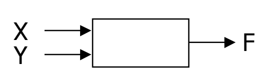
Combinational logic

- 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

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	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

- 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

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",
    that is, its easy to implement one using the other

X nand Y ≡ not ((not X) nor (not Y))
X nor Y ≡ not ((not X) nand (not Y))

- 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 { + , • }
  - and a unary operation { ' }
  - such that the following axioms hold:

1. the 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 • (Y + Z) = (X • Y) + (X • Z)
  - 8D. X + (Y • Z) = (X + Y) • (X + Z)
- uniting:
  - 9. X • Y + X • Y' = X
  - 9D. (X + Y) • (X + Y') = X
- absorption:
  - 10. X + X • Y = X
  - 10D. X • (X + Y) = X
  - 11. (X + Y') • Y = X • Y
  - 11D. (X • Y') + Y = X + Y
- factoring:
  - 12. (X + Y) • (X' + Z) = X • Z + X' • Y
  - 16D. X • Y + X' • Z = (X + Z) • (X' + Y)
- concensus:
  - 13. (X • Y) + (Y • Z) + (X' • Z) = X • Y + X' • Z
  - 17D. (X + Y) • (Y + Z) • (X' + Z) = (X + Y) • (X' + Z)

Axioms and theorems of Boolean algebra (cont')

- de Morgan's:
  - 14. (X + Y + ...)' = X' • Y' • ...
  - 12D. (X • Y • ...)' = X' + Y' + ...
- generalized de Morgan's:
  - 15. f'(X1,X2,...,Xn,0,1,+,•) = f(X1',X2',...,Xn',1,0,•,+)
- establishes relationship between • 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. $X + Y + \ldots \Leftrightarrow X \bullet Y \bullet \ldots$
- generalized duality:

  17. $f(X1,X2,\ldots,Xn,0,1,+,\bullet) \Leftrightarrow f(X1,X2,\ldots,Xn,1,0,\bullet,+)$
- 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: $X \bullet Y + X \bullet Y' = X$

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

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

identity (1D)	$X + X \bullet Y$	=	$X \bullet 1 + X \bullet Y$
distributivity (8)	$X \bullet 1 + X \bullet Y$	=	$X \bullet (1 + Y)$
identity (2)	$X \bullet (1 + Y)$	=	$X \bullet (1)$
identity (1D)	$X \bullet (1)$	=	X ➻

Proving theorems (perfect induction)

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

(X + Y)' = X' • Y'
NOR is equivalent to AND
with inputs complemented

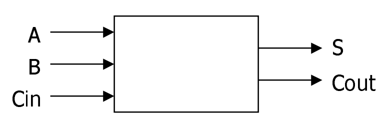
X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

(X • 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

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

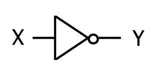
Apply the theorems to simplify expressions

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

```
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
```

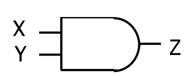

From Boolean expressions to logic gates

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



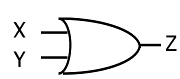
X	Y
0	1
1	0

- AND X • Y XY X ∧ Y



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

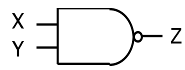
- OR X + Y X ∨ Y



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

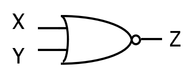
From Boolean expressions to logic gates (cont’d)

- NAND



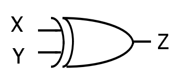
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- NOR



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

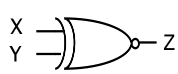
- XOR
  $X \oplus Y$



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

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

- XNOR
  X = Y

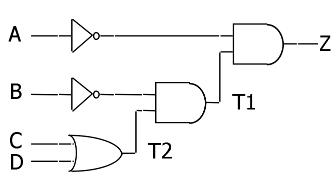


X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

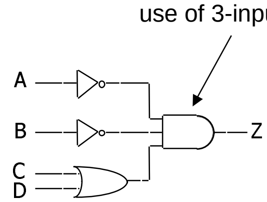
X <u>xnor</u> Y = X Y + X’ Y’
X and Y are the same
("equality", "coincidence")

From Boolean expressions to logic gates (cont’d)

- More than one way to map expressions to gates
  - e.g., Z = A' • B' • (C + D) = (A' • (B' • (C + D)))
    (C + D) = T2; (B' • (C + D)) = T1

