

Combinational logic

- 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
- Simplification
 - ▮ Boolean cubes and Karnaugh maps
 - ▮ two-level simplification

CSE 370 - Spring 1999 - Combinational Logic - 1

Possible logic functions of two variables

- There are 16 possible functions of 2 input variables:
 - ▮ in general, there are 2^{2^n} functions of n inputs



X	Y	16 possible functions (F0-F15)																
0	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	0	0	1	0	0	1	1	0	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1

Labels for the 16 functions (F0-F15) from left to right:

- 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

CSE 370 - Spring 1999 - Combinational Logic - 2

Cost of different logic functions

- Different functions are easier or 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
- ┆ thus, because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice

Minimal set of functions

- Can we implement all logic functions from NOT, NOR, and NAND?
 - ┆ For example, implementing X and Y is the same as implementing not (X nand Y)
- In fact, we can do it with only NOR or only NAND
 - ┆ NOT is just a NAND or a NOR with both inputs tied together

<u>X</u>	<u>Y</u>	<u>X nor Y</u>		<u>X</u>	<u>Y</u>	<u>X nand Y</u>	
0	0	1		0	0	1	
1	1	0		1	1	0	
 - ┆ and NAND and NOR are "duals", that is, its easy to implement one using the other

$$\begin{aligned} X \text{ nand } Y &\equiv \text{not} (\text{not } X \text{ nor } \text{not } Y) \\ X \text{ nor } Y &\equiv \text{not} (\text{not } X \text{ nand } \text{not } Y) \end{aligned}$$
- But lets not move too fast . . .
 - ┆ lets look at the mathematical foundation of logic

An algebraic structure

- An algebraic structure consists of

- a set of elements B
- binary operations $\{ +, \cdot \}$
- and a unary operation $\{ ' \}$
- such that the following axioms hold:

1. the set B contains at least two elements, a, b , such that $a \neq b$
2. closure: $a + b$ is in B $a \cdot b$ is in B
3. commutativity: $a + b = b + a$ $a \cdot b = b \cdot a$
4. associativity: $a + (b + c) = (a + b) + c$ $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
5. identity: $a + 0 = a$ $a \cdot 1 = a$
6. distributivity: $a + (b \cdot c) = (a + b) \cdot (a + c)$ $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
7. complementarity: $a + a' = 1$ $a \cdot a' = 0$

Boolean algebra

- Boolean algebra

- $B = \{0, 1\}$
- $+$ is logical OR, \cdot 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	X	Y	X'	X' • Y
0	0	0	0	0	1	0
0	1	0	0	1	1	1
1	0	0	1	0	0	0
1	1	1	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

$(X \bullet Y) + (X' \bullet Y') \equiv X = Y$

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
 - $X + 0 = X$
 - $X \bullet 1 = X$
- null
 - $X + 1 = 1$
 - $X \bullet 0 = 0$
- idempotency:
 - $X + X = X$
 - $X \bullet X = X$
- involution:
 - $(X')' = X$
- complementarity:
 - $X + X' = 1$
 - $X \bullet X' = 0$
- commutativity:
 - $X + Y = Y + X$
 - $X \bullet Y = Y \bullet X$
- associativity:
 - $(X + Y) + Z = X + (Y + Z)$
 - $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$

Axioms and theorems of Boolean algebra (cont'd)

- distributivity:

$$8. X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \quad 8D. X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$$

- uniting:

$$9. X \cdot Y + X \cdot Y' = X \quad 9D. (X + Y) \cdot (X + Y') = X$$

- absorption:

$$10. X + X \cdot Y = X \quad 10D. X \cdot (X + Y) = X$$

$$11. (X + Y') \cdot Y = X \cdot Y \quad 11D. (X \cdot Y') + Y = X + Y$$

- factoring:

$$12. (X + Y) \cdot (X' + Z) = X \cdot Z + X' \cdot Y \quad 16D. X \cdot Y + X' \cdot Z = (X + Z) \cdot (X' + Y)$$

- consensus:

$$13. (X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z \quad 17D. (X + Y) \cdot (Y + Z) \cdot (X' + Z) = (X + Y) \cdot (X' + Z)$$

