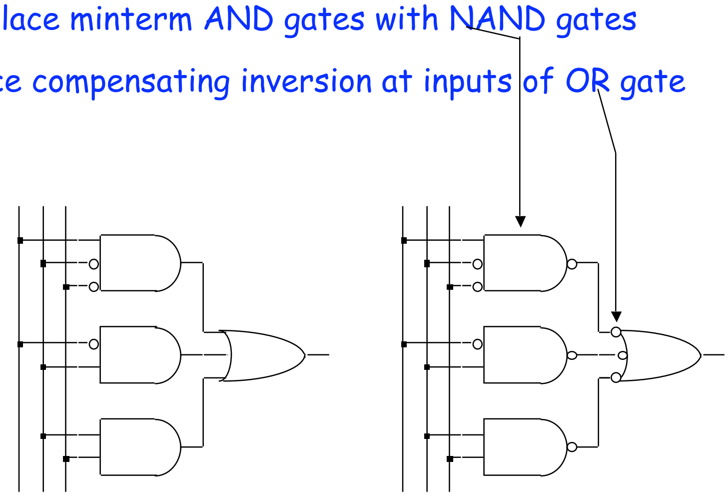


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



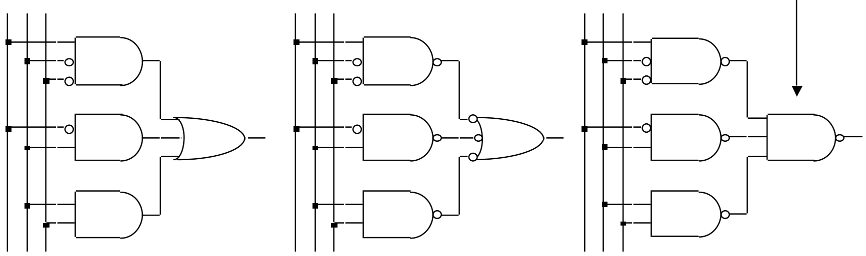
Two-level Logic Using NAND Gates

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



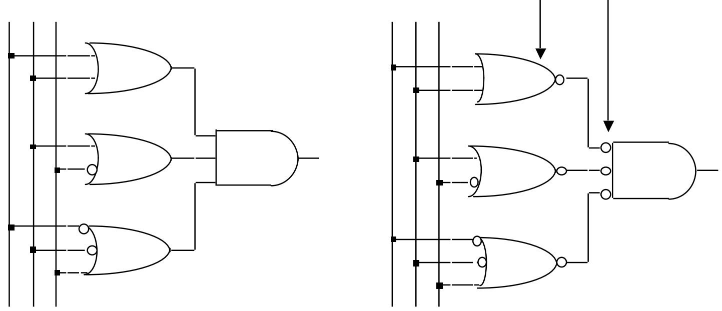
Two-level Logic Using NAND Gates (cont'd)

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



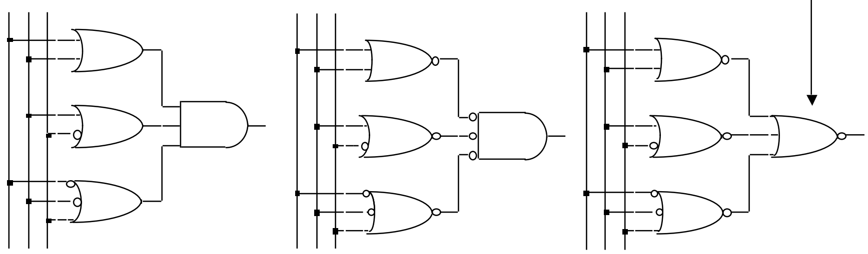
Two-level Logic Using NOR Gates

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



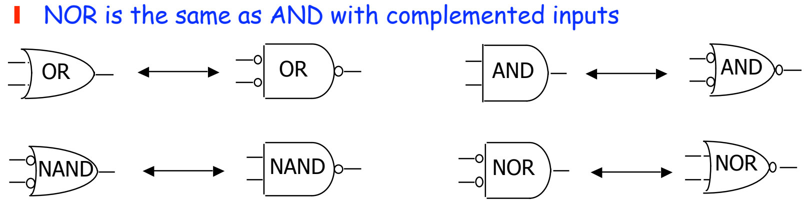
Two-level Logic Using NOR Gates (cont'd)