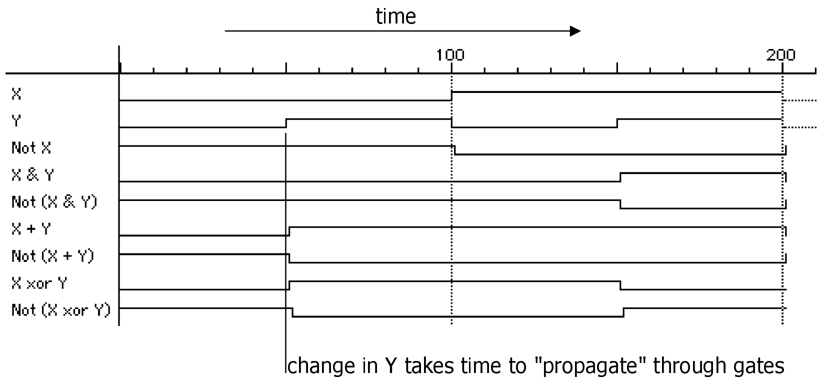


use of 3-input gate



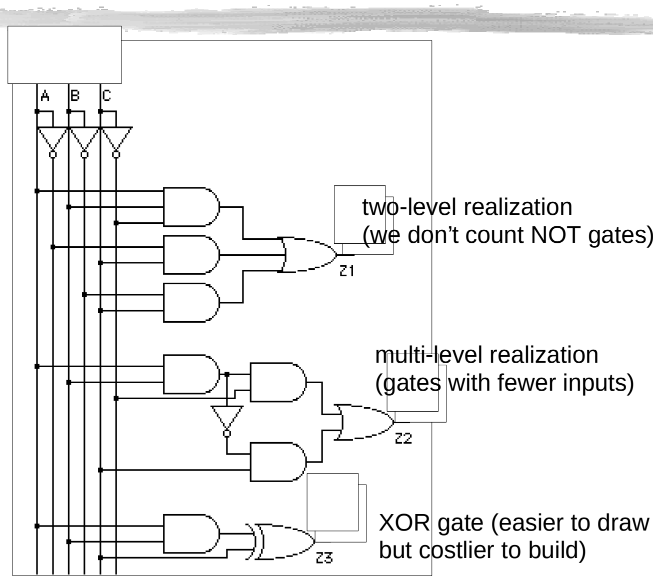
Waveform view of logic functions

- 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



Which realization is best?

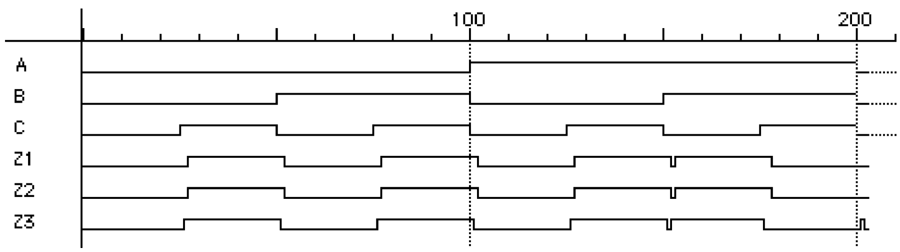
- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transitors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are smaller 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



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:

F(A, B, C) = Σm(1,3,5,6,7)
= m1 + m3 + m5 + m6 + m7
= A'B'C + A'BC + AB'C + ABC' + ABC

