Combinational Logic

Lecture 2
March 23rd, 2020

Jae Wook Lee (jaewlee@snu.ac.kr)
Department of Computer Science and Engineering
Seoul National University

Overview: Combinational logic

Textbook: [K&B2e] 2.1, 2.2, 2.3, 2.4, 2.5

- **Basic logic**
  - Boolean algebra
  - Proofs by re-writing, proofs by perfect induction
- **Realizing Boolean formulas**
  - Logic functions, minimal set
  - Logic gates: NOT, AND, OR, NAND, NOR, XOR, ...
  - Which realization is best?
- **Two-level logic**
  - Canonical forms
  - Incompletely-specified functions
- **Motivation for two-level simplification**
  - Uniting theorem
  - Cubes
  - Karnaugh maps (K-maps)

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:

Identity (element) 항등원

1. the set B contains at least two elements: 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}
  - variables
  - + 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

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

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. $X \bullet 1 = X$
- null
  2. $X + 1 = 1$ 2D. $X \bullet 0 = 0$
- idempotency:
  3. $X + X = X$ 3D. $X \bullet X = X$
- involution:
  4. $(X')' = X$
- complementarity:
  5. $X + X' = 1$ 5D. $X \bullet X' = 0$
- commutativity:
  6. $X + Y = Y + X$ 6D. $X \bullet Y = Y \bullet X$
- associativity:
  7. $(X + Y) + Z = X + (Y + Z)$ 7D. $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$

Idempotency: one may derive the same consequences from many instances of a hypothesis as from just one involution: a function that is its own inverse, so that $f(f(x)) = x$

Note that suffix "D" means the dual of the original expression.
Dual is the other symmetric part of a pair, which will be discussed later.
(at this moment, use two rules: AND<-> OR, 0<->1)

Axioms and theorems of Boolean algebra

- 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 — 12D. X • Y + X’ • Z = (X + Z) • (X’ + Y)
- concensus:
  - 13. (X • Y) + (Y • Z) + (X’ • Z) = X • Y + X’ • Z — 13D. (X + Y) • (Y + Z) • (X’ + Z) = (X + Y) • (X’ + Z)

Theorem 12. (X+Y)(X’+Z) = XX’+XZ+X’Y+YZ =XZ+X’Y+YZ(X+X’) =XZ(1+Y)+X’Y(1+Z) =XZ+X’Y

Axioms and theorems of Boolean algebra

- de Morgan's:
  14. $(X + Y + \ldots)' = X' \bullet Y' \bullet \ldots$ 14D. $(X \bullet Y \bullet \ldots)' = X' + Y' + \ldots$
- generalized de Morgan's:
  15. $f'(X_1,X_2,\ldots,X_n,0,1,+,\bullet) = f(X_1',X_2',\ldots,X_n',1,0,\bullet,+)$
- establishes relationship between • and +

Axioms and theorems of Boolean algebra

- **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. $f(X_1,X_2,\ldots,X_n,0,1,+,\bullet) \Leftrightarrow f(X_1,X_2,\ldots,X_n,1,0,\bullet,+)$
- **Different from de Morgan'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: $X \bullet Y + X \bullet Y' = X$ (uniting)

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: $X + X \bullet Y = X$ (absorption)

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

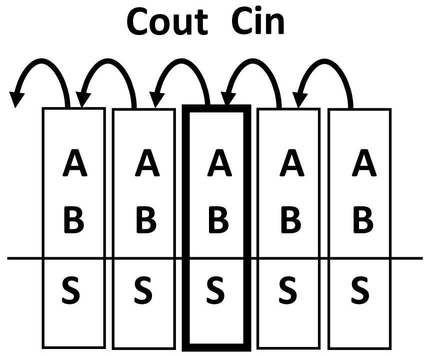
X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

(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	Cout	S
0	0	0	0	0
0	0	1	0	1
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	1

A
B
Cin
S
Cout

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

Note: This adder is called "full" adder. If there is no Cin, it is called "half" adder (you will see at this week's lab)!

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' + $\boxed{\text{A B Cin + A B Cin}}$
= 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' + $\boxed{\text{A B Cin + A B Cin}}$
= 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

adding extra terms creates new factoring opportunities

Overview: Combinational logic

Textbook: [K&B2e] 2.1, 2.2, 2.3, 2.4, 2.5

- **Basic logic**
  - Boolean algebra
  - Proofs by re-writing, proofs by perfect induction
- **Realizing Boolean formulas**
  - Logic functions, minimal set
  - Logic gates: NOT, AND, OR, NAND, NOR, XOR, ...
  - Which realization is best?
- **Two-level logic**
  - Canonical forms
  - Incompletely-specified functions
- **Motivation for two-level simplification**
  - Uniting theorem
  - Cubes
  - Karnaugh maps (K-maps)

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	F0	F1		F3			F6			F9			F12			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

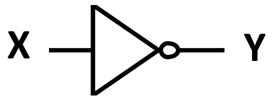
Cost of different logic functions

- **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

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

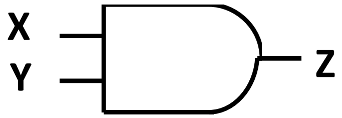
From Boolean expressions to logic gates

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



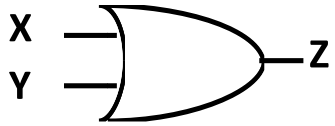
X	Y
0	1
1	0

- AND   X • Y   XY $X \wedge Y$



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

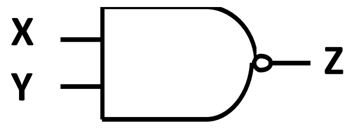
- OR   X + Y $X \vee Y$



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

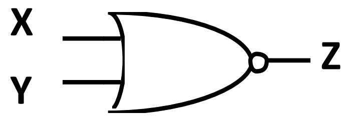
From Boolean expressions to logic gates

- **NAND**



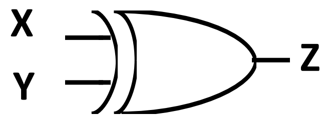
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- **NOR**



