Combinational logic

- Simplification
  - Boolean cubes and Karnaugh maps
  - Two-level simplification
- Two-level logic
  - Implementations of two-level logic
  - NAND/NOR

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'

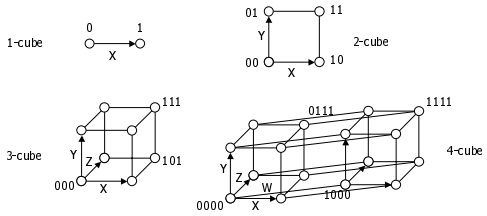
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

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

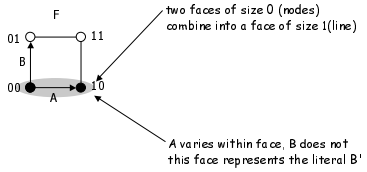
- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"



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

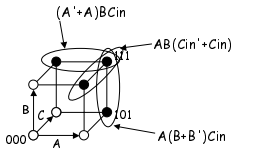


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

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

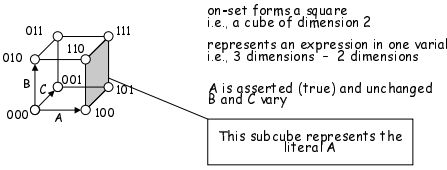


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

Cout = BCin+AB+ACin

Higher dimensional cubes

- Sub-cubes of higher dimension than 2



m-dimensional cubes in a 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

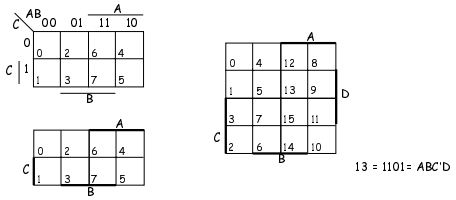
Karnaugh maps

- 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

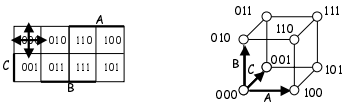
Karnaugh maps (cont'd)

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



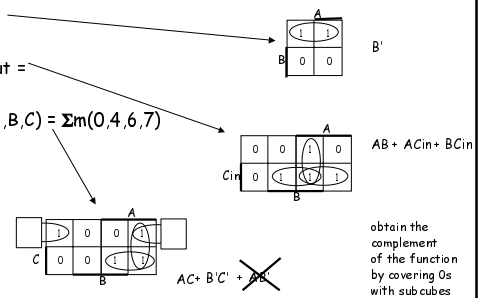
Adjacencies in Karnaugh maps

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

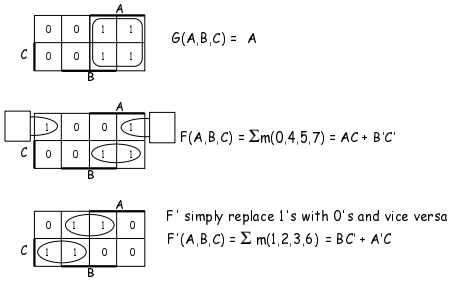


Karnaugh map examples

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

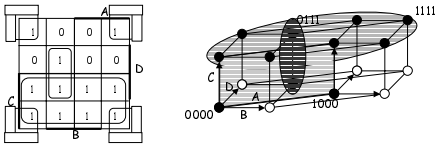


More Karnaugh map examples



Karnaugh map: 4-variable example

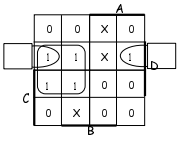
- F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)
  F = C + A'BD + B'D'



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

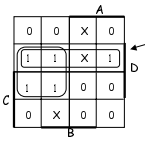
Karnaugh maps: don't cares

- 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)

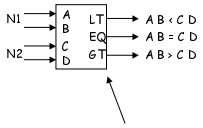
- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - f = A'D + B'C'D — without don't cares
  - f = A'D + C'D — with don't cares



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

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

Design example: two-bit comparator

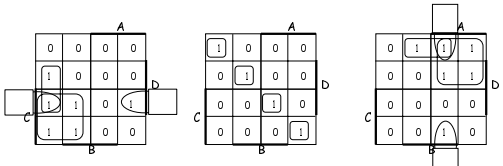


block diagram and truth table

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)

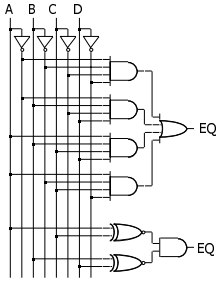


K-map for LT K-map for EQ K-map for ~~for~~ GT

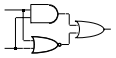
LT = A' B' D + A' C + B' C D
EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD' = (A xnor C) • (B xnor D)
GT = B C' D' + A C' + A B D'

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

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

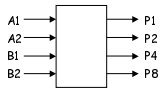


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier

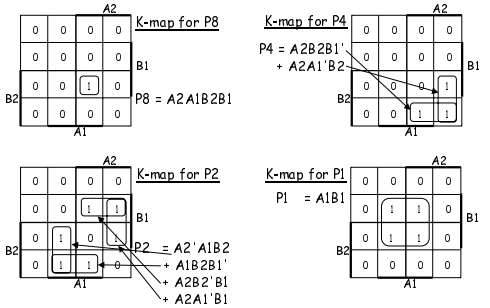


block diagram and truth table

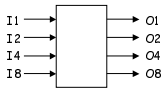
A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