canonical form ≠ minimal form

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

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

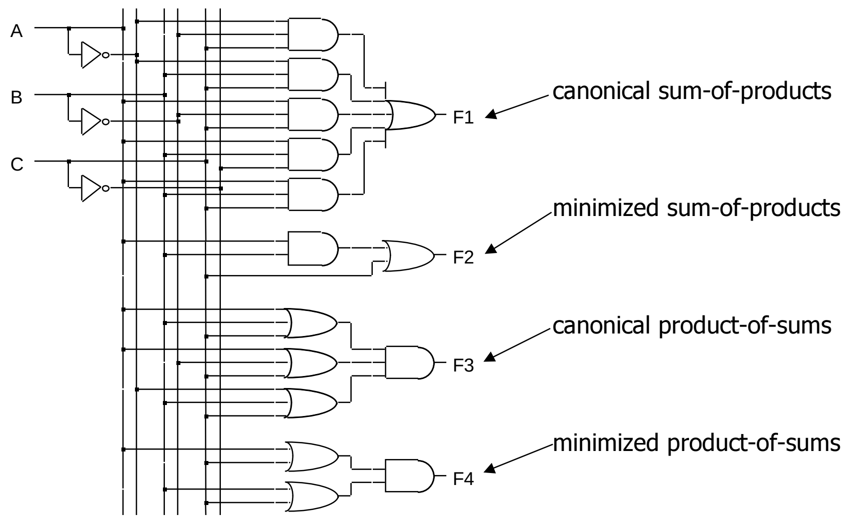
F(A, B, C) = (A + B + C) (A + B' + C) (A' + B + C)
= (A + B + C) (A + B' + C)
(A + B + C) (A' + B + C)
= (A + C) (B + C)

S-o-P, P-o-S, and 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)

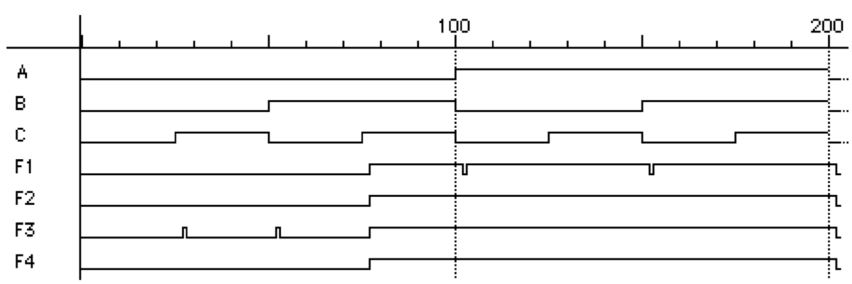
- Product-of-sums
  - F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')
- Apply de Morgan's
  - (F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'
  - F = A'B'C + A'BC + AB'C + ABC' + ABC

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



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

Incompleteley 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 = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15
  - Z = Σ [m(0,2,4,6,8) + d(10,11,12,13,14,15)]
  - Z = M1 • M3 • M5 • M7 • M9 • D10 • D11 • D12 • D13 • D14 • D15
  - Z = Π [M(1,3,5,7,9) • 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'

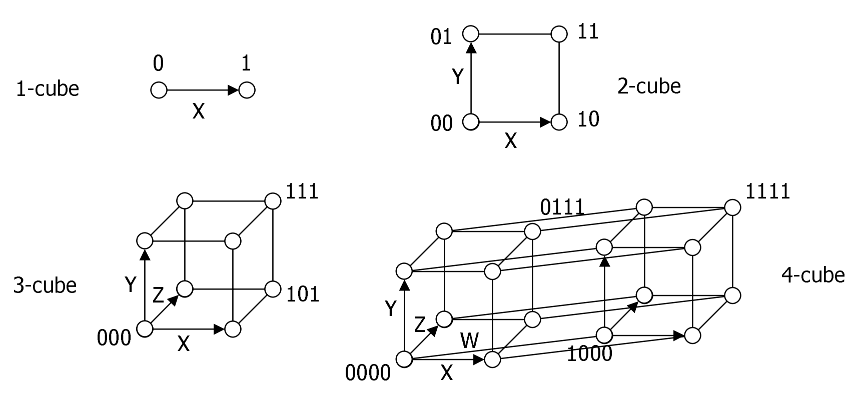
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

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

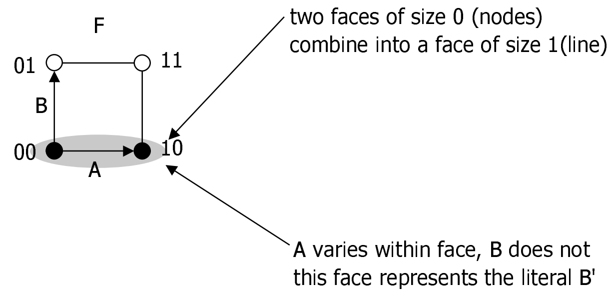
- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"