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

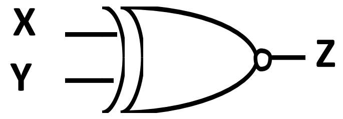
- **XOR**
  $X \oplus Y$



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



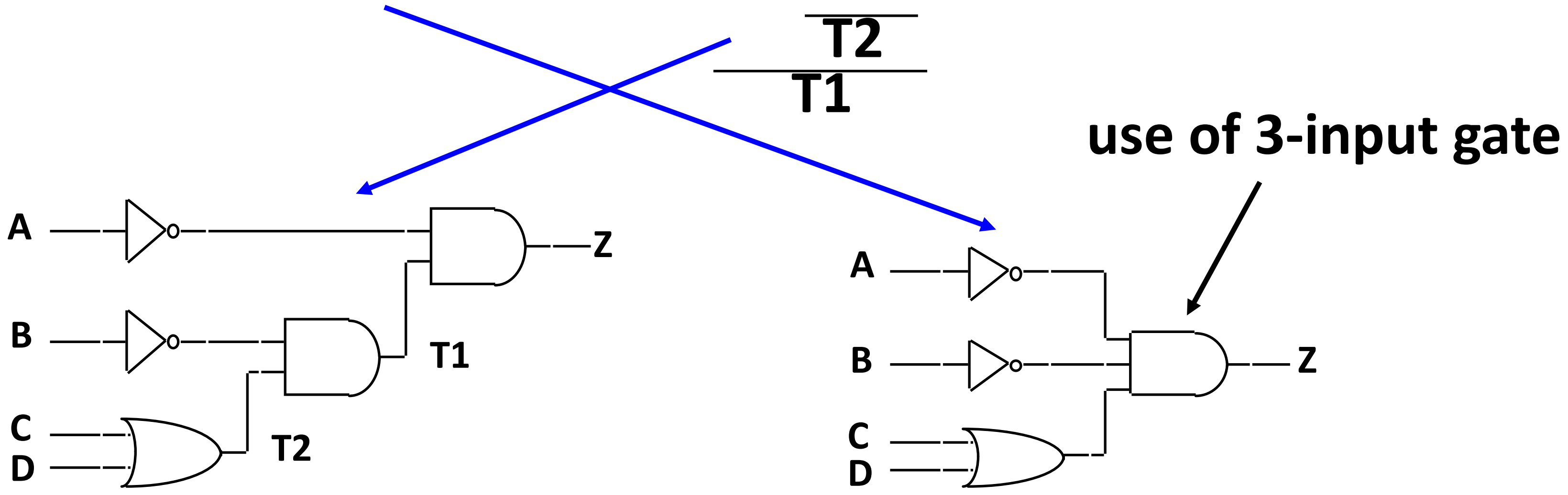
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

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

Note: XNOR is the negation of XOR

From Boolean expressions to logic gates

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



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

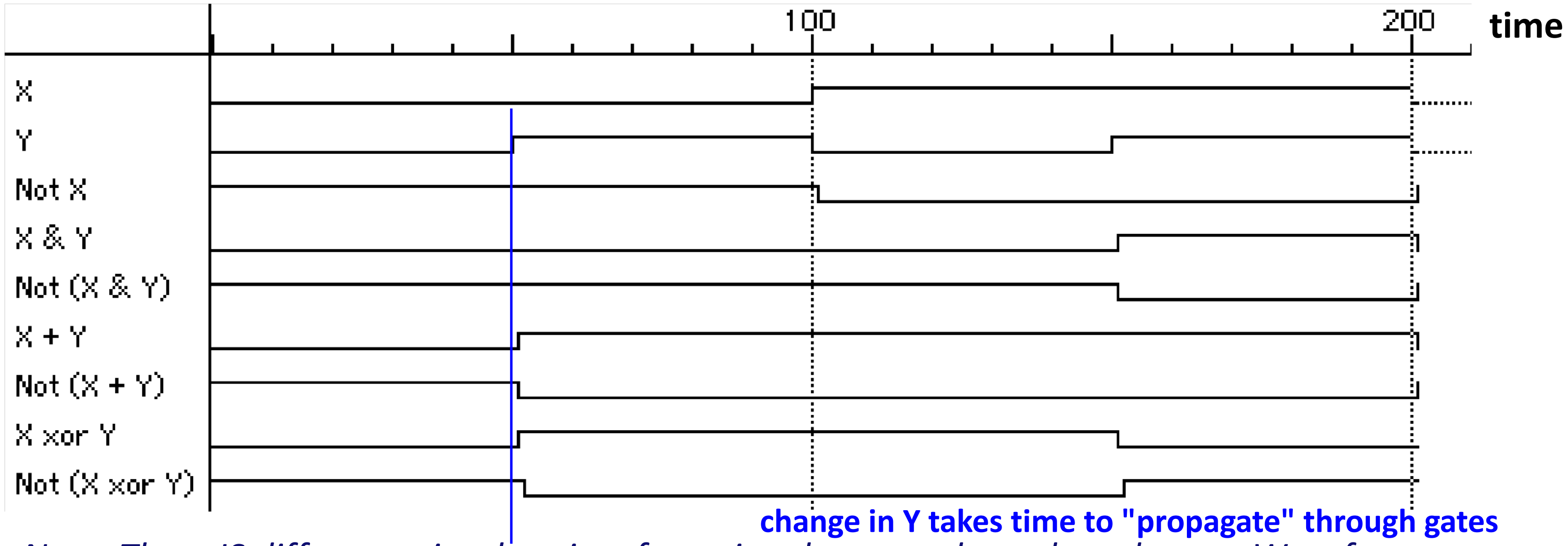
X	Y	X nand Y
0	0	1
1	1	0

  - NAND and NOR are "duals",
    that is, its easy to implement one using the other