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

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



Design example: BCD increment by 1

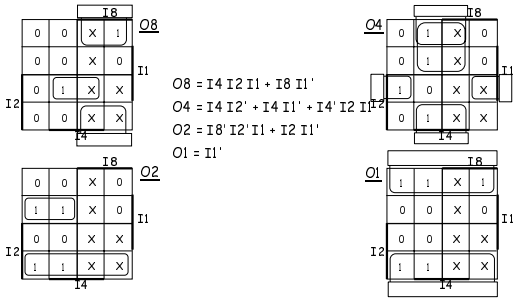


block diagram and truth table

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)

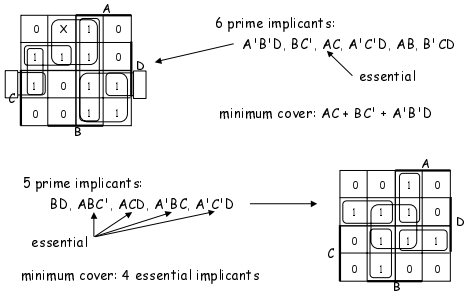


O8 = I4 I2 I1 + I8 I1'
O4 = I4 I2' + I4 I1' + I4' I2 I1
O2 = I8' I2' I1 + I2 I1'
O1 = I1'

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)

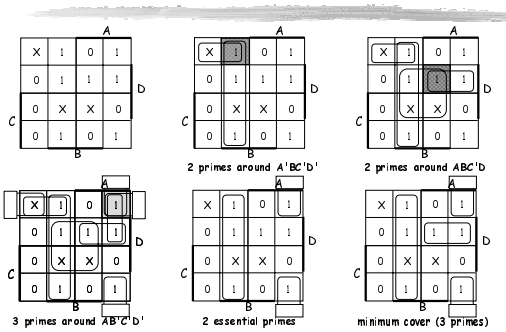
Examples to illustrate terms



Algorithm for two-level simplification

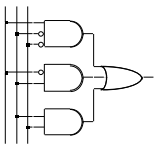
- Algorithm: minimum sum-of-products expression from a Karnaugh map
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number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

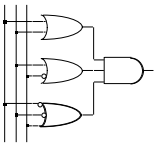


Implementations of Two-level Logic

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

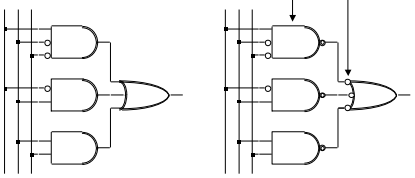


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



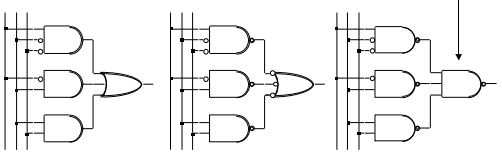
Two-level Logic using NAND Gates

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



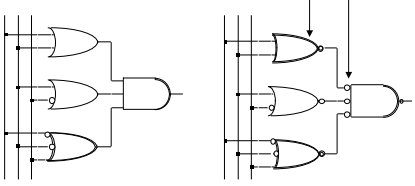
Two-level Logic using NAND Gates (cont'd)

- 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



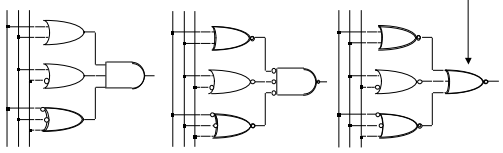
Two-level Logic using NOR Gates

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



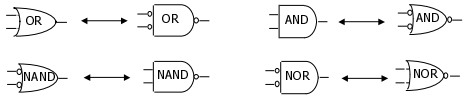
Two-level Logic using NOR Gates (cont'd)

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



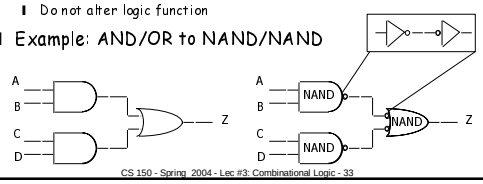
Two-level Logic using NAND and NOR Gates

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



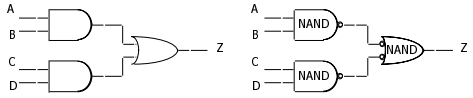
Conversion Between Forms

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



Conversion Between Forms (cont'd)

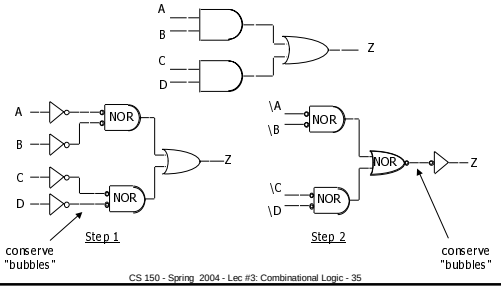
- Example: verify equivalence of two forms



$Z = [\,(A \cdot B)' \cdot (C \cdot D)'\,]'$
$= [\,(A' + B') \cdot (C' + D')\,]'$
$= [\,(A' + B')' + (C' + D')'\,]$
$= (A \cdot B) + (C \cdot D)$ ✓

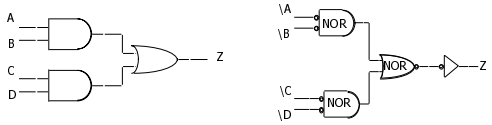
Conversion Between Forms (cont'd)

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



Conversion Between Forms (cont'd)

- Example: verify equivalence of two forms



$Z = \{\,[\,(A' + B')' + (C' + D')'\,]'\,\}'$
$= \{\,(A' + B') \cdot (C' + D')\,\}'$
$= (A' + B')' + (C' + D')'$
$= (A \cdot B) + (C \cdot D)$ ✓

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