Axioms and theorems of Boolean algebra (cont')

- de Morgan's:

$$14. (X + Y + \dots)' = X' \cdot Y' \cdot \dots \quad 12D. (X \cdot Y \cdot \dots)' = X' + Y' + \dots$$

- generalized de Morgan's:

$$15. f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot) = f(X_1', X_2', \dots, X_n', 1, 0, \cdot, +)$$

- establishes relationship between \cdot and $+$

Axioms and theorems of Boolean algebra (cont')

- Duality
 - a 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!
 - a meta-theorem (a theorem about theorems)
- duality:
 - 16. $X + Y + \dots \Leftrightarrow X \cdot Y \cdot \dots$
- generalized duality:
 - 17. $f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot) \Leftrightarrow f(X_1, X_2, \dots, X_n, 1, 0, \cdot, +)$
- Different than deMorgan's Law
 - this is a statement about theorems
 - this is not a way to manipulate (re-write) expressions

Proving theorems (rewriting)

- 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)	$X \cdot (1)$	$= X \Rightarrow$

 - 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)	$X \cdot (1)$	$= X \Rightarrow$

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'	$(X + Y)'$	$X' \cdot Y'$
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'	$(X \cdot Y)'$	$X' + Y'$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

A 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

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

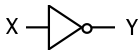


$$Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin$$

Apply the theorems to simplify expressions

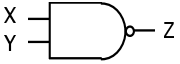



- The theorems of Boolean algebra can simplify Boolean expressions
 - e.g., full adder's carry-out function (same rules apply to any function)

$$\begin{aligned}
 \text{Cout} &= A' B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= A' B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} + A B \text{Cin} \\
 &= A' B \text{Cin} + A B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= (A' + A) B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= (1) B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} + A B \text{Cin} \\
 &= B \text{Cin} + A B' \text{Cin} + A B \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= B \text{Cin} + A (B' + B) \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= B \text{Cin} + A (1) \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\
 &= B \text{Cin} + A \text{Cin} + A B (\text{Cin}' + \text{Cin}) \\
 &= B \text{Cin} + A \text{Cin} + A B (1) \\
 &= B \text{Cin} + A \text{Cin} + A B
 \end{aligned}$$

From Boolean expressions to logic gates

■ NOT	X'	\bar{X}	$\sim X$		$X \rightarrow Y$	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>X</th> <th>Y</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </tbody> </table>	X	Y	0	1	1	0									
X	Y																				
0	1																				
1	0																				
■ AND	$X \cdot Y$	XY	$X \wedge Y$		$X \rightarrow Z$	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>X</th> <th>Y</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	X	Y	Z	0	0	0	0	1	0	1	0	0	1	1	1
X	Y	Z																			
0	0	0																			
0	1	0																			
1	0	0																			
1	1	1																			
■ OR	$X + Y$		$X \vee Y$		$X \rightarrow Z$	<table border="1" style="display: inline-table; vertical-align: middle;"> <thead> <tr> <th>X</th> <th>Y</th> <th>Z</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	X	Y	Z	0	0	0	0	1	1	1	0	1	1	1	1
X	Y	Z																			
0	0	0																			
0	1	1																			
1	0	1																			
1	1	1																			

From Boolean expressions to logic gates (cont'd)

■ NAND		<table border="1" data-bbox="812 409 941 525"> <thead> <tr><th>X</th><th>Y</th><th>Z</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	X	Y	Z	0	0	1	0	1	1	1	0	1	1	1	0	
X	Y	Z																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
■ NOR		<table border="1" data-bbox="812 556 941 672"> <thead> <tr><th>X</th><th>Y</th><th>Z</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	X	Y	Z	0	0	1	0	1	0	1	0	0	1	1	0	
X	Y	Z																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
■ XOR $X \oplus Y$		<table border="1" data-bbox="812 703 941 819"> <thead> <tr><th>X</th><th>Y</th><th>Z</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	X	Y	Z	0	0	0	0	1	1	1	0	1	1	1	0	$X \text{ xor } Y = X Y' + X' Y$ X or Y but not both ("inequality", "difference")
X	Y	Z																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
■ XNOR $X = Y$		<table border="1" data-bbox="812 850 941 966"> <thead> <tr><th>X</th><th>Y</th><th>Z</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	X	Y	Z	0	0	1	0	1	0	1	0	0	1	1	1	$X \text{ xnor } Y = X Y + X' Y'$ X and Y are the same ("equality", "coincidence")
X	Y	Z																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