X <u>nand</u> Y ≡ <u>not</u> ((<u>not</u> X) <u>nor</u> (<u>not</u> Y))
X <u>nor</u> Y ≡ <u>not</u> ((<u>not</u> X) <u>nand</u> (<u>not</u> Y))

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!



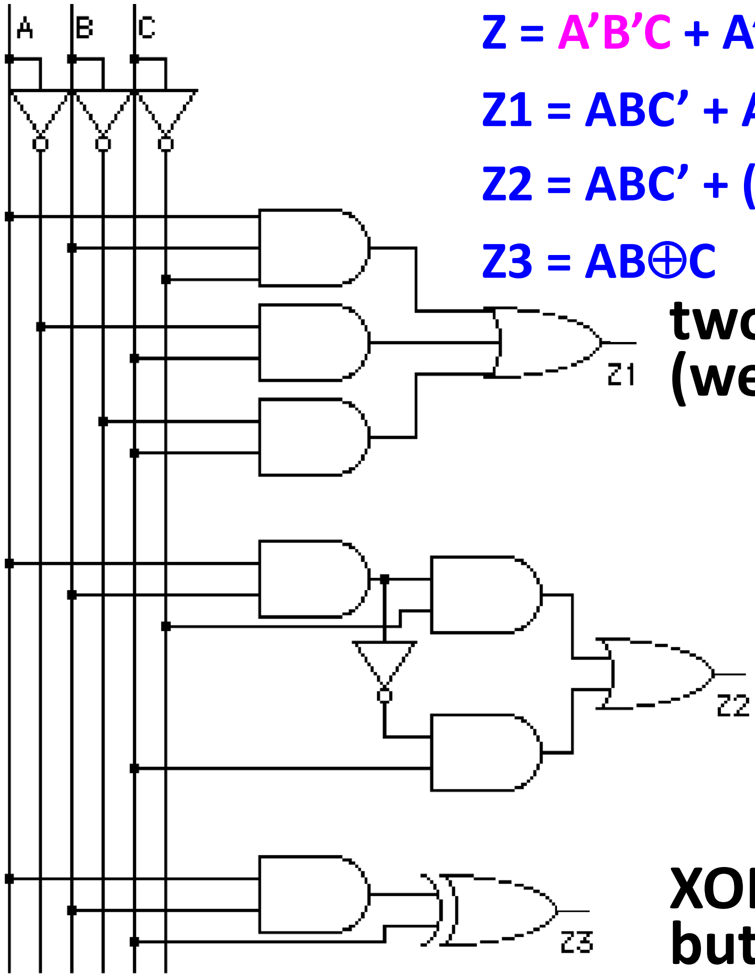
Note: There IS difference; it takes time for a signal to pass through each gate. Waveform describes how a signal at each point changes over time.

Suppose X and Y change at precise timing. Depending on the gate type, the gate passing delay can be slightly different. e.g. an XOR gate is complicated, which incurs a longer delay than other simple gates.

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

p.53



Z = A’B’C + A’BC + AB’C + ABC’

Z1 = ABC’ + A’C + B’C

Z2 = ABC’ + (AB)’C

Z3 = AB⊕C

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

multi-level realization
(gates with fewer inputs)

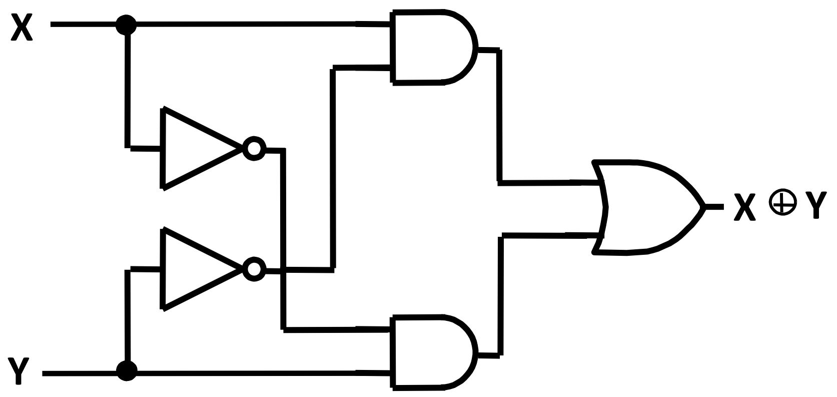
XOR gate (easier to draw
but costlier to build)