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



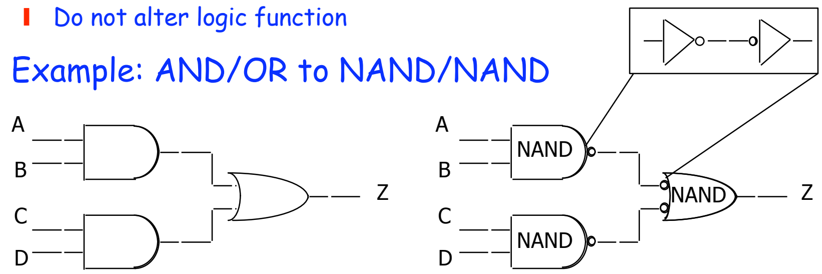
Two-level Logic Using NAND and NOR Gates

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



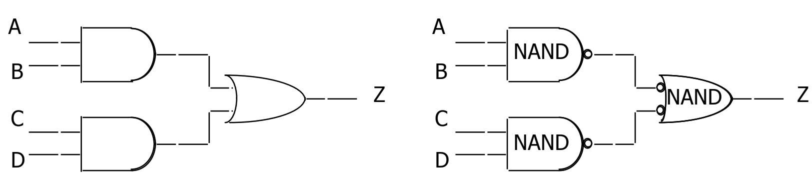
Conversion Between Forms

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



Conversion Between Forms (cont'd)

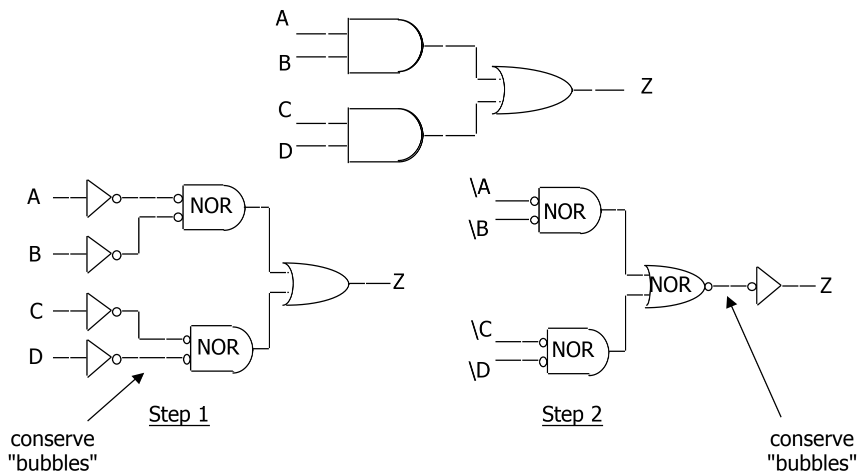
- Example: verify equivalence of two forms



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

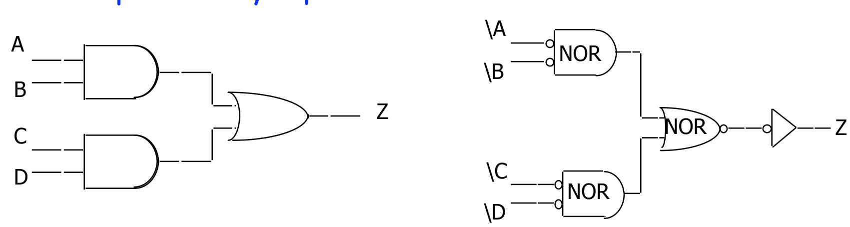
Conversion Between Forms (cont'd)

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



Conversion Between Forms (cont'd)

- Example: verify equivalence of two forms



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

Combinational Logic Summary

- Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Axioms and theorems of Boolean algebra
  - Proofs by re-writing and perfect induction
- Gate logic
  - Networks of Boolean functions and their time behavior
- Canonical forms
  - Two-level and incompletely specified functions
- Simplification
  - Two-level simplification