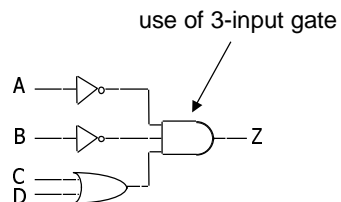
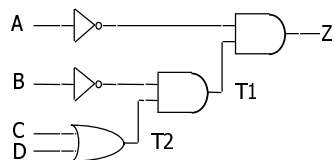
CSE 370 - Spring 1999 - Combinational Logic - 17

From Boolean expressions to logic gates (cont'd)

- More than one way to map expressions to gates

■ e.g., $Z = A' \cdot B' \cdot (C + D) = (A' \cdot (B' \cdot (C + D)))$

$$\frac{\quad T2}{T1}$$



CSE 370 - Spring 1999 - Combinational Logic - 18

Which realization is best?

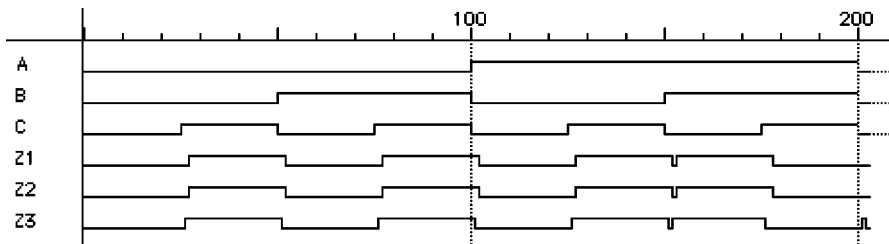
- Reduce number of inputs
 - ┆ literal: input variable (complemented or not)
 - ┆ can approximate cost of logic gate as 2 transistors per literal
 - ┆ why not count inverters?
 - ┆ fewer literals means less transistors
 - ┆ smaller circuits
 - ┆ fewer inputs implies faster gates
 - ┆ gates are smaller 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 input stimuli, the three 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
- The three implementations are functionally equivalent



CSE 370 - Spring 1999 - Combinational Logic - 23

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

CSE 370 - Spring 1999 - Combinational Logic - 24

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

					$F = 001 \quad 011 \quad 101 \quad 110 \quad 111$				
					$F = A'B'C + A'BC + AB'C + ABC' + ABC$				
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

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 \neq minimal form

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

short-hand notation for minterms of 3 variables

Product-of-sums canonical form

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

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

$$\begin{aligned}
 F &= \begin{matrix} 000 & 010 & 100 \\ (A + B + C) & (A + B' + C) & (A' + B + C) \end{matrix}
 \end{aligned}$$

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

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

F in canonical form:

$$\begin{aligned}
 F(A, B, C) &= \Pi M(0,2,4) \\
 &= M0 \cdot M2 \cdot M4 \\
 &= (A + B + C) (A + B' + C) (A' + B + C)
 \end{aligned}$$

canonical form \neq minimal form

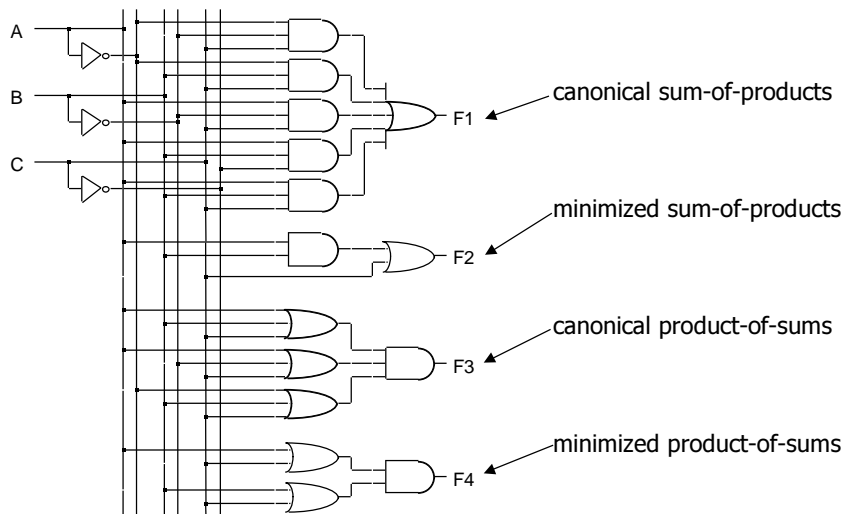
$$\begin{aligned}
 F(A, B, C) &= (A + B + C) (A + B' + C) (A' + B + C) \\
 &= (A + B + C) (A + B' + C) \\
 &\quad (A + B + C) (A' + B + C) \\
 &= (A + C) (B + C)
 \end{aligned}$$