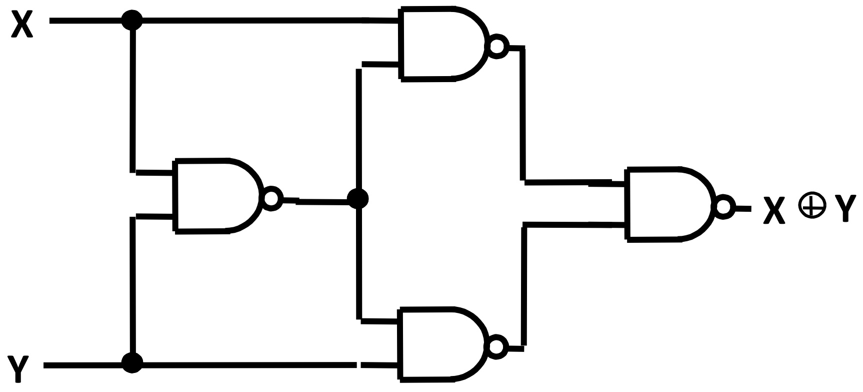
Z1=Z2=Z3=Z

Let’s consider Z1 first. 3 AND gates and 1 OR gate. Also we need to check the # of wires or inputs. In Z3, XOR is called a complex gate, which requires several NAND or NOR gates. So Z3 is likely to have the worst delay.

XOR implementations

- Three levels of logic inside a XOR gate
- X⊕Y = X'Y + XY'





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
  - 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 realization is best?

- **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

Depending on the criteria (e.g. minimize delay, minimize the # of gates), the CAD tools may yield different solutions.

Overview: Combinational logic

Textbook: [K&B2e] 2.1, 2.2, 2.3, 2.4, 2.5

- **Basic logic**
  - Boolean algebra
  - Proofs by re-writing, proofs by perfect induction
- **Realizing Boolean formulas**
  - Logic functions, minimal set
  - Logic gates: NOT, AND, OR, NAND, NOR, XOR, ...
  - Which realization is best?
- **Two-level logic**
  - Canonical forms
  - Incompletely-specified functions
- **Motivation for two-level simplification**
  - Uniting theorem
  - Cubes
  - Karnaugh maps (K-maps)

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

A Boolean function can take one of various expressions.

Canonical forms

- **Truth table is the unique signature of a Boolean function**
- **The same truth table can have many gate realizations**
- **Canonical forms**
  - standard forms for a Boolean expression
  - provides a unique algebraic signature

Sum-of-products (S-o-P) canonical forms

- Also known as (aka) disjunctive normal form
- Also known as minterm expansion

F = *001* *011* *101* *110* *111*

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'

Note: Just check all the cases when F becomes true and each case forms the product of input variables. And finally, ORing these products will yield the final expression.
This is also called minterm expansion; here, a minterm is a product of all the input literals.
Each literal should appear once in each minterm: asserted or complemented

Sum-of-products canonical forms

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

	A	B	C	minterms	
0	0	0	0	A'B'C'	m0
1	0	0	1	A'B'C	m1
2	0	1	0	A'BC'	m2
3	0	1	1	A'BC	m3
4	1	0	0	AB'C'	m4
5	1	0	1	AB'C	m5
6	1	1	0	ABC'	m6
7	1	1	1	ABC	m7

short-hand notation for minterms of 3 variables

F in canonical form:

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

canonical form ≠ minimal form

$$F(A, B, C) = A'B'C + A'BC + AB'C + ABC + ABC'$$
$$= (A'B' + A'B + AB' + AB)C + ABC'$$
$$= ((A' + A)(B' + B))C + ABC'$$
$$= C + ABC'$$
$$= ABC' + C$$
$$= AB + C$$

Note: Each product is called a minterm, and denoted by small **m** and **a decimal number** for the binary input values. Note that there is no reduction or minimization in canonical forms; each variable must appear once for each product

Product-of-sums (P-o-S) 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')

Note: The other canonical form is P-o-S. This one focuses on when F will be 0. P-o-S is like the dual of S-o-P. First of all, we check all the cases that make F false or 0 The variables for each case or term are first complemented and then connected by the OR operation. This ORed term is called a maxterm. Eventually, these terms are connected by AND. What does the final expression mean?

Product-of-sums canonical form

- **Sum term (or maxterm)**
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, true or inverted (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:

$$F(A, B, C) = \Pi M(0,2,4)$$
$$= M0 \bullet M2 \bullet M4$$
$$= (A + B + C)\,(A + B' + C)\,(A' + B + C)$$