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

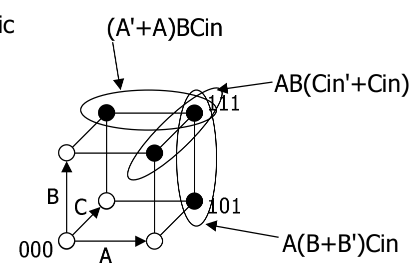


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

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

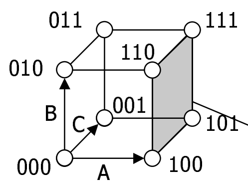


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

Cout = BCin+AB+ACin

Higher dimensional cubes

- Sub-cubes of higher dimension than 2



F(A,B,C) = Σm(4,5,6,7)

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

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

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

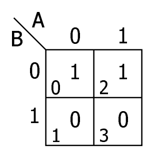
This subcube represents the literal A

m-dimensional cubes in a n-dimensional Boolean space

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

Karnaugh maps

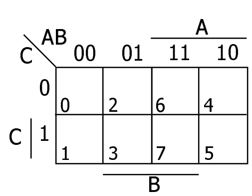
- Flat map of Boolean cube
  - wrap–around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - guide to applying the uniting theorem
  - on-set elements with only one variable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

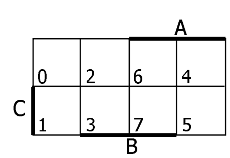


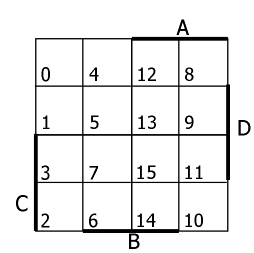
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

Karnaugh maps (cont'd)

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



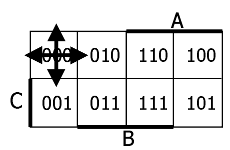


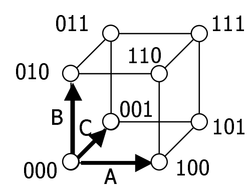


13 = 1101= ABC'D

Adjacencies in Karnaugh maps

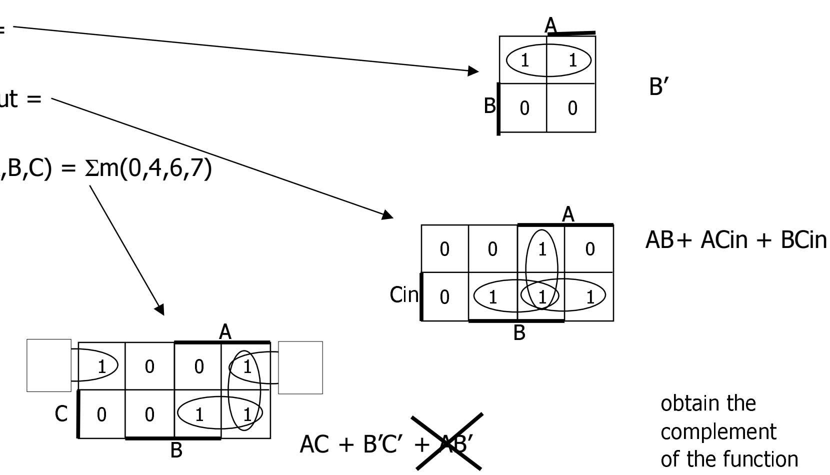
- Wrap from first to last column
- Wrap top row to bottom row





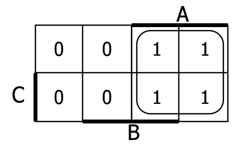
Karnaugh map examples

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

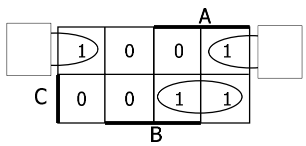


obtain the complement of the function by covering 0s with subcubes

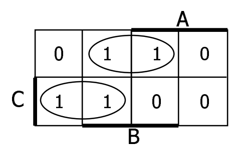
More Karnaugh map examples



G(A,B,C) = A



F(A,B,C) = Σm(0,4,5,7) = AC + B’C’



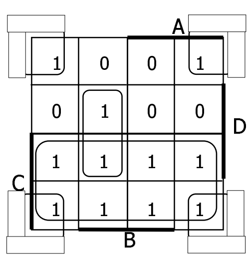
F’ simply replace 1’s with 0’s and vice versa