short-hand notation for maxterms of 3 variables

S-o-P, P-o-S, and de Morgan'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
 - $(F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'$
 - $F = A'B'C' + A'BC' + AB'C' + ABC' + ABC$

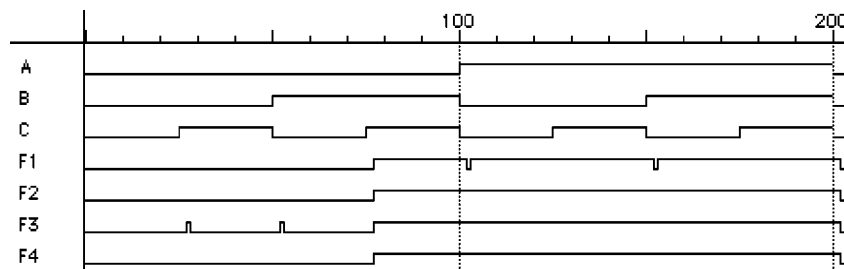
Four alternative two-level implementations of $F = AB + C$



CSE 370 - Spring 1999 - Combinational Logic - 31

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)



CSE 370 - Spring 1999 - Combinational Logic - 32

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

Incompletely specified functions

- Example: binary coded decimal increment by 1
 - ┆ BCD digits encode the decimal digits 0 – 9 in the 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:
 - ┆ $Z = m_0 + m_2 + m_4 + m_6 + m_8 + d_{10} + d_{11} + d_{12} + d_{13} + d_{14} + d_{15}$
 - ┆ $Z = \Sigma [m(0,2,4,6,8) + d(10,11,12,13,14,15)]$

 - ┆ $Z = M_1 \cdot M_3 \cdot M_5 \cdot M_7 \cdot M_9 \cdot D_{10} \cdot D_{11} \cdot D_{12} \cdot D_{13} \cdot D_{14} \cdot D_{15}$
 - ┆ $Z = \Pi [M(1,3,5,7,9) \cdot D(10,11,12,13,14,15)]$

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
 - ┆ 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

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

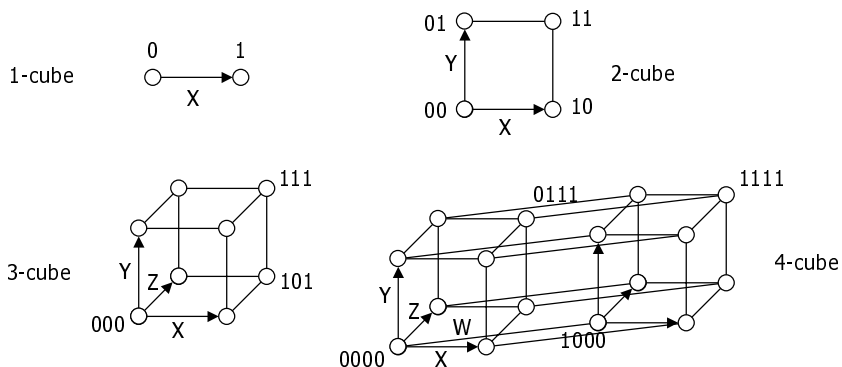
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