canonical form ≠ minimal form

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

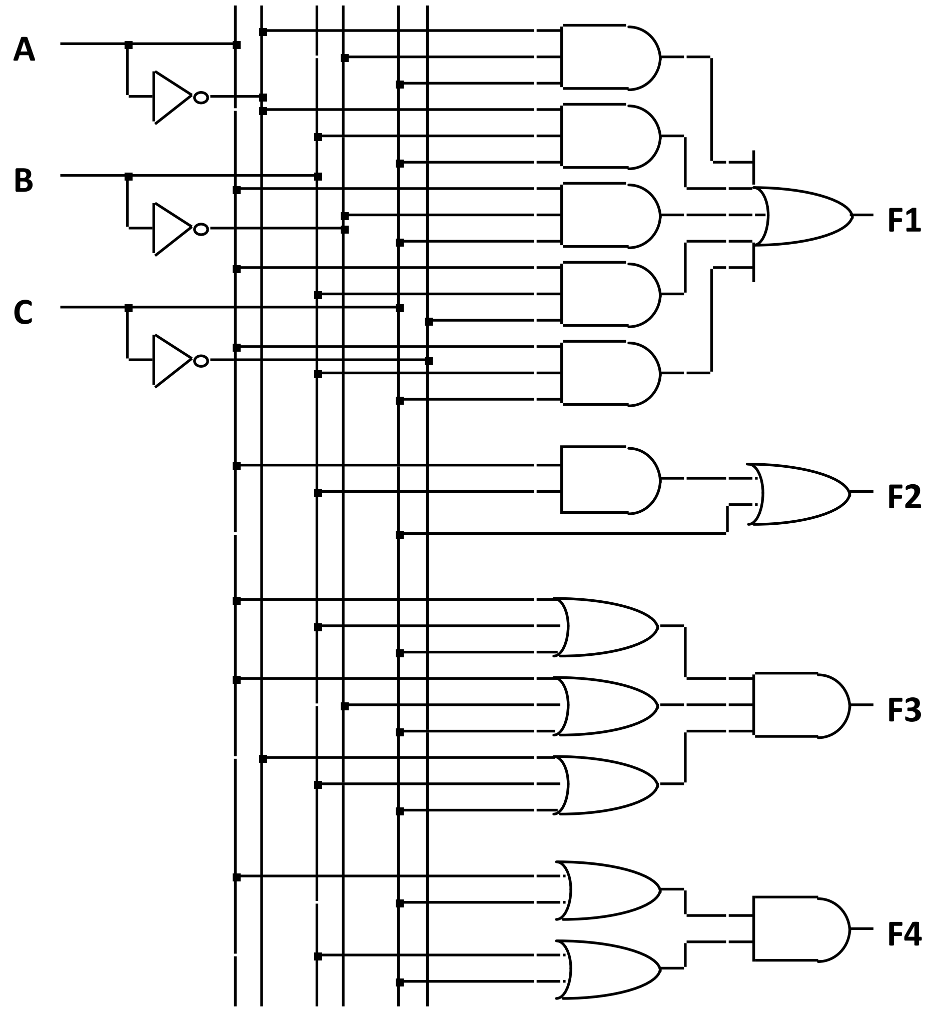
Note: Each maxterm is denoted by the capital M and the decimal value of input variables.

S-o-P, P-o-S, and de Morgan’s theorem

- **Sum-of-products**
  - F’ = A’B’C’ + A’BC’ + AB’C’
- **Product-of-sums for F??**
- **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’)
- **Sum-of-products for F?**
- **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

4 alternative two-levelimplementations

- F = AB + C



canonical sum-of-products

minimized sum-of-products

canonical product-of-sums

minimized product-of-sums

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 (BCD) 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

On-set: the set of cases whose output is 1

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

Note: BCD coding uses only ten values from 0 to 9. With 4 input lines, we have 6 don't care cases of input values. For these don't care values, the function can have any arbitrary output values.

Notation for incompletely specified functions

- **Don't cares and canonical forms**
  - so far, we focus on either on-set or off-set
  - There can be 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:**
  - Minterm expansion?
    - 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)]
  - Maxterm expansion?
    - 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)]

Overview: Combinational logic

Textbook: [K&B2e] 2.1, 2.2, 2.3, 2.4, 2.5

- **Basic logic**
  - Boolean algebra
  - Proofs by re-writing, proofs by perfect induction
- **Realizing Boolean formulas**
  - Logic functions, minimal set
  - Logic gates: NOT, AND, OR, NAND, NOR, XOR, ...
  - Which realization is best?
- **Two-level logic**
  - Canonical forms
  - Incompletely-specified functions
- **Motivation for two-level simplification**
  - Uniting theorem
  - Cubes
  - Karnaugh maps (K-maps)

Simplification of two-level comb. logic

- **Finding a minimal S-o-P or P-o-S 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 (CAD) 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)

Two minimization techniques

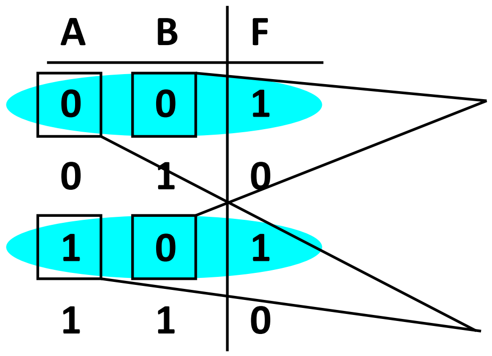
- Boolean cubes
- Karnaugh-maps (K-maps)

- Both of them are based on the uniting theorem

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'



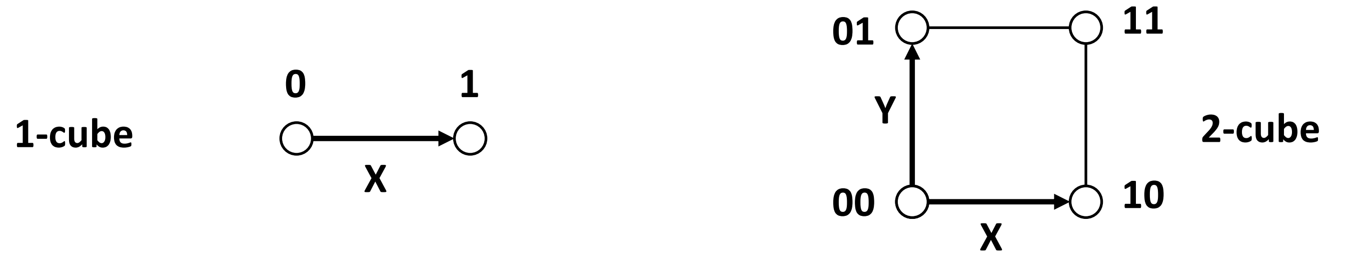
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