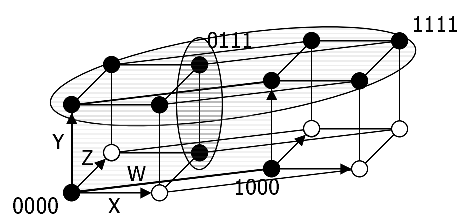
F’(A,B,C) = Σ m(1,2,3,6) = BC’ + A’C

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’

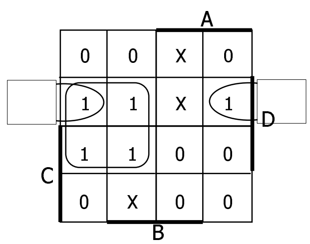




find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)

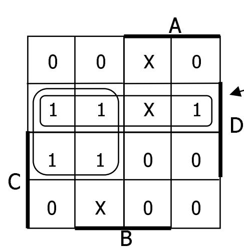
Karnaugh maps: don’t cares

- 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)

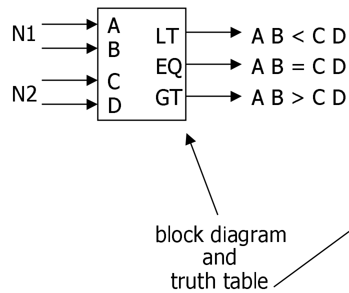
- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - f = A'D + B'C'D without don't cares
  - f = A'D + C'D with don't cares



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

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

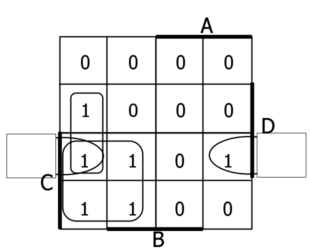
Design example: two-bit comparator



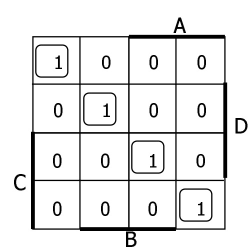
A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

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

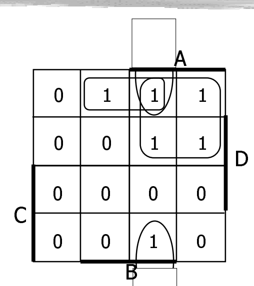
Design example: two-bit comparator (cont'd)



K-map for LT



K-map for EQ



K-map for GT

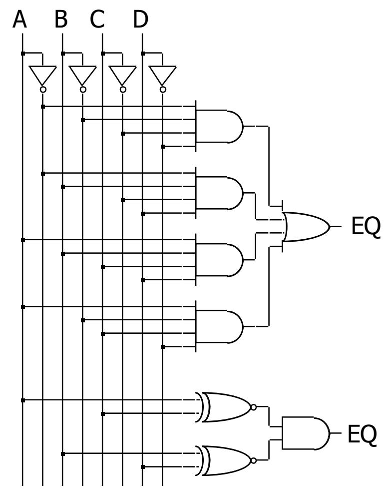
LT = A' B' D + A' C + B' C D

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

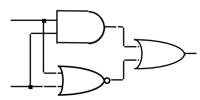
GT = B C' D' + A C' + A B D'