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

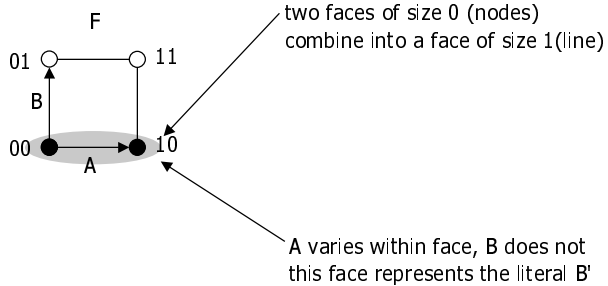
- Visual technique for identifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"



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

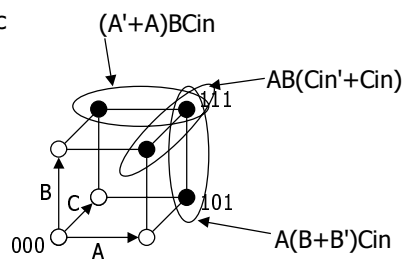


ON-set = solid nodes
 OFF-set = empty nodes
 DC-set = x'd nodes

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

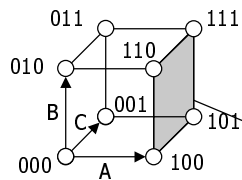


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

$$Cout = BCin + AB + ACin$$

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 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	0	1
B	0	1	2
1	0	1	3

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

CSE 370 - Spring 1999 - Combinational Logic - 43

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

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

	A	0	2	6	4
C	1	1	3	7	5

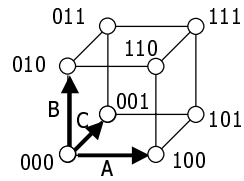
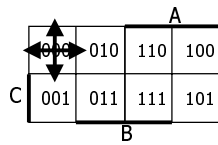
	A	0	4	12	8
C	1	1	5	13	9
3	2	3	6	14	10

13 = 1101 = ABC'D

CSE 370 - Spring 1999 - Combinational Logic - 44

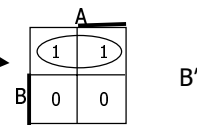
Adjacencies in Karnaugh maps

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



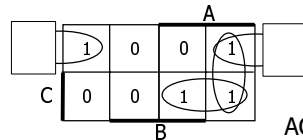
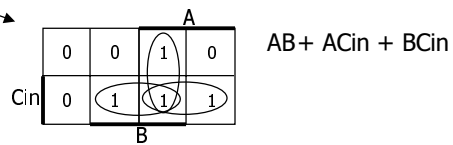
Karnaugh map examples

■ $F =$



■ $\text{Cout} =$

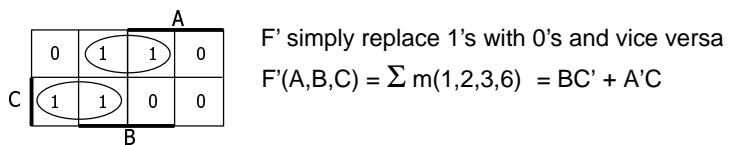
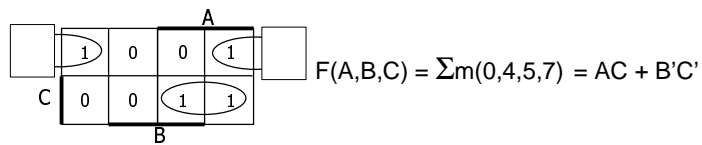
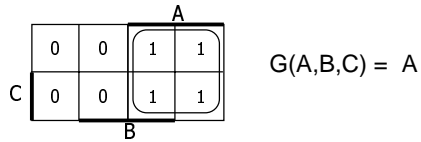
■ $f(A,B,C) = \Sigma m(0,4,6,7)$



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

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

More Karnaugh map examples

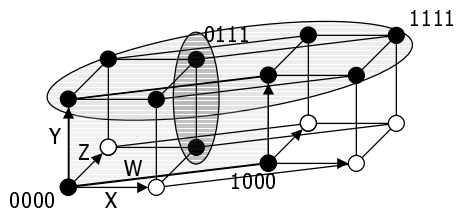
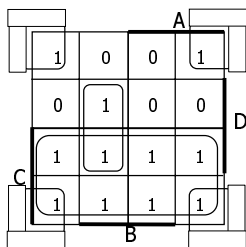