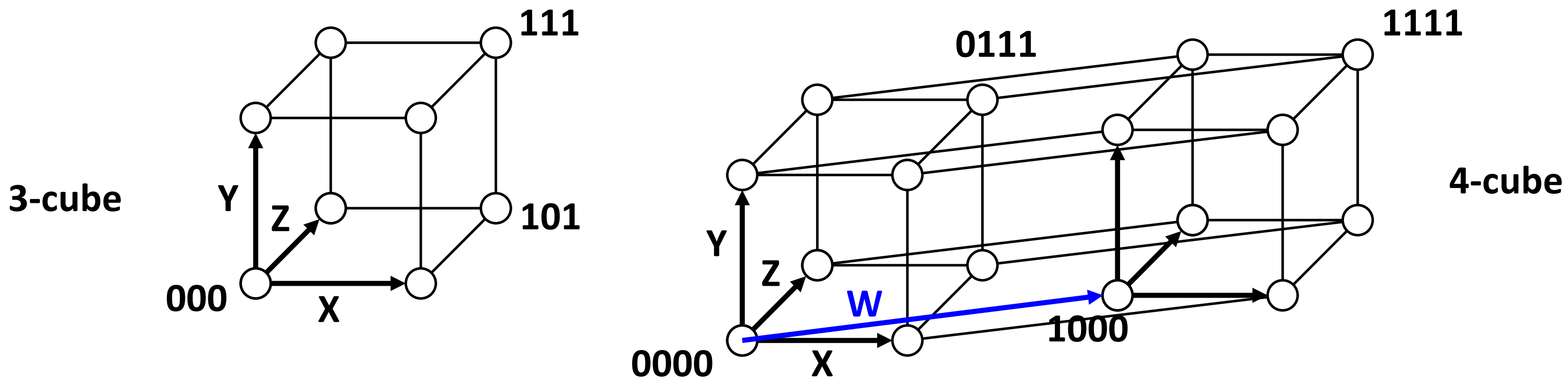
B has the same value in both on-set rows – B remains (in complemented form)

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"

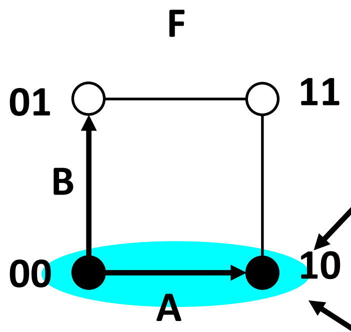




Mapping truth tables onto Boolean cubes

- 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



two faces of size 0 (nodes) combine into a face of size 1(line)

A varies within face, B does not this face represents the literal B'

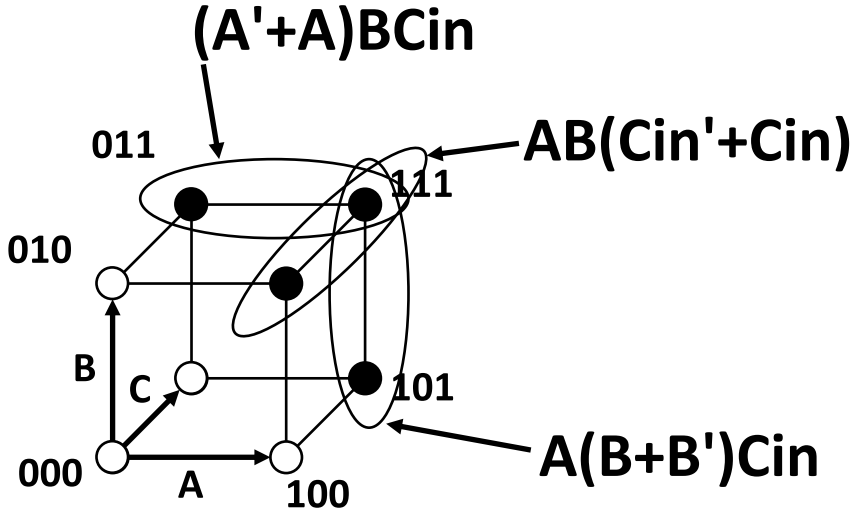
ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

fill in the nodes that correspond to the elements of the ON-set.
If there are two adjacent solid nodes, we can use the uniting theorem.

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

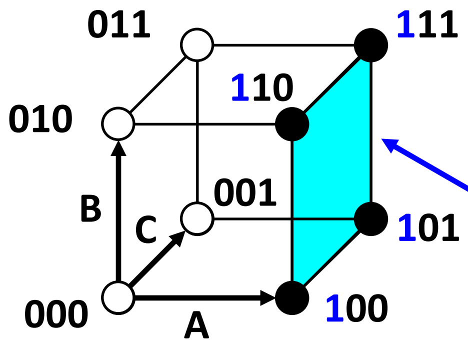


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that “111” is covered three times

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

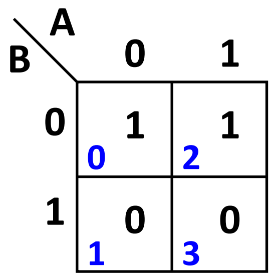
Note: Output function is Σm(4,5,6,7) in S-O-P form.
In this case, the on-set nodes form a square.
Here, we use the uniting theorem at a greater scale. A(BC+BC'+B'C+B'C') = A

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,**
  - In an n-cube, an m-subcube ($m < n$) 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