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

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

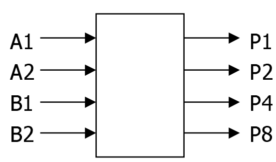


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier

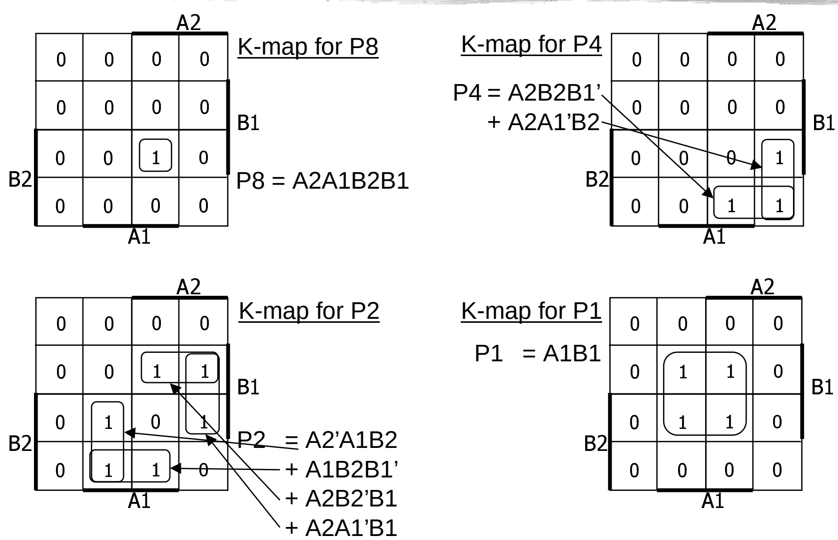


block diagram and truth table

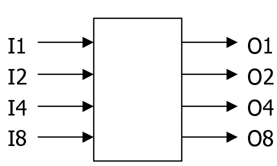
A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

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

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



Design example: BCD increment by 1

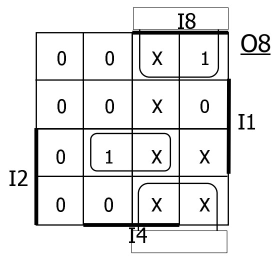


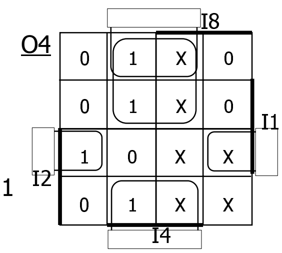
block diagram
and
truth table

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)



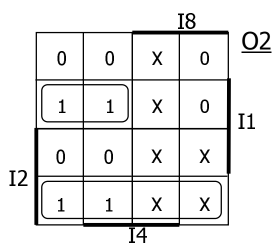


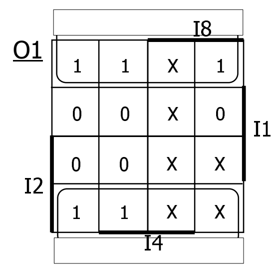
O8 = I4 I2 I1 + I8 I1'

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

O2 = I8' I2' I1 + I2 I1'

O1 = I1'

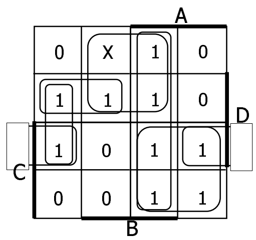




Definition of terms for two-level simplification

- 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

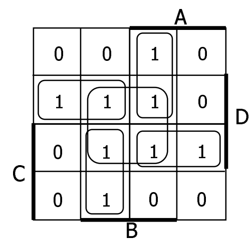
essential

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

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

essential

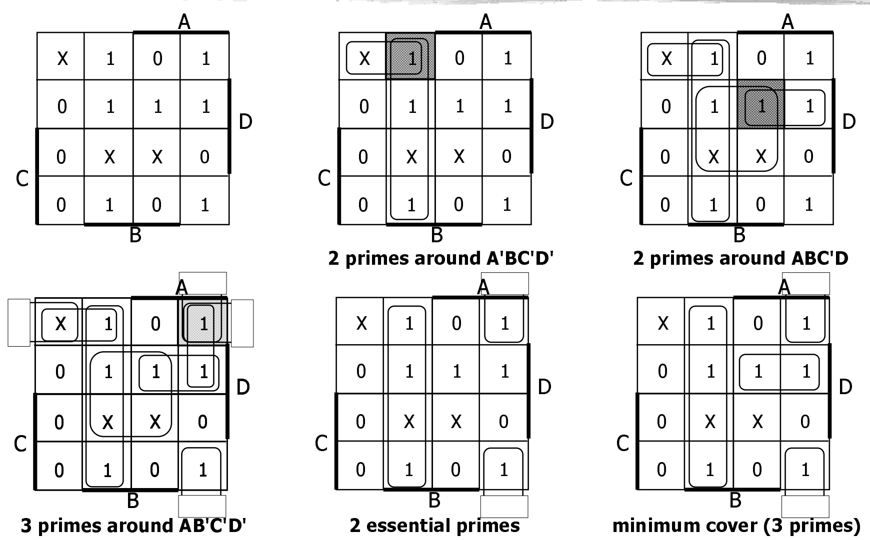
minimum cover: 4 essential implicants



Algorithm for two-level simplification

- Algorithm: minimum sum-of-products expression from a 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)



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