CSE 370 - Spring 1999 - Combinational Logic - 47

Karnaugh map: 4-variable example

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

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



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

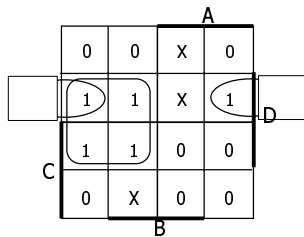
CSE 370 - Spring 1999 - Combinational Logic - 48

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

■ $f = A'D + B'C'D$



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

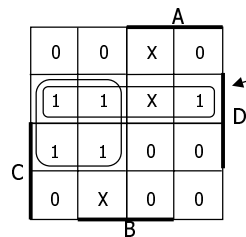
■ $f(A,B,C,D) = \Sigma m(1,3,5,7,9) + d(6,12,13)$

■ $f = A'D + B'C'D$

without don't cares

■ $f = A'D + C'D$

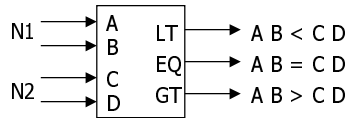
with don't cares



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

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

Design example: two-bit comparator

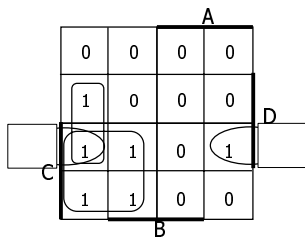


block diagram
and
truth table

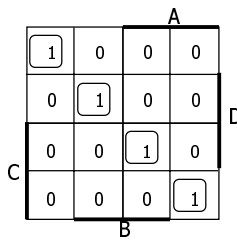
A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

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

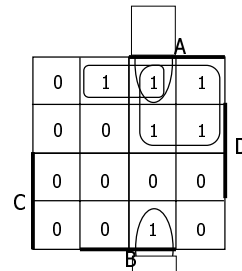
Design example: two-bit comparator (cont'd)



K-map for LT



K-map for EQ



K-map for GT

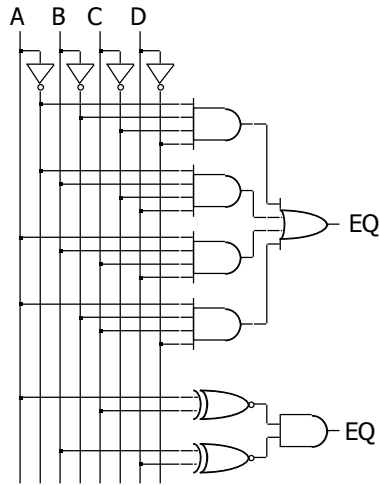
$$LT = A' B' D + A' C + B' C D$$

$$EQ = A' B' C' D' + A' B' C' D + A B C D + A B' C' D' = (A \text{ xnor } C) \cdot (B \text{ xnor } D)$$