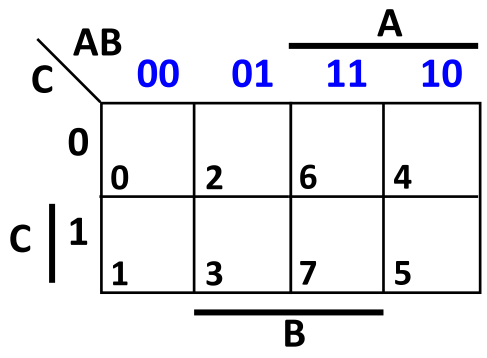


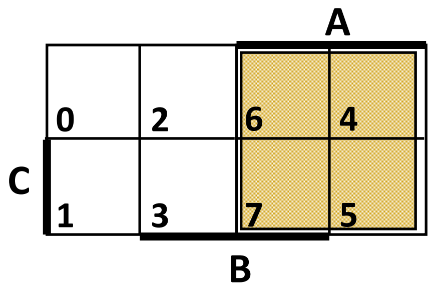
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

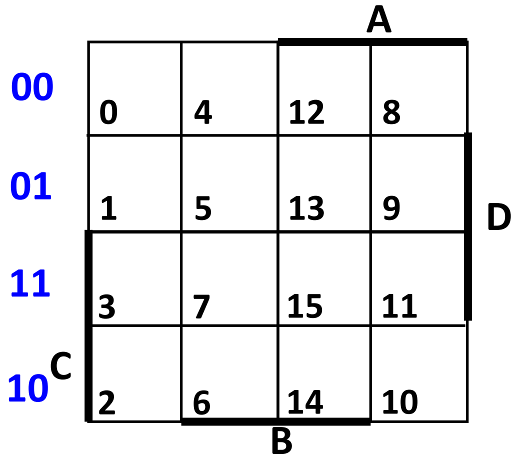
Another technique is using a Karnaugh map, which is kind of a flat version of the Boolean cube technique.

Karnaugh maps

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





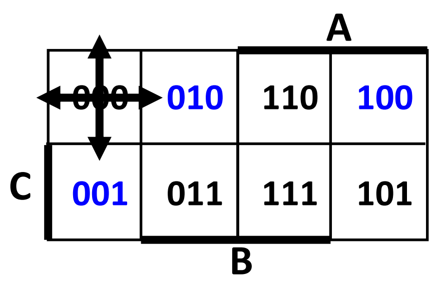


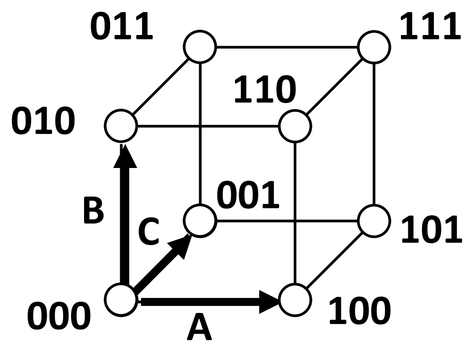
13 = 1101= ABC’D

Note: This slide shows Karnaugh maps of 3 and 4 inputs . The thick line segment represents the domain (in the perpendicular direction) where each variable is always TRUE. The complement of the above domain indicate the inverted variable.

Adjacencies in Karnaugh maps

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

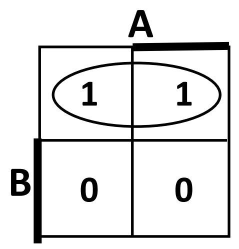




Let’s focus on cell 000; there are three adjacent cells.
Note that the number of adjacent cells is the same as the number of input variables since it is equal to the number of bits.

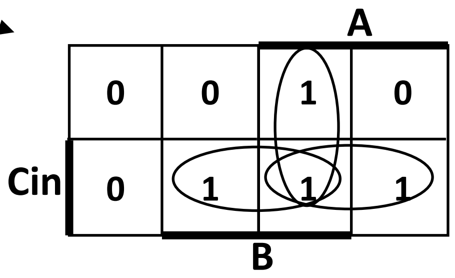
Karnaugh map examples

- F =



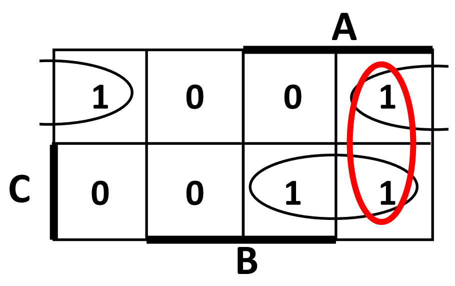
B'

- Cout =



$AB+ACin+BCin$

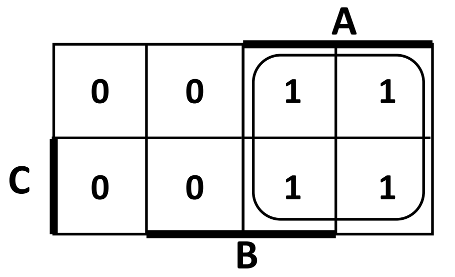
- f(A,B,C) = Σm(0,4,5,7)



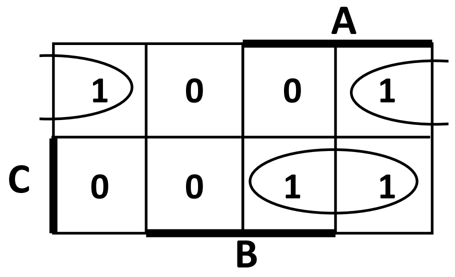
$AC+ B'C'+ \cancel{AB'}$

The on-set included in the red oval is already covered by two other adjacencies

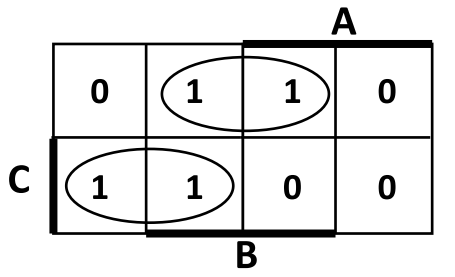
More Karnaugh map examples



G(A,B,C) = A



F(A,B,C) = Σm(0,4,5,7) = AC + B’C’