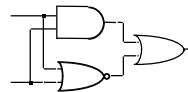
$$GT = B' C' D' + A' C + A B D'$$

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

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

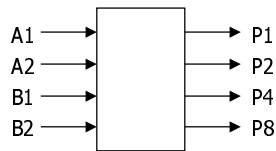


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier

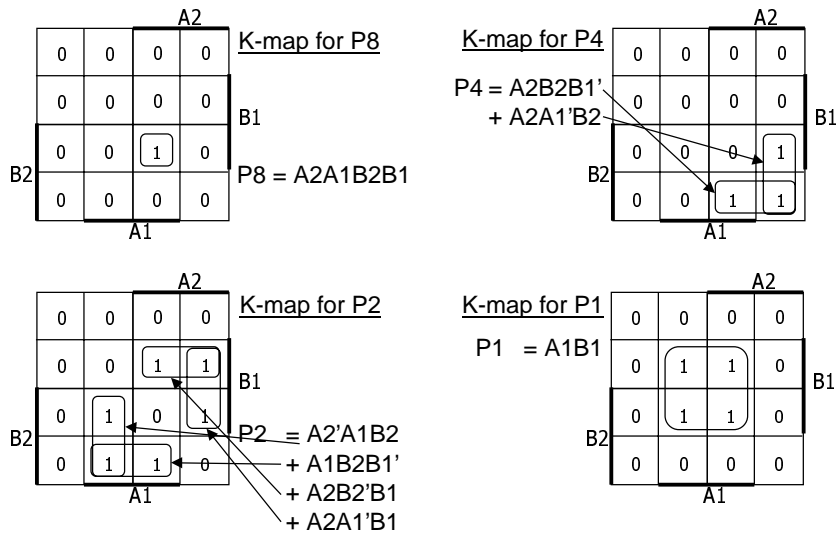


block diagram and truth table

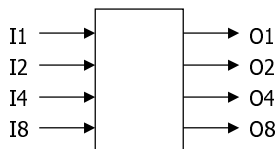
A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

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)

		I8		
	0	0	X	1
	0	0	X	0
I2	0	1	X	X
	0	0	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		
	0	0	X	0
	1	1	X	0
I2	0	0	X	X
	1	1	X	X
			I4	

		I8		
O4	0	1	X	0
	0	1	X	0
I2	1	0	X	X
	0	1	X	X
			I4	

		I8		
O1	1	1	X	1
	0	0	X	0
I2	0	0	X	X
	1	1	X	X
			I4	

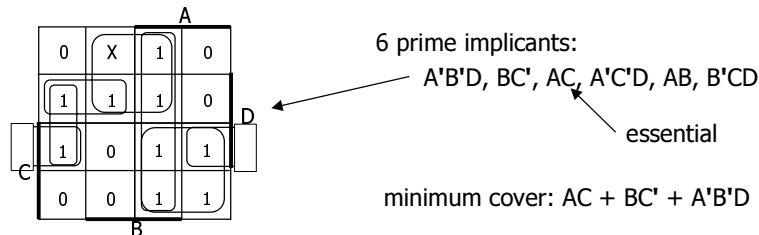
CSE 370 - Spring 1999 - Combinational Logic - 57

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)

CSE 370 - Spring 1999 - Combinational Logic - 58

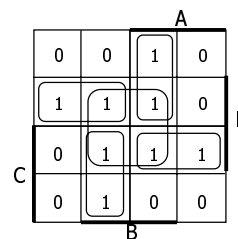
Examples to illustrate terms



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

essential

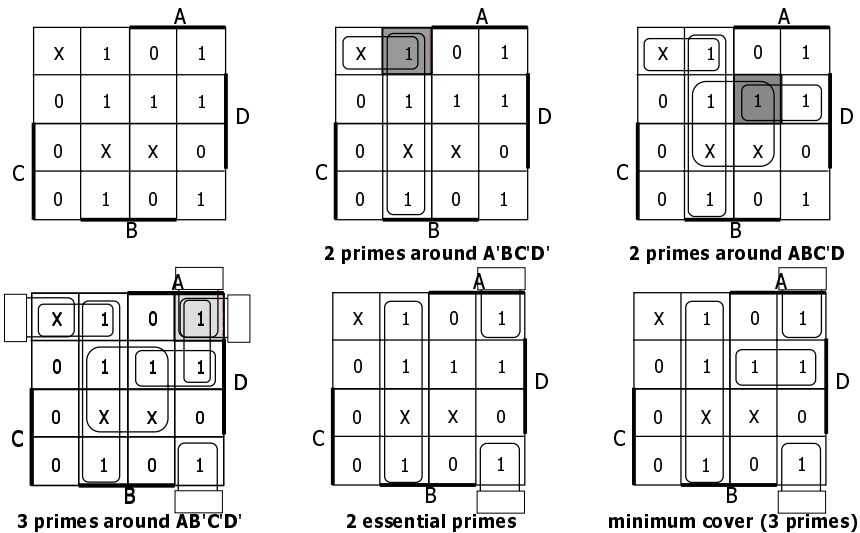
minimum cover: 4 essential implicants



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)



CSE 370 - Spring 1999 - Combinational Logic - 61

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
- Later
 - automation of simplification
 - multi-level logic
 - design case studies
 - time behavior

CSE 370 - Spring 1999 - Combinational Logic - 62