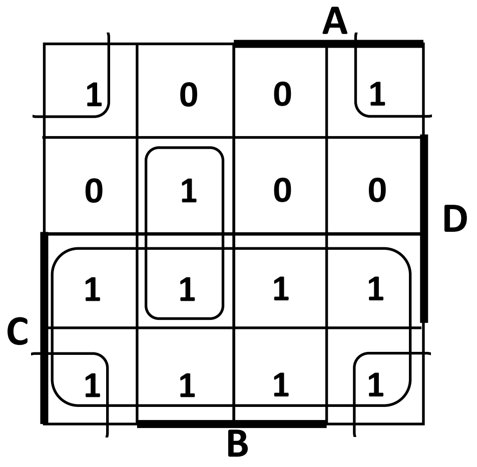


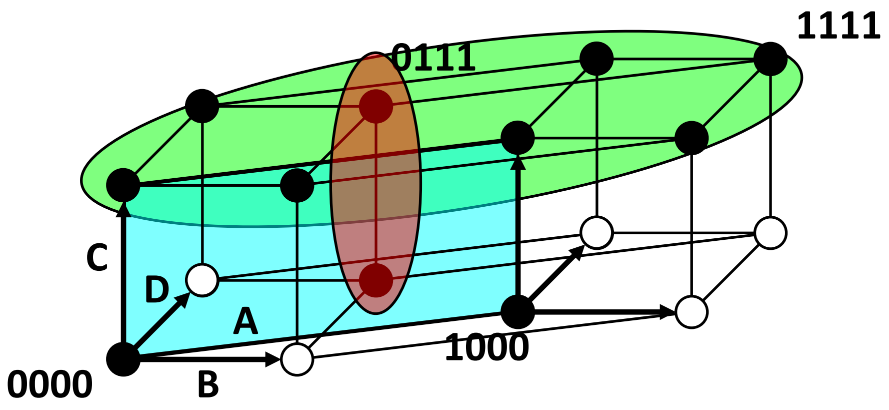
F' simply replace 1's with 0's and vice versa
F'(A,B,C) = Σ m(1,2,3,6) = BC’ + A’C

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'

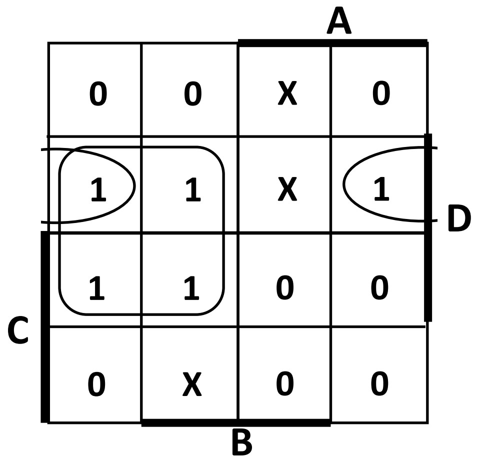




find the smallest number of the largest possible subcubes to cover the ON-set (fewer terms with fewer inputs per term)

Karnaugh maps: don’t cares (DCs)

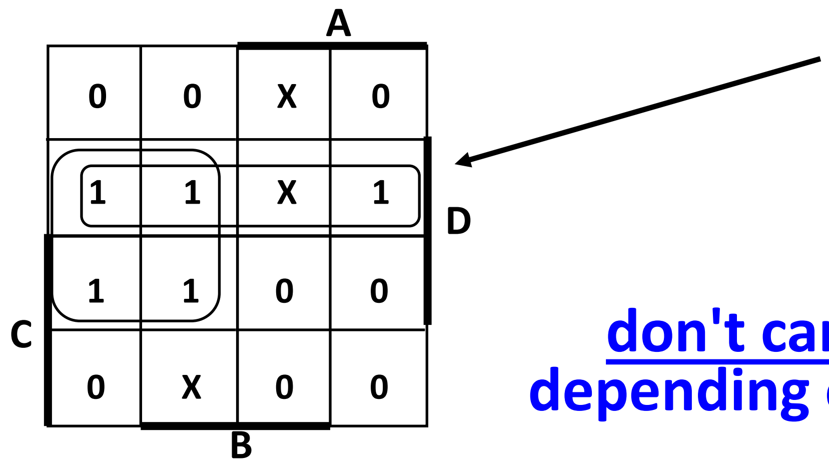
- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - without don't cares
    - f = A’D + B’C’D



Now let’s see how we can utilize don’t care (DC) terms in the Karnaugh map technique. If we don’t use DC terms, the logic function f is A’D +B’C’D

Karnaugh maps: don’t cares

- 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

By interpreting DCs as 1s opportunistically, we can utilize the uniting theorem at greater scale.

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**
  - a start at understanding two-level simplification
- **Later**
  - automation of simplification
  - multi-level logic
  - time behavior
  - hardware description languages
  - design case studies

So... What?

- **Why you bother to learn this stuff:**
  - Boolean algebra is the mathematical foundation of digital logic design
  - Also very important for design testing, verification, and debugging of both hardware and software.
  - CAD software makes billion dollar business.